Assignment - 4
Satya Akhil Chowdary Kuchipudi
Section - 1

High-level description of Spark methods used in the code :

(Reference : Spark Scala docs)

conf.setMaster()
The master URL to connect to, such as "local" to run locally with one thread, "local[4]" to run locally with 4 cores, or "spark://master:7077" to run on a Spark standalone cluster.

conf.setAppName()
Set a name for your application.

sc.textfile()
Read a text file from HDFS, a local file system (available on all nodes), or any Hadoop-supported file system URI, and return it as an RDD of Strings. If there are multiple files in the folder, the number of partitions used will be same as number of files in the folder

pairRDD.map(k => new Bz2WikiParser().getAdjList(k)) : Transformation
Return a new RDD by applying a function to all elements of this RDD.
Lambda : String (line of text) => String (nodeid + adjacency list)

filter(k => (k != null)) : Transformation
Return a new RDD containing only the elements that satisfy a predicate.
Lambda : RDD[String] => RDD[String] (all nulls are removed)

map(fields => (fields(0), buildArray(fields(1))))
Returns a pairRDD from RDD based on the lambda passed

adjRDD.count : Action
Returns count of the (k,v) pairs in RDD

sc.accumulator[Double](0.0)
Initializes the double accumulator to 0.0

adjRDD.join(rankRDD) : Transformation
Lambda : RDD, RDD => RDD
Joins 2 pairRDD based on same field among both RDDs. (a,b).join(a,c) => (a,(b,c))

.flatmap{ case (url, (links, rank)) => some steps on url, links, rank} Transformation
Lambda : RDD => RDD
Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results. In my code, lambda iterates on each adjacency link in links and then emits link, pagerank/list.size

def evaluate(contributions: RDD[(String, Double)]) = {
contributions.sparkContext.runJob(contributions,(iter: Iterator[(String, Double)]) => {
while(iter.hasNext) iter.next() })}
Forces to do lazy evaluation of RDD. Used this to make sure accumulator is being updated properly by creating an action.

contributions.reduceByKey((a, b) => a + b) : Transformation (Reduce is an Action)
Lambda : RDD => RDD
Merge the values for each key using an associative and commutative reduce function.For same key, we add the page rank contributions.

mapValues(v => 0.15 / docuCount + 0.85 * (v + (danglingvalue / docuCount)))
MapValues considers, only values from pairRDD and applies the lambda to the value. Retains the original partitioning of RDD.

rankRDD.subtractByKey(newrankRDD)
Lambda : RDD, RDD => RDD
Removes all (k, v) pairs common in rankRDD & newrankRDD from rankRDD.

finallist = rankRDD.sortBy(v => v._2, false)
Lambda : RDD => RDD
Sorts all the (k, v) pairs in pairRDD based on the lambda function passed. false option uses descending order.

take(100)
Lambda : RDD => Array
Takes the top 100 (k, v) pairs from RDD and gives as a sequence array.

saveAsTextFile(args(1))
Save this RDD as a compressed text file, using string representations of elements.

persist()
Persist this RDD with the default storage level (MEMORY_ONLY).

Steps:

1. Load a text file into RDD.
2. Apply map over this RDD to do parse over each line and get nodeid and adjacency list
3. Filter out if nulls are returned and copy into Adjacency RDD or graph
4. Build a rank RDD from original RDD by initializing values to 1/total Nodes.
5. Now create iterative loop for 10 times
    5.1. In each iteration, do a join over rankRDD and adjRDD and apply flatmap over each (k,v) pair and if adjacency list is null, add the rank into dangling loss
    5.2. For other case, emit for each link in Adjacency List , emit the nodeid and page rank contribution to each node from current node.
    5.3. Then apply reduce by key for this contributions and apply dampening factor as well
    5.4. For remaining nodes that are missing, add those ranks to dangling loss and update back to pagerank.
6. Now sort based on map values and take the top 100.

For each line of your Scala Spark program, describe where and how the respective functionality is implemented in your Hadoop jobs.

val pairRDD = sc.textFile(args(0), 20)
This is done using addInputPath function in Hadoop. Spark's version can be used to set number of partitions used for reading files and it's much easy to configure.

```
val adjRDD = pairRDD.map(k => new Bz2WikiParser().getAdjList(k))
    //filter if return is null
    .filter(k => (k != null))
    .map(line => line.split("``"))
    // convert into RDD with pagenode as key, adjacency list as value
    .map(fields => (fields(0), buildArray(fields(1)))).persist()
```

This is done in Pre-Processor in Hadoop PageRank in separate Job. It generates Adjacency List. Spark does it using map, passing the parse function as lambda. Then filter null out of the values and then split accordingly and build a pairRDD for further transformations. The spark API is much simpler compared to hadoop as it abstracts many functions and makes it easier for programmer to code. And the program itself is very verbose and easy to understand.

```
var rankRDD = adjRDD.map(k => (k._1, 1 / docuCount.toDouble))
```

