INFO 7250 Final Project

Page Ranking Algorithm using Hadoop MapReduce

-BY:

SAMRUDDHI PASARI

(001306800)

Contents

1. Abstract

Searching the most relevant information in hundreds and billions of web pages can be a task. That is where different Search Algorithms plays a major role. In this project I have implemented one such algorithm namely PageRank (PR), an algorithm used by **Google Search** to rank web pages in their search engine results.

2. Problem Statement

Given a dataset with nodes and its outgoing links, implement a Page Ranking Algorithm to determine the importance of each web pages by calculating ranks and providing the top 10 and most relevant ones among them.

3. Introduction

The importance of webpage depends upon the readers interests, intentions, knowledge and attitudes. The amount of web pages available for each search result makes it difficult for a random surfer to reach to the most relevant information needed.

PageRank relies on the uniquely democratic nature of the web by using its vast link structure as an indicator of an individual page's value. If page A is linking to page B , it is an outgoing link from that page and all such outgoing links from that page receive the vote of page A, i.e. the page rank values of that page is equally distributed among all the pages. Along with that page A will receive summation of ranks from all the incoming links.

Page Ranking Algorithm is iterative graph algorithm and where convergence of rank values can be achieved by repeating the page ranking calculation process.

4. Our Approach

Following are the approaches to implement Page Ranking Algorithm. There are Fixed Iteration approach and two alternative convergence criteria:

a. Iterating page rank calculation until page rank value convergence
b. Iterating page rank calculation until page rank ranking remains unchanged

In this project I have implemented Fixed Iteration approach. It will include two MapReduce jobs: first to create adjacency list, which is a list of all the outgoing edges from a given node, for each node with nodes as key and adjacency list as value of the Map Reduce job. The second job will calculate and assign PageRank's to each webpage which will keep updating over several iterations until the value for each webpage converges. It is based on a mathematical formula which assign weights to each webpage based on number of incoming and outgoing edges. The additional challenge is handling PageRank's of dangling nodes for which various approaches can be implemented and the results can be analyzed to find the best approach. The result will be a sorted order of all the pages with respect to their page ranking. Using this result, we can analyze top N webpages in the Google dataset.

5. Data Source

The dataset includes Web graph from the Google programming contest, 2002. It contains 2 columns nodes and directed edges. Nodes represent web pages and directed edges represent hyperlinks between them.

Dataset source:

https://snap.stanford.edu/data/web-Google.html

Dataset Statistics:

Column	**Number of entries**
Nodes	875713
Edges	5105039

6. Implementation

There are 4 stages of implementing Page Rank Algorithm:

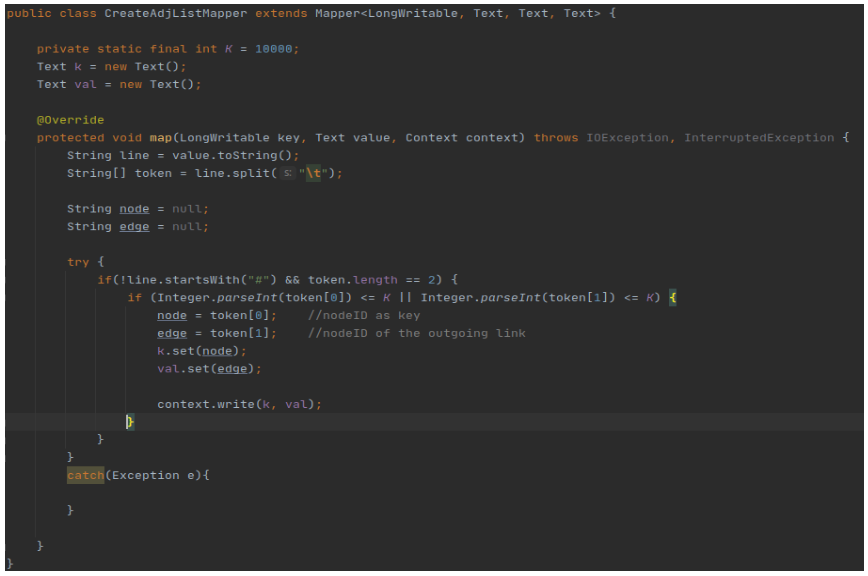
a. Creating Adjacency List from a given dataset
b. Calculating Page Rank for all the nodes
c. Handling Dangling Nodes
d. Finding Top 10 Web pages

a. Creating Adjacency List:

The dataset consists of two columns "From" and "To", where "From" is the nodeID and "To" is the outgoing web page from that node. An adjacency list is the list of all the outgoing links from a given node.

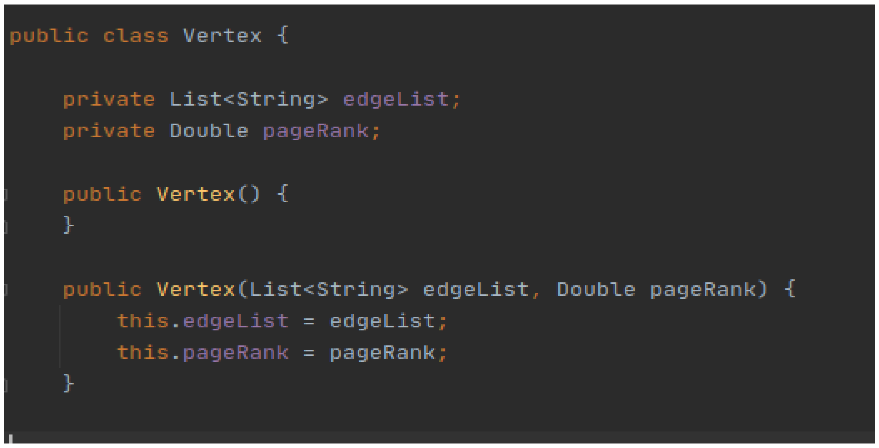
A MapReduce job is used to create an adjacency list where the key is nodeID and value is the adjacency list. In Mapper class we emit nodeID as key and the nodeID of outgoing web page for the given key. A vertex object is defined per node, which consist of corresponding adjacency list and page rank value. In Reducer class we create a vertex and initialize page rank for the given nodeID(key).

Mapper Class



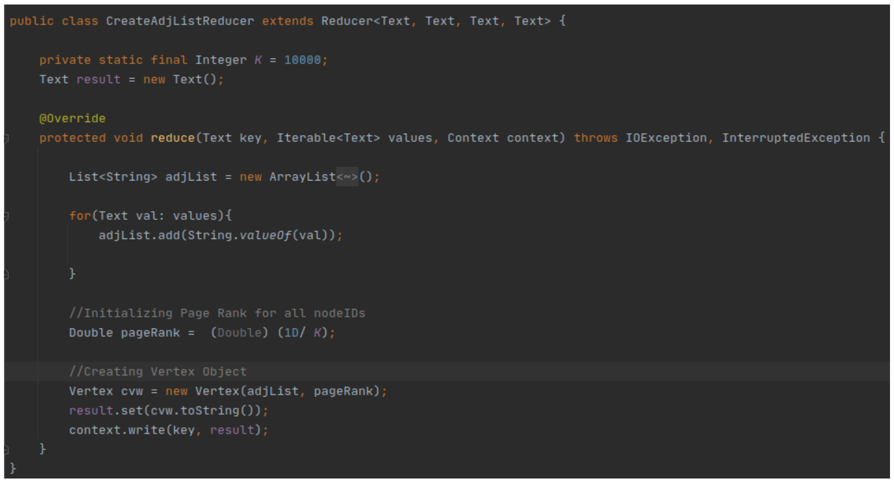
```
public class CreateAdjListMapper extends Mapper<LongWritable, Text, Text, Text> {

    private static final int K = 10000;
    Text k = new Text();
    Text val = new Text();

    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        String line = value.toString();
        String[] token = line.split( s: "\t");

        String node = null;
        String edge = null;

        try {
            if(!line.startsWith("#") && token.length == 2) {
                if (Integer.parseInt(token[0]) <= K || Integer.parseInt(token[1]) <= K) {
                    node = token[0];    //nodeID as key
                    edge = token[1];    //nodeID of the outgoing link
                    k.set(node);
                    val.set(edge);

                    context.write(k, val);
                }
            }
        }
        catch(Exception e){

        }

    }
}
```

