CS744 - Big Data Systems
Assignment 1

Ushmal Ramesh
uramesh2@wisc.edu

Varsha Pendyala
varsha.pendyala@wisc.edu

Varun Thumbe
thumbe@wisc.edu

1. *Setup, Configuration and Issues*

We followed the setup instructions posted on the assignment page to get HDFS and Spark running on the cluster machines. Following were the issues faced during this phase:

a. *DataNodes not starting up correctly*: The issue was that after formatting the namenode the datanodes were not getting started. Deleting the datanode directory from master and slave, recreating it formatting the namenode and then starting the hdfs daemon solved the issue

b. *Connection refused errors while starting up HDFS/Spark:* We faced intermittent connection errors between master and slave VMs which threw up errors while bringing up HDFS and Spark. This was traced to automatic deletion of the ssh keys we added to the authorized keys file. Restoring the ssh keys resolved the issue.

c. *PageRank Space Issues*: While executing pagerank algorithm we had to move the hadoop installation to mnt/data directory because of space issue and hence change all relevant configuration files in the hadoop folders. However we found later on that the better solution was to read the data directly from the data folder provided in the assignment.

d. *Permission Issue while mounting HDFS on mnt/data*: While mounting HDFS on mnt/data we observed errors like '*EPERM: Operation not Permitted*" while starting up the datanodes. Upon troubleshooting, we discovered that this was caused by the /mnt/data folder having ownership only to the root and not any other user. We resolved the issue by changing both the read/write permission of the mnt/data folder as well as changing the ownership of the folder(using chown) to our usernames.

e. *Profile Without Routable IPs:* We used the profile 'shivaram-cs744-fa19-assign1' while starting to do the part3. It turns out that since those instances don't have routable ip and we were unable to monitor running spark jobs. This was later resolved after instructions on how to access non routable IP servers was posted on Piazza. We followed the instructions and were able to access the web URLs.

f. *Spark Web URL inaccessible* : The spark web url at port 8080 lists the running jobs and the jobs url can be accessed by clicking on the running job. This is at port 4040 by default. We started observing that this page becomes inaccessible after a while. Upon checking the port via lsof we killed the process running on that port. After this, the page was accessible again.

g. *Spark History Server*: To access the logs after the completion of the spark job we started a spark history server to keep to get a structured log of the completed applications.

h. *Switching to Experimental Network* - We switched to using internal IP addresses by changing the IP addresses in all the configuration files described on the assignment page. However, even after changing these files, upon running *netstat -anp* | *grep LISTEN* we saw that the spark master was listening on public IP and on other nodes the workers were listening on PublicIP. After reading the Spark documentation we added the following flags to Master and Worker machines in the file spark-env.sh:
    - SPARK_MASTER_HOST - Sets the IP of the master
    - SPARK_LOCAL_IP - Sets the IP of the worker

    In order to set the spark master IP to private IP in code we used the following command:

    ```
    /sbin/ifconfig eth1 | grep 'inet addr:' | cut -d: -f2 | awk '{ print $1}'
    ```

    The above command searches for '*inet addr:*' in eth1 (where the private IP address usually gets listed) and then extracts the IP.

2. Implementing Sort Using Spark

We implemented the Sort after going through the input file format and used the PySpark interactive shell to cook the script. The given problem is to implement a Secondary Sort using a composite key. The following is the code:

```
from pyspark.sql import SparkSession
from pyspark import SparkConf, SparkContext
import commands
ip_addr = commands.getoutput("/sbin/ifconfig eth1 | grep 'inet addr:' | cut -d:
-f2 | awk '{ print $1}'")
url = "hdfs://"+ip_addr+":9000/"  # url of hdfs
conf = SparkConf().setAppName('Sort').setMaster('spark://'+ ip_addr +':7077')
sc = SparkContext.getOrCreate(conf=conf)
rdd1 = sc.textFile(url+"input") #read input from hdfs
rdd1 = rdd1.filter(lambda l: not l.startswith("battery_level"))
rdd1 = rdd1.sortBy(lambda l: (l.split(",")[2], l.split(",")[-1]))
rdd1.saveAsTextFile (url+"output") #save output to hdfs
spark.stop()
```

We first filter out the file header by applying a filter transformation on the RDD read in from the file. After this we sort the RDD after constructing a composite key made of the two quantities we need to sort on(Country Code, Timestamp). The composite key is constructed using a lambda function which is then passed into the sort function call on the RDD.

Spark context configuration issue: In order to make the code machine independent to avoid entering IP addresses repeatedly, we initially used sc.uiweburl to retrieve the mater node's IP and then use it to configure the spark context object. However, we found that this doesn't give the correct url for the spark master, and as a result the driver program ran in the local machine and not on the spark cluster. We resolved this issue by running ifconfig to retrieve the IP from inside the driver program and then used spark://master_ip:7071 as the master url.

3. PageRank Implementation

3.1 Task 1 - Algorithm Implementation

We implemented the page rank algorithm on the smaller web-BerkStan dataset to test the outputs on a single node. As before with sort, we used interactive PySpark Session to cook the script. Following is our implementation of pagerank for web-BerkStan.

```
from pyspark.sql import SparkSession
from pyspark import SparkConf, SparkContext
from operator import add
import commands
ip_addr = commands.getoutput("/sbin/ifconfig eth1 | grep 'inet addr:' | cut -d: -f2 |
awk '{ print $1}'")

url = "hdfs://"+ip_addr+":9000/"
conf = SparkConf().setAppName('PageRank').setMaster('spark://'+ ip_addr +':7077')
sc = SparkContext(conf=conf)
lines = sc.textFile(url+'ranks_input').filter(lambda l: not l.startswith('#'))
links = lines.map(lambda l: (l.split('\t')[0],l.split('\t')[1])).groupByKey()
ranks = links.keys().map(lambda l: (l,1))
for i in range(10):
        contribs=links.join(ranks).flatMap(lambdal:map(lambda  a   :  (a  ,  l[1][1]  /
len(l[1][0])),l[1][0]))
    ranks = contribs.reduceByKey(add).mapValues(lambda l: 0.15 + 0.85*l)
ranks.saveAsTextFile(url+'ranks_output')
```

For this, we implemented the PageRank algorithm as presented in the RDD paper. We first read the input RDD from hdfs (we store the dataset in hdfs using *hdfs dfs -put* command) and filter out commented lines from the input RDD. Then we construct a ‘links’ RDD by first converting the input RDD into a key, value pair RDD (representing source and destination) and then running a groupBy query to collect destinations corresponding to individual sources. We also construct a ‘ranks’ RDD to initialize ranks for all sources to 1.

