CS744 - Big Data Systems
Assignment 1

Group 23:

Akul Gupta
Chahak Tharani
Deepti Rajagopal

Index

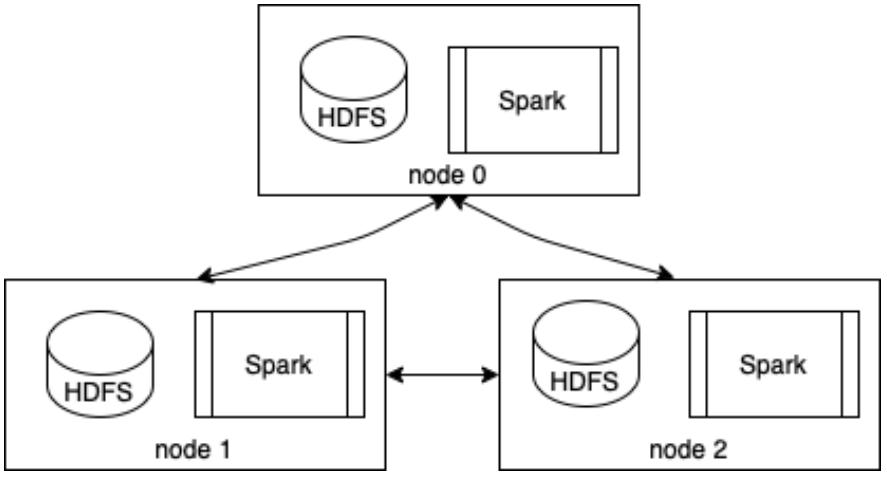
Setup

We provisioned the cloudlab nodes as per the documentation shared for this assignment. Briefly, the following steps were taken:

1. Mounted additional space, and updated the permissions of the directories so that all members could write to them.
2. Installed all the required applications - JDK, Hadoop, and Spark.
3. Updated HDFS & Spark config files to designate namenode, datanodes, master & worker nodes.

Some of the issues in our setup phase which we navigated around were:

1. SSH keys would periodically get removed from ~/.ssh/authorized_keys, so each of us had to update them regularly.
2. Hadoop would sometimes not run on all 3 nodes. When this happens, we delete the namenode and datanode directories from all 3 nodes, stop all nodes, reformat, and then start all of them again.
3. We also saw an out of disk space error in Spark, which happened because spark defaulted to “/tmp” as the local directory. We had to set the path where spark could put temporary files, as a variable in spark-env.sh on all three nodes
4. Spark would sometimes not kill all the workers on all nodes, when stop-all.sh was run. This was overcome by explicitly killing the process(es).



Sorting Application in Spark

We performed the sorting using DataFrames, and it took 28 seconds for Spark to finish the execution. Although we did not implement this using RDD, the implementation using RDD might probably take longer, because sorting is a column based process, and DataFrames are better suited for handling data that can be properly tabularized.

PageRank Application in Spark

"PageRank (**PR**) is an algorithm used by Google Search to rank web pages in their search engine results. PageRank is a way of measuring the importance of website pages. According to Google:

PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites."

Below, we explore different ways in which we implemented the PageRank algorithm.

Naive Approach:

Algorithm details:

1. We initially started with an implementation consisting of transformations on RDDs. (Code in file: *naive_pagerank_rdd.py*). Then, we shifted some of our transformations to DataFrames to evaluate performance differences. (Code in file: *naive_pagerank_rdd_df.py*) Lastly, we converted our entire code to use Spark DataFrames. (Code in file : *naive_pagerank_df.py*)

2. Transformations done:
   2.1. After reading the data into a DF, we filter the values and split the columns. For RDD implementation, we used the map and filter functions and for DF we did transformations on columns.

   2.2. We group the values by key to get the outgoing edges from a node for computation of pagerank. We also initialize all initial ranks as 1.0.

   2.3. We perform a join operation on our grouped dataset. We initially started with an inner join function, however, we realized that the joined dataset may become null after some iterations if we don't have back edges to the nodes in the first column. Hence, we shifted to the left outer join and preserved the rank of all nodes which were not in the sinks dataset.

   2.4. We flattened the grouped nodes column to get ranks of each node and then we aggregated the ranks by key to get the final rank which we saved in HDFS.

