Class Number: CS6240 Parallel Data Processing in Map Reduce
HW Number: 4
Name: Neha Pradhan

ASSIGNMENT – 4 REPORT

Program Discussion:

Describe briefly how each step of your program is transforming the data. Be precise, e.g., by showing the structure of the input and output as a table.

1. Preprocessing: The following chunks of code describe the transformations that take place during the preprocessing:

```
//Takes each line in the input file and applies a parse function to it that returns a string with the page
//name and its outlinks in a format that can be easily transformed in to an RDD of pageName and its
//list of outlinks. The filter transformation filters out any erroneous lines that might have thrown an
//exception while being parsed. The flatmap transformation splits the line delimited by "#", to take
//into account all the page's outlinks for page rank evaluation
val preprocessedRDD = fileContent
    .map(line => Bz2WikiParser.parseXML(line))
    .filter(preprocessedLine => (preprocessedLine.trim.length >= 1))
    .flatMap(line => line.split("#"));
```

Step	Input Structure	Output Structure
fileContent.map(line => Bz2WikiParser.parseXML(line))	File of lines (pageName:XML)	RDD[(pageName~List[Outlinks]#Outlink1~#Outlink2~#...)]
.filter(preprocessedLine => (preprocessedLine.trim.length >= 1))	RDD[(pageName~List[Outlinks]#Outlink1~#Outlink2~#...)]	RDD[(pageName~List[Outlinks]#Outlink1~#Outlink2~#...)]
.flatMap(line => line.split("#"))	RDD[(pageName~List[Outlinks]#Outlink1~#Outlink2~#...)]	RDD[(pageName~List[Outlinks]), (Outlink1~), (Outlink2~), …]

```
//Takes the RDD of String and transforms it into RDD[(String, List[String])] that has been reduced so
//that there exists distinct page names as keys along with a list of its outlinks as values
val pairRDD = preprocessedRDD
    .map(createPairRDD)
    .reduceByKey(_++_);
```

Step	Input Structure	Output Structure
preprocessedRDD.map(createPairRDD)	RDD[(pageName~List[Outlinks]), (Outlink1~), (Outlink2~), …]	Pair RDD[(pageName, List[Outlinks]), (Outlink1, List()), (Outlink2, List()), …]
.reduceByKey(_++_);	Pair RDD[(pageName, List[Outlinks]), (Outlink1, List()), (Outlink2, List()), …]	Pair RDD[(pageName, List[Outlinks]), (Outlink1, List()), (Outlink2, List()), …] with distinct keys

Class Number: CS6240 Parallel Data Processing in Map Reduce
HW Number: 4
Name: Neha Pradhan

Note: createPairRDD is a helper function that splits the incoming parsed and transformed line at pre-determined points to create a pairRDD with the pageName as key and its list of outlinks as values.

Following table outlines the transformations taking place during the preprocessing:

Program Phase	Input Structure	Output Structure
Input	File of lines	RDD of lines
Parsing	RDD of lines	RDD[String]
Accounting for outlinks as individual pages	RDD[String]	RDD[String]
Transformation to obtain required format	RDD[String]	RDD[(String, List[String])]

2. PageRank evaluation: The following chunks of code describe the transformations that take place during the PageRank evaluation:

```
//The input RDD is transformed to an RDD that only has each outlink's pageName as key and the
//pageRank contribution from the current page as its value. The current page is also added to this
//RDD with the pageRank set to zero, so that it is accounted for in case it is a page with in-links
val outlinksPageRanksDistribution = pageRankRDD
    .flatMap{ case(pageName, (urls, rank)) =>
    val currentRecord = (pageName, 0.0);
    val outLinksSize = urls.size;
    urls.map(url => (url, rank/outLinksSize)).union(List(currentRecord));
    }
```

Step	Input Structure	Output Structure
pageRankRDD.flatMap{ case(pageName, (urls, rank)) => val currentRecord = (pageName, 0.0); val outLinksSize = urls.size; urls.map(url => (url, rank/outLinksSize)) .union(List(currentRecord));}	Pair RDD[(pageName, List[Outlinks]), (Outlink1, List()), (Outlink2, List()), …]	Pair RDD[(pageName, pageRank)]

```
//The RDD created with the pageName and pageRank contribution towards it is reduced by key which
// sums up all the pageRank contributions for the page and then goes on to evaluate its final
//pageRank for the current iteration using the pageRank formula and taking into account dangling
//node contribution
val reducedPageRanksRDD = outlinksPageRanksDistribution
            .reduceByKey((sum, pr) => sum + pr)
            .mapValues(v => (alphaContribution + (alphaInverse * (danglingLinkMass + v))));
```

