CS744 Assignment1 Report

Group 14

Suyash Raj | Amr Alazali | Handan Hu

Part 0: Environment Setup

Our group set up the environment on Cloudlab and created a cluster of 3 VMs.
We have executed the required commands, including adding additional storage, installing docker and verifying overlay network.

Part 1: Software Deployment

We followed the provided instructions and successfully deployed Apache Hadoop and Spark.

- For Hadoop, we launched the namenode and datanodes with 3 replications.
- For Spark, we launched one master and three worker nodes.

As a result, we can check HDFS files on *http://10.10.1.1:9870* and visit Spark UI on *http://10.10.1.1:8080*

Part 2: A Simple Spark Application

We utilized PySpark to implement this simple application:

1. Read exports.csv file from HDFS
2. Sort DataFrames based on country code and timestamp
3. Store the output to HDFS in csv format

Part 3: PageRank

Task 1

We implemented the PageRank algorithm with Python as follows:

1. Read the file and parse the edges.
2. Group all neighbors by their source node and initialize every node's rank to 1.0
3. For each iteration, have each node distribute its share of rank to its neighbors, then sum contributions to get new ranks.

This program took **42 minutes** to execute, including one job with **12 stages** and **1164 tasks**, as shown in the two figures below:

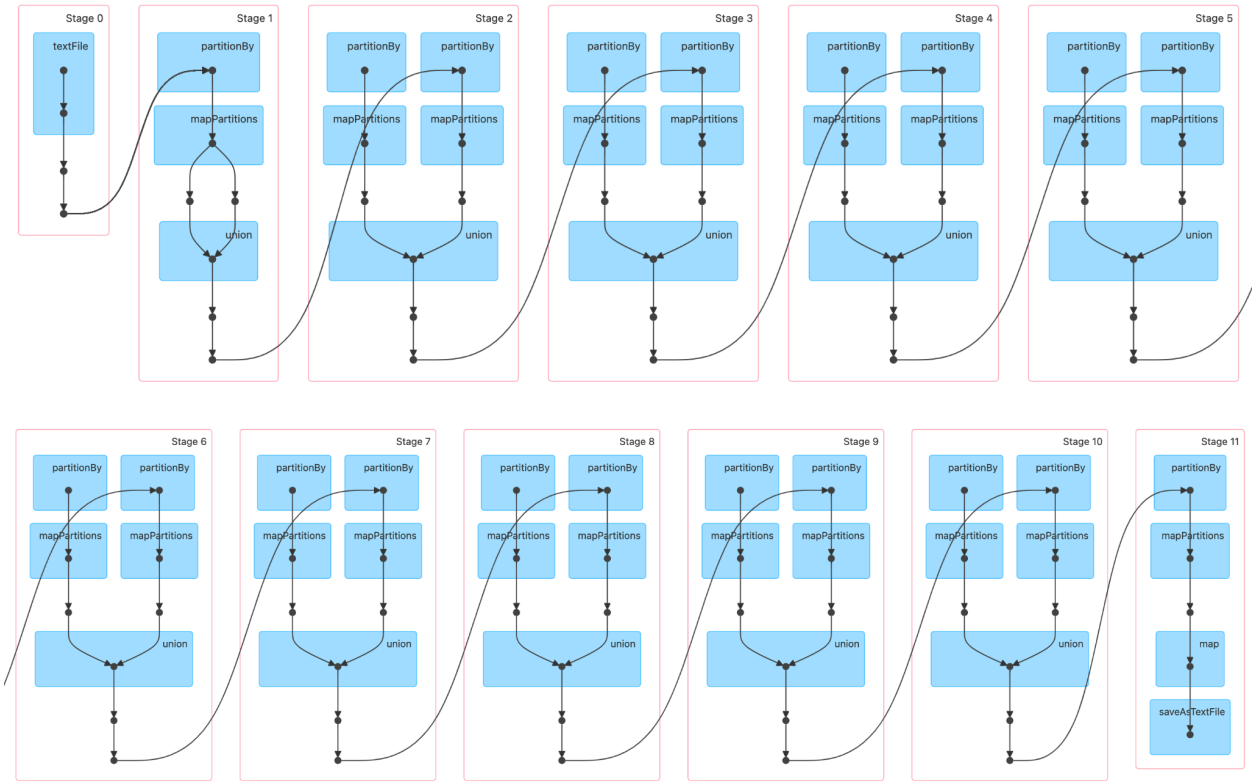


Fig 1.1 DAG Visualization

Stage Id ▾	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
11	runJob at SparkHadoopWriter.scala:83	+details	2025/02/04 22:48:45	22 s	97/97		1243.6 MiB	2.5 GiB	
10	reduceByKey at /src/t3.py:53	+details	2025/02/04 22:44:45	4.0 min	97/97			6.1 GiB	2.5 GiB
9	reduceByKey at /src/t3.py:53	+details	2025/02/04 22:40:37	4.1 min	97/97			6.1 GiB	2.5 GiB
8	reduceByKey at /src/t3.py:53	+details	2025/02/04 22:36:36	4.0 min	97/97			6.1 GiB	2.5 GiB
7	reduceByKey at /src/t3.py:53	+details	2025/02/04 22:32:42	3.9 min	97/97			6.1 GiB	2.5 GiB
6	reduceByKey at /src/t3.py:53	+details	2025/02/04 22:28:49	3.9 min	97/97			6.1 GiB	2.5 GiB