With Partitioning:

We used Spark's HashPartitioner on the key of both links and ranks dataframes in order to ensure that in a given partition, both the links for a specific URL and the ranks for that URL are present. This would ensure that there's minimal shuffling during the join operations.

With Caching:

We cached the links dataframe. We expect a noticeable improvement in completion time after caching the links dataframe since it is reused across the iterations. The improvement should be proportional to the number of iterations.

Killing one of the workers at % of process completion:

1. **At 25% of job completion:**
   We cleared the caches on one of the nodes, and killed the worker node running on it after 25% of the job completed, and restarted the worker in 5 mins. We expect the job completion time to increase as the lost state will need to be recomputed by reapplying the transactions.

2. **At 75% of job completion:**
   We repeated the same process, but instead killed the worker node after 75% of the job completed, and restarted the worker in 10 mins. We expect the job completion time to increase even more than killing a worker at 25% completion as more transformations would need to be applied to recompute the lost state.

Using DataFrame:

We tried to implement PageRank using just the DataFrame APIs as well. Row RDD transformations were converted to corresponding DataFrame column transformations.

Functionality	**RDD Transformation**	**DF Transformation**
Group values by a key	groupBy().agg()	groupByKey()
Aggregate values for a key	groupBy().sum()	reduceByKey()
Map 1 k, v to multiple k,v	explode()	flatMap()
Map k,v with a function	withColumn(col , function(col))	map()
Aggregate values in a list	F.collect_list()	mapValues(list)
Add a column	functions.lit()	map() every row

Observations / Inferences:

We benchmarked the code with two code-styles. One with only DataFrames and a second with a mix of RDDs and DF APIs.

Task 1:

Cache: We cached the links dataset so that we don't have to recreate it from the lineage. This **saves computation costs** and time since the dataset is huge. We observed a **speedup** in completion time by **4 min**. We should observe even better performance improvements when dealing with task failures or partition loss. We did not cache the ranks dataset since the values are different after every iteration. In case of a large number of iterations, we can cache it after every few iterations to save full recomputation in case of task failure.

In the case of killing workers at 25% and 75% time of the job execution, we are clearing the memory cache before killing the worker. Since *.cache()* only stores in memory, even after recovering the worker, we wouldn't get any benefit from caching, which is what we observe.

Task 2:

For partitioning, our idea was to **partition on the key of links and ranks dataset** so that the join operation has **lesser shuffling**. We used the default partitioner(hash partitioner on *key*) with a custom number of partitions in pyspark. We observed the shuffle rate goes up in the case we increased the number of partitions from the default one(repartition). This is inline with our understanding as we can attribute this increase to the **increase in shuffling in case of repartitioning**. For coalesce(reducing the number from default partitions), **shuffle read/write data decreased significantly** since coalesce merges the partitions in memory and doesn't cause shuffling.

Task 3:

On killing our worker at 25% job completion time, we observed an **increase in the shuffle read/write rate** of other 2 workers. Disk I/O also increased since during shuffles, disk read/writes happen to safeguard against memory failure since shuffle is a costly operation. Job completion time after reviving the worker at 25% was **31 mins**.

After killing at 75%, we saw an **increase in shuffle read/write and disk IO** but the **increase was much larger** as compared to when the worker went down after 25%. We understand that till 75% many more iterations would have been completed. So, **recreating the data from lineage** from the beginning **would be very costly**.

Other Observations:

1) We observed that while using DataFrames in our code, many **stages of reduceByKey got skipped.** We found out that this happens because during the shuffle stages the **shuffle map reduce files are preserved** until the corresponding DFs are no longer used and are garbage collected. This is done so the shuffle files don't need to be re-created if the lineage is re-computed.

