Assignment - 5
Section-01
Satya Akhil Chowdary Kuchipudi

Pseudo Code :

ROWXCOLUMN :

ıNDEX CONSTRUCTION FOR NODES

```
Reducer

// Main Job is to create index for easier read
// Used Counters
COUNTERS = { total_nodes, danglingloss }

Mapper

hashmap = {}

// Read line and use a counter to keep track as index
map(offset B, line L)
    increment total_nodes by 1
    parts[] = parse nodeid and adjacency list from line L
    nodeid = parts[0]
    adj_list[] = parts[1]

    mapuploader(nodeid)

    if adjlist.length > 0
        adj_list foreach(helper1(adjacency_element))

// for inmapper combiner, emit out once map tasks are finished
def mapuploader(nodeid)
    if hashmap not contains nodeid
        emit(1, nodeid)
        hashmap.add(nodeid, adj_list)


Reducer

hashmap = {}
counter = 0
```

```
// for each key emit out the node as well as index
reduce(key , [node1, node2,...])

        for each iterable[nodes]
                if hashmap.keyset not contains node
                        emit(node, counter)
                        counter++
```

// Main purpose of this job is to build matrix M file for further pagerank calculations

MATRIX CONSTRUCTION

Reducer = 1

Mapper

```
map [string, int]

// DFC will hold index file for lookup and update nodes with indexed
dfc - index output

// loads the index file into map for lookup
setup()
        paths[] = read paths from distributed file cache
        read each file from the path and add the index data into map(nodeid => index)

// for each node in each line, update nodeid with index from map
map(offset B, line L)
        input[] = parse nodeid and adjacency list from line L
        nodeid = input[0]
        adjlist[] = input[1]

        if adjlist.length > 0
                foreach adjnode in adjlist
                        text = text + map[adjnode] + "##" + (1/adjlist.length)
                        text = text + ","
        emit(1, text)
```

Reducer

```
map [string, int]
dfc - index output

// Reducer is used to find out the node that are missed from graph and add them into the graph
setup()
        paths[] = read paths from distributed file cache
        read each file from the path and add the index data into map(nodeid => index)
```

```
reduce(key , [value1, value2,...])
        // update each line according to the index and emit
        foreach iterable[values]
                parsedValue[] = parse each value based on delimiter
                // normal node
                if parsedValue.length >= 2
                        emit(nodeid, value)
                        remove nodeid from map
        // fr sink node , emit “”
           else
                //sink node
                emit(nodeid, "")

           // remove value from Distributed file cache to distinguish between secondary node

cleanup

        for key in map
                // emit empty value for all secondary nodes that are not in the graph
                emit(key, "")
```

RANK MATRIX CONSTRUCTION

// Defaults a rank vector with 1/v value. Used as 0th iteration result. No reducer needed.
Mapper (Force Reducer to set 0)

```
map(offset B, line L)
        input[] = parse nodeid and adjacency list from line L
        nodeid = input[0]
        adjlist[] = input[1]

        // initialize the rank of each node to default => (1/totalnodes)

        emit(nodeid, 1/nodes)
```

// Generates a new temp pagerank vector which misses nodes that don't have inlinks
ROW BY COLUMN - PRIMARY JOB

Mapper

map = [string, double]

Distributed file Cache - previous pagerank iteration result

// loads the rank file for next iteration calculation

```
// for each rowid, multiply all the adjacency nodes with the pagerank of that rowed and emit
// columnid as key
map(offset B, line L)
    input[] = parse nodeid and adjacency list from line L

    nodeid = input[0]
    pagerank = map[nodeid]
    adjlist[] = input[1]
    if adjlist.length > 0
        foreach adnode in adjlist
        emit(nodeid, pagerank)

    else
        rank = map[nodeid]
        increment danglingloss by rank
```

Reducer

```
// Add up all the values for that particular row.. This is actually sum of all matrices, however
since rank is 1-d vector it's much simplified
reduce(key , [value1, value2,...])
    double aggregated_value = 0.0
    foreach value in values
        aggregated_value += value;
```