5	reduceByKey at /src/t3.py:53	+details	2025/02/04 22:24:50	4.0 min	97/97			6.1 GiB	2.5 GiB
4	reduceByKey at /src/t3.py:53	+details	2025/02/04 22:20:56	3.9 min	97/97			6.2 GiB	2.5 GiB
3	reduceByKey at /src/t3.py:53	+details	2025/02/04 22:17:04	3.9 min	97/97			6.3 GiB	2.6 GiB
2	reduceByKey at /src/t3.py:53	+details	2025/02/04 22:13:29	3.6 min	97/97			7.4 GiB	2.8 GiB
1	reduceByKey at /src/t3.py:53	+details	2025/02/04 22:08:36	4.9 min	97/97			7.2 GiB	3.8 GiB
0	groupByKey at /src/t3.py:35	+details	2025/02/04 22:06:46	1.8 min	97/97	9.9 GiB			3.6 GiB

Fig 1.2 Stage Status

Observations

- When running this program on the smaller dataset (Berkeley-Stanford web graph), the job got done in less than 5 minutes - averaging about 3 min per run.
- We verified the correctness of our results by writing normal python code using NumPy library and running it on the smaller dataset and then comparing the final results.
- When running on the larger dataset (enwiki-pages-articles), it took about 33 minutes to execute on average.

Run	Cores	Memory per Executor	Duration
1	15	30.0 GiB	31 minutes
2	15	30.0 GiB	35 minutes

Task 2

In this task, we explored different partitioning strategies that could impact the performance. We changed the number of partitions using `df.repartition()` and observed the execution time. Specifically, we experimented with partition of 10, 50, 100, 200 and 300, and then compared the result.

Version	App ID	App Name	Started	Completed	Duration	Spark User	Last Updated
3.3.4	app-20250205012352-0018	t2-10	2025-02-04 19:23:49	2025-02-04 20:23:06	59 min	root	2025-02-04 20:23:06
3.3.4	app-20250205005223-0017	t2-100	2025-02-04 18:52:20	2025-02-04 19:17:56	26 min	root	2025-02-04 19:17:56
3.3.4	app-20250205050307-0020	t2-200	2025-02-04 23:03:04	2025-02-04 23:37:23	34 min	root	2025-02-04 23:37:24
3.3.4	app-20250205024641-0019	t2-300	2025-02-04 20:46:39	2025-02-04 21:35:58	49 min	root	2025-02-04 21:35:58
3.3.4	app-20250204235419-0016	t2-50	2025-02-04 17:54:17	2025-02-04 18:28:40	34 min	root	2025-02-04 18:28:41

