CS744 Assignment 1

Benita Kathleen Britto, Devansh Goenka, Hemalkumar Patel

Introduction

This assignment is designed to support our in-class understanding of how data analytics stacks work and get some hands-on experience in using them. We have deployed Apache Hadoop as the underlying file system and Apache Spark as the execution engine. We then implemented a Sorting application as a starter. Later, we implemented the PageRank algorithm as described in the assignment notes and verified it on a small dataset, i.e. Berkeley-Stanford web graph. We then executed the same script on enwiki-pages-articles, which is a much bigger dataset to experience Spark and Hadoop in action for larger datasets. This report contains our observations and findings.

Our code is hosted at: benitakbritto/CS744-P1 (github.com)

Part 0: Mounting disks

Our experiment with CloudLab spins 3 machines, each having only 16GB of disk space. However, it is not sufficient for our experiment. Our machines can quickly run out of disk space if we ignore these limitations. To solve this problem, we were given another 96GB disk that we mounted on each node and deployed HDFS.

Part 1: Software Deployment

Architecture:

As shown in the above figure, HDFS is used as underlying file systems. It will be involved to contain input files, output files and intermediate results generated by the framework. Then we run Spark cluster for the actual computation. Spark uses HDFS as an underlying storage system. Spark SQL used by DataFrame is run directly on top of Spark (it does automatically).

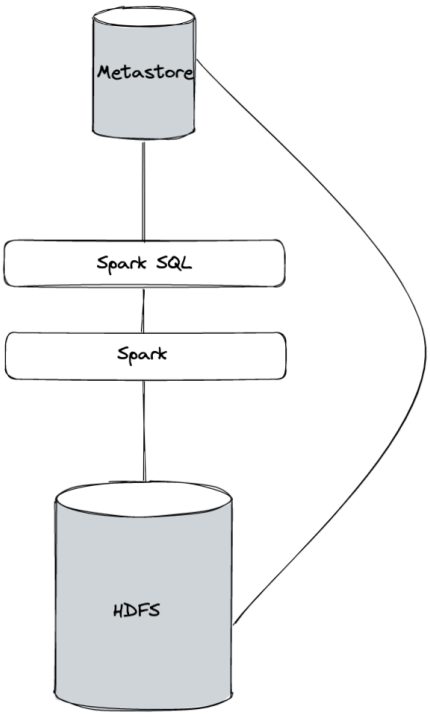


Fig 1: Deployment Stack

Hadoop:

We have 3 nodes, node0, node1 and node2. Node0 is set to be both the Namenode and the Datanode for HDFS, and the rest of them are used only as Datanodes.

Spark:

We have 3 nodes, node0, node1 and node2. Node0 acts as both Master and Worker, and the rest of them are used only as Workers.

To further verify that we have deployed the frameworks correctly, we ran the simple Spark application on the cluster and inspected the execution timeline (see figure below), where we could see all our 3 Worker nodes, each of them processing a max of 5 concurrent tasks at a time.

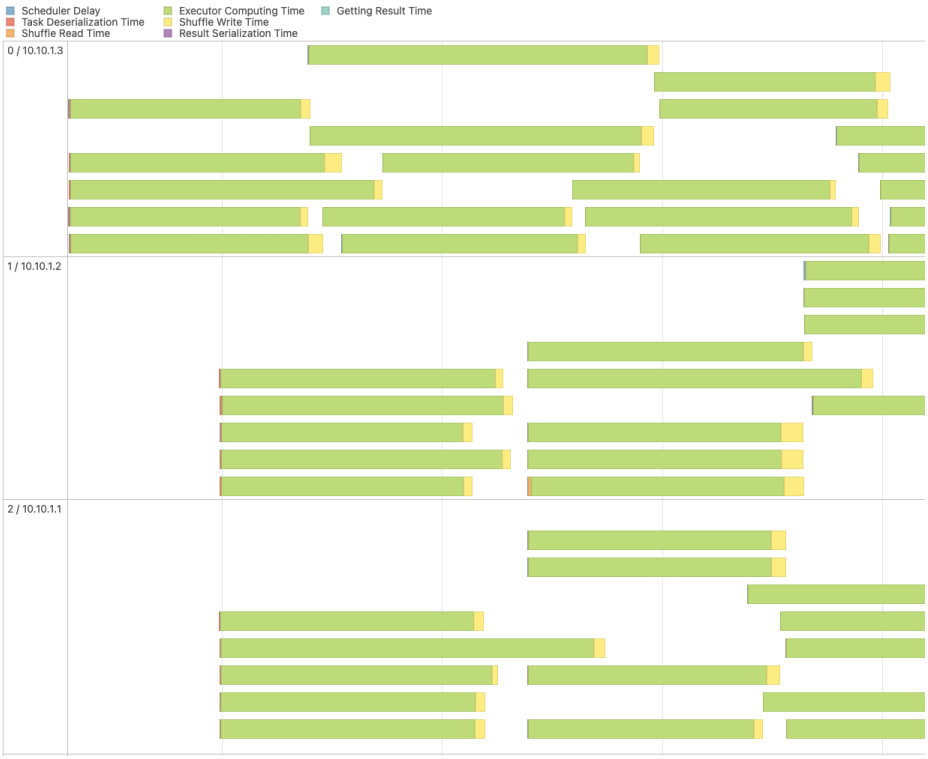


Fig 2: Event timeline of a stage of the a Spark computation

Part 2: A simple Spark application

We use PySpark to implement the Sorting algorithm. The figure below shows the lineage graph of the transformations involved in our implementation.

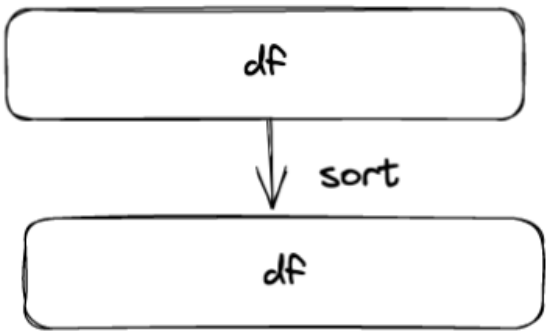
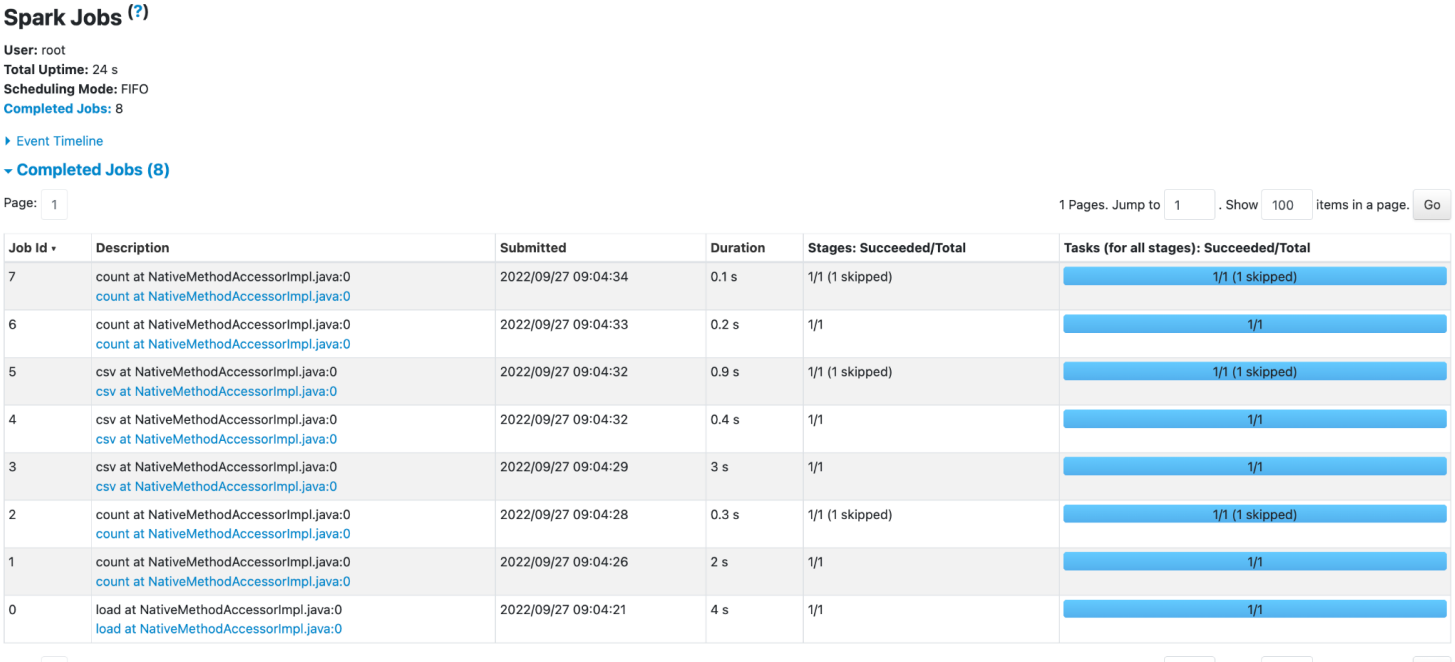