ROW BY COLUMN SECONDARY JOB

//Calculates the final rank by adding the damping factor and the dangling loss

<apper

map = [string, double]

Dfc - loads previous page rank result file and uses it add dangling nodes loss to all nodes as well as add the missing nodes from previous iteration.

```
// Loads the pagerank of previous iteration and temppagerank of current iteration
// adds the missing nodes and dangling loss to all the page ranks
map(offset B, line L)
    input[] = parse nodeid and adjacency list from line L
    nodeid = input[0]

    danglingloss = danglingloss/totalnodes
    if nodeid exists in map
        //Calculate rank for primary node
        newrank = map[nodeid] + danglingloss
        pagerank = (0.15/totalnodes) + (0.85 * newrank)
        emit(nodeid, pagerank)
    else
        // Calculate rank for sink and secondary nodes
```

```
                    pagerank = (0.15/totalnodes) + (0.85 * danglingloss)
                    emit(nodeid, pagerank)
```

TOP K:

```
//New class for handling priority order
PageRankNode(nid, pagerank) { nid = nid,
pagerank = pagerank
}

// override compare to method to compare page ranks of rank node
compareto(newnode){
return Double.comapre(newnode.pagerank, this.pagerank)
}
}

Mapper
new priority_queue, map(nid, pagerank) {
//add to priority queue
priority_queue.offer(new PageRankNode(nid, pagerank));
}

// sorts based on pagerank
cleanup(){
while (i < 100)
priority_queue.poll(); emit(nid, pagerank)
}


Reducer
// emit only 100
reduce(null, [p1, p2, p3 ...]) {
for(p in [p1, p2, p3 ...] and count <100) {
}
emit (p1.nid, p1.pagerank)
count++ }
```

COLUMNXROW :

ColumnXRow uses same almost same code as RowxColumn except for a minor change in the RankMapper Class. The main change is that, when a line is read from Matrix M, instead of multiplying all adjacency nodes with pagerank of rowid, we multiply with adjacency pagerank with pagerank of columnid.

Mapper

map = [string, double]

Distributed file Cache - previous pagerank iteration result

```
// loads the rank file for next iteration calculation

// for each rowid, multiply all the adjacency nodes with the pagerank of that rowed and emit
// columnid as key
map(offset B, line L)
    input[] = parse nodeid and adjacency list from line L

    nodeid = input[0]
        adjlist[] = input[1]
    if adjlist.length > 0
        foreach adnode in adjlist
        pagerank = map[adnode]
        emit(nodeid, pagerank)

    else
        rank = map[nodeid]
        increment danglingloss by rank
```

Design Discussion:

Main : The input for all the further programs is actually adjacency output from hw3, not bz2 wiki files.

RowXColumn :

The basic idea, for this has been implemented by reading Matrix M while Rank Vector R in Distributed File Cache.

I am using sparse representation of Matrix and also few changes to the original representation in the given assignment M(i,j) means i points to j not otherwise. However, I achieved this without using transpose of matrix and slightly modifying the way matrix product is calculated.

I have 3 pre-processing jobs, 10 page rank iterations, 10 dangling jobs for each iteration and top-100 job.

Pre-processing :
1) Build Index
2) Build Matrix M
3) Build Intial Rank R

Initially, I updated M by replacing all the 0 with 1/v as discussed in the assignment. However, this lead to the worst speedup I have ever seen due to large data shuffle.
So I had to come up with a new method to handle dangling nodes. I had created a new job which takes care of dangling nodes and also nodes that don't have unlinks and therefore missed out from previous pagerank iteration. This is achieved by using new pagerank as DFC and looking up with original pagerank to see nodes that are missing and also adding dangling loss via counter while doing that.

Now coming to the transpose part, the RowXColumn idea is mainly handled by taking row of Rank vector and multiplying with Column of Matrix M, but hence our matrix in itself is transpose we do a modified product by actually multiplying Row of Rank Vector with Row of Matrix M. In Math, this might be weird but logically this is actually same as doing transpose and proceeding with multiplication normally. So consider 0:(1,a),(2,b),(4,c). For this sparse representation, we get pagerank of node 0 and then multiply with a, b and c and emit with 0, 1 and 2 as keys respectively. The idea behind this is it actually creates a matrix temporarily from each product and each is added in reducer based on key. So the final result would be a product between M and R.

