Name: Atindra Mardikar
Class: Nat Tuck (Tue/Fri 1:35-3:15pm)
HW-05 Report

Page Rank Using Matrix Multiplication Source Code:

In the submitted HW folder there is a folder called source code inside which the both the versions are present.

** To test update the arguments in the makefile for input.

Program Design:

Pseudo Code:

I have handled the dangling nodes by calculating DR and then doing DR+MR.

Column By Row:

So I have 2 jobs to implement DR and MR each.

Index Creation(2 MapReduce jobs):

Job 1:

```
//emit everypage we get. The reducer will have just one copy of each page as it will reduce by key
map(.. , (pageId,adjacencyList)){

        emit(pageId,null)

        for(every page: adjacencyList){
                emit(page,null)
        }
}

// calculate the number of partitions and the totalnumber of records per partitions
// Set this number to counters
val=0;
reduce(pageId,(...)){
        val++;
}
```

```
cleanup(){
        incrementCounter(NoOfPartitions);
        setCounter(partitionId,val);
}

//In the driver class I use this counters to set offsets
setCounter(offset0,0)
for(i=1...getValueForCounter(NoOfPartitions)){
        sum+=getValueForCounter(i);
        setCounter(offseti,sum);
}
```

Job 2:

```
//The mapper is same as the previous job.
//emit everypage we get. The reducer will have just one copy of each page as it will
reduce by key
map(.. , (pageId,adjacencyList)){

        emit(pageId,null)

        for(every page: adjacencyList){
                emit(page,null)
        }
}

// assign index to each page and store it in a page.
val=0;

setup(){
        Id= getPartitionId();
        offset= getValueForCounter(offsetId);
}

reduce(pageId,(...)){
        emit(pageId,val+offset);
        val++;
}
```

Matrices(M,D,R[0]) Creation(1 MapReduce job):

```
//emit as it is.
map(.. , (pageId,adjacencyList)){

        emit(pageId,adjacencyList)
```

```
}

// in the reduce we create the actual matrices
setup(){
        // get the index created in previous step in a HashMap;
        HashMap<String,Long> pageToRow;
        // Created another data structure to keep track of the dead nodes
        HashSet<Long> remaining;
        j=0;
        for each entry in the index{
                pageToRow.put(pageId,index);
                remaining.put(index);
                emit(j++,0,1/totalnodes,R)//I emit his to a diff folder iter[0]
        }
}
reduce(pageId,(adjacencyList)){
        remaining.remove(pageId)//remove the non dead nodes
        j= pageToRow.get(pageId);
        d=1/adjacencyList.size();
        for each link in adjacencyList{
                i=pageToRow.get(link);
                emit(i,j,d,M);//creation of M matrix
        }
        if(d==0){
                emit(0,j,1/totalNodes,D);//creation of D matrix
        }
}

cleanup(){
        // we treat dead nodes as dangling nodes
        for each r in remaining{
                j=pageToRow.get(r);
                emit(0,j,1/totalNodes,D);
        }
}
```

```
//I do DR first and then use this value after doing MR;
//so both the DR and MR used the same logic for matrix multiplication;
//these two steps are done 10 times.
```

Calculating DR (2 Map reduce jobs):

Job 1:

```
map(.. , (row,col,val,MatrixID)){
        //Partition D into columns
        if(matrixID=='D')
                emit(col,(matrixID,row,val))

        //Partition R into rows
        if(matrixID=='R')
                emit(row,(matrixID,col,val))
}

//reduce recieves all the common columns from D and equivalent row from R. we
just multiply the val for different D and R.
reduce(commonDcol_Rrow,[(matrixID,index,val)...]){
        for each (matrixID,index,val){
                if(matrixID=='D') D_List.add(index,val)
                else if(matrixID=='R') R_List.add(index,val)
        }

        for each d in D_List
                for each r in R_List
                        emit((d.index,r.index),d.val*r.val)
}
```

Job 2:

```
//emit as it is
map((i,j),mul){
        emit((i,j),mul)
}

// we add all the values together to get the final result from matrix multiplication
//since D is 1XV and R is VX1 we get 1X1 value which we could assign to counters.
sum=0;
reduce((i,j),[mul1,mul2,...]){
        for each mul in [mul1 ,mul2,...]{
                sum+=mul;
        }
        setCounter(danglingPR,sum)
}
```

Calculating MR(2 Map reduce jobs):

Job 1:
Same as DR just change D to M;

Job 2:

```
//emit as it is
map((i,j),mul){
        emit((i,j),mul)
}

// we add all the values together to get the final result from matrix multiplication
sum=0;
reduce((i,j),[mul1,mul2,...]){
        for each mul in [mul1 ,mul2,...]{
                sum+=mul;
        }
        dpr= getValueofCounter(danglingPR)
        sum+=dpr;
        sum*0.85+ 0.15/totalNoofNodes;
        emit(i,j,sum,R);//create new R
}
```

Row By Column:

For this version everything elese is same just the matrix multiplication changes.
Now we dont need 2 jobs to perform matrix multiplication.
As its row by column we send one row for first matrix and complete R as it has only one column;
The overhead here is getting complete R in every reducer.

