Big Data Systems (CS744)

Assignment 1

Group 6

Submitted By :
Chakshu Ahuja
Sushma Kudlur Nirvanappa
Vibhor Goel

Contents

1 Apache Hadoop and Spark Setup 2
2 A simple Spark application 3
3 PageRank 5
3.1 Naive Page Rank Algorithm 5
3.2 Partitioned Page Rank Algorithm 6
3.3 Page Rank Graph 7
3.4 Cached Page Rank 7
3.5 Cached Page Rank after killing one worker 8
4 Experimental Results 9
4.1 PageRank Partitioning 9
4.2 PageRank Persistence 11
4.3 Graph vs Naive Implementation 12
4.4 Cached Implementation along with killing worker when the job is half-way done 13

1 Apache Hadoop and Spark Setup

We setup the Apache Hadoop and Apache Spark on the cluster machines using the instructions provided in the assignment as below:

- We first setup SSH service among the nodes on the cluster, by designating one node as master and syncing its public key with the authorized keys in the other nodes. We later learned that the authorized keys get updated within certain interval, we resolved this by adding keys in the Cloudlab.
- We setup Apache Hadoop on each of the nodes, and specified the master-node ip in the '$HADOOP-INSTALL-DIR$/etc/hadoop/core-site.xml' to tell each of the nodes about which is the master node. We also add properties for datanode and namenode directories on the nodes.
- Next, we add paths at appropriate places. The information about which all nodes are present in the cluster is added in the 'hadoop-2.7.6/etc/hadoop/slaves' file in cluster.
- The complete setup could be tested by formatting the namenode and starting the distributed file system using start-dfs.sh command.
- The setup for Apache Spark was done similarly on all nodes, by adding the information about the nodes in cluster in the $SPARK-INSTALL-DIR$/conf/slaves.

Some issues which we faced while setting up and learning for the same are summarized below:

- While setting up Hadoop on the cluster, we faced the issue of NameNode not getting created because of port already being in use. We found out that we had previously created NameNode on the same and did not stop our hdfs properly, due to some permission issues. Therefore, a process which was using that port earlier did not get killed. We killed the same and made permission changes to resolve the issue.
- We faced another problem of NameNode not being present when we used the jps command. We realized that the namenode was not getting formatted properly because of permission issues on the namenode directory on the master. We further added property for secondary namenode in the hdfs-site.xml file and restarted with the start-dfs.sh command, which resolved our problem.
- While submitting an application to spark, we found error in the spark context being already in use. We found that another process started by a teammate was already running on the same, annd only one SparkContext is being allowed by default. We later also found out about the possibility of using multiple spark contexts and the issues they cause, as they are only meant for use in internal spark tests.

2 A simple Spark application

Sort

The task involves creating a simple spark application to sort a 1000 line records (CSV file) in the order of two specific columns (Country Code and Timestamp). We tried solving this problem using two different approaches:

- **DataFrames:** A DataFrame is a distributed collection of data organized into named columns. DataFrames offers simple APIs to read and store CSV files, and to sort based on single or multiple data columns. Here, we first got a tabular representation of the dataset by reading the input CSV file. We sorted the records by specific columns in the order of Country Code first and then timestamp to retrieve the order within the same country code.

- **RDD:** Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster (provides better performance when dealing with huge dataset). Here, we first converted the input CSV dataset into RDD then we sorted the line on index 2 (for country code) and index 14 (timestamp) after splitting each line by comma (,). The sorted data are partitioned RDDs which then will be aggregated and written to a text file.

We have programmed this application in Scala programming language. We conducted experiments to benchmark the performance of the framework with both RDD and DataFrames. We tried to track the performance of the cluster by varying executor memory size and also conducted trials to compare the performance of cluster execution against the local machine.

