CS 6240: Assignment 4

Goals: (1) Gain deeper understanding of action, transformation, and job submission in Spark. (2) Implement PageRank in MapReduce and Spark.

This homework is to be completed individually (i.e., no teams). You must create all deliverables yourself from scratch: it is not allowed to copy someone else's code or text, even if you modify it. (If you use publicly available code/text, you need to cite the source in your report!)

Please submit your solution through Canvas by the due date shown online. For late submissions you will lose one percentage point per hour after the deadline. This HW is worth 100 points and accounts for 15% of your overall homework score. To encourage early work, you will receive a 10-point bonus if you submit your solution on or before the early submission deadline stated on Canvas. (Notice that your total score cannot exceed 100 points, but the extra points would compensate for any deductions.)

To enable the graders to run your solution, make sure your project includes a standard **Makefile** with the same top-level targets (e.g., *local* and *aws*) as the one presented in class. As with all software projects, you must include a **README** file briefly describing all the steps necessary to build and execute both the standalone and the AWS Elastic MapReduce (EMR) versions of your program. This description should include the build commands and fully describe the execution steps. This README will also be graded, and you will be able to reuse it on all this semester's assignments with little modification (assuming you keep your project layout the same).

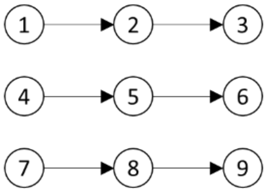
You have about 2 weeks to work on this assignment. Section headers include recommended timings to help you schedule your work. The earlier you work on this, the better.

PageRank in Spark (Week 1)

In addition to implementing a graph algorithm from scratch to better understand the BFS design pattern and the influential PageRank algorithm, this first part of the assignment will also explore the subtleties of Spark's actions and transformations, and how they affect lazy evaluation and job submission. We could in principle work with the Twitter followership data to find the most important users based on PageRank, but we leave that as an optional challenge. Instead, this assignment will work with synthetic data. Synthetic data avoids the high pre-processing and cleaning cost of real data, and it allows us to focus on the essence of a challenge. Thoughtful creation of synthetic data is an important skill for big-data program design, testing, and debugging.

Recall that Spark *transformations* describe data manipulations, but do not trigger execution. This is the "lazy evaluation" in Spark. *Actions* on the other hand force an immediate execution of all upstream operations needed to produce the desired result. Hence an action triggers the submission and execution of the corresponding Spark job, which is defined by all operations in the lineage of the result. What will happen when an iterative program performs both actions and transformations in a loop? What goes into the lineage after 1, 2,…, or k loop iterations? And will the entire lineage be executed?

Let us find out by exploring a program that computes PageRank with dangling pages for a simple synthetic graph. Your program should work with two data tables: **Graph** stores pairs (v1, v2), each encoding an edge from some vertex v1 to another vertex v2. **Ranks** stores pairs (v1, pr), encoding the PageRank pr for each vertex v1. To fill these tables with data, create a graph that consists of k linear chains, each with k vertices. Number the vertices from 1 to k^2, chain by chain. Here k should be a program parameter to control problem size. The figure shows an example for k=3. Notice that the last vertex in each linear chain is a dangling page. We will use the single-dummy-vertex approach to deal with dangling pages. This means that your program also must create a single dummy vertex, let's give it number 0 (zero), and add it to Ranks. Similarly, add an edge (d, 0) for each dangling page d. Set the initial PR value for the k^2 real vertices in Ranks to $1/k^2$; set the initial PR value of the dummy vertex to 0.



For simplicity, we recommend you implement the program using (pair) RDDs, but you may choose to work with DataSet instead. The following instructions assume an RDD-based implementation. Start by exploring the PageRank Scala program included in the Spark distribution. Make sure you fully understand what each statement is doing. Create a simple example graph and step through the program, e.g., on paper or using the interactive Spark shell. You will realize that the example program does not handle dangling pages, i.e., dangling pages lose their PR mass in each iteration. Are there other problems?