The ranks and links RDDs are then fed to the iterative pagerank algorithm to calculate the ranks iteratively. Finally we store the output RDD in hdfs. We checked the correctness by loading the hdfs file back into local fs and doing some sanity checks based on the number of times a certain destination article is present in the dataset.

3.2 Task2 - Pagerank on the enwiki dataset

Based on the format of the files in the enwiki dataset, we added some more preprocessing code to process the data. The preprocessing discards any lines that do not have the required 'category:' string.

Following are the further modifications/optimizations we made to the original code for Pagerank.

Modification 1:
In order to make sure that source articles which never appear as destination articles get a rank of 0.15 we perform a leftOuterJoin operation on the computed ranks RDD with the links RDD and then perform a map operation to identify any source articles that do not appear as a destination article and assign them a rank of 0.15.

Modification2:
We observed that our implementation of with leftOuterJoin wrote nearly 5GB of shuffle data to the local disks of the slave nodes and therefore fails to complete even 3 iterations due space shortage errors. In order to overcome this, we changed the above logic to apply a *keys().keyby()* operation before the leftOuterJoin on the links RDD. After this, we observed that the shuffle write data corresponding to that stage has reduced to a much lower value of about 0.9 GB. We believe that the improved storage utilization is because the modification (*links.keys.keyby()*) results in a much smaller RDD compared to *links* RDD. However, it did not result in reduced shuffle read but instead improved shuffle write for the leftOuterJoin() stage.

This modification enabled us to complete 3 iterations of the algorithm without failure with 24GB of disk space on each worker. We lost all logs of these runs after the cloudlab cluster was suddenly shut down.

Modification 3:
We observed that the leftOuterJoin was an expensive operation and was the cause of the disk space issue faced while running the experiment for 5 iterations. We decided to modify the code by removing the leftOuterJoin operation and adding a distinct call before the groupBy operation. With these modifications, we could run our experiment on the entire enwiki dataset for 5 iterations successfully. However, we understand that in this implementation we are not addressing the source articles that do not appear as destination for any other article. Since those source articles have very less contribution to the rank of other articles, we believe this modification is not required even if we have more disk space available on workers to run our code with modification 2 in place. Also, we observed that we could optimize the groupby operation by first calling distinct on the RDD to get distinct (source, destination) pairs.

3.3 Task 3 - Pagerank with Partitioning and Persistence

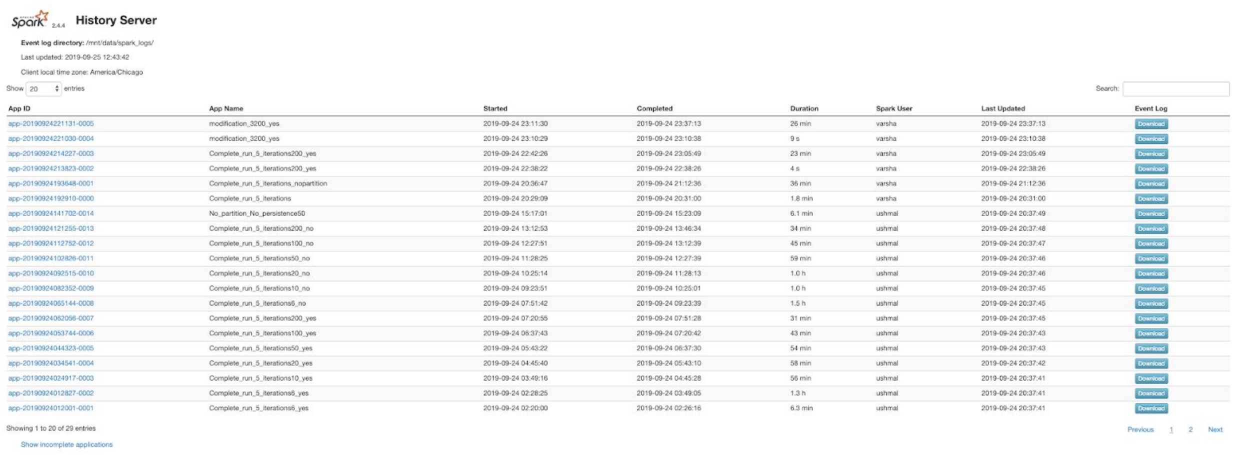
We changed the default partitioning and persistence level of the links RDD by explicitly calling partitionBy on the links RDD. We made this choice because of the following reasons:

1. The links RDD has operations like groupBy and leftOuterJoin applied on it at various stages during the algorithm and therefore supplying a partitioner will achieve performance gains. Once we partition the RDD we need to persist it immediately to avoid recomputing the partitions
2. The ranks RDD is also a candidate for persistence. However, we decided against it since ranks is updated iteratively and therefore persistence would not yield any benefits.

3.3.1 Experiment Setup

1. *Parameters* : We decided to run the experiments with varying number of partitions namely 6, 10, 20, 50 and 100 and also with persistence and without persistence.
2. *Logging with Spark History Server* - We setup the *Spark History Server* to log the experiments so that data from each finished job can be collected after completion. This is not possible without this as the job url is no longer accessible after its completion. To enable the history server we made changes to the spark

configuration file to specify the spark event logs directory and turned on event logging. Then we started the history server via the start-history-server shell script in sbin folder. Following is a screenshot of the history server page which is accessible at *<master_ip>:18080*. Enabling the history server helped us get access to individual jobs' DAG after completion as normally they are accessible only during the lifetime of the job.