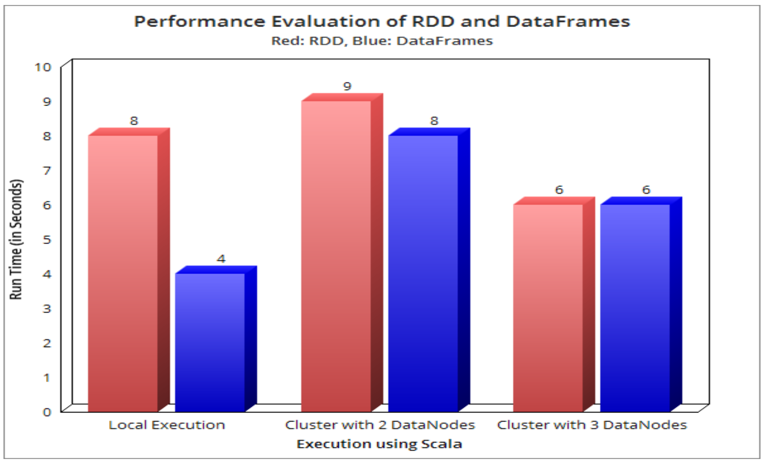


Figure 1: RDD vs DataFrame

We represent some of our observations below:

- **RDD vs DataFrames** We expected the performance with DataFrames to be better than RDD owing to its built-in query optimization. And clearly observed the same result where execution time with DataFrames(DF) was better than RDD. Although we observed fluctuating results on multiple runs, DF was still performing better than RDD. We witnessed a very marginal difference in execution time between them as we started working with the small dataset of 1000 lines but we did observe a observable difference when we conducted experiments on copies of these 1000 line records. And thus it is acceptable that the query optimization support in DF provides better performance than RDDs.
- **Local Execution Vs Cluster Execution** A single multi-threaded local machine outperformed the cluster execution with both RDD and DataFrames. Remote reads have resulted in network overheads increasing the execution time of cluster setup when compared with local machine execution time. This is acceptable on a small dataset of 1000 lines but a cluster setup provides better performance when dealing with a huge dataset which has been observed in our next experiment (Pagerank computation).
- **Cluster with 2 DataNodes and 3 DataNodes:** A NameNode performing as DataNode (cluster of 3 nodes with 3 DataNodes and 1 NameNode) has clearly increased the execution time of the task with both RDD and DataFrames. This is due to the overhead at the NameNode which is expected behaviour.

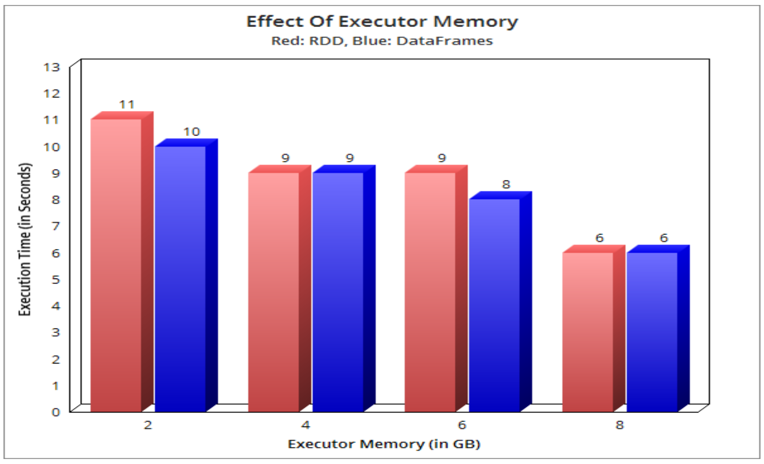


Figure 2: Performance evaluation with change in Executor memory size

- **Executor Memory:** As computation happens in-memory with RDDs, we tried to record the performance by varying the executor memory size. We can clearly notice that there is definitely some improvement in performance with increasing memory size. This is expected as the dataset is small but while working with huge dataset the intermediate results won't fit in memory and would be spilled over to disk which adds to the performance overheads.

3 PageRank

Page Rank Algorithm has been designed to evaluate the quality of the links to a webpage. The Wiki definition states how it has been used to rank web pages in Google Search's results.

3.1 Naive Page Rank Algorithm

Task 1. Write a Scala/Python/Java Spark application that implements the PageRank algorithm.

The naive implementation of PageRank algorithm is based on the idea of using the edges given. The edges here imply the mapping of one source url to one destination url. In the given dataset, each record/line represents an edge.

Algorithm

Step 1: We iterate through each record and form a pair in order to form an RDD of edges represented as RDD[(String, String)].

Step 2: Next we work on the edges to get the count of neighbours for each source url.