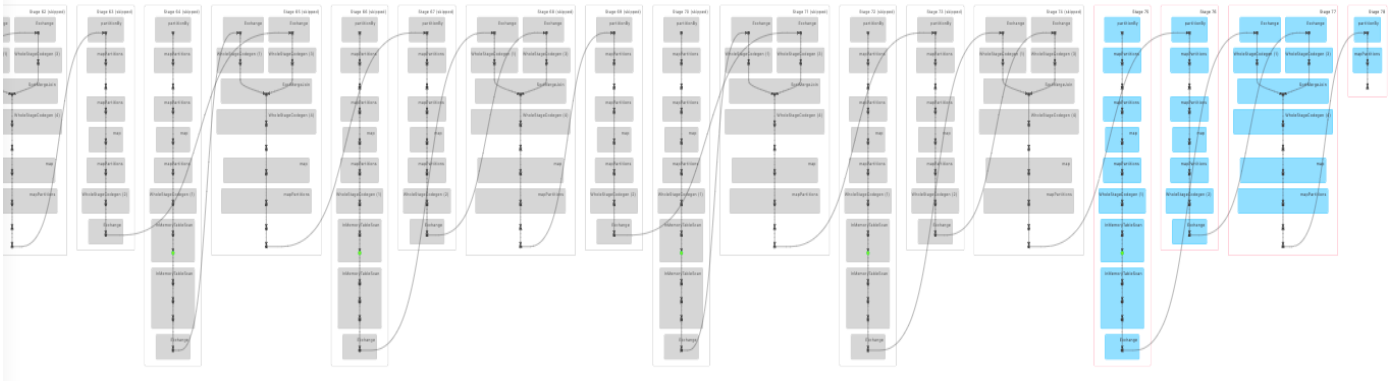


Fig: DAG showing skipped stages

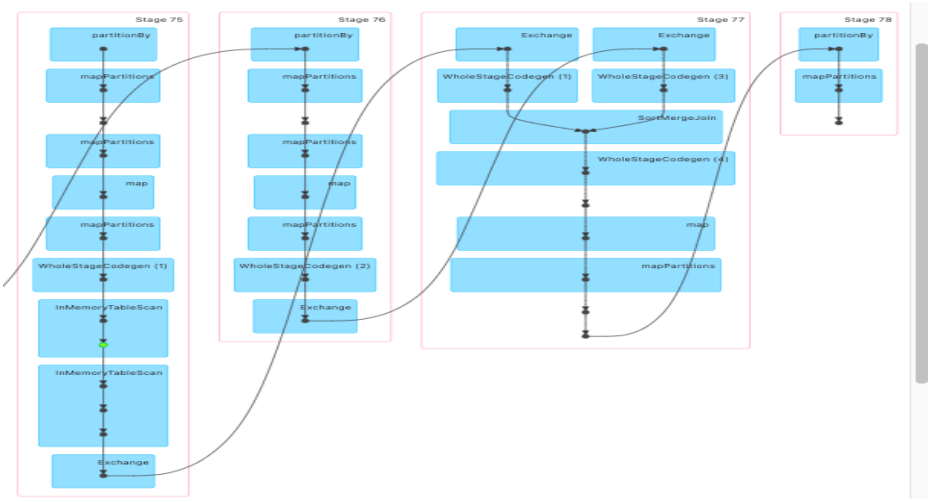


Fig: DAG showing cached DataFrame

2) When we ran the scala implementation, we noticed that the number of stages are in accordance with our understanding. Of the 12 stages, there were 10 stages, each of which represented an iteration of the page rank algo.

Fig: Stages in a Scala Job

3) DataFrames implementation gave a huge performance benefit over RDDs. This is as per our expectations since our dataset is structured, and dataframes are known to perform well on such datasets. Transformations are optimized on DataFrames since they store data in a named columnar format, so there is metadata to help perform query operations optimally.