Vertex Object



```
public class Vertex {

    private List<String> edgeList;
    private Double pageRank;

    public Vertex() {
    }

    public Vertex(List<String> edgeList, Double pageRank) {
        this.edgeList = edgeList;
        this.pageRank = pageRank;
    }

```

Reducer Class



```
public class CreateAdjListReducer extends Reducer<Text, Text, Text, Text> {

    private static final Integer K = 10000;
    Text result = new Text();

    @Override
    protected void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {

        List<String> adjList = new ArrayList<~>();

        for(Text val: values){
            adjList.add(String.valueOf(val));

        }

        //Initializing Page Rank for all nodeIDs
        Double pageRank =  (Double) (1D/ K);

        //Creating Vertex Object
        Vertex cvw = new Vertex(adjList, pageRank);
        result.set(cvw.toString());
        context.write(key, result);
    }
}
```

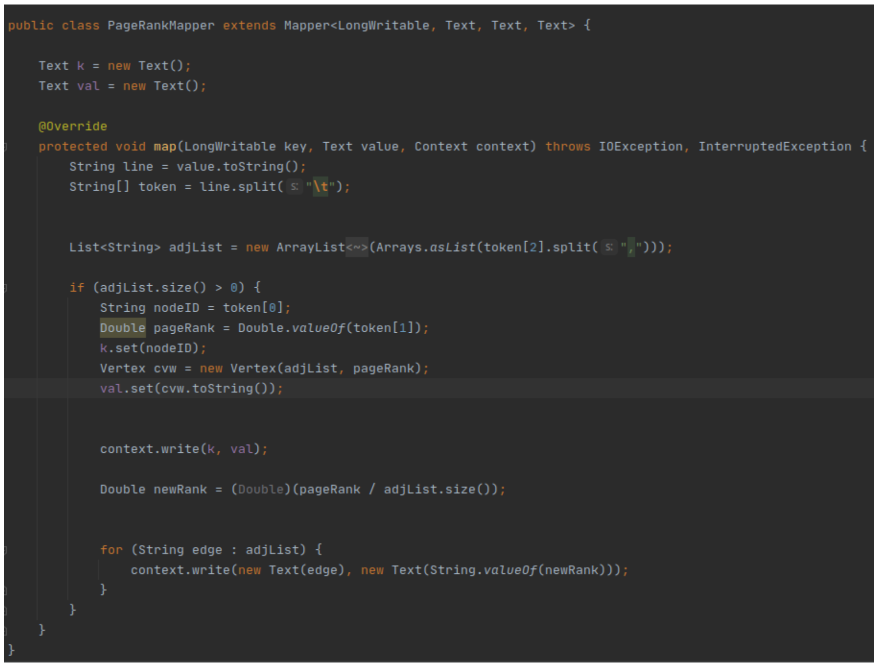
b. *Calculating Page Rank:*

After emitting nodeID as key and Vertex object as value, we pass this Stage 1 output to next MapReduce job to calculate new page rank value for all the outgoing links in the adjacency list.

In Mapper Class, we calculate new page rank value for all nodes in adjacency list per node, by equally dividing the page rank value of key nodeID among all nodes in adjacency list.

p = N.pageRank / N.adjacencyList.size()

Mapper Class



```
public class PageRankMapper extends Mapper<LongWritable, Text, Text, Text> {

    Text k = new Text();
    Text val = new Text();

    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        String line = value.toString();
        String[] token = line.split("\t");


        List<String> adjList = new ArrayList<~>(Arrays.asList(token[2].split(",")));

        if (adjList.size() > 0) {
            String nodeID = token[0];
            Double pageRank = Double.valueOf(token[1]);
            k.set(nodeID);
            Vertex cvw = new Vertex(adjList, pageRank);
            val.set(cvw.toString());


            context.write(k, val);

            Double newRank = (Double)(pageRank / adjList.size());


            for (String edge : adjList) {
                context.write(new Text(edge), new Text(String.valueOf(newRank)));
            }
        }
    }
}
```

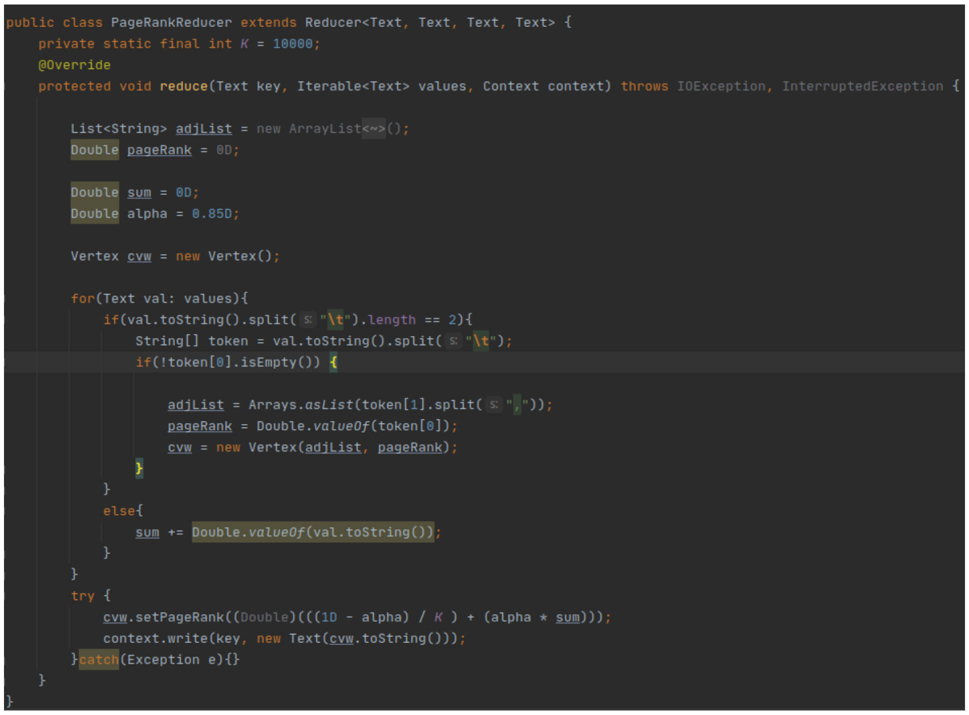
In Reducer Class, we calculate the actual page rank value for all the nodes using the following formula:

$$P(n) = (1-\alpha)\frac{1}{|V|} + \alpha \sum_{m \in L(n)} \frac{P(m)}{C(m)}$$

The variables in the formula have following meaning:

