HW 5 - Report

Table of Contents

1. Design Discussion

1.1 Pre-Processing Job

I converted page names to unique node Ids in Pre-Processing job. I made this job as a single reducer job so that we can assign the unique Ids to each node. In reducer, I am keeping the global map with page name as key and node id as value and whenever new page comes, it assigns the new Id and adds it to map.

PreProcessingJob emits the nodeId and its adjacentList (all Ids as an integer) for all successfully parsed documents in one folder and the node Id to node Name mapping in another folder. And the job returns the total number of pages from global counter.

Psuedo-Code

```
class Mapper (No Change)
{
	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
	HashMap<pageName, nodeId> nodeIdMap
	CurrentId = 0

	setup()
	{
		localNumberOfPages = 0;
		nodeIdMap = new Map();
	}
```

```
        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++;

                if nodeId and adjacentList has pageName which are not yet found then assign new Id to
                them and add into map
                else replace them with integer and emit them

                // 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;
                emit(nodeIdMap)
        }
}
```

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 MatrixGeneration Job

1.2.1 For Row by Column approach

Below is the pseudo code. The detailed design and working of the algorithm is described by an example after this chapter.

```
map(nodeId, adjacencyList)
{
    emit(nodeId, "") // to handle nodeId with no inlinks nodes
    for each adjacentNodeId in adjacencyList)
            emit(adjacentNodeId, (nodeId, 1/adjacencyList.length))
}

reduce(nodeId, [inlink1, inlink2, ...])
{
    emit(nodeId, input List) // to matrix files
    emit(nodeId, (initial page rank, isDanglingNodeFlag)) // to page rank files
}
```

1.2.2 For Column by Row approach

Below is the pseudo code. The detailed design and working of the algorithm is described by an example after this chapter. Only generates the Initial page rank output.

```
map(nodeId, adjacencyList)
{
    emit(nodeId, (initial page rank,isDanglingNodeFlag)) // to page rank files
}
```

1.3 PageRank Job (Matrix Multiplication)

1.3.1 For Row by Column approach

Below is the pseudo code. The detailed design and working of the algorithm is described by an example after this chapter. The page rank calculated in the previous iteration is shared in distributed cache. (Replicated join approach)

```
Map
{
    double[] pageRanks = new double[noOfPages];
    double delta = 0.0

    setup()
    {
        for each line in the page rank file from distributed cache
        {
            line -> (nodeId, (pageRank, isDanglingNodeFlag))
            pageRank[nodeId] = pageRank

            if(isDanglingNodeFlag)
                delta += pageRank * 1/noOfPages
        }
    }
    map(nodeId, inLinks)
    {
        double pageRankSum = delta;
        for each inlink
            pageRankSum += pageRanks[inlink.nodeId] * inlink.ratio
        pageRankSum = alpha/noOfPages + (1-alpha) * pageRankSum
        emit(nodeId, (pageRankSum, isDanglingNodeFlag))
    }
}
```

1.3.2 For Column by Row approach

Below is the pseudo code. The detailed design and working of the algorithm is described by an example after this chapter. The page rank calculated in the previous iteration is shared in distributed cache. (Replicated join approach)

```
Map
{
    double[] pageRanks = new double[noOfPages];
    double delta = 0.0


setup()
    {
        for each line in the page rank file from distributed cache
        {
            line -> (nodeId, (pageRank, isDanglingNodeFlag))
            pageRank[nodeId] = pageRank

            if(isDanglingNodeFlag)
                delta += pageRank * 1/noOfPages
        }
    }
    map(nodeId, outLinks)
    {
        // emit the contribution to each adjacent node
        for each outLinks
            emit(outLink.nodeId, pageRanks[nodeId] * outLink.ratio)
        emit(nodeId, delta)
    }
}

reduce(nodeId, [pageRank1, pageRank2,...])
{
    double pageRankSum = 0.0
    for each rankContri in input list
        pageRankSum += rankContri

    pageRankSum = alpha/noOfPages + (1-alpha) * pageRankSum
    emit(nodeId, (pageRankSum, isDanglingNodeFlag))
}
```

1.4 Top-K Job (No Change)

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.