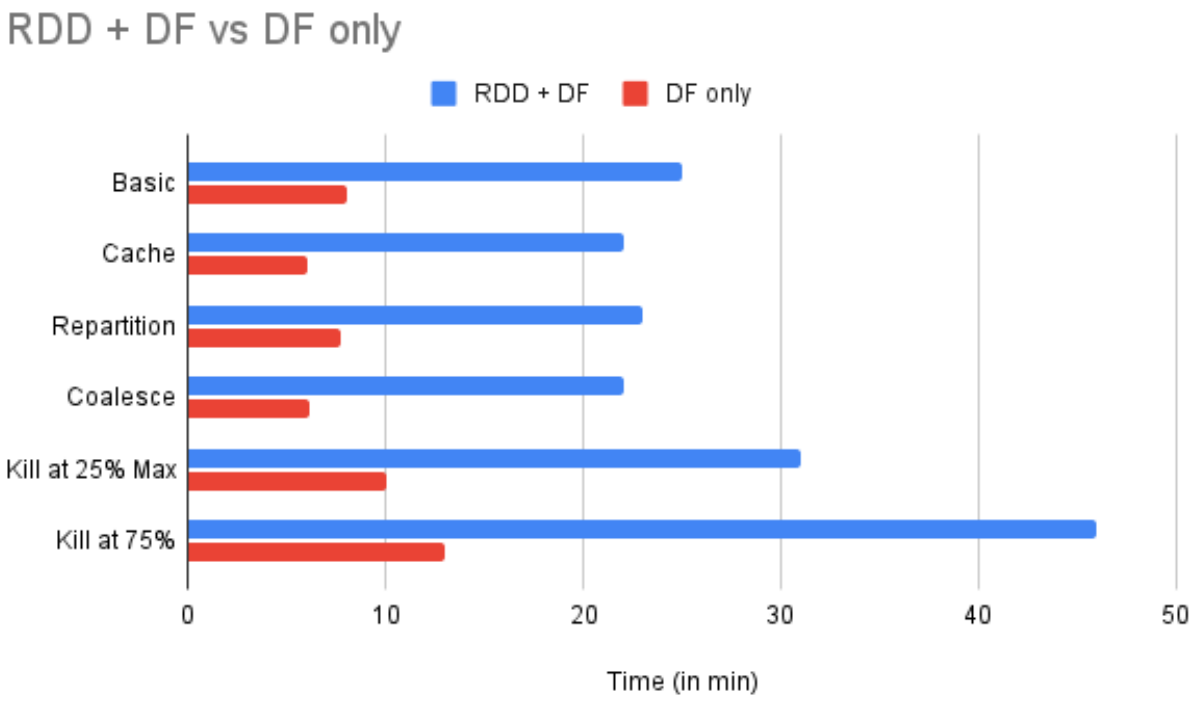
Performance Metrics

Collection Methodology

Metrics were collected using the Spark job web UI and the *dstat* utility. Since *dstat* is not installed by default on Ubuntu systems, we installed it with *apt*. Metrics were averaged over sixty second intervals. The network activity represents the total activity across all network interfaces. Similarly, I/O activity is measured across all devices. Data sizes are measured in bytes unless specified otherwise.

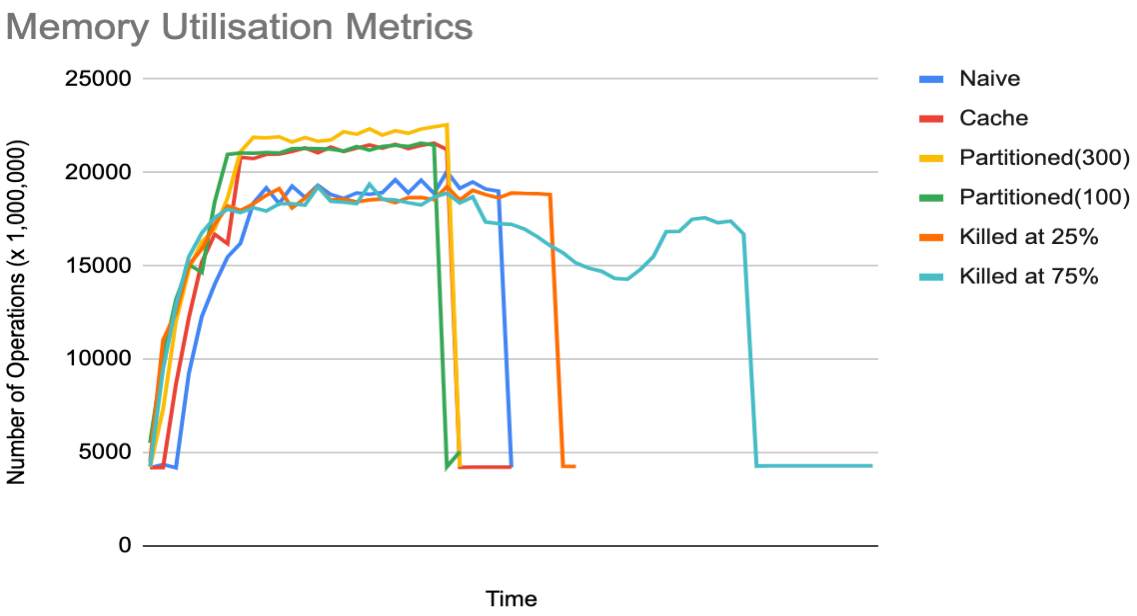
Completion Time

Both cached and repartitioned implementations outperform the naive implementation of PageRank in terms of job completion time. The performance improvement by caching is more significant than partitioning. Killing the job mid-completion increases the overall time taken, however when the job is killed closer to completion, the time penalty is more severe. This is because Spark would need to apply even more transformations to recompute the lost partitions. We also observed that implementations using only Dataframe APIs outperformed implementations using a mix of RDD and Dataframe APIs across-the-board.