Fig 2.1 completed apps with different partitions

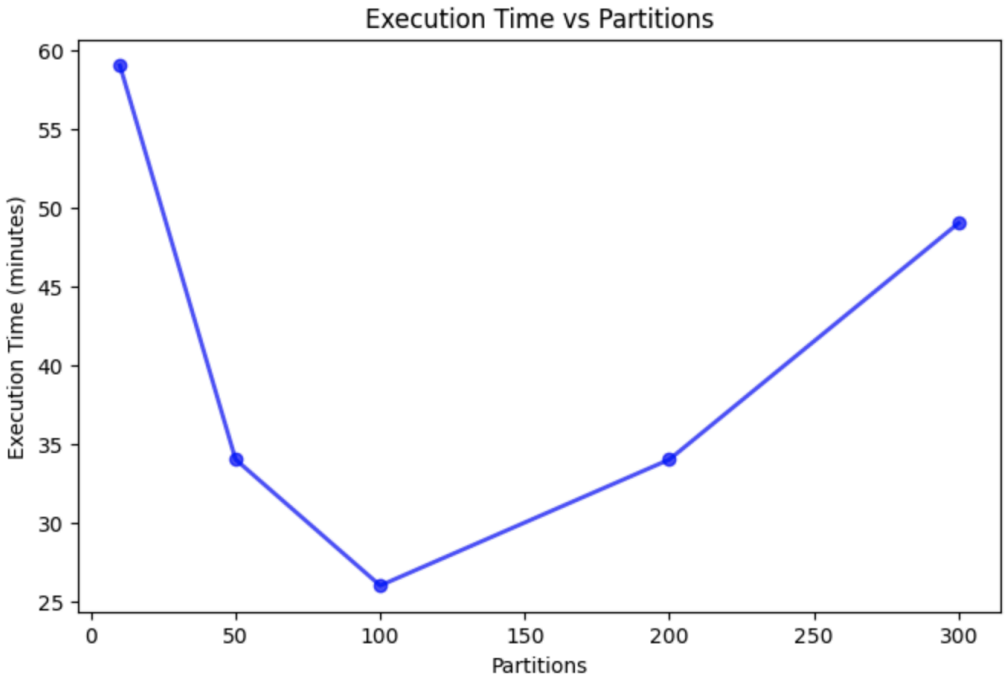


Fig 2.2 Execution Time vs Partitions

From figure 2.2 above, we can see that execution time initially decreases as the number of partitions increases but then starts to rise again.

- **With smaller partitions** (10 and 50 partitions), Spark cannot efficiently parallelize the workload, which leads to more execution time. And if one task takes longer than others, it can also slow down the overall job.
- **With larger partitions** (200 and 300 partitions), overhead increases due to excessive small tasks. Each task has scheduling overhead, and frequent data shuffling between nodes can have a negative impact on the performance.
- In our experiment, **the optimal partition size** should be around 100 partitions, because it takes significantly less time compared to others, indicating a balance between parallelism and overhead.

Task 3

For this task we investigated how caching affected the performance of our pagerank algorithm.

Experiment (caching the adjacency list):

Using `df.cache()` we cached the RDD of the links dataframe. The `links` dataframe stores an adjacency list of the nodes in our graph. As shown below:

```
# source node => (node, [neighbors...])
links = edges.groupByKey().repartition(partition).cache()
```

We compared the performance of the spark job with the `links` dataframe cached to the spark job with no caching, while keeping the number of partitions (100) and number of workers (2) the same.

We chose to cache the `links` because each iteration of our pagerank algorithm we call `df.join()` on the `links` dataframe as shown below:

```
for i in range(NUM_ITERATIONS):
    contributions = links.join(ranks).flatMap(lambda n: calc_rank(n))
    ranks = contributions.reduceByKey(lambda x, y: x + y)
                         .mapValues(lambda rank: 0.15 + 0.85 * rank)
```

Observations:

With Caching:

The cached spark job took **35 minutes** to execute.

