HW 3 - Report

Table of Contents

1. Design Discussion

1.1 Pre-Processing Job

I have followed the same per-processing approach as described in the Assignment. I have used given SAX XML Parser. I have modified three things in the given parser:

1. Do not include self links
2. Removed the duplicate nodes from the adjacency list
3. Fixed the “ & “ issue, some pages was not parsed due to this issue.

I have created my own input file which covers all cases required in assignment and calculated manual page ranks for this input for 10 iterations. And then compared each output of my all programs to the manual calculation and they are the same.

PreProcessingJob emits the nodeId and its adjacentList for all successfully parsed documents. And the job returns the total number of pages from global counter.

Psuedo-Code

```
class Mapper
{
        map(Line line)
        {
                (nodeID, adjacencyList) = parse(line);

                // Emit nodeId with its adjacencyList
                emit(nodeId, adjacencyList);

                // Emit each adjacentNode with empty adjacencyList
                // If some nodes are in adjacencyList but do not have document for them
                // then this dangling nodes need to be considered
                for(String adjacentNodeId : adjacencyList)
                {
                        emit(adjacentNodeId, EMPTY_LIST);
                }
        }
}
```

```
class Reducer
{
	Long localNumberOfPages;

	setup()
	{
		localNumberOfPages = 0;
	}

	reduce(nodeId, [adjacencyList1, adjacencyList2,..])
	{
		adjacencyList = EMPTY_LIST;

		// get the first non empty adjacencyList if present
		// There can be multiple documents with same nodeId in input
		// so consider the first one only
		for each val in input list do
		{
			if(val is not EMPTY_LIST)
			{
				adjacencyList = val;
				break;
			}
		}

		// increment the number of pages
		localNumberOfPages++;

		// emit nodeId, its adjacencyList and its default page rank
		emit((nodeId, adjacencyList), NULL);
	}

	// cleanup increments the global counter by localNumberOfPages count.
	cleanup()
	{
		increment global counter with localNumberOfPages;
	}
}
```

On successful completion of above job we will return the total number of pages from global counters. Total number of pages should be same as number of reduce outputs.

1.2 Page Rank Job

For calculating page rank (computing dangling factor delta), I have used Solution – 2 (Merge computation of delta in previous reduce phase) approach as per Learning module 6 - Graph Algorithms. Below are the all three approaches for calculating delta and the reasons why I have choose solution 2.

Solution-1 – Add a separate phase to each iteration to compute delta
During an iteration, first execute a Map Reduce program that computes delta. This is a simple global aggregation job, summing up Page Rank values for all dangling nodes. Then pass the newly computed delta as a parameter to modified Map Reduce program that updates all Page Ranks using new formula with delta. The drawback of this approach is that, in each iteration we are reading all the records in map to calculate the delta in global aggregation job. So here we have an overhead of reading all the inputs for each iteration.

Solution-2: Merge computation of delta in previous reduce phase. (Used Approach)
In this approach, instead of computing delta in a separate job in the beginning of iteration (i+1), it could already be computed at the end of iteration i in reduce call. In this approach we don't need the extra reading of all input records as we need in above approach. Only problem with this approach is, the page rank emitted by the reducer is not the correct one so in last iteration we need to add one more map only job to correct the page ranks calculated by the last iteration. In our program, we have a PageRankCorrectionJob which does the same. So here we can argue that we need one extra map read of all the inputs per entire job, but this can also be eliminated. In our program, We just need TopK page ranks and we can correct the output emitted by the last iteration in TopK job. But this will give the correct page rank of only top K pages and it is possible that we might need other records in future. So for the safer side, I have provided PageRankCorrectionJob to run before TopKJob.

Solution-3 – Order Inversion
The order inversion pattern can be applied to make sure each Reducer receives the old PageRank values of all dangling nodes. Each reducer can then compute delta right before executing any of the normal reduce calls. This approach has the drawback same as what we have seen in other Order Inversion algorithms. We need to have one reduce task only so we cannot have reduce task granularity. We can overcome this issue by emitting multiple copies of same dangling node's page rank such that all reducers receive exact one copy of it. But this creates map output duplication issue. We can overcome this output duplication issue in some extent by in-mapper combining. So this approach with in-mapper combining to dangling factor output in each map task and emitting that combined delta to all reducer (multiple reduce tasks) can be a good solution. But as per our problem needs we required only top k page ranks and in our solution 2 approach we can do that without using PageRankCorrectionJob which makes Solution 2 better than other two approaches.

