CS744A1

Hao Fu, Kan Wu, Huayu Zhang

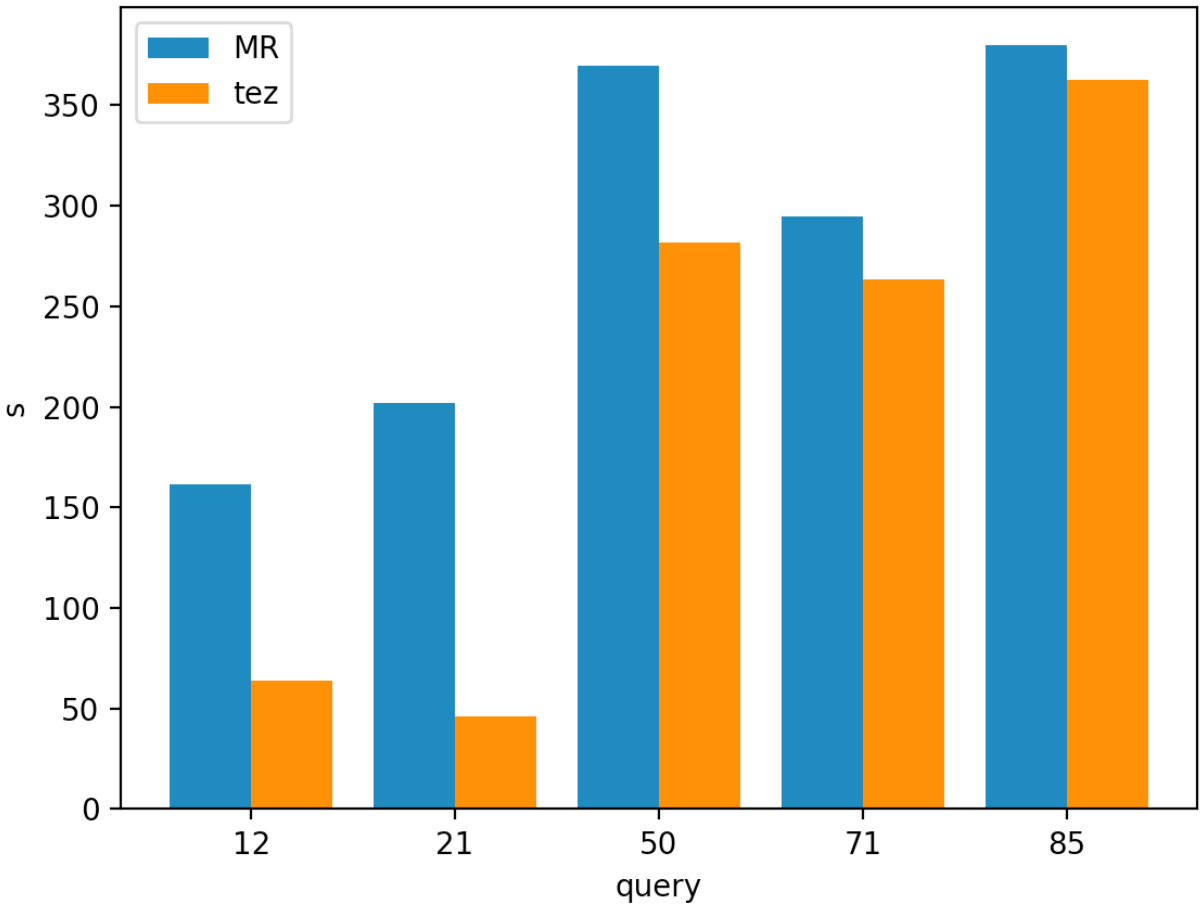
September 2017

Part A.

Question 1

a

Yes,The observation is that Hive/Tez is always better than Hive/MR. However, how much it is better varies from query to query. It is not a constant because the job assignment and structure differs from query to query.



b

Please see the plots for total amount of data for each query and the bandwidth for each query below.
Bandwidth: since we can only calculate throughput instead of bandwidth, which is the upper bound of the throughput. The throughput is shown below and it is a approximation of bandwidth.
Observation: TEZ has more writing to the disks. Explanation: Tez uses different strategies of reduce from MR.
Trend: Generally speaking, Tez has a higher disk reading throughput. It can be verified by the DAG of the jobs, Tez has more jobs reading from the disk independently(simultaneously) and thus reach a higher reading throughput.

Throughput for disks: (MB)

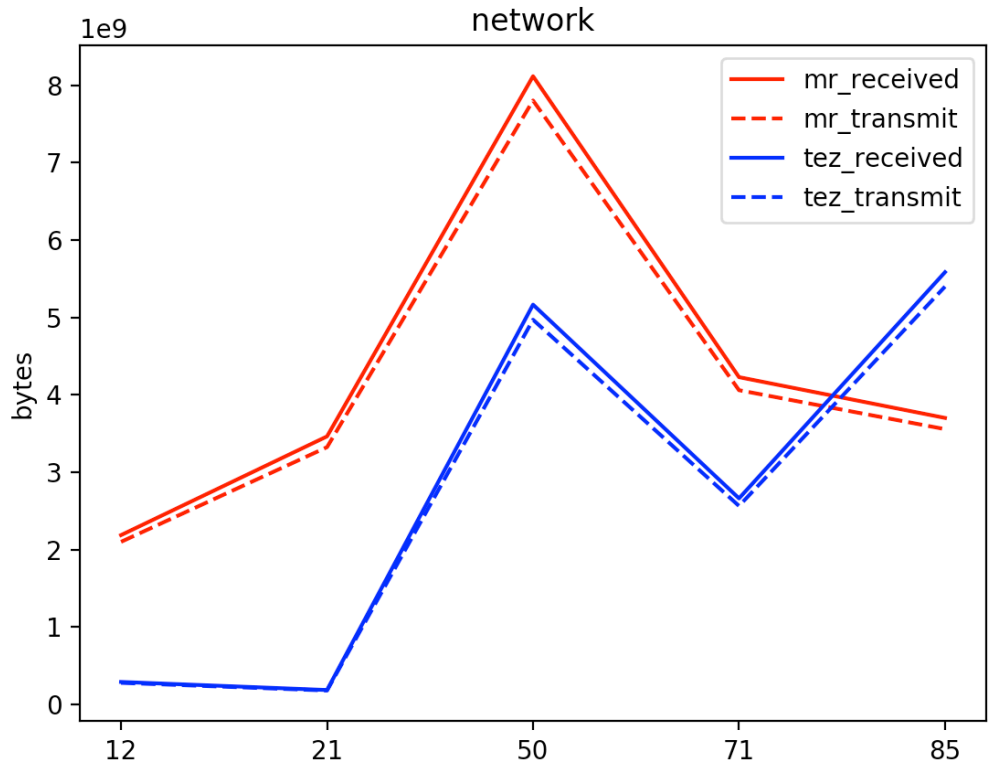
disk, query 12
mr read: 48.95850362496357
tez read: 122.76038139230036
mr write: 14.758073218682364
tez write: 0.9111593139932272
disk, query 21
mr read: 7.780409191435345
tez read: 33.11108693286911
mr write: 16.42801463806492
tez write: 0.8436949189424439
disk, query 50
mr read: 60.12029497743321
tez read: 78.59067666398163
mr write: 22.773393619639204
tez write: 28.76923022330727
disk, query 71
mr read: 144.8298323036187
tez read: 161.93326621711464
mr write: 4.454002308371241
tez write: 143.63922879665685
disk, query 85
mr read: 22.47439027923749
tez read: 32.29191693555059
mr write: 13.041562950260907
tez write: 250.79196933344366