Your program will have a structure similar to the example program, but follow these requirements and suggestions:

- We strongly recommend you make k a parameter of your Spark Scala program and generate RDDs Graph and Ranks directly in the program. There are many examples on the Web on how to create lists of records and turn them into (pair) RDDs.
    - If you absolutely cannot figure this out, you may pre-create the graph and PR data and load them from a text file. However, that approach is more complicated.
- Make sure you add dummy vertex 0 to Ranks and the corresponding dummy edges to Graph.
- Initialize each PR value in Ranks to $1/k^2$, except for vertex 0, whose initial PR value should be zero.
    - The example program in the Spark distribution sets initial PR values to 1.0, and its PR computation adds 0.15 instead of 0.15/#vertices for the random jump probability. Intuitively, they multiply each PR value by #vertices. This is a valid approach, and you may do the same. Just make sure you do not get the formulas from the example program and from the module confused.
- (Optional) Make sure that Graph and Ranks have the same Partitioner. This will significantly reduce join cost.

- Check if the join computes exactly what you want. Does it matter if you use an inner or an outer join in your program?
- To read out the total dangling PR mass accumulated in dummy vertex 0, use the *lookup* method of pair RDD. Then re-distribute this mass over all real vertices.
- When debugging your program, see if the PR values still add up to 1 (or to #vertices if you use the alternative formulas like in the example program) after each iteration.
- Add a statement right after the end of the for-loop for the PR iterations to write the debug string of Ranks to the log file.

Now you are ready to explore the subtleties of Spark lazy evaluation. First explore the lineage of Ranks as follows:

1. Set the loop condition so that exactly 1 iteration is performed. Report the lineage for Ranks after that iteration.
2. Change the loop condition so that exactly 2 iterations are performed. Report the lineage for Ranks after those 2 iterations. Did it change?
3. Change the loop condition again, so that exactly 3 iterations are performed. Report the lineage for Ranks after those 3 iterations. Did it change?

The lineage describes the job that would be executed for an action applied to that version of Ranks. Now take a closer look at pair RDD's lookup method. **Is it a transformation or an action?** If it is an action, and it is executed in each loop iteration, what job would be triggered by it?

Now that you understand the lineage of Ranks in each iteration of the for-loop, and when a job would be submitted, let's try to find out **what actually gets executed**. Is Spark smart enough to avoid re-computation of intermediate results that had been computed earlier? For example, assume a Spark program looks like this:

```
1. val myRDD1 = oldRDD.map(…)
2. myRDD1.collect()
3. val myRDD2 = oldRDD.reduceByKey(…).map(…)
4. myRDD2.collect()
```

The first job execution is triggered by line 2. Assume it computes myRDD1 from scratch, including some expensive computation to create oldRDD, which is in its lineage. The next job execution is triggered by line 4. Since myRDD2 also depends on oldRDD, Spark could create oldRDD again from scratch. What if oldRDD was still available in memory from the earlier job triggered by line 2? Then it would be more efficient for Spark to simply **re-use** the existing copy of oldRDD!

Use Spark textbooks and online resources to find out if Spark is smart enough to realize such RDD re-use opportunities. Then study this empirically in your PageRank program where the lineage of Ranks in iteration i depends on all previous (i-1) iterations:

1. Can you instrument your program with the appropriate printing or logging statements to find out execution details for each job triggered by an action in your program?
2. See if you can find other ways to make Spark tell you which steps of an RDD lineage were executed, and when Spark was able to avoid execution due to availability of intermediate results from earlier executions.
3. Change the caching behavior of your program by using cache() or persist() on Ranks. Does it affect the execution behavior of your program? Try this for small k, then for really large k (so that Ranks barely fits into memory).

For an optional 5-point bonus (final score cannot exceed 100), try to run your PageRank program on the Twitter followership data.

PageRank in MapReduce (Week 2)

Implement the PageRank program in MapReduce and run it on the synthetic graph. You may choose any of the methods we discussed in module and in class for handling dangling pages, including global counters (try if you can read it out in the Reduce phase) and order inversion.

In contrast to the Spark program, generate the synthetic graph in advance and feed it as input data to your PageRank program. Follow the approach from the module and store the graph as a set of vertex objects, each containing the adjacency list and the PageRank value.