Pseudo Code:

```
// Map processes the node with id n.
// N stores node n's current PageRank and its adjacency list
map(nid n, node N)
{
	if( isfirstIteration )
	{
		N.pageRank = 1/|V|
	}
	else
	{
		delta = getDeltaFromConfiguration()
		N.pageRank = alpha/|V| + (1-alpha)(delta/|V| + N.pageRank)
	}

	// Pass along the graph structure
	emit(nid n, N)

	if(|N.adjacencyList| != 0)
	{
		//compute contributions to send along outgoing links
		p = N.pageRank / |N.adjacencyList|
		for all nid m in N.adjacencyList do
			emit(nid m, p)
	}
	else
	{
		emit("Delta", N.pageRank)
	}
}

// In code, I have used In-mapper combining to combine delta and also used Combiner to
// combine the map outputs

// Reduce receives the node object from node m and
// the PageRank contributions for all m's inlinks
reduce(nid m, [p1, p2,...])
{
	s=0
	M=NULL

	for all p in [p1,p2,...] do
		if isNode(p) then
```

```
                    // The node object was found : recover graph structure
                    M = p
            else
                    // A PageRank contribution from an inlink was
                    // found : add it to the running sum
                    s += p1

    if (m == “Delta”)
            incrementGlobalCounter(“Delta”, s)
    else
            M.pageRank = s
            emit(nid m, node M)
}
// After each iteration we will pass computed delta from global counters to next job’s
configuration.
```

1.3 Top-K Job

There are two approaches to find the top-K elements. First is to sort the input data and then the K largest records can easily be selected from the sorted file. This is an efficient method for very large K. Second approach is to avoid sorting if we have smaller value of K. The idea is to scan the input only once and use in-mapper combining to keep track of the top-K records in each map task. A single reduce call receives these local top-k lists and merges them into final result.

In our assignment K=100 which is a very small number, So I have used the second approach and not sorting as we do not need to transfer all input records from map to reduce, read again in reduce and write from reduce to file.

Pseudo-Code:

Pseudo code is same as the pseudo code given in Module – 5 : Basic Algorithms.
[2.13: Top-K records – Second Approach – find the local TopK in each Map task and merge them into global TopK in one reduce task.]
Addition to the given code, I have implemented MyTreeMap to handle the different pages with same page rank issue.

1.4 Data Transferred

Below table shows the amount of data transferred from Mappers to Reducers, and from Reducers to S3 (on AWS, job has configured to write on S3 instead of HDFS) in each phase of the entire program execution and in each iteration of page rank job. Data from syslog file of AWS EMR run of Wikipedia-full-html input execution on 11 machines (1 master and 10 workers).

Phase - Iteration	S3: Number of bytes read (S3 to Mappers)	Map output bytes	Reduce shuffle bytes (Combiner/Mapper to Reducer)	S3: Number of bytes written (Reducer to S3)
Pre-Pro.	7091101822	2088430787	1008848757	1057795115
1	1057795115	3210315684	1217336429	1180704724
2	1180704724	3214348041	1317781918	1182955026
3	1182955026	3212780522	1317996064	1181970571
4	1181970571	3213168630	1318206121	1183014094
5	1183014094	3213382361	1318341266	1183023055
6	1183023055	3213765772	1318386360	1183036262
7	1183036262	3212932819	1318206906	1181992025
8	1181992025	3212833986	1318236724	1181995559
9	1181995559	3214408272	1318455864	1183051088
10	1183051088	3213143682	1318350598	1181993998
Correction	1181993998	Map Only Job, Mapper to S3 : 142752032 bytes		
TopK	142752032	65692	59740	3211

Below table shows the number of records transferred between mapper, combiner and reducer. Data from syslog file of AWS EMR run of Wikipedia-full-html input execution on 11 machines (1 master and 10 workers).

Iteration	Map input records	Map output records	Combine input records	Combine output records	Reduce input records	Reduce output records
Pre-Pro.	7012253	54626996	54626996	19434887	19434887	3178227
1	3178227	54718394	54718394	19556653	19556653	3178227
2	3178227	54718394	54718394	19562914	19562914	3178227
3	3178227	54718394	54718394	19559839	19559839	3178227
4	3178227	54718394	54718394	19560635	19560635	3178227
5	3178227	54718394	54718394	19561348	19561348	3178227
6	3178227	54718394	54718394	19561719	19561719	3178227
7	3178227	54718394	54718394	19560157	19560157	3178227
8	3178227	54718394	54718394	19560043	19560043	3178227
9	3178227	54718394	54718394	19562772	19562772	3178227
10	3178227	54718394	54718394	19560535	19560535	3178227
Correction	3178227	3178227	Map Only Job			
TopK	3178227	1900	0	0	1900	100

In pre-processing, the input document size is very high. But in pre-processing, we discard most of the data from document and end up to only graph representation (nodeId : Adjacency List). So output of the reducer to S3 is comparatively smaller.

