**Assignment Hadoop Project Report**

**Primary Classes for this project**

**DiffMap1**

String currentNode = sections[0].split(" ")[0];  
String currentWeight = sections[0].split(" ")[1];  
currentWeight = currentWeight.replace("[", "");  
currentWeight = currentWeight.replace("]", "");  
  
context.write(new Text(currentNode), new Text(currentWeight));

This function handles input lines by extracting node and weight information, eliminating brackets from the weight, and emitting key-value pairs for the node and weight. These pairs are then ready for subsequent processing.

### DiffRed1

String node = key.toString();  
 List<String> inputList = textToList(values);  
 double absoluteDifference = 0.0;  
  
 if (inputList.size() == 1) {  
 absoluteDifference = Double.*parseDouble*(inputList.get(0));  
 } else if (inputList.size() == 2) {  
 double rank1 = Double.*parseDouble*(inputList.get(0));  
 double rank2 = Double.*parseDouble*(inputList.get(1));  
 absoluteDifference = Math.*abs*(rank1 - rank2);  
 } else {  
 throw new IOException("Incorrect data format");  
 }  
  
 context.write(key, new Text(Double.*toString*(absoluteDifference)));  
}  
  
private List<String> textToList(Iterable<Text> values) {  
 List<String> list = new ArrayList<String>();  
 for (Text item : values) {  
 list.add(item.toString());  
 }  
  
 return list;  
}

This reducer computes and outputs the absolute difference in PageRank values assigned to a node across various iterations. It accommodates both single and double values, ensuring the correct data format.

### DiffRed2

List<Double> differences = textToDoubleList(values);  
 for(double diff : differences) {  
 if(diff > diff\_max) {  
 diff\_max = diff;  
 }  
 }  
 context.write(new Text(), new Text(Double.*toString*(diff\_max)));  
}  
  
private List<Double> textToDoubleList(Iterable<Text> values) {  
 List<Double> list = new ArrayList<Double>();  
 for(Text item: values) {  
 list.add(Double.*parseDouble*(item.toString()));  
 }  
 return(list);  
}

This reducer calculates and outputs the maximum difference in PageRank values assigned to a node across different iterations. This step is crucial for monitoring the variations in PageRank values throughout the iterative computation process.

### FinMapper

String[] sections = line.split("\t"); // Splits each line  
if (sections.length > 2)   
{throw new IOException("Incorrect data format");}  
String currentNode = sections[0].split(" ")[0];  
String currentWeight = sections[0].split(" ")[1];  
currentWeight = currentWeight.replace("[", "");  
currentWeight = currentWeight.replace("]", "");  
  
context.write(new DoubleWritable(Double.*parseDouble*(currentWeight)), new Text(currentNode));

The FinMapper processes input lines with defined formats, extracting node and weight information, and emits the weight as a double and the node as text for subsequent processing.

### InitMapper

String[] values = line.split(":");  
String nodeId = values[0];  
String edgeNodes = values[1].trim();  
  
context.write(new Text(nodeId), new Text(edgeNodes));

This mapper handles lines containing node and edge information in the adjacency list format. It extracts the node and edge nodes and emits them as key-value pairs for further use in subsequent stages.

### IterMapper

String currentNode = sections[0].split(" ")[0];  
String currentWeight = sections[0].split(" ")[1];  
currentWeight = currentWeight.replace("[", "");  
currentWeight = currentWeight.replace("]", "");  
  
String[] edgeNodes = sections[1].split(" ");  
double newWeight = ((double)1/edgeNodes.length) \* Double.*parseDouble*(currentWeight);  
for(String edgeNode: edgeNodes) {  
 context.write(new Text(edgeNode), new Text(Double.*toString*(newWeight)));  
}  
context.write(new Text(currentNode), new Text("@" + sections[1] + "@"));

### The IterMapper processes lines containing node ranks and lists of adjacent nodes. It computes and outputs key-value pairs for adjacent nodes with calculated weights. Additionally, it emits a key-value pair for the current node along with a flagged adjacency list.IterReducer

String node = key.toString();  
 double weight = 0;  
 String adjList = "";  
 List<String> valueList = textToStringList(values);  
   
 //Sum incoming edge weights  
 for(String incomingWeight: valueList) {  
 if(incomingWeight.contains("@")) {  
 adjList = incomingWeight.replaceAll("@", "");  
 }  
 else {  
 weight += Double.*parseDouble*(incomingWeight);  
 }  
 }  
 weight \*= d;  
 weight += (1-d);  
 context.write(new Text(node + " [" + Double.*toString*(weight) + "]"),new Text(adjList));  
}  
  
private List<String> textToStringList(Iterable<Text> values) {  
 List<String> list = new ArrayList<String>();  
 for(Text item: values) {  
 list.add(item.toString());  
 }  
 return(list);  
}

### The reducer calculates PageRank values by considering incoming edge weights and adjacency lists. It accumulates the weights of incoming edges, applies the PageRank formula, and emits node identifiers along with the calculated weights and adjacency lists.

