Report

I. Cluster Configuration

The Spark cluster setup was consistent with one Master node and ten worker nodes. The master node takes an application and splits it up into jobs that are allocated by the Spark Driver to the different Workers.

The spark driver is in charge of breaking down a program into a Directed Acyclic Graph. The Workers are given jobs by the spark driver and return the result back to the spark driver. From the figure below, the 10 slaves are allocated their own cores and memory to be used to compute their tasks that are either assigned from the spark driver and resources are allocated by the cluster manager.

Workers are EC2 instances, in this case, they are Ubuntu Linux instances that contain 2 VCPUS and 8 Gb RAM each, they are responsible for running their executors. The executors are the ones that process the task that the worker is given.

URL: spark://ec2-54-172-106-255.compute-1.amazonaws.com:7077
REST URL: spark://ec2-54-172-106-255.compute-1.amazonaws.com:6066 *(cluster mode)*
Alive Workers: 10
Cores in use: 20 Total, 0 Used
Memory in use: 65.5 GB Total, 0.0 B Used
Applications: 0 Running, 22 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20191103235039-172.31.13.41-60765	172.31.13.41:60765	ALIVE	2 (0 Used)	6.6 GB (0.0 B Used)
worker-20191103235040-172.31.12.2-52363	172.31.12.2:52363	ALIVE	2 (0 Used)	6.6 GB (0.0 B Used)
worker-20191103235040-172.31.12.23-41365	172.31.12.23:41365	ALIVE	2 (0 Used)	6.6 GB (0.0 B Used)
worker-20191103235040-172.31.3.16-44483	172.31.3.16:44483	ALIVE	2 (0 Used)	6.6 GB (0.0 B Used)
worker-20191103235040-172.31.4.247-38177	172.31.4.247:38177	ALIVE	2 (0 Used)	6.6 GB (0.0 B Used)
worker-20191103235040-172.31.5.152-47781	172.31.5.152:47781	ALIVE	2 (0 Used)	6.6 GB (0.0 B Used)
worker-20191103235042-172.31.12.108-53463	172.31.12.108:53463	ALIVE	2 (0 Used)	6.6 GB (0.0 B Used)
worker-20191103235042-172.31.12.231-55907	172.31.12.231:55907	ALIVE	2 (0 Used)	6.6 GB (0.0 B Used)
worker-20191103235042-172.31.6.220-48295	172.31.6.220:48295	ALIVE	2 (0 Used)	6.6 GB (0.0 B Used)
worker-20191103235045-172.31.9.28-60577	172.31.9.28:60577	ALIVE	2 (0 Used)	6.6 GB (0.0 B Used)

Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20191105034612-0040 (kill)	PageRank	20	2.0 GB	2019/11/05 03:46:12	root	RUNNING	11 min

Completed Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20191105034500-0039	PageRank	20	2.0 GB	2019/11/05 03:45:00	root	FINISHED	0.9 s
app-20191105034006-0038	PageRank	20	2.0 GB	2019/11/05 03:40:06	root	FINISHED	1.0 s
app-20191105033604-0037	PageRank	20	2.0 GB	2019/11/05 03:36:04	root	FINISHED	2.8 min
app-20191105025612-0036	PageRank	20	2.0 GB	2019/11/05 02:56:12	root	FINISHED	33 min
app-20191105025006-0035	PageRank	20	2.0 GB	2019/11/05 02:50:06	root	FINISHED	5.9 min
app-20191105024723-0034	PageRank	20	2.0 GB	2019/11/05 02:47:23	root	FINISHED	29 s
app-20191105014708-0033	PageRank	20	2.0 GB	2019/11/05 01:47:08	root	FINISHED	11 min
app-20191105000752-0031	PageRank	20	2.0 GB	2019/11/05 00:07:52	root	FINISHED	31 min
app-20191105000947-0032	PageRank	0	2.0 GB	2019/11/05 00:09:47	root	FINISHED	2.9 min
app-20191105000440-0030	PageRank	20	2.0 GB	2019/11/05 00:04:40	root	FINISHED	1.0 s

II. Execution

a. NoInOutLink

Completed Jobs (5)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
4	take at NoInOutLink.scala:58	2019/11/04 05:21:38	2 s	2/2 (3 skipped)	11/11 (30 skipped)
3	sortByKey at NoInOutLink.scala:58	2019/11/04 05:21:18	20 s	4/4	40/40
2	take at NoInOutLink.scala:52	2019/11/04 05:21:16	2 s	2/2 (3 skipped)	11/11 (30 skipped)
1	sortByKey at NoInOutLink.scala:52	2019/11/04 05:21:00	15 s	4/4	40/40
0	zipWithIndex at NoInOutLink.scala:44	2019/11/04 05:20:58	2 s	1/1	9/9

Stage skipped means that data has been fetched from cache and there was no need to re-execute the given stage. Whenever there is shuffling involved, Spark will automatically cache generated data. Shuffle generated a number of intermediate files on disk, these files are preserved until the corresponding RDDs are no longer used and are garbage collected.