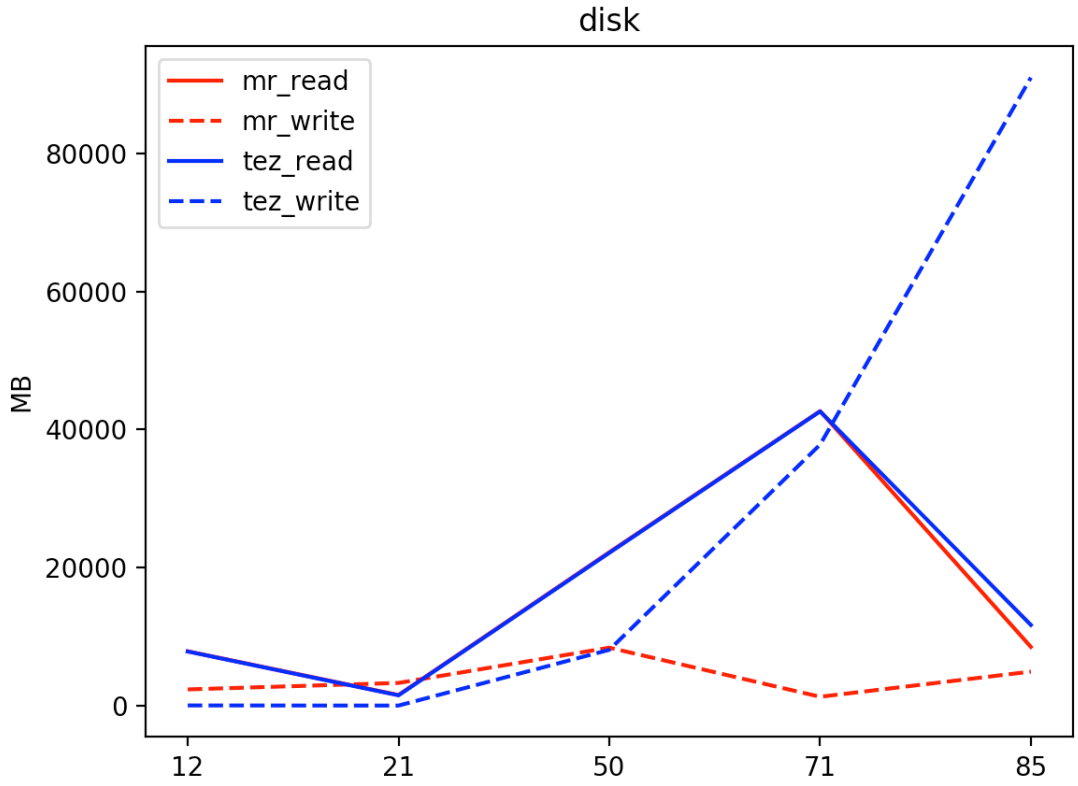
Throughput for net: (bytes)

net, query 12
mr reveived: 13569929.97525474
tez received: 4541608.261419141
mr transmit: 13030853.99676265

tez transmit: 4358742.263697508
net, query 21
mr reveived: 17172014.657899696
tez received: 4059002.6765313894
mr transmit: 16483401.584798628
tez transmit: 3921021.4122511153
net, query 50
mr reveived: 21947591.65041875
tez received: 18328306.441596966
mr transmit: 21099852.03804311
tez transmit: 17636962.903952677
net, query 71
mr reveived: 14354141.642338244
tez received: 10115218.976161078
mr transmit: 13781936.832099939
tez transmit: 9742852.191091271
net, query 85
mr reveived: 9750606.596758517
tez received: 15401145.746118419
mr transmit: 9368998.78306084
tez transmit: 14887260.497504205

Plots for total amount of data:


network
1e9
bytes
mr_received
mr_transmit
tez_received
tez_transmit
0
1
2
3
4
5
6
7
8
12
21
50
71
85

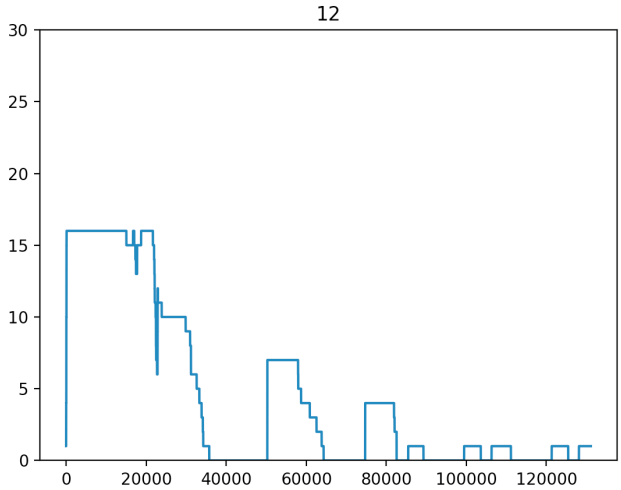

disk
MB
mr_read
mr_write
tez_read
tez_write
0
20000
40000
60000
80000
12
21
50
71
85

C

For map reduce:

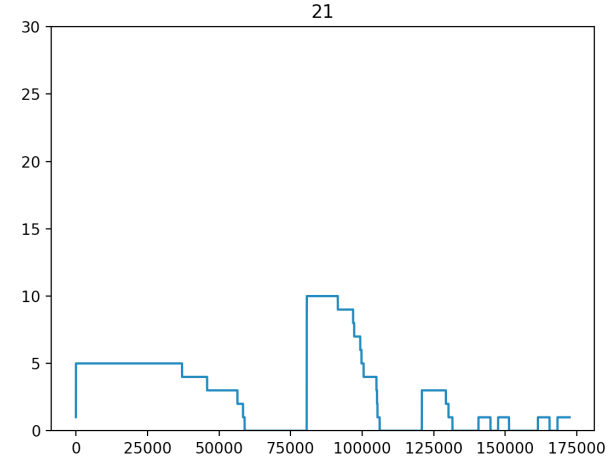
Query 12

total tasks: 42
number of map tasks: 39
number of reduce tasks: 3
ratio of reduce versus map: 0.07692307692307693



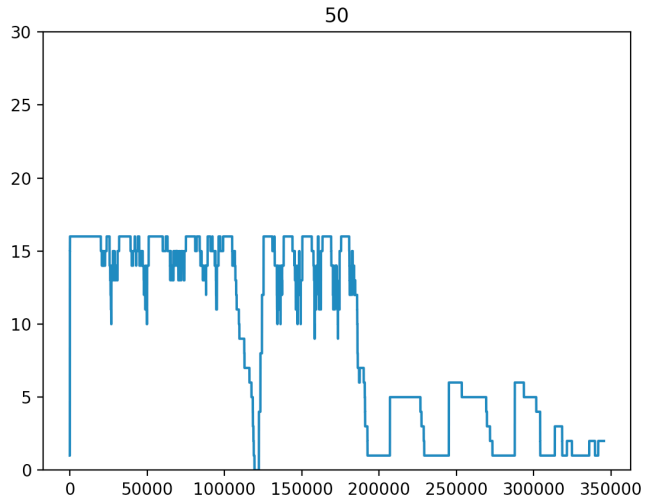
Query 21

total tasks: 22
number of map tasks: 20
number of reduce tasks: 2
ratio of reduce versus map: 0.1



Query 50

total tasks: 188
number of map tasks: 99
number of reduce tasks: 89
ratio of reduce versus map: 0.898989898989899



