Task 1

Implementing Page Rank Algorithm using PySpark and Hadoop HDFS.

PseudoCode:

```
//Clean data and split it into tuples of source and destination
Var data_Rdd = spark.textFile(….).map(….)
//Converting (Source, [Destination]) into (Source, [Destination], Destination2…])
Var links_Rdd = data_Rdd.reduceByKey(val1, val2 => val1+val2)
//Initializing source ranks with 1
Var ranks_Rdd = links_Rdd.map(data => (data[0],1))
for (i =1 to Iterations) {
// Converting (Source, [Destination], Destination2…]) into (Source, [Destination1, Destination2…], SourceRank) by joining links_Rdd and ranks_Rdd and performing flat map to get contributions
        Contributions_Rdd = links_Rdd.join(ranks_Rdd).flatMap(sourceDestRankData => {
                sourceDestRankData[1].map( data => {
                        (data[0], 0.85 *sourceDestRankData[2]/sourceDestRankData.size)
                })
        }
//Updating ranks_Rdd with latest contributions
        ranks_rdd = Contributions_Rdd.reduceByKey(val1, val2 => val1+val2)
}
```

The above pseudo code was implemented in python v3.7.13 and was executed by **3 Spark workers** with **HDFS as underlying file system**. **Total number of Iterations** for Task 1 was set to be **10**. DAG created by spark can be referred in figure 1 where there were **23 stages of execution** in total. Step 3 and Step 4 corresponds to a total of 20 stages, since these are repeated for 10 iterations

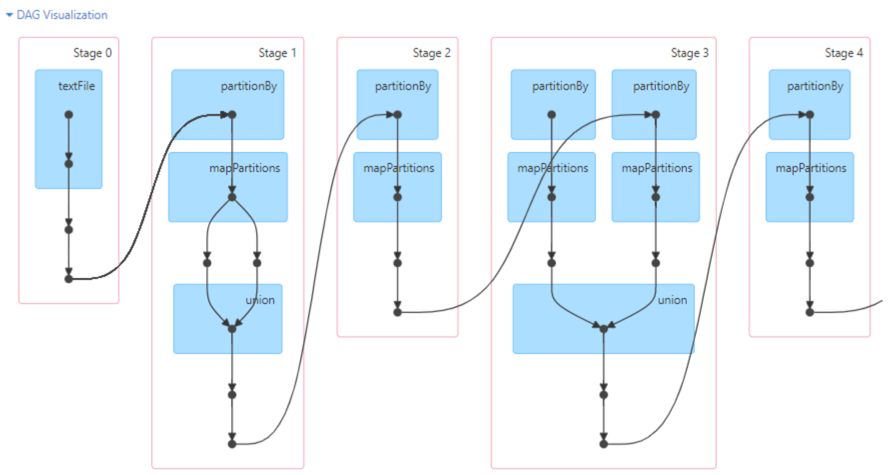


Figure 1 DAG Visualization

Learnings from DAG:

Each wide operation such as reduceByKey, groupByKey, join etc results in a data shuffle and map which can be seen in DAG (with partitionBy and mapPartitions operations).

Stage 0: Corresponds to the data read from text files along with the data cleaning and pre-processing (I.e., Splitting each line into tuples of (Source, Destination))

Stage 1: Correspond to the creation of "*links_Rdd*" using reduceByKey on pre-processed data. Since reduceByKey is a wide operation, we see partitionBy and MapPartition operations in stage 1.

Stage 2: Since "*ranks_Rdd*" is eventually a result of reduceBykey operation on "*contributions_Rdd"* we see partitionBy and MapPartitions operations happening again.

Stage 3: Corresponds to Join operation on "*links_Rdd*" and "*ranks_Rdd*".

Stage 4: Updating "*ranks_Rdd*" from the "*contributions_Rdd*" using reduceByKey operation.

Results from Task 1

- The Total execution time of Task 1 on enwiki-page-articles dataset for 10 iterations and 3 Spark workers was **2995.02 seconds ≈ 50 minutes**

CPU Utilization

- CPU utilization was minimal until 500 seconds of the program execution and this could be explained by the fact that CPU was waiting for disk read operation to complete.

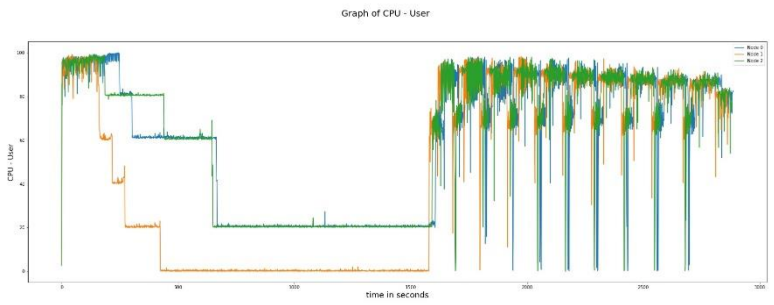


Figure 2 Represents the CPU Utilization by User tasks

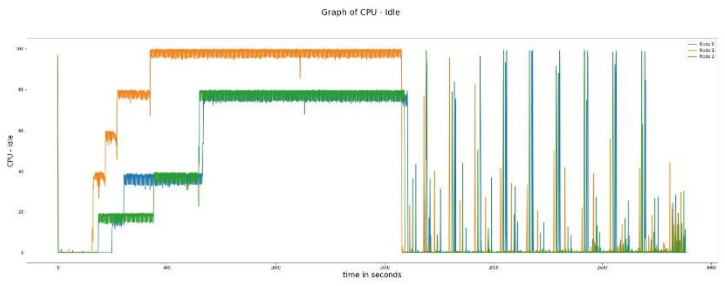


Figure 3 Represents the Idle time of CPU

Disk Utilization

- Disk read spiked to a maximum of 8MB/s for around 500 seconds. Post 500 seconds of program execution, disk reads were minor and this might be corresponding to the shuffle read on the data stored on the disk.
- Disk write started after 1500 seconds of execution and this is caused due to the shuffle writes and storage of intermediate results on the disk.