Event log directory: /mnt/data/spark_logs/

Last updated: 2019-09-25 12:43:42

Client local time zone: America/Chicago

App ID	App Name	Started	Completed	Duration	Spark User	Last Updated	Event Log
app-20190924221131-0005	modification_3200_yes	2019-09-24 23:11:30	2019-09-24 23:37:13	26 min	varsha	2019-09-24 23:37:13	Download
app-20190924221030-0004	modification_3200_yes	2019-09-24 23:10:29	2019-09-24 23:10:38	9 s	varsha	2019-09-24 23:10:38	Download
app-20190924214227-0003	Complete_run_5_iterations200_yes	2019-09-24 22:42:26	2019-09-24 23:05:49	23 min	varsha	2019-09-24 23:05:49	Download
app-20190924213823-0002	Complete_run_5_iterations200_yes	2019-09-24 22:38:22	2019-09-24 22:38:26	4 s	varsha	2019-09-24 22:38:26	Download
app-20190924193648-0001	Complete_run_5_iterations_nopartition	2019-09-24 20:36:47	2019-09-24 21:12:36	36 min	varsha	2019-09-24 21:12:36	Download
app-20190924192910-0000	Complete_run_5_iterations	2019-09-24 20:29:09	2019-09-24 20:31:00	1.8 min	varsha	2019-09-24 20:31:00	Download
app-20190924141702-0014	No_partition_No_persistence50	2019-09-24 15:17:01	2019-09-24 15:23:09	6.1 min	ushmal	2019-09-24 20:37:49	Download
app-20190924121255-0013	Complete_run_5_iterations200_no	2019-09-24 13:12:53	2019-09-24 13:46:34	34 min	ushmal	2019-09-24 20:37:48	Download
app-20190924112752-0012	Complete_run_5_iterations100_no	2019-09-24 12:27:51	2019-09-24 13:12:39	45 min	ushmal	2019-09-24 20:37:47	Download
app-20190924102826-0011	Complete_run_5_iterations50_no	2019-09-24 11:28:25	2019-09-24 12:27:39	59 min	ushmal	2019-09-24 20:37:46	Download
app-20190924092515-0010	Complete_run_5_iterations20_no	2019-09-24 10:25:14	2019-09-24 11:28:13	1.0 h	ushmal	2019-09-24 20:37:46	Download
app-20190924082352-0009	Complete_run_5_iterations10_no	2019-09-24 09:23:51	2019-09-24 10:25:01	1.0 h	ushmal	2019-09-24 20:37:45	Download
app-20190924065144-0008	Complete_run_5_iterations6_no	2019-09-24 07:51:42	2019-09-24 09:23:39	1.5 h	ushmal	2019-09-24 20:37:45	Download
app-20190924062056-0007	Complete_run_5_iterations200_yes	2019-09-24 07:20:55	2019-09-24 07:51:28	31 min	ushmal	2019-09-24 20:37:45	Download
app-20190924053744-0006	Complete_run_5_iterations100_yes	2019-09-24 06:37:43	2019-09-24 07:20:42	43 min	ushmal	2019-09-24 20:37:43	Download
app-20190924044323-0005	Complete_run_5_iterations50_yes	2019-09-24 05:43:22	2019-09-24 06:37:30	54 min	ushmal	2019-09-24 20:37:43	Download
app-20190924034541-0004	Complete_run_5_iterations20_yes	2019-09-24 04:45:40	2019-09-24 05:43:10	58 min	ushmal	2019-09-24 20:37:42	Download
app-20190924024917-0003	Complete_run_5_iterations10_yes	2019-09-24 03:49:16	2019-09-24 04:45:28	56 min	ushmal	2019-09-24 20:37:41	Download
app-20190924012827-0002	Complete_run_5_iterations6_yes	2019-09-24 02:28:25	2019-09-24 03:49:05	1.3 h	ushmal	2019-09-24 20:37:41	Download
app-20190924012001-0001	Complete_run_5_iterations6_yes	2019-09-24 02:20:00	2019-09-24 02:26:16	6.3 min	ushmal	2019-09-24 20:37:41	Download

Showing 1 to 20 of 29 entries

Figure 1 : History Server Page

3. *Experiment Bash Script*: In order to automate the jobs, we cooked up a bash script that selects the parameters of a particular job (partitions, persistence), clears the output HDFS directories and then feeds the job with parameters to the spark-submit file. The bash script therefore eliminates the manual work of feeding in each of the experiments along with parameters. This along with the history server automates the entire experiment and data gathering process and saved us a lot of time.

3.3.2 Experiment Results

All the test results below are for our Modification3 (without leftOuterJoin and distinct call added before groupBy) code.

Partitions	Tasks	Stages	Time(min) With Persistence	Time(min) without Persistence	Pagerank Iterations
6	391	10	78	90	5

10	427	10	56	60	5
20	517	10	58	60	5
50	787	10	54	59	5
100	1237	10	43	45	5
200	2137	10	31	34	5