In first iteration, in input file (S3 to mapper), we have only nodeId and adjacencyList, but after processing the first iteration of page rank, reducers emit the page rank of each page along with its nodeId and adjacencyList. This increases the number of bytes written to S3 from reducers for first iteration.

For all other iteration, there is no bigger difference in amount of data transferred in each step, as we are processing & passing the kind of same data and just modifying the page rank values.
In each iteration, amount of data transferred from Mapper to Reducer increases because we are passing the page rank contribution to each adjacent node along with node structure entries. Though Combiner helped a lot to reduce the reduce shuffle bytes from original Map output bytes. Reducer combines the map outputs and writes it to S3 so the number of bytes reduces again.

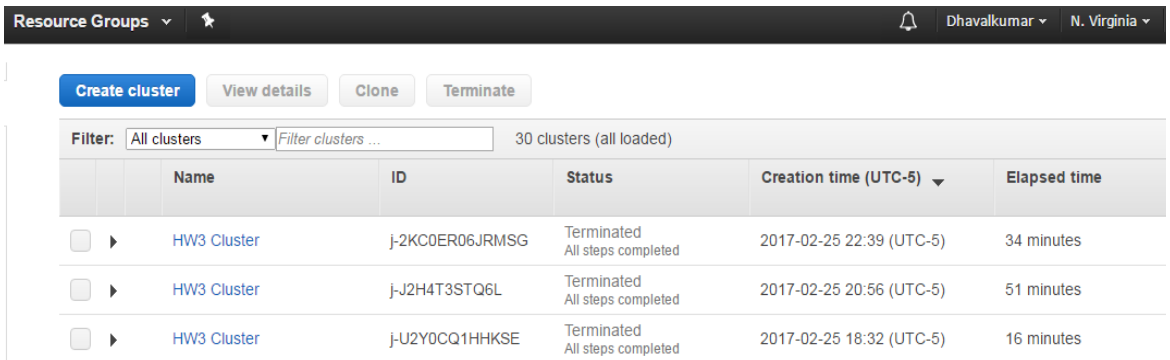
Last Correction job writes only NodeId and PageRank values amount of data transferred to S3 from reducer drastically reduced same for TopK(only K=100 records written) job as well.

2. Performance Comparison

2.1 Running Time

I have executed program on AWS EMR for both the inputs. I have used alpha = 0.15 for my page rank calculations (Source: Wikipedia page of page rank algorithm). Though the value of alpha is not hardcoded and you can configure in Makefile.

Below is the screenshot of AWS EMR successful run:



1st row: Wikipedia-full-html input run on 11 m4.large machines (1 master and 10 workers)
2nd row: Wikipedia-full-html input run on 6 m4.large machines (1 master and 5 workers)
3rd row: Wikipedia-simple-html input run on 6 m4.large machines (1 master and 5 workers)

Below is the running time of full Wikipedia input on both configurations:

(Time in Seconds)

Job	6 m4.large Machine	11 m4.large Machines
Pre-Processing Time	1021	631
Time to Run ten Iterations Of Page Rank	1532	947
Time to Find The Top -100 Pages	40	29

2.2 Running Time Comparison

As we see in running times, all the computation phases shown good speed up by increasing machines as expected. With higher number of machines, our work load is divided among them and due to parallel work overall time is reduced.

If we compare the speed up of all computational phases then first two phases (PreProcessing and PageRank) has shown higher speed up then the last phase (TopK). The running time decreased to 38% in first to phases while it reduced 27% in last phase. I have also expected the same as in first two phases our MapReduce framework utilizes all the machines whereas in TopK job we are creating only on reduce task so we cannot get advantage of increase in machines in reduce work of TopK job.

Number of reduce task depends on the available nodes to MapReduce framework. As per hadoop documentation "*The right number of reduces seems to be 0.95 or 1.75 multiplied by (<no. of nodes> * <no. of maximum containers per node>).*" If we have higher number of machines then it will create more reduce task and we will get good parallelism which will reduce the time taken and increase the performance.

In our case for first two phases, for 11 machines, total number of reduce task is 20 whereas for 6 machines they are reduced to 10. So in first two phases, higher machine gave better performance as the load is distributed among them nicely. For last phase, number of reduce tasks are independent of machines as we are setting it manually to 1. So here we are not getting advantage of increase in machines in reduce phase and it shows lesser speed up then above two phases. Adding to

2.3 Top-100 Wikipedia Pages

Below is the top-100 Wikipedia pages with highest page rank along with their page values sorted from highest to lowest for both the simple and full datasets. I find them reasonable as I guess there will be higher in links to the pages like United_States_09d4 from the other pages which result into higher page rank of this page. And we can see other countries as well in the list this means, the other wiki pages will have links in their web page to the related country page and which will boost the page rank of the country pages.

