CS 6240: Assignment 3

Goals: (1) Gain deeper understanding of grouping and aggregation in Spark. (2) Implement 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 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 to github.** 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.

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. Your programs should output the number of followers for each user **who has at least one follower and whose ID is divisible by 100**, 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)

You need to implement 5 different programs in Spark (all very short):

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

RDD-R: This program groups and aggregates an RDD or pair RDD, not a DataSet or DataFrame. The grouping and aggregation step must be implemented using reduceByKey.

RDD-F: This program groups and aggregates an RDD or pair RDD, not a DataSet or DataFrame. The grouping and aggregation step must be implemented using foldByKey.

RDD-A: This program groups and aggregates an RDD or pair RDD, not a DataSet or DataFrame. The grouping and aggregation step must be implemented using aggregateByKey.

DSET: The grouping and aggregation step must be implemented using DataSet or DataFrame, with groupBy on the appropriate column, followed by the corresponding aggregate function. (You may directly instantiate the DataSet or DataFrame from the input file, or you can load the data into an RDD and then convert it to a DataSet or DataFrame. Either way, the grouping and aggregation here must be applied to the DataSet/DataFrame.)

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

Solve the Twitter-follower *triangle*-counting problem from Homework 2, but this time in Spark (Scala, Python, or Java should all work well). Write 4 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 must be applied to DataSets/DataFrames only.
- Do not use SparkSession.sql() 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, use them accordingly.
    - For hash+shuffle, note that any strategy that groups the input based on join key(s) during the shuffle phase is acceptable here, no matter if grouping is implemented by hashing or by sorting. In particular, on the "Reducer" side, the local join may be implemented using hash-based matching or sort-merge.
    - DataSet/DataFrame: You may need to explore optimizer hints and broadcast-size-threshold settings as discussed in class. If despite your best efforts the optimizer keeps choosing only one, but never the other join strategy, then report the settings you tried and explain which strategy was chosen by the optimizer and **how you found out**.
    - (Pair) RDD: Last time we checked, the join was implemented using hash+shuffle, but not partition+broadcast. Verify if this still holds true. If so, you need to implement partition+broadcast in user code as shown in class.
        - When using broadcast(), check that the data structure you broadcast is properly representing the edge-RDD information. For instance, for a hashmap or similar, can the same key appear with different values? If not, then you need to find a workaround, e.g., encode (k1, v1), (k1, v2), (k1, v3) as (k1, (v1, v2, v3)).
- 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 your report and source code. E.g., look at the Spark textbook join chapter we reference in the module.

We strongly encourage collaboration via the online discussion forum for all Spark syntax issues, especially questions like "which data structure should I use to broadcast the edges list," "what is the syntax to look up values associated with a specific key in that data structure" and so on.

Report

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

1. [10 points] Show the pseudo-code for all 5 Twitter-follower-count programs (RDD-G, RDD-R, RDD-F, RDD-A, DSET) 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.
2. [10 points] Show the link(s) to the source code for these programs in your Github Classroom repository.

3. [10 points] Run these programs *locally* (not on AWS) and determine if Spark performs aggregation before data is shuffled, i.e., the equivalent of MapReduce's Combiner. Look for evidence that supports your answer, e.g., by using toDebugString() and explain(), looking at the log files, and consulting the Spark documentation. For each of the 5 programs, (1) state clearly if it aggregates before shuffling (like a Combiner) and (2) present evidence to support your answer.
4. [30 points] Show the pseudo-code for the 4 triangle-counting programs, including MAX-filter functionality. 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.
    a. [10 points] RS-R
    b. [5 points] RS-D
    c. [10 points] Rep-R
    d. [5 points] Rep-D
5. [16 points] Show the link(s) to the source code for these programs in your Github Classroom repository.
6. [8 points] Run each triangle-counting program on EMR using 1 master and 4 worker nodes, using the same machine type and MAX setting as for the corresponding MapReduce program (RS-R and RS-D like RS-join in HW 2; Rep-R and Rep-D like Rep-join in HW 2). If a program runs faster than 15 min for the same settings as HW 2, simply report the faster time. If a program runs more than twice as long as the corresponding MapReduce version, you may choose a lower MAX setting. If the Spark version crashes with out-of-memory error for the same settings where the MapReduce program worked fine, explore Spark parameters to control memory size (container size, heap size), but do not use larger machine instances. If you still cannot resolve the memory issue, report the settings you tried and fix the problem by using a lower MAX. For each of the 4 programs (RS-R, RS-D, Rep-R, Rep-D), report machine type, MAX value, running time, and triangle count. For comparison, also report machine type, MAX, running time, and triangle count for the corresponding join versions from HW 2. The machine type must be the same. If you changed MAX, briefly explain why (see discussion above).
7. [4 points] Show the link to the stderr file and the file that contains the triangle count for the run that produced the above running-time numbers for each of the 4 programs (RS-R, RS-D, Rep-R, Rep-D).
8. [8 points] Run the triangle-counting programs on the larger cluster with 1 master and 8 workers (or 7, if 8 caused quota issues on AWS Academy), following the same instructions (same configuration as for HW 2; change MAX only if necessary). For each of the 4 programs (RS-R, RS-D, Rep-R, Rep-D), report machine type, MAX, running time, and triangle count. For comparison, also report machine type, MAX, running time, and triangle count for the corresponding join versions from HW 2. The machine type must be the same. If you changed MAX, briefly explain why (see discussion above).
9. [4 points] Show the link to the stderr file and the file that contains the triangle count for the run that produced the above running-time numbers for each of the 4 programs (RS-R, RS-D, Rep-R, Rep-D).

Important Notes

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.