Table 1 : Tests with Different Partitions and Persistence Levels

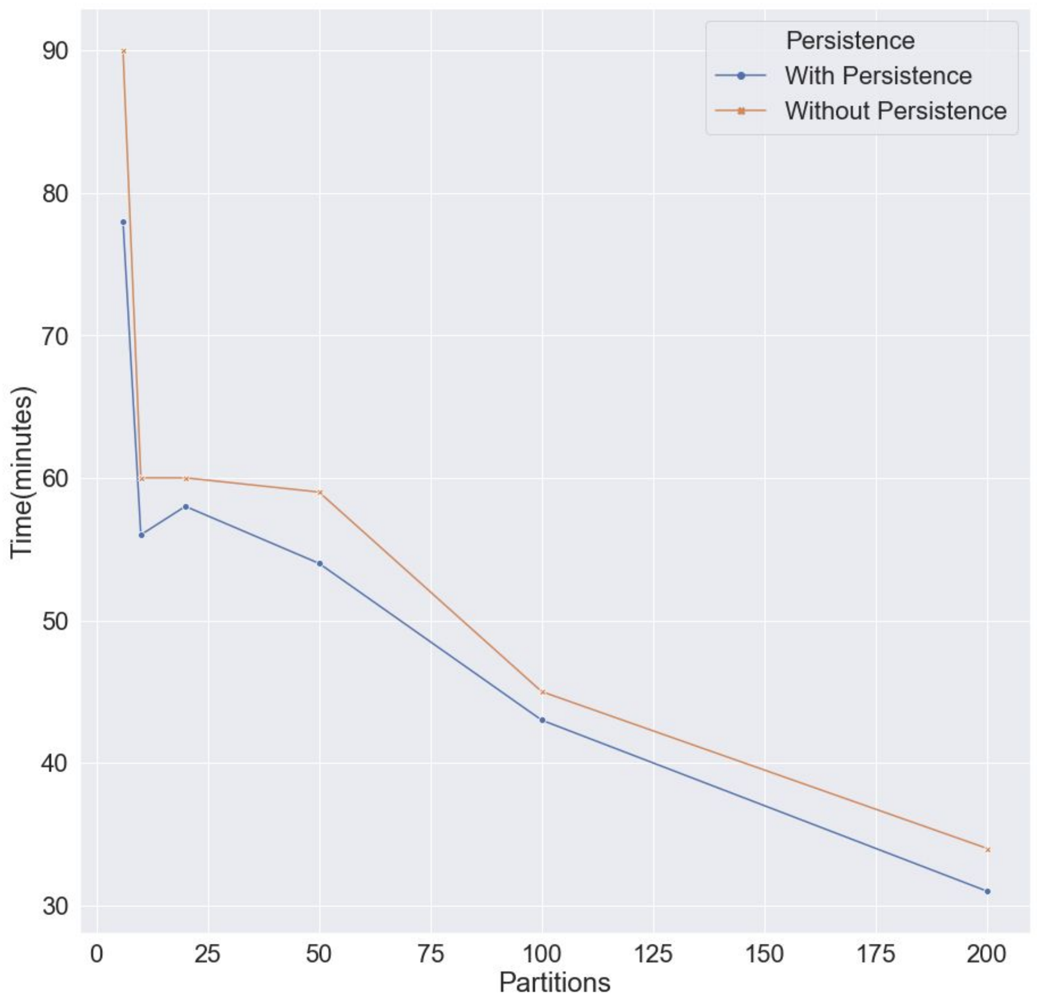
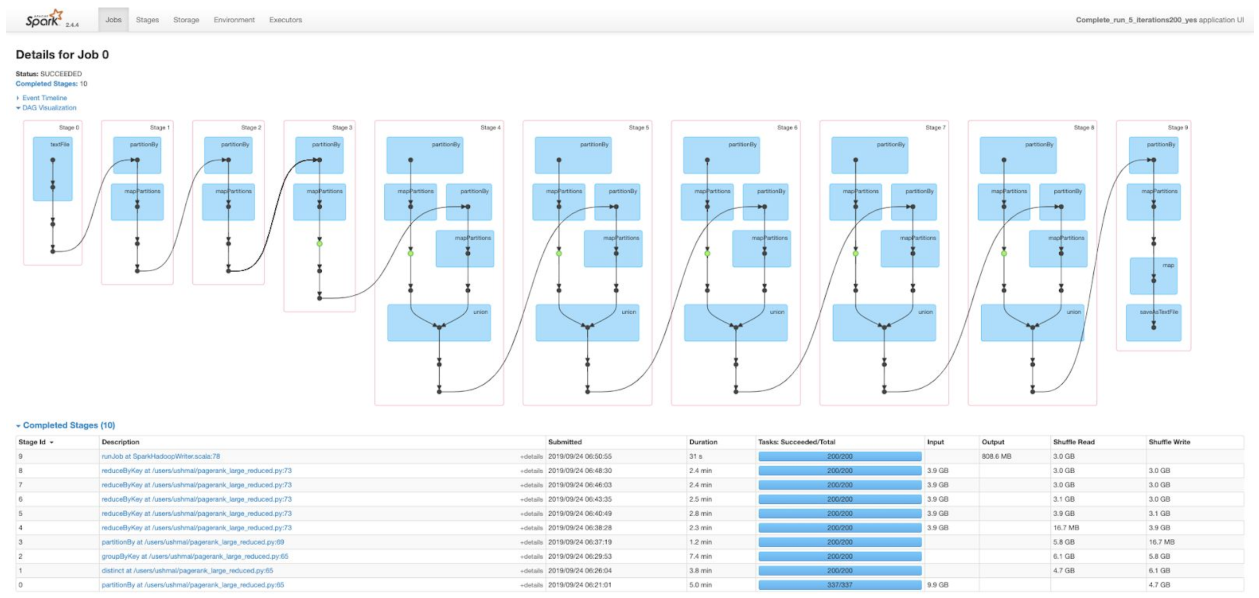


Figure 2 - Lineplot showing the trend in completion times based on partitions and persistence

3.3.3 Lineage Graph Visualization

The spark history server logs also save the lineage graphs for viewing later. These graphs are not visible on the master node web url after the job has finished and are only visible on the job url while the job is running. Following is a visualization of a typical lineage graph from one of our experiments runs.