Completed Stages (2)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
18	take at NoInOutLink.scala:58 +details	2019/11/04 05:21:40	0.6 s	1/1			3.7 MB	
17	subtractByKey at NoInOutLink.scala:56 +details	2019/11/04 05:21:38	2 s	10/10			123.3 MB	37.0 MB

Skipped Stages (3)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
16	map at NoInOutLink.scala:45 +details	Unknown	Unknown	0/10				
15	map at NoInOutLink.scala:55 +details	Unknown	Unknown	0/10				
14	distinct at NoInOutLink.scala:39 +details	Unknown	Unknown	0/10				

In this case, skipped happened at the “take” step, and we don’t have to redo the mapping and the “distinct” operation anymore because the value is cached.

Longest Job

Details for Job 3

Status: SUCCEEDED
Completed Stages: 4

- Event Timeline
- DAG Visualization

Completed Stages (4)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
13	sortByKey at NoInOutLink.scala:58 +details	2019/11/04 05:21:37	1 s	10/10			123.3 MB	
12	map at NoInOutLink.scala:55 +details	2019/11/04 05:21:35	2 s	10/10			87.9 MB	22.8 MB
11	distinct at NoInOutLink.scala:39 +details	2019/11/04 05:21:18	17 s	10/10	1009.4 MB			87.9 MB
10	map at NoInOutLink.scala:45 +details	2019/11/04 05:21:18	1 s	10/10	711.6 MB			100.6 MB

Job 3 took most of the time, we can see it is because of the “distinct” function. The distinct() function traverses the RDD that we created for our links. The method searches and discards any duplicates in our key-value RDD leaving only the unique links. This takes the most time because it requires keeping track of previously known keys and must traverse the entire dataset which takes up a significant amount of time compared to other RDD operations: sortByKey or map.

b. PageRank

Completed Jobs (5)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
4	take at PageRank.scala:83	2019/11/05 16:26:09	14 s	2/2 (28 skipped)	11/11 (280 skipped)
3	sortBy at PageRank.scala:82	2019/11/05 16:25:51	18 s	2/2 (24 skipped)	20/20 (240 skipped)
2	sum at PageRank.scala:75	2019/11/05 16:17:43	8.1 min	25/25	250/250
1	count at PageRank.scala:52	2019/11/05 16:17:21	22 s	1/1	10/10
0	zipWithIndex at PageRank.scala:39	2019/11/05 16:17:16	5 s	1/1	9/9

As I mentioned before, the stage skipped means that data has been fetched from cache and there was no need to re-execute the given stage.

Completed Stages (2)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
52	sortBy at PageRank.scala:82	+details	2019/11/05 16:25:59	10 s	10/10			511.3 MB	
27	map at PageRank.scala:40	+details	2019/11/05 16:25:51	8 s	10/10	106.4 MB			107.9 MB

Skipped Stages (24)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
51	flatMap at PageRank.scala:61	+details	Unknown	Unknown	0/10				
50	map at PageRank.scala:31	+details	Unknown	Unknown	0/10				
49	flatMap at PageRank.scala:61	+details	Unknown	Unknown	0/10				
48	map at PageRank.scala:31	+details	Unknown	Unknown	0/10				
47	flatMap at PageRank.scala:61	+details	Unknown	Unknown	0/10				
46	map at PageRank.scala:31	+details	Unknown	Unknown	0/10				
45	flatMap at PageRank.scala:61	+details	Unknown	Unknown	0/10				
44	map at PageRank.scala:31	+details	Unknown	Unknown	0/10				
43	flatMap at PageRank.scala:61	+details	Unknown	Unknown	0/10				
42	map at PageRank.scala:31	+details	Unknown	Unknown	0/10				
41	flatMap at PageRank.scala:61	+details	Unknown	Unknown	0/10				
40	map at PageRank.scala:31	+details	Unknown	Unknown	0/10				

Completed Stages (2)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
82	take at PageRank.scala:83	+details	2019/11/05 16:26:21	2 s	1/1			24.1 MB	
81	sortBy at PageRank.scala:82	+details	2019/11/05 16:26:09	12 s	10/10			511.3 MB	212.0 MB

Skipped Stages (28)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
80	map at PageRank.scala:40	+details	Unknown	Unknown	0/10				
79	map at PageRank.scala:40	+details	Unknown	Unknown	0/10				
78	map at PageRank.scala:47	+details	Unknown	Unknown	0/10				
77	distinct at PageRank.scala:46	+details	Unknown	Unknown	0/10				
76	flatMap at PageRank.scala:61	+details	Unknown	Unknown	0/10				
75	map at PageRank.scala:31	+details	Unknown	Unknown	0/10				
74	flatMap at PageRank.scala:61	+details	Unknown	Unknown	0/10				
73	map at PageRank.scala:31	+details	Unknown	Unknown	0/10				
72	flatMap at PageRank.scala:61	+details	Unknown	Unknown	0/10				
71	map at PageRank.scala:31	+details	Unknown	Unknown	0/10				
70	flatMap at PageRank.scala:61	+details	Unknown	Unknown	0/10				
69	map at PageRank.scala:31	+details	Unknown	Unknown	0/10				
68	flatMap at PageRank.scala:61	+details	Unknown	Unknown	0/10				
67	map at PageRank.scala:31	+details	Unknown	Unknown	0/10				
66	flatMap at PageRank.scala:61	+details	Unknown	Unknown	0/10				