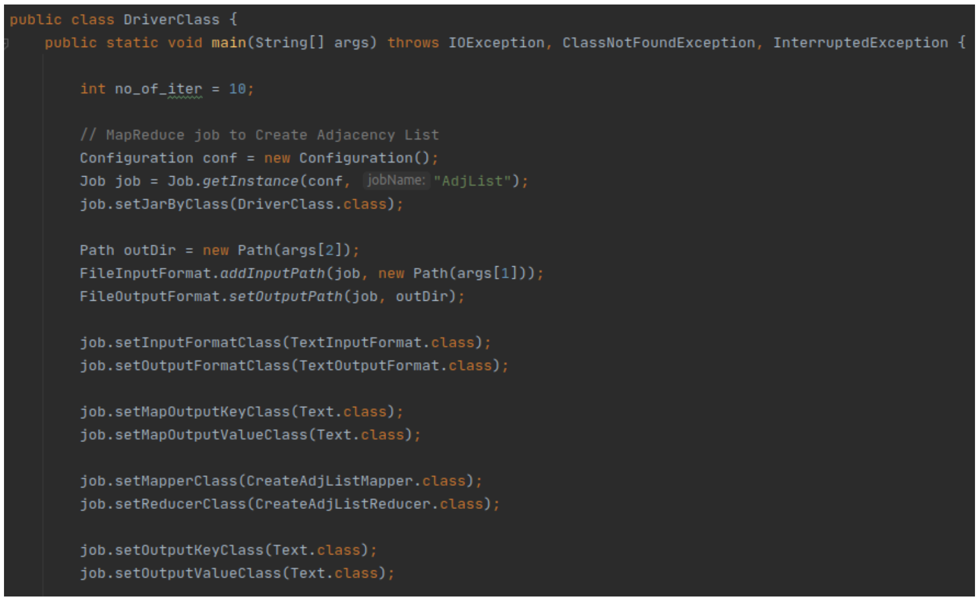
- |V| is the number of pages (vertices) in the input dataset Web Graph
- α is the probability of the surfer following a link
- 1- α is the probability of making a random jump
- L(n) is the set of all the pages in the graph linking to n
- P(m) is the page rank of another page m
- C(m) is the out-degree of page m, i.e., the number of links on that page

Reducer Class

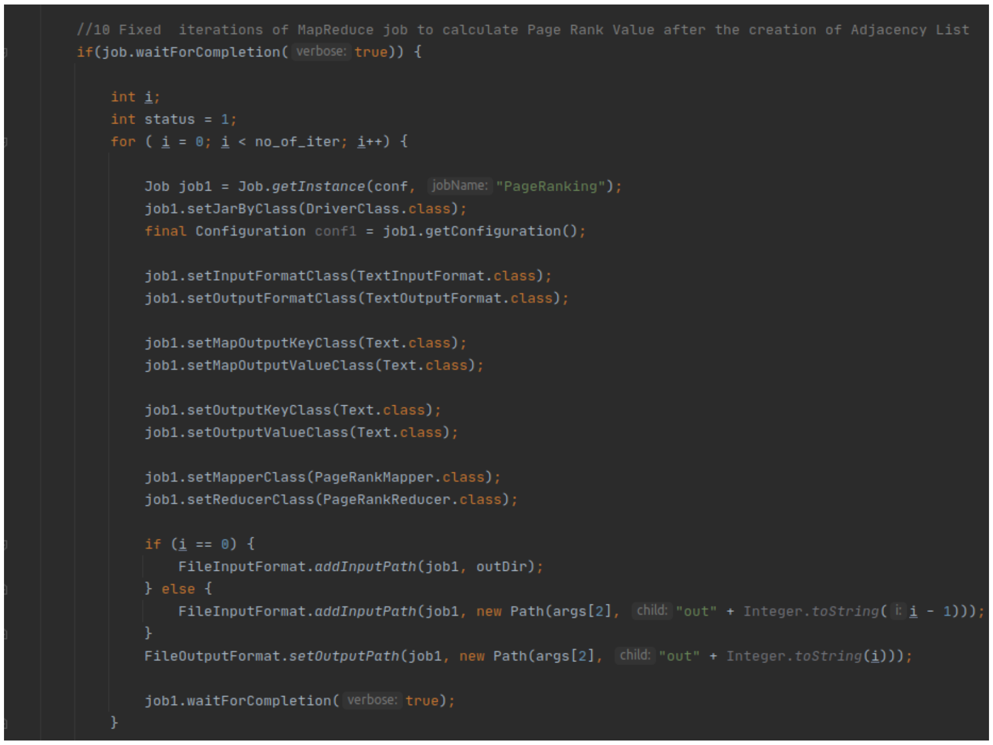


```java
public class PageRankReducer extends Reducer<Text, Text, Text, Text> {
    private static final int K = 10000;
    @Override
    protected void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {

        List<String> adjList = new ArrayList<~>();
        Double pageRank = 0D;

        Double sum = 0D;
        Double alpha = 0.85D;

        Vertex cvw = new Vertex();

        for(Text val: values){
            if(val.toString().split( s: "\t").length == 2){
                String[] token = val.toString().split( s: "\t");
                if(!token[0].isEmpty()) {

                    adjList = Arrays.asList(token[1].split( s: ","));
                    pageRank = Double.valueOf(token[0]);
                    cvw = new Vertex(adjList, pageRank);
                }
            }
            else{
                sum += Double.valueOf(val.toString());
            }
        }
        try {
            cvw.setPageRank((Double)(((1D - alpha) / K ) + (alpha * sum)));
            context.write(key, new Text(cvw.toString()));
        }catch(Exception e){}
    }
}
```

In Driver Class, the Page Rank MapReduce job is run for 10 fixed iteration for value convergence to attain more accuracy in Page Rank calculation.



```java
public class DriverClass {
    public static void main(String[] args) throws IOException, ClassNotFoundException, InterruptedException {

        int no_of_iter = 10;

        // MapReduce job to Create Adjacency List
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, jobName: "AdjList");
        job.setJarByClass(DriverClass.class);

        Path outDir = new Path(args[2]);
        FileInputFormat.addInputPath(job, new Path(args[1]));
        FileOutputFormat.setOutputPath(job, outDir);

        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);

        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);

        job.setMapperClass(CreateAdjListMapper.class);
        job.setReducerClass(CreateAdjListReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
```



```
//10 Fixed  iterations of MapReduce job to calculate Page Rank Value after the creation of Adjacency List
if(job.waitForCompletion( verbose: true)) {

    int i;
    int status = 1;
    for ( i = 0; i < no_of_iter; i++) {

        Job job1 = Job.getInstance(conf,  jobName: "PageRanking");
        job1.setJarByClass(DriverClass.class);
        final Configuration conf1 = job1.getConfiguration();

        job1.setInputFormatClass(TextInputFormat.class);
        job1.setOutputFormatClass(TextOutputFormat.class);

        job1.setMapOutputKeyClass(Text.class);
        job1.setMapOutputValueClass(Text.class);

        job1.setOutputKeyClass(Text.class);
        job1.setOutputValueClass(Text.class);

        job1.setMapperClass(PageRankMapper.class);
        job1.setReducerClass(PageRankReducer.class);

        if (i == 0) {
            FileInputFormat.addInputPath(job1, outDir);
        } else {
            FileInputFormat.addInputPath(job1, new Path(args[2],  child: "out" + Integer.toString( i: i - 1)));
        }
        FileOutputFormat.setOutputPath(job1, new Path(args[2],  child: "out" + Integer.toString(i)));

        job1.waitForCompletion( verbose: true);
    }
```

c. *Handling Dangling Nodes:*

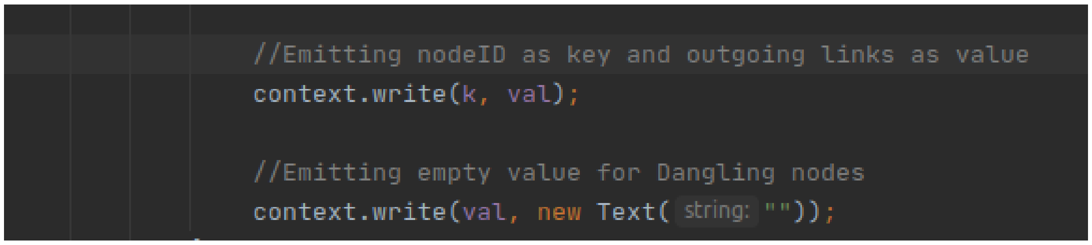
The above approach has one limitation. It does not consider the nodes with no outgoing links. Since there are no out going links from those nodes, their nodeID won't be included in "From" table, causing it to miss as a key in first Mapper job. This will cause the loss of probability mass calculated for all nodes. The loss will increase with each iteration ultimately resulting into a page rank value of 0 for all nodes.

