Akshay Syal

1. **[12 points] Show the pseudo-code for the PR program in Spark Scala. Since many Scala functions are similar to pseudo-code, you may copy-and-paste well-designed (good variable naming!) and well-commented Scala code fragments here. Notes: Your program must support k and the number of PR iterations as parameters. Your program may take shortcuts to exploit the structure of the synthetic graph, in particular that each page has at most 1 outgoing link. (Your program should work on the synthetic graphs, no matter the choice of k>0, but it does not need to work correctly on more generally structured graphs.)**

```
val conf = new SparkConf().setAppName("Page Rank")
   val sc = new SparkContext(conf)

   // Taking input of params k and iter and converting to int
   val k = args(0).toInt
   val iter = args(1).toInt



   // Since a special synthetic graph is being created such that
   // each node has at most 1 outgoing edge, the following approach is undertaken:
   // A list of numbers from 1 to k^2 is created.
   // They are mapped to the next number or zero [if it's a dangling node]
   // Graph and ranks must have same partitioner to avoid shuffling during join
   // graph is cached as it won't be computed again

   val partitioner = new org.apache.spark.HashPartitioner(k)

   val graph = sc.parallelize((1 to k*k).flatMap(i=>{
     if (i % k == 0) List((i,0))
     else List((i,i+1))
   })).partitionBy(partitioner).cache()

   var ranks = sc.parallelize((0 to k*k).flatMap(i=>{
     if (i == 0) List((i,0.0))
     else List(( i, 1.0/(k*k) ))
   })).partitionBy(partitioner)

  // inner join graph and ranks to produces page rank of each node
  // however nodes having in-degree 0 are filtered output [1,4,7 for k=3]
  // they are  added to explicitly with rank 0
  // page rank of 0 is redistributed, its page rank is made 0

  for (i <- 1 to iter) {
   val joinedGraphRanks = graph.join(ranks).values
   val ranksExcludingFirstNodes = joinedGraphRanks.reduceByKey((a, b) => a + b)

   var firstNodes = sc.parallelize((1 to k).flatMap(i=>{
     List((1+k*(i-1),0.0))
    }))

    val allNodesRanks = ranksExcludingFirstNodes.union(firstNodes)

    val adjustedPROfzero = allNodesRanks.lookup(0).head/(k*k)
    ranks = allNodesRanks.flatMap{ case (node, rank) =>
        if (node==0) List((node,0))
        else List((node,rank+adjustedPROfzero))
      }

  }


  val output = ranks.collect()
  output.foreach(tup => println(s"${tup._1} has rank:  ${tup._2} ."))
  println(ranks.values.reduce(_+_))
```

2. **[10 points] Show the link to the source code for this program in your Github Classroom repository.**

   https://github.com/CS6240/hw4-spark-AkshaySyal/blob/main/src/main/scala/wc/PageRank.scala

3. **[10 points] Run the PR program locally (not on AWS) for k=100 for 10 iterations. Report the PR values your program computed for pages 0 (dummy), 1, 2,…, 19.**

   Q3_output

```
0 has rank: 0.0.
1 has rank: 1.0936852726843594E-6.
2 has rank: 2.17654197831244E-6.
3 has rank: 3.2486773304194495E-6.
4 has rank: 4.310197481020451E-6.
5 has rank: 5.361207531120449E-6.
6 has rank: 6.40181154112045E-6.
7 has rank: 7.432112541120451E-6.
8 has rank: 8.452212541120447E-6.
9 has rank: 9.462212541120451E-6.
10 has rank: 1.046221254112045E-5.
```

```
11 has rank: 1.1046221254112045E-4.
12 has rank: 1.1046221254112045E-4.
13 has rank: 1.1046221254112045E-4.
14 has rank: 1.1046221254112045E-4.
15 has rank: 1.1046221254112045E-4.
16 has rank: 1.1046221254112045E-4.
17 has rank: 1.1046221254112045E-4.
18 has rank: 1.1046221254112045E-4.
19 has rank: 1.1046221254112045E-4.
```

4. **[19 points] Run the PR program locally (not on AWS) for k=100. Set the loop condition so that exactly 1 iteration is performed and report the lineage for Ranks after that iteration. Change the loop condition so that exactly 2 iterations are performed and report the lineage for Ranks after those 2 iterations. Then change the loop condition again so that exactly 3 iterations are performed and report the lineage for Ranks after those 3 iterations.**

   Q4_Output

5. **[15 points] Find out if Spark executes the complete job lineage or if it re-uses previously computed results. Make sure you are not using cache() or persist() on the Ranks RDD. (You may use it on the Graph RDD.) Since the PR values in RDD Ranks in iteration 10 depend on Ranks from iteration 9, which in turn depends on Ranks from iteration 8, and so on, we want to find out if the job triggered by the lookup action in iteration 10 runs all 10 iterations from scratch, or if it uses Ranks from iteration 9 and simply applies one extra iteration to it.**
   a. **Let's add a print statement as the first statement inside the loop that performs an iteration of the PR algorithm. Use println(s"Iteration ${i}") or similar to print the value of loop variable i. The idea is to look at the printed messages to determine what happened. In particular, if a job executes the complete lineage, we might hope to see "Iteration 1" when the first job is triggered, then "Iteration 1" (again) and "Iteration 2" for the second job (because the second job includes the result of the first iteration in its lineage, i.e., a full execution from scratch would run iterations 1 and 2), then "Iteration 1," "Iteration 2," and "Iteration 3" when the third iteration's job is triggered, and so on. But would that really happen? To answer this question, show the lineage of Ranks after 3 iterations and report if adding the print statement changed the lineage.**

      Q5_a_Output
      I ran the program for k=100 and iter=3
      After inspecting the log I found that there are 3 instances of "Iteration": Iteration 1, Iteration 2, Iteration 3.