Step 3: Set the initial ranks to be 1 of each source url (node) and run the first iteration of the algorithm.

Step 4: In each iteration, we perform a join on RDDs of edges, previous ranks and count of neighbours in order to calculate the contribution received by a url (node).

Step 5(optional): We add contribution of 0 for every source url(node) since it might not have any URL pointing to it. This ensures that we calculate ranks of url with no incoming nodes.

Step 6: With the RDD obtained till now of RDD[(String, Double)], we have a mapping of node and the contribution received. We reduce this RDD on the key but sum operations which results in forming an RDD to know the contribution received per node(url).

Step 7: Finally, we calculate the ranks for this iteration using the given formula of 0.15 + 0.85 * contribution received from all neighbours.

Step 8: Repeat the steps from 4-8 to perform another iterations until specified number of iterations.

The inefficiency in the model results due to couple of factors:

- There is no partitioning done on the edges/data which leads to no co-location.
- On each iteration, we are performing two joins: one on the edges and count of neighbours and other being with the ranks RDD in the previous iteration.
- Since there was no caching performed, it further leads to getting the edges data all the way by doing union on whole data everytime which further leads to reading all blocks of input file from the HDFS.
- The experimental results as shown in Section 4 reveal about quite large number of tasks and stages created. That in-turn results into high CPU time, high disk write, high shuffle writes, high network utilization and longer duration of the job.

3.2 Partitioned Page Rank Algorithm

Task 2. Add appropriate custom RDD partitioning and see what changes.

In the naive approach, for a given source url, the various destination urls might be in different partitions. Performing a Hash Partition based on the key as the source URL gives the guarantee the destination URLs are now present in the same partition. As a result, this optimized the join operation that needs to be performed since all the keys of the join are now in same partition.

Another optimization done in this case is using ***mapValues*** instead of ***map*** operation in calculating count of neighbours of source node so as to not lose the partition. The re-shuffle happens because since Spark has no idea about what map operation would do and there is a possibility of key change in map operation, the partition already performed gets reshuffled. However, that is not the case with ***mapValues*** because ***mapValues*** ensures that the value of the inputRDD is going to change and not the keys. This helps in preserving the partition performed in the previous step. The same concept of *mapValues* is used when calculating the rank after each iteration from the contribution received per node so that the partition performed in previous reduceBy operation is preserved.

Another advantage of having a Partitioner and making sure that we retain the partitioning across iteration is that it prevents re-evaluation of the Edges and Initial Rank RDD from before the ***partitionBy*** clause. This is called **ShuffleMapStage sharing**. Basically, when Spark does a shuffle (which in this case happens during the partitionBy operation), it remembers the lineage of the RDD along with the paritioning information. Since this data is written to disk (due to the shuffle), we don't need to evaluate the RDD before the shuffle phase. This can be seen from the DAG diagram: on each iteration, each task doesn't re-read the edges from HDFS, rather they are stored on the local disk and read by the reducers.

We have experimented using

- HashParitioning,
- Range Partitioning.

Heuristic based Graph Partitioner

One thing that can potentially increase performance is if we do partitioning by considering connections within the graph itself. That is, if we partition the graph in such a way that the nodes which are connected end up on the same partition, then we can expect that the reduceByKey inside each iteration to shuffle less data. Since this shuffle happens 10 times, this will lead to drastic reduction in total shuffle write. The strategy would be to first use a graph partitioning heuristic to read the graph and partition it in P partitions and then construct a custom partitioner based on that knowledge. Then we can use this custom partitioner to partition the graph, and the initialRank RDD.

3.3 Page Rank Graph

The next intuition that comes into picture is since we are anyway partitioning the data such that all destination urls belonging to same source url can be placed together. We can form a graph represented as adjacency list (rdd *graph*). Instead of keeping track of count of neighbours of a node, we store the neighbours of every node. In a well connected graph, that would save quite lot of memory in order to store the initial data. In this case, in every iteration. We perform only a single join of graph with the previous calculated ranks. The potential downside of this approach can be when dealing with large graphs which have one node connected to a a lot of destination nodes. This case would lead to creation of really large objects which can cause Out of Memory (OOMs) or increase the Garbage Collection (GC) pressure.