This issue can be handled by accommodating the lost probability of mass by adding the page rank value of all the dangling nodes, i.e., δ and accounting that value in page rank calculation. The following modification in the page rank calculation is one of the approaches in handling the dangling nodes:

$$P(n) = (1-\alpha)\frac{1}{|V|} + \alpha\left(\frac{\delta}{|V|} + \sum_{m \in L(n)} \frac{P(m)}{C(m)}\right)$$

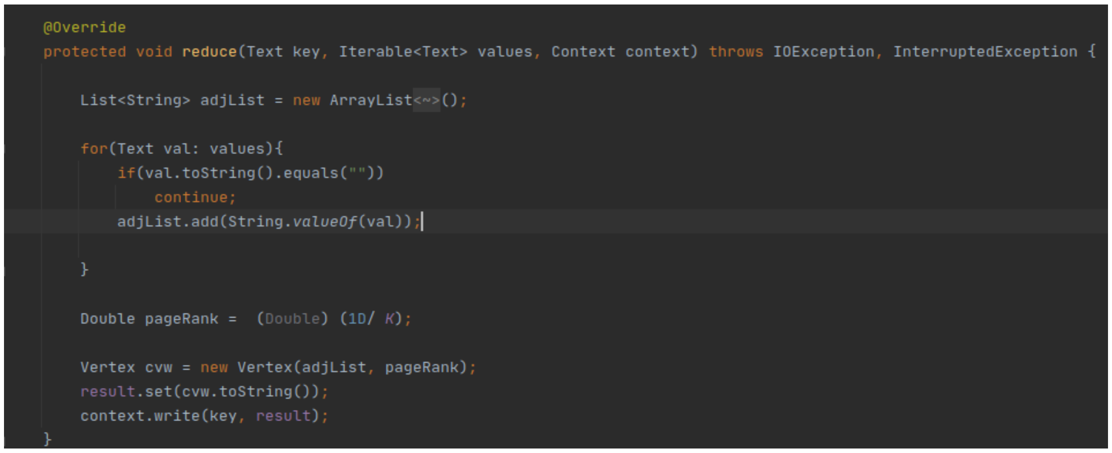
We emit empty string from Adjacency list Mapper to create an empty adjacency list in the Adjacency List Reducer.

CreateAdjListMapper class modification to handle Dangling nodes



```
    //Emitting nodeID as key and outgoing links as value
    context.write(k, val);

    //Emitting empty value for Dangling nodes
    context.write(val, new Text( string: ""));
```

CreateAdjListReducer Class modification to handle Dangling nodes

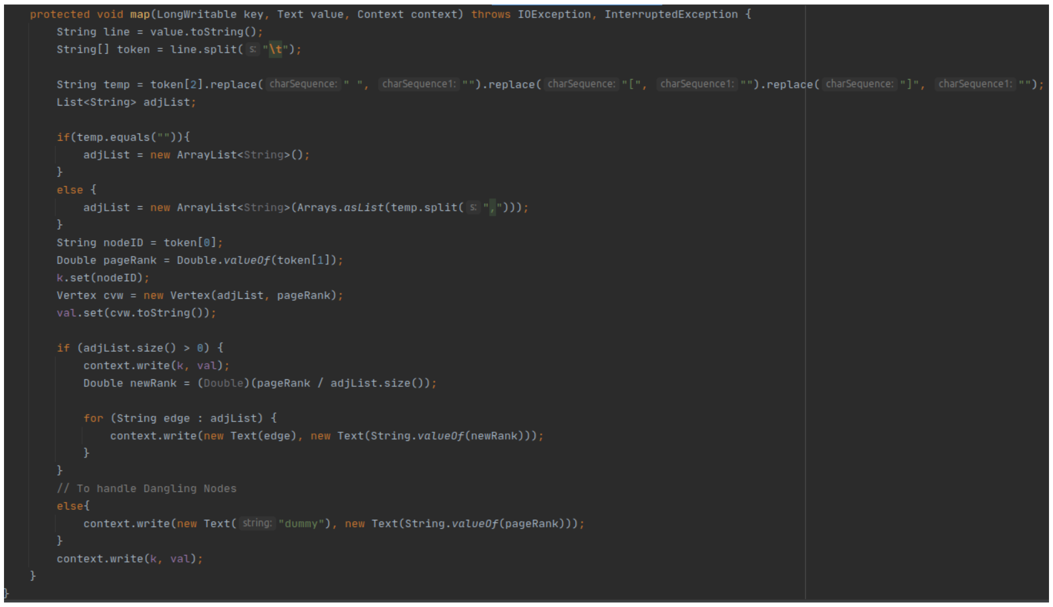


```
@Override
protected void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {

    List<String> adjList = new ArrayList<~>();

    for(Text val: values){
        if(val.toString().equals(""))
            continue;
        adjList.add(String.valueOf(val));

    }

    Double pageRank =  (Double) (1D/ K);

    Vertex cvw = new Vertex(adjList, pageRank);
    result.set(cvw.toString());
    context.write(key, result);
}
```

The output of above MapReduce phase is then passed to page rank calculation MapReduce Job.

In Stage 1 Mapper, along with emitting the vertex object for all nodes including the dangling nodes, another key-value pair is emitted where key is “dummy” and all the page rank values of dangling nodes as value.

PageRankMapper modification to handle Dangling nodes

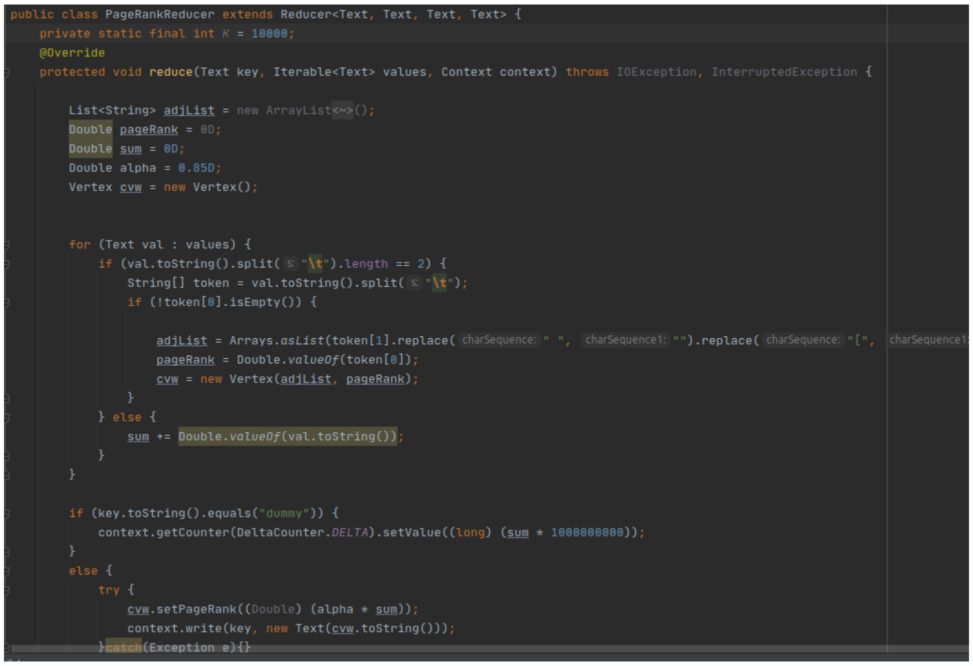


```java
    protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        String line = value.toString();
        String[] token = line.split( s: "\t");

        String temp = token[2].replace( charSequence: " ", charSequence1: "").replace( charSequence: "[", charSequence1: "").replace( charSequence: "]", charSequence1: "");
        List<String> adjList;

        if(temp.equals("")){
            adjList = new ArrayList<String>();
        }
        else {
            adjList = new ArrayList<String>(Arrays.asList(temp.split( s: ",")));
        }
        String nodeID = token[0];
        Double pageRank = Double.valueOf(token[1]);
        k.set(nodeID);
        Vertex cvw = new Vertex(adjList, pageRank);
        val.set(cvw.toString());

        if (adjList.size() > 0) {
            context.write(k, val);
            Double newRank = (Double)(pageRank / adjList.size());

            for (String edge : adjList) {
                context.write(new Text(edge), new Text(String.valueOf(newRank)));
            }
        }
        // To handle Dangling Nodes
        else{
            context.write(new Text( string: "dummy"), new Text(String.valueOf(pageRank)));
        }
        context.write(k, val);
    }
}
```

In Stage 1 Reducer, we have summed up the value of all page ranks as value of δ and from the "dummy" key and set the value of it as a Counter. A counter is an enum that can be passed as a parameter to Driver and can be extracted in chained MapReduce job. Since the approach is iterative computation, we must pass this calculated new delta each iteration and pass it to Stage 2 Mapper only MapReduce job to updates all Page Ranks using the new formula with δ.

