CS 6240: Assignment 2

Goals: (1) Gain deeper understanding of combining in Spark. (2) Implement non-trivial joins in MapReduce and Spark, which require careful analysis of (intermediate) result sizes to determine feasibility of possible solutions.

This homework is to be completed individually (i.e., no teams). You have to 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., *alone* and *cloud*) as the one Joe 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. Of course, the earlier you work on this, the better.

Combining in Spark (First Half of Week 1)

The first part of this assignment takes a closer look at the Twitter follower counting problem. For the last assignment, we did not require any specific implementation. Now you will compare several possible solutions. We only work with the edges data from http://socialcomputing.asu.edu/datasets/Twitter Like in the previous assignment, your programs should output the number of followers for each user, returning output formatted like this, 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: This program has to convert the input immediately into a DataSet. Then only DataSet operations can be applied. The grouping and aggregation step must be implemented using groupBy on the appropriate column, followed by the corresponding aggregate function.

Joins in MapReduce and Spark

In many classes, homework assignments are designed so that they can be solved in the "obvious" way as taught in the course. Unfortunately, that is not how the real world tends to function. We therefore decided to give you a slightly more realistic challenge. Imagine, you have a customer who is interested in finding out, if it is common on Twitter for a user X to follow another user Y, who follows a user Z, who in turn follows X. Let's call this pattern a *social amplifier triangle*, or simply a *triangle*.

Formally, we want to count the number of distinct triangles in the Twitter graph. A triangle (X, Y, Z) is a triple of user IDs, such that there exist three edges (X, Y), (Y, Z), and (Z, X). Clearly, if (X, Y, Z) is a triangle, then so are (Y, Z, X) and (Z, X, Y). **Make sure that your program does not triple-count the same triangle**.

Your customer would like to get the most accurate triangle count possible, ideally the *exact* number. Unfortunately, as it often happens with Big Data in the real world, you initially do not know if this problem can be solved on a small AWS cluster. This means that you need to perform a careful analysis to determine (1) if there is any hope for solving the problem exactly, and (2) what to do if an exact solution is not feasible.

Careful Problem Analysis (Second Half of Week 1)

We will solve the triangle counting problem with the join operator. For simplicity, assume our input is a table called "Edges" with columns named "from" and "to," i.e., tuple (X, Y) represents the followership from X to Y. The conceptual solution for triangle counting is as follows:

1. Join the Edges table with itself to find paths of length 2. Two tuples (X, Y) and (A, B) join if and only if Y=A. In SQL notation, we compute SELECT E1.from, E1.to, E2.to FROM Edges AS E1, Edges AS E2 WHERE E1.to = E2.from. Let us call this intermediate result "Path2" and assume it has schema (start, mid, end) for each length-2 path.
2. Next, we join Path2 with Edges to "close" the triangle. Essentially this step checks for each path (X, Y, Z), if edge (Z, X) is in Edges. In SQL notation, we compute SELECT P.start, P.mid, P.end FROM Path2 AS P, Edges AS E WHERE P.start = E.to AND P.end = E.from.

3. Count the number of result tuples and divide this number by 3 to compensate for triple-counting each triangle.

In the following discussion, we will focus on the MapReduce solution. The Spark approach is analogous, but requires looking for the corresponding Scala functions, instead of writing the Java code yourself. Since all join conditions are equalities, we can use any equi-join algorithm. The lecture material introduces Reduce-side join and Replicated (or Map-only) join. Make sure you fully understand these algorithms and their differences.

When analyzing a given problem and possible solutions, start by determining the size of input, data shuffled, and output for each MapReduce job. (In Spark, we would analogously analyze each stage of the computation.) Fill out the cells in the table below.

	RS join input	**RS join shuffled**	**RS join output**	**Rep join input**	**Rep join file cache**	**Rep join output**
Step 1	Total cardinality and volume of input	Total cardinality and volume of data sent from Mappers to Reducers	Total cardinality and volume of output	Total cardinality and volume of input	Total cardinality and volume of data broadcast to all machines	Total cardinality and volume of output
Step 2	…	…	…	…	…	…

In the table, RS stands for Reduce-side, and Rep for Replicated. Note that Replicated join is Map-only, i.e., does not shuffle data. On the other hand, it copies data using the file cache. For each table cell, try to find both the cardinality (number of records) and the volume (in bytes—use an educated guess if you do not know the exact per-tuple size).