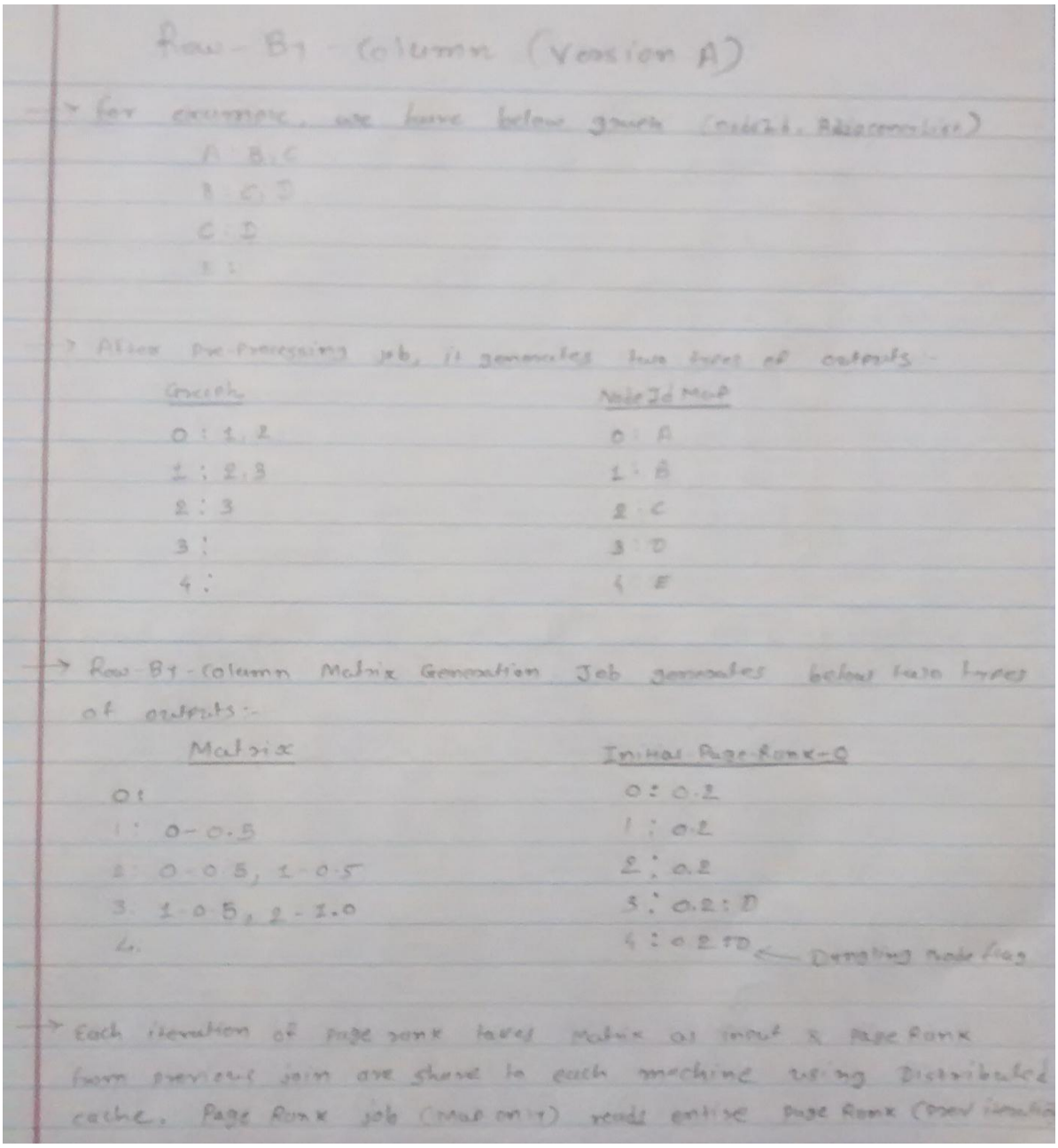
Pseudo-Code:

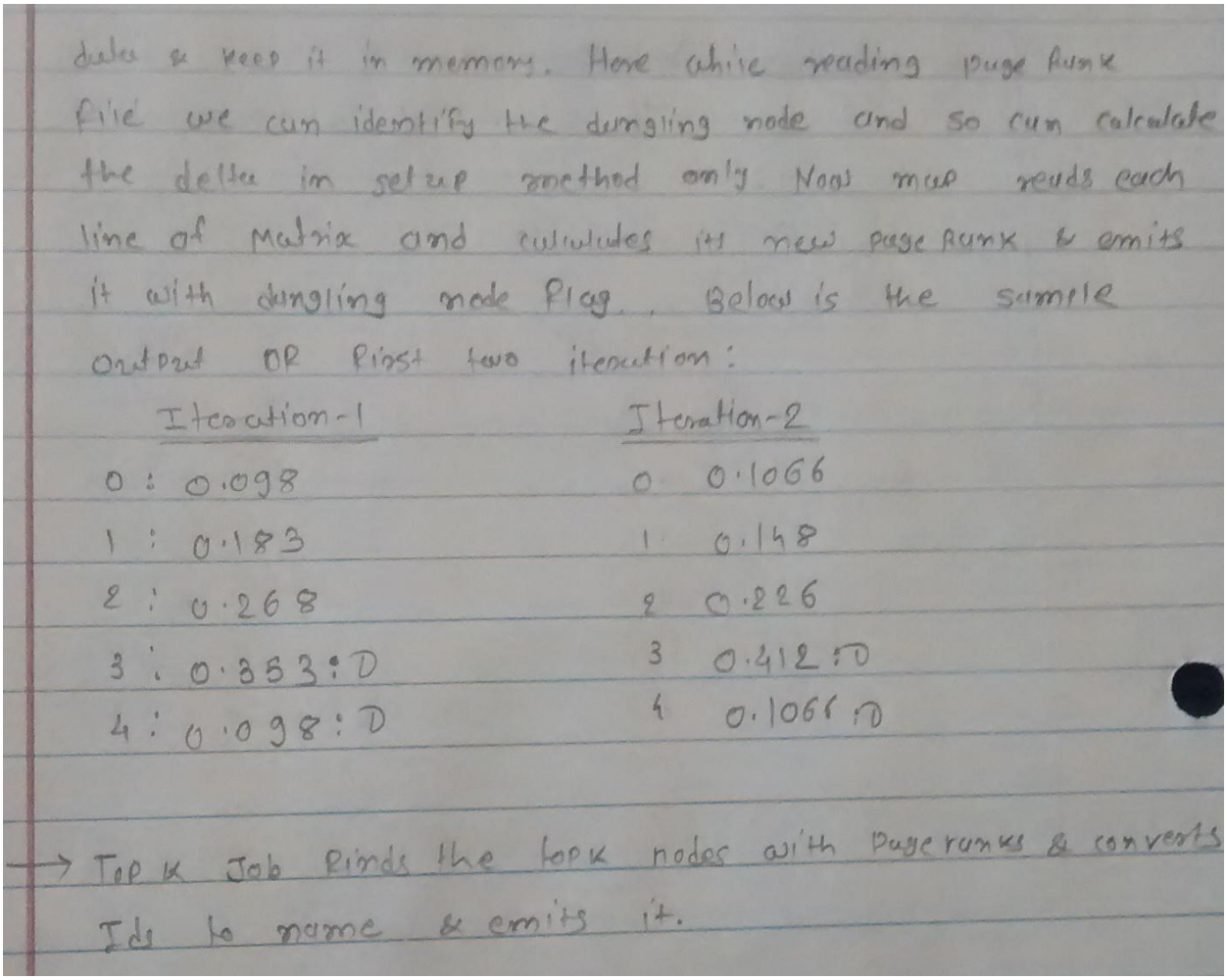
Pseudo code is same as the pseudo code given in Module – 5 : Basic Algorithms.
[2.13: Top-K records – Second Approach – find the local TopK in each Map task and merge them into global TopK in one reduce task.]
Addition to the given code, I have implemented MyTreeMap to handle the different pages with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

1.5 Matrix Representation and Dangling Node Handling

Row by column approach:

For row by column approach, the M' matrix or the graph's matrix representation is partitioned by each row. As per the algorithm we need to replicate the matrix B (page rank matrix) no of row times. But here we are sharing page rank matrix in distributed cache and the algorithm follows replicated join approach. So the page rank matrix will be duplicated #of machines times only.





Column by Raw approach:

For column by row approach, the M' matrix or the graph's matrix representation is partitioned by each column. As per the algorithm we need to replicate the matrix B (page rank matrix) no of column times. But here we are sharing page rank matrix in distributed cache and the algorithm follows replicated join approach. So the page rank matrix will be duplicated #of machines times only.