3.4 Cached Page Rank

Task 3. Persist the appropriate RDD as in-memory objects and see what changes.

So till this point the whole data of edges or graph was re-calculated. For each iteration, in the join, we have to re-read the data. This can be avoided if we use Spark's cache(). After using the cache(), we observed that the graph data is being read from the memory hereby resulting in faster computation. The green dots in the DAG Visualization show that the data is cached.

3.5 Cached Page Rank after killing one worker

Task 4. Kill a Worker process and see the changes. You should trigger the failure to a desired worker VM when the application reaches 50

We wrote a simple script that uses the Spark's REST API to query the progress of the job and then kills a worker node application is about half way done by using

wait_for_spark_app.py 50 sudo sh -c 'sync; echo 3 >/proc/sys/vm/drop_caches' ./spark-2.2.0-bin-hadoop2.7/sbin/stop-slave.sh

The consequence of running this script is that when the number of completed task is about 50 percent, the executor of the worker node on which this script is executed dies. The name of the dead executor can be seen by going on the Executors tab on Spark UI. When this happens, Spark's Driver is able to detect this, and is able to spawn new tasks to be performed in order to do the calculations which had been performed by the killed executor.

This can be seen very clearly by first going to the ***"Stages"*** tab of the UI. Then we go to the Stage 1 to figure out how many tasks was performed by the killed executor.

4 Experimental Results

4.1 PageRank Partitioning

In our code, the rdd *graph* stores the links between the source to the destinations. In the computation of new ranks for each iteration, this *graph* is used to join with the ranks in previous iteration to calculate contributions per node.
Since, there is a join between the *ranks* and the *graph* rdd, we can optimize this operation for shuffle reads if the rdd *graph* is partitioned upon the key same as *ranks*. We first evaluate the effect of HashPartitioner which is the default partitioner in Scala. We compare the running times and shuffle reads for different partitions as shown in 1.

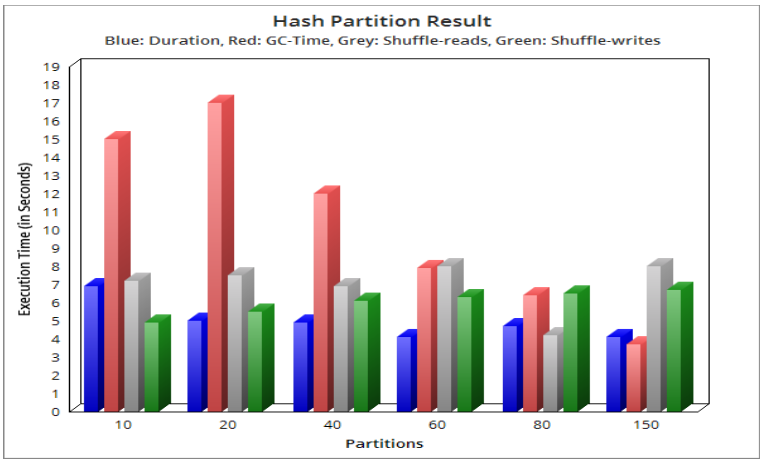


Table 1: Hash Partitioning results

Partitions	Duration	GC-Time	Tasks	Shuffle-read	Shuffle-write
10	6.9	15	143	7.2	4.9
20	5.0	17	253	7.5	5.5
40	4.9	13	473	6.9	6.1
60	4.1	7.9	693	8	6.3
80	4.7	6.4	913	4.2	6.5
150	4.1	3.7	1683	8	6.7