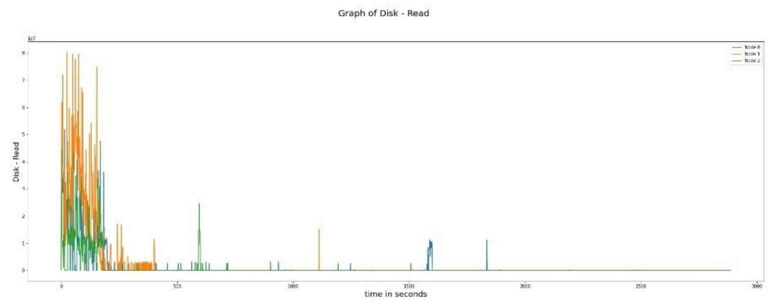


Figure 4 Disk Reads in MB

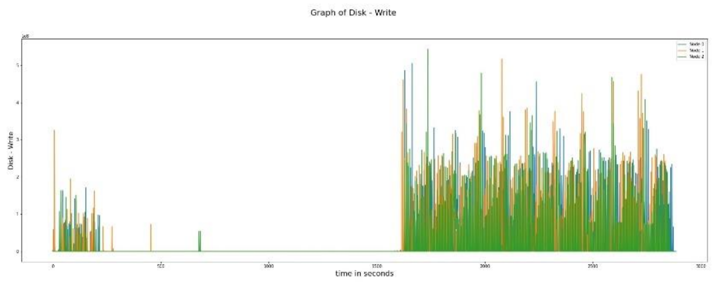


Figure 5 Disk Writes in MB

Network I/O

- Both network reads and writes spiked up close to 1.4MB per sec and this was post 1500 seconds of execution caused due to the wide operation dependency (on reduceByKey, join)

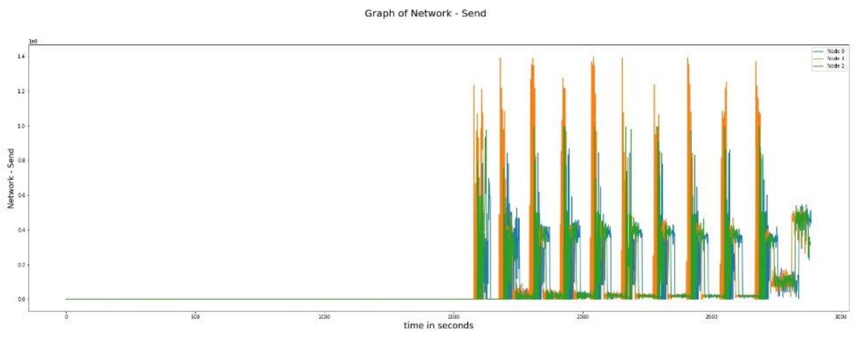


Figure 6 Network writes in MB

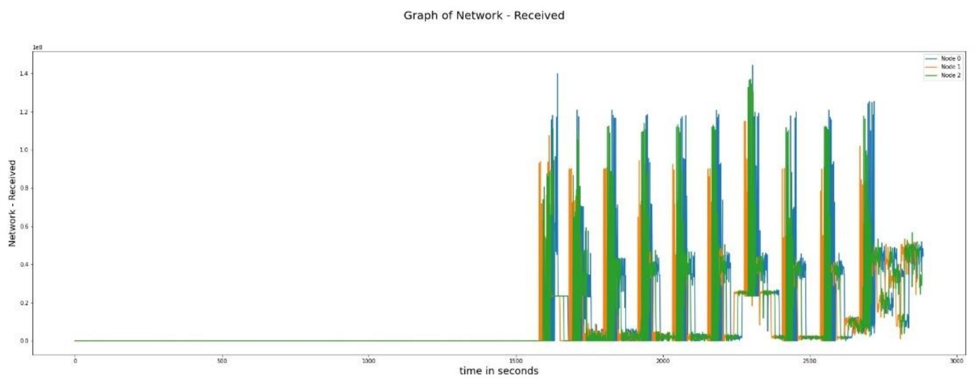


Figure 7 Network Reads in MB

Memory Utilization

- Memory used spanned from 4 GB to 16 GB over the process execution without use of PySpark cache() or persist() functions on RDD.