Below is the shuffle reads at each stage of the spark job:

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read
22	runJob at SparkHadoopWriter.scala:83	+details	2025/02/06 00:12:53	1.2 min	1100/1100		1243.6 MiB	3.7 GiB
21	reduceByKey at /src/t3.py:54	+details	2025/02/06 00:11:17	1.6 min	1100/1100			4.4 GiB
20	join at /src/t3.py:53	+details	2025/02/06 00:09:34	1.7 min	1100/1100	3.6 GiB		3.6 GiB
19	reduceByKey at /src/t3.py:54	+details	2025/02/06 00:08:01	1.5 min	1000/1000			4.4 GiB
18	join at /src/t3.py:53	+details	2025/02/06 00:06:22	1.7 min	1000/1000	3.6 GiB		3.5 GiB
17	reduceByKey at /src/t3.py:54	+details	2025/02/06 00:04:53	1.5 min	900/900			4.4 GiB
16	join at /src/t3.py:53	+details	2025/02/06 00:03:18	1.6 min	900/900	3.6 GiB		3.4 GiB
15	reduceByKey at /src/t3.py:54	+details	2025/02/06 00:01:51	1.4 min	800/800			4.4 GiB
14	join at /src/t3.py:53	+details	2025/02/06 00:00:20	1.5 min	800/800	3.6 GiB		3.3 GiB
13	reduceByKey at /src/t3.py:54	+details	2025/02/05 23:58:54	1.4 min	700/700			4.4 GiB
12	join at /src/t3.py:53	+details	2025/02/05 23:57:26	1.5 min	700/700	3.6 GiB		3.1 GiB
11	reduceByKey at /src/t3.py:54	+details	2025/02/05 23:56:04	1.4 min	600/600			4.4 GiB
10	join at /src/t3.py:53	+details	2025/02/05 23:54:42	1.4 min	600/600	3.6 GiB		3.0 GiB
9	reduceByKey at /src/t3.py:54	+details	2025/02/05 23:53:22	1.3 min	500/500			4.4 GiB
8	join at /src/t3.py:53	+details	2025/02/05 23:52:02	1.3 min	500/500	3.6 GiB		2.9 GiB
7	reduceByKey at /src/t3.py:54	+details	2025/02/05 23:50:41	1.3 min	400/400			4.4 GiB
6	join at /src/t3.py:53	+details	2025/02/05 23:49:21	1.3 min	400/400	3.6 GiB		3.0 GiB
5	reduceByKey at /src/t3.py:54	+details	2025/02/05 23:47:53	1.5 min	300/300			4.7 GiB
4	join at /src/t3.py:53	+details	2025/02/05 23:46:17	1.6 min	300/300	3.6 GiB		4.2 GiB
3	reduceByKey at /src/t3.py:54	+details	2025/02/05 23:44:15	2.0 min	200/200			3.7 GiB
2	join at /src/t3.py:53	+details	2025/02/05 23:42:50	1.4 min	200/200	3.6 GiB		3.6 GiB
1	coalesce at NativeMethodAccessorImpl.java:0	+details	2025/02/05 23:42:18	31 s	97/97			3.6 GiB
0	groupByKey at /src/t3.py:36	+details	2025/02/05 23:39:39	2.7 min	97/97	9.9 GiB		

Each sequential pair of join and reduceByKey tasks corresponds to a single iteration of the loop in our pagerank algorithm (see above code). As shown boxed in the above figure, a single iteration of the algorithm results in around **4.4 GiB** of data shuffle read by the `df.reduceByKey()` transformation and results in around **3.1 GiB** data shuffle read by the `df.join()` transformation.

Without Caching:

The spark job without caching also took **35 minutes** to execute.
However, we observe a significant difference in the amount of data shuffle read as shown below:

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read
22	runJob at SparkHadoopWriter.scala:83	+details	2025/02/06 01:06:36	1.2 min	1100/1100		1243.6 MiB	3.7 GiB
21	reduceByKey at /src/t3.py:54	+details	2025/02/06 01:05:00	1.6 min	1100/1100			4.4 GiB
20	join at /src/t3.py:53	+details	2025/02/06 01:02:57	2.1 min	1100/1100			7.2 GiB
19	reduceByKey at /src/t3.py:54	+details	2025/02/06 01:01:24	1.5 min	1000/1000			4.4 GiB
18	join at /src/t3.py:53	+details	2025/02/06 00:59:44	1.7 min	1000/1000			7.1 GiB
17	reduceByKey at /src/t3.py:54	+details	2025/02/06 00:58:15	1.5 min	900/900			4.4 GiB
16	join at /src/t3.py:53	+details	2025/02/06 00:56:39	1.6 min	900/900			7.0 GiB
15	reduceByKey at /src/t3.py:54	+details	2025/02/06 00:55:12	1.4 min	800/800			4.4 GiB
14	join at /src/t3.py:53	+details	2025/02/06 00:53:40	1.5 min	800/800			6.9 GiB
13	reduceByKey at /src/t3.py:54	+details	2025/02/06 00:52:15	1.4 min	700/700			4.4 GiB
12	join at /src/t3.py:53	+details	2025/02/06 00:50:47	1.5 min	700/700			6.8 GiB
11	reduceByKey at /src/t3.py:54	+details	2025/02/06 00:49:25	1.4 min	600/600			4.4 GiB
10	join at /src/t3.py:53	+details	2025/02/06 00:48:01	1.4 min	600/600			6.6 GiB
9	reduceByKey at /src/t3.py:54	+details	2025/02/06 00:46:41	1.3 min	500/500			4.4 GiB
8	join at /src/t3.py:53	+details	2025/02/06 00:44:53	1.8 min	500/500			6.5 GiB
7	reduceByKey at /src/t3.py:54	+details	2025/02/06 00:43:34	1.3 min	400/400			4.4 GiB
6	join at /src/t3.py:53	+details	2025/02/06 00:42:14	1.3 min	400/400			6.6 GiB
5	reduceByKey at /src/t3.py:54	+details	2025/02/06 00:40:46	1.5 min	300/300			4.7 GiB
4	join at /src/t3.py:53	+details	2025/02/06 00:39:09	1.6 min	300/300			7.8 GiB
3	reduceByKey at /src/t3.py:54	+details	2025/02/06 00:37:04	2.1 min	200/200			3.7 GiB
2	join at /src/t3.py:53	+details	2025/02/06 00:36:08	56 s	200/200			7.2 GiB
1	coalesce at NativeMethodAccessorImpl.java:0	+details	2025/02/06 00:35:38	30 s	97/97			3.6 GiB
0	groupByKey at /src/t3.py:36	+details	2025/02/06 00:33:13	2.4 min	97/97	9.9 GiB		

Without caching each iteration of our algorithm results in around **7.7 GiB** of data shuffle read by the `df.join()` transformation, as opposed to **3.1 GiB** data shuffle read by `df.reduceByKey()` in the job with caching.

Discussion:

Caching significantly reduces the amount of shuffle reads during each iteration of our page rank algorithm, by reducing the amount of data shuffle read by each `df.join()` called on the `links` dataframe from **7.7 GiB** to **3 GiB** on average.

Caching the `links` dataframe significantly reduces data shuffle reads because it preserves the expensive shuffle work (like grouping and partitioning) that happens during its creation, allowing subsequent iterations to reuse this data. For example, after grouping the edges:

```
# source node => (node, [neighbors...])
links = edges.groupByKey().repartition(partition).cache()
```

the pre-partitioned `links` is reused in every iteration when it's joined with `ranks`.

```
contributions = links.join(ranks).flatMap(calc_rank)
```

Without caching, Spark would re-execute the full grouping and shuffle each time `links` is needed, leading to a much higher volume of network I/O and slower overall performance.

Task 4

We ran the task with 100 partitions, in order to optimize our running time.