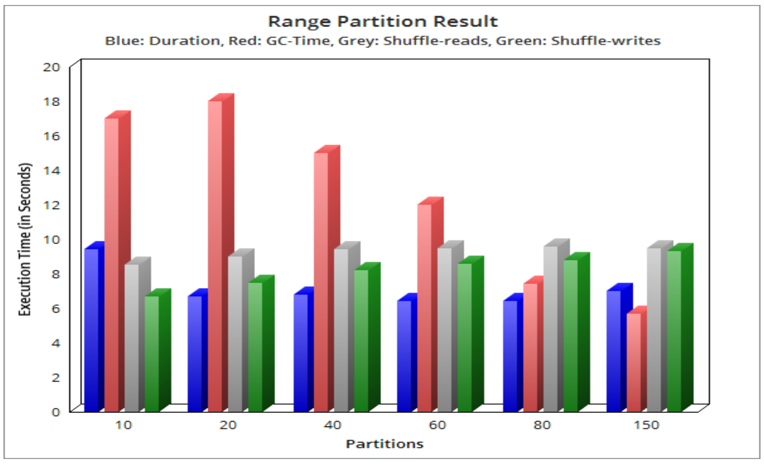


Table 2: Range Partitioning results

Partitions	Duration	GC-Time	Tasks	Shuffle-read	Shuffle-Write
10	9.4	17	233	8.5	6.7
20	6.7	18	433	9.0	7.5
40	6.8	15	833	9.4	8.2
60	6.4	12	1233	9.5	8.6
80	6.4	7.4	1633	9.6	8.8
150	7.0	5.7	3033	9.5	9.3

From the table 2, we can observe the results for **hash based partioning** on the rdd *graph* [which stores our mapping from source to adj list of destinations]. We can see that the duration for process slowly decreases with partition, though this relation is non-linear. The total duration reaches a minimum signifying that further partitioning does not impact results much, as the other tasks become the bottleneck. GC-time, which is the **sum** of times taken on different nodes also gets decreased with partition size. Similarly, we can also see the no of tasks increases almost in a linear manner as we increase the number of partitions. Also, there is slight increase in the number of shuffle reads and shuffle writes with minor variations as we increase the no of partitions. This is expected for hash partitioning as there will be slight increase in data transfer because the data for keys is now getting more dispersed.

From table 2, we can observe the results for **range based partitioning** on the rdd *graph*. We can again observe decrease in total process duration and GC-time similar to that with hash based partitioning. Also, the no of tasks increase again with partitions in almost linear manner, and increase in shuffle reads/writes is witnessed.

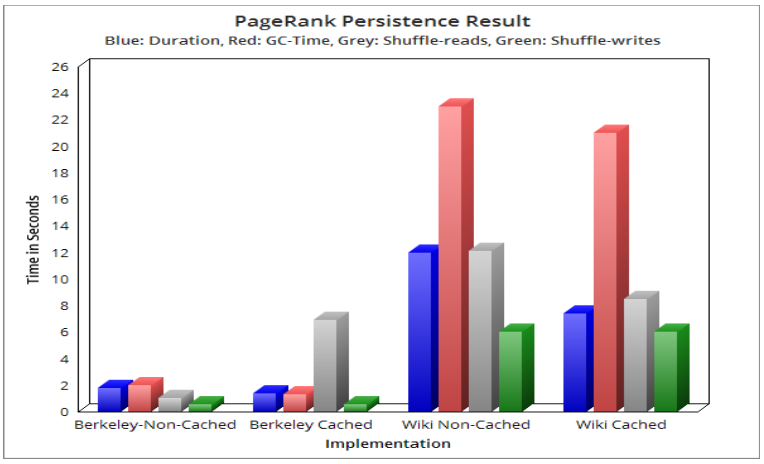
In comparison to hash based partitioning, we observe that range based partitioning takes more time. We attribute this to the fact that for range partitioning, a shuffle between partitions would be required while sorting, but for hash based partitioning, we only need to shuffle within a partition for our *edges* rdd [it stores pairs of src and dest, from which the rdd *graph* is computed]. Also, since we are not using persistence on *edges*, this computation significantly affects the total duration, tasks and shuffle reads.

Table 3: File based Partitioning results for 150 partitions

Type of Partitions	Partition function	Duration	Tasks	Shuffle-read
File-based	Hash	7.9	1801	7.9
Combined rdd	Hash	4.1	1683	8.0
File-based	Range	7.6	3451	9.5
Combined-rdd	Range	7.0	3033	9.5

Another method which we observed in partitioning was creating rdd of each text file seperately, and combining them without repartition [Table 3]. This approach creates 150 partitions by default, 15 for each file, on 3 nodes and 5 executors based on hdfs default configurations. We compare this approach with the above described method of partitioning on graph. From here we don't see much change apart from the duration of hash based partitioning over combined-rdd, and file-based rdd. We attribute this to the fact that hash based partitioning on combined-rdd provides a really efficient way of partitioning which does not require interim data transfer between partitions. While for file-based partitioning, it would require interim data transfer between partitions again while forming them due to multiple union operations involved. The reasoning for range-based partitioning is similar as above, and is not very efficient.