Calculating DR (1 Map reduce jobs):

```
//We dont need to emit R as we will retrieve complete R in the reducer;
map(.. , (row,col,val,MatrixID)){
        //Partition D into rows
        if(matrixID=='D')
                emit(row,(matrixID,col,val))
}

//reduce recieves all the common rows from D and equivalent column from R. we just multiply the val for different D and R and add.
sum=o;
setup(){
        // get the index created in previous step in a HashMap;
        HashMap<String,Long> R;
        j=0;
        for each entry in the R[i-1]{
                R.put(index,rank);
        }
}

reduce(commonDcol_Rrow,[(matrixID,index,val)...]){
        for each (matrixID,index,val){
                sum+=val*R.get(index)
        }
        setCounter(danglingPR,sum)
}
```

Calculating MR (1 Map reduce jobs):

```
//We dont need to emit R as we will retrieve complete R in the reducer;
map(.. , (row,col,val,MatrixID)){
        //Partition D into rows
        if(matrixID=='D')
                emit(row,(matrixID,col,val))
}

//reduce recieves all the common rows from D and equivalent column from R. we
just multiply the val for different D and R and add.
sum=o;
setup(){
        // get the index created in previous step in a HashMap;
        HashMap<String,Long> R;
        j=0;
        for each entry in the R[i-1]{
                R.put(index,rank);
        }
}

reduce(commonDcol_Rrow,[(matrixID,index,val)...]){
        for each (matrixID,index,val){
                sum+=val*R.get(index)
        }
        dpr= getValueofCounter(danglingPR)
        sum+=dpr;
        sum*0.85+ 0.15/totalNoofNodes;
        emit(i,j,sum,R);//create new R
}
```

Top-K records (common for both the versions) (from modules):

```
class Mapper{
        localtopK;

        setup(){
                inititalize localtopk;
        }

        map(... , x){
                if(x is in localtopk)
                        //doing this deletes the last entry in localtopk
                        // localtopk just keeps the top K values
                        localtopk.add(x);
        }

        cleanup(){
                for each x in localtopk
                        emit(dummy,x);
        }
}


reduce(dummy , [x1,x2,...]){

inititalize globaltopk;

for each x in [x1,x2,..]{
        if(x is in globaltopk)
                        //doing this deletes the last entry in globaltopk
                        // globaltopk just keeps the top K values
                        globaltopk.add(x);
}

for each x in globaltopk
                        emit(NULL,x);
}
```

Briefly explain how each matrix and vector is stored for versions A and B:

Creation of matrix remains same for both the versions. Only the partitioning differs. So for both the versions I saved the matrix M and D is sparse matrix form. As for M lot of the cell have values 0 (as I am creating a different matrix for D), I just sav it in the form (row,col,value). Same is the case for D as well as the number of dangling node is much less when compared to the total number of nodes, it makes sense to store the values in sparse matrix form.
For R vector I save all the data for each row in the file. Since value of R cannot be 0 for any page I keep all the values and store it in a file.
Also I save the matrix M and D at common location as we use it in every iteration.

Discuss how you handle dangling nodes.

While trying around few things I noticed that creating M' takes way more time as for example we have 20,000 pages and 5,000 are dangling then we have to change values of 20,000*5,000 cells which is quiet time consuming. So I followed one method given in assignment and did DR+MR. This introduces 1 mapreduce job per iteration but is way more faster than emitting 20,000*5,000 values.
So I created a D matrix which is 1XV size where V is the total number of pages. So DR gives me 1X1 matrix(1XV *VX1) which is a single value and I can just add this with every value in MR.

Performance Comparison:

** All times are approx. and calculated from the syslog

I have kept the parser in my program. So it first parses the bz2 files then uses the adjacency list to create index and matrices.