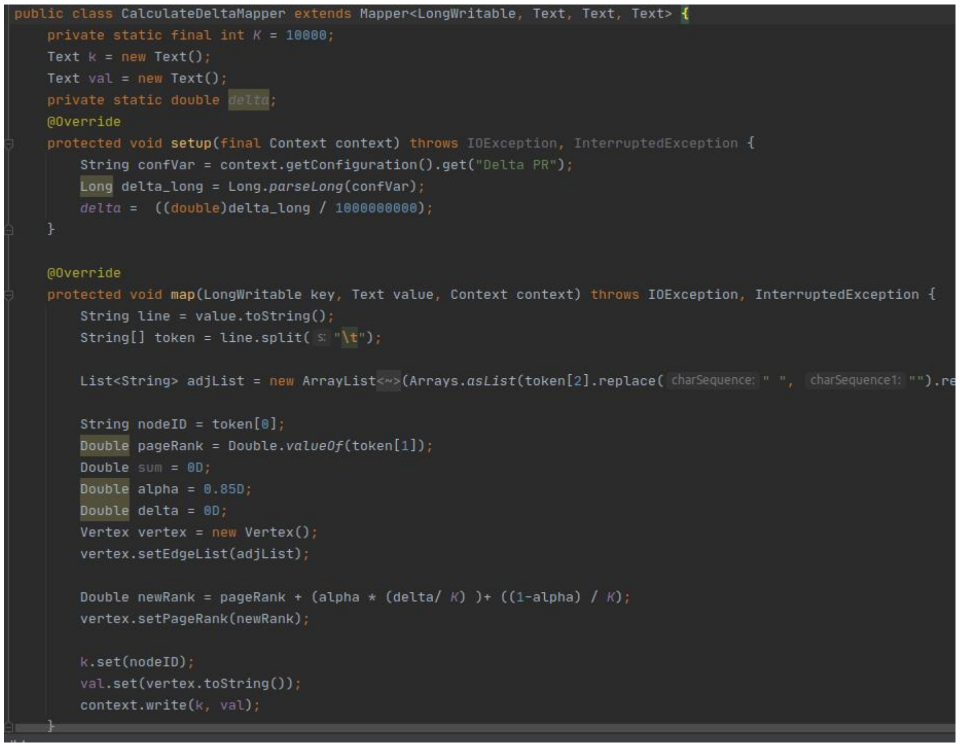
In Stage 2 Mapper only job, we extract Counter value of δ in setup method and first remaining of the page rank formula is calculated to update all the page ranks. This job is executed in loop after the Stage 1 MapReduce job.

PageRankReducer modification to handle Dangling nodes



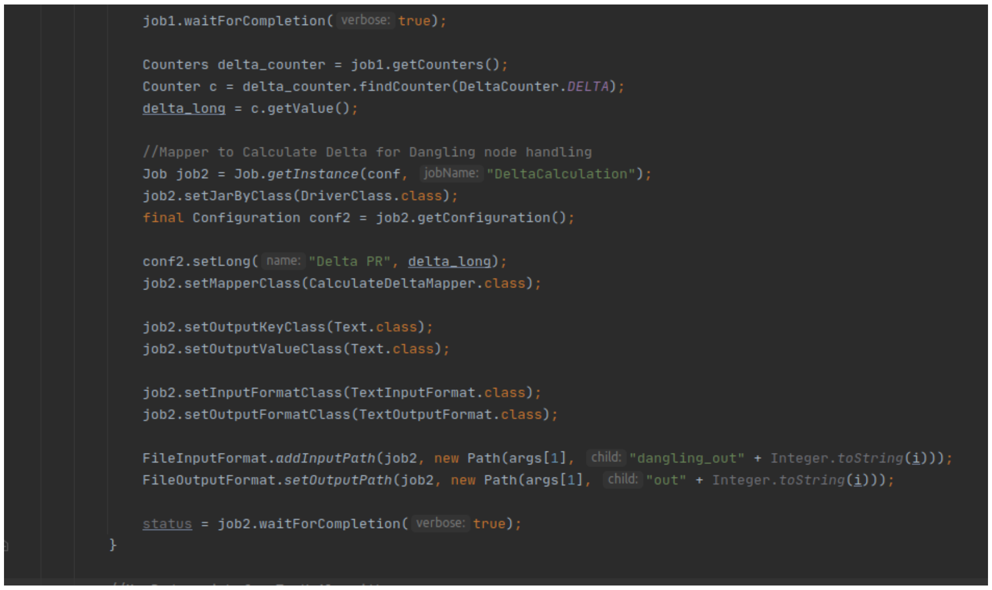
```java
public class PageRankReducer extends Reducer<Text, Text, Text, Text> {
    private static final int K = 10000;
    @Override
    protected void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {

        List<String> adjList = new ArrayList<~>();
        Double pageRank = 0D;
        Double sum = 0D;
        Double alpha = 0.85D;
        Vertex cvw = new Vertex();


        for (Text val : values) {
            if (val.toString().split( s: "\t").length == 2) {
                String[] token = val.toString().split( s: "\t");
                if (!token[0].isEmpty()) {

                    adjList = Arrays.asList(token[1].replace( charSequence: " ",  charSequence1: "").replace( charSequence: "[",  charSequence1:
                    pageRank = Double.valueOf(token[0]);
                    cvw = new Vertex(adjList, pageRank);
                }
            } else {
                sum += Double.valueOf(val.toString());
            }
        }

        if (key.toString().equals("dummy")) {
            context.getCounter(DeltaCounter.DELTA).setValue((long) (sum * 1000000000));
        }
        else {
            try {
                cvw.setPageRank((Double) (alpha * sum));
                context.write(key, new Text(cvw.toString()));
            }catch(Exception e){}
```

CalculateDeltaMapper to handle Dangling nodes



```java
public class CalculateDeltaMapper extends Mapper<LongWritable, Text, Text, Text> {
    private static final int K = 10000;
    Text k = new Text();
    Text val = new Text();
    private static double delta;
    @Override
    protected void setup(final Context context) throws IOException, InterruptedException {
        String confVar = context.getConfiguration().get("Delta PR");
        Long delta_long = Long.parseLong(confVar);
        delta =  ((double)delta_long / 1000000000);
    }

    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        String line = value.toString();
        String[] token = line.split( s: "\t");

        List<String> adjList = new ArrayList<~>(Arrays.asList(token[2].replace( charSequence: " ",  charSequence1: "").re

        String nodeID = token[0];
        Double pageRank = Double.valueOf(token[1]);
        Double sum = 0D;
        Double alpha = 0.85D;
        Double delta = 0D;
        Vertex vertex = new Vertex();
        vertex.setEdgeList(adjList);

        Double newRank = pageRank + (alpha * (delta/ K) )+ ((1-alpha) / K);
        vertex.setPageRank(newRank);

        k.set(nodeID);
        val.set(vertex.toString());
        context.write(k, val);
    }
```

