Assignment 2 Page Rank - Report

9961244 EE14 吳德霖

1 INSTRUCTION

The following instructions indicate how to compile and execute my program.

1.1 PART1

To compile:

- javac -classpath ../../hadoop-core-1.2.1.jar -d classFolder/ ./*.java
- jar -cvf PageRank1.jar -C classFolder .

To run:

- hadoop jar PageRank1.jar hw2.PageRank1 [input] [output]

1.2 PART2

To compile:

- javac -classpath ../../hadoop-core-1.2.1.jar -d classFolder/ ./*.java
- jar -cvf PageRank2.jar -C classFolder .

To run:

- hadoop jar PageRank2.jar hw2.PageRank2 [input] [output]
- The [input] here should be the output folder of part1 plus “/part-00000”

1.3 PART3

To compile:

- javac -classpath ../../hadoop-core-1.2.1.jar -d classFolder/ ./*.java
- jar -cvf PageRank3.jar -C classFolder .

To run:

- hadoop jar PageRank3.jar hw2.PageRank3 [input] [output]
- The [input] here should be the output folder of part2’s iter10 plus “/part-00000”

1.4 PART4

To compile:

- javac -classpath ../../hadoop-core-1.2.1.jar:../../hbase-0.94.18.jar -d lhFolder/ ./*.java
- jar -cvf loadToHbase.jar -C lhFolder .
- javac -classpath ../../hadoop-core-1.2.1.jar -d classFolder/ javaFolder/*
- jar -cvf InvIndex.jar -C classFolder .

To run:

- hadoop jar InvIndex.jar hw2.InvIndex [input] [output]
- The input is the original input file with the output be the table.
- hadoop jar loadToHbase.jar loadToHbase [input1] [hbase Table1] [input2] [hbase Table2]
- Input 1 and 2 are page rank result and the inverted index table respectively.

1.5 PART5

To compile:

- javac -classpath ../../hadoop-core-1.2.1.jar:../../hbase-0.94.18.jar -d qeFolder/ ./*.java
- jar -cvf queryEngine.jar -C qeFolder .

To run:

- hadoop jar queryEngine.jar queryEngine [hbase Table1] [hbase Table2]
- Hbase table 1 is the table of the inverted index table, while the second table is the page rank's hbase table.

2 DESIGN

2.1 PART1

2.1.1 Mapper

```
//String patternStr = "<title>[\\w+\\s+/+]+";
String patternStr = "<title>[^<]+";
Pattern pattern = Pattern.compile(patternStr);
String patternStr2 = "\\[\\[[^\\]]+\\]\\]";
//String patternStr2 = "\\[\\[[\\w+\\s+/+]+";
Pattern pattern2 = Pattern.compile(patternStr2);
```

As shown above, I simply use the regular expression to do the initial parsing, to filter out the actual information of title and the list of link as long with the matcher function.

2.1.2 Reducer

```
for(i=1; i<str.length; i++){
  if(!content.contains(str[i])){
    content.add(str[i]);
    outStr = outStr + str[i] + "]";
  }
}
```

The reducer designed here in part 1 is also relatively simple, the only requirement for the basic version is to output the links separated by "]" just as the sample output did.

2.1.3 Illegal Nodes and Dangling Nodes

The illegal nodes are that they're not exist in the list of the title but appear in the out link list, by using the JobConf API, I can share the whole title list with the mapper and reducer, so that each of the element in the out link can be checked if it's listed in the title list. For the current section and the next, I designed another mapper and reducer to be executed in main, so I can comment out this function if I require to match the answers with those provided by the TA.

```
public void configure(JobConf job){
  String titleStr = new String();
  String nodeStr = new String();
  titleStr = job.get("titleList");
  String tstr[] =  titleStr.split("]");
  int i;
  for(i=0; i<tstr.length; i++){
    titleList.add(tstr[i]);
  }
  nodeStr = job.get("nodeList");
  String nstr[] =  nodeStr.split("]");
  for(i=0; i<nstr.length; i++){
    if(!nodeList.contains(nstr[i])){
      nodeList.add(nstr[i]);
    }
  }
}
```

By taking advantage of the conf.set() and conf.get(), I could store any useful information I needed to share with the whole structure.