Fig 3: Sort lineage graph

The sorting algorithm completed on the cluster in 24 seconds. As we can see from the execution summary below, there was only 1 task per stage of the job because the entire data was small enough to fit in 1 partition.



Spark Jobs (?)

User: root
Total Uptime: 24 s
Scheduling Mode: FIFO
Completed Jobs: 8

▸ Event Timeline

▾ **Completed Jobs (8)**

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
7	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2022/09/27 09:04:34	0.1 s	1/1 (1 skipped)	1/1 (1 skipped)
6	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2022/09/27 09:04:33	0.2 s	1/1	1/1
5	csv at NativeMethodAccessorImpl.java:0 csv at NativeMethodAccessorImpl.java:0	2022/09/27 09:04:32	0.9 s	1/1 (1 skipped)	1/1 (1 skipped)
4	csv at NativeMethodAccessorImpl.java:0 csv at NativeMethodAccessorImpl.java:0	2022/09/27 09:04:32	0.4 s	1/1	1/1
3	csv at NativeMethodAccessorImpl.java:0 csv at NativeMethodAccessorImpl.java:0	2022/09/27 09:04:29	3 s	1/1	1/1
2	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2022/09/27 09:04:28	0.3 s	1/1 (1 skipped)	1/1 (1 skipped)
1	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2022/09/27 09:04:26	2 s	1/1	1/1
0	load at NativeMethodAccessorImpl.java:0 load at NativeMethodAccessorImpl.java:0	2022/09/27 09:04:21	4 s	1/1	1/1

Fig 4: Sorting App execution timeline

Part 3: PageRank

Task 1. Implementing PageRank

We used PySpark to implement the PageRank algorithm. The figure below shows the lineage graph of the transformations involved in our implementation.

To explain the logic briefly, we first create a `neighboursRDD` that stores the page to its links mapping. Further, we have another `ranksRDD` that stores the page to its rank mapping (initially set to 1). Then, in each iteration, we perform a join between these two RDDs and project the contributions to each link via a `flatMap` operation. The next stage is a `reduceByKey` which simply sums up all contributions towards a page and then the final equation for the new rank is mapped to the `ranksRDD`.

Finally, at the end of 10 iterations we store all the ranks of each node in HDFS. If the `ranksRDD` has `N` partitions at the end of the last iteration, we observe there are `N` different files that are written to HDFS.

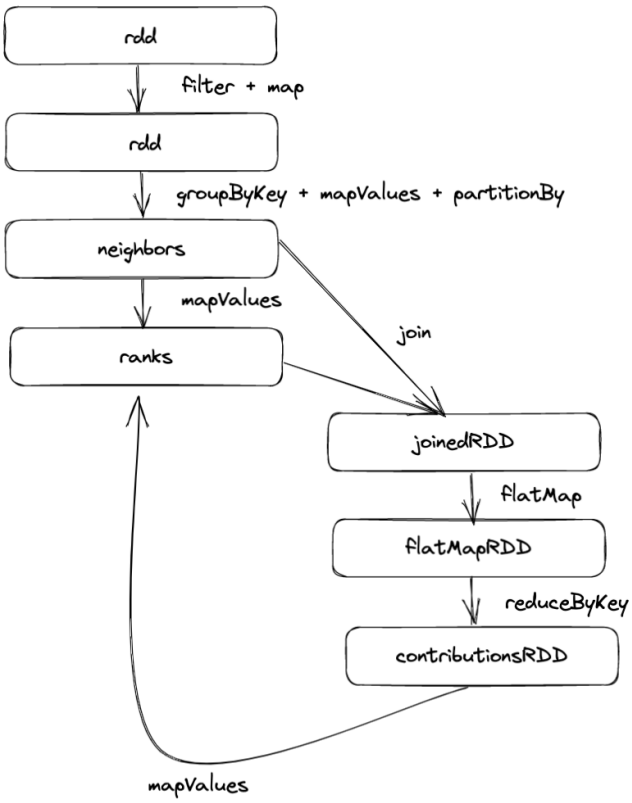


Fig 5: PageRank lineage graph

Task 2. Partitioning

For the small Berkley-Stanford dataset, we run the described algorithm on our cluster and observe that the job succeeds within ~4 minutes (see figure below). We do not try with different number of partitions due to the size of the input. We note that the input is small (~100 MB) and hence Spark creates just 2 partitions for this file, both of which reside in node0.

Spark Jobs (?)

User: root
Total Uptime: 4.3 min
Scheduling Mode: FIFO
Completed Jobs: 2

▸ Event Timeline

▾ **Completed Jobs (2)**

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	csv at NativeMethodAccessorImpl.java:0 csv at NativeMethodAccessorImpl.java:0	2022/09/27 09:08:55	2 s	1/1 (11 skipped)	2/2 (22 skipped)
0	runJob at PythonRDD.scala:166 runJob at PythonRDD.scala:166	2022/09/27 09:04:47	4.1 min	12/12	23/23

Fig 6: Execution Time for PageRank on a small dataset

We perform most of our analysis on the larger (~10GB) en-wiki dataset that was provided to us.

For this dataset, we deploy the PageRank algorithm using a different number of partitions for the shuffle operations as well as while writing to the disk. We see that the completion time reduces as we increase the number of partitions and plateaus around the 200 mark. The completion ime starts to increase again beyond 300 partitions.

The range of partitions chosen for our experiments lie from [50, 100, ... , 500] in increments of 50. This is because we observed that by default Spark creates 97 partitions for all of the input data (entire 10GB dataset).