DriverClass modification to set Counter for next MapReduce job

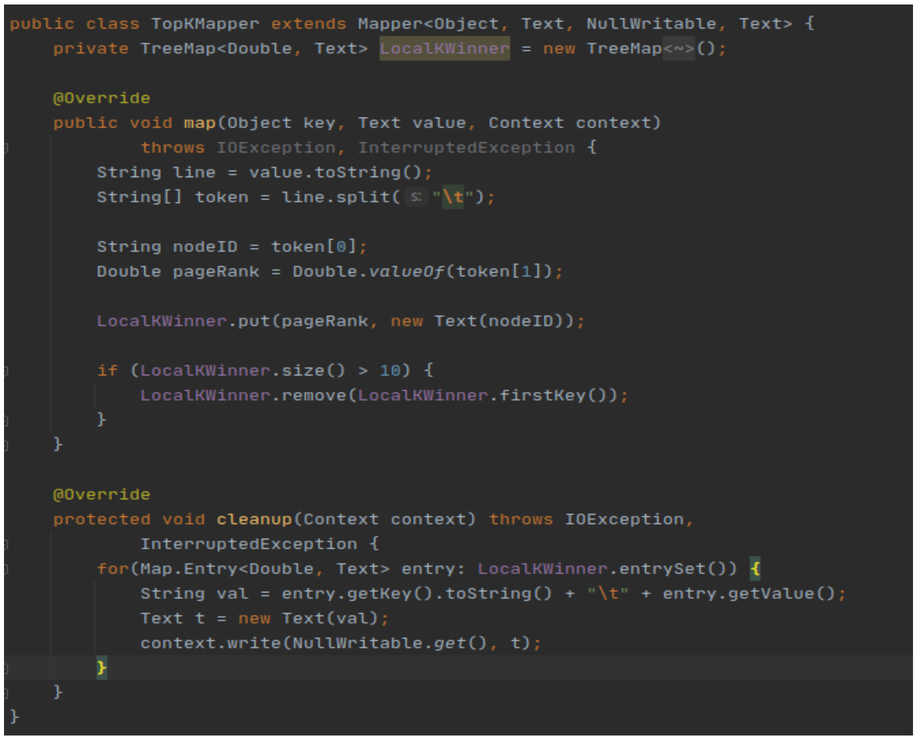


```java
            job1.waitForCompletion( verbose: true);

            Counters delta_counter = job1.getCounters();
            Counter c = delta_counter.findCounter(DeltaCounter.DELTA);
            delta_long = c.getValue();

            //Mapper to Calculate Delta for Dangling node handling
            Job job2 = Job.getInstance(conf,  jobName: "DeltaCalculation");
            job2.setJarByClass(DriverClass.class);
            final Configuration conf2 = job2.getConfiguration();

            conf2.setLong( name: "Delta PR", delta_long);
            job2.setMapperClass(CalculateDeltaMapper.class);

            job2.setOutputKeyClass(Text.class);
            job2.setOutputValueClass(Text.class);

            job2.setInputFormatClass(TextInputFormat.class);
            job2.setOutputFormatClass(TextOutputFormat.class);

            FileInputFormat.addInputPath(job2, new Path(args[1],  child: "dangling_out" + Integer.toString(i)));
            FileOutputFormat.setOutputPath(job2, new Path(args[1],  child: "out" + Integer.toString(i)));

            status = job2.waitForCompletion( verbose: true);
        }
```

d. *Finding Top 10 Web Pages:*

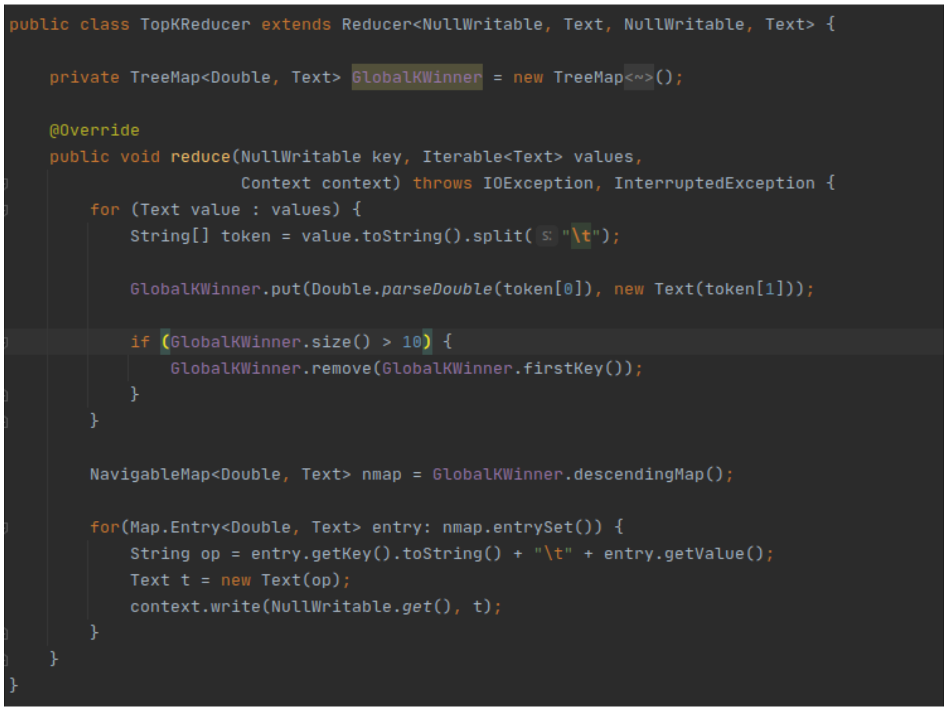
After computing Page Rank values for all the nodeID for 10 iterations, we find the top 10 web pages using Top-K algorithm.

TopKMapper Class



```java
public class TopKMapper extends Mapper<Object, Text, NullWritable, Text> {
    private TreeMap<Double, Text> LocalKWinner = new TreeMap<~>();

    @Override
    public void map(Object key, Text value, Context context)
            throws IOException, InterruptedException {
        String line = value.toString();
        String[] token = line.split( s: "\t");

        String nodeID = token[0];
        Double pageRank = Double.valueOf(token[1]);

        LocalKWinner.put(pageRank, new Text(nodeID));

        if (LocalKWinner.size() > 10) {
            LocalKWinner.remove(LocalKWinner.firstKey());
        }
    }

    @Override
    protected void cleanup(Context context) throws IOException,
            InterruptedException {
        for(Map.Entry<Double, Text> entry: LocalKWinner.entrySet()) {
            String val = entry.getKey().toString() + "\t" + entry.getValue();
            Text t = new Text(val);
            context.write(NullWritable.get(), t);
        }
    }
}
```

TopKReducer Class



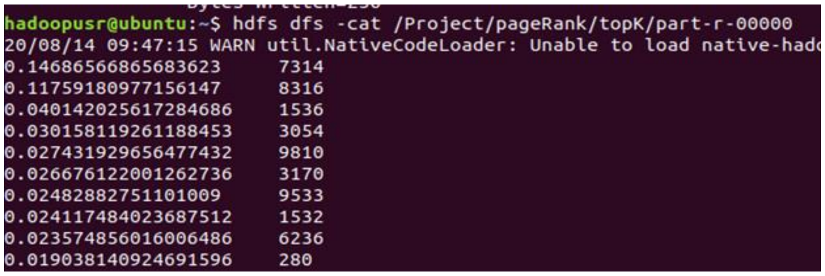
```
public class TopKReducer extends Reducer<NullWritable, Text, NullWritable, Text> {

    private TreeMap<Double, Text> GlobalKWinner = new TreeMap<~>();

    @Override
    public void reduce(NullWritable key, Iterable<Text> values,
                       Context context) throws IOException, InterruptedException {
        for (Text value : values) {
            String[] token = value.toString().split( "\t");

            GlobalKWinner.put(Double.parseDouble(token[0]), new Text(token[1]));

            if (GlobalKWinner.size() > 10) {
                GlobalKWinner.remove(GlobalKWinner.firstKey());
            }
        }

        NavigableMap<Double, Text> nmap = GlobalKWinner.descendingMap();

        for(Map.Entry<Double, Text> entry: nmap.entrySet()) {
            String op = entry.getKey().toString() + "\t" + entry.getValue();
            Text t = new Text(op);
            context.write(NullWritable.get(), t);
        }
    }
}
```

7. Result and Analysis

It can be observed from the above project that the accuracy in page rank calculation we achieved in 2nd approach, the implementation with handling of dangling node is greater than the accuracy in 1st approach, the implementation without handling of dangling node.