Experiment 1 (one executor killed at 25%)

- Started the job with 100 partitions, normal completion time - 26 minutes
- After 6.5 minutes had passed, scaled down the worker pods using
    - `docker service scale worker=2`
- Very soon, the application threw *org.apache.spark.shuffle.FetchFailedException*

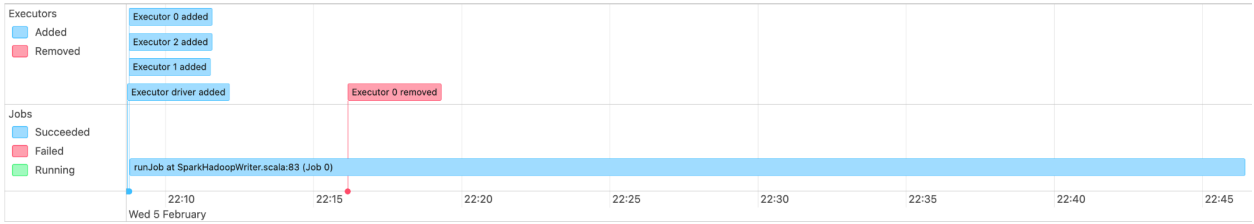
```
25/02/05 22:16:12 INFO TaskSetManager: task 114.0 in stage 6.0 (TID 1308)
failed, but the task will not be re-executed (either because the task
failed with a shuffle data fetch failure, so the previous stage needs to be
re-run, or because a different copy of the task has already succeeded).

25/02/05 22:16:12 WARN TaskSetManager: Lost task 111.0 in stage 6.0 (TID
1305) (10.0.1.150 executor 2): FetchFailed(null, shuffleId=17, mapIndex=-1,
mapId=-1, reduceId=11, message=

org.apache.spark.shuffle.MetadataFetchFailedException: Missing an output
location for shuffle 17 partition 11
```

```
25/02/05 22:16:12 INFO TaskSetManager: task 114.0 in stage 6.0 (TID 1308) failed, but the task will not be re-executed (either because the task failed with a shuffle
 data fetch failure, so the previous stage needs to be re-run, or because a different copy of the task has already succeeded).
25/02/05 22:16:12 WARN TaskSetManager: Lost task 111.0 in stage 6.0 (TID 1305) (10.0.1.150 executor 2): FetchFailed(null, shuffleId=17, mapIndex=-1, mapId=-1, reduce
Id=11, message=
org.apache.spark.shuffle.MetadataFetchFailedException: Missing an output location for shuffle 17 partition 11
        at org.apache.spark.MapOutputTracker$.validateStatus(MapOutputTracker.scala:1705)
        at org.apache.spark.MapOutputTracker$.$anonfun$convertMapStatuses$10(MapOutputTracker.scala:1652)
        at org.apache.spark.MapOutputTracker$.$anonfun$convertMapStatuses$10$adapted(MapOutputTracker.scala:1651)
        at scala.collection.Iterator.foreach(Iterator.scala:943)
        at scala.collection.Iterator.foreach$(Iterator.scala:943)
        at scala.collection.AbstractIterator.foreach(Iterator.scala:1431)
        at org.apache.spark.MapOutputTracker$.convertMapStatuses(MapOutputTracker.scala:1651)
        at org.apache.spark.MapOutputTrackerWorker.getMapSizesByExecutorIdImpl(MapOutputTracker.scala:1294)
        at org.apache.spark.MapOutputTrackerWorker.getMapSizesByExecutorId(MapOutputTracker.scala:1256)
        at org.apache.spark.shuffle.sort.SortShuffleManager.getReader(SortShuffleManager.scala:140)
        at org.apache.spark.shuffle.ShuffleManager.getReader(ShuffleManager.scala:63)
        at org.apache.spark.shuffle.ShuffleManager.getReader$(ShuffleManager.scala:57)
        at org.apache.spark.shuffle.sort.SortShuffleManager.getReader(SortShuffleManager.scala:73)
        at org.apache.spark.rdd.ShuffledRDD.compute(ShuffledRDD.scala:106)
        at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:365)
        at org.apache.spark.rdd.RDD.iterator(RDD.scala:329)
        at org.apache.spark.rdd.MapPartitionsRDD.compute(MapPartitionsRDD.scala:52)
        at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:365)
        at org.apache.spark.rdd.RDD.iterator(RDD.scala:329)
        at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:65)
        at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:365)
        at org.apache.spark.rdd.RDD.iterator(RDD.scala:329)
        at org.apache.spark.rdd.UnionRDD.compute(UnionRDD.scala:106)
        at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:365)
        at org.apache.spark.rdd.RDD.iterator(RDD.scala:329)
        at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:65)
        at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:365)
        at org.apache.spark.rdd.RDD.iterator(RDD.scala:329)
        at org.apache.spark.api.python.PairwiseRDD.compute(PythonRDD.scala:115)
        at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:365)
        at org.apache.spark.rdd.RDD.iterator(RDD.scala:329)
        at org.apache.spark.shuffle.ShuffleWriteProcessor.write(ShuffleWriteProcessor.scala:59)
        at org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scala:99)
        at org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scala:52)
        at org.apache.spark.scheduler.Task.run(Task.scala:136)
        at org.apache.spark.executor.Executor$TaskRunner.$anonfun$run$3(Executor.scala:548)
        at org.apache.spark.util.Utils$.tryWithSafeFinally(Utils.scala:1504)
        at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:551)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
```