4.2 PageRank Persistence



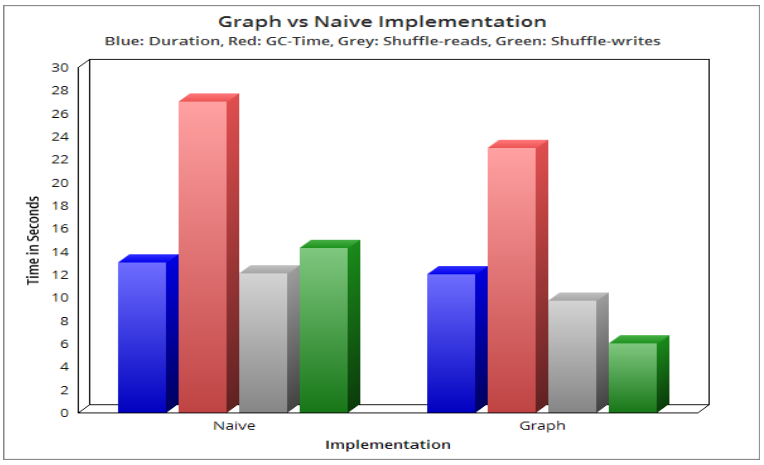
We also analyze the performance improvement when we cache our *graph* rdd. Since, in our each iteration, the graph rdd is being used to join with previous ranks and computed again. It makes sense to persist it in memory to avoid extra computations. We show our results for the same for both hash and range partitions in Table 4.

Metric	Berk Non-cached	Berk-Cached	Wiki-Non Cached	Wiki-Cached
Tasks	1666	1666	1801	1801
Duration	1.8	1.4	12	7.4
GC-Time (min)	2.0	1.3	23	21
Shuffle-read (GB)	1	699.7	12.1	8.5
Shuffle-write (GB)	0.571	0.571	6	6

Table 4: Persistence Comaprison on PageRank for Berkeley and Wiki datasets

We can clearly see that in case of persistence, a significant reduction in total task duration is present. Similarly, the amount of shuffle-read is also reduced by approx 30% in both the cases. This is expected, because a large amount of data gets read for computing the *graph* for each iteration. However, no changes in the metric shuffle write is seen as recomputation of edges does not cause any shuffle writes.

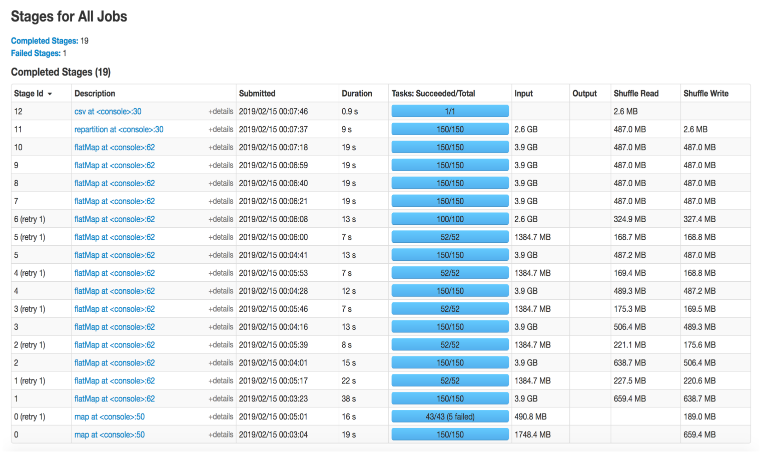
4.3 Graph vs Naive Implementation



Metric	Wiki Graph	Wiki Naive
Tasks	1801	5251
Duration	12	13
GC-Time (min)	23	27
Shuffle-read (GB)	12.1	9.7
Shuffle-write (GB)	6	14.3

Table 5: Naive implementation vs graph implementation comparison

In the table 5, we can see the performance evaluation between the two implementations which we thought about. We can observe that the graph based implementation improves over the naive implementation in terms of duration. We believe the reason



