CS 6240: Assignment 1

Goals: Set up your environment for developing and running Hadoop MapReduce and Spark Scala programs. Test your setup by writing and executing a Word-Count inspired program.

This homework is a special **collaborative assignment**. You can ask anybody in class for help and in your report, you can acknowledge the person who helped you the most. (Students with most acknowledgements may receive participation bonus credit at the instructor's discretion.)

Please submit your solution as a *single PDF file* on Gradescope (see Gradescope link on the left in Canvas) by the due date and time shown there. During the submission process, 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 two points per hour after the deadline. This HW is worth 100 points and accounts for $1/3$ 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 included in your starter project. You can use the example Makefile and simply customize the variable settings in the beginning. For this Makefile to work on your machine, you need **Maven** and make sure that the Maven plugins and dependencies in the **pom.xml** file are correct. (If you are familiar with other major dependency-management software, you may use that instead. However, we can only provide examples for Maven.) To use the Makefile for executing your job in the cloud, you also need to set up the AWS CLI on your machine. More detailed instructions are included below.

As with all software projects, you must include a **README** file, like the one provided in the starter project, briefly describing all the steps necessary to build and execute both the local/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).

We allow and encourage you to share with everybody in class your configuration and setup files and any setup info like settings of environment variables, compatible software version numbers, etc. with everybody in class, including your Makefile, pom.xml, and README.

Set Up Github and Your IDE

Find the **Github Classroom** repositories we created for you in the Canvas version of the assignment. They contain the Word Count example, which you will modify. In addition to the source code, you need to customize the Makefile and update the README and pom.xml files. M**ake sure that you select software versions that are compatible with one of the configurations Amazon offers for EMR Hadoop MapReduce and Spark jobs**. We strongly recommend using an IDE such as Eclipse or VS Code for code development and interacting with Github (to pull code from and push code to the repository).

As you are working on your code, **commit changes regularly.** This may seem unnecessary for this assignment where you will make only small modifications in the Word Count code, but we want you to practice committing and pushing code so that you are ready for future assignments. As a rule of thumb for more programming-intensive assignments, the "delta" between consecutive snapshots of your source code should be equivalent to about **15 minutes' worth of coding**. Committing large, complete chunks of code (other than code provided with a homework, like the demo examples) that look like you just copied from someone else will result in point loss and may trigger an academic integrity investigation. If you are new to git, follow this procedure:

- Work for about 15 min on your code.
- Then, instead of simply saving the new version, use your git client to also **commit** your changes. If you work with an IDE, committing is practically as easy as a simple file save.
- Notice that 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. For the comment, just add a short explanation, e.g., "added reduce function for word count."
- Repeat this work-commit(-push) cycle until you are done. This way we see your progress in terms of the intermediate commits.

AWS Account and Development Environment Setup

Amazon generously offers documentation and some amount of credit (typically $100 per student) for educational purposes. Unfortunately, they have been changing their setup frequently. This semester, for you to receive the free credit, the instructor will sign you up for an AWS Learner Lab course. You should at some point during the first week of this assignment receive an invite from AWS. Follow its instructions to get your free credit.

If in doubt, follow the latest documentation on AWS and post your questions on the discussion board. If you do not want to use the Amazon cloud, you can alternatively work with any equivalent cloud environment, e.g., those hosted by other companies (Google, Microsoft, IBM), Northeastern's Discovery cluster, or the Mass Open Cloud. However, we are not able to provide custom instructions or support for those environments.

Fallback option if you absolutely cannot get your program to run in a cloud environment:

- Do not give up. Ask other students and our TAs for help. Some students struggled initially, but ultimately everybody managed to run their programs in the Amazon cloud.
- If you have a working cloud setup and your situation is "my program runs correctly locally, but it does not work/is incorrect in the cloud," then you have a *bug* in your code that you should fix. Most likely there is a hard-coded file path or you are trying to use a global variable, e.g., a static variable for a Java class, which works by accident when the program runs in a single JVM but will not work when the objects of that class are distributed over different JVMs.
- If despite all efforts, you cannot get the cloud setup to work, then you have this fallback option: Write the program and perform all homework tasks as requested, except for the run-it-in-the-cloud parts where you run it on your laptop instead. If you do so, your homework will be graded out of 90 points, instead of out of 100. By taking advantage of early-submission-bonus options, you may still be able to reach 100 points.