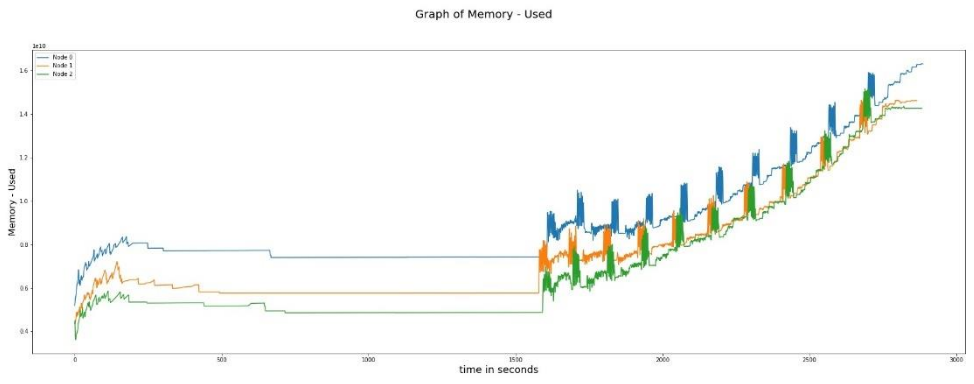


Figure 8 Memory Utilization with 10GB scale

Scope for Optimization in Task 1

- Problem 1: Code uses wide operations such as reduceByKey, Join causing increased network communications, and shuffle reads on disk resulting in poor performance.
    - Approach: Use hash function while performing shuffle operations during reducebyKey or Join operation which might change the nature of wide operation to a narrow operation.
- Problem 2: Number of Partitions of the process have not been fine tuned.
    - Approach: Utilize Number of partitions parameter to improve the performance.

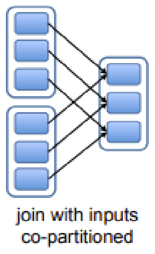


Figure 9 Narrow Operation on Join (Matei Zaharia, 2012)

Task 2

As discussed above in order to optimize we have followed the below steps:

1) Defined custom partitioner which uses python's inbuilt polynomial rolling hash function aiming to reduce network communications and shuffle reads across the nodes
2) Assuming that increase in partitions improve parallelization, we have tested the setup with number of Partitions set as 100 and 500.

Results from Task 2

- The Total execution time of Task 2 on enwiki-page-articles dataset for 10 iterations and 3 Spark workers was **2054.65 seconds ≈ 35 minutes (30% improvement)**

In this section we would continue to discuss the observed performance improvements along with the effects of increasing the number of partitions in the setup.

Performance Improvements:

- When compared with Task 1 the overall network I/O communications
- reduced by ≈ 30 percent for reads (refer figures 7,10 for network reads) and by ≈ 42 percent for writes (refer figures 6,11 for network writes). This effect is caused due to the introduction of custom partitioner during partition creation process that has made the input data as co-partitions.

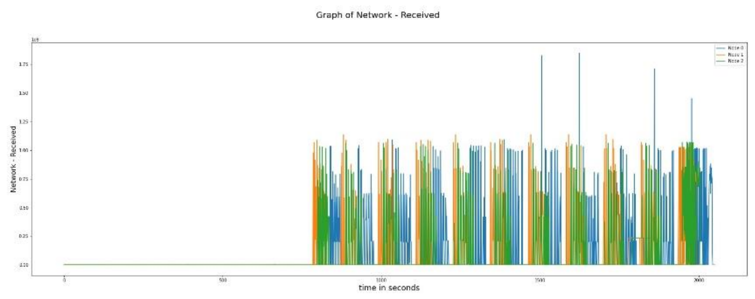


Figure 10 Network reads in MB for Task 2 with 100 Partitions

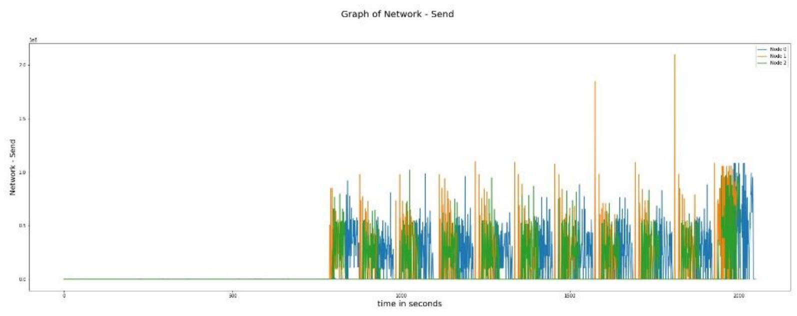
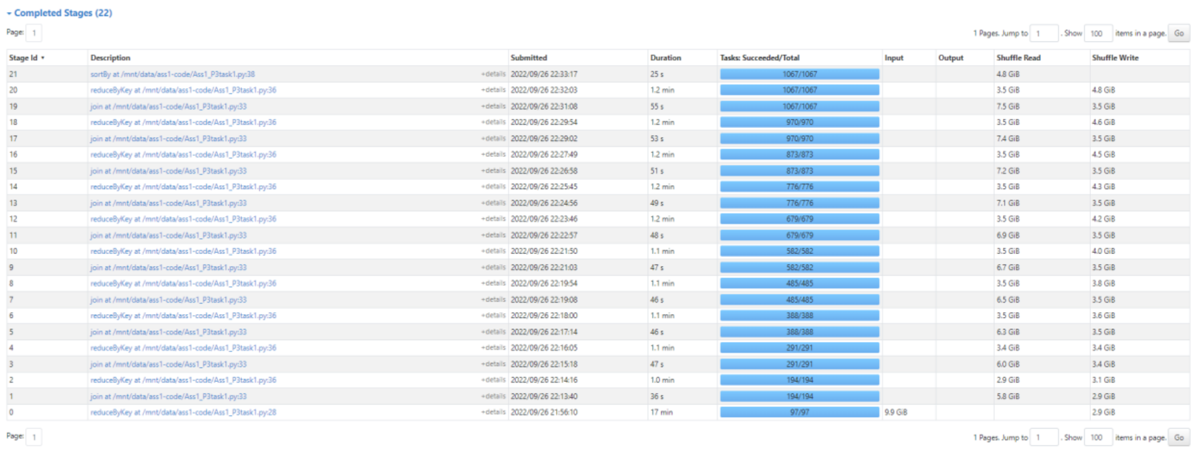