Initializes the first iteration pagerank values to 1/N. In the Hadoop code, I have to do if else check to see if the iteration is first or not and then update accordingly. Spark makes it much simpler.

```
for (iteration <- 0 to 15) {
    //val danglingval = sc.doubleAccumulator("DELTA_LOSS")
    var danglingval = sc.accumulator[Double](0.0)
    // contributions from each node
    val contributions = adjRDD.join(rankRDD).flatMap {
      //signature for joined k,v; for each link in links, emit (link, rank/links.size)
      case (url, (links, rank)) => {
        // if dangling node, add the value to dangling
        if (links.size == 0) {
          danglingval+=rank
          None
          //links.map(adjnode => (adjnode, rank / links.size))
        }
        else
        //emit each adjnode contribution
          links.map(adjnode => (adjnode, rank / links.size))
      }
    }.persist()
```

This is the most significant step between Hadoop & Spark. In hadoop, we need to specify 10 jobs separately and pass around the common variables like delta loss and total nodes across the jobs using Conf object or Counters. This created a lot of mess as there is loss of precision due to counter only accepting long values and also due to reducer not able to access counters. Spark abstracts the whole process and makes it as simple as a for loop, however we need to be careful regarding using vars inside the loop. As we need shared objects across each map.
In Hadoop, the logic inside each job is handled in map as each link in adjacency list will be emitted with whatever contribution to that node. If the node is dangling, don't emit anything and update the dangling loss directly.

```
// new Rank RDD is achieved by doing reducebyKey and then applying page rank formulae
val newrankRDD = contributions.reduceByKey((a, b) => a + b)
      .mapValues(v => 0.15 / docuCount + 0.85 * (v + (danglingvalue / docuCount)))
// add the dangling value nodes which are missing back to original rankRDD
rankRDD = rankRDD.subtractByKey(newrankRDD).mapValues(x => 0.15 / docuCount + 0.85 *
(danglingvalue / docuCount))
      .union(newrankRDD)
```

This is where we update the pagerank with dampening factor and other values. This is done in spark by joining rankRDD and newrankRDD and seeing which keys are missing through subtractbyKey. Then we find which primary dangling nodes are missing from newrankRDD and then update the page ranks accordingly and then add these back to rankRDD. This is done in Hadoop using if else check for dangling node and updating dangling loss accordingly. To build

```
val newcontribs = contribs.subtractByKey(graph).mapValues(x=> {danglingval.add(x)}).count()
```

This will update the dangling loss to include secondary dangling nodes. This is done in Hadoop using a boolean to differentiate if the node is traversed in the given pages or not.

```
 var finallist = rankRDD.sortBy(v => v._2, false).take(100)
 sc.parallelize(finallist, 1).saveAsTextFile(args(1))
```

This is the top-100 in Hadoop. take creates an array from RDD and we can parellielize to write to a text file specified. This is very simple compared to the Hadoop since spark abstracts the logic from user and allowing api to handle complex calculations easily.

Comparisons :

The main advantage over Hadoop is usage of RDDs and pairRDDs. We can save more time and can do it more efficiently if the in-memory is capable of it. In hadoop, we had to shuffle the adjacency list from mapper to reducer and then reducer to hdfs and so on, this requires lot of disk IOs and lot of latency. But in spark, everything is saved in-memory as RDD and all map/reduce can access it with much ease. Another big advantage over Hadoop is it's API, provides many functions with simple API calls. This makes it much easier to code in Spark than Hadoop. And on top of it, Spark has full Scala support, which makes the whole thing much more easier. We can use java whenever needed and can use Scala to simplify the code. Third advantage over Hadoop is it's Iterative capability, Spark can run iterative algorithms with much ease while hadoop makes it much complex if each iterative job is dependent over previous job. However, I think if Spark in-memory runs out either when caching or evaluation, the efficiency of overall will degrade.

Run Times :

Hadoop Run time for 6 machines
1 hr 15 mins

Hadoop Run time for 11 machines
35 mins 45 sec

Spark Run Time for 6 machines
1 hr 54 mins

Spark Run Time for 11 machines
53 mins 45 sec

Technically Spark must be faster than Hadoop. However, the parser is written in Java and pre-processing step is taking lot of time compared to hadoop. Pagerank calculation is actually happening more faster than hadoop's. This may be due to the parser written in Java or due to poor handling of reading and pre-processing the text files. I am not sure about the exact reason why this is happening, I think it maybe because of hadoop is superior in batch processing than spark. But, in reality due to RDD spark turns out more faster than Hadoop.

Hadoop Top-100 Simple Dataset

Please find the Result File in hw4/report/hadoop-output/local

Hadoop Top-100 Full Dataset

Please find the Result File in hw4/report/hadoop-output/aws

Spark Top-100 Simple Dataset

Please find the Result File in hw4/report/spark-output/local

Spark Top-100 Full Dataset

Please find the Result File in hw4/report/spark-output/aws

There is minor changes among pageranks.. This is because of few reasons. One being the way I am updating page ranks in hadoop vs spark. Second, in hadoop I used double to long and long to double conversion which had some loss of precision compared to spark. I am assuming, the way of updating pagerank is modified mainly due to how spark processes iterative algorithms differently from hadoop's implementation.