Adding the print statement did not change the lineage.

b. **Remove the print statement, run 10 iterations for k=100, and look at the log file. You should see lines like “Job … finished: lookup at …, took …” that tell you the jobs executed, the action that triggered the job (lookup), and how long it took to execute. If Spark does not re-use previous results, the growing lineage should cause longer computation time for jobs triggered by later iterations. On the other hand, if Spark re-uses Ranks from the previous iteration, then each job runs only a single additional iteration and hence job time should remain about the same, even for later iterations. Copy these lines from the log file for all jobs executed by the lookup action in the 10 iterations. Based on the times reported, do you believe Spark re-used Ranks from the previous iteration?**

Q5_b_Output

```
2024-03-10 23:47:42,469 INFO scheduler.DAGScheduler: Job 0 finished: lookup
at PageRank.scala:59, took 6.237185 s

2024-03-10 23:47:44,707 INFO scheduler.DAGScheduler: Job 1 finished: lookup
at PageRank.scala:59, took 2.150717 s

2024-03-10 23:47:46,243 INFO scheduler.DAGScheduler: Job 2 finished: lookup
at PageRank.scala:59, took 1.408840 s

2024-03-10 23:47:47,723 INFO scheduler.DAGScheduler: Job 3 finished: lookup
at PageRank.scala:59, took 1.409009 s

2024-03-10 23:47:48,962 INFO scheduler.DAGScheduler: Job 4 finished: lookup
at PageRank.scala:59, took 1.178487 s

2024-03-10 23:47:50,197 INFO scheduler.DAGScheduler: Job 5 finished: lookup
at PageRank.scala:59, took 1.170765 s

2024-03-10 23:47:51,733 INFO scheduler.DAGScheduler: Job 6 finished: lookup
at PageRank.scala:59, took 1.503336 s

2024-03-10 23:47:53,759 INFO scheduler.DAGScheduler: Job 7 finished: lookup
at PageRank.scala:59, took 1.998367 s

2024-03-10 23:47:54,877 INFO scheduler.DAGScheduler: Job 8 finished: lookup
at PageRank.scala:59, took 1.052412 s
```

```
2024-03-10 23:47:56,242 INFO scheduler.DAGScheduler: Job 9 finished: lookup
at PageRank.scala:59, took 1.336699 s
```

Apart from the first 2 jobs all the other job times are similar. I believe Spark re-used Ranks from the previous iteration.

c. **So far we have not asked Spark to cache() or persist() Ranks. Will this change Spark's behavior? To find out, add ".cache()" to the command that defines Ranks in the loop. Run your program again for 10 iterations for k=100 and look at the log file. What changed after you added cache()? Look for lines like "Block … stored as values in memory" and "Found block … locally". Report some of those lines and discuss what they tell you about the caching behavior and reuse of previously computed versions of Ranks. (Do not report those lines if they are related to RDD Graph.) Were you able to find those lines also in the log file created by the program that did not apply cache() to Ranks?**

   Q5_c_Output

   Lines like these where found 900 times in log:

```
2024-03-11 00:13:06,217 INFO storage.BlockManager: Found block
rdd_1_{some number between 1 and 99} locally
```

   Lines like these where found 1167 times in log:

```
2024-03-11 00:13:06,506 INFO memory.MemoryStore: Block broadcast_24
stored as values in memory (estimated size 6.8 KiB, free 430.7 MiB)
2024-03-11 00:13:06,954 INFO memory.MemoryStore: Block rdd_83_4 stored as
values in memory (estimated size 3.5 KiB, free 430.7 MiB)
```

   For the program that did not apply cache to ranks, I was able to find lines like "Block … stored as values in memory" (131 times) and "Found block … locally" (900 times same as ranks).

6. [6 points] Set k=10,000 and run 10 iterations of your program on EMR, using 1 master and 5 worker nodes (all cheap machines as for HW 1). Report the running time on EMR and show the links to the log file(s) and the output file(s) for this run. The output is the final content of Ranks.

a. **Creating the entire graph to turn it into an RDD at once may cause memory problems. (Think about where this happens: on the master/driver or in the tasks/executors?)**
b. **Think about a creative solution to address those memory issues. E.g., you can create a small portion of the graph first and then use RDD operations to add the missing pieces. Here you may exploit the special graph structure (structure of**

each “linear chain” itself or the fact that linear chains are identical to each other except that the nodes have different IDs).

7. [12 points] Show the pseudo-code for the MapReduce program. Make sure it clearly shows how you solved the dangling-page problem.

8. [10 points] Show the link to the source code for this program in your Github Classroom repository.

9. [6 points] Set k=10,000 and run 10 iterations of your program on EMR, using 1 master and 5 worker nodes (the same machines as for the Spark program). Report the running time on EMR and show the links to the log file(s) and the output file(s) for this run. The output is the final content of Ranks.

10. [5 bonus points] If you solved the bonus challenge, show the pseudo-code for your Spark program and report the PR values for the 15 users that have the highest PR values after 10 iterations.