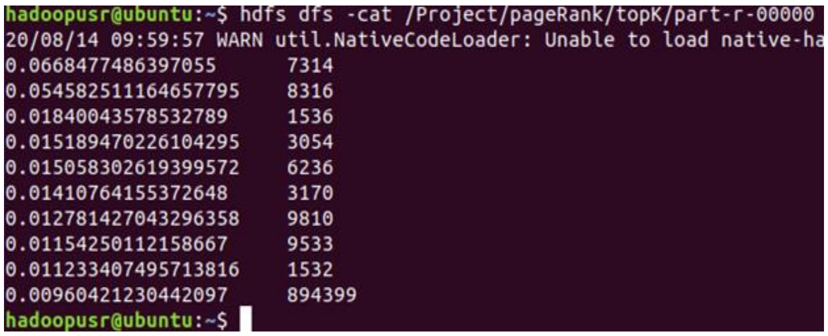
Also, the following image describes the effect of number of Fixed iteration used in calculation of page rank value.

For 3 iterations



```
hadoopusr@ubuntu:~$ hdfs dfs -cat /Project/pageRank/topK/part-r-00000
20/08/14 09:47:15 WARN util.NativeCodeLoader: Unable to load native-hado
0.14686566865683623     7314
0.11759180977156147     8316
0.040142025617284686    1536
0.030158119261188453    3054
0.027431929656477432    9810
0.026676122001262736    3170
0.02482882751101009     9533
0.024117484023687512    1532
0.023574856016006486    6236
0.019038140924691596    280
```

For 10 iterations



```
hadoopusr@ubuntu:~$ hdfs dfs -cat /Project/pageRank/topK/part-r-00000
20/08/14 09:59:57 WARN util.NativeCodeLoader: Unable to load native-ha
0.0668477486397055      7314
0.054582511164657795    8316
0.01840043578532789     1536
0.015189470226104295    3054
0.015058302619399572    6236
0.01410764155372648     3170
0.012781427043296358    9810
0.01154250112158667     9533
0.011233407495713816    1532
0.00960421230442097     894399
hadoopusr@ubuntu:~$ 
```

The first column represents page rank value and the second column represents nodeID of top 10 web pages. The Ids as well as the page rank value changed as we increased the number of iterations providing us more accurate top 10 web pages.

8. Conclusion

The Page Ranking Algorithm is an iterative graph algorithm which can be used to determine the most relevant web pages for a given search. The algorithm is implemented in such a way that it is applicable to any dataset provided in the node edges format. The page ranking implemented in this project can be enhanced by merging the two Stages of MapReduce job.

9. References

- https://snap.stanford.edu/data/web-Google.html
- Bhawiyuga and A. P. Kirana, "Implementation of page rank algorithm in Hadoop MapReduce framework," 2016 International Seminar on Intelligent Technology and Its Applications (ISITIA), Lombok, 2016, pp. 231-236, doi: 10.1109/ISITIA.2016.7828663.
- http://www.ccs.neu.edu/home/mirek/classes/CS6240-stable/GraphAlgorithms.pdf

10. Appendix

//Vertex Object

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class Vertex {

    private List<String> edgeList;
    private Double pageRank;

    public Vertex() {
    }

    public Vertex(List<String> edgeList, Double pageRank) {
        this.edgeList = edgeList;
        this.pageRank = pageRank;
    }

    public List<String> getEdgeList() {
        return edgeList;
    }

    public void setEdgeList(List<String> edgeList) {
        this.edgeList = edgeList;
    }

    public Double getPageRank() {
        return pageRank;
    }

    public void setPageRank(Double pageRank) {
        this.pageRank = pageRank;
    }

    @Override
    public String toString() {
        //String list = edgeList.toString().replace(" ", "").replace("[", "").replace("]", "");
        String list = edgeList.toString().replace(" ", "");
        return pageRank + "\t" + list;

    }
}
```

//Create Adjacency List Mapper

```
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class CreateAdjListMapper extends Mapper<LongWritable, Text, Text, Text> {

  private static final int K = 10000;
  Text k = new Text();
  Text val = new Text();

  @Override
  protected void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
    String line = value.toString();
    String[] token = line.split("\t");

    String node = null;
    String edge = null;

    try {
      if (Integer.parseInt(token[0]) <= K || Integer.parseInt(token[1]) <= K) {
        node = token[0];
        edge = token[1];
        k.set(node);
        val.set(edge);

        //Emitting nodeID as key and outgoing links as value
        context.write(k, val);

        //Emitting empty value for Dangling nodes
        context.write(val, new Text(""));
      }

    }
    catch(Exception e){

    }

  }
}
```

//Create Adjacency List Reducer

```
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class CreateAdjListReducer extends Reducer<Text, Text, Text, Text> {

  private static final Integer K = 10000;
  Text result = new Text();

  @Override
  protected void reduce(Text key, Iterable<Text> values, Context context) throws
IOException, InterruptedException {

    List<String> adjList = new ArrayList<String>();

    for(Text val: values){
      if(val.toString().equals(""))
        continue;
      adjList.add(String.valueOf(val));

    }

    Double pageRank =  (Double) (1D/ K);

    Vertex cvw = new Vertex(adjList, pageRank);
    result.set(cvw.toString());
    context.write(key, result)
       }
}
```

//Page Rank Mapper

```
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class PageRankMapper extends Mapper<LongWritable, Text, Text, Text> {
  Text k = new Text();
  Text val = new Text();
  @Override
  protected void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
    String line = value.toString();
    String[] token = line.split("\t");

    String temp = token[2].replace(" ", "").replace("[", "").replace("]", "");
    List<String> adjList;

    if(temp.equals("")){
      adjList = new ArrayList<String>();
    }
    else {
      adjList = new ArrayList<String>(Arrays.asList(temp.split(",")));
    }
    String nodeID = token[0];
    Double pageRank = Double.valueOf(token[1]);
    k.set(nodeID);
    Vertex cvw = new Vertex(adjList, pageRank);
    val.set(cvw.toString());

    if (adjList.size() > 0) {
      context.write(k, val);
      Double newRank = (Double)(pageRank / adjList.size());

      for (String edge : adjList) {
        context.write(new Text(edge), new Text(String.valueOf(newRank)));
      }
    }
```

```
        // To handle Dangling Nodes
        else{
           context.write(new Text("dummy"), new Text(String.valueOf(pageRank)));
        }
        context.write(k, val);
    }
}
```

//Page Rank Reducer

```
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class PageRankReducer extends Reducer<Text, Text, Text, Text> {
    private static final int K = 10000;
    @Override
    protected void reduce(Text key, Iterable<Text> values, Context context) throws
IOException, InterruptedException {

        List<String> adjList = new ArrayList<String>();
        Double pageRank = 0D;
        Double sum = 0D;
        Double alpha = 0.85D;
        Vertex cvw = new Vertex();


        for (Text val : values) {
           if (val.toString().split("\t").length == 2) {
              String[] token = val.toString().split("\t");
              if (!token[0].isEmpty()) {

                 adjList = Arrays.asList(token[1].replace(" ", "").replace("[", "").replace("]",
"").split(","));
                 pageRank = Double.valueOf(token[0]);
                 cvw = new Vertex(adjList, pageRank);
              }
           } else {
              sum += Double.valueOf(val.toString());
           }
        }
        if (key.toString().equals("dummy")) {
            context.getCounter(DeltaCounter.DELTA).setValue((long) (sum * 1000000000));
        }
        else {
            try {
                cvw.setPageRank((Double) (alpha * sum));
                context.write(key, new Text(cvw.toString()));
            }catch(Exception e){}
        }
    }
}
```

//Calculate Delta Mapper