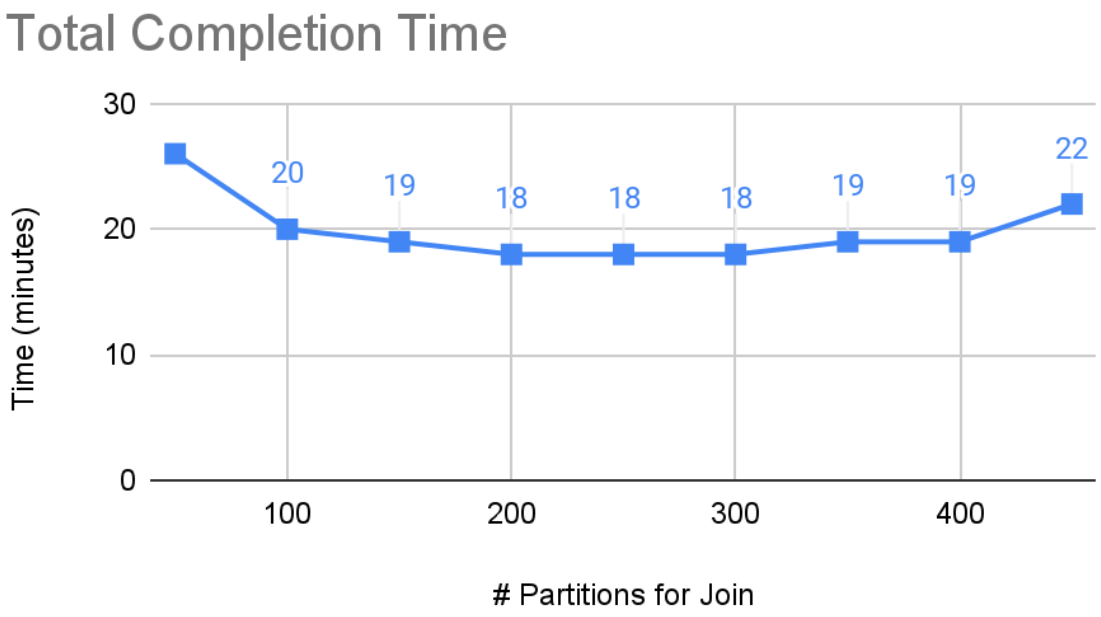


Fig 7: Total Completion Time and runJob Completion Time for PageRank on different number of partitions

To investigate further, we look into the number of tasks created for different partition sizes and the amount of data read and written along the different transformations to complete the PageRank algorithm. As the number of tasks increases (because Spark assigns 1 task for each partition), Spark is able to optimally parallelize these tasks until a certain threshold. This is why when the partitions are smaller (eg. partitions = 50 and partitions = 100), the completion time is slightly more than when the partitions are set to 200-300. Beyond these partition sizes, the number of tasks are too high and the scheduling as well as shuffling overhead outweighs the benefits and does not improve the total completion time.

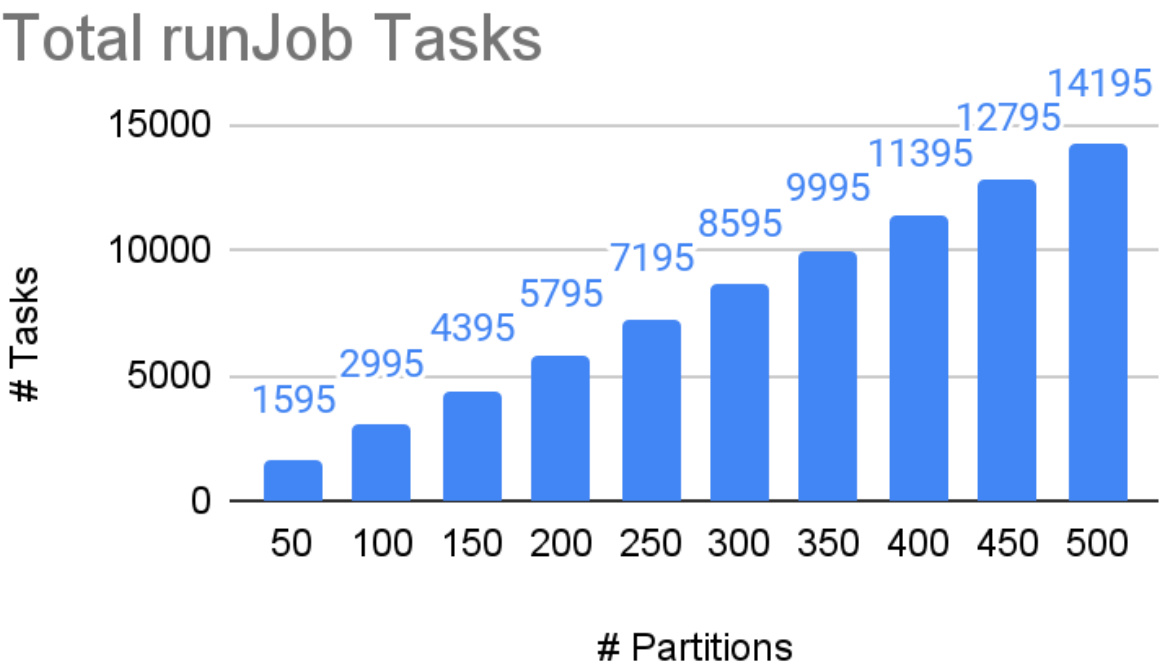


Fig 8: Total runJob Tasks on different number of join partitions

We further analyze the performance of shuffle transformations `groupByKey`, `partitionBy`, `join` and `reduceByKey`. As we use 10 iterations, we calculate the median times for the `join` and `reduceByKey` transformations.

- `groupBy`:
  Since this step is the first shuffle that the RDDs encounter, this value increases very slightly (~5% of median time) as we increase the number of partitions.

- `partitionBy`:
  The partition time increases ever so slightly as we increase the number of partitions as it requires more coordination to create these partitions.

- `join`:
  The join time decreases as we increase the number of partitions as Spark is able to optimally parallelize smaller tasks containing lesser data per partition. However, we see that on continuously increasing the number of partitions, this number trends upwards again.

- `reduceByKey`:
  Similar to the `join` transformation, the `reduceByKey` time decreases as we increase the number of partitions as it needs to work with smaller amounts of data.

