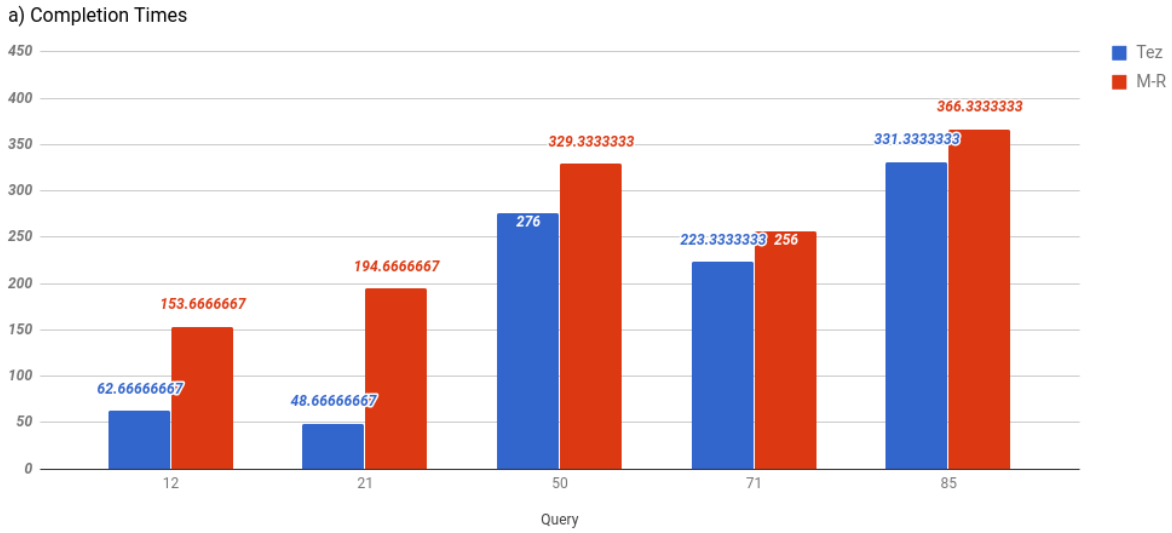
Group 23 Part A

Authors – Rohit Damkondwar, Tarun Bansal, Pavan Kemparaju

1a.

Query	Tez	M-R
12	62.66666667	153.6666667
21	48.66666667	194.6666667
50	276	329.3333333
71	223.3333333	256
85	331.3333333	366.3333333



Tez is always faster than Map reduce.

Due to the architecture of Tez the job's steps are computed before execution and the system can cache intermediate job results in memory. But, in MapReduce all intermediate data between MapReduce phases are written to HDFS i.e. disk adding latency.

Part 1b:
Network bandwidth in bytes

Tez		Avg
	12	6,361,292,950,666,670,000
	21	1,157,242,779,666,670,000
	50	24,489,100,656,333,300,000
	71	35,492,911,377,000,000,000
	85	18,967,835,682,666,700,000
MR	12	8,009,428,532,500,000,000
	21	6,430,727,694,333,330,000
	50	30,626,058,147,666,700,000
	71	35,426,918,164,000,000,000
	85	14,537,329,973,000,000,000

Disk bandwidth:

Query	TEZ	MR
12	418181120	5736886272
21	22323200	4653715456
50	8594477056	19173908480
71	69925629952	10406600704
85	122368	7858442240

Part 1c:

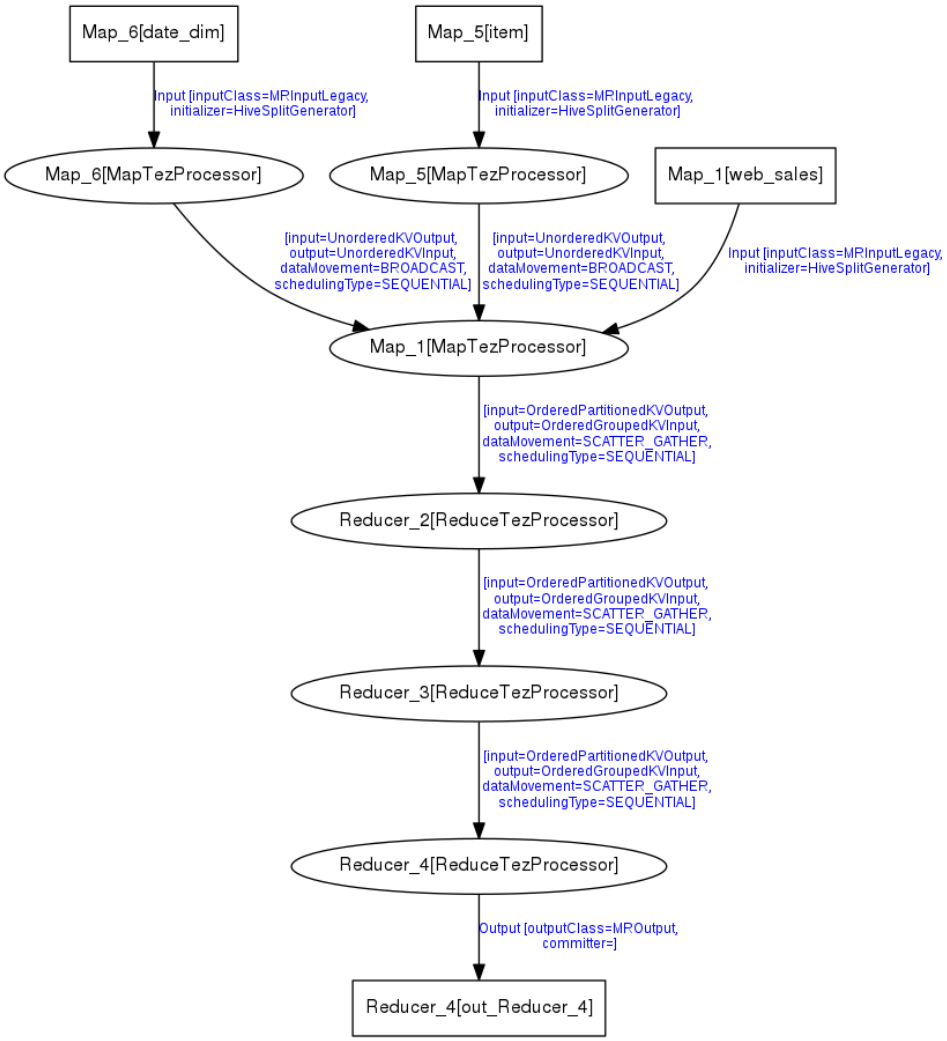
Tasks		Aggregate(red)	Read from HDFS(map)	Ratio	Total
MR	12	3	39	0.07692307692	42
	21	2	20	0.1	22
	50	2	98	0.02040816327	100
	71	2	173	0.01156069364	175
	85	8	51	0.1568627451	59

Tez	12	7	3	2.333333333	10
	21	7	4	1.75	11
	50	9	5	1.8	14
	71	8	14	0.5714285714	22
	85	8	14	0.5714285714	22

Part 1D:

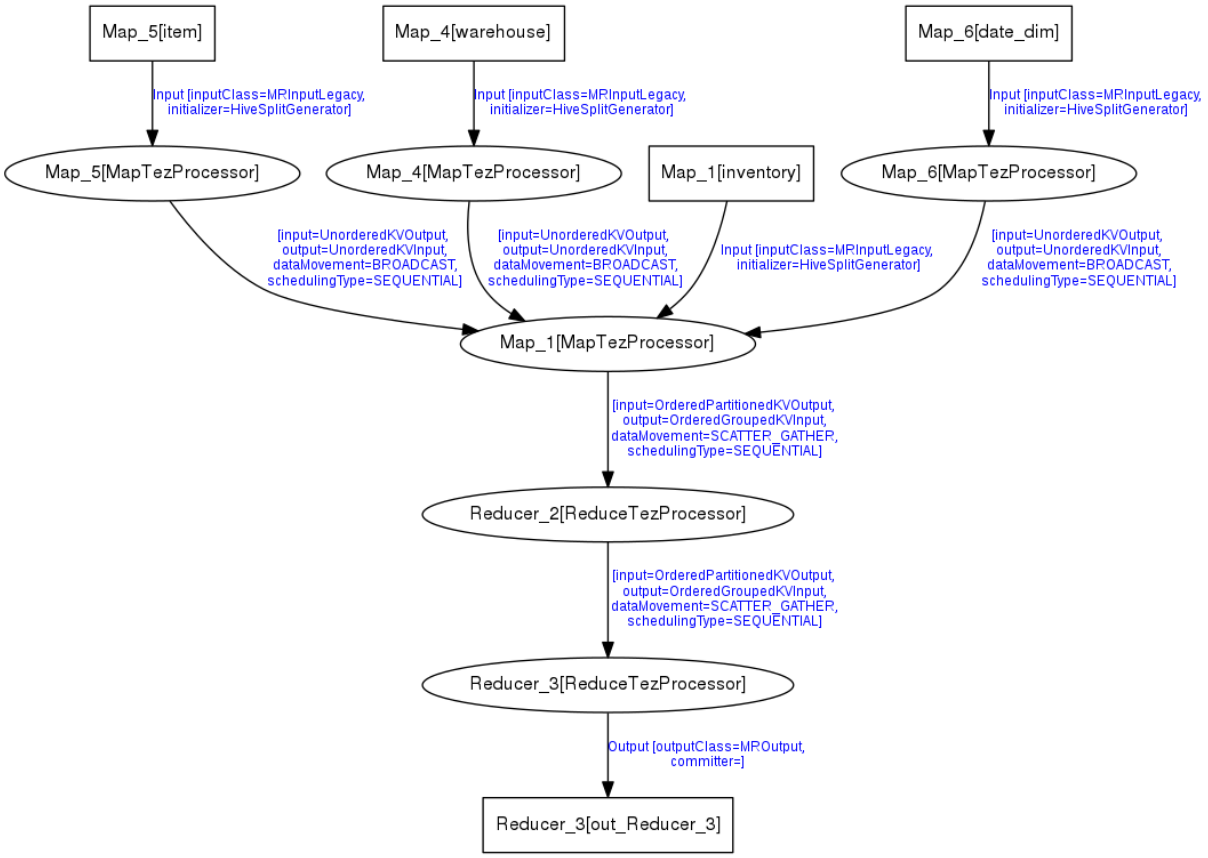
Tez:

Query 12:



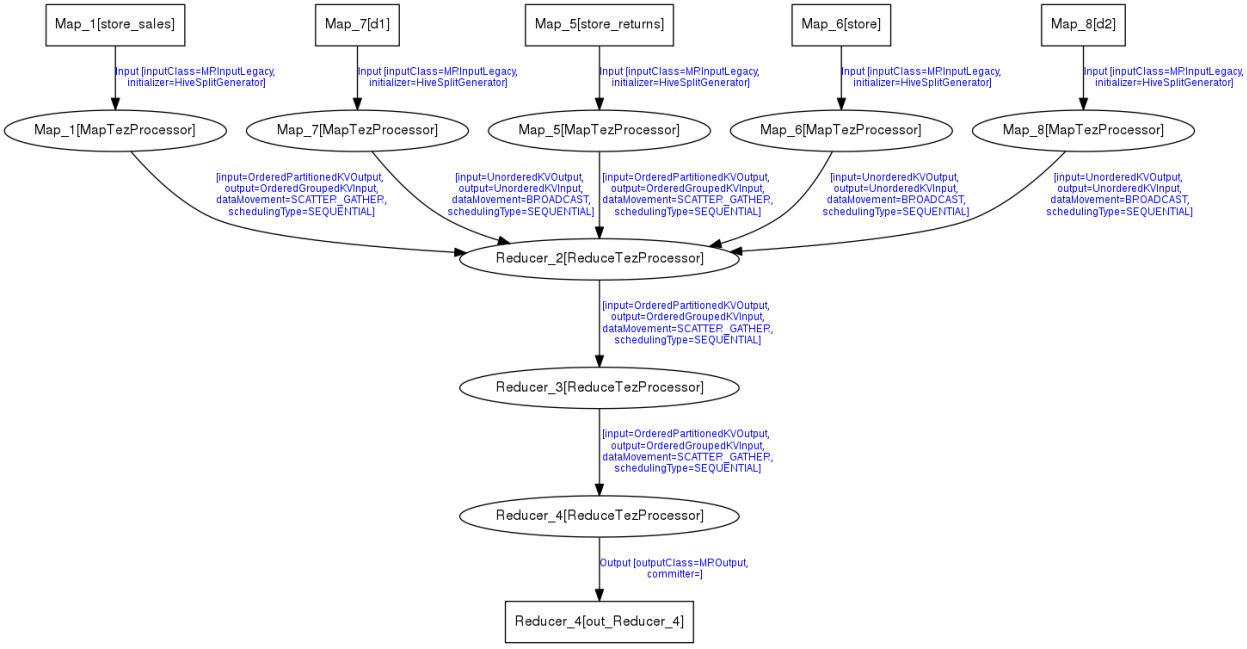
ubuntu_20170929224946_4ddfe079_b48e_43ea_a622_86af9c16cb3e_1

Query 21:



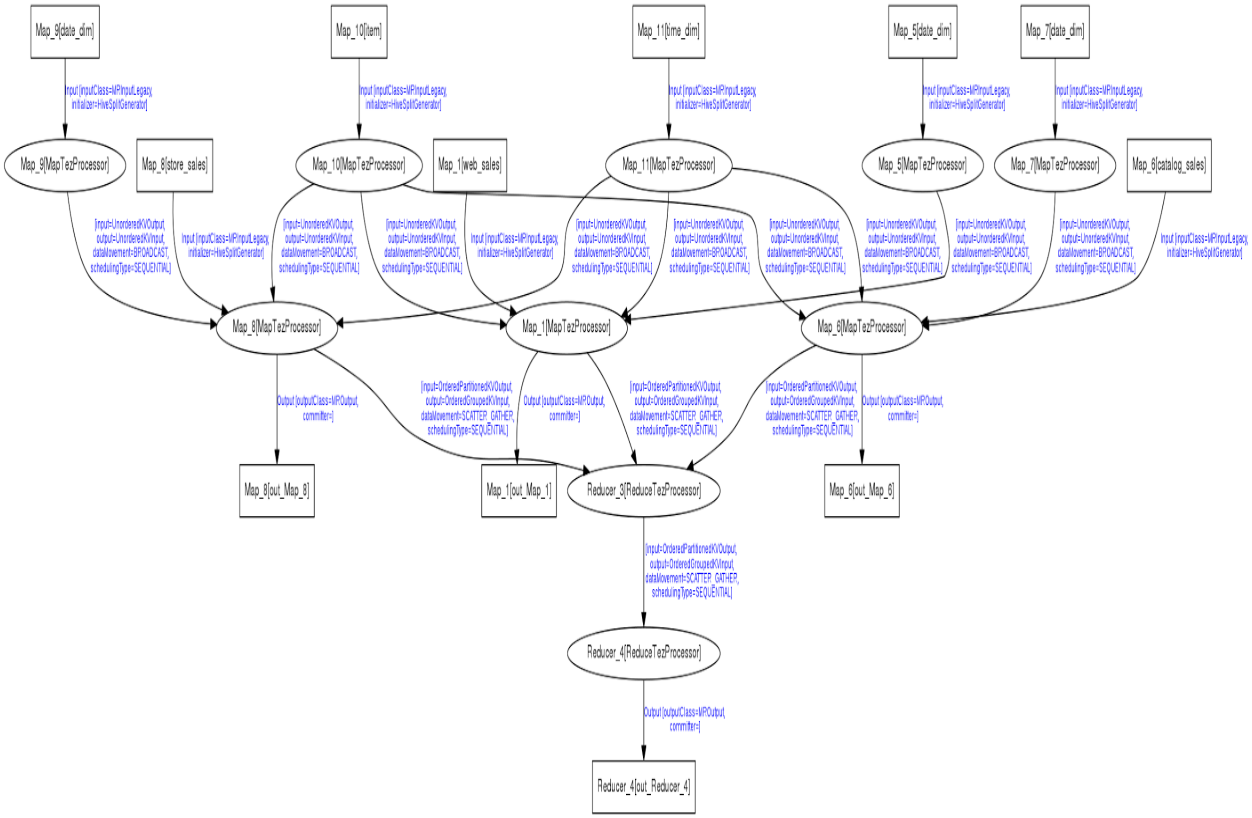
ubuntu_20170929230408_8559a1d0_97bd_47e1_ad0e_4f038dac4393_1

Query 50



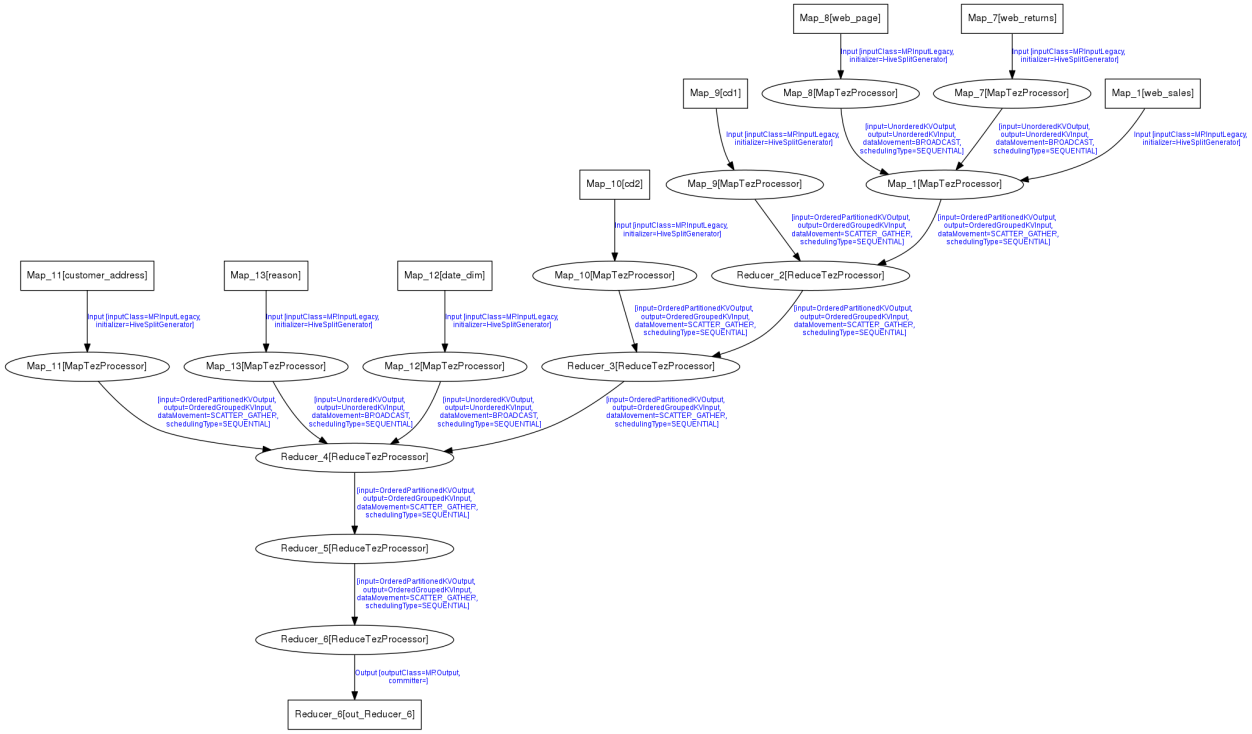
ubuntu_20170930035816_2f1a22e2_1100_4b1a_8422_172470fde80f_1

Query 71:



ubuntu_20170930041441_5fbae3d1_c017_49ac_a0ef_989e3a52aa57_1

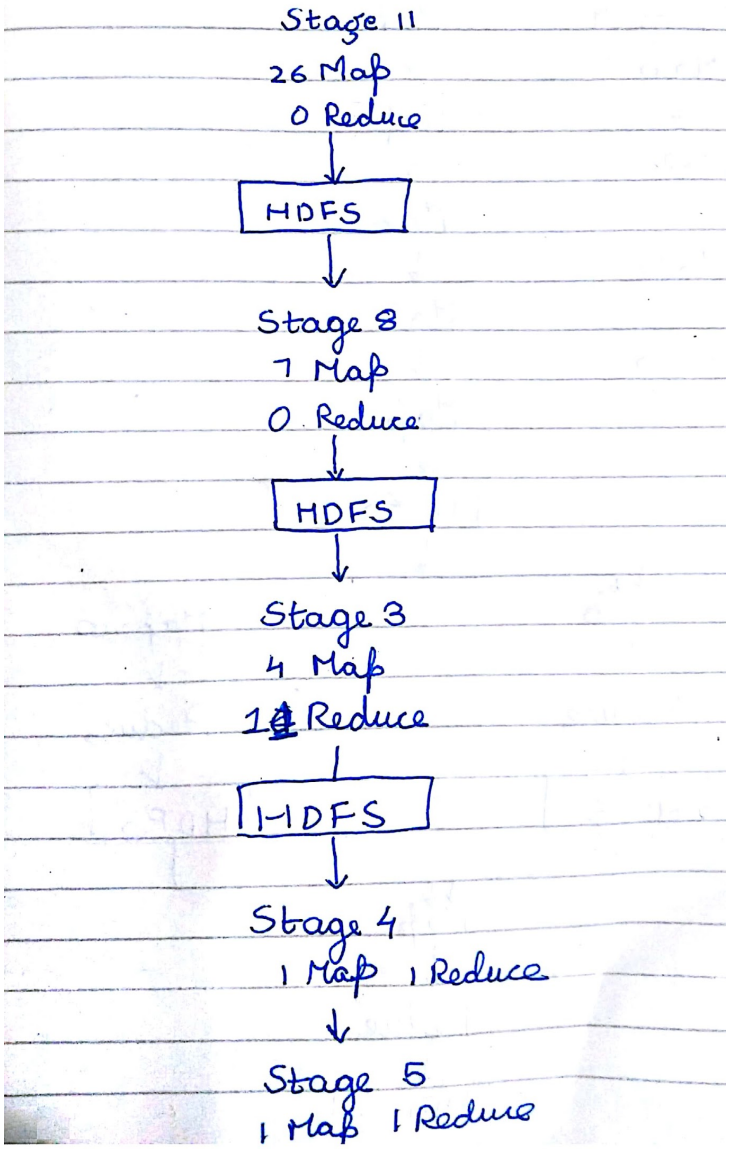
Query 85:



ubuntu_20170930042057_11aed344_655f_473d_89d4_0a61716921cd_1

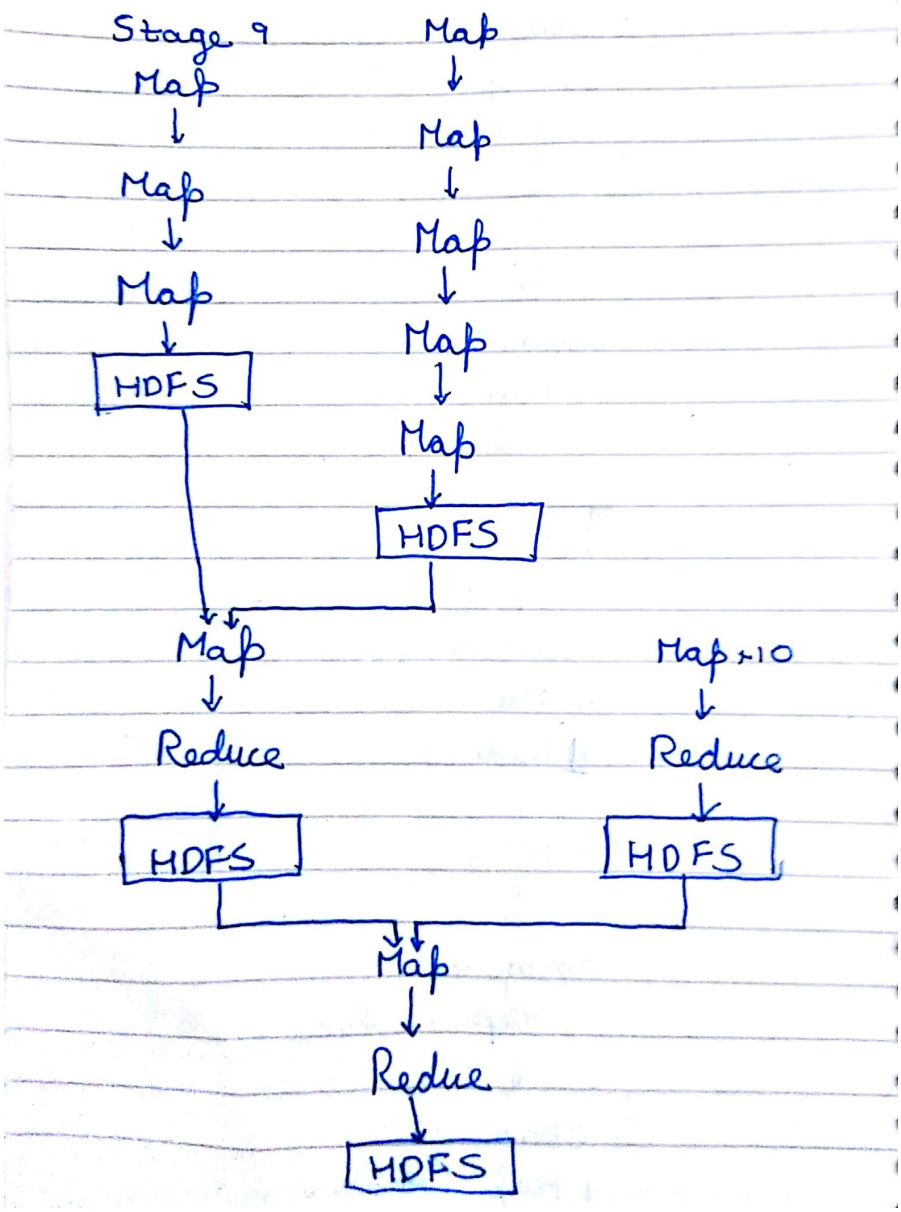
Query 12:

12

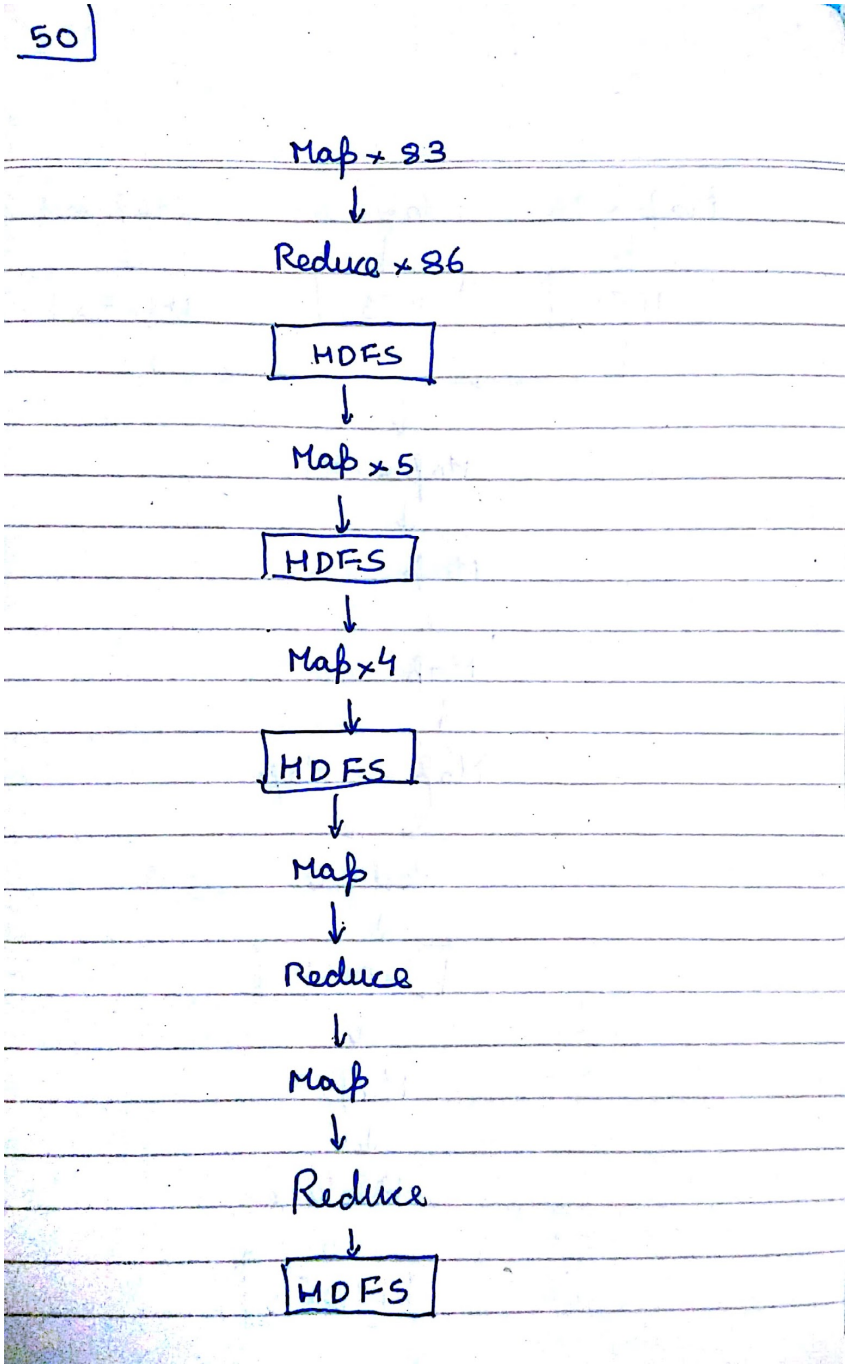

Stage 11
26 Map
0 Reduce
HDFS
Stage 8
7 Map
0 Reduce
HDFS
Stage 3
4 Map
1 Reduce
HDFS
Stage 4
1 Map 1 Reduce
Stage 5
1 Map 1 Reduce