Column-By-Row (version 8)

→ for example, we have below graph (nodeId, Adjacency List)

A: B,C
B: C,D
C: D
E:

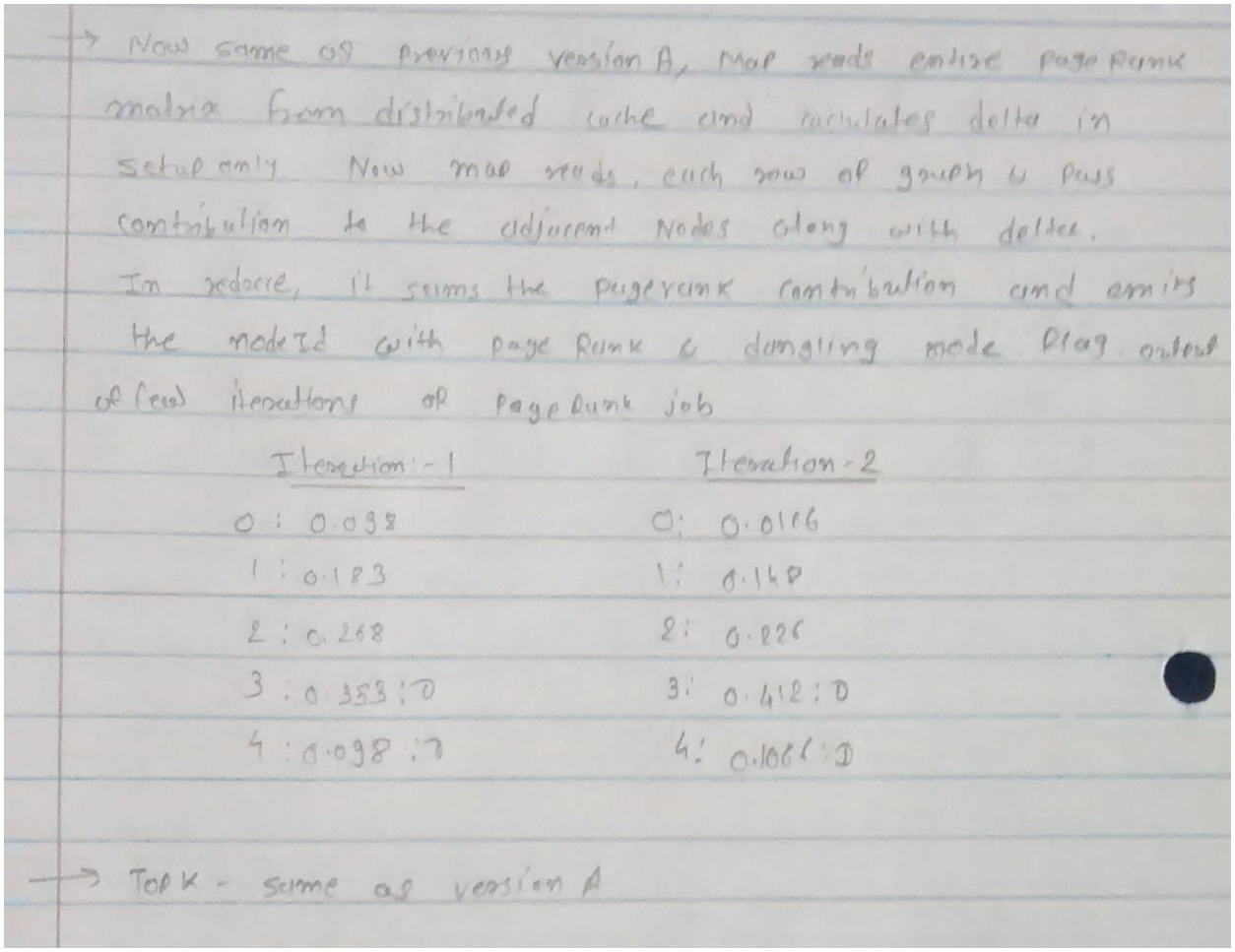
→ After preprocessing job, it generates two types of outputs

Graph	NodeId Map
0 : 1, 2	0 : A
1 : 2, 3	1 : B
2 : 3	2 : C
3 :	3 : D
4 :	4 : E

→ Column By Row Matrix Generation Job generates only Initial page Rank output as the Matrix is same as graph structure:

Initial-Page-Rank-0

0: 0.2
1 : 0.2
2: 0.2
3 ; 0.2 : D
4 ; 0.2 : D



→ Now same as previous version A, Map reads entire page Rank matrix from distributed cache and calculates delta in setup only. Now map reads, each row of graph & pass contribution to the adjacent Nodes along with delta. In reduce, it sums the pagerank contribution and emits the nodeId with page Rank & dangling node Prog. output of few iterations of PageRank job