Dangling nodes are that without an out link, but is out links of some other title nodes.

```
// judge title if it's a dangling node
if(str2[1].equals("]")){
  if(nodeList.contains(str1[0])){
    word.set("nullNode");
    outVal.set(str1[0]);
  }
  else{
    word.set(str1[0]);
    outVal.set("1]");
  }
}
```

As shown above, I defined the virtual node as the nullNode, and if the system detects that a title node is one of the elements in an out link but does not have any out links itself, the system would store the nullNode string to the output key value pair reversely. Since I could aggregate the result of such nullNode in the reducer, the possibility of adding a virtual node to the table by map reduce job is achieved as follows.

```
if(inputKey.equals("nullNode")){
  while (values.hasNext()) {
    Text value = (Text) values.next();
    nullStr = "nullNode]";
    String outputKey = new String();
    outputKey = value.toString();
    //outStr = outStr + outputKey + "]";
    word1.set(outputKey);
    outVal1.set("1]"+nullStr);
    output.collect(word1, outVal1);
  }
  outStr = titleStr;
}
```

And such nullNode would be connected as the mother node of all the other title nodes.

2.2 PART2

2.2.1 Algorithm

```
map( key: [url, pagerank], value: outlink_list )
    for each outlink in outlink_list
        emit( key: outlink, value: pagerank/size(outlink_list) )

    emit( key: url, value: outlink_list )

reducer( key: url, value: list_pr_or_urls )
    outlink_list = []
    pagerank = 0

    for each pr_or_urls in list_pr_or_urls
        if is_list( pr_or_urls )
            outlink_list = pr_or_urls
        else
            pagerank += pr_or_urls

    pagerank = 1 - DAMPING_FACTOR + ( DAMPING_FACTOR * pagerank )

    emit( key: [url, pagerank], value: outlink_list )
```

The pseudo code of the page rank algorithm.

2.2.2 Mapper

```
for(int i=1; i<inputValue.length; i++){
  if(!outLinkList.contains(inputValue[i])){
    outLinkList.add(inputValue[i]);
    outLinkTmp = outLinkTmp + inputValue[i] + "]";
    word.set(inputValue[i]);
    double ratio = pageRank / (inputValue.length-1);
    outVal.set(ratio+"]"+"v");
    output.collect(word, outVal);
  }
}
```

I stored an extra element v as a tag to justify if it's a page rank whose type is double.

2.2.3 Reducer

```
if(firstLink.equals("nullNode")){
  pageRank = 1.0;
}
else{
  pageRank = randomJump / titleNum + ((1-randomJump) * pageRank);
}
```

I kept the page rank of dangling nodes unchanged.

2.3 PART3

2.3.1 Mapper

```
double pageRank = Double.parseDouble(inputValue[0]);
String PR = new String();
PR = inputValue[0];
word.set(PR);
outVal.set(inputKey[0]);
output.collect(word, outVal);
```

Simply output the page rank and the out links, but in the reversed form.

2.3.2 Reducer

```
while(values.hasNext()){
  Text value = (Text) values.next();
  valueTmp = value.toString();
  strTmp = strTmp + valueTmp + "]";
  //word.set(title);
}
outVal.set(strTmp);
```

Since the key now is the page rank sorted, the value would be aggregated as those links corresponding to these page ranks, the reducer function could do the sorting itself.

2.3.3 Key Comparator

To ensure the resulting list is sorted from the top page with the highest page rank to the last page with the least page rank, I added a key comparator to reverse the original sorting list.

```
public int compare(WritableComparable w1, WritableComparable w2){
        Text t1 = (Text) w1;
        Text t2 = (Text) w2;
        String str1 = new String();
        String str2 = new String();
        str1 = t1.toString();
        str2 = t2.toString();
        double val1 = Double.parseDouble(str1);
        double val2 = Double.parseDouble(str2);
        int result = Double.compare(val1, val2) * (-1);
        return result;
}
```

2.4 PART4

2.4.1 Mapper

```
String patternStr = "<title>[^<]+";
Pattern pattern = Pattern.compile(patternStr);
int i;
            try{
  String lineOffset = key.toString();
  String inputStr = value.toString();
  Matcher matcher = pattern.matcher(inputStr);
  String term = new String();
  boolean matchFound = matcher.find();
  while(matchFound) {
    for(i=0; i<=matcher.groupCount(); i++) {
      String groupStr = matcher.group(i);
      //word.set(groupStr.substring(7,groupStr.length()));
      term = groupStr.substring(7,groupStr.length());
    }
    if(matcher.end() + 1 <= inputStr.length()) {
      matchFound = matcher.find(matcher.end());
    }
    else{
      break;
    }
  }
```