Query 21:

21)

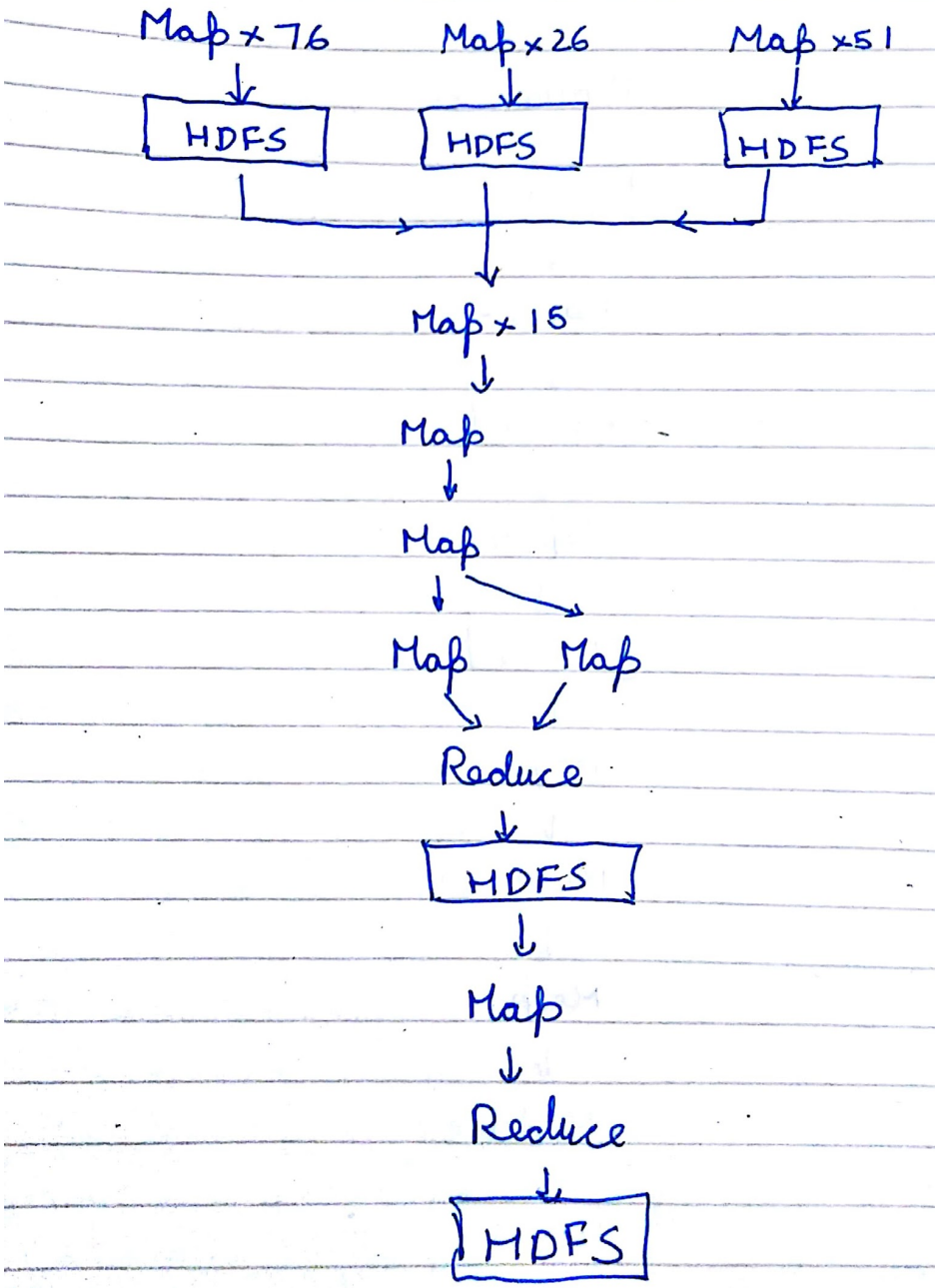

Stage 9
Map
Map
Map
HDFS
Map
Map
Map
Map
Map
HDFS
Map
Reduce
HDFS
Map x10
Reduce
HDFS
Map
Reduce
HDFS

Query 50:

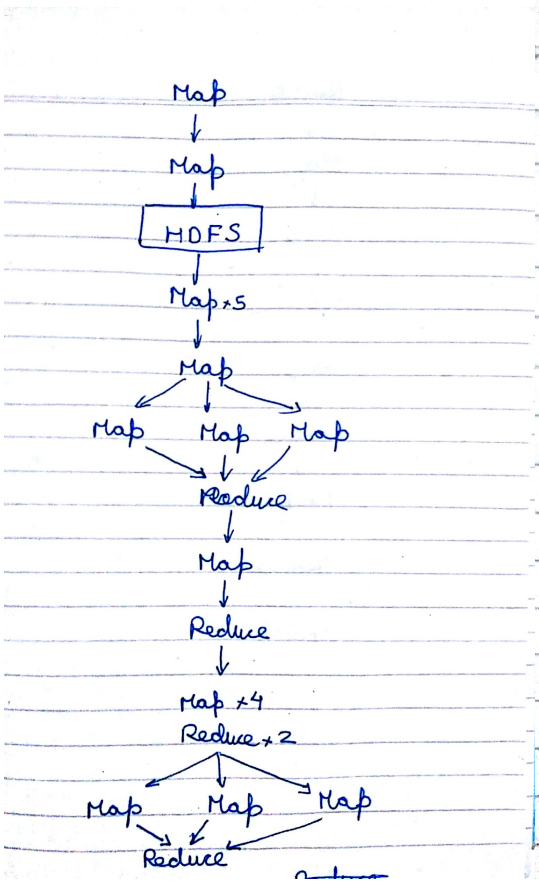

50
Map x 83
Reduce x 86
HDFS
Map x 5
HDFS
Map x 4
HDFS
Map
Reduce
Map
Reduce
HDFS

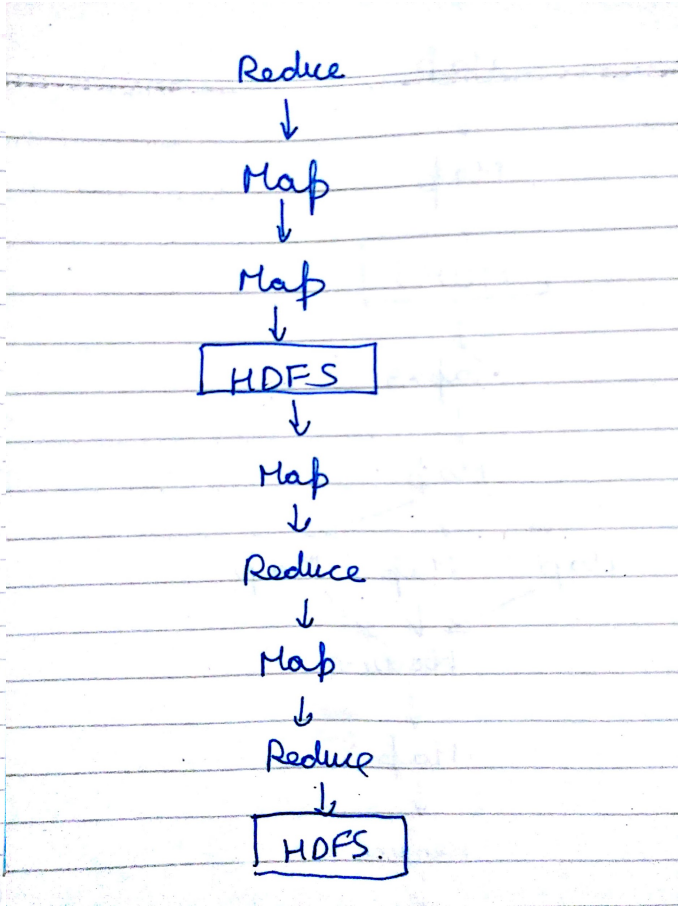
Query 71:

7.)