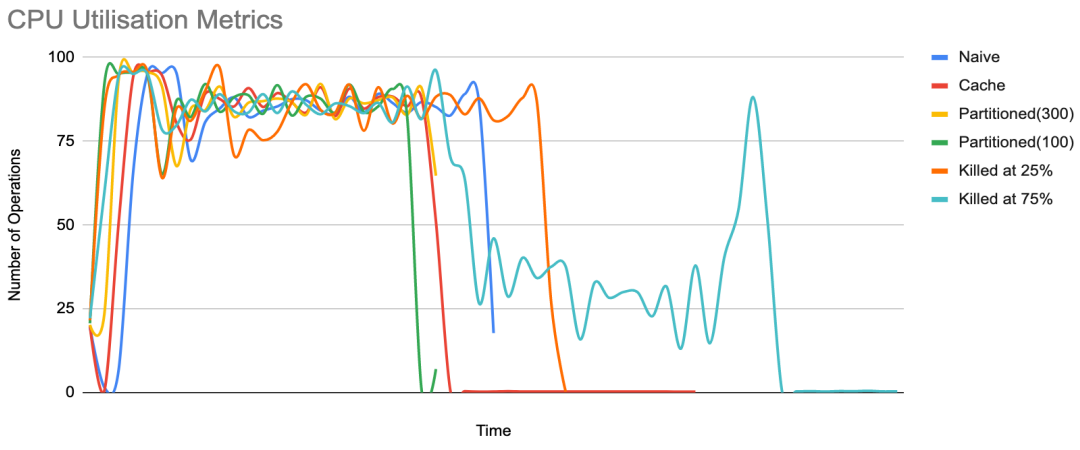
Memory Usage

We observed that overall memory usage was lower for naive PageRank runs compared to cached or partitioned runs. This is inline with our expectations as caching RDDs in memory should lead to higher memory utilization. We also observed a higher memory utilization for partitioned runs which might probably be due to shuffling overheads. Increasing the number of partitions from hundred to three-hundred did not come with a significant impact to memory usage.



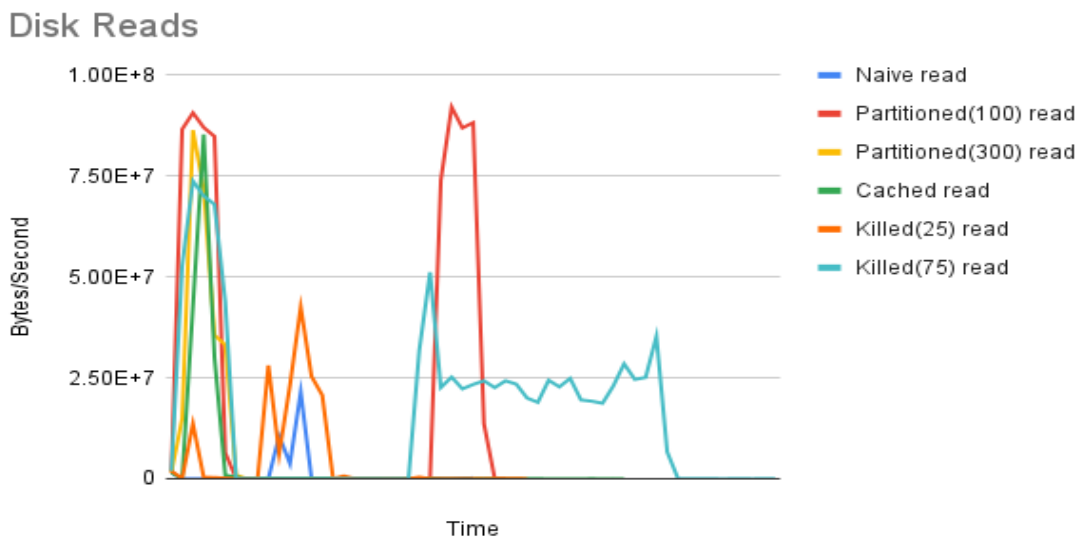
CPU Usage

CPU Usage seems to be pretty similar across all jobs however, the jobs that were killed at 25% and 75% of their completion took much longer to complete since some partitions that were lost were re-computed.