In this, partition the Matrix M is partitioned into groups of rows and the Rank vector R will be duplicated across all partitions.

ColumnXRow :

ColumnXRow is slightly different from what was discussed in the module. The original idea would be to have Matrix M read while Rank Vector R be partitioned accordingly and the matrix be built from reducers. I did a small optimization and used Distributed File Cache for this as well. The main difference is when we multiplied each adjacency node with rowid. We wont do that here, main reason is we are not following the original representation of M. My matrix is actually a transpose of the required. So I modified a bit by doing it in the said way. For example, 0:(1,a),(2,b),(4,c). In this case, unlike how we did for previous method. We multiply each adjacent node pagerank with the contribution of that node and emit with columnid as key.

In this, Matrix M is partitioned into groups of columns. Rank Vector is supposed to be partitioned, but for efficiency purposes I have loaded into file cache and duplicated across all partitions

Matrix Representation :

I am using sparse matrix representation to save the Matrix M. For example

Graph:
0, [1, 3]
1, [0,2]
3, []

Matrix representation:
0:(1, val1), (3, val2)
1:(0, val4), (2, val5)
3:

Matrix M is serialized in the above format. In addition to that, during building matrix job I also add all the dangling nodes that are not in the graph into the graph for calculating loss.

0:(1, val1), (3, val2)
1:(0, val4), (2, val5)
3:
4:

The basic idea is 0 represents the row id (nodeid). (1, val) in this 1 represents column id and val represents the contribution from 1 to 0.

This is the same representation I used for both A & B.

Dangling Nodes :

I am not replacing the dangling nodes values in matrix M with 1/v to transform into M1. Instead I am running another job after each pagerank iteration to handle dangling loss and all the nodes that missed in the pagerank iteration. In the extra job, I compare index file to see what all nodes are missing and I will update the page ranks by adding dangling loss. And for other nodes, I ll add dangling loss to existing pagerank and written to file. For each iteration, this gives the correct pagerank output.

Performance Comparison (Matrix Representation):

	RxC 11 machines	RxC 6 machines	CxR 11 machines	CxR 6 machines
Step 1	3	4	3.5	3
Step 2	3	3	3.5	3
Step 3	30	45	29.5	46
Step 4	0.5	1	0.5	1

Performance Comparison (Adjacency List Representation):

	6 machines	**11 machines**
Time	58 min (~4 mins)	38 min (~3 mins)

Comparatively, Matrix ran faster than before. One reason is that, we didn't include Pre-processing job of adjacency list. And on other hand, there is much improvement in terms of Matrix. Like I could have joined multiple jobs and done it much efficiently. However, due to time constraints I couldn't achieve this. We could attribute this to amount of data shuffled across mapper to reducer. This would be less mainly because everything is converted in form of numerical index. However, this would be much more prominent if we were to run large number of iterations. One more reason could be, loading of so many files into distributed file cache.

Single Step :

Adjacency List Representation Shuffle bytes : 1205475680

Matrix Representation Shuffle bytes : 578222057(Normal Page Rank job) + 3361043 (Dangling job)

Top-100 Reports :

Matrix Representaion Full output :
/Users/akhil0/Documents/NEU/MR/hw5/report/full-new

Adjacency List Representaion Full output :
/Users/akhil0/Documents/NEU/MR/hw5/report/full-old

Matrix Representaion Simple output :
/Users/akhil0/Documents/NEU/MR/hw5/report/simple-new

Adjacency List Representaion Simple output :
/Users/akhil0/Documents/NEU/MR/hw5/report/simple-old

Results are slightly different compared to Matrix and List based representation. This was mainly because of converting and including dangling nodes also in the graph. In hw3, I calculated pagerank differently, in terms that I had actually used 11 jobs to update dangling loss across 10 iterations.

References :

1) Stack Overflow
2) Piazza