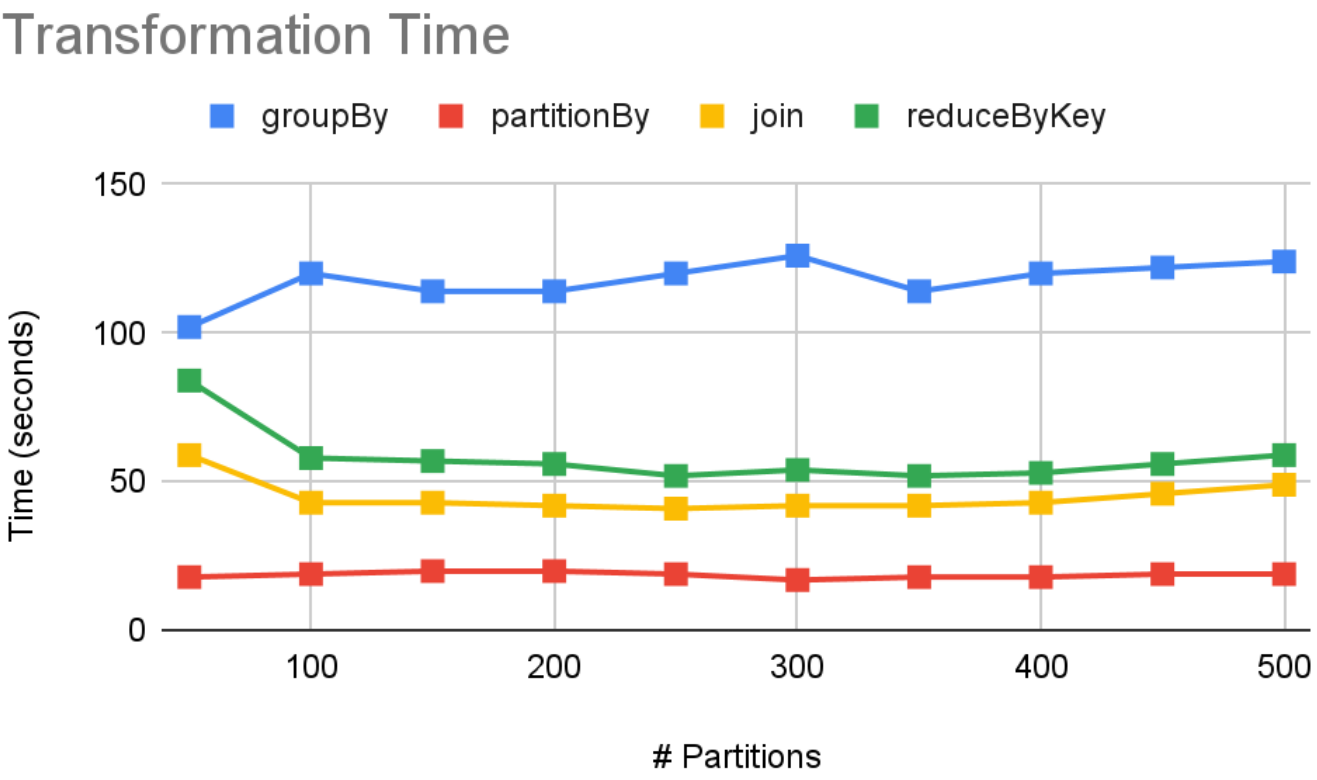


Fig 9: Time taken by the different transformations on different number of join partitions

We drill down to the amount of data read and written across the different shuffle transformations for different partition sizes. We analyze the transformation `groupBy`, `partitionBy`, `join` and `reduceByKey`.

- `groupBy` Write :
  The value remains unchanged because this shuffle causes a hash partitioning to occur and the entire contents of the `neighboursRDD` is written to disk.

- `partitionBy` Read:
  This remains unchanged as the partitioning data from the `groupByKey` must read all the data to create the partitions.

- `partitionBy` Write:
  This remains unchanged as the partitioning must write all the data to create the partitions.

- `join` Read:

The join reads almost double the amount of data wrt to `partitionBy` Write. This is because to execute a join algorithm, it needs to scan the partition (or precomputes this) and then finally joins a record when it finds a matching key.

- `join` Write:
  This remains unchanged as the number of partitions does not change the result of the join.

- `reduceByKey` Read:
  `reduceByKey` reads the output of the flatMap which is constant across iterations and so, the number of partitions does not change the amount of data read across different partition sizes.

- `reduceByKey` Write:
  This value keeps increasing as we increase the partitions. This is because reduceByKey first reduces the operation at the partition level before broadcasting it across the network. If the number of partitions increase, then the number of intermediate states also increase and more data has to be shuffled.

We show the min, max times and read/write shuffle data for the `join` and `reduceByKey` transformations as we had previously used the median to compare the results for the other transformations.

Time Statistics for Join and reduceByKey Transformations				
# Partitions	**Min join Time (seconds)**	**Max join Time (seconds)**	**Min reduceByKey Time (seconds)**	**Max reduceByKey Time (seconds)**
50	58	66	84	132
100	43	49	58	78
150	43	51	55	78
200	42	49	55	78
250	41	48	52	78
300	42	48	53	96
350	42	50	51	108
400	43	49	51	114
450	44	52	52	120
500	45	53	52	124

Fig 10: Time Statistics for the join and reduceByKey transformations on different partitions

R/W Data Statistics for Join and reduceByKey Transformations								
# Partitions	Min join Read (GB)	Max join Read (GB)	Min reduceByKey Read (GB)	Max reduceByKey Read (GB)	Min join Write (GB)	Max join Write (GB)	Min reduceByKey Write (GB)	Max reduceByKey Write (GB)
50	4.6	5.4	3.4	5.9	3.4	3.5	1.7	2.5
100	5	5.8	3.4	5.9	3.4	3.5	2.1	2.8
150	5.3	6.1	3.4	5.9	3.4	3.5	2.2	3.1
200	5.4	6.3	3.4	5.9	3.4	3.5	2.3	3.1
250	5.3	6.2	3.4	5.9	3.4	3.5	2.4	3.3
300	5.2	6.1	3.4	5.9	3.4	3.5	2.5	3.4
350	5.4	6.4	3.4	5.9	3.4	3.5	2.5	3.5
400	5.5	6.5	3.4	5.9	3.4	3.5	2.6	3.5
450	5.5	6.6	3.4	5.9	3.4	3.5	2.6	3.5
500	5.6	6.8	3.4	5.9	3.4	3.5	2.7	3.6

Fig 11: Read and Write Data Statistics for the join and reduceByKey transformations on different partitions

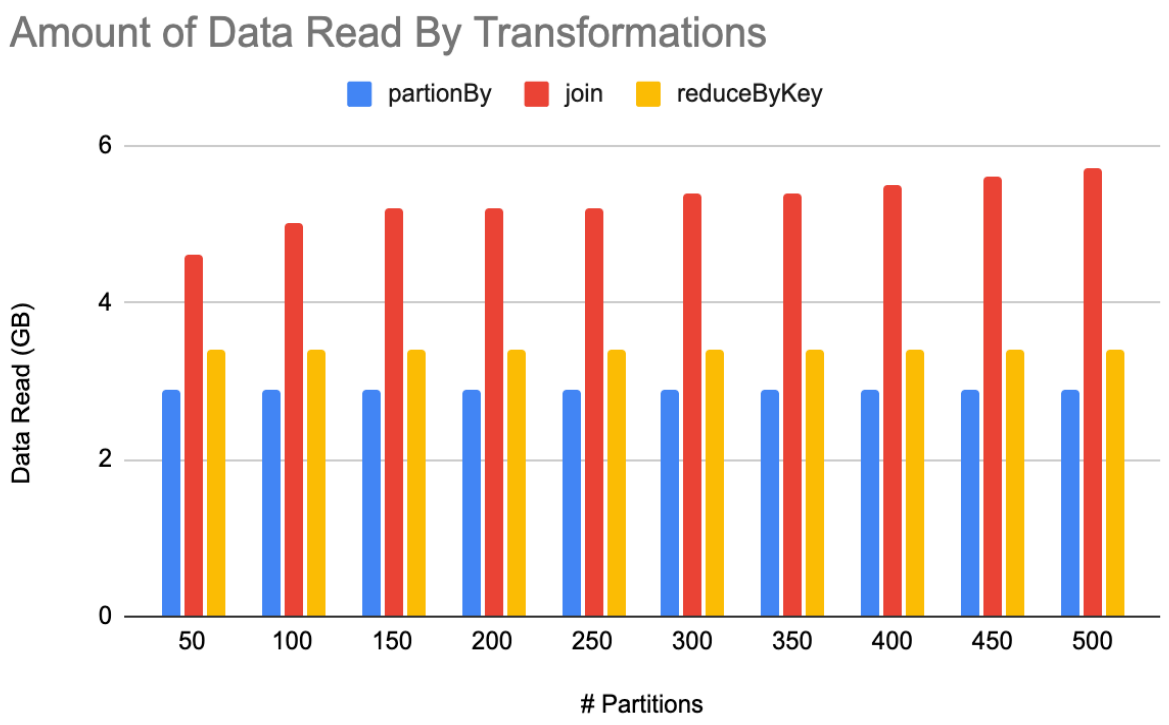


Fig 12: Amount of data read by the different transformations on different number of join partitions.

Fig 13: Amount of data written by the different transformations on different number of join partitions.