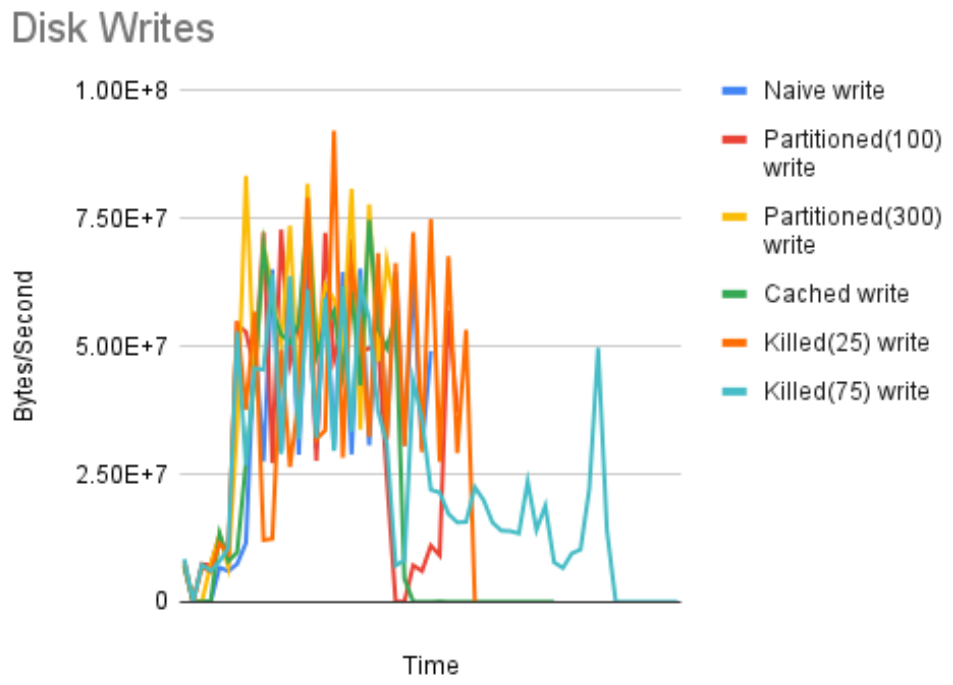


Disk I/O

We observed an initial spike in read activity on all jobs as data was being loaded. Subsequently we see spikes in disk activity when tasks are killed mid-completion. This is likely because the lost partitions needed to be rebuilt and thus the data was loaded from disk again.