Some entries are obvious, e.g., the initial input cardinality and volume. Some can be derived easily, e.g., the cardinality and volume for the data shuffled by Reduce-side join in step 1. Others can be tricky, e.g., the output cardinality and volume for step 1, which is also the input for step 2. *How can we determine the unknown statistics, or at least estimate them reasonably well?* This is a common and difficult challenge that has been the subject of numerous papers on database query optimizers. For this homework, we recommend the following approach (you may explore other ideas in addition to the ones outlined below):

1. Estimating the number of tuples in Path2: The simplest solution would be to simply execute the join from step 1. Unfortunately, this defeats the purpose of determining if that execution is feasible at all. What if there are hundreds of billions of length-2 paths? Just writing them to HDFS could take too long. This means that we need a cheaper method. Here we can exploit that we are interested in the *count statistics*, not yet the actual join. Try to exploit the following observation: Consider some user Y. How many length-2 paths will go through Y? If Y has m

incoming edges—from followers $x_1, \ldots, x_m$—and n outgoing edges—to followed users $z_1, \ldots, z_n$—then there exactly m·n length-2 paths through Y—one for each combination (x_i, Y, z_j). This implies that if we can efficiently count the incoming and outgoing edges for each user, then we can easily determine the desired count statistics for intermediate result Path2.

2. Estimating the total number of triangles: This is more challenging, because we need to know more about the actual users in the rows of Path2—not just the count statistics—to determine how many of them join with an edge. Fortunately, we know the following: given a length-2 path $X \rightarrow Y \rightarrow Z$, there either is an edge $Z \rightarrow X$ completing triangle (X, Y, Z), or there is not. This means the final result cannot be larger than the intermediate result in Path2. Stated differently, we at least know an upper bound on the final output and can focus on the intermediate result as the bottleneck of the computation. Hence it is acceptable to simply put the cardinality and volume estimate of Path2 as the estimate for the output of step 2.

Sometimes, despite your best efforts exploiting problem insights and computation shortcuts as discussed above, you might still not be able to get the exact count statistics for Path2. Then you will have to resort to a universal fail-safe strategy for Big Data: data reduction and approximation. This usually means that you will estimate the count statistics from a smaller data **sample**. Unfortunately, sampling from graphs is tricky, because it often destroys the structures we are looking for. For instance, consider a user Y with 1 incoming and 1000 outgoing edges. If we randomly sample 1 in 10 edges, then it is very likely that the incoming edge will not be sampled, while 100 outgoing edges are sampled. This results in output size underestimation, because for Y we estimate zero results, even though it actually contributes 1000 paths of length 2. (Random sampling on the input is known to be problematic for non-trivial problems such as joins. See [S. Chaudhuri, R. Motwani, and V. Narasayya. On Random Sampling over Joins. In Proc. ACM SIGMOD, pages 263-274, 1999] available at https://scholar.google.com/scholar?cluster=11714299693819318683&hl=en&as_sdt=0,22.)

You may voluntarily try to implement a distributed version of the Chaudhuri et al. algorithm, but first work with the following simpler approach: Given a threshold MAX, remove all edges that contain a user ID greater than or equal to MAX. In SQL notation, this corresponds to SELECT * FROM Edges E WHERE E.from < MAX AND E.to < MAX. Then perform the counting, or even the full triangle algorithm, on that reduced dataset, instead of the full input. Notice that this approach preserves all edges between users with small IDs. Hence statistics for those users will be accurate; but we miss all counts involving users with larger IDs.

Try to set MAX as high as possible, to get the most accurate result. You may need to start with fairly small MAX values to get an idea how quickly computation cost and result size for that MAX value grow. Look at several data points of the type (MAX threshold, number of output tuples for step 1) and see if you can find a trend. Then try to estimate the result size for MAX = infinity based on that trend. Note that all this can be done on your local machine.

With a reasonably accurate estimate of the cardinality and volume of Path2 in hand, complete the entries for step 2 in the table above.

Program Design and Implementation: MapReduce (MR) (Week 2)

Design and implement the following programs in MapReduce (Java).

MAX-filter: Write a simple filter program the removes all edges containing a node ID greater than or equal to MAX. For example, given edges (1, 2), (1, 100), (50, 2), (60, 40) and MAX=10, only edge (1, 2) should be kept.

RS-join: Implement the Reduce-side join to compute Path2, the set of length-2 paths in the Twitter follower graph. Since this is a self-join between the Edges dataset and itself, make sure you use each input edge twice in the appropriate way in your program—do *not* duplicate the edges file beforehand in the input directory! Then implement another Reduce-side join between Path2 and Edges to "close the triangle." Recall that we want only the number of triangles, not the triangles themselves. See if you can exploit this to reduce program cost. For example, can you perform "early counting" in the joins, avoiding a third MR job for counting result size?