```
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class CalculateDeltaMapper extends Mapper<LongWritable, Text, Text, Text> {
    private static final int K = 10000;
    Text k = new Text();
    Text val = new Text();
    private static double delta;
    @Override
    protected void setup(final Context context) throws IOException, InterruptedException {
        String confVar = context.getConfiguration().get("Delta PR");
        Long delta_long = Long.parseLong(confVar);
        delta =  ((double)delta_long / 1000000000);
    }

    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
        String line = value.toString();
        String[] token = line.split("\t");

        List<String> adjList = new ArrayList<String>(Arrays.asList(token[2].replace(" ",
"").replace("[", "").replace("]", "").split(",")));

        String nodeID = token[0];
        Double pageRank = Double.valueOf(token[1]);
        Double sum = 0D;
        Double alpha = 0.85D;
        Double delta = 0D;
        Vertex vertex = new Vertex();
        vertex.setEdgeList(adjList);

        Double newRank = pageRank + (alpha * (delta/ K) )+ ((1-alpha) / K);
        vertex.setPageRank(newRank);

        k.set(nodeID);
        val.set(vertex.toString());
        context.write(k, val);
    }
}
```

//Calculate Delta Counter Enum

```
public enum DeltaCounter {
    DELTA;
}
```

//Top-K Algorithm Mapper

```
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.mapreduce.Mapper;
import java.io.IOException;

import java.util.*;

public class TopKMapper extends Mapper<Object, Text, NullWritable, Text> {
    private TreeMap<Double, Text> LocalKWinner = new TreeMap<Double, Text>();

    @Override
    public void map(Object key, Text value, Context context)
            throws IOException, InterruptedException {
        String line = value.toString();
        String[] token = line.split("\t");

        String nodeID = token[0];
        Double pageRank = Double.valueOf(token[1]);

        LocalKWinner.put(pageRank, new Text(nodeID));

        if (LocalKWinner.size() > 10) {
```

```
        LocalKWinner.remove(LocalKWinner.firstKey());
      }
    }

    @Override
    protected void cleanup(Context context) throws IOException,
        InterruptedException {
      for(Map.Entry<Double, Text> entry: LocalKWinner.entrySet()) {
        String val = entry.getKey().toString() + "\t" + entry.getValue();
        Text t = new Text(val);
        context.write(NullWritable.get(), t);
      }
    }
}
```

//Top-k Reducer

```
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;

import java.io.IOException;
import java.util.*;

public class TopKReducer extends Reducer<NullWritable, Text, NullWritable, Text> {

    private TreeMap<Double, Text> GlobalKWinner = new TreeMap<Double, Text>();

    @Override
    public void reduce(NullWritable key, Iterable<Text> values,
                Context context) throws IOException, InterruptedException {
      for (Text value : values) {
        String[] token = value.toString().split("\t");

        GlobalKWinner.put(Double.parseDouble(token[0]), new Text(token[1]));

        if (GlobalKWinner.size() > 10) {
          GlobalKWinner.remove(GlobalKWinner.firstKey());
        }
      }

      NavigableMap<Double, Text> nmap = GlobalKWinner.descendingMap();

      for(Map.Entry<Double, Text> entry: nmap.entrySet()) {
        String op = entry.getKey().toString() + "\t" + entry.getValue();
```

```
            Text t = new Text(op);
            context.write(NullWritable.get(), t);
        }
    }
}
```

//Driver Class

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Counters;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.apache.hadoop.mapreduce.Counter;

import java.io.IOException;
//import java.nio.file.Path;

public class DriverClass extends Configured implements Tool {



//    private static final Logger logger = LogManager.getLogger(DriverClass.class);

    public static void main(String[] args) throws IOException, ClassNotFoundException,
InterruptedException {
        // write your code here
        // Create a new Job

        if (args.length != 2) {
            throw new Error("Two arguments required:\n<input-dir> <output-dir>");
        }

        try {
            ToolRunner.run(new DriverClass(), args);
        } catch (final Exception e) {
            //logger.error("", e);
            System.out.println(e);
        }

    }

    @Override
    public int run(String[] args) throws Exception {
        double delta = 0D;
        long delta_long = 0L;
        int no_of_iter = 10;

        // MapReduce job to Create Adjacency List
        final Configuration conf = getConf();;
        Job job = Job.getInstance(conf, "AdjList");
        job.setJarByClass(DriverClass.class);

        Path outDir = new Path(args[1]);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, outDir);

        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);

        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);

        job.setMapperClass(CreateAdjListMapper.class);
        job.setReducerClass(CreateAdjListReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);

        boolean status = false;

        //10 Fixed  iterations of MapReduce job to calculate Page Rank Value after the creation
of Adjacency List
        int i;
        if(job.waitForCompletion(true)) {
            for (i = 0; i < no_of_iter; i++) {
                Job job1 = Job.getInstance(conf, "PageRanking");
                job1.setJarByClass(DriverClass.class);
                final Configuration conf1 = job1.getConfiguration();

                job1.setInputFormatClass(TextInputFormat.class);
                job1.setOutputFormatClass(TextOutputFormat.class);

                job1.setMapOutputKeyClass(Text.class);
                job1.setMapOutputValueClass(Text.class);

                job1.setOutputKeyClass(Text.class);
                job1.setOutputValueClass(Text.class);

                job1.setMapperClass(PageRankMapper.class);
                job1.setReducerClass(PageRankReducer.class);

                if (i == 0) {
                   FileInputFormat.addInputPath(job1, outDir);
                } else {
                   FileInputFormat.addInputPath(job1, new Path(args[1], "out" + Integer.toString(i
- 1)));
                }
                FileOutputFormat.setOutputPath(job1, new Path(args[1], "dangling_out" +
Integer.toString(i)));

                job1.waitForCompletion(true);

                Counters delta_counter = job1.getCounters();
                Counter c = delta_counter.findCounter(DeltaCounter.DELTA);
                delta_long = c.getValue();

                //Mapper to Calculate Delta for Dangling node handling
                Job job2 = Job.getInstance(conf, "DeltaCalculation");
                job2.setJarByClass(DriverClass.class);
                final Configuration conf2 = job2.getConfiguration();

                conf2.setLong("Delta PR", delta_long);
                job2.setMapperClass(CalculateDeltaMapper.class);

                job2.setOutputKeyClass(Text.class);
                job2.setOutputValueClass(Text.class);

                job2.setInputFormatClass(TextInputFormat.class);
                job2.setOutputFormatClass(TextOutputFormat.class);

                FileInputFormat.addInputPath(job2, new Path(args[1], "dangling_out" +
Integer.toString(i)));
                FileOutputFormat.setOutputPath(job2, new Path(args[1], "out" +
Integer.toString(i)));

                status = job2.waitForCompletion(true);
            }

            //MapReduce job for TopK Algorithm
```

```
            if (i == no_of_iter) {
                Job job3 = new Job(conf, "Top Ten Users by Reputation");
                job3.setJarByClass(DriverClass.class);
                final Configuration conf3 = job3.getConfiguration();

                job3.setMapperClass(TopKMapper.class);
                job3.setReducerClass(TopKReducer.class);
                job3.setNumReduceTasks(1);
                job3.setOutputKeyClass(NullWritable.class);
                job3.setOutputValueClass(Text.class);
                FileInputFormat.addInputPath(job3, new Path(args[1] ,"out" +
Integer.toString(no_of_iter - 1)));
                FileOutputFormat.setOutputPath(job3, new Path(args[1], "topK"));
                job3.waitForCompletion(true);
            }
            return 0;
        } else {
            return 1;
        }
    }
}
```