In this case, at the step “take” and “sortBy”, which is at the end of the code, all the mapping before has already done and the intermediate data was saved in the cache, so the mapping operations are skipped

Longest Job

Completed Stages (25)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
26	sum at PageRank.scala:75 +details	2019/11/05 16:25:43	8 s	10/10			403.3 MB	
25	flatMap at PageRank.scala:61 +details	2019/11/05 16:25:00	43 s	10/10			1199.2 MB	267.7 MB
23	flatMap at PageRank.scala:61 +details	2019/11/05 16:24:15	45 s	10/10			1199.6 MB	267.7 MB
21	flatMap at PageRank.scala:61 +details	2019/11/05 16:23:22	53 s	10/10			1199.4 MB	267.7 MB
19	flatMap at PageRank.scala:61 +details	2019/11/05 16:22:37	45 s	10/10			1200.0 MB	267.7 MB
17	flatMap at PageRank.scala:61 +details	2019/11/05 16:21:53	44 s	10/10			1200.5 MB	267.7 MB
15	flatMap at PageRank.scala:61 +details	2019/11/05 16:21:11	42 s	10/10			1200.1 MB	267.7 MB
13	flatMap at PageRank.scala:61 +details	2019/11/05 16:20:23	49 s	10/10			1199.7 MB	267.7 MB
11	flatMap at PageRank.scala:61 +details	2019/11/05 16:19:41	42 s	10/10			1200.1 MB	267.6 MB
9	flatMap at PageRank.scala:61 +details	2019/11/05 16:18:54	47 s	10/10			1174.5 MB	267.5 MB
7	flatMap at PageRank.scala:61 +details	2019/11/05 16:18:21	32 s	10/10			831.0 MB	241.6 MB
4	map at PageRank.scala:47 +details	2019/11/05 16:18:19	10 s	10/10			87.9 MB	28.0 MB
24	map at PageRank.scala:31 +details	2019/11/05 16:17:44	25 s	10/10	1009.4 MB			797.3 MB
22	map at PageRank.scala:31 +details	2019/11/05 16:17:44	23 s	10/10	1009.4 MB			797.3 MB
20	map at PageRank.scala:31 +details	2019/11/05 16:17:44	24 s	10/10	1009.4 MB			797.3 MB
18	map at PageRank.scala:31 +details	2019/11/05 16:17:44	37 s	10/10	1009.4 MB			797.3 MB
16	map at PageRank.scala:31 +details	2019/11/05 16:17:44	35 s	10/10	1009.4 MB			797.3 MB
14	map at PageRank.scala:31 +details	2019/11/05 16:17:44	28 s	10/10	1009.4 MB			797.3 MB
12	map at PageRank.scala:31 +details	2019/11/05 16:17:44	30 s	10/10	1009.4 MB			797.3 MB
10	map at PageRank.scala:31 +details	2019/11/05 16:17:44	36 s	10/10	1009.4 MB			797.3 MB
8	map at PageRank.scala:31 +details	2019/11/05 16:17:44	31 s	10/10	1009.4 MB			797.3 MB
6	map at PageRank.scala:31 +details	2019/11/05 16:17:44	37 s	10/10	1009.4 MB			797.3 MB
5	mapValues at PageRank.scala:53 +details	2019/11/05 16:17:44	34 s	10/10	1009.4 MB			33.8 MB
3	distinct at PageRank.scala:46 +details	2019/11/05 16:17:44	35 s	10/10	1009.4 MB			87.9 MB
2	map at PageRank.scala:40 +details	2019/11/05 16:17:44	4 s	10/10	106.4 MB			107.9 MB

These jobs took the most time because they were utilizing both the map() and flatMap() functions. The function flat map() is a transformation that returns a new RDD but with each element in the original RDD as the resulting element on a single layer. This is a time-intensive operation because it requires the transformation of the entire source RDD, and to take all the elements in those different indexes and separating them out to be mapped individually into the destination RDD.

In the for-loop, I first start by calculating the page ranks by taking the values from my key, value pairs and flat map them to contain their new ranks and their keys. These will be used to calculate the next values for the ranks of all the pages that have inlinks and the ranks of values with no inlinks.

Spark uses Lazy Evaluation to complete its operations which results in a slow down of certain operations, this is why some operations take longer than others. The lazy evaluation means that operations are not executed immediately. Thus the for-loop that is responsible for looping and calculating the ranks per iteration will take a longer time because the flatMap and unions are not started right after each task is finished.