Stages for All Jobs

Completed Stages: 19
Failed Stages: 1

Completed Stages (19)

Stage Id ▾	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
12	csv at <console>:30	+details	2019/02/15 00:07:46	0.9 s	1/1			2.6 MB	
11	repartition at <console>:30	+details	2019/02/15 00:07:37	9 s	150/150	2.6 GB		487.0 MB	2.6 MB
10	flatMap at <console>:62	+details	2019/02/15 00:07:18	19 s	150/150	3.9 GB		487.0 MB	487.0 MB
9	flatMap at <console>:62	+details	2019/02/15 00:06:59	19 s	150/150	3.9 GB		487.0 MB	487.0 MB
8	flatMap at <console>:62	+details	2019/02/15 00:06:40	19 s	150/150	3.9 GB		487.0 MB	487.0 MB
7	flatMap at <console>:62	+details	2019/02/15 00:06:21	19 s	150/150	3.9 GB		487.0 MB	487.0 MB
6 (retry 1)	flatMap at <console>:62	+details	2019/02/15 00:06:08	13 s	100/100	2.6 GB		324.9 MB	327.4 MB
5 (retry 1)	flatMap at <console>:62	+details	2019/02/15 00:06:00	7 s	52/52	1384.7 MB		168.7 MB	168.8 MB
5	flatMap at <console>:62	+details	2019/02/15 00:04:41	13 s	150/150	3.9 GB		487.2 MB	487.0 MB
4 (retry 1)	flatMap at <console>:62	+details	2019/02/15 00:05:53	7 s	52/52	1384.7 MB		169.4 MB	168.8 MB
4	flatMap at <console>:62	+details	2019/02/15 00:04:28	12 s	150/150	3.9 GB		489.3 MB	487.2 MB
3 (retry 1)	flatMap at <console>:62	+details	2019/02/15 00:05:46	7 s	52/52	1384.7 MB		175.3 MB	169.5 MB
3	flatMap at <console>:62	+details	2019/02/15 00:04:16	13 s	150/150	3.9 GB		506.4 MB	489.3 MB
2 (retry 1)	flatMap at <console>:62	+details	2019/02/15 00:05:39	8 s	52/52	1384.7 MB		221.1 MB	175.6 MB
2	flatMap at <console>:62	+details	2019/02/15 00:04:01	15 s	150/150	3.9 GB		638.7 MB	506.4 MB
1 (retry 1)	flatMap at <console>:62	+details	2019/02/15 00:05:17	22 s	52/52	1384.7 MB		227.5 MB	220.6 MB
1	flatMap at <console>:62	+details	2019/02/15 00:03:23	38 s	150/150	3.9 GB		659.4 MB	638.7 MB
0 (retry 1)	map at <console>:50	+details	2019/02/15 00:05:01	16 s	43/43 (5 failed)	490.8 MB			189.0 MB
0	map at <console>:50	+details	2019/02/15 00:03:04	19 s	150/150	1748.4 MB			659.4 MB

Figure 3: Stages 1-6 retried after one worked was killed at around 50% execution

behind this is the fact that in the graph based approach, only a single join is required with previous ranks for each iteration, instead of two required in the naive based. This also reduces the no of tasks required significantly.

4.4 Cached Implementation along with killing worker when the job is half-way done

In our test, when the job was around half done, the number of tasks which the killed executor performed was 52 out of total 150 tasks for this stage.

This means that those 52 tasks need to be re-performed by some other executor. Spark creates the "retry" stage where it re-runs the stage, but only to calculate the 52 tasks (rather than the full 150 tasks). Similarly, we see the same thing happening for all the subsequent stages, up till stage 6. Stage 6 is the stage where we ended up killing the executor. After the stage 6, in every subsequent stage, Spark assumes the dead executor and proceed by dividing the rest of the tasks between the two alive executors. This can be seen by inspecting these stages and metrics under "Aggregated Metrics Per Executor" section.

The failed executor doesn't affect the completion of the task in anyway. This is because Spark's design gives a fault-tolerant way to design shared memory computation. However, we note that the duration of the job increases from 3 minute to 4.5 minutes. Which is expected as we have lost part of the computation on which all the future iteration depends on. And so, that lineage needs to be recomputed, hence the increased time as can be seen in Figure 3.