Interesting Observations

Custom Partitioning:
We decided against using a custom partitioning algorithm because after inspection, we found that Spark's default hash partitioning was resulting in an almost uniform distribution. The figure below shows the statistics of elements present in each partition when using the default partitioning scheme.

```
Characteristics about data partitioned by the default hash partitioner of Spark:
Number of partitions: 97
Maximum number of elements in one partition: 12072
Minimum number of elements in one partition: 11445
Mean number of elements in one partition: 11698.670103092783
Median number of elements in one partition: 11703
```

Fig 14: Statistics of elements present in the default partitioning scheme

PySpark Join:
We notice that the number of partitions doubles when we perform the join transformation. Peeking into the DAG, we see that the PySpark `join` operation performs a Union (as described in this pull request) operation which causes the number of partitions to become 2x (as shown in Figure 16 where the number of tasks were 50 and then increased to 100). Due to the iterative nature of our algorithm, the number of partitions kept on increasing due to this repetitive join and it significantly hurt performance (see Figure 15 for reference). Hence, we started capping the join to maintain the same number of partitions as in the beginning and we saw a huge speedup in the completion time **(19 minutes vs 25 minutes)**.

Stage Id ▾	Description		Submitted	Duration	Tasks: Succeeded/Total	Input
20	runJob at PythonRDD.scala:166	+details	2022/09/24 18:18:26	0.4 s	1/1	
19	reduceByKey at /users/dgoenka/assignment1/part2/pagerank.py:43	+details	2022/09/24 18:16:52	1.6 min	970/970	
18	join at /users/dgoenka/assignment1/part2/pagerank.py:41	+details	2022/09/24 18:15:34	1.3 min	970/970	
17	reduceByKey at /users/dgoenka/assignment1/part2/pagerank.py:43	+details	2022/09/24 18:14:05	1.5 min	873/873	
16	join at /users/dgoenka/assignment1/part2/pagerank.py:41	+details	2022/09/24 18:12:50	1.3 min	873/873	
15	reduceByKey at /users/dgoenka/assignment1/part2/pagerank.py:43	+details	2022/09/24 18:11:27	1.4 min	776/776	
14	join at /users/dgoenka/assignment1/part2/pagerank.py:41	+details	2022/09/24 18:10:14	1.2 min	776/776	
13	reduceByKey at /users/dgoenka/assignment1/part2/pagerank.py:43	+details	2022/09/24 18:08:54	1.3 min	679/679	
12	join at /users/dgoenka/assignment1/part2/pagerank.py:41	+details	2022/09/24 18:07:43	1.2 min	679/679	
11	reduceByKey at /users/dgoenka/assignment1/part2/pagerank.py:43	+details	2022/09/24 18:06:23	1.3 min	582/582	
10	join at /users/dgoenka/assignment1/part2/pagerank.py:41	+details	2022/09/24 18:05:15	1.1 min	582/582	
9	reduceByKey at /users/dgoenka/assignment1/part2/pagerank.py:43	+details	2022/09/24 18:03:57	1.3 min	485/485	
8	join at /users/dgoenka/assignment1/part2/pagerank.py:41	+details	2022/09/24 18:02:49	1.1 min	485/485	
7	reduceByKey at /users/dgoenka/assignment1/part2/pagerank.py:43	+details	2022/09/24 18:01:32	1.3 min	388/388	
6	join at /users/dgoenka/assignment1/part2/pagerank.py:41	+details	2022/09/24 18:00:26	1.1 min	388/388	
5	reduceByKey at /users/dgoenka/assignment1/part2/pagerank.py:43	+details	2022/09/24 17:59:08	1.3 min	291/291	
4	join at /users/dgoenka/assignment1/part2/pagerank.py:41	+details	2022/09/24 17:58:00	1.1 min	291/291	
3	reduceByKey at /users/dgoenka/assignment1/part2/pagerank.py:43	+details	2022/09/24 17:56:31	1.5 min	194/194	
2	join at /users/dgoenka/assignment1/part2/pagerank.py:41	+details	2022/09/24 17:55:12	1.3 min	194/194	
1	reduceByKey at /users/dgoenka/assignment1/part2/pagerank.py:43	+details	2022/09/24 17:53:02	2.2 min	97/97	
0	groupByKey at /users/dgoenka/assignment1/part2/pagerank.py:29	+details	2022/09/24 17:50:47	2.3 min	97/97	9.9 GiB

Fig 15: The number of tasks keep on increasing without capping the partitions on a join

3	join at /users/dgoenka/assignment1/part3/pagerank.py:76	+details	2022/09/26 13:28:46	1.1 min	100/100	
2	reduceByKey at /users/dgoenka/assignment1/part3/pagerank.py:84	+details	2022/09/26 13:26:36	2.2 min	50/50	

Fig 16: The number of tasks double as with the join transformation

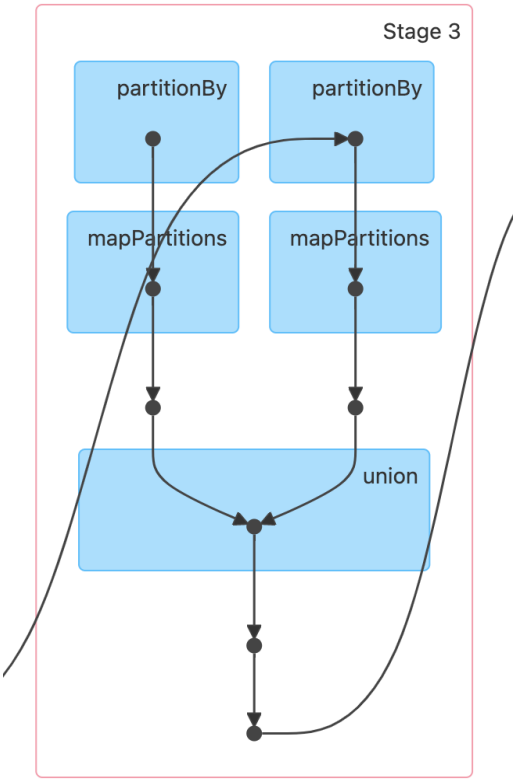


Fig 17: Join transformation

First shuffle skipped:
Additionally, to perform 10 iterations of the PageRank algorithm, we see that there are 9 Join stages and 10 reduceByKey stages. We expected that there would be 10 stages for each of these transformations. The reason why we have one less join stage is because the `ranksDD` is derived from the `neighboursRDD` when being initialized. Therefore, when encountering the first shuffle operation which is a join, these co-located partitions are not joined at all.

We also partition the data when writing to the output file. The graph below shows the time taken to write the output of the PageRank algorithm to these partitions on disk.

From all our above experiments, we conclude that the cluster achieves the best results with the partition size set to 200. Hence, for all our further experiments we use this constant value of 200 partitions.