Class Number: CS6240 Parallel Data Processing in Map Reduce
HW Number: 4
Name: Neha Pradhan

Step	Input Structure	Output Structure
outlinksPageRanksDistribution .reduceByKey((sum, pr) => sum + pr)	Pair RDD[(pageName, pageRank)]	Pair RDD[(pageName, pageRank)] with distinct keys along with the sum of pageRank contribution
.mapValues(v => (alphaContribution + (alphaInverse * (danglingLinkMass + v))));	Pair RDD[(pageName, pageRank)] with distinct keysalong with the sum of pageRank contribution	Pair RDD[(pageName, pageRank)], the pageRank is one calculated using the PageRank formula

```
//This transformation is done merely to bring the RDD to be returned in the required format so that
//it can be processed until the pageRank values converge
val newPageRankRDD = pageRankRDD.join(reducedPageRanksRDD)
                .map{ case(pageName, ((urls, oldPR), newPR)) => (pageName, (urls, newPR))};
```

Step	Input Structure	Output Structure
pageRankRDD.join(reducedPageRanksRDD)	Pair RDD[(pageName, pageRank)]	Pair RDD[(pageName, (List[Outlinks], old pageRank), new pageRank)]
.mapValues(v => (alphaContribution + (alphaInverse * (danglingLinkMass + v))));	Pair RDD[(pageName, (List[Outlinks], old pageRank), new pageRank)]	Pair RDD[(pageName, (List[Outlinks], new pageRank))]

Following table outlines the transformations taking place during the PageRank evaluation:

Program Phase	Input Structure	Output Structure
PageRank distribution among outlinks	RDD[(String, (List[String], Double)]	RDD[String, Double]
Summing PageRanks with formula	RDD[String, Double]	RDD[String, Double]
Transformation to obtain required format	RDD[String, Double]	RDD[(String, (List[String], Double)]

3. Main program: The main program does not necessarily have too many transformations that entirely change the structure of the RDDs, whatever transformations done in the main program are just trivial transformations in order to transform the RDDs in a format required for pageRank evaluation.

```
//Transforming the RDD to include a default page rank before the first run of page rank evaluation
var pairRDDWithPR = preprocessedPairRDD
                .map{ case(k, v) => (k, (v, startingPR)) }
                .persist;
```

Class Number: CS6240 Parallel Data Processing in Map Reduce
HW Number: 4
Name: Neha Pradhan

Step	Input Structure	Output Structure
preprocessedPairRDD .map{ case(k, v) => (k, (v, startingPR)) }	RDD[(pageName, List[Outlinks])	RDD[(pageName, (List[Outlinks],0.0))]

```
//Computing the dangling node contribution to the page rank for this iteration. This is an action and
//it calculates the total of the page ranks of dangling nodes to distribute among all the pages
val danglingPageRanksSum = pairRDDWithPR.values
            .filter(record => (record._1.isEmpty))
            .values
            .sum();
```

Step	Input Structure	Output Structure
pairRDDWithPR.values	RDD[(pageName, (List[Outlinks], pageRank))	RDD[(List[Outlinks], pageRank)]
.filter(record => (record._1.isEmpty))	RDD[(List[Outlinks], pageRank)]	RDD[(List[Outlinks], pageRank)]
.values	RDD[(List[Outlinks], pageRank)]	RDD[(pageRank)]
.sum();	RDD[(pageRank)]	Dangling mass contribution : Double

```
//Takes the top 100 pages alongwith their page ranks and saves the results to an output file
val topKRDD = sc.parallelize(pairRDDWithPR
    .map(record => (record._2._2, record._1))
    .top(100), 1)
    .saveAsTextFile(args(1));
```

Step	Input Structure	Output Structure
sc.parallelize(pairRDDWithPR .map(record => (record._2._2, record._1)) .top(100), 1) .saveAsTextFile(args(1));	RDD[(pageName, (List[Outlinks], pageRank))	RDD[(pageRank, pageName)]