Map x 76
Map x 26
Map x 51
HDFS
HDFS
HDFS
Map x 15
Map
Map
Map
Map
Reduce
HDFS
Map
Reduce
HDFS

Query 85:





Part 2:

Percent at which data node is killed vs Execution time

Framework		Execution time
Tez	Normal	220
	Killed at 25%	216
	Killed at 75%	225
MR	Normal	317
	Killed at 25%	320
	Killed at 75%	350

We observe a small but proportionally insignificant increase in execution times when 1 datanode is killed at various times.
This is due to two reasons:

1. Replication factor of 2 implies there are multiple copies of the data from the failing data node. Thus the framework does not lose any necessary inputs
2. Speculative execution: Hadoop schedules redundant copies of the same task. So if the tasks fail at one node, there are redundant executions of the same whose output can be used.

Group 23 Spark Part C
Authors – Rohit Damkondwar, Tarun Bansal, Pavan Kemparaju

Question 1. Write a Scala/Python/Java Spark application that implements the PageRank algorithm without any custom partitioning (RDDs are not partitioned the same way) or RDD persistence. Your application should utilize the cluster resources to it's full capacity. Explain how did you ensure that the cluster resources are used efficiently. (Hint: Try looking at how the number of partitions of a RDD play a role in the application performance)

Solution 1: We had 5 workers in the cluster. To utilize the cluster to its full capacity, RDD should have at least 5 partitions. Every partition is executed by a task. We got least running time with number of Partitions = 50.

Number of tasks created = 751

Following are the IO and Network stats for 4 runs of Spark App(Spark driver and 5 workers):

Time	IO_Read (in KB)	IO_Write (in KB)	Net_Send (in B)	Net_Recv (in KB)
51	384296	89216	89216	184116168
66	371128	57036	57264	211206387
66	371860	158356	158760	209757881
55	374136	152496	152496	210491896

Average:
Completion Time=59.5s
IO_Read = 362.92 MB
IO_Write = 133.89 MB
Net_send = 134.68 KB
Net_recv = 603.78 MB

The above stats are for entire cluster (1 Driver + 5 Workers).

Question 2. Modify the Spark application developed in Question 1 to implement the PageRank algorithm with appropriate custom partitioning. Is there any benefit of custom partitioning? Explain. (Hint: Do not assume that all operations on a RDD preserve the partitioning)

Solution2:
We have used RangePartitioning to replace Spark's Default partitioning scheme. We also considered HashPartitioning but found that Range Partitioning offered better performance.
Number of partitions = 50
Number of tasks created = 751

Time	IO_Read	IO_Write	Net_Send	Net_Recv
59	373264	206080	206080	861086615
59	371412	99320	181536	889569678
56	372580	160452	251864	874254597
52	373212	70656	70656	882879520

Average:
Completion Time=56.5s

IO_Read = 363.88 MB
IO_Write = 130.98 MB
Net_send = 173.37 KB
Net_recv = 836.32 MB

Why partitioning increased the Job turn-around time?
The Spark App with default partitioning took 59.5 sec(average) to calculate all the Stages.

The Spark App with Range partitioning had 1 extra job of creating Partitioner (Stage 0) which took 8 sec. But, the core computation of Page Rank stages was completed in 39sec!

The RangePartitioning improved the completion time by 3sec.

Question 3. Extend the Spark application developed in Question 2 to leverage the flexibility to persist the appropriate RDD as in-memory objects. Is there any benefit of persisting RDDs as in-memory objects in the context of your application? Explain.

With respect to Question 1-3, for your report you should:
Report the application completion time under the three different scenarios.
Compute the amount of network/storage read/write bandwidth used during the application lifetime under the four different scenarios.
Compute the number of tasks for every execution.
Present / reason about your findings and answer the above questions. Apart from that you should compare the applications in terms of the above metrics and reason out the difference in performance, if any.

Number of partitions = 50
Number of tasks created = 751

Time	IO_Read	IO_Write	Net_Send	Net_Recv
43	372908	14528	15136	489547063
39	369840	14144	14604	534893531
39	371912	120292	120292	528281961
40	371008	29844	30124	554625993

Average:
Completion Time=40.25s
IO_Read = 362.71 MB
IO_Write = 43.65 MB
Net_send = 43.98 KB
Net_recv = 502.43 MB

Few important RDD's which were being re-used are now persisted in Spark with persistence level of MEMORY_ONLY via api 'cache()'.
The persisted RDDs are being re-used and hence, we got speed up in the job execution.

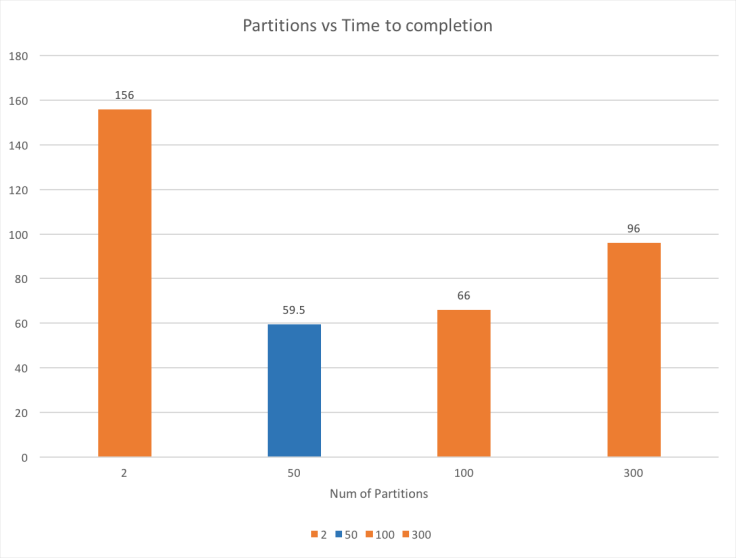
Question 4. Analyze the performance of CS-744-Assignment1-PartC-Question1 by varying the number of RDD partitions from 2 to 100 to 300. Does increasing the number of RDD partitions always help? If no, could you find a value where it has a negative impact on performance and reason about the same.