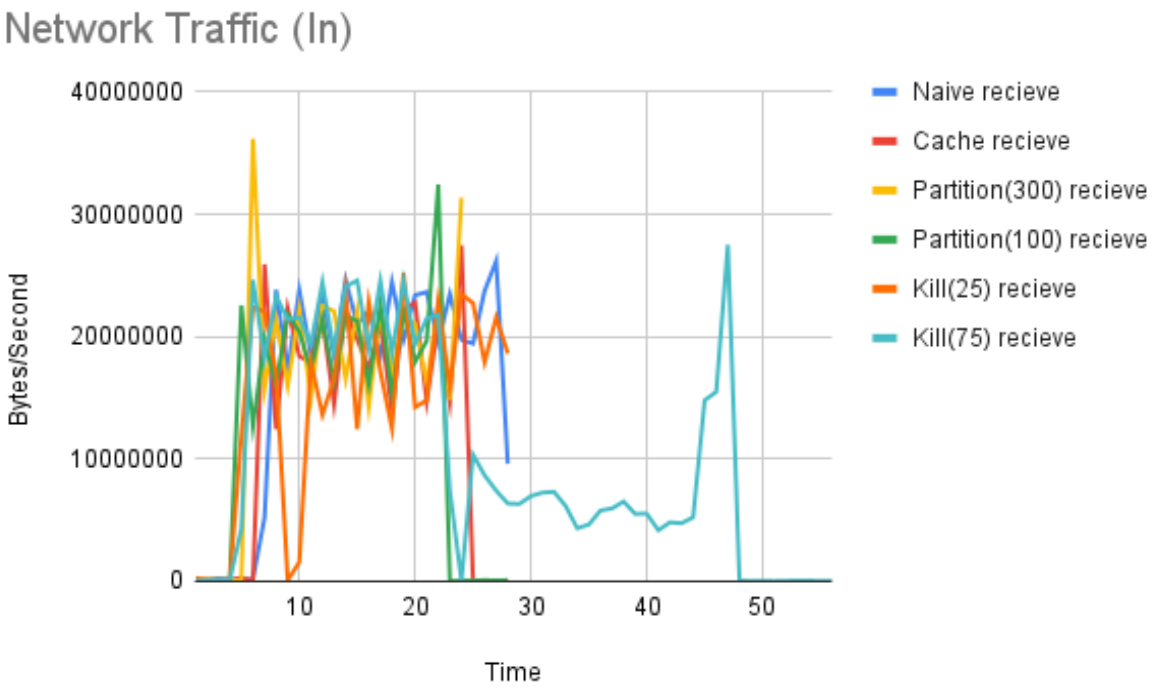


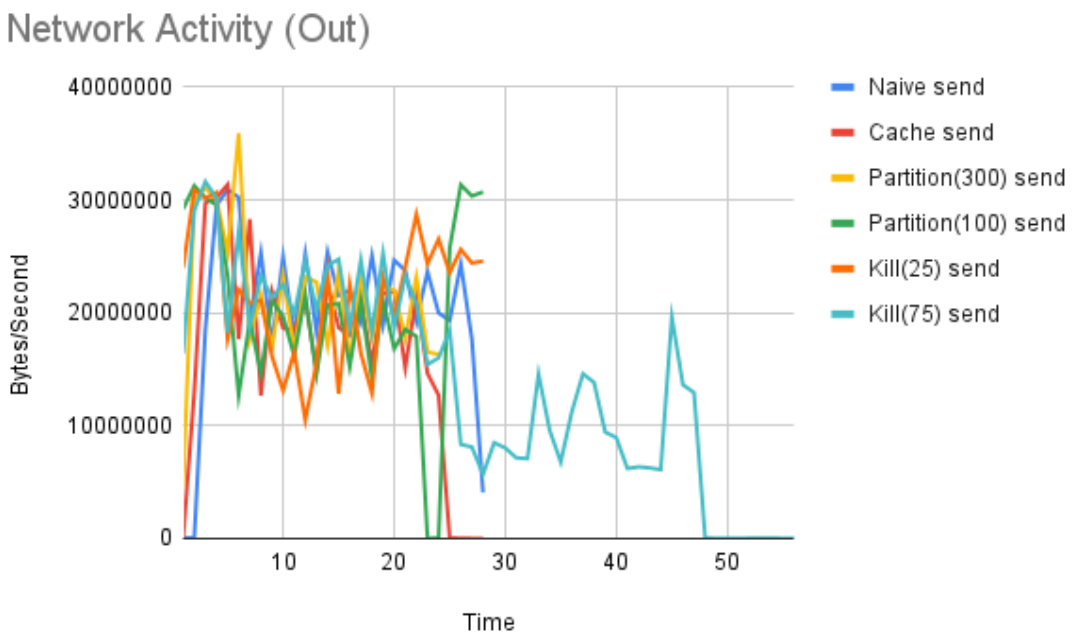
We observe a pretty similar writing pattern across all types of jobs. There are spikes when intermediate computation is finished and data is flushed to disk and periods of low write activity when computations are taking place.