Complete_run_5_iterations200_yes application UI

Details for Job 0

Status: SUCCEEDED
Completed Stages: 10

Event Timeline
DAG Visualization

Completed Stages (10)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
9	runJob at SparkHadoopWriter.scala:78	+details	2019/09/24 06:50:55	31 s	200/200		808.6 MB	3.0 GB	
8	reduceByKey at /users/ushmal/pagerank_large_reduced.py:73	+details	2019/09/24 06:48:30	2.4 min	200/200	3.9 GB		3.0 GB	3.0 GB
7	reduceByKey at /users/ushmal/pagerank_large_reduced.py:73	+details	2019/09/24 06:46:03	2.4 min	200/200	3.9 GB		3.0 GB	3.0 GB
6	reduceByKey at /users/ushmal/pagerank_large_reduced.py:73	+details	2019/09/24 06:43:35	2.5 min	200/200	3.9 GB		3.1 GB	3.0 GB
5	reduceByKey at /users/ushmal/pagerank_large_reduced.py:73	+details	2019/09/24 06:40:49	2.8 min	200/200	3.9 GB		3.9 GB	3.1 GB
4	reduceByKey at /users/ushmal/pagerank_large_reduced.py:73	+details	2019/09/24 06:38:28	2.3 min	200/200	3.9 GB		16.7 MB	3.9 GB
3	partitionBy at /users/ushmal/pagerank_large_reduced.py:69	+details	2019/09/24 06:37:19	1.2 min	200/200			5.8 GB	16.7 MB
2	groupByKey at /users/ushmal/pagerank_large_reduced.py:65	+details	2019/09/24 06:29:53	7.4 min	200/200			6.1 GB	5.8 GB
1	distinct at /users/ushmal/pagerank_large_reduced.py:65	+details	2019/09/24 06:26:04	3.8 min	200/200			4.7 GB	6.1 GB
0	partitionBy at /users/ushmal/pagerank_large_reduced.py:65	+details	2019/09/24 06:21:01	5.0 min	337/337	9.9 GB			4.7 GB

Figure 3- DAG Visualization for 200 partition pagerank job

As is seen in Figure 3, the DAG clearly shows the tasks involved in each stage of the spark Job. The five iterative reduceBykey stages are also easily identifiable in the DAG. Also, Modification 3 in our code gets rid of leftOuterJoin and adds a new stage for the distinct operation(Stage 1 in the figure).

3.3.3 Observations:

1. *Optimal number of partitions* : We observed that as the number of partitions becomes close to 2 times the available executor cores, we reach an optimal trade-off between storage utilization and application duration.

2. *Impact on application duration:* As the number of partitions is increased, it takes a shorter time to finish execution. This is expected, because with more partitions the scheduler can schedule those partitions which are ready to process the data while the other partitions are waiting for either reading/writing data from/to the disk.

3. *Impact on shuffle write during `reduceByKey()`:* We have observed that the choice of number of partitions only affects the shuffle data for reduceByKey() transformation. Larger number of partitions leads to more amount of data being shuffled. We believe this could be because a large number of partitions will lead to reduced amount of data on each partition and thereby requiring more shuffling to access the values corresponding to a key from other partitions. This trend is evident from the table 2 shown below.

No. of partitions	Duration(min)	reduceByKey Shuffle write (avg)
6	10	1.3 GB
10	6.5	1.5 GB
20	6.1	1.9 GB
50	5.5	2.2 GB
100	2.7	2.5 GB
200	2.5	3.0 GB

Table 2 : Effect of varying Partition levels on reduceByKey stages

4. *Read/Write bandwidth by Stage type:* The logs also provide information about Input data and Shuffle read/Write data for each stage. Table 3 shows the data for a particular experiment with 200 partitions with persistence caching and Table 4 shows the data without persistence caching. We can observe that shuffle write in a stage will be exactly equal to the Shuffle read in the next stage as we would expect it to be.

Stage No.	Stage type	Input (GB)	Shuffle write (GB)	Shuffle read (GB)
0	partitionBy()	9.9	4.7	0
1	distinct()	0	6.1	4.7
2	groupByKey()	0	5.8	6.1
3	partitionBy()	0	16.7 MB	5.8
4	reduceByKey()	3.9	3.9	16.7MB
5	reduceByKey()	3.9	3.1	3.9
6	reduceByKey()	3.9	3.0	3.1

Table 3: Shuffle Read/Write data for first 5 stages with Persistence on 200 partitions

Stage No.	Stage type	Input (GB)	Shuffle write (GB)	Shuffle read (GB)
0	partitionBy()	9.9	4.7	0
1	distinct()	0	6.1	4.7
2	groupByKey()	0	5.8	6.1
3	partitionBy()	0	19.9MB	5.8
4	reduceByKey()	0	3.9	19.9MB
5	reduceByKey()	0	3.1	3.9
6	reduceByKey()	0	3.0	3.1

Table 4: Shuffle Read/Write data for first 5 stages without Persistence on 200 partitions

5 Effect of Persistence on Stage completion time : For all the stages, the duration for completion improves when we persist `links` RDD. However, we observe an uptick in the completion time for the partitionBy stage where the RDD is first created. This is clear from the following table.

Stage	**Description**	**Time without Persistence**	**Time with persistence**
8	reduceByKey	5.9 min	4.8 min
7	reduceByKey	5.9 min	4.8 min
6	reduceByKey	6.0 min	4.9 min
5	reduceByKey	6.7 min	5.6 min
4	reduceByKey	7.6 min	6.1 min
3	**partitionBy**	**1.3 min**	**1.7 min**
2	groupByKey	13 min	13 min
1	distinct	8.1 min	8.3 min
0	**partitionBy**	**4.5 min**	**4.7 min**

Table 5 - Effect of Persistence on Stage Completion times

The bolded row shows that when the RDD is cached the Stage incurs additional time(1.7 minutes vs 1.3 minutes) to perform the computation and caching and thereafter all stages have reduced completion time because of the caching.

3.3.4 Test with Custom Partitioner :

In this test we chose to implement our own 'Custom Partitioner' to see the impact on performance. The custom key partitioner is just a function that hashes on the key top generate a partition number. The Following are our results:

Number of Partitions	**Type of Partitioner**	**Number of Tasks**	**Time Taken(min)**
200	Default(Hash)	2137	31
200	Custom Key Partitioner	7137	34

Table 6 - Results with Custom Partitioner

It is evident that the default partitioner performs better that our custom partitioner and is more optimized.