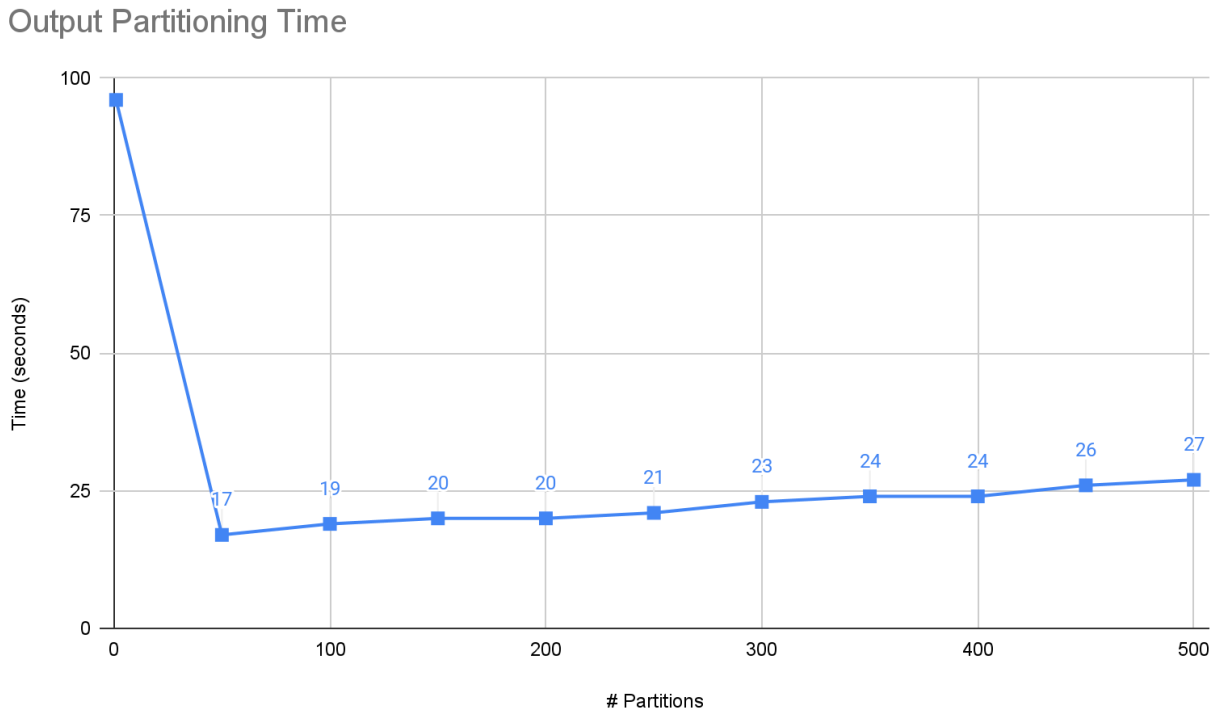


Fig 18: Time taken to write to the output partitions (Note: high value is for 1 partition - Spark has to collect ALL data on one node and then write to disk)

Task 3: Persistence

We test for *Memory Only* and *Memory and Disk* persistence levels in Spark and compare the results to when we do not use persistence. We expect *Memory Only* to outperform the other persistence levels. We note that since the available memory is sufficiently large (30GB per executor) in comparison to the dataset, we expect similar performance between the two persistence modes.

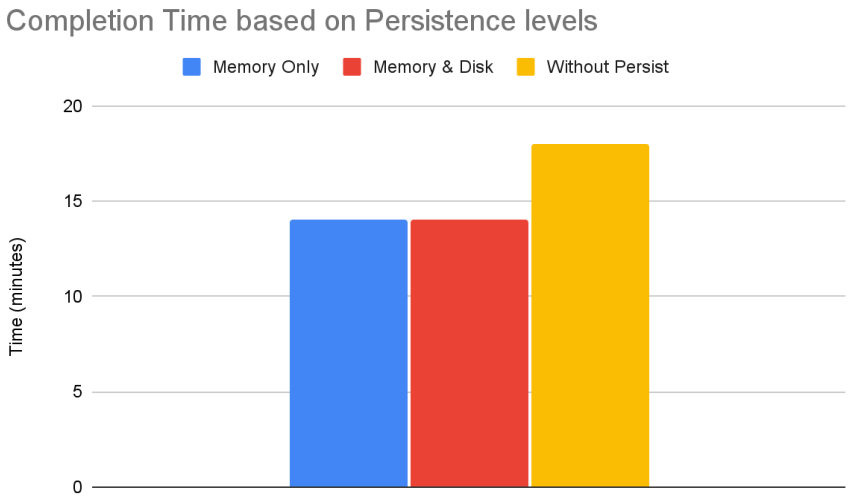


Fig 19: PageRank completions time on changing the RDD persistence levels

Unsurprisingly, from the results above, we observe that indeed *Memory Only* and the *Memory and Disk* persistence modes both outperform the case when we do not persist the RDD. If we further take a look at the execution timeline (see Figure 20 below), we see that the number of shuffle stages has halved (there are no more joins present) and total completion time also reduces by 50%. This is because the `neighboursRDD` is persisted in-memory and resides on the same partition that the `reduceByKey` results in.

Stage Id ▾	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
12	runJob at PythonRDD.scala:166	+details	2022/09/26 18:08:30	0.4 s	1/1			11.4 MiB	
11	reduceByKey at /users/dgoenka/assignment1/part3/pagerank.py:86	+details	2022/09/26 18:07:20	1.2 min	200/200	2.9 GiB		2.3 GiB	2.3 GiB
10	reduceByKey at /users/dgoenka/assignment1/part3/pagerank.py:86	+details	2022/09/26 18:06:11	1.2 min	200/200	2.9 GiB		2.3 GiB	2.3 GiB
9	reduceByKey at /users/dgoenka/assignment1/part3/pagerank.py:86	+details	2022/09/26 18:05:01	1.2 min	200/200	2.9 GiB		2.3 GiB	2.3 GiB
8	reduceByKey at /users/dgoenka/assignment1/part3/pagerank.py:86	+details	2022/09/26 18:03:51	1.2 min	200/200	2.9 GiB		2.3 GiB	2.3 GiB
7	reduceByKey at /users/dgoenka/assignment1/part3/pagerank.py:86	+details	2022/09/26 18:02:40	1.2 min	200/200	2.9 GiB		2.3 GiB	2.3 GiB
6	reduceByKey at /users/dgoenka/assignment1/part3/pagerank.py:86	+details	2022/09/26 18:01:30	1.2 min	200/200	2.9 GiB		2.3 GiB	2.3 GiB
5	reduceByKey at /users/dgoenka/assignment1/part3/pagerank.py:86	+details	2022/09/26 18:00:20	1.2 min	200/200	2.9 GiB		2.3 GiB	2.3 GiB
4	reduceByKey at /users/dgoenka/assignment1/part3/pagerank.py:86	+details	2022/09/26 17:59:08	1.2 min	200/200	2.9 GiB		2.4 GiB	2.3 GiB
3	reduceByKey at /users/dgoenka/assignment1/part3/pagerank.py:86	+details	2022/09/26 17:57:50	1.3 min	200/200	2.9 GiB		3.1 GiB	2.4 GiB
2	reduceByKey at /users/dgoenka/assignment1/part3/pagerank.py:86	+details	2022/09/26 17:56:31	1.3 min	200/200	2.9 GiB		2.9 GiB	3.1 GiB
1	partitionBy at /users/dgoenka/assignment1/part3/pagerank.py:55	+details	2022/09/26 17:56:10	21 s	97/97			2.9 GiB	2.9 GiB
0	groupByKey at /users/dgoenka/assignment1/part3/pagerank.py:50	+details	2022/09/26 17:54:36	1.6 min	97/97	9.9 GiB			2.9 GiB

Fig 20: Reduced number of stages when using persistence in Spark

We also capture system-level metrics using **Ganglia** at each node to further evaluate the memory consumption. We see (from Figure 21 below) that the memory usage is higher for the case when we persist in-memory vs when we do not. The CPU utilization % however, does not change. However, the stark difference can be seen in the network I/O graph where we observe a drastic reduction in network bandwidth (both in/out) due to the absence of the `join` stage which required network communication.