Network Activity

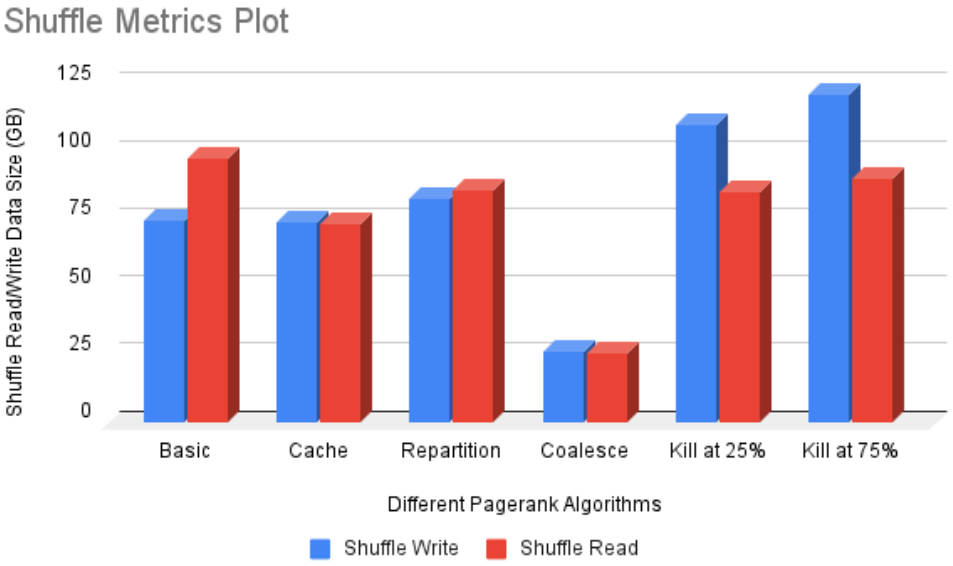
Overall network activity patterns are pretty similar across the jobs, however we observed slightly higher network activity with the partitioned job with three-hundred partitions. This is likely due to the initial data shuffling operations. Cached PageRank also seems to have slightly lower network activity over the Naive implementation probably due to lesser information exchange needed amongst nodes.





Shuffle Read/ Write Data

Shuffle read/write data usage increases with repartition to a larger number. We attribute this to the shuffling operation in *repartition()* function. On the other hand, we see a significant decrease in the shuffles when using *coalesce()* to a lesser number of partitions. In the case of bringing a node down, we observe that the shuffling rate has increased significantly since the intermediate data structures have to be recomputed from lineage for the partitions present on the lost node. The stages include *reduceByKey()*, *groupByKey()* and *join()* operations which will lead to shuffling.



Code Validation:

1) We created 2 small test datasets for testing purposes which we are including in our folder. (TestSet in file: test-data1.txt, test-data2.txt). The results matched the computed values.

2) Web-graph dataset description: We computed some statistics on web graph data to validate the correctness of our code.
   Total Nodes : 685230
   Source Nodes(src) : 680486
   Sink Nodes(snk) : 617094
   Intersection of Nodes : 612350
   Nodes with only outgoing edges : 68136
   Nodes with only incoming edges : 4744

   With our initial implementation of inner join, the dataset size after each join operation was changing because inner join takes only nodes which are present in both datasets. So the nodes which had only outgoing edges were skipped. This led to the understanding that the correct operation for this join would be leftOuterJoin. We are filling the null values in join with rank 1. Finally, we write our ranks data on HDFS for all nodes except the nodes with only outgoing edges.

3) EnWiki-pages dataset:
   Our python implementation gives us around ~20 million distinct nodes with ranks at the end of all 10 iterations.

Member Contributions

1) Each member setup Hadoop and Spark on the cluster at-least once.
2) All members looked at the implementation for the Simple Spark Sorting application. The implementation was done in Scala by Deepti Rajagopal.
3) For the Pagerank algorithm, every member wrote individual PageRank implementations. Chahak Tharani and Akul Gupta wrote their code using Python (PySpark) and Deepti in Scala. After discussion, we moved forward with one PySpark implementation that everyone contributed to.
4) Chahak and Deepti worked on optimizing the code from RDDs to a mixture of RDDs and DataFrames and then to just DataFrames.
5) Akul modularized the final code, fixed bugs, and added comments to the codebase. Akul also wrote the run.sh launch script for the PySpark jobs.
6) Akul and Chahak looked into and tried different tools and methods for collecting system metrics.
7) Chahak and Deepti worked on code correctness verification (on test and Berkley dataset) and collected system metrics for different algorithms.
8) All members contributed to this report and added their observations and inferences to it.

References

[1] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica : Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

[2] Spark reference implementation for PageRank https://github.com/apache/spark/blob/master/examples/src/main/python/pagerank.py (Accessed on Sep 24 2021)

[3] Apache Spark RDD Tutorial https://sparkbyexamples.com/spark-rdd-tutorial/ (Accessed Sept 27 2021)

[4] PySpark Tutorial For Beginners https://sparkbyexamples.com/pyspark-tutorial/ (Accessed Sept 27 2021)

[5] Apache Spark Docks https://spark.apache.org/docs/latest/ (Accessed Sept 24 2021)