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 indicate what was copied and cite the source in your report!)

Please submit your solution as a *single PDF file* on Gradescope (see link in Canvas) by the due date and time shown there. During the submission process, you need to tell Gradescope on which page the solution to each question is located. Not doing this will result in point deductions. In general, treat this like a professional report. There will also be point deductions if the submission is not neat, e.g., it is poorly formatted. (We want our TAs to spend their time helping you learn, not fixing messy reports or searching for solutions.)

For late submissions you will lose one 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.

Important Programming Reminder

As you are working on your code, **commit and push changes frequently.** The commit history should show a natural progression of your code as you add features and fix bugs. Committing large, complete chunks of code may result in significant point loss. (You may include existing code for standard tasks like adding files to the file cache or creating a buffered file reader, but then the corresponding commit comment must indicate the source.) If you are not sure, better commit too often than not often enough. Here are some rules of thumb:

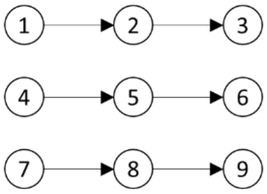
- Work for about 1 hour on your code, then commit your changes and include a comment explaining concisely what changed. Early on, when you produce a lot of initial code quickly, you may need to commit more frequently. Later, when you maybe spend 50 minutes on tracking down a single bug that results in a 1-line change in your code, you may commit less frequently. Use common sense: you want your commit history to tell a sufficiently detailed story about how you worked on your code, but you do not need to commit every single line you modify.
- To commit and periodically push your changes, use your git client, e.g., your IDE. There are explicit commit and push operations, and the git client will ask you for a comment summarizing the changes you made. You must execute those commit/push commands—simply saving your code from your IDE will not automatically commit it. The commit comment is a short explanation, e.g., "added reduce function for word count" or "fixed memory error in RDD join."

Repeat this work-commit-push cycle until you are done. This way we see your progress in terms of the intermediate commits. If you simply commit a complete program in a single step, it looks like you just got the solution from somebody else, and this may trigger an academic integrity investigation.

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, the first part of this assignment also explores the subtleties of Spark's actions and transformations, and how they affect lazy evaluation and job submission. We will work with synthetic data to simplify the program a little and to make it easier to create inputs of different sizes. 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 operations needed to produce the desired result. Stated differently, transformations only define the lineage of a result, while actions force the execution of that lineage. What will happen when an iterative program performs both actions and transformations in a loop? What goes into the lineage after 1, 2, or more 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 (p1, p2), each encoding a link from some page p1 to another page p2. **Ranks** stores pairs (p, pr), encoding the PageRank pr for each page p. To fill these tables with data, create a graph that consists of k linear chains, each with k pages. Number the pages from 1 to k^2, chain by chain. Here k is a program parameter to control problem size. The figure shows an example for k=3. Notice that the last page in each linear chain is a dangling page. We will use the single-dummy-page approach to deal with dangling pages. This means that your program also must create a single dummy page—let's give it the number 0 (zero)—and add it to Ranks. Add an edge (d, 0) for each dangling page

d. Set the initial PR value for each of the k^2 real pages in Ranks to $1/k^2$; set the initial PR value of the dummy page 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 like the example program, but follow these requirements and suggestions:

- 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.
- Make sure you add dummy page 0 to Ranks and the k dummy edges to Graph.
- Initialize each PR value in Ranks to $1/k^2$, except for page 0, whose initial PR value should be zero. Be careful when you look at the example PR program in the Spark distribution. It sets initial PR values to 1.0, and its PR computation adds 0.15 instead of 0.15/#pages for the random jump probability. Intuitively, they multiply each PR value by #pages. While that is a valid approach, it is not allowed for this assignment.
- Make sure that Graph and Ranks have the same Partitioner to avoid shuffling for the join.
- 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 page 0, use the *lookup* method of pair RDD. Then re-distribute this mass evenly over all *real* pages.
- When debugging your program, see if the PR values add up to 1 after each iteration. Small variations are expected, especially for large graphs, due to numerical precision issues. However, if the PR sum significantly deviates from 1, this may indicate a bug in your program.
- Add a statement right after the end of the for-loop (i.e., outside the 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 and look at the lineage for Ranks.
2. Change the loop condition so that exactly 2 iterations are performed and look at the lineage for Ranks after those 2 iterations. Did it change?

The lineage describes the job needed to compute the result of the action that triggered it. Since pair RDD's lookup method is an **action**, a new job is executed in each iteration of the loop. Can you describe in your own words what the job triggered in the i-th iteration computes?

An interesting aspect of Spark, and a reason for its high performance, is that it can reduce computation cost by **re-using** previously computed results. This means that in practice, only a part of the lineage may get executed. To understand this better, consider the following simple example program:

```
1. val myRDD1 = some_expensive_transformations_on_some_big_input()
2. myRDD1.collect()
3. val myRDD2 = myRDD1.some_more_transformations()
4. myRDD2.collect()
```

This program executes 2 jobs. The first is triggered by line 2 and it computes all steps defined by the corresponding transformations in the lineage of myRDD1. The next job is triggered by line 4. Since myRDD2 depends on myRDD1, all of myRDD1's lineage is also included in the lineage of myRDD2. But will Spark execute the entire lineage? What if myRDD1 was still available from the earlier job triggered by line 2? Then it would be more efficient for Spark to simply re-use the existing copy of myRDD1 and only apply the additional transformations to it!

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 would not completely fit into the combined memory of all machines in the cluster).

Bonus challenge: For an optional 5-point bonus (final score cannot exceed 100), run your PageRank program on the Twitter followership data. If you took shortcuts for the synthetic data, e.g., by exploiting that no page has more than 1 outgoing link, you need to appropriately generalize your program to work correctly on the Twitter 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 the 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 an input file 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 relatively small input, make sure that your program creates **at least 20 Map tasks**. You can use NLineInputFormat to achieve this.

Report

Write a brief report about your findings, answering the following questions:

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, e.g., 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 other graphs.)
2. [10 points] Show the link to the source code for this program in your Github Classroom repository.
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.
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. **Describe in your own words what the job triggered in iteration 3 computes**.
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 would 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.

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 from later iterations. 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?**

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 re-use 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?**

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.
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.

Important Notes

Choose carefully where you host the log and output files. You can copy them to the corresponding Github Classroom repo, but Github has a size limit for version-managed files and things can get tricky. If you leave the files on S3, make sure the TAs and instructor have access to them and keep in mind that Amazon charges per GB of data on S3. So, possibly the easiest solution would be to use Google Drive, Dropbox or similar. Again, make sure the TAs and instructor (but nobody else) can access the corresponding directories and files.

Check that the log file is not truncated—there might be multiple pieces for large log files!

The **submission time** of your homework is the *latest timestamp of any of the deliverables included*. For the PDF it is the time reported by Gradescope; for the files on Github it is the time the files were pushed to Github, according to Github. If you want to keep things simple, do the following:

1. Push/copy all requested files to github and make sure everything is there. (Optional: Create a version number for this snapshot of your repository.)
2. Submit the report on Gradescope. Open the submitted file to verify everything is okay.
3. Do not push any more changes to the files for this HW on Github.

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

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.