- Overall the job took **38 minutes to finish**



- Out of this about 1.1 minutes were spent in writing
- Out of **22 stages** - **7 had to be retried**

Experiment 2 (one executor killed at 75%)

- Started the job with 100 partitions, normal completion time - 26 minutes
- After 19.5 minutes had passed, scaled down the worker pods using
    - `docker service scale worker=2`

```
sneo@node0:~/cs744-sp25-a1$ docker service scale worker=2
worker scaled to 2
overall progress: 2 out of 2 tasks
1/2: running   [==================================================>]
2/2: running   [==================================================>]
verify: Service converged
```

- The same exceptions as in experiment 1 were reported
- Overall the job took **45 minutes to finish**

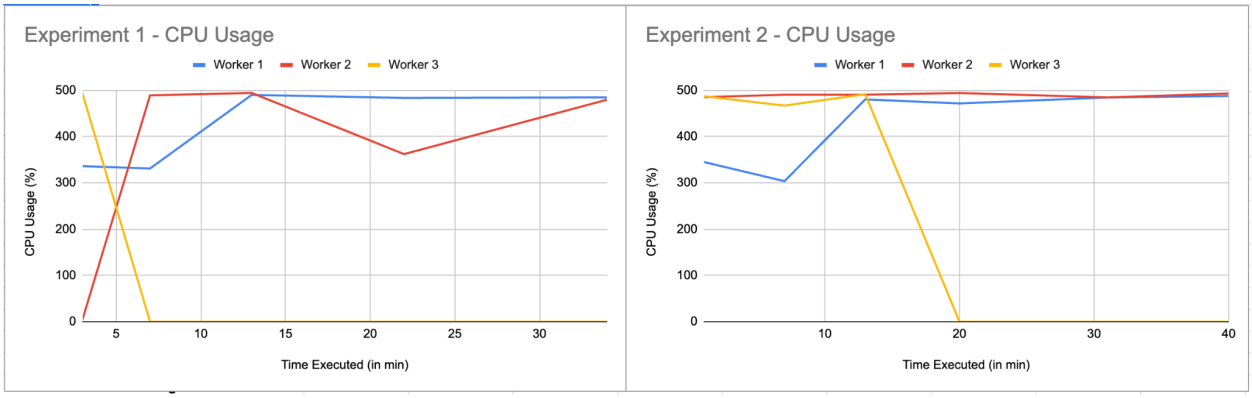
- Out of **22 stage - 16 had to be retried**

