CS6240, Assignment 4, Rakesh Buchan Krishna Murthy

<u>Program Discussion</u>

Steps	Input Structure	Output Structure	Shuffling Type
Reads each line from the input file and converts it into a pair RDD of (PageName, List(outLink1, outLink2, …)) format			
.map(line => Preprocessor.readLine(line))	PageName : xml content	RDD of (PageName # outLink1~outLink2~…)	Narrow
.filter(line => !line.equals(""))	RDD of (PageName # outLink1~outLink2~…)	Removes all RDDs with ""	Narrow
.map(line => line.split(" # "))	RDD of (PageName # outLink1~outLink2~…)	RDD of (PageName, outLink1~outLink2~…)	Narrow

.map(line => if (line.length == 1) {
(line(0), List())
} else {
(line(0),
line(1).split("~").toList) }
	RDD of (PageName, outLink1~outLink2~…)	Pair RDD of (PageName, List(outLink1, outLink2, …))	Narrow

Ensures that all pages and outlinks present in the corpus are considered			
var allPagesWithLinks = input.values	Pair RDD of (PageName, List(outLink1, outLink2, …))	List(outLink1, outLink2, …)	Narrow
.flatMap { link => link }	List(outLink1, outLink2, …)	outLink1 outLink2 . . .	Narrow
.keyBy(link => link)	outLink1 outLink2 . . .	outLink1, outLink1 outLink2, outLink2 . . .	Narrow
.map(line => (line._1, List[String]()))	outLink1, outLink1 outLink2, outLink2 . . .	outLink1, List[] outLink2, List[] . . .	Narrow
Combines the above created RDD for each outlink with the original RDD, so that any outLinks which were not present in Corpus get added to the RDD			
.union(input) .reduceByKey((outlink1, outlink2) => outlink1.++(outlink2))	Pair RDD of (PageName, List(outLink1, outLink2, …)) & outLink1, List[] outLink2, List[]	Pair RDD of (PageName, List(outLink1, outLink2, …))	Union - narrow reduceByKey-Wide

	. . .		
Creates a new pair RDD for PageName and it's Rank			
var pageWithPageRanks = allPagesWithLinks.keys .map { page => (page, initialPageRank) }	Pair RDD of (PageName, List(outLink1, outLink2, …))	Pair RDD of (PageName, PageRank)	Narrow
var updatedPageRank = allPagesWithLinks .join(pageWithPageRanks)	Pair RDD of (PageName, List(outLink1, outLink2, …)) & Pair RDD of (PageName, PageRank)	Pair RDD of (PageName, (List(outLink1, outLink2, …), PageRank))	Wide
.values	Pair RDD of (PageName, (List(outLink1, outLink2, …), PageRank))	List(outLink1, outLink2, …), PageRank)	Narrow
If the outlink was a dangling node, update danglingScore, else distribute the current page's rank to it's outlinks.			

.flatMap {
case (outlinks, pageRank) =>
val outlinksCount = outlinks.size
if (outlinksCount == 0) {
danglingScore += pageRank //updates dangling score
List()
} else {
outlinks.map { link => (link, pageRank / outlinksCount) }
}
}
	List(outLink1, outLink2, …), PageRank)	List ((outLink1, PageRank1), (outlink2, PageRank2))
		Simplified to
		Pair RDD of (outLink, PageRank)
			Narrow

.reduceByKey(_ + _)	Pair RDD of (outLink, PageRank)	Pair RDD of (outLink, PageRank)	Wide
For the pages, whose ranks didn't get updated by other page's contribution, their ranks will be assigned to 0.0			
pageWithPageRanks = pageWithPageRanks .subtractByKey(updatedPageRank) .map(pageName => (pageName._1, 0.0))	Pair RDD of (PageName, PageRank)	Pair RDD of (PageName, PageRank)	Narrow

Extracts just the PageRank and calculates the latest PageRank			
.mapValues[Double](pageRank => alpha * initialPageRank + (1 - alpha) * (finalDanglingScore / noOfPages + pageRank))	PageRank	PageRank	Narrow
Extract top 100 ranks and the respective pages			
val top100Pages = sc.parallelize(pageWithPageRanks .map(x => (x._2, x._1)) .top(100), 1)	Pair RDD of (PageName, PageRank)	Pair RDD of (PageName, PageRank)	Wide

My Spark program has 15 stages, which performs some kind of transformation on the input data.

Performance Comparison

Workers	Scala Program (in seconds)	MR Program (in seconds)
5	7020	3194
10	3180	2043

Theoretically, Scala program should be faster because-

i) An option to persist RDD's reduce IO operations
ii) Smarter task allocation to the executors which already contains the input data

However in my case MR approach produced better results. The reasons for the same could be –

i) In Spark implementation, I couldn't parallelize the preprocessing, since the program is written in java
ii) In Spark implementation, I persisted only a single RDD, (the one which is generated after the preprocessing stage), may be persisting more RDD's could make the processing fast.

With increasing the number of worker machines, the difference in execution time seems to reduce as per my findings.