Following table outlines the transformations taking place in the Main program:

Program Phase	Input Structure	Output Structure
Initializing pageRank RDD with default page ranks before first iteration	RDD[(String, List[String])	RDD[(String, (List[String], Double)]
Calculating dangling node contribution	RDD[(String, (List[String], Double)]	Double
Top K evaluation	RDD[(String, (List[String], Double)]	File of output lines

Class Number: CS6240 Parallel Data Processing in Map Reduce
HW Number: 4
Name: Neha Pradhan

For each step, state if the dependency is narrow (no shuffling) or wide (shuffling). How many stages does your Spark have?

Narrow dependency: Occurs when each partition of the parent RDD is used by at most one partition of the child RDD
Wide dependency: Occurs when each partition of the parent RDD is used by multiple partitions of the child RDD
The above explanation indicates that transformations like map, filter, flatmap etc. exhibit narrow dependency and hence usually do not require a shuffle and actions like join, groupByKey, reduceByKey etc. exhibit wide dependency and usually require shuffling. Looking at the steps in the program, we can define the dependencies as mentioned below:

Preprocessing:
The transformation below requires no shuffling as all operations performed on the RDD have narrow dependency

- val preprocessedRDD = fileContent
  .map(line => Bz2WikiParser.parseXML(line))
  .filter(preprocessedLine => (preprocessedLine.trim.length >= 1))
  .flatMap(line => line.split("#"));

The transformation below may require shuffling as the reduceByKey transformation exhibits wide dependency

- val pairRDD = preprocessedRDD
  .map(createPairRDD)
  .reduceByKey(_++_);

PageRank:
The transformation below may not require shuffling as all operations performed on the RDD have narrow dependency

- val outlinksPageRanksDistribution = pageRankRDD
  .flatMap{ case(pageName, (urls, rank)) =>
  val currentRecord = (pageName, 0.0);
  val outLinksSize = urls.size;
  urls.map(url => (url, rank/outLinksSize)).union(List(currentRecord));
  }

The transformation below may require shuffing as the reduceByKey operation performed on the RDD has wide dependency

- val reducedPageRanksRDD = outlinksPageRanksDistribution
  .reduceByKey((sum, pr) => sum + pr)
  .mapValues(v => (alphaContribution + (alphaInverse * (danglingLinkMass + v))));

The transformation below requires shuffling as the join operation performed on the RDD has wide dependency

Class Number: CS6240 Parallel Data Processing in Map Reduce
HW Number: 4
Name: Neha Pradhan

- val newPageRankRDD = pageRankRDD.join(reducedPageRanksRDD)
  .map{ case(pageName, ((urls, oldPR), newPR)) => (pageName, (urls, newPR))};

Main program:

The transformation below may not require shuffling as all operations performed on the RDD have narrow dependency

- var pairRDDWithPR = preprocessedPairRDD
  .map{ case(k, v) => (k, (v, startingPR)) }
  .persist;

The transformation below may not require shuffling as all operations performed on the RDD have narrow dependency

- val danglingPageRanksSum = pairRDDWithPR.values
  .filter(record => (record._1.isEmpty))
  .values
  .sum();

The transformation below may not require shuffling as all operations performed on the RDD have narrow dependency

- val topKRDD = sc.parallelize(pairRDDWithPR
  .map(record => (record._2._2, record._1))
  .top(100), 1)
  .saveAsTextFile(args(1));

There are around 20 stages in my Spark program where each transformation is considered as a stage.

Performance Comparison:

Machines	Processing Time (Hadoop) in seconds	Processing Time (Spark) in seconds
6 machines	2643	4980
11 machines	1641	2700

Discuss which system is faster and briefly explain what could be the main reason for this performance difference.

The Spark system should be faster as it works with RDDs which is usually loaded into memory once and computations are performed in memory. In contrast, map reduce has a lot of IO operations which include reading from and writing files to the HDFS. Spark has lazy execution style and therefore can analyze and optimize entire computations whereas in MapReduce each computation comprises of independent jobs which read from and write to slow storage. MapReduce programs are low-level whereas Spark programs are high-level and more elegant.