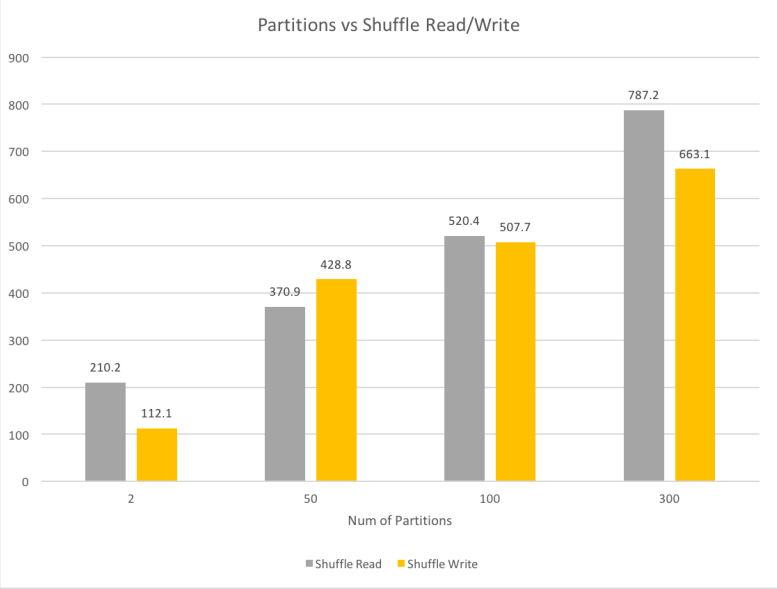
Solution4:
We ran PartC-1 Spark App (with default partitioning) for 4 times with 2,50,100 and 300 partitions.

Partitions	Time (in sec)	Shuffle Read (in MB)	Shuffle Write (in MB)
2	156	210.2	112.1
50	59.5	370.9	428.8
100	66	520.4	507.7
300	96	787.2	663.1

For partitions=2, the cluster was being under-utilized as only 2 tasks were being spawned. Also, since only 2 tasks shared the load, the task size was high. As a result, the Spark Worker Garbage collection time was high.

For partitions=100 and 300, the overhead of Shuffle read and write beats the benefits of data parallelism. We got the best performance for 50 partitions.

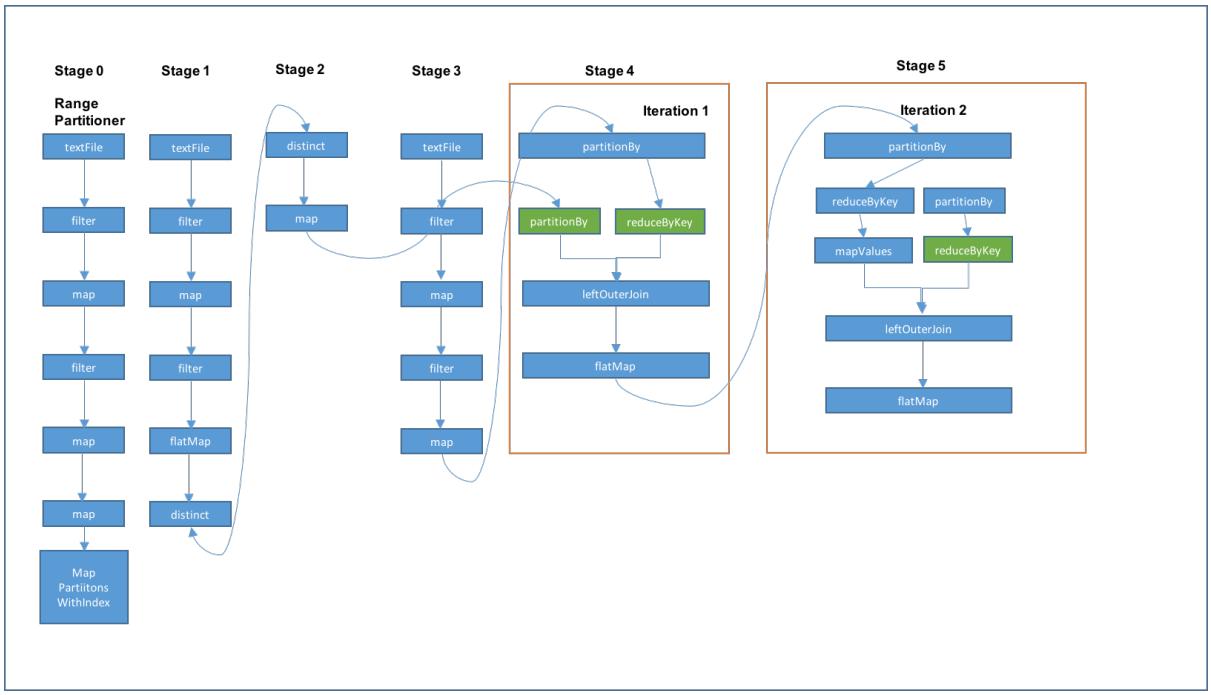




Thus, increasing number of partitions to a large number doesn't benefit the turn-around time for the job execution.

Question 5. Visually plot the lineage graph of the CS-744-Assignment1-PartC-Question3 application. Is the same lineage graph observed for all the applications developed by you? If yes/no, why? The Spark UI does provide some useful visualizations.

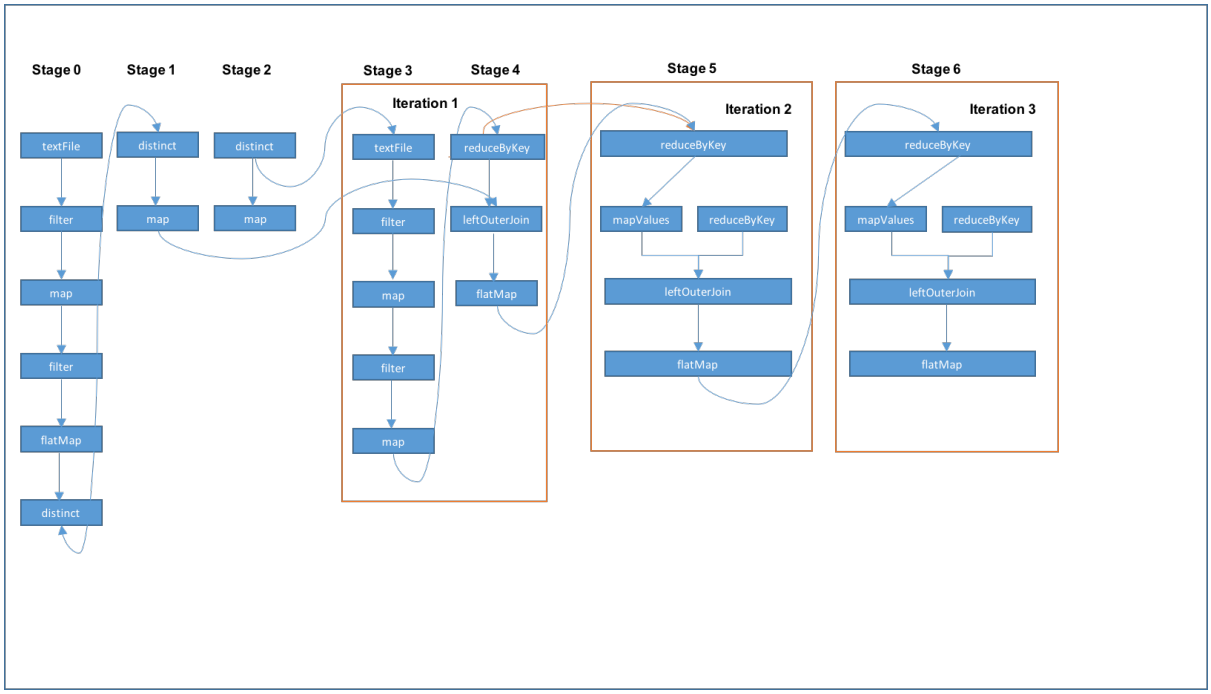
Solution 5:



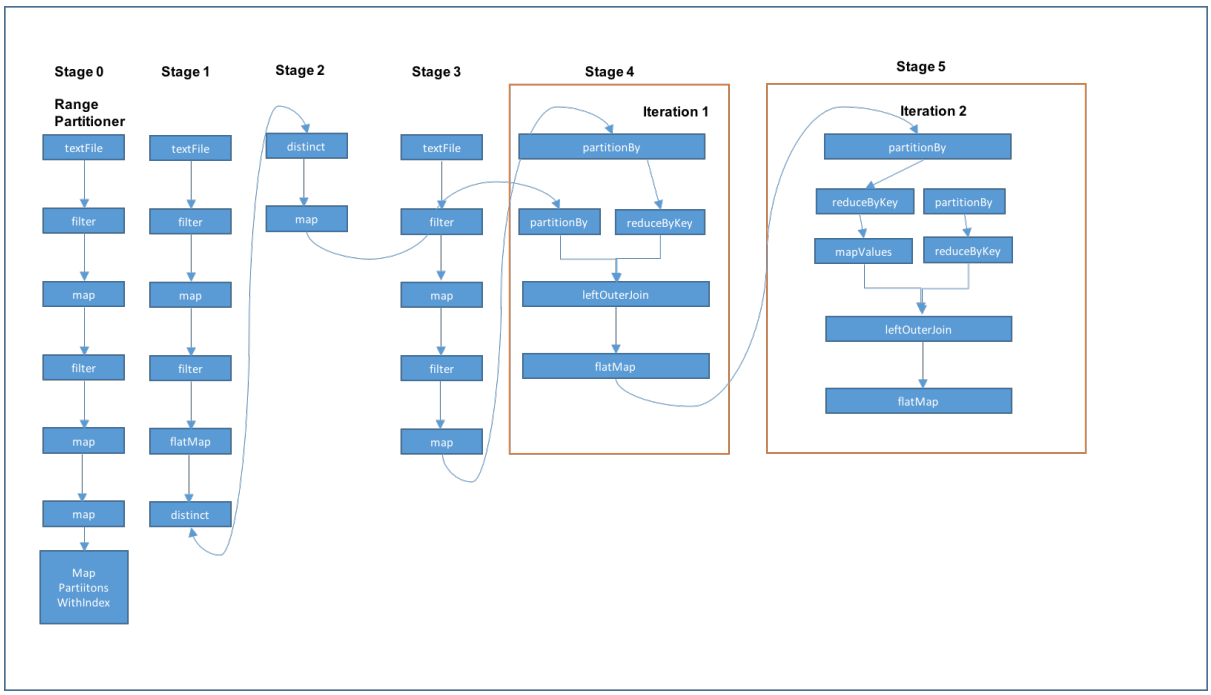
Remaining iterations are not shown in the graph. The Stage 0 represents Range Partitioner computation. Stages 1-3 represent data cleaning (Eliminate irrelevant lines), rank initialization (= 1), etc. Stages 4 and 5 represent iterations of Page Rank.
The lineage graphs for PartC2 and PartC3 are almost same as both employ same partitioning technique. PartC3 persists some of the crucial RDDs and hence, the green highlights represent cached stages.
PartC1 uses default partitioning and hence, its Lineage graph is slightly different.

Question 6. Visually plot the Stage-level Spark Application DAG (with the appropriate dependencies) for all the applications developed by you till the second iteration of PageRank. The Spark UI does provide some useful visualizations. Is it the same for all the applications? Id yes/no, why? What impact does the DAG have on the performance of the application?

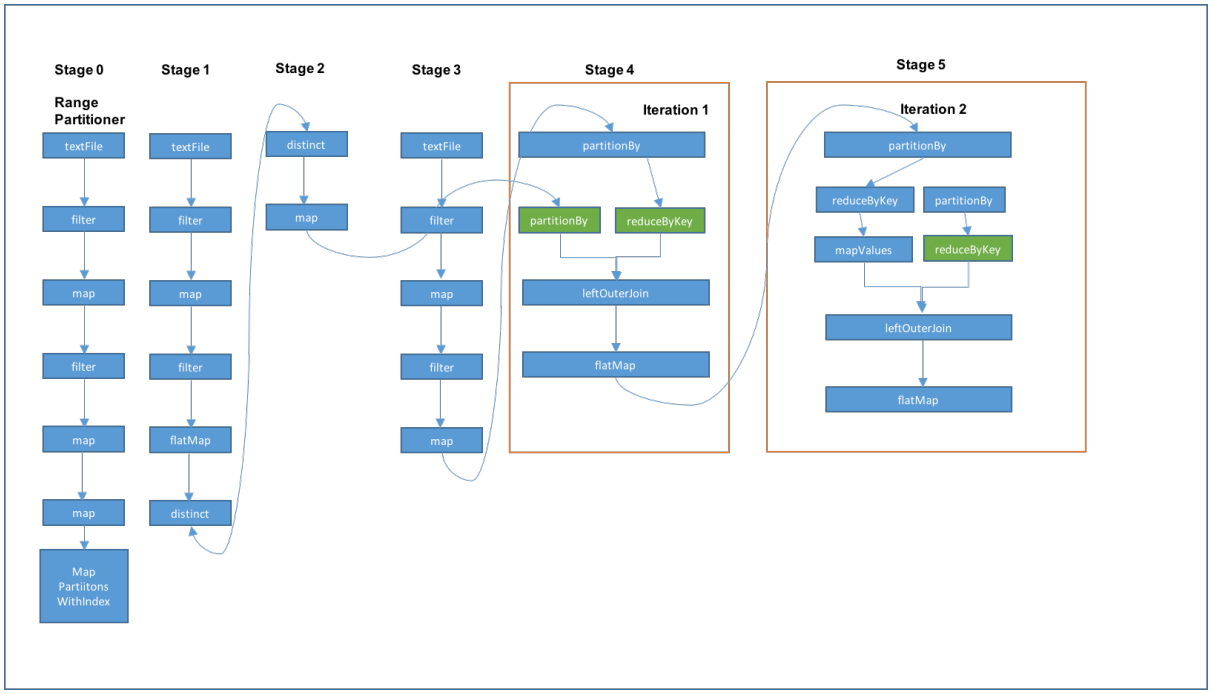
DAG for PartC1:



DAG for PartC2:



DAG for PartC3:



DAGs for Parts C2 and C3 are almost same. PartC3 uses RDD persistence (in Memory) and hence the green boxes. PartC1 uses different partitioning technique. Hence, Part C1 doesn't have Partitioner stage and PartitionBy stages in each iteration.

Question 7. Analyze the performance of CS-744-Assignment1-PartC-Question3 and CS-744-Assignment1-PartC-Question1 in the presence of failures. For each application, you should trigger two types of failures on a desired Worker VM when the application reaches 25% and 75% of its lifetime.

Solution 7:
All the metrics are in seconds.

	Part C1	**Recovery Time**	**Part C3**	**Recovery Time**
Without Failure	59.5		40.25	
25%	78	18.5	54	13.75
75%	90	30.5	55	14.75

The recovery time for PartC3 is smaller than that of PartC1.
Also, recovery time for failure at 75% is much larger than failure at 25% for PartC1.
But, for PartC3, the recovery time is almost same for 25% and 75%. This is probably due to persisted RDDs cutting down recovery time for PartC3.