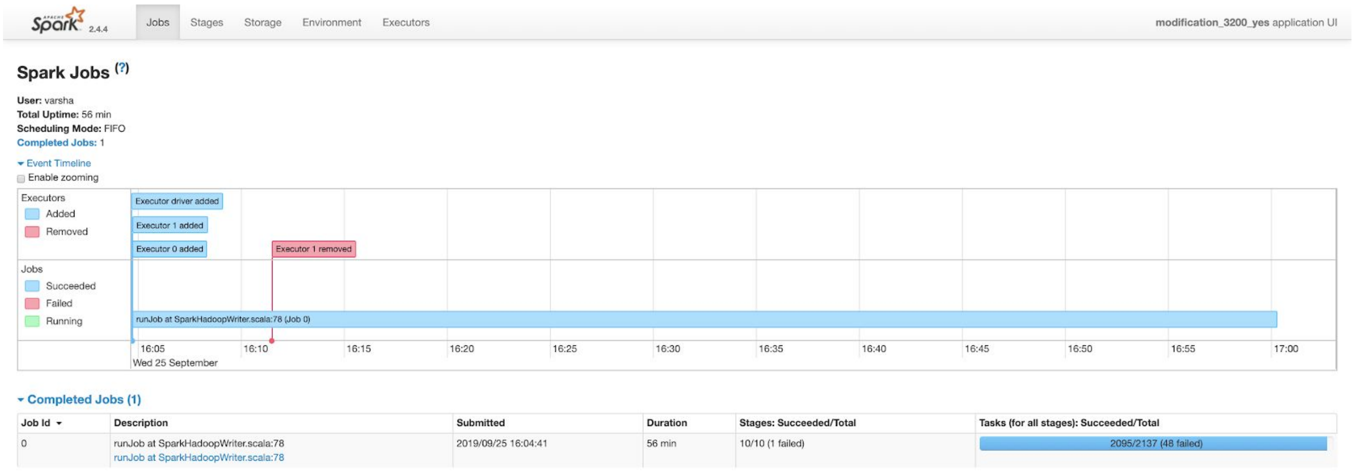
3.4 Effect of Killing Worker Processes

We studied the effect of killing a worker process on a typical run of a Spark Job. For this we chose the ‘Modification 3’ configuration of our code as discussed in section 3.2.

3.4.1 Worker Killed at 25%

We chose the 200 - partition job we ran earlier with persistence for this test. We calculated 25% of the lifetime of the process(Around 7 minutes) and killed one worker while the job was running as per the instructions. Following were our observations:

1. The failure was triggered in Executor 1 while it was running stage 1(distinct()) operation.
2. Upon detecting the failure, the spark master delegates the rest of the stages to the remaining worker. The remaining worker retries some of the tasks in both Stage 1 as well as stage 0 (Refer figure 4 below). This is because the Stage 0 data was also lost when the worker 1 was killed.
3. The remaining execution proceeds on the remaining worker until Stage 5 when the worker runs out of memory and the job is killed.
4. The job was completed by the lone worker and the completion time was *56 minutes* which is 25 minutes more than the time taken if no worker is killed.



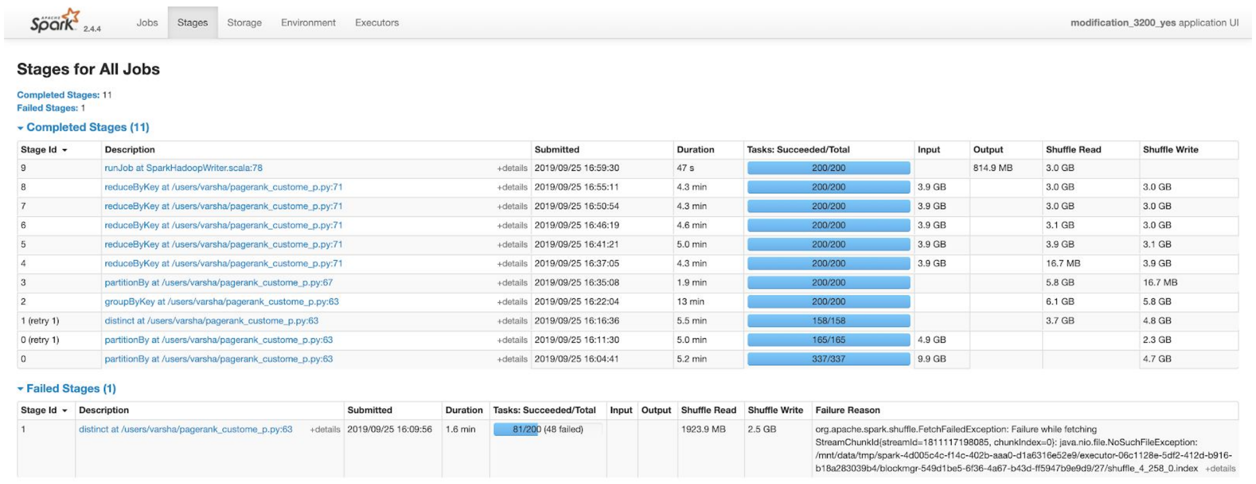
Spark Jobs (?)

User: varsha
Total Uptime: 56 min
Scheduling Mode: FIFO
Completed Jobs: 1

Completed Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	runJob at SparkHadoopWriter.scala:78 runJob at SparkHadoopWriter.scala:78	2019/09/25 16:04:41	56 min	10/10 (1 failed)	2095/2137 (48 failed)