Use the regular expression and matcher again to parse the title list in order not to lost essential information separated by space.

```
//TODO eliminate the useless characters
String str1[] = inputStr.split("<text xml:space=");
String str2[] = str1[1].split("</text>");
String inputStr2 = str2[0];
String str[] = inputStr2.split("\\W");
for(i=1; i<str.length; i++){
  if(str[i].compareTo("") != 0){
    outVal.set(term);
    word.set(str[i]);
    output.collect(word, outVal);
  }
}
```

Use the splitting function with respect to “\\W” to discard all the useless notations to make the word to be searched clear.

2.4.2 Reducer

```
 while (values.hasNext()) {
   Text value = (Text) values.next();
   inputKey = value.toString();
   if(!fileName.contains(inputKey)){
     fileName.add(inputKey);
     sumTmp = String.valueOf(sum);
```

Reuse the reducer of inverted index creation designed in HW1, but here we discard the function of gathering information about the line offsets.

2.4.3 Load to Hbase

```
while((line = in.readLine()) != null){
  // handle the data you get from HDFS
  String str[] = line.split("\t");
  String str2[] = str[1].split("]");
  for(i=0; i<str2.length; i++){
    Put put_data = new Put(str2[i].getBytes());
    put_data.add("Page".getBytes(),"PageRank".getBytes(),str[0].getBytes());
    ITable.put(put_data);
  }
            // put them into the table you create in HBase
}
in.close();
```

Using the hbase APIs to create two new hbase tables for page rank and the inverted index, by doing the jobs as shown above twice, I could continuously put the processed data with parsing functions to the table just created.

2.5 PART5

2.5.1 Query Engine

```
String queryWord = new String();
queryWord = inputData;
Get get_data = new Get(queryWord.getBytes());
Result res = iiTable.get(get_data);
byte[] value = res.getValue(Bytes.toBytes("InvIndex"), Bytes.toBytes("Table"));
String valueStr = Bytes.toString(value);
```

Use the hbase API to search for the word from the table with Get function and store it into the Result typed result. Since the returning value is of type Result, I had to transform it into byte by the getValue function and finally do the toString() to store the returned information as a string typed object.

```
String queryPage = new String();
for(i=0; i<page.size(); i++){
  queryPage = page.get(i);
  Get get_PR = new Get(queryPage.getBytes());
  Result resPR = prTable.get(get_PR);
  byte[] valuePR = resPR.getValue(Bytes.toBytes("Page"), Bytes.toBytes("PageRank"));
  String prStr = Bytes.toString(valuePR);
```

The above shows how I get the needed value of the page rank corresponding to those containing the word to be searched. Each result would be stored into an array list and waited to be sorted. But this time the returned value should be stored as a double typed object.

```
double prVal = Double.parseDouble(prStr);
pageRank.add(prVal);
```

