CS 6240: Assignment 3

Goals: (1) Gain deeper understanding of combining in Spark. (2) Implement non-trivial joins in 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 Blackboard 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 Blackboard. (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.

Combining in Spark (First 2/3 of Week 1)

The first part of this assignment compares different implementations of the Twitter-follower counting problem in Spark. We only work with the edges.csv data from http://socialcomputing.asu.edu/datasets/Twitter. Like in the first assignment, your programs should output the number of followers for each user who has at least one follower, returning output formatted with each user and follower count in a different line:

(userID1, number_of_followers_this_user_has)
(userID2, number_of_followers_this_user_has)

This time we only focus on Spark, but you need to implement different programs:

RDD-G: This program only uses RDD and pair RDD, but not DataSet or DataFrame. The grouping and aggregation step must be implemented using groupByKey, followed by the corresponding aggregate function.

RDD-R: This program only uses RDD and pair RDD, but not DataSet or DataFrame. The grouping and aggregation step must be implemented using reduceByKey.

RDD-F: This program only uses RDD and pair RDD, but not DataSet or DataFrame. The grouping and aggregation step must be implemented using foldByKey.

RDD-A: This program only uses RDD and pair RDD, but not DataSet or DataFrame. The grouping and aggregation step must be implemented using aggregateByKey.

DSET: The grouping and aggregation step must be implemented using DataSet, with groupBy on the appropriate column, followed by the corresponding aggregate function.

Joins in Spark (Last 1/3 of Week 1, All Week2)

Solve the Twitter-follower *triangle*-counting problem from homework 2, but this time in Spark Scala. Write four different versions of the program:

1. RS-R implements the equivalent of Reduce-side join (hash+shuffle) using RDD and pair RDD only.
2. RS-D implements the equivalent of Reduce-side join (hash+shuffle) using DataSet or DataFrame only.
3. Rep-R implements the equivalent of Replicated join (partition+broadcast) using RDD and pair RDD only.
4. Rep-D implements the equivalent of Replicated join (partition+broadcast) using DataSet or DataFrame only.

Hints and requirements:

- For the DataSet/DataFrame programs, it is allowed to load the data initially as an RDD, but then the actual join has to be applied to DataSets.
- Do not use SparkSession.sql(someSQLquery) to implement the join. Instead, use functions such as join, joinWith, map, filter, flatMap etc.
- For functions such as join and joinWith, find out if they implement hash+shuffle and/or partition+broadcast. If they do, just use them accordingly. Otherwise implement the corresponding join "from scratch."
  - For instance, if partition+broadcast is not directly supported, then you have to implement it in user code yourself. Here you can use the SparkContext.broadcast feature. Check carefully what type of data can be broadcast (RDD? DataSet? Collection?), how to encode your data correctly, and how to control size limits. Instead of join, you may have to use map or mapPartitions with a lookup on the collection that was broadcast. Take a look at the discussion in the module. (We just uploaded a slightly updated version with a new example for partition+broadcast with RDDs.)
- Like for the MapReduce program, you may use the MAX-filter idea to reduce data size by eliminating all input records for users with large IDs, if you believe you cannot resolve excessive memory/storage consumption or running time in any other way.

- It is perfectly acceptable to search for example programs in textbooks and on the Web to get inspiration and resolve syntax issues, as long as you cite them in report and source code. E.g., look at the Spark textbook join chapter we reference in the module.

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 CCIS Github repository for this homework. Make sure TAs and instructor have access to it. (4 points)**

Combining in Spark (20 points total)

Show the pseudo-code for each of the Twitter-follower count programs in Spark Scala. Since many Scala functions are similar to pseudo-code, you may copy-and-paste them here whenever appropriate. (10 points)

Using log files from successful runs and Scala functions such as toDebugString() and explain(), find out which of the different programs performs aggregation before data is shuffled, i.e., the equivalent of MapReduce's in-Mapper combining.

For each of the four RDD-based programs, report the information returned by toDebugString(). (4 points)

For the DataSet-based program, report the **logical** and **physical** plans returned by explain(). (2 points)

Based on this information, state clearly which of the programs performs aggregation before shuffling, and which does not. (4 points)

Note: You are *not* required to run these programs on AWS.

Join Implementation (48 points total)

Show the pseudo-code for all four Spark Scala programs (RS-R, RS-D, Rep-R, Rep-D), including Max-filter functionality if you used it. Since many Scala functions are similar to pseudo-code, you may copy-and-paste them here whenever appropriate. (20 points)

Work with the same MAX threshold as the corresponding MapReduce programs in homework 2. Run each program with the corresponding MAX value on AWS on the full Twitter edges dataset, using the following two configurations:

- 6 m4.large machines (1 master and 5 workers)
- 11 m4.large machines (1 master and 10 workers)

As usually, if you have difficulty with the *.large machines, try *.xlarge instead. Report your results in a table like this: (20 points)

Configuration	**Small Cluster Result**	**Large Cluster Result**
RS-R, MAX = ???	Running time: ???, Triangle count: ???	Running time: ???, Triangle count: ???
RS-D, MAX = ???	Running time: ???, Triangle count: ???	Running time: ???, Triangle count: ???
Rep-R, MAX = ???	Running time: ???, Triangle count: ???	Running time: ???, Triangle count: ???
Rep-R, MAX = ???	Running time: ???, Triangle count: ???	Running time: ???, Triangle count: ???

Copy to your Github the syslog files of the runs you are reporting the above measurements for. Check that the log is not truncated—there might be multiple pieces for large log files! Include a link to each log file/directory (4 links total) in the report. Similarly, copy the output produced (all parts of it) to your Github and include the links to the output directories (4 links total) in the report. (8 points)

Note: instead of Github, you can alternatively make the log and output files available on a cloud storage service such as S3, Google Drive or Dropbox. If you do so, make sure TAs and instructor (1) can clearly see the upload time of the files on that service and (2) are able to access those files.

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 Blackboard; for the files on Github it is the time the files were pushed to Github, according to Github. (The same applies if you leave log and output files on a cloud storage service. To avoid clutter, we will simply say "Github" below.) We recommend the following approach:

1. Push all files to Github and make sure everything is there (see deliverables below).
2. Submit the report on Blackboard. 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 (**not** txt, doc, docx, tex etc!) on Blackboard. 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 Blackboard; 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 **CCIS Github** repository:

2. Log file for each job you used to fill in the corresponding running time cells in the above table. (8 syslog/stderr or similar, 4 points)
3. All output files produced by those same successful runs on AWS (8 sets of part-r-… or similar files). (4 points)
4. The Spark Scala project, including source code, build scripts etc. (20 points)

Note: 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.

IMPORTANT: Please ensure that your code is properly documented. In particular, 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.