Figure 4 - Event Timeline Showing the worker kill event



Stages for All Jobs

Completed Stages: 11
Failed Stages: 1

Completed Stages (11)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
9	runJob at SparkHadoopWriter.scala:78	+details	2019/09/25 16:59:30	47 s	200/200		814.9 MB	3.0 GB	
8	reduceByKey at /users/varsha/pagerank_custome_p.py:71	+details	2019/09/25 16:55:11	4.3 min	200/200	3.9 GB		3.0 GB	3.0 GB
7	reduceByKey at /users/varsha/pagerank_custome_p.py:71	+details	2019/09/25 16:50:54	4.3 min	200/200	3.9 GB		3.0 GB	3.0 GB
6	reduceByKey at /users/varsha/pagerank_custome_p.py:71	+details	2019/09/25 16:46:19	4.6 min	200/200	3.9 GB		3.1 GB	3.0 GB
5	reduceByKey at /users/varsha/pagerank_custome_p.py:71	+details	2019/09/25 16:41:21	5.0 min	200/200	3.9 GB		3.9 GB	3.1 GB
4	reduceByKey at /users/varsha/pagerank_custome_p.py:71	+details	2019/09/25 16:37:05	4.3 min	200/200	3.9 GB		16.7 MB	3.9 GB
3	partitionBy at /users/varsha/pagerank_custome_p.py:67	+details	2019/09/25 16:35:08	1.9 min	200/200			5.8 GB	16.7 MB
2	groupByKey at /users/varsha/pagerank_custome_p.py:63	+details	2019/09/25 16:22:04	13 min	200/200			6.1 GB	5.8 GB
1 (retry 1)	distinct at /users/varsha/pagerank_custome_p.py:63	+details	2019/09/25 16:16:36	5.5 min	158/158			3.7 GB	4.8 GB
0 (retry 1)	partitionBy at /users/varsha/pagerank_custome_p.py:63	+details	2019/09/25 16:11:30	5.0 min	165/165	4.9 GB			2.3 GB
0	partitionBy at /users/varsha/pagerank_custome_p.py:63	+details	2019/09/25 16:04:41	5.2 min	337/337	9.9 GB			4.7 GB

Failed Stages (1)

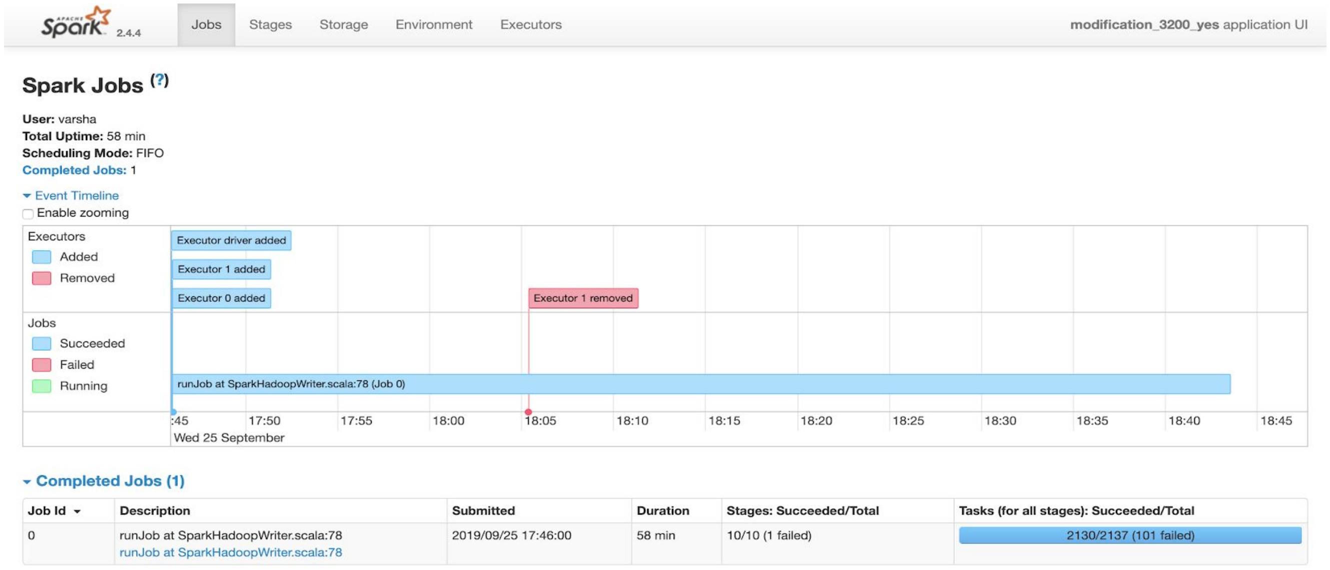
Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write	Failure Reason
1	distinct at /users/varsha/pagerank_custome_p.py:63	+details	2019/09/25 16:09:56	1.6 min	81/200 (48 failed)			1923.9 MB	2.5 GB	org.apache.spark.shuffle.FetchFailedException: Failure while fetching StreamChunkId{streamId=1811117198085, chunkIndex=0}: java.nio.file.NoSuchFileException: /mnt/data/tmp/spark-4d005c4c-f14c-402b-aaa0-d1a6316e52e9/executor-06c1128e-5df2-412d-b916-b18a283039b4/blockmgr-549d1be5-6f36-4a67-b43d-ff5947b9e9d9/27/shuffle_4_258_0.index +details

Figure 5: Screenshot showing retried stages after one of the worker is killed

3.4.2 Worker Killed at 75%

We reran the job and after the job reached 75% of its completion time, we again killed Executor1. Following are our observations:

1. The Executor 1 was running Stage 4, a reduceByKey operation when it was killed.
2. This prompted the driver to retry some tasks from all previous stages(0,1,2 and 3) to recover data that was stored in worker1.
3. Not all tasks from previous stages were retried.
4. The lone worker then continued the job and the completion time was 58 minites which is quite close to the time taken after worker 1 was killed at 25%.