6 m4.large machines (1 master and 5 workers):

Version ColByRow
Creation of Matrix: 8 mins
Iterations: 1 hour 31 mins
Top-k: 1 mins

Version RowByCol
Creation of Matrix: 7 mins
Iterations: 1 hour 5 mins
Top-k: 1 mins

11 m4.large machines (1 master and 10 workers):
Version ColByRow
Creation of Matrix: 6 mins
Iterations: 58 mins
Top-k: less than a minute

Version RowByCol
Creation of Matrix: 6 mins
Iterations: 48 mins
Top-k: less than a minute

The iterations take a lot of time in matrix multiplication. This is because matrix-multiplication involves more map reduce jobs than normal adjacency list based approach. Also for the colByRow approach there are even more number of jobs as each matrix multiplication takes 2 MR jobs and we are doing 2 matrix multiplications. So there are total 4 MR jobs executed for 1 iteration. For the RowbyCol 2 MR jobs are executed per iteration.
I totally expected this version of pagerank to take more time as it has multiple overheads involved. So for RowbyCol we need complete R vector in each iteration so accessing this is very time consuming. The adjacencylist based approach was quiet straight forward and simple and had minimum number of MR jobs.

HW03 adjacency-list based Hadoop implementation Performance:

6 m4.large machines (1 master and 5 workers):

Pre-processing: 50 mins
Ten iterations of pagerank: 42 mins
Top 100: 1 mins 30 secs

11 m4.large machines (1 master and 10 workers):

Pre-processing: 23 mins
Ten iterations of pagerank: 31 mins
Top 100: 1 mins

Top 100 MapReduce Pagerank:

Simple dataset:

United_States_09d4 0.006285955388829367
Wikimedia_Commons_7b57 0.004796671097253103
Country 0.0039224382405842325
England 0.002682622147721756
Europe 0.002622080709665762
United_Kingdom_5ad7 0.002616230430514357
Germany 0.0026042705532042434
Water 0.002584497221249347
France 0.0025363239447877546
Animal 0.002454369171240718
Earth 0.002428715453830068
City 0.0024062237499459503
Week 0.0024124547792674387
Asia 0.0019321340468770312
Sunday 0.0018720896700419546
Wiktionary 0.0018618723396648228
Monday 0.0018448788268678464
Money 0.0018406440624429935
Wednesday 0.001826602516582644
Plant 0.001809025216999627
Friday 0.0017823440728751939
Saturday 0.001762279639460181
Computer 0.001758630868721822
English_language 0.0017518903988358702
Thursday 0.0017396324597718916

Tuesday	0.001727204870177261
Italy	0.0017250277812977186
Government	0.001715027792620142
India	0.0017085327259061616
Number	0.0015876825625422133
Spain	0.0015773512893744938
Japan	0.0015201625494897844
Canada	0.0015058862161908004
Day	0.0014737586002971927
People	0.0014499916501110532
Human	0.0014209364839374468
Wikimedia_Foundation_83d9	0.0013848855612459145
China	0.0013704986609578624
Australia	0.0013691094790021319
Energy	0.0013331138754942493
Food	0.0013152408888342196
Sun	0.0012943141801420885
Science	0.001291548356814037
Mathematics	0.001275973736574264
index	0.0012655431922724196
Television	0.001226019285363822
Capital_(city)	0.0012027915724235063
Russia	0.0011872438148647425
State	0.0011745879763614554
Music	0.0011581738842250109
Year	0.0011368288912242468
Greece	0.001116420000037706
Language	0.00111564470768701
Scotland	0.0011077537288302843
Metal	0.0010814015925980616
Wikipedia	0.0010784440935179586
2004	0.0010701339291897224
Greek_language	0.001065735831284468
Planet	0.0010336072046317015
Sound	0.001029476006764391
Religion	0.0010255628356702832
London	0.0010208914344294168
Africa	0.0010113053541823608
20th_century	9.627773021943762E-4
Poland	9.618007882151685E-4
Law	9.521206787860108E-4
Geography	9.508425375496019E-4
19th_century	9.416561257934388E-4
Liquid	9.36643169409884E-4
World	9.296343739733175E-4
Society	9.134387417304141E-4
Scientist	9.125373635365744E-4

History 8.824703746742198E-4
Latin 8.821346871472971E-4
Atom 8.764513654297136E-4
Sweden 8.750595406681096E-4
War 8.732096990203899E-4
Light 8.644901826325814E-4
Netherlands 8.625143526133756E-4
Culture 8.549074933869976E-4
Building 8.416788477249255E-4
God 8.291670192723625E-4
Turkey 8.274081872553086E-4
Plural 8.189313777532135E-4
Information 8.183080706775151E-4
Inhabitant 8.089044970843776E-4
Centuries 8.071990756197943E-4
Portugal 7.988103093360046E-4
Chemical_element 7.911671277015457E-4
Capital_city 7.909478138096838E-4
Denmark 7.847428282777628E-4
Austria 7.777275664651701E-4
Cyprus 7.64712908875249E-4
University 7.585918228417333E-4
Ocean 7.580595576935304E-4
Book 7.553724241965877E-4
Species 7.55349236540735E-4
North_America_e7c4 7.553347614103821E-4
Disease 7.525133180679138E-4
Biology 7.471092043182994E-4