Figure 11 Network writes in MB for Task 2 with 100 partitions

- Similarly, we can observe that the overall shuffle reads and writes across all the stages have also reduced when compared to task 1 (refer shuffle read and write in figures 12,13).



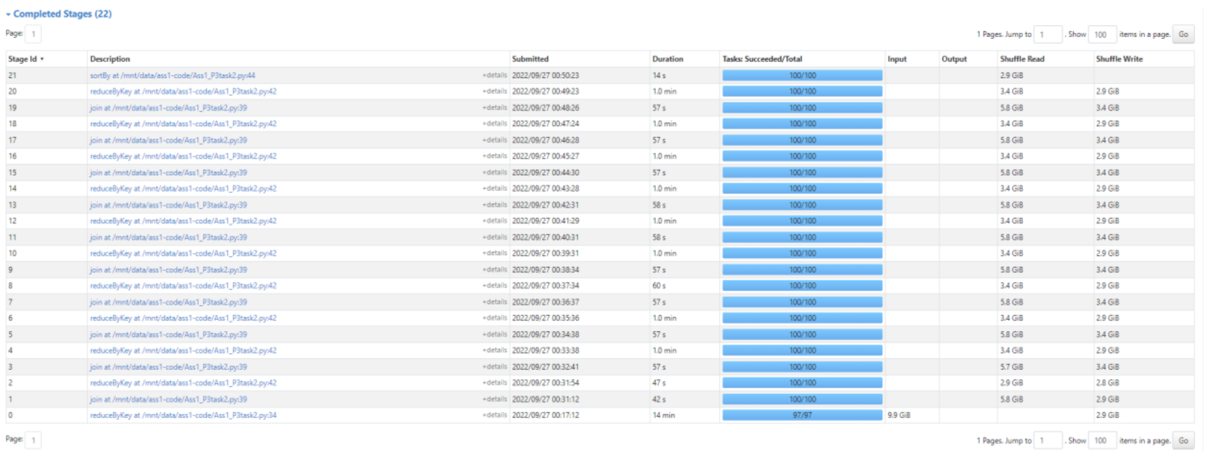
Completed Stages (22)

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

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
21	sortBy at /mnt/data/ass1-code/Ass1_P3task1.py:38 +details	2022/09/26 22:33:17	25 s	1067/1067			4.8 GiB	
20	reduceByKey at /mnt/data/ass1-code/Ass1_P3task1.py:36 +details	2022/09/26 22:32:03	1.2 min	1067/1067			3.5 GiB	4.8 GiB
19	join at /mnt/data/ass1-code/Ass1_P3task1.py:33 +details	2022/09/26 22:31:08	55 s	1067/1067			7.5 GiB	3.5 GiB
18	reduceByKey at /mnt/data/ass1-code/Ass1_P3task1.py:36 +details	2022/09/26 22:29:54	1.2 min	970/970			3.5 GiB	4.6 GiB
17	join at /mnt/data/ass1-code/Ass1_P3task1.py:33 +details	2022/09/26 22:29:02	53 s	970/970			7.4 GiB	3.5 GiB
16	reduceByKey at /mnt/data/ass1-code/Ass1_P3task1.py:36 +details	2022/09/26 22:27:49	1.2 min	873/873			3.5 GiB	4.5 GiB
15	join at /mnt/data/ass1-code/Ass1_P3task1.py:33 +details	2022/09/26 22:26:58	51 s	873/873			7.2 GiB	3.5 GiB
14	reduceByKey at /mnt/data/ass1-code/Ass1_P3task1.py:36 +details	2022/09/26 22:25:45	1.2 min	776/776			3.5 GiB	4.3 GiB
13	join at /mnt/data/ass1-code/Ass1_P3task1.py:33 +details	2022/09/26 22:24:56	49 s	776/776			7.1 GiB	3.5 GiB
12	reduceByKey at /mnt/data/ass1-code/Ass1_P3task1.py:36 +details	2022/09/26 22:23:46	1.2 min	679/679			3.5 GiB	4.2 GiB
11	join at /mnt/data/ass1-code/Ass1_P3task1.py:33 +details	2022/09/26 22:22:57	48 s	679/679			6.9 GiB	3.5 GiB
10	reduceByKey at /mnt/data/ass1-code/Ass1_P3task1.py:36 +details	2022/09/26 22:21:50	1.1 min	582/582			3.5 GiB	4.0 GiB
9	join at /mnt/data/ass1-code/Ass1_P3task1.py:33 +details	2022/09/26 22:21:03	47 s	582/582			6.7 GiB	3.5 GiB
8	reduceByKey at /mnt/data/ass1-code/Ass1_P3task1.py:36 +details	2022/09/26 22:19:54	1.1 min	485/485			3.5 GiB	3.8 GiB
7	join at /mnt/data/ass1-code/Ass1_P3task1.py:33 +details	2022/09/26 22:19:08	46 s	485/485			6.5 GiB	3.5 GiB
6	reduceByKey at /mnt/data/ass1-code/Ass1_P3task1.py:36 +details	2022/09/26 22:18:00	1.1 min	388/388			3.5 GiB	3.6 GiB
5	join at /mnt/data/ass1-code/Ass1_P3task1.py:33 +details	2022/09/26 22:17:14	46 s	388/388			6.3 GiB	3.5 GiB
4	reduceByKey at /mnt/data/ass1-code/Ass1_P3task1.py:36 +details	2022/09/26 22:16:05	1.1 min	291/291			3.4 GiB	3.4 GiB
3	join at /mnt/data/ass1-code/Ass1_P3task1.py:33 +details	2022/09/26 22:15:18	47 s	291/291			6.0 GiB	3.4 GiB
2	reduceByKey at /mnt/data/ass1-code/Ass1_P3task1.py:36 +details	2022/09/26 22:14:16	1.0 min	194/194			2.9 GiB	3.1 GiB
1	join at /mnt/data/ass1-code/Ass1_P3task1.py:33 +details	2022/09/26 22:13:40	36 s	194/194			5.8 GiB	2.9 GiB
0	reduceByKey at /mnt/data/ass1-code/Ass1_P3task1.py:28 +details	2022/09/26 21:56:10	17 min	97/97	9.9 GiB			2.9 GiB