No.	Wikipedia-full-html-output-11-machines		Wikipedia-simple-html-output-6-machines	
	Page Name	Page Rank	Page Name	Page Rank
1	United_States_09d4	0.002622889346164360	United_States_09d4	0.005189009000272900
2	2006	0.001228497839402600	Wikimedia_Commons_7b57	0.004806766474708800
3	United_Kingdom_5ad7	0.001203136760956090	Country	0.003940284687712630
4	Biography	0.000982071558189593	England	0.002752481436110630
5	2005	0.000917059863941628	Water	0.002687809623446530
6	England	0.000880207134127707	Animal	0.002554087565149160
7	Canada	0.000855905079347993	City	0.002510824080782450
8	Geo-graphic_coordinate_system	0.000771726229889266	United_Kingdom_5ad7	0.002358647093612220
9	France	0.000725024187225341	Germany	0.002350401697711510
10	2004	0.000719896807279462	Earth	0.002324734859954630
11	Australia	0.000680476695292216	France	0.002323607947142090
12	Germany	0.000654345280141820	Europe	0.002038097037167730
13	2003	0.000587386977828930	Wiktionary	0.001753884214276070
14	India	0.000583419774005353	English_language	0.001749677121754420
15	Japan	0.000582856047373971	Government	0.001732344652103290
16	Inter-net_Movie_Database_7ea7	0.000533506966003842	Computer	0.001716840484713360
17	Europe	0.00050926763789	India	0.00171317091838

		9670		4910
18	Record_label	0.000491459567584876	Money	0.001667383698022790
19	2001	0.000487012158303275	Japan	0.001551690568535440
20	2002	0.000482863246819372	Plant	0.001523559509359920
21	World_War_II_d045	0.000478050436747285	Italy	0.001507433090497990
22	Population_density	0.000470342996138859	Canada	0.001481407343452890
23	Music_genre	0.000467211751796516	Spain	0.001471123692223520
24	2000	0.000464661032401972	Food	0.001424686848967640
25	Italy	0.000445792746035966	Human	0.001412097006269630
26	Wiktionary	0.000436209787029988	China	0.001396715061272910
27	Wikimedia_Commons_7b57	0.000435294723979010	People	0.001382248525055760
28	London	0.000434802659060111	Australia	0.001329854240750500
29	English_language	0.000418503522954299	Asia	0.001284436171136100
30	1999	0.000405936985787724	Capital_(city)	0.001274268421251940
31	Spain	0.000362953791929990	Television	0.001264997225760360
32	1998	0.000356310636734167	Sun	0.001260210081178010
33	Russia	0.000343906624978832	Number	0.001243236228928820
34	1997	0.000337282024933117	State	0.001240375681454610
35	Television	0.000336297123800393	Sound	0.001235211667221920
36	New_York_City_1428	0.000334628776843839	Science	0.001232543175359430
37	Football_(soccer)	0.000326146421344763	Mathematics	0.001231056639295570
38	1996	0.000323627866091999	Metal	0.001192304623749440
39	Census	0.000323553371886843	Year	0.001177092583510610
40	Scotland	0.000322189155586006	2004	0.001173357313768480

41	1995	0.000310154641047029	Language	0.001150165884857740
42	China	0.000308643012896061	Russia	0.001146181779212580
43	Population	0.000304320482640489	Wikipedia	0.001123330280988190
44	Square_mile	0.000304056100720908	Religion	0.001098566699966040
45	Scientific_classification	0.000304011968028030	19th_century	0.001096539141780080
46	California	0.000301667404294186	Music	0.001087431323214420
47	1994	0.000290691158575429	Scotland	0.001054800735006320
48	Sweden	0.000287620803401212	20th_century	0.001053704983258870
49	Public_domain	0.000287416106781891	Greece	0.001049222732934630
50	Film	0.000286269139275013	Latin	0.001029860613187450
51	Record_producer	0.000284110190159314	London	0.001027355442851320
52	New_Zealand_2311	0.000283102054245546	Greek_language	0.001004357256650290
53	New_York_3da4	0.000278882635557677	Energy	0.000999011810379402
54	Netherlands	0.000276673678318118	World	0.000986350847997666
55	Marriage	0.000275813292268387	Centuries	0.000975905865136575
56	1993	0.000274824898720027	Culture	0.000945203965211297
57	United_States_Census_Bureau_2c85	0.000274667106948521	History	0.000936469603425431
58	1991	0.000271895256129761	Liquid	0.000914523096800025
59	1990	0.000268326118798734	Netherlands	0.000905724507648975
60	1992	0.000266369246377862	Planet	0.000904932262239007
61	Politician	0.000264894907636039	Light	0.000901676352686388
62	Album	0.000260564453918753	Society	0.000901492062145202
63	Latin	0.000260456361478826	Atom	0.000890022640652958
64	Actor	0.00025833956084	Wikime-	0.00088844007077