Since we will work with small input, make sure that your program creates **at least 20 Map tasks**. You can use NLineInputFormat to achieve this.

For an optional 5-point bonus (final score cannot exceed 100), try to run your PageRank program on the Twitter followership data.

Report

Write a brief report about your findings, using the following structure.

Header (4 points)

This should provide information like class number, HW number, and your name. **Also include a link to your Github Classroom repository for this homework. (4 points)**

PageRank in Spark (40 points total)

Show the pseudo-code for your PageRank program in Spark Scala. Since many Scala functions are similar to pseudo-code, you may copy-and-paste them here whenever appropriate. (20 points)

Report, which of the operations in your program perform an action. (4 points)

Run the program for 10 iterations for k=100 and report the final PageRanks of the vertices with ID numbers 0 (dummy), 1,…, 99, 100. (4 points)

Show the lineage for RDD Ranks after 1, 2, and 3 loop iterations. (6 points)

Discuss how you determined what was actually executed by a job triggered by your program. (2 points)

Report your observations: (1) Is Spark smart enough to figure out that it can re-use RDDs computed for an earlier action? (2) How do persist() and cache() change this behavior? (4 points)

Note: It is not required to run these programs on AWS. When running on your local machine, make sure you do so using **make local**. The IDE is for development and debugging, not for production runs.

PageRank in MapReduce (20 points total)

Show the MapReduce pseudo-code. Do not just copy-and paste Java code! (10 points)

Describe briefly (1 paragraph) how you solved the dangling-page problem. (4 points)

Run your program for k=1000 and 10 iterations on the following two configurations:

- 5 cheap machines (1 master and 4 workers)
- 9 cheap machines (1 master and 8 workers)

Use the same machine type for all experiments.

Make sure the program creates at least 20 Map tasks, e.g., by using NLineInputFormat. (4 points)

Report the running time for each cluster. (2 numbers, 1 point each)

Deliverables

IMPORTANT: The submission time of your solution is the latest timestamp of any of the deliverables included. For the PDF it is the time reported by Canvas; for the files on Github it is the time the files were pushed to Github, according to Github. We recommend the following approach:

1. Push all files to Github and make sure everything is there (see deliverables below). (Optional: Create a version number for this snapshot of your repository.)
2. Submit the report on Canvas. Make sure you hit "submit," not just "save." Open the submitted file to verify everything is okay.
3. Do not make any more changes in the Github repository.

Submit the report as a **PDF** file on Canvas. To simplify the grading process, please name this file yourFirstName_yourLastName_HW#.pdf. Here yourFirstName and yourLastName are your first and last name, as shown in Canvas; the # character should be replaced by the HW number. So, if you are Amy Smith submitting the solution for HW 7, your solution file should be named Amy_Smith_HW7.pdf:

1. The report as discussed above. Make sure it includes the links to the project, log and output files on Github as described above. (1 PDF file)

Make sure the following is in your **Github Classroom** repository:

2. Log files for PageRank execution for k=100 on Spark. (syslog/stderr or similar, 2 points)
3. Log files for the two MapReduce executions—one on small the other on large cluster—for which you reported running times. (2 syslog/stderr or similar, 4 points)
4. All output files produced by those same successful runs of the Spark program (local is fine) and the MapReduce program (small and large cluster) (3 sets of part-r-… or similar files). (6 points)
5. The Spark Scala project, including source code, build scripts etc. (12 points)
6. The MapReduce project, including source code, build scripts etc. (12 points)

Note: If you cannot get your MapReduce program to run on AWS, then you can instead include the log files and output from execution on your local machine for partial credit.

IMPORTANT: Please ensure that your code is properly documented. There should be comments concisely explaining the role/purpose of a class. Similarly, if you use carefully selected keys or custom Partitioners, make sure you explain their purpose (what data will be co-located in a Reduce call; does input to a Reduce function have a certain order that is exploited by the function, etc.). But do not over-comment! For example, a line like "SUM += val" does not need a comment. As a rule of thumb, you want to add a brief comment for a block of code performing some non-trivial step of the computation. You also need to add a brief comment about the role of any major data structure you introduce.