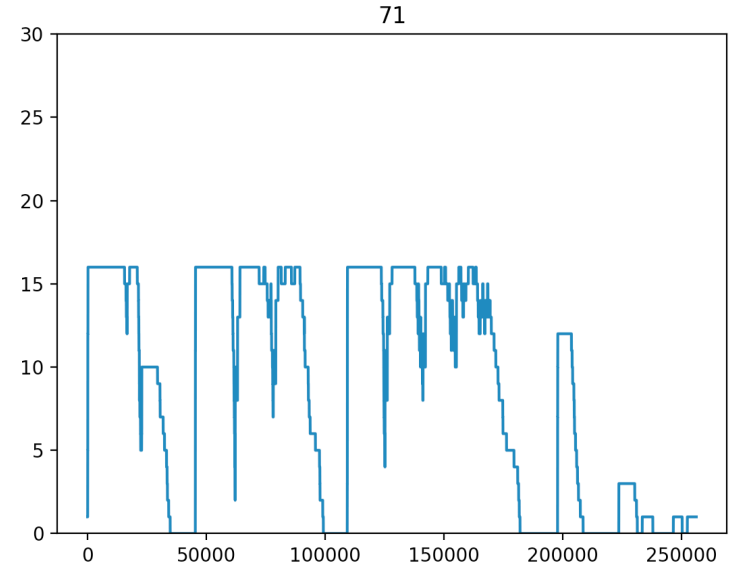
Query 71

total tasks: 170

number of map tasks: 168

number of reduce tasks: 2

ratio of reduce versus map: 0.011904761904761904



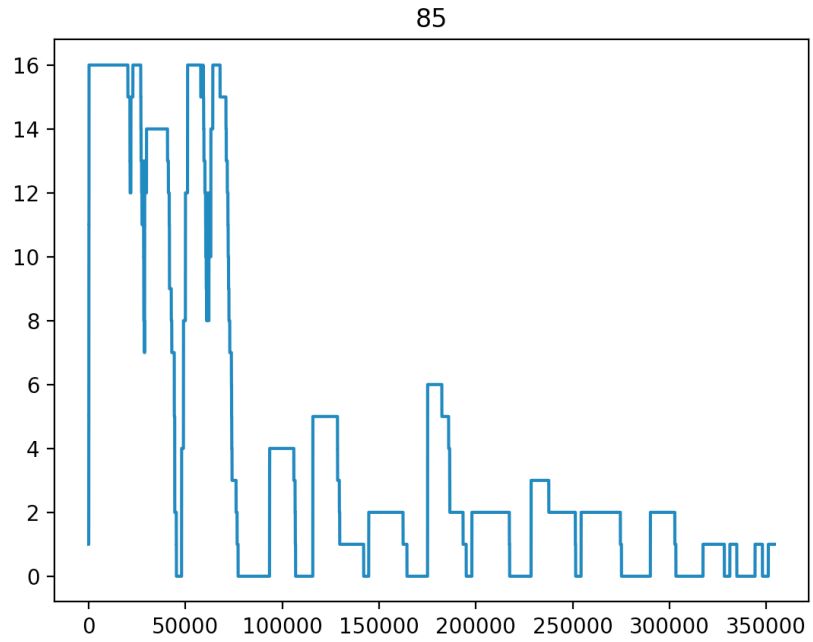
Query 85

total tasks: 92

number of map tasks: 52

number of reduce tasks: 40

ratio of reduce versus map: 0.7692307692307693



For Tez:

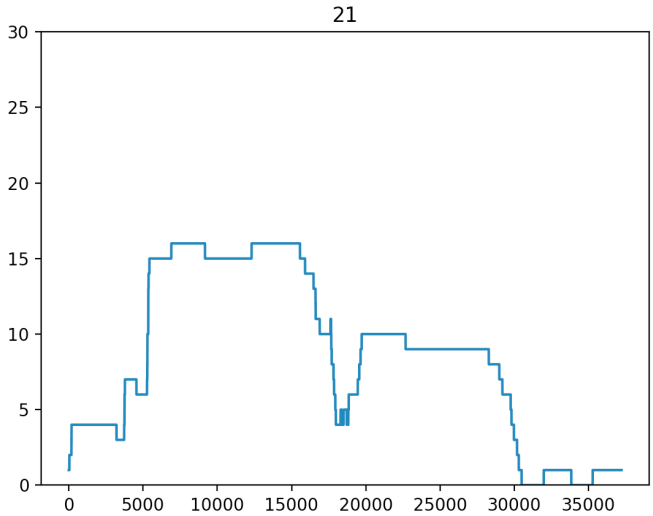
query 12

total tasks : 55

number of map tasks : 52

number of reduce tasks: 3

ratio of reduce versus map: 0.057692307692307696



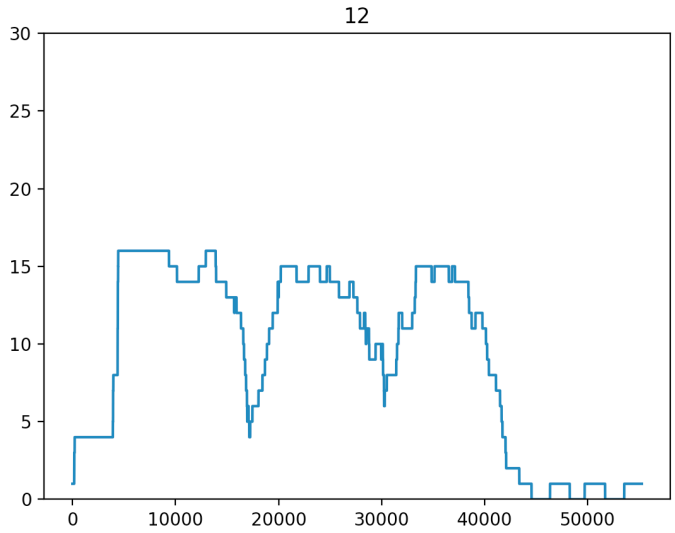
query 21

total tasks : 30

number of map tasks : 28

number of reduce tasks: 2

ratio of reduce versus map: 0.07142857142857142



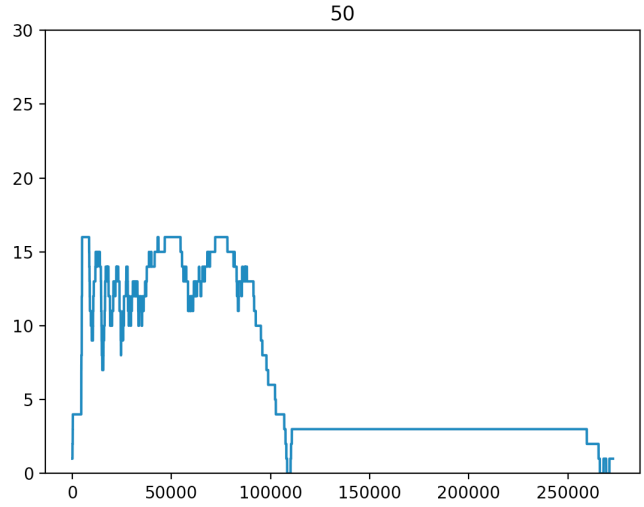
query 50

total tasks : 97

number of map tasks : 92

number of reduce tasks: 5

ratio of reduce versus map: 0.05434782608695652



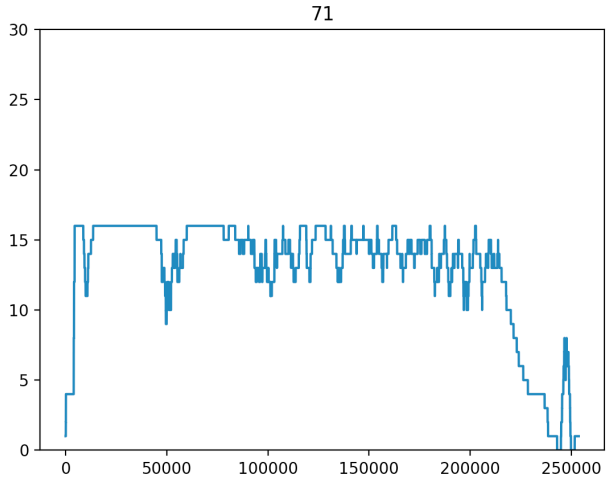
query 71

total tasks : 144

number of map tasks : 129

number of reduce tasks: 15

ratio of reduce versus map: 0.11627906976744186



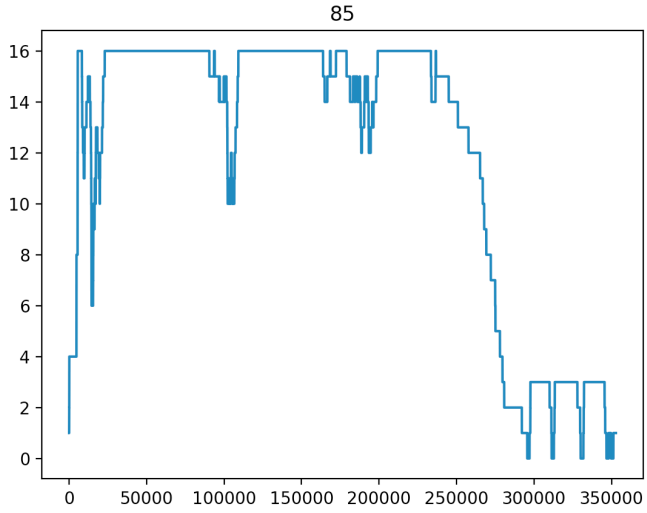
query 85

total tasks : 83

number of map tasks : 72

number of reduce tasks: 11

ratio of reduce versus map: 0.1527777777777778



Correlation:

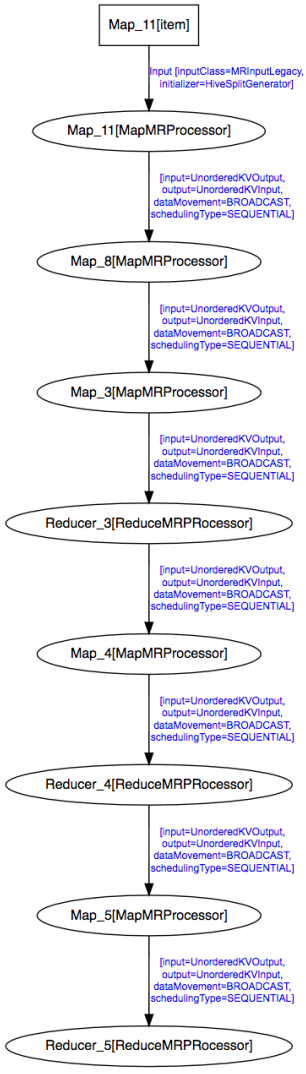
The higher number of total tasks, the higher the overall throughput will be.

The higher number of map tasks, the higher the reading throughput will be, and same story happens to the number of reduce tasks and writing throughput.

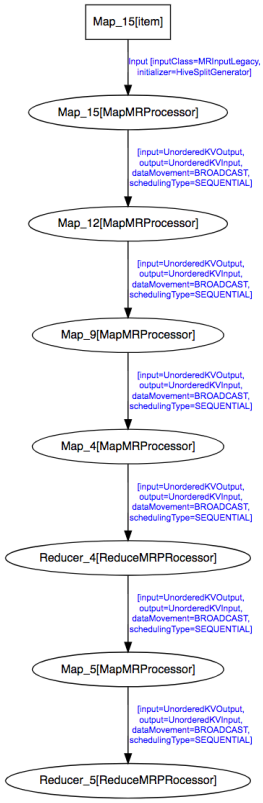
d.

MR:

12:


Map_11[item]
Input [inputClass=MRInputLegacy, initializer=HiveSplitGenerator]
Map_11[MapMRProcessor]
[input=UnorderedKVOutput, output=UnorderedKVInput, dataMovement=BROADCAST, schedulingType=SEQUENTIAL]
Map_8[MapMRProcessor]
[input=UnorderedKVOutput, output=UnorderedKVInput, dataMovement=BROADCAST, schedulingType=SEQUENTIAL]
Map_3[MapMRProcessor]
[input=UnorderedKVOutput, output=UnorderedKVInput, dataMovement=BROADCAST, schedulingType=SEQUENTIAL]
Reducer_3[ReduceMRPRocessor]
[input=UnorderedKVOutput, output=UnorderedKVInput, dataMovement=BROADCAST, schedulingType=SEQUENTIAL]
Map_4[MapMRProcessor]
[input=UnorderedKVOutput, output=UnorderedKVInput, dataMovement=BROADCAST, schedulingType=SEQUENTIAL]
Reducer_4[ReduceMRPRocessor]
[input=UnorderedKVOutput, output=UnorderedKVInput, dataMovement=BROADCAST, schedulingType=SEQUENTIAL]
Map_5[MapMRProcessor]
[input=UnorderedKVOutput, output=UnorderedKVInput, dataMovement=BROADCAST, schedulingType=SEQUENTIAL]
Reducer_5[ReduceMRPRocessor]

21:

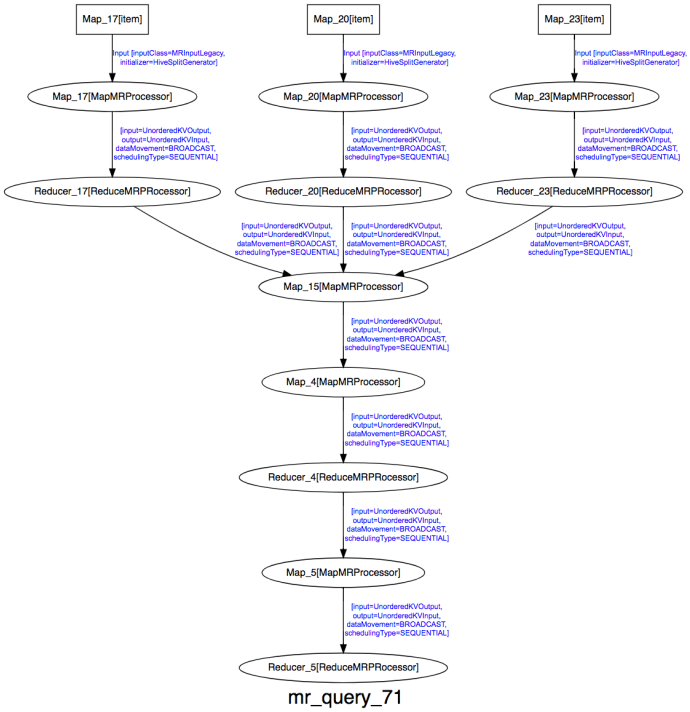


mr_query_21

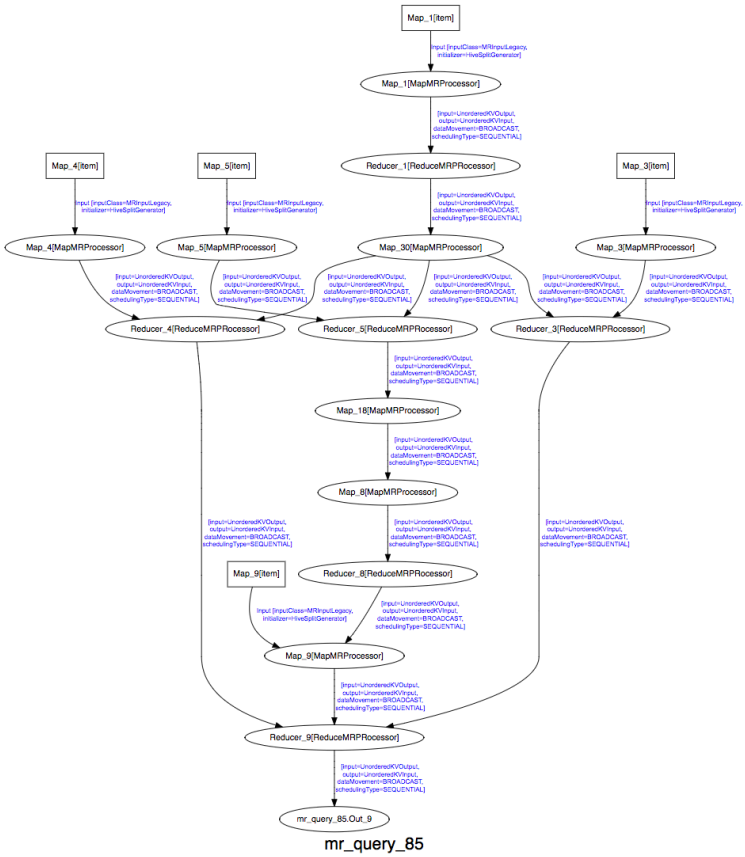
50:

mr_query_50

71:

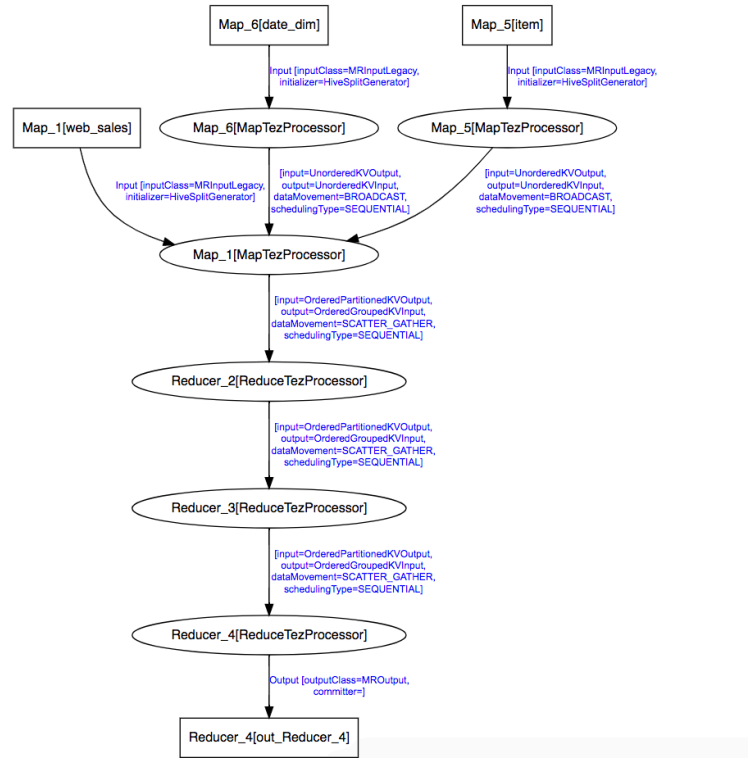

Map_17[item]
Map_20[item]
Map_23[item]
Input [inputClass=MRInputLegacy, initializer=HiveSplitGenerator]
Map_17[MapMRProcessor]
Map_20[MapMRProcessor]
Map_23[MapMRProcessor]
[input=UnorderedKVOutput, output=UnorderedKVInput, dataMovement=BROADCAST, schedulingType=SEQUENTIAL]
Reducer_17[ReduceMRPRocessor]
Reducer_20[ReduceMRPRocessor]
Reducer_23[ReduceMRPRocessor]
Map_15[MapMRProcessor]
Map_4[MapMRProcessor]
Reducer_4[ReduceMRPRocessor]
Map_5[MapMRProcessor]
Reducer_5[ReduceMRPRocessor]
mr_query_71

85:

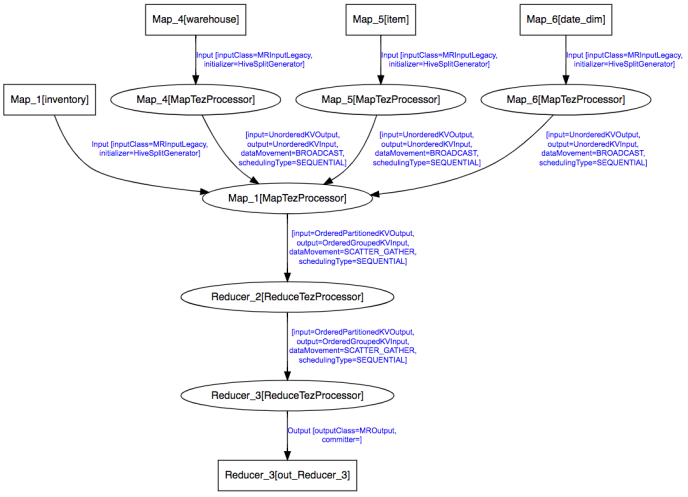

Map_1[item]
Map_1[MapMRProcessor]
Reducer_1[ReduceMRPRocessor]
Map_4[item]
Map_5[item]
Map_3[item]
Map_4[MapMRProcessor]
Map_5[MapMRProcessor]
Map_30[MapMRProcessor]
Map_3[MapMRProcessor]
Reducer_4[ReduceMRPRocessor]
Reducer_5[ReduceMRPRocessor]
Reducer_3[ReduceMRPRocessor]
Map_18[MapMRProcessor]
Map_8[MapMRProcessor]
Map_9[item]
Reducer_8[ReduceMRPRocessor]
Map_9[MapMRProcessor]
Reducer_9[ReduceMRPRocessor]
mr_query_85.Out_9
mr_query_85

Tez:

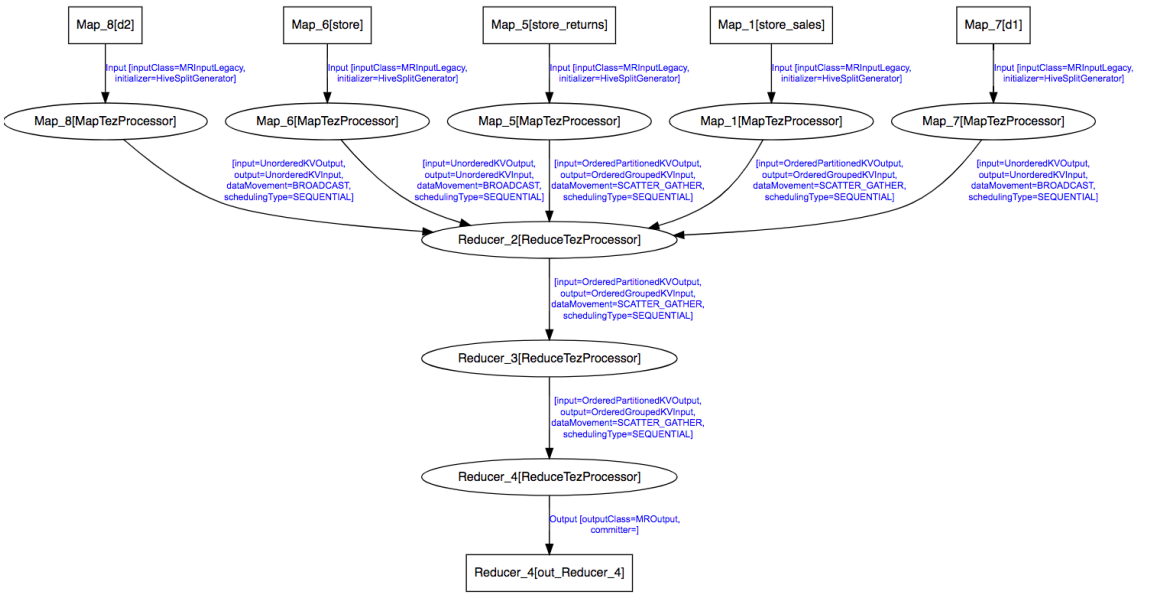
12:


Map_6[date_dim]
Map_5[item]
Input [inputClass=MRInputLegacy, initializer=HiveSplitGenerator]
Input [inputClass=MRInputLegacy, initializer=HiveSplitGenerator]
Map_1[web_sales]
Map_6[MapTezProcessor]
Map_5[MapTezProcessor]
Input [inputClass=MRInputLegacy, initializer=HiveSplitGenerator]
[input=UnorderedKVOutput, output=UnorderedKVInput, dataMovement=BROADCAST, schedulingType=SEQUENTIAL]
[input=UnorderedKVOutput, output=UnorderedKVInput, dataMovement=BROADCAST, schedulingType=SEQUENTIAL]
Map_1[MapTezProcessor]
[input=OrderedPartitionedKVOutput, output=OrderedGroupedKVInput, dataMovement=SCATTER_GATHER, schedulingType=SEQUENTIAL]
Reducer_2[ReduceTezProcessor]
[input=OrderedPartitionedKVOutput, output=OrderedGroupedKVInput, dataMovement=SCATTER_GATHER, schedulingType=SEQUENTIAL]
Reducer_3[ReduceTezProcessor]
[input=OrderedPartitionedKVOutput, output=OrderedGroupedKVInput, dataMovement=SCATTER_GATHER, schedulingType=SEQUENTIAL]
Reducer_4[ReduceTezProcessor]
Output [outputClass=MROutput, committer=]
Reducer_4[out_Reducer_4]

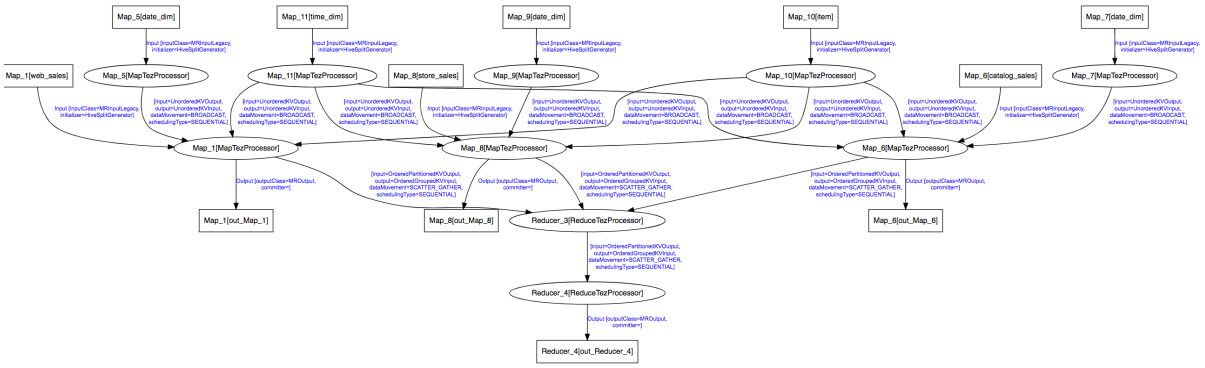
21:


Map_4[warehouse]
Map_5[item]
Map_6[date_dim]
Input [inputClass=MRInputLegacy, initializer=HiveSplitGenerator]
Input [inputClass=MRInputLegacy, initializer=HiveSplitGenerator]
Input [inputClass=MRInputLegacy, initializer=HiveSplitGenerator]
Map_1[inventory]
Map_4[MapTezProcessor]
Map_5[MapTezProcessor]
Map_6[MapTezProcessor]
Input [inputClass=MRInputLegacy, initializer=HiveSplitGenerator]
[input=UnorderedKVOutput, output=UnorderedKVInput, dataMovement=BROADCAST, schedulingType=SEQUENTIAL]
[input=UnorderedKVOutput, output=UnorderedKVInput, dataMovement=BROADCAST, schedulingType=SEQUENTIAL]
[input=UnorderedKVOutput, output=UnorderedKVInput, dataMovement=BROADCAST, schedulingType=SEQUENTIAL]
Map_1[MapTezProcessor]
[input=OrderedPartitionedKVOutput, output=OrderedGroupedKVInput, dataMovement=SCATTER_GATHER, schedulingType=SEQUENTIAL]
Reducer_2[ReduceTezProcessor]
[input=OrderedPartitionedKVOutput, output=OrderedGroupedKVInput, dataMovement=SCATTER_GATHER, schedulingType=SEQUENTIAL]
Reducer_3[ReduceTezProcessor]
Output [outputClass=MROutput, committer=]
Reducer_3[out_Reducer_3]