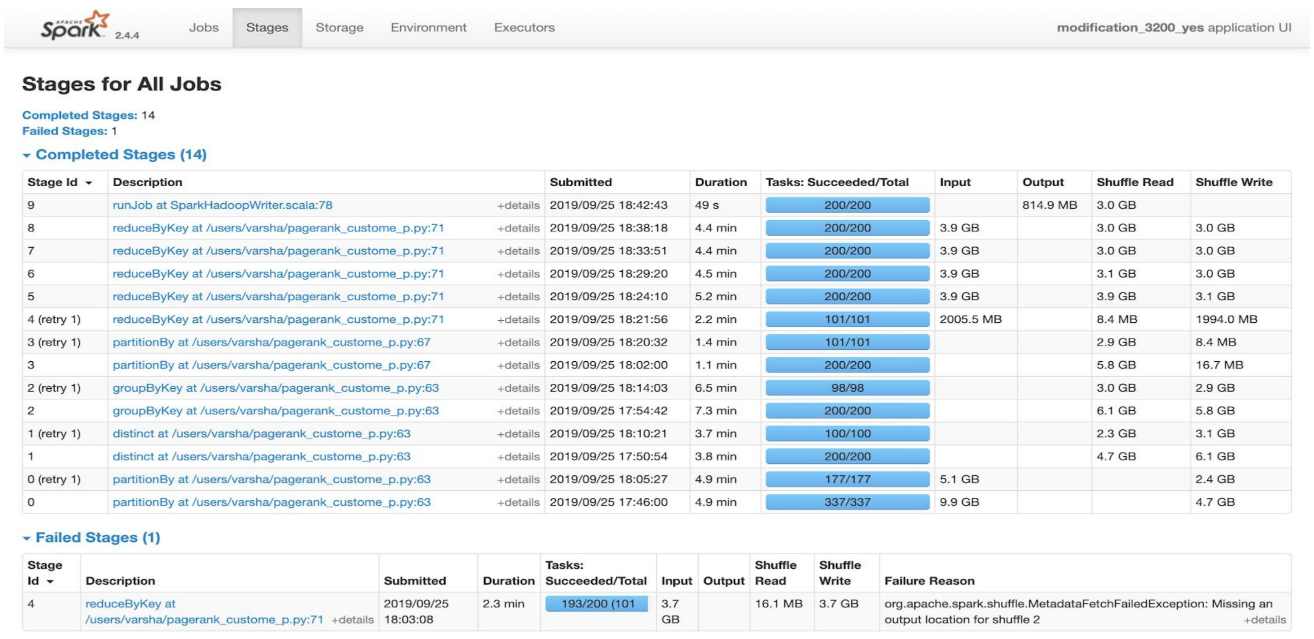
Spark Jobs (?)

User: varsha
Total Uptime: 58 min
Scheduling Mode: FIFO
Completed Jobs: 1

Completed Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	runJob at SparkHadoopWriter.scala:78 runJob at SparkHadoopWriter.scala:78	2019/09/25 17:46:00	58 min	10/10 (1 failed)	2130/2137 (101 failed)

Figure 6 Event Timeline Showing the worker kill event



Stages for All Jobs

Completed Stages: 14
Failed Stages: 1

Completed Stages (14)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
9	runJob at SparkHadoopWriter.scala:78	+details	2019/09/25 18:42:43	49 s	200/200		814.9 MB	3.0 GB	
8	reduceByKey at /users/varsha/pagerank_custome_p.py:71	+details	2019/09/25 18:38:18	4.4 min	200/200	3.9 GB		3.0 GB	3.0 GB
7	reduceByKey at /users/varsha/pagerank_custome_p.py:71	+details	2019/09/25 18:33:51	4.4 min	200/200	3.9 GB		3.0 GB	3.0 GB
6	reduceByKey at /users/varsha/pagerank_custome_p.py:71	+details	2019/09/25 18:29:20	4.5 min	200/200	3.9 GB		3.1 GB	3.0 GB
5	reduceByKey at /users/varsha/pagerank_custome_p.py:71	+details	2019/09/25 18:24:10	5.2 min	200/200	3.9 GB		3.9 GB	3.1 GB
4 (retry 1)	reduceByKey at /users/varsha/pagerank_custome_p.py:71	+details	2019/09/25 18:21:56	2.2 min	101/101	2005.5 MB		8.4 MB	1994.0 MB
3 (retry 1)	partitionBy at /users/varsha/pagerank_custome_p.py:67	+details	2019/09/25 18:20:32	1.4 min	101/101			2.9 GB	8.4 MB
3	partitionBy at /users/varsha/pagerank_custome_p.py:67	+details	2019/09/25 18:02:00	1.1 min	200/200			5.8 GB	16.7 MB
2 (retry 1)	groupByKey at /users/varsha/pagerank_custome_p.py:63	+details	2019/09/25 18:14:03	6.5 min	98/98			3.0 GB	2.9 GB
2	groupByKey at /users/varsha/pagerank_custome_p.py:63	+details	2019/09/25 17:54:42	7.3 min	200/200			6.1 GB	5.8 GB
1 (retry 1)	distinct at /users/varsha/pagerank_custome_p.py:63	+details	2019/09/25 18:10:21	3.7 min	100/100			2.3 GB	3.1 GB
1	distinct at /users/varsha/pagerank_custome_p.py:63	+details	2019/09/25 17:50:54	3.8 min	200/200			4.7 GB	6.1 GB
0 (retry 1)	partitionBy at /users/varsha/pagerank_custome_p.py:63	+details	2019/09/25 18:05:27	4.9 min	177/177	5.1 GB			2.4 GB
0	partitionBy at /users/varsha/pagerank_custome_p.py:63	+details	2019/09/25 17:46:00	4.9 min	337/337	9.9 GB			4.7 GB

Failed Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write	Failure Reason
4	reduceByKey at /users/varsha/pagerank_custome_p.py:71 +details	2019/09/25 18:03:08	2.3 min	193/200 (101	3.7 GB		16.1 MB	3.7 GB	org.apache.spark.shuffle.MetadataFetchFailedException: Missing an output location for shuffle 2 +details

Figure 7 Stages retried after worker was killed at 75%

Experiment	Time to Completion
Both workers functional	31
Worker 1 killed at 25%	56
Worker 1 killed at 75%	58

Table 7 Summary of worker kill experiments

3.4.3 Conclusions

From our experiments in killing spark workers, we conclude the following:

a) As expected, the completion time after killing a worker goes up from the normal operation.
b) Killing a worker at 25% and at 75% did not drastically affect the final completion time. Table 7 summarizes the results. In both cases the job was completed by the lone worker in around an hour.
c) After a worker is killed, the tasks for previous stages are retried, however, only some of the tasks are retried. For example, for Stage 1, the first successful pass by both workers had 200 tasks, however, after the worker 1 was killed during Stage 4 , only 100 of the 200 tasks were retried(Refer Figure). This shows that the only the data lost by the killed worker is restored and spark uses lineage to recover and reconstruct the lost RDD in the remaining worker.