Iteration:-1	Iteration-2
0 : 0.098	0: 0.0166
1 : 0.183	1: 0.148
2 : 0.268	2: 0.225
3 : 0.353 : D	3: 0.412 : D
4 : 0.098 : D	4: 0.1661 : D

→ Top K - same as version A

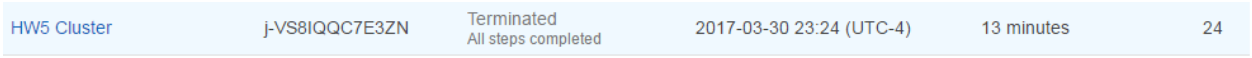
2. Performance Comparison

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:

Wikipedia-simple-html input run on 6 m4.large machines (1 master and 5 workers) – Version A

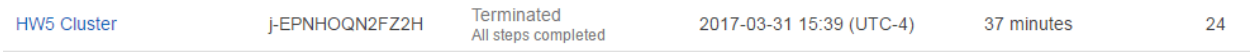


Wikipedia-simple-html input run on 6 m4.large machines (1 master and 5 workers) – Version B

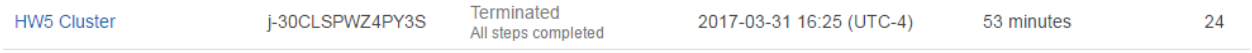
Wikipedia-full-html input run on 11 m4.large machines (1 master and 10 workers) – Version A

Wikipedia-full-html input run on 11 m4.large machines (1 master and 10 workers) – Version B

Wikipedia-full-html input run on 6 m4.large machines (1 master and 5 workers) – Version A



Wikipedia-full-html input run on 6 m4.large machines (1 master and 5 workers) – Version B



Below is the running time of full Wikipedia input on both configurations:
Note: My programs works with original input files.

(Time in Seconds)

Job	Version A		Version B		HW3	
	6 machines	11 machines	6 machines	11 machines	6 machines	11 machines
Pre-Processing Time (Step 1&2)	1139	705	1279	636	1021	631
Time to Run 10 Iterations of Page Rank (3)	542	514	1270	976	1532	947
Time to Find The Top -100 Pages (4)	40	39	42	39	40	29
Total Time	1721	1258	2591	1651	2593	1607

2.2 Running Time Comparison

As we can see in the version A (row by column multiplication approach) of HW5 is faster than the other two, and Version B and HW3 total running times are almost same.

Version A vs. Version B or HW3

As in version A of the program the page rank calculation is very faster than other approaches. This is because we have an in link kind of graph here and for each node in map we can calculate its page rank and not needed the reduce job (Used Replicated join approach). So here we needs Map only job. In version B, in each iteration we are sending 953998514 bytes to reducer to process which we do not need in Version A and this gives us the better performance.

Version B vs. HW3

The total running time of both the algorithm is almost same. As if we see in detail, for both the solution we have a kind of same input structure (nodeId, adjacencyList) and we send contribution to each node in map and reducer combines the total. So logic is almost the same. But if we see step wise, then Page rank calculation of Version B is faster than HW3. This is because we are not sending entire node structure to reducer in each iteration. Data transferred from mapper to reducer in Version B in each iteration is 953998514 bytes whereas it is 1318314419 bytes in HW3.

2.3 Top-100 Wikipedia Pages

The top-100 Wikipedia pages with highest page rank along with their page values sorted from highest to lowest are provided at below paths. The results of the page ranks are almost the same. The pages are ordered perfectly the same in all execution but there are very minor differences in the page rank values (at 12^{th} decimal point). This minor difference can be because of our double to long conversion in MapReduce program while storing delta or can be dependent on double precision calculation of the specific environment. As the differences are very minor and page rank order is perfectly same we can ignore them.

4_AWSOutputFiles/Wikipedia-full-html-VersionA (2 files for each configuration)
4_AWSOutputFiles/Wikipedia-full-html-VersionB (2 files for each configuration)
4_AWSOutputFiles/Wikipedia-simple-html-VersionA (1 file for 1 master 5 worker configuration)
4_AWSOutputFiles/Wikipedia-simple-html-VersionB (1 file for 1 master 5 worker configuration)