		6043	dia_Foundation_83d9	6097
65	Ireland	0.000258106387703505	Scientist	0.000888383610573494
66	Per_capita_income	0.000255642724549733	Image	0.000887688486021999
67	Studio_album	0.000251860261600276	Law	0.000886290805598419
68	Poverty_line	0.000251165008634912	Geography	0.000878845161454898
69	Km²	0.000249506597225139	List_of_decades	0.000878574294283701
70	1989	0.000246893732523362	Uni-form_Resource_Locator_1b4e	0.000861884506363216
71	Norway	0.000240921928712262	Africa	0.000860569967152458
72	Website	0.000239017858425252	Turkey	0.000844886367889011
73	1980	0.000235321990653687	Inhabitant	0.000830479488232369
74	Animal	0.000229378639768423	Capital_city	0.000823048814043760
75	Area	0.000229208702962197	Plural	0.000821515595510230
76	1986	0.000227033046834427	Electricity	0.000813723001666497
77	Personal_name	0.000226236538987703	Poland	0.000797237904315345
78	Poland	0.000226115442905689	Building	0.000797123892572039
79	Brazil	0.000225684266637636	Car	0.000794654060623909
80	1985	0.000224029069042517	Sweden	0.000791712556234121
81	1987	0.000223305403258049	Book	0.000791488470531959
82	1983	0.000221753386636217	Biology	0.000786932896431402
83	1982	0.000221096532736380	War	0.000770817294548049
84	French_language	0.000219384526654706	Chemical_element	0.000768160795919691
85	1981	0.000219348011952457	God	0.000760935721891388
86	1979	0.000219328593475849	North_America_e7c4	0.000756286864416695
87	1984	0.00021879019421	September_7	0.00075477818126

		2315		4075
88	World_War_I_9429	0.000218692393695080	Website	0.000746297350060417
89	1988	0.000218574180218445	Nation	0.000742667152640605
90	Paris	0.000218019680115686	Politics	0.000739710378758902
91	1974	0.000217974864664915	2006	0.000733290017225940
92	Mexico	0.000215673596427163	Fish	0.000732237111290969
93	19th_century	0.000211856357736199	Species	0.000730871117629328
94	1970	0.000211324295679975	Mammal	0.000721674413594913
95	January_1	0.000210864269910462	Island	0.000717809020303589
96	USA_f75d	0.000210708682432733	Portugal	0.000717107059660586
97	1975	0.000208601835120045	Gas	0.000715551536653913
98	1976	0.000208467449973305	River	0.000711577751300913
99	Africa	0.000207798052562159	Switzerland	0.000706107507438502
100	South_Africa_1287	0.000207362441973637	World_War_II_d045	0.000702030493158166

3. Convergence Estimation

I have coded to measure the convergence achieved after each iteration of page rank calculation. In each iteration, program calculates the absolute difference in old and new page rank of each node and sums it up. And at the end of iteration, the sum is divided by the number of total pages to get the average difference per node and logs it to log file.

Below is the convergence output from logs of AWS EMR execution of Wikipedia-full-html input with both configurations:

Iteration	6 m4.large Machine	11 m4.large Machines
1	0.000000184996236977	0.000000184996609594
2	0.000000076656124798	0.000000076656300012
3	0.000000030632383622	0.000000030632328128
4	0.000000013365304956	0.000000013365282971
5	0.000000006365162230	0.000000006365152615
6	0.000000003233759976	0.000000003233754581
7	0.000000001721090299	0.000000001721086772
8	0.000000000951050433	0.000000000951048336
9	0.000000000543148803	0.000000000543147593

(Note: It doesn't calculate for last iteration, as last page rank is corrected in PageRankCorrectionJob)

As we can see in above diagram, average difference in page rank in each iteration reduces which means page ranks are converging.

If we need the convergence up to specific decimal point then we can count the number of nodes whose page rank change is higher than specified margin in each iteration. This count will decrease in each iteration and when it reaches to zero, we will stop calculating the page rank.