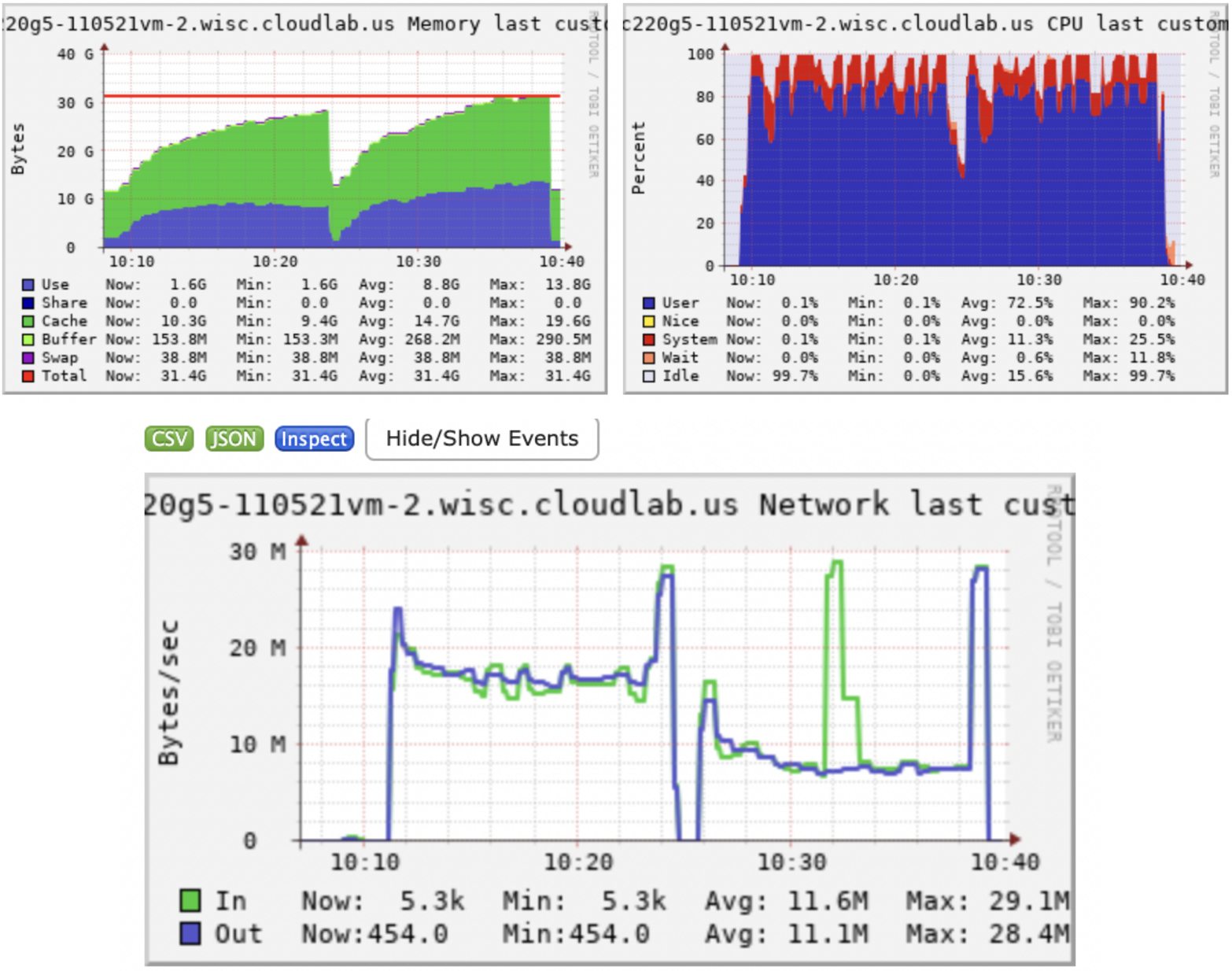


Fig 21: Memory Usage (Top Left), CPU Util % (Top Right), Network I/O (Bottom) for two back to back runs of PageRank. The first run (from 10:10 AM to 10:25 AM) was without persistence mode. The second run (from 10:26 AM to 10:40 AM) was with *Memory Only* persistence mode.

Task 4. Kill a Worker process when the application reaches 25% and 75% of its lifetime

We compare the completion times of killing a single worker process when the application reaches 25% and 75% with the completion time of a job that ran successfully. We use 200

partitions and the *Memory Only* scheme to collect the results due to this being the fastest configuration. We notice that it takes longer for the application to complete when we kill it at 75%, because it needs to recreate the state from Stage 0 to Stage 9 (`groupBy`, `partitionBy`, `reduceByKey x 8`)

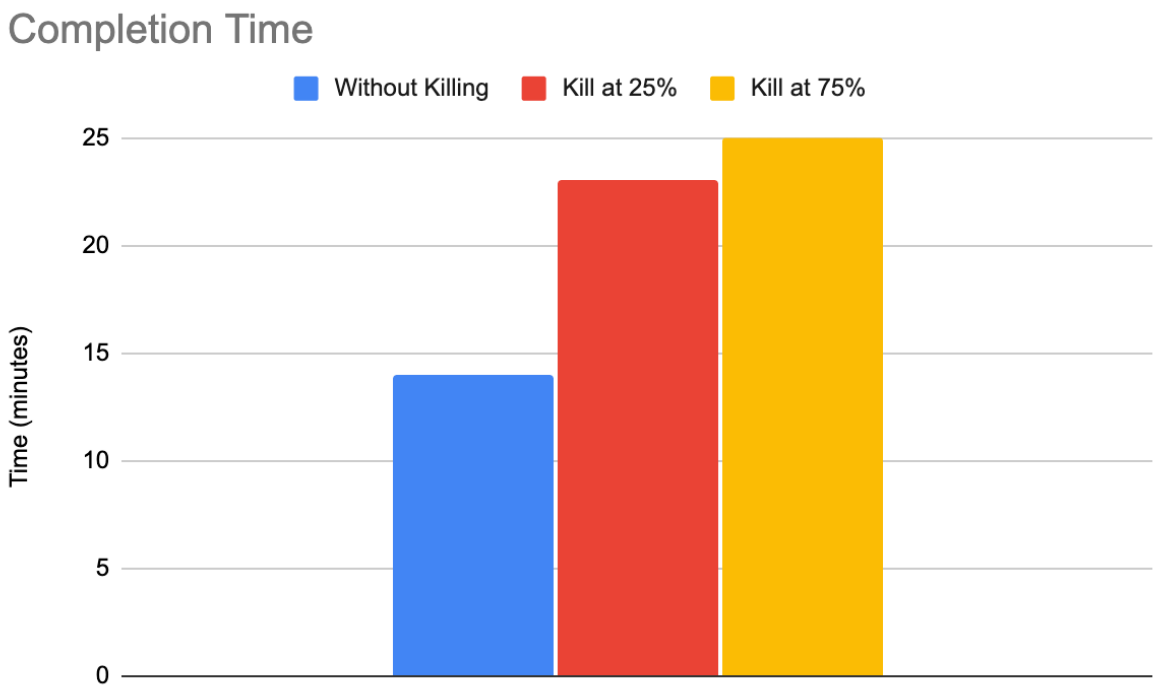


Fig 22: Completion time on killing a worker at different stages of the PageRank algorithm execution

We also capture the system metrics using Ganglia to further evaluate the behavior of the cluster during this operation. We run two jobs back to back and kill Worker 2 at 75% of completion in the first job and kill Worker 2 at 25% of completion of the second job. In the first figure below (Figure 23), we present the Memory and CPU usage of Worker 2 respectively. However, more interesting is to look at the behavior of Worker 1 during the entire lifetime of both these jobs.

We notice that even though Spark has to recompute the operations on partitions that were present on Worker 2 now on Workers 1 and 0, there is no significant impact to the overall Memory or CPU characteristics of Workers 1 or 0 due to this. We reason that Spark throws away the current operations at hand and starts re-computing the values lost in the partition that Worker 2 possessed.