Full dataset:

United_States_09d4 0.0028721740762318137
2006 0.0025683611349989898
United_Kingdom_5ad7 0.0013657282202906161
2005 0.0011843661457172374
Biography 9.423390704185306E-4
Canada 8.930099291571345E-4
England 8.873191305830984E-4
France 8.778280739141111E-4
2004 8.249109077340098E-4
Germany 7.542555937001934E-4
Australia 7.305609453282984E-4
Geographic_coordinate_system 7.157694816622438E-4
2003 6.6439379320624E-4
India 6.44024463050385E-4

Japan 6.383208619630372E-4
Italy 5.357769048278127E-4
2001 5.334286958661272E-4
2002 5.273763528337615E-4
Internet_Movie_Database_7ea7 5.222377544044592E-4
Europe 5.07942696507908E-4
2000 4.992029957034937E-4
World_War_II_d045 4.812308205947173E-4
London 4.645028941413261E-4
Population_density 4.474926092254699E-4
Record_label 4.422391591128756E-4
1999 4.4122837006049895E-4
English_language 4.381064231169587E-4
Spain 4.3796005483551865E-4
Russia 4.1325156548653765E-4
Race_(United_States_Census)_a07d 4.103324004665E-4
Wiktionary 4.0379380500119515E-4
Wikimedia_Commons_7b57 3.8442270466348906E-4
1998 3.8140007768704625E-4
Music_genre 3.724594621671542E-4
1997 3.638236853687485E-4
Scotland 3.580582171498726E-4
New_York_City_1428 3.576359999439566E-4
Football_(soccer) 3.490880289331022E-4
1996 3.4132933056624503E-4
Sweden 3.365027297845802E-4
Television 3.358401454865275E-4
Square_mile 3.2403425949388624E-4
Census 3.2335869475428035E-4
1995 3.215478466214148E-4
California 3.1882470661153415E-4
China 3.1473064420436286E-4
Netherlands 3.0995437235862395E-4
New_Zealand_2311 3.089401554313044E-4
1994 3.0691674824203947E-4
1991 2.9283187340099797E-4
1993 2.9021699648110096E-4
1990 2.8845250795351407E-4
New_York_3da4 2.8660178435756896E-4
Public_domain 2.86441633274951E-4
1992 2.781496424426107E-4
United_States_Census_Bureau_2c852.760919763427364E-4
Film 2.7575495733158105E-4
Actor 2.737849673431083E-4
Scientific_classification 2.733013194458721E-4
Norway 2.7118442987387174E-4
Ireland 2.6952673703111097E-4

Population	2.6782265665151135E-4
Poland	2.6733092685873E-4
1989	2.608200808956598E-4
1980	2.5478254062360605E-4
January_1	2.5449701523501373E-4
Marriage	2.531363095868134E-4
Brazil	2.5274105563673194E-4
Mexico	2.5107786773428985E-4
Latin	2.5081524535322324E-4
Politician	2.492144219197558E-4
1986	2.478421469322863E-4
1985	2.4176677201302138E-4
1979	2.4149827286779052E-4
1982	2.408730347972165E-4
1981	2.4065534009302136E-4
French_language	2.4061888960462798E-4
Per_capita_income	2.3879989157835207E-4
1974	2.3840654065887832E-4
Album	2.3746239683104223E-4
Switzerland	2.364760799563359E-4
1984	2.361937227162568E-4
1987	2.3598974047920988E-4
South_Africa_1287	2.3595859775454766E-4
1983	2.359115444749447E-4
Record_producer	2.338866084390173E-4
1970	2.3213474352844262E-4
1988	2.3058643259443468E-4
1976	2.294490553754084E-4
Km²	2.27065912238393E-4
1975	2.2679581218519777E-4
Paris	2.2383111838854812E-4
1969	2.2364602286097446E-4
Greece	2.2331778654286876E-4
1945	2.2247191445713436E-4
1972	2.2221152520119042E-4
Personal_name	2.2122812547045137E-4
1977	2.2040222423978342E-4
Soviet_Union_ad1f	2.195746250888832E-4
1978	2.193481799230175E-4

The results are different in value as well as in order. The top pages are some what same in order but as we go down they change. The main reason I feel is dangling node. In HW03 dangling nodes were handled in different way. I calculated the danglingSumPR and used it in next iteration. Here we keep on adding the danglingSumPR in that iteration giving us more precise and correct answers. Also in this version I have handled the dead nodes so that affect the pagerank values.