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

Figure 13 Summary of all the Stages involved in Task1



Completed Stages (22)

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

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
21	sortBy at /mnt/data/ass1-code/Ass1_P3task2.py:44 +details	2022/09/27 00:50:23	14 s	100/100			2.9 GiB	
20	reduceByKey at /mnt/data/ass1-code/Ass1_P3task2.py:42 +details	2022/09/27 00:49:23	1.0 min	100/100			3.4 GiB	2.9 GiB
19	join at /mnt/data/ass1-code/Ass1_P3task2.py:39 +details	2022/09/27 00:48:26	57 s	100/100			5.8 GiB	3.4 GiB
18	reduceByKey at /mnt/data/ass1-code/Ass1_P3task2.py:42 +details	2022/09/27 00:47:24	1.0 min	100/100			3.4 GiB	2.9 GiB
17	join at /mnt/data/ass1-code/Ass1_P3task2.py:39 +details	2022/09/27 00:46:28	57 s	100/100			5.8 GiB	3.4 GiB
16	reduceByKey at /mnt/data/ass1-code/Ass1_P3task2.py:42 +details	2022/09/27 00:45:27	1.0 min	100/100			3.4 GiB	2.9 GiB
15	join at /mnt/data/ass1-code/Ass1_P3task2.py:39 +details	2022/09/27 00:44:30	57 s	100/100			5.8 GiB	3.4 GiB
14	reduceByKey at /mnt/data/ass1-code/Ass1_P3task2.py:42 +details	2022/09/27 00:43:28	1.0 min	100/100			3.4 GiB	2.9 GiB
13	join at /mnt/data/ass1-code/Ass1_P3task2.py:39 +details	2022/09/27 00:42:31	58 s	100/100			5.8 GiB	3.4 GiB
12	reduceByKey at /mnt/data/ass1-code/Ass1_P3task2.py:42 +details	2022/09/27 00:41:29	1.0 min	100/100			3.4 GiB	2.9 GiB
11	join at /mnt/data/ass1-code/Ass1_P3task2.py:39 +details	2022/09/27 00:40:31	58 s	100/100			5.8 GiB	3.4 GiB
10	reduceByKey at /mnt/data/ass1-code/Ass1_P3task2.py:42 +details	2022/09/27 00:39:31	1.0 min	100/100			3.4 GiB	2.9 GiB
9	join at /mnt/data/ass1-code/Ass1_P3task2.py:39 +details	2022/09/27 00:38:34	57 s	100/100			5.8 GiB	3.4 GiB
8	reduceByKey at /mnt/data/ass1-code/Ass1_P3task2.py:42 +details	2022/09/27 00:37:34	60 s	100/100			3.4 GiB	2.9 GiB
7	join at /mnt/data/ass1-code/Ass1_P3task2.py:39 +details	2022/09/27 00:36:37	57 s	100/100			5.8 GiB	3.4 GiB
6	reduceByKey at /mnt/data/ass1-code/Ass1_P3task2.py:42 +details	2022/09/27 00:35:36	1.0 min	100/100			3.4 GiB	2.9 GiB
5	join at /mnt/data/ass1-code/Ass1_P3task2.py:39 +details	2022/09/27 00:34:38	57 s	100/100			5.8 GiB	3.4 GiB
4	reduceByKey at /mnt/data/ass1-code/Ass1_P3task2.py:42 +details	2022/09/27 00:33:38	1.0 min	100/100			3.4 GiB	2.9 GiB
3	join at /mnt/data/ass1-code/Ass1_P3task2.py:39 +details	2022/09/27 00:32:41	57 s	100/100			5.7 GiB	3.4 GiB
2	reduceByKey at /mnt/data/ass1-code/Ass1_P3task2.py:42 +details	2022/09/27 00:31:54	47 s	100/100			2.9 GiB	2.8 GiB
1	join at /mnt/data/ass1-code/Ass1_P3task2.py:39 +details	2022/09/27 00:31:12	42 s	100/100			5.8 GiB	2.9 GiB
0	reduceByKey at /mnt/data/ass1-code/Ass1_P3task2.py:34 +details	2022/09/27 00:17:12	14 min	97/97	9.9 GiB			2.9 GiB