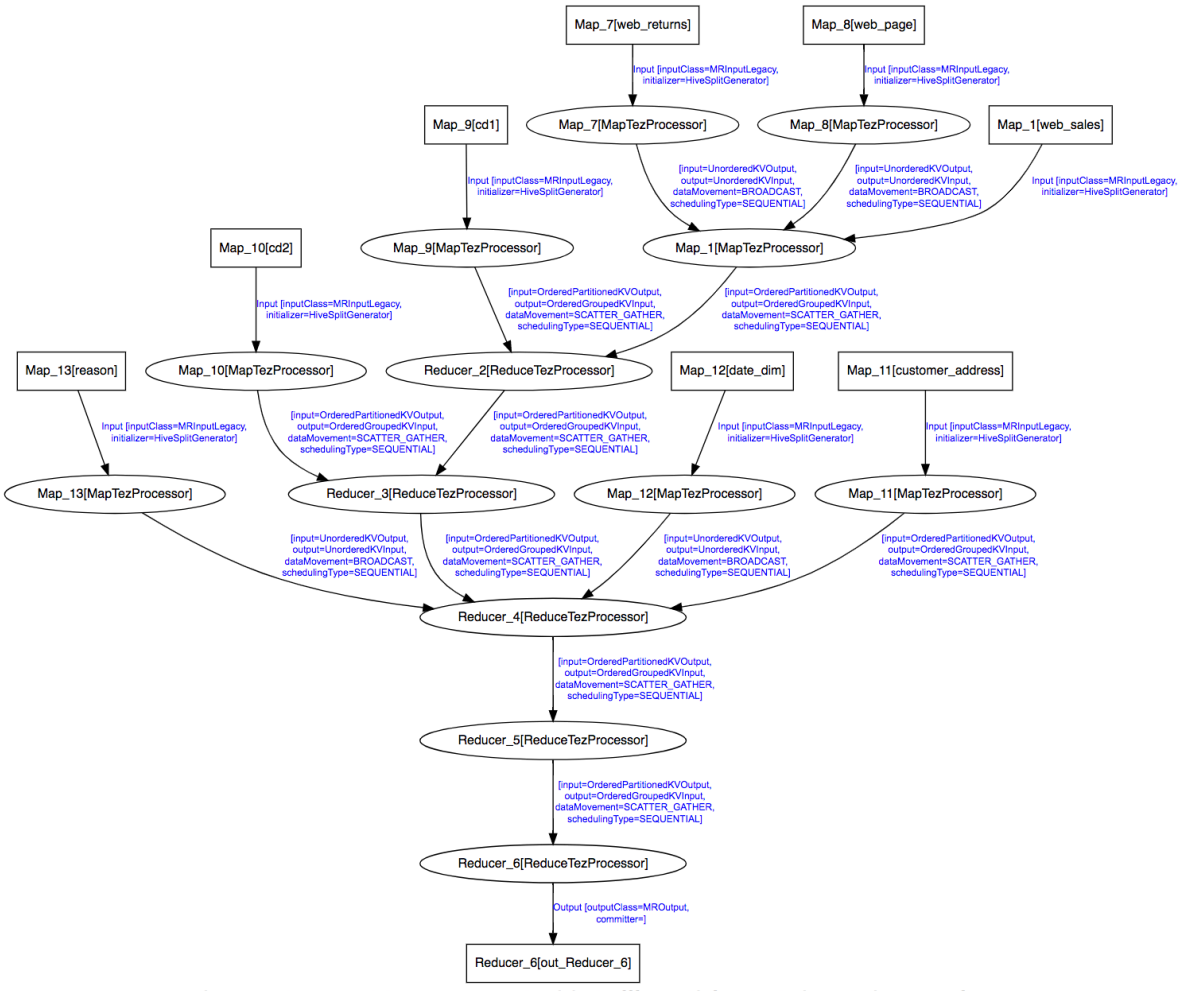
50


Map_8[d2]
Map_6[store]
Map_5[store_returns]
Map_1[store_sales]
Map_7[d1]
Input [inputClass=MRInputLegacy, initializer=HiveSplitGenerator]
Map_8[MapTezProcessor]
Map_6[MapTezProcessor]
Map_5[MapTezProcessor]
Map_1[MapTezProcessor]
Map_7[MapTezProcessor]
[input=UnorderedKVOutput, output=UnorderedKVInput, dataMovement=BROADCAST, schedulingType=SEQUENTIAL]
[input=OrderedPartitionedKVOutput, output=OrderedGroupedKVInput, dataMovement=SCATTER_GATHER, schedulingType=SEQUENTIAL]
Reducer_2[ReduceTezProcessor]
Reducer_3[ReduceTezProcessor]
Reducer_4[ReduceTezProcessor]
Output [outputClass=MROutput, committer=]
Reducer_4[out_Reducer_4]

71:


Map_5[date_dim]
Map_11[time_dim]
Map_9[date_dim]
Map_10[item]
Map_7[date_dim]
Map_1[web_sales]
Map_5[MapTezProcessor]
Map_11[MapTezProcessor]
Map_8[store_sales]
Map_9[MapTezProcessor]
Map_10[MapTezProcessor]
Map_6[catalog_sales]
Map_7[MapTezProcessor]
Map_1[MapTezProcessor]
Map_8[MapTezProcessor]
Map_6[MapTezProcessor]
Map_1[out_Map_1]
Map_8[out_Map_8]
Reducer_3[ReduceTezProcessor]
Map_6[out_Map_6]
Reducer_4[ReduceTezProcessor]
Reducer_4[out_Reducer_4]

85:



These DAGs are very different.

Yes, the structure can impact the performance. Say we have many independent mapping tasks reading from HDFS simultaneously, then we have a higher reading from disk throughput. If we just have a sequential structure, we can only have a lower throughput because the nest stage depends of the result of the previous one. This can be verified by observing difference of reading throughput of Tez and MR in query 12, query 21 and query 50.

2.

Mapreduce:
Normal: 296.7 s
25%: 386.5 s
75%: 355.5 s

Tez:
Normal: 263.2s
25%: 325.9s
75%: 301.3s
We observed that In both MR and Tez, failing a slave can slow the query. The earlier the slave is failed, the slower it will be.
Firstly, failures happened and it takes time to recover from the failures and re-schedule the particular jobs.
Secondly, some of the jobs can be executed in a parallel way at the first place, however after the data node fails, they can no longer be executed in a parallel style because the only copy of the data on the failed node is in the busy node and it can only be executed after the current job is finished. It explains that the earlier it fails, the slower it will be: the earlier it fails, less jobs can be done in parallel.

Workers

Worker Id	Address	State	Cores	Memory
worker-20170925020848-10.254.0.132-33025	10.254.0.132:33025	ALIVE	4 (4 Used)	18.6 GB (1024.0 MB Used)
worker-20170925020848-10.254.0.133-33895	10.254.0.133:33895	ALIVE	4 (4 Used)	18.6 GB (1024.0 MB Used)
worker-20170925020851-10.254.0.135-40578	10.254.0.135:40578	ALIVE	4 (4 Used)	18.6 GB (1024.0 MB Used)
worker-20170927013239-10.254.0.134-51462	10.254.0.134:51462	ALIVE	4 (4 Used)	18.6 GB (1024.0 MB Used)

Figure 1: Caption

1 Part C

Question 1. I ensure the resource are efficiently used by monitoring the Spark dashboard. As shown in Figure 1, all cores of the four workers are active and each worker has 1GB memory in use.

Regarding the network bandwidth, I only compute the transmit size as the transmit time cannot be estimated from the application completion time. More precisely, computing the bandwidth is in fact computing the throughput (TransmitSize / IOTime). Assume the latency is negligible, the bandwidth is approximately equal to the throughput. Each configuration are run three times and the average value is taken. I vary the partitioning size (Table 5- 8). From the results, 10 partitions in RDD has the minimum completion time. I will choose RDD partitioning size = 10 for question 2 and question 3.

hosts	Completion time (s)	Network Transmit Size (MiB)		Disk Bandwidth (MB/s)		Number of tasks
		Receive	Transmit	Read	Write	
vm-21-1	75.0	1.47	1.12	4.15	1.40	23
vm-21-2		0.32	112.52	9.36	1.63	
vm-21-3		0.09	0.24	4.58	1.74	
vm-21-4		117.25	0.78	4.10	4.23	
vm-21-5		0.09	0.24	4.70	1.16	

Table 1: Qeustion 1 Page Rank

Question 2. I use two customed partitions, RangePartitioner and HashPartitioner. The metrics are shown in Table 2, 3. The application completion times with and without customed partitioning are nearly the same. Regarding the page rank algorithm, the customed partitioning is slightly faster. However there is an overhead to partition the data. For Range partitioner, the application consists of two jobs. One is range partitioning, which takes 16s. The other is pagerank, which takes 29s.

hosts	Completion time (s)	Network Transmit Size (MiB)		Disk Bandwidth (MB/s)		Number of tasks
		Receive	Transmit	Read	Write	
vm-21-1	41.9	3.70	3.25	4.14	1.49	225
vm-21-2		140.83	255.12	7.08	2.37	
vm-21-3		208.00	64.79	3.36	3.12	
vm-21-4		144.78	291.07	3.66	2.51	
vm-21-5		199.62	58.62	3.41	3.43	

Table 2: Qeustion 2 Page Rank with Hash Partitioner

Question 3. As there is no improvement using the customed partitioning, in this question I use the partitioning scheme in Question 1. Fix the RDD partition size as 10, cache the variable graph. The experimental

hosts	Completion time (s)	Network Transmit Size (MiB)		Disk Bandwidth (MB/s)		Number of tasks
		Receive	Transmit	Read	Write	
vm-21-1	45.2	3.00	3.02	4.28	1.31	227
vm-21-2		121.97	228.55	8.26	2.52	
vm-21-3		176.77	55.26	3.52	2.91	
vm-21-4		143.68	250.42	3.81	2.53	
vm-21-5		168.06	55.06	3.67	3.21	

Table 3: Qeustion 2 Page Rank with Range Partitioner

result is in Table 4. The application with cached graph is only 0.4s faster than the uncached one. The reason is not clear. I suspect Spark do some optimization by automatically storing the graph variable in memory.

hosts	Completion time (s)	Network Transmit Size (MiB)		Disk Bandwidth (MB/s)		Number of tasks
		Receive	Transmit	Read	Write	
vm-21-1	41.8	3.06	2.98	4.36	1.35	215
vm-21-2		188.82	158.37	8.70	3.01	
vm-21-3		185.58	45.10	4.66	2.67	
vm-21-4		121.92	406.97	4.09	2.17	
vm-21-5		181.96	44.66	4.82	2.72	

Table 4: Qeustion 3 Page Rank

Question 4. In this question, I experimented over four different paritionings i.e. 2, 10, 100, 300. From the results (Table 5-8), when RDD partition size is 10 the applicaton completion time is minimum. When the partition size increases, the parallelism increases so that the application completion time decrease. However, when the partition size are too large, the network transmit overhead will dominate and make the application completion time longer.