We recommend using **Linux** for MapReduce and Spark development. **MacOS** has worked fine as well, and the Windows Subsystem for Linux on Windows may be another viable option. Or you can run Linux in a virtual machine, but note that Macbooks with the M1 chip had issues with virtualization. We tested Oracle VirtualBox and VMware Workstation Player on Windows and they worked fine: install the virtual machine player (free) and create a virtual machine running Linux, e.g., Ubuntu (free). If you are using a virtual machine, then you need to apply the setup steps to the virtual machine.

On Amazon AWS, you can run your jobs using EMR or plain EC2. With EMR, it is easier to set up a virtual cluster for the assignments in this course. EC2 offers slightly lower hourly rates per machine instance. We generally recommend EMR for this course and will provide setup help and instructions only for EMR. However, you are welcome to work with plain EC2—but you need to deal with it on your own. (There are tutorials on the Web that should make it relatively easy for advanced users to fire up a cluster of MapReduce or Spark nodes on EC2.)

Check the versions of Hadoop MapReduce and Spark available on EMR. **To avoid incompatibility issues, make sure your local development environment matches one of them, at least in terms of the main version number(s)**, e.g., all MapReduce 2.8.* should be compatible with each other. Instead of the software setup mentioned in the demo README files, you may use Docker containers, pre-packaged environments or ready-made virtual machines. Whatever you choose, the goal is for you to be able to develop, debug, and test your MapReduce (Java) and Spark (Scala) programs locally, then run them on EMR. Remember that to run from the command line, you must edit the variables in the Makefile as described in the demo README. **Instructor and TAs must be able to run your project following the instructions in your README—only customizing the variable settings at the top of the Makefile.**

Hint for those new to Maven and dependency management software in general: In your IDE, create a Maven project. This makes it easy to build "fat jars", which recursively include dependent jars used in

your program. There are many online tutorials for installing Maven and for creating Maven projects via archetypes. These projects can be imported into your IDE or built from a shell. Sometimes you may need to modify the provided pom.xml file slightly. Use the discussion board to get help when you are stuck with Maven; or talk to our friendly TAs. It is allowed to post your pom.xml in a discussion and to copy someone else's pom.xml (or equivalent for other dependency managers).

Write a Grouping-and-Aggregation Program

Look carefully at the Word Count program in the two demos. It extracts all "words" from a line of text, then groups by the word and aggregates all counts associated with the same word. Your task is to write a very similar program, but for the Twitter-follower dataset that is included in this assignment.

Before designing your algorithm and program, become familiar with the data by consulting the readme file and looking at a few lines of the data. Make sure you understand the direction of the follower edges. Then write a program that takes *edges.csv* as input and **counts the number of incoming edges for each user ID that is divisible by 100**. Output this result as plain text, each line containing userID and follower count, e.g.:

userID1, number_of_followers_this_user_has
userID2, number_of_followers_this_user_has

The output may be partitioned into multiple files. It does not need to be sorted, and each line may contain other formatting characters, e.g., "(userID1, number_of_followers_this_user_has)," which may be added by default by MapReduce or Spark output writers.

Hint: The structure of the program should be similar to Word Count. You may start with the provided demos and modify them. Instead of words, group by the appropriate user ID of an edge.

Start with the MapReduce program and get it to run in the IDE on your development machine. Notice that you will need to provide an input directory and a path to an output directory. Once the program runs fine, look at it closely and see what Map and Reduce are doing. Use the debugging perspective, set breakpoints in the Map and Reduce functions, then experiment by stepping through the code to see how it is processing the input file. Make sure you work with a small data sample. Then work on the Spark Scala program and find out what the RDD *toDebugString* method does. Also look at the *explain* method of DataSet in the Spark Scala API.

Optional bonus challenge 1 (2 extra points): Modify your MapReduce program so that it outputs the result in order of the number of followers, showing the user with most followers at the top.

Quick note about correct sorting if your job produces multiple output files named part-r-* or similar: Assume for simplicity that the program outputs the numbers 1, 2,…, 9 in 3 different output files like part-r-0000, part-r-0001, and part-r-0002 and the goal is to sort them in increasing order. An output where these files contain (3,5,7), (1,4,9), and (2,6,8) is not correct because each part-r-* file is only locally sorted. A correct output would be (1,2,3), (4,5,6), and (7,8,9). Here the individual output files are

locally sorted, but they also represent a correct global order. You do not need to manually concatenate the individual files.

Optional bonus challenge 2 (2 extra points): By counting incoming edges in edges.csv, we only obtain follower counts for users *with at least 1 follower*. Modify your MapReduce program so that it also outputs users with zero followers (again, only for user IDs that are divisible by 100). Think about how to find these user IDs in *nodes.csv*, making minimal changes to your program. (There is an elegant solution if your MapReduce job reads both input files a simple text files, in the map function using the content of a line of text to determine if that line comes from *edges.csv* or from *nodes.csv*.)