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

Figure 12 Summary of all the Stages involves in Task2

Impacts of Increasing Number of Partitions from 100 to 500:

- We have tried to increase the number of partitions from 50 to 100 and then from 100 to 500. During this process we have seen a substantial rise in performance when we increased the number of partitions from 50 to 100 as this resulted in partitions with similar partition size. However, when we increased the partition size from 100 to 500, we have seen increased network communications leading to poor performance (refer Figures 10,11,14,15).

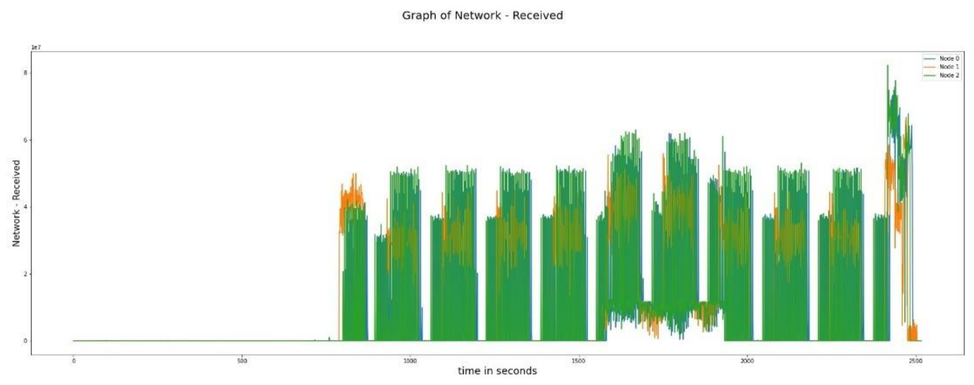


Figure 14 Network Reads for Optimized code with 500 partitions

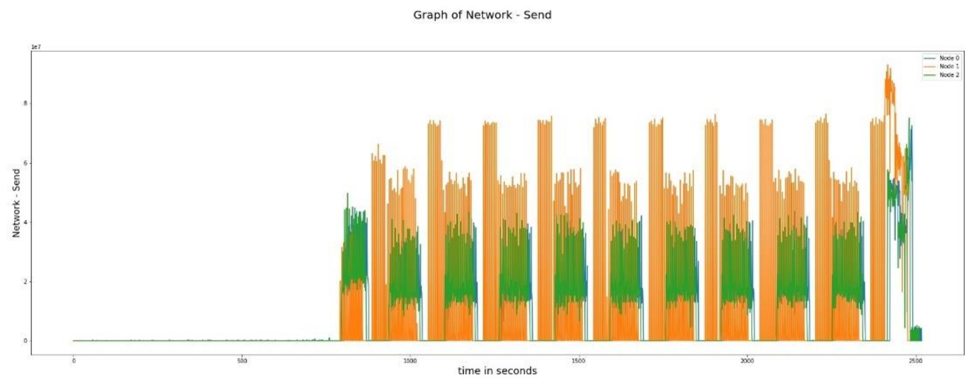


Figure 15 Network Write for Optimized code with 500 partitions

Task 3

We have utilized PySpark's inbuilt cache() function to store the data in-memory rather than writing the data on to disk. We have observed that the memory utilization dramatically increased (starting from 4GB reaching a peak of 20GB) on every worker node when compared to Task-1(started from 3GB and reached closed to 15GB) (refer Figures 8,16) and the overall data written on the disk has marginally decreased (refer Figures 5,17).