Observations

Experiment 1 vs 2

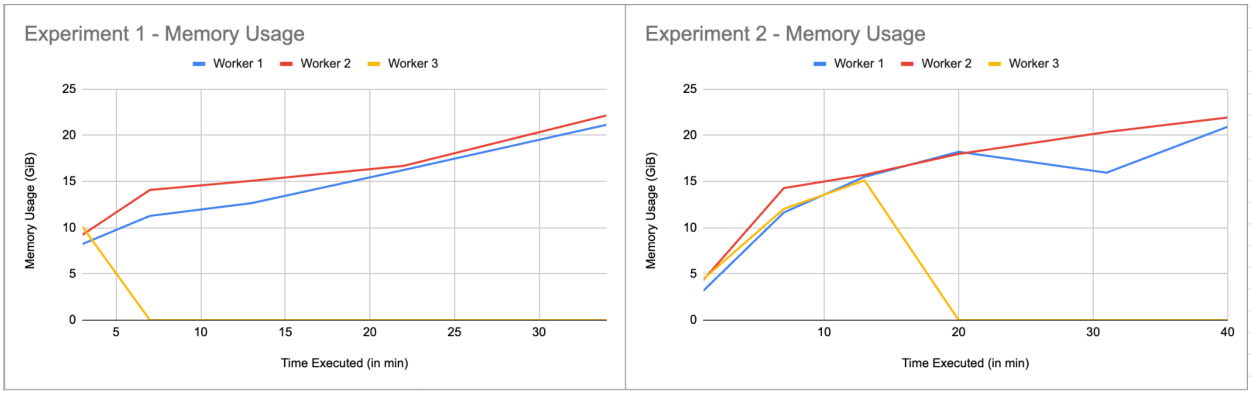
- Clearly experiment 2 takes longer to run
- As mentioned above, this is because in experiment 2
    - A worker was killed much later in the job's progression
    - As a result, more of the work had already been done (RDDs calculated), which were stored in that worker's memory
    - And those partitions were lost when the worker was brought down
    - As a result more stages had to be retried (partitions to be rebuilt)
        - 16 had to be retried as compared to 7 in experiment 1.

Spark Workers - CPU Usage Comparison



- We see very high CPU usage for spark workers in both cases.
- The usage for Worker 3 goes to 0 once it's terminated

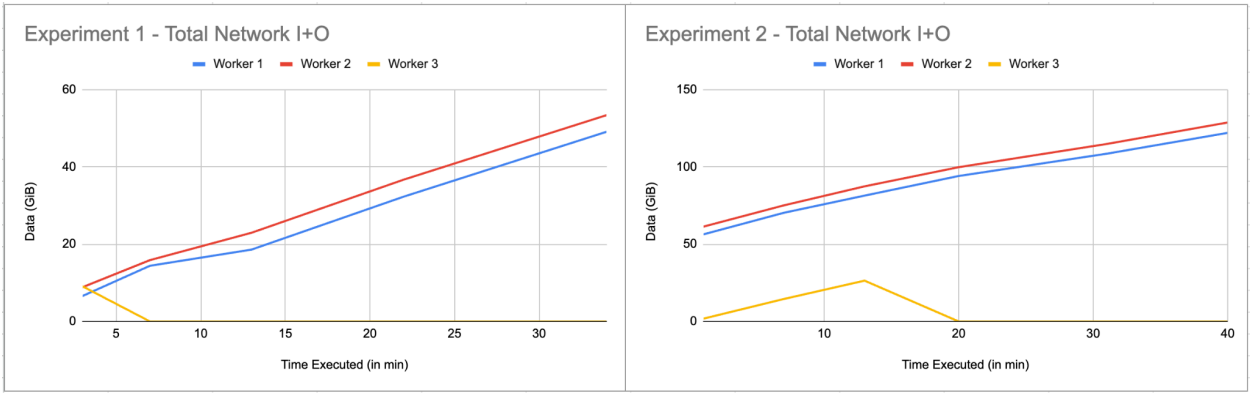
Spark Workers - Memory Usage Comparison



- Memory usage of all 3 workers grows parallelly until one of them is brought down

- At which point, the other two workers keep growing in their memory usage as expected.

Spark Workers - Total Network Input + Output



- These graphs show the total data transferred in and out of the two containers.
- However, please note that Workers 1&2 were not brought down between the two experiments, therefore the second graph shows net usage starting from where the first graph ends.
- However worker 3 starts from 0

Contributions

All team members participated in the process of coding, troubleshooting and report writing.

- **Suyash Raj**
  - Environment setup and deployment
  - Part3 - Task1, Task4
- **Amr Alazali**
  - Part2 A simple Spark application
  - Part3 - Task3 Caching
- **Handan Hu**
  - Part3 - Task1 PageRank algorithm
  - Part3 - Task2 Partition strategy