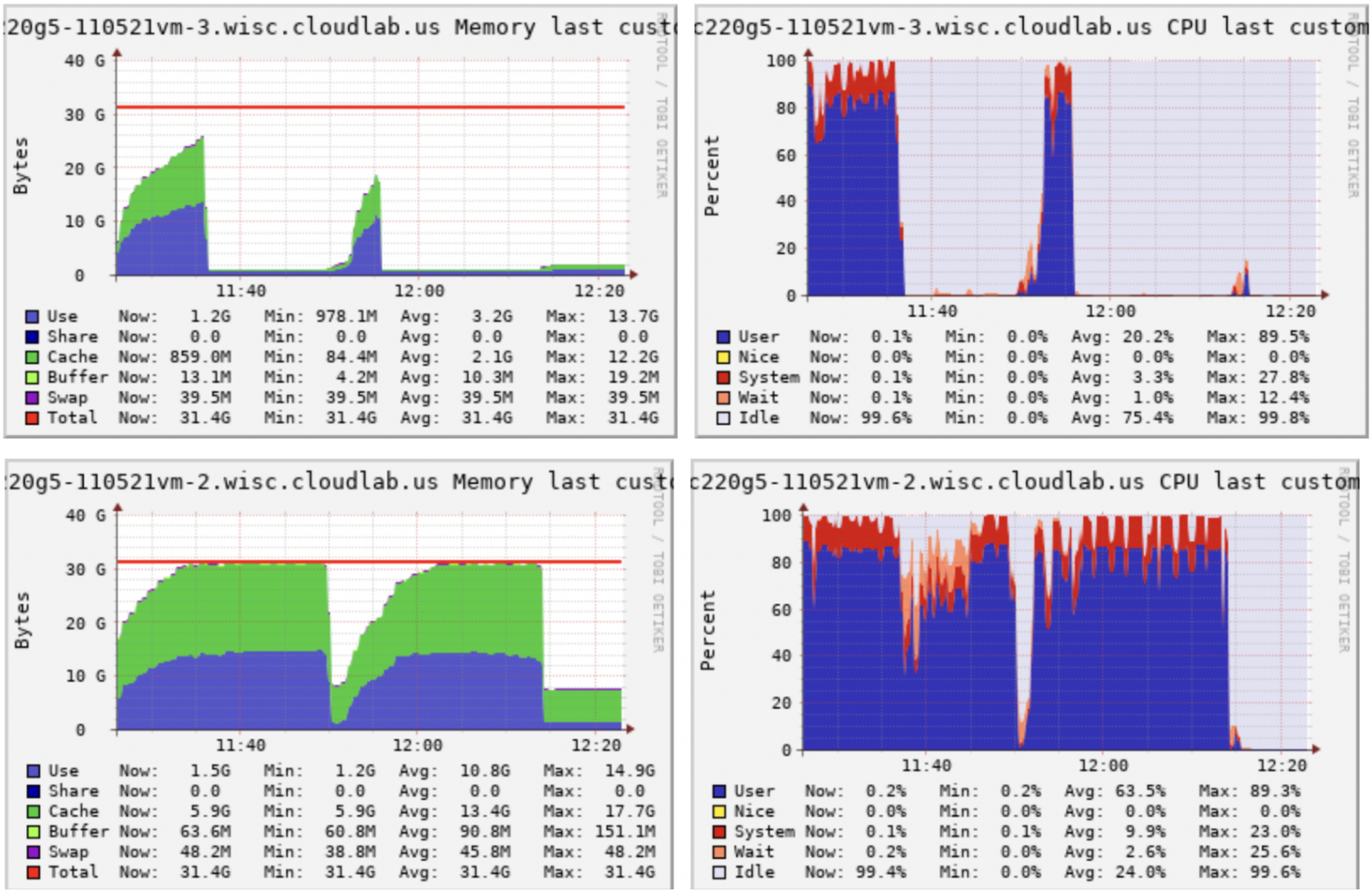


Fig 23: Worker 2 Memory Usage. (Top-Left), CPU Util (Top-Right).
Worker 1 Memory Usage (Bottom-Left), CPU Util (Bottom Right).
The first job submitted was killed at 75% and the second one at 25% completion time.

Moreover, we also confirm that Spark only re-computes the missing partition values by looking at the executing summary in the figure below.

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
12	runJob at PythonRDD.scala:166	+details	2022/09/27 10:49:16	0.4 s	1/1			11.4 MiB	
11	reduceByKey at /users/dgoenka/assignment1/part3/pagerank.py:109	+details	2022/09/27 10:47:29	1.8 min	200/200	2.9 GiB		2.3 GiB	2.3 GiB
10	reduceByKey at /users/dgoenka/assignment1/part3/pagerank.py:109	+details	2022/09/27 10:45:42	1.8 min	200/200	2.9 GiB		2.3 GiB	2.3 GiB
9 (retry 1)	reduceByKey at /users/dgoenka/assignment1/part3/pagerank.py:109	+details	2022/09/27 10:44:01	1.7 min	160/160	2.3 GiB		1857.4 MiB	1854.6 MiB
8 (retry 1)	reduceByKey at /users/dgoenka/assignment1/part3/pagerank.py:109	+details	2022/09/27 10:43:09	53 s	72/72	1074.8 MiB		835.2 MiB	831.4 MiB
7 (retry 1)	reduceByKey at /users/dgoenka/assignment1/part3/pagerank.py:109	+details	2022/09/27 10:42:16	52 s	70/70	1044.8 MiB		812.7 MiB	808.6 MiB
6 (retry 1)	reduceByKey at /users/dgoenka/assignment1/part3/pagerank.py:109	+details	2022/09/27 10:41:26	51 s	71/71	1059.7 MiB		824.8 MiB	820.2 MiB
5 (retry 1)	reduceByKey at /users/dgoenka/assignment1/part3/pagerank.py:109	+details	2022/09/27 10:40:29	57 s	71/71	1059.6 MiB		831.8 MiB	821.4 MiB
4 (retry 1)	reduceByKey at /users/dgoenka/assignment1/part3/pagerank.py:109	+details	2022/09/27 10:39:36	53 s	71/71	1060.5 MiB		869.7 MiB	828.3 MiB
3 (retry 1)	reduceByKey at /users/dgoenka/assignment1/part3/pagerank.py:109	+details	2022/09/27 10:38:26	1.2 min	70/70	1044.7 MiB		1125.2 MiB	853.8 MiB
2 (retry 1)	reduceByKey at /users/dgoenka/assignment1/part3/pagerank.py:109	+details	2022/09/27 10:37:30	56 s	68/68	1015.2 MiB		1016.5 MiB	1090.4 MiB
1 (retry 1)	partitionBy at /users/dgoenka/assignment1/part3/pagerank.py:78	+details	2022/09/27 10:36:57	32 s	35/35			1082.0 MiB	1088.5 MiB
0 (retry 1)	groupByKey at /users/dgoenka/assignment1/part3/pagerank.py:73	+details	2022/09/27 10:36:06	51 s	33/33	3.4 GiB			1041.0 MiB

Fig 24: Re-Execution pipeline when the job is killed at 75% completion time. We see that the input to the reduceByKey operation is ⅓ of the total input without killing the worker -– only the missing partition is recomputed.

Contributions

We all contributed equally in all aspects of the assignment.

Conclusion

The experiments conducted by us on this large dataset demonstrate different aspects of the Spark computation engine and answers questions on its performance, persistence and fault tolerance.

One of the biggest performance gains that we noticed was from optimizing the joins in Spark and in doing so, requiring less data to be shuffled around. Moreover, the persistence mode really speeds up iterative algorithms like PageRank which require re-use of the same intermediate data.