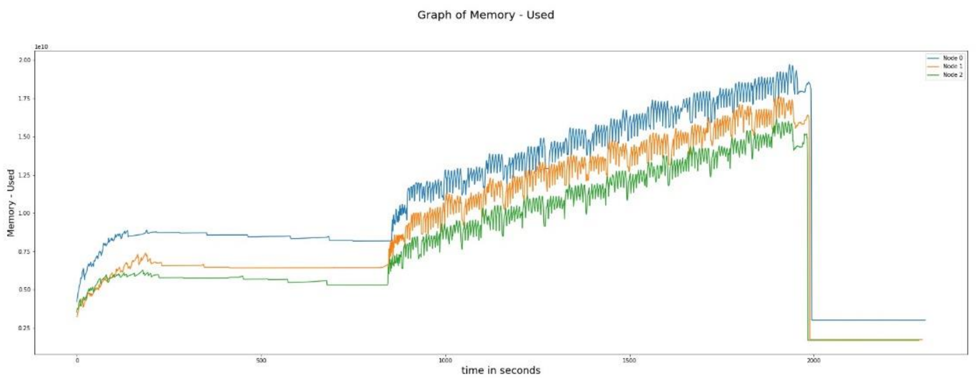


Figure 16 Memory Usage (10GB Scale on y-axis) for Task 3 with 100 partitions

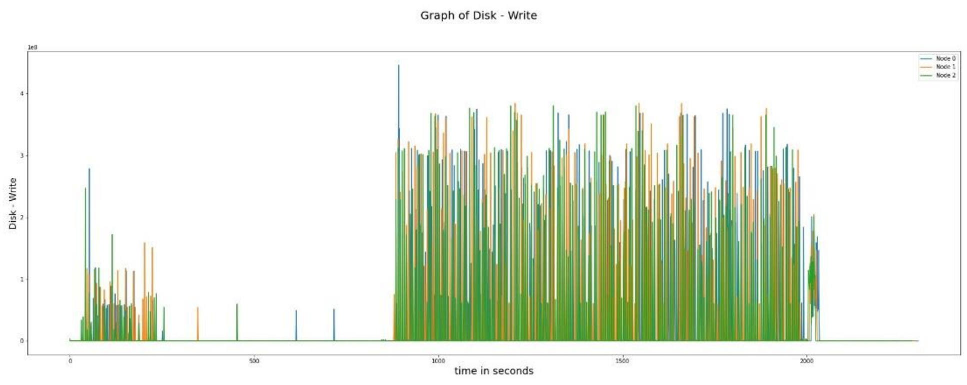


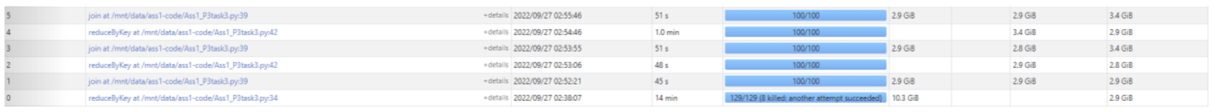
Figure 17 Disk Writes in MB for Task 3 with 100 Partitions

Results from Task 3

- The Total execution time of Task 3 on enwiki-page-articles dataset for 10 iterations and 3 Spark workers was **2001.66 seconds ≈ 33 minutes (6% improvement).**

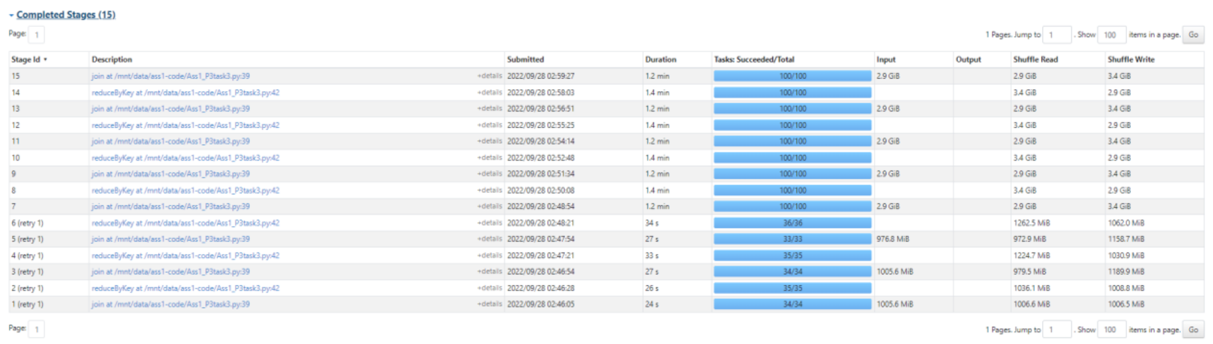
Task 4

- In task 4 we have tried to understand how spark deals with Machine failures. Out of the 3 workers we have killed one of the nodes at 25% completion (after 6 stages were completed) and the other node at 75% completion of the execution (after 16 sages were completed) process.
- As assumed, when a worked is dead, since each RDD has enough information about how it was derived from other datasets (by using lineage to identify the input and transformation function), a worker was triggered to re-compute the results which were saved on previous worker.
- Until the 6th stage with 3 workers, each job was taking around 1min to complete. However, after the 1st worker was killed, Spark Master identified that worker 1 is no longer available and decided to recompute all the data partitions/Intermediated results which required to proceed.
- In our case we identified that all the stages from 0-5 were recomputed (with required data partitions) and the time taken to complete each stage with 2 workers was close to 90 seconds (refer Figure 18,19).
- When the overall execution process reached 75% of competition (i.e., around 16th stage) we have killed 2nd worker, leaving only the master node up and running.
- Similarly, as discussed above, master node identified all the re-computations that are required and to get back the intermediate results which were lost when 2nd worker was killed.
- All the step from 0-15 were recomputed (with appropriate partitions) and since there was only 1 worker available each stage was taking around 110 additional seconds to complete.