The bonus challenges are intended for students who already completed all required tasks. Including bonus points, the final HW score cannot exceed 100 points.

WARNING: Leaving large data on S3 and using EMR costs real money. Read carefully what Amazon charges and estimate how much you would pay per hour of computation time before running a job. Use the smallest/cheapest available general-purpose machine instances. And remember to test your code as much as possible on your development machine.

Report

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

1. [0 points] Did any student in class help you in a major way with the setup process? As a rule of thumb, this would be somebody who spent *an hour or more* with you, e.g., helping you install Maven and Scala, figuring out why the Makefile did not work for you and so on. You can credit this student here (*at most one person can be credited—if multiple students helped you, only mention the one you think helped you the most*), so that this can be taken into account for that person's course-participation credit.
2. [28 points] Show the links to your repository with source code etc on Github Classroom.
    a. Show the link to your MapReduce repository.
    b. Show the link to your Spark repository.
3. [20 points] Show the *pseudo-code* for your Twitter-follower-count program in MapReduce. Look in the online modules and your lecture notes for examples like the Word Count program. Pseudo-code captures the essence of the algorithm and avoids wordy syntax. Do *not* just copy-and-paste source code!
4. [10 points] Briefly discuss the main idea of your solution. Example: "My program reads the input line by line. The map function parses a line to extract … For each … it outputs … These … are grouped by … and then the reduce function computes … for each group."
5. [20 points] Show the pseudo-code for your Twitter-follower-count 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.
6. [10 points] For the Spark program, report the lineage of the final result. If you use (pair) RDDs, let's assume you called the RDD containing the final result *countsRDD*. (This is the one that you

output in the end, using countsRDD.saveAsTextFile(args(1)).) Right before the save-statement, add a call to countsRDD.toDebugString and make sure this information is written to your log file, e.g., using *logger.info(countsRDD.toDebugString)*. Copy-and-paste the lines produced by this toDebugString call to answer this question. If you use DataSet, then do the same for the *explain* function of DataSet instead.

7. [5 points] Run your MapReduce and your Spark program on AWS on the full Twitter edges dataset. Use 6 *cheap* machine instances (1 master and 5 workers). Look for the cheapest machine instance you can select on EMR. In the past these were usually general-purpose machines of type m[?].[*], e.g., m1.small or m5.large. (For some instance types, possibly those from a previous generation, you may need to submit a subnet-id when creating the cluster.) After successful execution, look through the log files (syslog, stderr) it generated to find the following information:
    a. Report the running time of your MapReduce program. To determine it, look for lines in the log file that indicate when the actual MapReduce computation started and finished.
    b. Report the amount of data transferred (in bytes and in records) to the Mappers.
    c. Report the amount of data transferred (in bytes and in records) from Mappers to Reducers.
    d. Report the amount of data transferred (in bytes and in records) from Reducers to output.
    e. Report the running time of your Spark program.
8. [3 points] To determine the maximum degree of parallelism possible for your program, we need to understand how many Map and Reduce tasks are created (or possible). To this end, analyze your program's scalability potential by answering the following questions: (a) Which phase will most likely be the bottleneck: Map or Reduce? Briefly justify your response. (b) How many Map tasks did MapReduce create for the Twitter data? Briefly justify your response by exploring the log. (c) How many reduce() function calls does your program make for the Twitter data? Briefly justify your response.
9. [4 points] Show the following 4 links:
    a. A link to your syslog file created by the run of your MapReduce program for which you reported the running time and data transfer numbers.
    b. A link to your output file (or the directory containing the output files if there are multiple) created by the run of your MapReduce program for which you reported the running time and data transfer numbers.
    c. A link to your stderr file created by the run of your Spark program for which you reported the running time.
    d. A link to your output file (or the directory containing the output files if there are multiple) created by the run of your Spark program for which you reported the running time.
10. [2 optional points] If you attempted bonus challenge 1, do the following:
    a. Briefly explain how you achieved sorting. Did you need an additional MapReduce job?

b. Replace the above links for log and output files by the corresponding links based on the execution of your modified program that performs sorting. (You do *not* need to provide log and output files for both the sorted and unsorted versions.)

11. [2 optional points] If you attempted bonus challenge 2, do the following:
    a. Briefly explain how you modified the program to find the users with zero followers. Did you need an additional MapReduce job?
    b. Replace the above links for log and output files by the corresponding links based on the execution of your modified program. (You do *not* need to provide log and output files for both versions.)

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 Northeastern's OneDrive space, 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.