### PageRankDriver

### PageRankDriver handles data initialization, iterative computation, convergence checking, data joining, and result summarization. It's designed to work with a distributed computing framework to handle large-scale graph data for PageRank calculations.

### Testing

Testing Results: I conducted various tests on the Hadoop project, evaluating its performance under different scenarios. In one test, I used the following input data:

2 3

1 3 4 5

1 2

1

1 4

I executed this test with 5 reducers and received the result in around 6 minutes:

2.0623  
1  
0.7875  
0.7875  
0.36250000000000004

With 20 reducers and I received the result in 7 mintues :

1 2.0666666666666666  
3 1.3541666666666665  
5 1.0708333333333333  
2 0.7458333333333333  
4 0.36250000000000004

Test 2

During the assessment of the Hadoop project, I conducted multiple tests to analyze its performance under diverse conditions. One test involved the following input data:

2 3

1 3 4 5

1 2

1

1 4

Test 1: I executed this test with 5 reducers and received the results in approximately 6 minutes:

* Node 1: 2.0666666666666664
* Node 3: 1.3541666666666665
* Node 5: 0.7458333333333333
* Node 2: 0.7458333333333333
* Node 4: 0.36250000000000004

I repeated the same test with 20 reducers, and the results were consistent:

* Node 1: 2.0666666666666664
* Node 3: 1.3541666666666665
* Node 5: 0.7458333333333333
* Node 2: 0.7458333333333333
* Node 4: 0.36250000000000004

Test 2: Another test utilized the input:

2 3

1 3 4 5

With 5 reducers, the test took around 3 minutes to complete:

* Node 1: 0.3625000000000004
* Node 3: 0.7875
* Node 5: 0.3625000000000004
* Node 2: 0.575
* Node 4: 0.3625000000000004

Similarly, with 20 reducers, the results were obtained in approximately 8 minutes:

* Node 1: 0.3625000000000004
* Node 3: 0.7875
* Node 5: 0.3625000000000004
* Node 2: 0.575
* Node 4: 0.3625000000000004

Notably, the latter test demonstrated significantly faster execution times compared to the former.

PageRank Algorithm on Wikipedia Page Names: I applied the PageRank algorithm to Wikipedia page names using 10 reducers running in AWS EMR, resulting in the following results:

... (more entries)

The\_King\_of\_Fighters 1.5524999999999998  
Tales\_of\_the\_Jedi 1.5524999999999998  
!!! 2.6617499999999987  
Matthew\_18 4.740000000000002  
Castlevania 4.740000000000002  
List\_of\_nicknames\_of\_European\_Royalty\_and\_Nobility 4.740000000000002  
Matthew\_26 9.840000000000014  
Theme\_Time\_Radio\_Hour 9.840000000000014  
Library\_of\_Congress\_Classification 13.155000000000022  
List\_of\_drugs 16.97999999999999  
Star\_Wars 35.33999999999982This computation took approximately 50 minutes.

Another test using 20 reducers produced similar results, showcasing consistency in performance:

... (more entries)

America 1.5524999999999998  
List\_of\_Austrian\_films 1.5524999999999998  
List\_of\_Medal\_of\_Honor\_recipients 1.5524999999999998  
The\_King\_of\_Fighters 1.5524999999999998  
Tales\_of\_the\_Jedi 1.5524999999999998  
!!! 2.6617499999999987  
Matthew\_18 4.740000000000002  
Castlevania 4.740000000000002  
List\_of\_nicknames\_of\_European\_Royalty\_and\_Nobility 4.740000000000002  
Matthew\_26 9.840000000000014  
Theme\_Time\_Radio\_Hour 9.840000000000014  
Library\_of\_Congress\_Classification 13.155000000000022  
List\_of\_drugs 16.97999999999999  
Star\_Wars 35.33999999999982

This computation took approximately 45 minutes to complete.

These tests provided insights into the behavior and efficiency of the Hadoop project under various conditions, allowing for a comprehensive evaluation of its performance.

### Conclusion

Determining the optimal number of reducers for a Hadoop job involves a nuanced process, taking into account factors such as data size, available resources (including AWS configuration if applicable), scalability requirements, and potential overhead. Striking the right balance among these elements is crucial.

When deciding on the number of reducers, it is essential to consider the data volume, the capabilities of your hardware or cloud infrastructure (such as an AWS setup), the potential for uneven data distribution (data skew), and the associated overhead of having too many reducers.

After conducting experiments with different reducer counts and testing various data sizes, it is recommended to start with a moderate number of reducers. Monitor the job's progress and performance closely as it runs and evolves. Through trial and error, experiment with different reducer counts to identify the most suitable configuration. It's important to note that there is no fixed formula for determining the perfect number of reducers; the decision depends on factors such as the complexity of the task, the specific cluster setup, and your intentions for future scalability.