5	join at /mnt/data/ass1-code/Ass1_P3task3.py:39 +details	2022/09/27 02:55:46	51 s	100/100	2.9 GiB		2.9 GiB	3.4 GiB
4	reduceByKey at /mnt/data/ass1-code/Ass1_P3task3.py:42 +details	2022/09/27 02:54:46	1.0 min	100/100			3.4 GiB	2.9 GiB
3	join at /mnt/data/ass1-code/Ass1_P3task3.py:39 +details	2022/09/27 02:53:55	51 s	100/100	2.9 GiB		2.8 GiB	3.4 GiB
2	reduceByKey at /mnt/data/ass1-code/Ass1_P3task3.py:42 +details	2022/09/27 02:53:06	48 s	100/100			2.9 GiB	2.8 GiB
1	join at /mnt/data/ass1-code/Ass1_P3task3.py:39 +details	2022/09/27 02:52:21	45 s	100/100	2.9 GiB		2.9 GiB	2.9 GiB
0	reduceByKey at /mnt/data/ass1-code/Ass1_P3task3.py:34 +details	2022/09/27 02:38:07	14 min	129/129 (8 killed: another attempt succeeded)	10.3 GiB			2.9 GiB

Figure 18 Completed Stages summary before worker was killed



Completed Stages (15)

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

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
15	join at /mnt/data/ass1-code/Ass1_P3task3.py:39 +details	2022/09/28 02:59:27	1.2 min	100/100	2.9 GiB		2.9 GiB	3.4 GiB
14	reduceByKey at /mnt/data/ass1-code/Ass1_P3task3.py:42 +details	2022/09/28 02:58:03	1.4 min	100/100			3.4 GiB	2.9 GiB
13	join at /mnt/data/ass1-code/Ass1_P3task3.py:39 +details	2022/09/28 02:56:51	1.2 min	100/100	2.9 GiB		2.9 GiB	3.4 GiB
12	reduceByKey at /mnt/data/ass1-code/Ass1_P3task3.py:42 +details	2022/09/28 02:55:25	1.4 min	100/100			3.4 GiB	2.9 GiB
11	join at /mnt/data/ass1-code/Ass1_P3task3.py:39 +details	2022/09/28 02:54:14	1.2 min	100/100	2.9 GiB		2.9 GiB	3.4 GiB
10	reduceByKey at /mnt/data/ass1-code/Ass1_P3task3.py:42 +details	2022/09/28 02:52:48	1.4 min	100/100			3.4 GiB	2.9 GiB
9	join at /mnt/data/ass1-code/Ass1_P3task3.py:39 +details	2022/09/28 02:51:34	1.2 min	100/100	2.9 GiB		2.9 GiB	3.4 GiB
8	reduceByKey at /mnt/data/ass1-code/Ass1_P3task3.py:42 +details	2022/09/28 02:50:08	1.4 min	100/100			3.4 GiB	2.9 GiB
7	join at /mnt/data/ass1-code/Ass1_P3task3.py:39 +details	2022/09/28 02:48:54	1.2 min	100/100	2.9 GiB		2.9 GiB	3.4 GiB
6 (retry 1)	reduceByKey at /mnt/data/ass1-code/Ass1_P3task3.py:42 +details	2022/09/28 02:48:21	34 s	36/36			1262.5 MiB	1062.0 MiB
5 (retry 1)	join at /mnt/data/ass1-code/Ass1_P3task3.py:39 +details	2022/09/28 02:47:54	27 s	33/33	976.8 MiB		972.9 MiB	1158.7 MiB
4 (retry 1)	reduceByKey at /mnt/data/ass1-code/Ass1_P3task3.py:42 +details	2022/09/28 02:47:21	33 s	35/35			1224.7 MiB	1030.9 MiB
3 (retry 1)	join at /mnt/data/ass1-code/Ass1_P3task3.py:39 +details	2022/09/28 02:46:54	27 s	34/34	1005.6 MiB		979.5 MiB	1189.9 MiB
2 (retry 1)	reduceByKey at /mnt/data/ass1-code/Ass1_P3task3.py:42 +details	2022/09/28 02:46:28	26 s	35/35			1036.1 MiB	1008.8 MiB
1 (retry 1)	join at /mnt/data/ass1-code/Ass1_P3task3.py:39 +details	2022/09/28 02:46:05	24 s	34/34	1005.6 MiB		1006.6 MiB	1006.5 MiB

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

Figure 19 Stages recomputed after 1st worker was killed

Contributions

- We used to have regular meetings and discuss what to do and how to do it.
- **Agam** did the Hadoop and Spark setup.
- And then each one of us did the part-2 (Sorting the data) to get the familiarity of how the data is stored in Hadoop and commands related to the Hadoop and spark.
- Then we had a discussion on how to implement the page rank algorithm. Each one of us came with our own approaches and after discussions we finalised with **Girish's** algorithm since it was better than others as he used flatmap() to update the source rank directly instead of storing them separately to update them in later steps.
- Then **Rishideep** worked mostly on optimising the code by using different kinds of partitioning techniques. We were using repartition but it was initially increasing the time and later with the help of spark UI he observed that repartition is creating new stages and making it delayed. So later we used the reduceBykey function only to change the partitions using different hash functions which improved the performance.
- Once we were able to find the improvement each one of us had worked on different kinds of partitions, hashing and caching and we also used to discuss why we are making these changes by checking the statistics.
- We used to share all our findings regularly and note the points and add the end we just organised our points and wrote the report.
- Apart from the algorithms we also faced many issues with the environment:
    - Issues regarding the setup are handled by **Agam** especially when the data nodes are not working or unable to connect to the master.
    - Issues regarding the spark UI and cluster health and storing the data (which was one of our main issues since we used to get no space error) were handled by **Girish**.
    - Issues regarding partitions, hash functions, getting the statistics data and visualising them were handled by **Rishideep**.