2.5.2 Sample Running (input-1M)

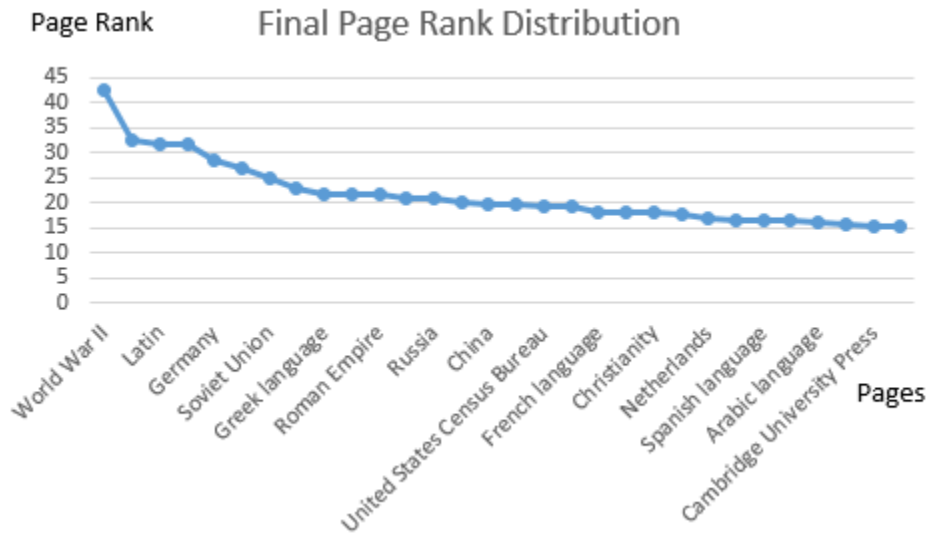
```
********** The Query Engine **********
Please enter the keyword you'd like to query:
To
You required to search for: To

******* Query Result *******
Page Ranking 1 : Academy Award
PageRank = 0.5030466926009548
-----------------------------------
Page Ranking 2 : Aristotle
PageRank = 0.04852178705750257
-----------------------------------
Page Ranking 3 : Ayn Rand
PageRank = 0.03170545992045452
-----------------------------------
Page Ranking 4 : Alabama
PageRank = 0.0017647058823529415
-----------------------------------
Page Ranking 5 : Achilles
PageRank = 0.0017647058823529415
-----------------------------------
Page Ranking 6 : Abraham Lincoln
PageRank = 0.0017647058823529415
-----------------------------------
Page Ranking 7 : Anarchism
PageRank = 0.0017647058823529415
-----------------------------------
```

```
********** The Query Engine **********
Please enter the keyword you'd like to query:
aas
You required to search for: aas

******* Query Result *******
Sorry the result of aas is empty!
```

```
********** The Query Engine **********
Please enter the keyword you'd like to query:
Much
You required to search for: Much

******* Query Result *******
Page Ranking 1 : Anarchism
PageRank = 0.0017647058823529415
-----------------------------------
Page Ranking 2 : Altruism
PageRank = 0.0017647058823529415
-----------------------------------
```

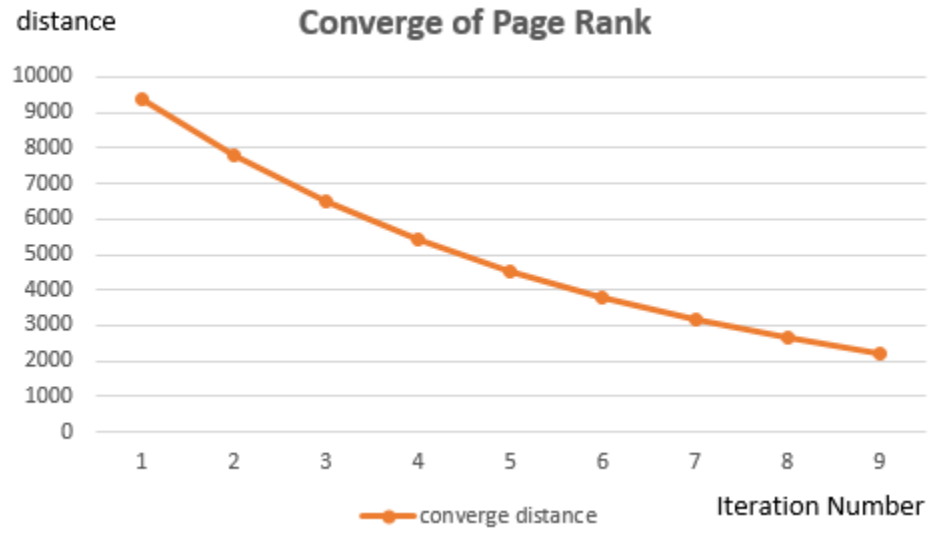
3 EXPERIMENTS

3.1 DISTRIBUTION OF PAGE RANK



The graph above is the distribution of page rank, with the x axis ranging from the page that has the top page rank to the thirtieth, though some of the page names are omitted by the excel graphing procedure. We can see that the top page exhibits page rank around 42, while the last page has page rank indicating roughly 18.

3.2 CONVERGE RATE



The graph above is the converging rate with respect to the distance after each iteration. It's obviously the graph is exhibiting a descending tendency, and which is what we expected since after each iteration, the distance of page ranks between the two corresponding page rank

algorithm should decrease and eventually converge to a given value. If we keep doing the iteration beyond the tenth, we would observe the distance is as well decreasing, in other words, more and more converge.