How AWS distribute word among worker nodes:

The work is distributed on one master and 10 slave nodes.

The Master is used to run the Driver Program. A master node is an EC2 instance. It handles resource allocation for various jobs to the spark cluster. The master identifies the resource (CPU time, memory) needed to run, when jobs are submitted. It also provides the resources (CPU time, memory) to the Driver Program that initiated the job as Executors, and facilitates communication between workers.

The Driver Program downloads dependencies, assigns tasks to executors, breaks apart the job or program (generally written in Python, Scala or Java) into a Directed Acyclic Graph (DAG), allows communication between the workers to share data during shuffling.

Each worker is an EC2 instance. They run the Executors. Executors: Executors are assigned tasks for a job. They run on a particular Worker to complete the task. Job has a separate set of executors.

Why some stages skipped?

Skipped stages mean the data can be retrieved from the cache, so there is no need to execute the stage again. The mechanism shuffles the MapReduce engine, reshuffling the output of the map phase. grouping it on the intermediate key, and sending to the reduce phase.

Completed Stages (3)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
10	sortByKey at NoInOutLink.scala:71	+details	2019/11/05 05:13:48	21 s	10/10			200.0 MB	
9	map at NoInOutLink.scala:47	+details	2019/11/05 05:11:44	2.1 min	10/10	1009.4 MB			102.6 MB
8	map at NoInOutLink.scala:56	+details	2019/11/05 05:11:44	2 s	10/10	106.4 MB			97.3 MB

Completed Jobs (5)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
4	take at NoInOutLink.scala:71	2019/11/05 05:14:09	20 s	2/2 (2 skipped)	11/11 (20 skipped)
3	sortByKey at NoInOutLink.scala:71	2019/11/05 05:11:43	2.4 min	3/3	30/30
2	take at NoInOutLink.scala:64	2019/11/05 05:11:31	12 s	2/2 (2 skipped)	11/11 (20 skipped)
1	sortByKey at NoInOutLink.scala:64	2019/11/05 05:11:05	26 s	3/3	30/30
0	zipWithIndex at NoInOutLink.scala:55	2019/11/05 05:11:02	3 s	1/1	9/9

For the NoInOutLink, the flatMap function takes the most time, since returning a new RDD takes time as we have numerous RDD. I used join() and flatMap()to get the pairs and rank of each element in the array of links. The flatMap() function returns a new RDD by applying a function to all elements of this RDD then flattening the results. I also use subtract() to return an RDD with elements not in order. I extract links with no inlinks, by subtracting links with inlinks, and the number of overall links is large, so iterating the two elements takes a long time.

Active Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	sum at PageRank.scala:69	2019/11/05 03:46:18	31 min	19/23	198/230

Completed Jobs (2)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	count at PageRank.scala:41	2019/11/05 03:46:16	2 s	1/1	10/10
0	zipWithIndex at PageRank.scala:38	2019/11/05 03:46:13	3 s	1/1	9/9

Active Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
19	flatMap at PageRank.scala:51 +details (kill)	2019/11/05 04:10:44	6.5 min	8/10			4.1 GB	216.3 MB

Pending Stages (3)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
24	sum at PageRank.scala:69 +details	Unknown	Unknown	0/10				
23	flatMap at PageRank.scala:51 +details	Unknown	Unknown	0/10				
21	flatMap at PageRank.scala:51 +details	Unknown	Unknown	0/10				

Completed Stages (19)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
17	flatMap at PageRank.scala:51 +details	2019/11/05 04:05:25	5.3 min	10/10			3.6 GB	270.4 MB
15	flatMap at PageRank.scala:51 +details	2019/11/05 04:00:38	4.8 min	10/10			3.1 GB	270.4 MB
13	flatMap at PageRank.scala:51 +details	2019/11/05 03:56:10	4.5 min	10/10			2.6 GB	270.4 MB
11	flatMap at PageRank.scala:51 +details	2019/11/05 03:53:56	2.2 min	10/10			2.0 GB	270.3 MB
9	flatMap at PageRank.scala:51 +details	2019/11/05 03:50:48	3.1 min	10/10			1545.2 MB	270.3 MB
7	flatMap at PageRank.scala:51 +details	2019/11/05 03:49:25	1.4 min	10/10			1004.9 MB	270.2 MB
4	flatMap at PageRank.scala:51 +details	2019/11/05 03:47:12	2.2 min	10/10			522.5 MB	241.2 MB
22	map at PageRank.scala:28 +details	2019/11/05 03:46:18	18 s	10/10	1009.4 MB			493.1 MB
20	map at PageRank.scala:28 +details	2019/11/05 03:46:18	19 s	10/10	1009.4 MB			493.1 MB
18	map at PageRank.scala:28 +details	2019/11/05 03:46:18	19 s	10/10	1009.4 MB			493.1 MB
16	map at PageRank.scala:28 +details	2019/11/05 03:46:18	23 s	10/10	1009.4 MB			493.1 MB
14	map at PageRank.scala:28 +details	2019/11/05 03:46:18	42 s	10/10	1009.4 MB			493.1 MB
12	map at PageRank.scala:28 +details	2019/11/05 03:46:18	15 s	10/10	1009.4 MB			493.1 MB
10	map at PageRank.scala:28 +details	2019/11/05 03:46:18	45 s	10/10	1009.4 MB			493.1 MB
8	map at PageRank.scala:28 +details	2019/11/05 03:46:18	59 s	10/10	1009.4 MB			493.1 MB
6	map at PageRank.scala:28 +details	2019/11/05 03:46:18	17 s	10/10	1009.4 MB			493.1 MB
5	mapValues at PageRank.scala:43 +details	2019/11/05 03:46:18	18 s	10/10	106.4 MB			29.4 MB
3	mapValues at PageRank.scala:43 +details	2019/11/05 03:46:18	7 s	10/10	106.4 MB			29.4 MB
2	map at PageRank.scala:28 +details	2019/11/05 03:46:18	54 s	10/10	1009.4 MB			493.1 MB

For PageRank, the flatMap() method takes the most time. I used the join() function, which returns a dataset of (K,(V,W)) when we call it on dataset type (K,V) and (K,W). SortByKey() function is used call on a dataset (K,V) pairs where K implements Orders, returns a dataset (K,V) pairs sorted by keys. This may take a long time since there are a lot of pairs. Also, spark has Lazy Evaluation property, so operations are not executed immediately, which adds the latency.

I apply zip() function to zip outlink and inlink, together with a key-value pair in RDD form. I also use join() and flatMap() here to get the pair of one link and the rank of each element in the array. I use reduceByKey((a,b) — a+b) function to count the occurrence of each lines of text in the file, and I used union() function to get a new dataset that contains the union of the elements in the source dataset. This way we can calculate the rank for each link, the functions retrieve original links and title RDD to compute the ranks of each link, which takes a long time.