Question 5. See Figure 2. Same for all three application because they have the same algorithm. The difference in the partitioning and cache does not have an impact on the lineage graph.

Question 6. Figure 3. All applications have the same DAGs. The reason is they share the same repartitioning size (10) and the same algorithm. Impact: if we have a repartition step, we will distributed the data across the workers so all the workers will be used. This increases parallelism and speeds up the application.

Question 7. In this question, I failed vm-21-4 for all cases. The result is in Table 9. The impact of clearing cache is not obvious. The spark jobs are executed as the normal case. However killing one running worker increase the completion time much, especially when the failure is at 75% progress. The reason is on killing the worker, the ongoing stage failed and Spark will retry it on other workers. For example, in the case where the failure b happens at 75% for question 1's app, to deal with the failure, spark runs 7 more stages (14 more tasks) and the overhead is about 35s.

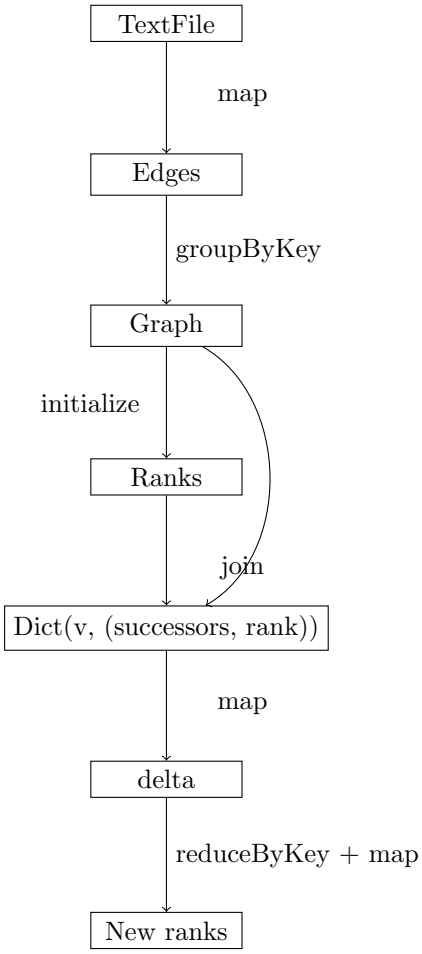
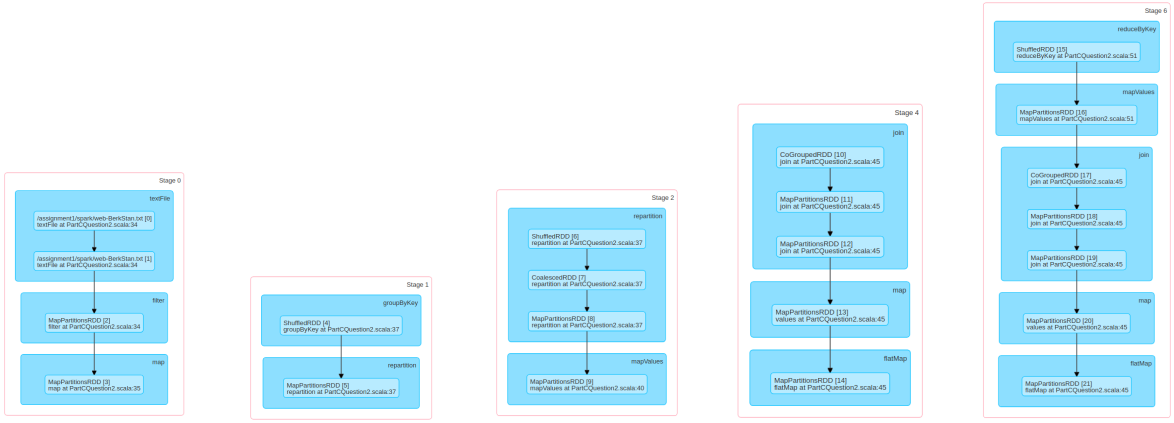


Figure 2: Lineage graph for Question 3



(a) Stage 0 (b) Stage 1 (c) Stage 2 (d) Stage 4 (e) Stage 6

Figure 3: Question 2/3 Stage-level DAGs

hosts	Completion time (s)	Network Transmit Size (MiB)		Disk Bandwidth (MB/s)		Number of tasks
		Receive	Transmit	Read	Write	
vm-21-1	81.6	1.43	1.35	4.60	1.66	47
vm-21-2		94.57	191.65	9.37	2.48	
vm-21-3		94.26	79.12	4.50	2.57	
vm-21-4		362.50	271.52	4.01	2.96	
vm-21-5		94.05	78.73	4.93	2.35	

Table 5: Qeustion 4 Number pf partitions = 2

hosts	Completion time (s)	Network Transmit Size (MiB)		Disk Bandwidth (MB/s)		Number of tasks
		Receive	Transmit	Read	Write	
vm-21-1	42.2	3.07	2.88	5.00	1.79	215
vm-21-2		222.59	175.95	9.04	2.94	
vm-21-3		174.27	71.74	4.77	2.83	
vm-21-4		180.48	398.66	3.97	2.30	
vm-21-5		166.13	70.75	4.68	3.32	

Table 6: Qeustion 4 Number pf partitions = 10

hosts	Completion time (s)	Network Transmit Size (MiB)		Disk Bandwidth (MB/s)		Number of tasks
		Receive	Transmit	Read	Write	
vm-21-1	50.8	18.24	18.96	5.53	3.31	2105
vm-21-2		235.60	275.60	8.94	2.29	
vm-21-3		190.84	139.12	4.80	2.60	
vm-21-4		407.09	504.37	3.74	2.46	
vm-21-5		274.00	154.27	4.79	2.84	

Table 7: Qeustion 4 Number pf partitions = 100

hosts	Completion time (s)	Network Transmit Size (MiB)		Disk Bandwidth (MB/s)		Number of tasks
		Receive	Transmit	Read	Write	
vm-21-1	80.8	54.41	56.77	4.21	4.69	6305
vm-21-2		461.86	443.06	8.77	1.89	
vm-21-3		456.85	316.62	4.38	1.93	
vm-21-4		467.47	728.71	3.94	2.28	
vm-21-5		456.67	328.13	4.45	2.05	

Table 8: Qeustion 4 Number pf partitions = 300

	failure a at 25%	failure a at 75%	failure b at 25%	failure b at 75%
Q1	71.3s	75.3s	73.3s	108.4s
Q3	44.2s	42.2s	52.2s	54.2s

Table 9: Completion time