Rep-join: Instead of Reduce-side join, implement the same program using Replicated join (Map-only join). Think about reducing cost, e.g., by performing multiple steps in a single job.

Also try to integrate the functionality of MAX-filter directly into the Map phase of RS-join and Rep-join.

Before running the programs on the full Edges dataset, use the job input/shuffle/output cardinality/volume table you prepared to estimate the efficiency and feasibility of the programs. Which one moves less data through the network on a 5- or 10-worker cluster? Will they be able to process the complete input in about 1 hour on 5 m4.large or m4.xlarge worker machines?

The last question will be difficult to answer without running the programs on AWS with different MAX values. Start with a small MAX first and observe how long the entire triangle-counting computation takes on 5 EMR machines. Then change MAX until the computation takes close to, but less than, 1 hour. It does not have to hit exactly 59 minutes. You may stop exploring MAX values as soon as you found one where your program takes between 20 and 60 minutes.

Program Design and Implementation: Spark (Week 2)

Implement the equivalent of RS-join and Rep-join, each with built-in MAX-filter, in Spark Scala. For each program, create two versions: one uses only RDD and pair RDD, the other only uses DataSet. This involves some searching for the right join function or join function parameter settings. If you cannot find the right function for one of the program versions, state so in your report and briefly explain where you looked for the functions.

Report

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

Header

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

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. (5 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 all **logical** and **physical** plans returned by explain(). (2 points)

For each of the five programs, report the *time taken* (running time) and the *data shuffle* amount. (5 points)

Based on this information, state clearly (1) which of the programs performs aggregation before shuffling, and (2) cite the evidence from the above numbers to support your claim. (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**. Do *not* report timing measurements obtained when running in the IDE!

Joins in MapReduce and Spark (60 points total)

Show the MapReduce pseudo-code for the program you used to determine the cardinality (and maybe volume) of Path2. (4 points)

Show the table with all 12 cardinality and all 12 volume estimates for the two join steps and RS-join vs. Rep-join. If you merge the two steps into one for one or both join types, state so clearly in the report. Then you only have to report the corresponding input/shuffle/file cache/output numbers for the merged program. (12 points)

If you were able to obtain the exact cardinality of Paths, then report it. *Otherwise* report multiple pairs of (MAX value, Path2 cardinality for that MAX value pairs) that you used to estimate the full cardinality of Path2. Present these pairs in a graph whose x-axis is the MAX value and whose y-axis is the size of Path2 for that MAX value. (4 points)

Show the Map-Reduce pseudo-code for MAX-filter, RS-join, and Rep-join. If you found a way to merge multiple steps into one, only show the version of your program with the fewest MR jobs. If you integrated the MAX-filter functionality into the Map phase of the join, then you not need to show the separate MAX-filter pseudo-code. (20 points)

Show the pseudo-code for the pair RDD version of MAX-filter/RS-join and MAX-filter/Rep-join in Spark Scala. Since many Scala functions are similar to pseudo-code, you may copy-and-paste them here whenever appropriate. (10 points)

Show the pseudo-code for the DataSet version of MAX-filter/RS-join and MAX-filter/Rep-join in Spark Scala. Since many Scala functions are similar to pseudo-code, you may copy-and-paste them here whenever appropriate. (10 points)

Pick the MAX threshold for each program separately, so that it finishes on the smaller cluster in 20-60 minutes. 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)

Report your results in a table like this: (15 points)

Configuration	**Small Cluster Result**	**Large Cluster Result**
RS-join in MR, MAX = ???	Running time: ???, Triangle count: ???	Running time: ???, Triangle count: ???
Rep-join in MR, MAX = ???	Running time: ???, Triangle count: ???	Running time: ???, Triangle count: ???
RS-join in Spark, RDD, MAX = ???	Running time: ???, Triangle count: ???	Running time: ???, Triangle count: ???
RS-join in Spark, DataSet, MAX = ???	Running time: ???, Triangle count: ???	Running time: ???, Triangle count: ???
Rep-join in Spark, RDD, MAX = ???	Running time: ???, Triangle count: ???	Running time: ???, Triangle count: ???
RS-join in Spark, DataSet, MAX = ???	Running time: ???, Triangle count: ???	Running time: ???, Triangle count: ???

Deliverables

Submit the following in a **single standard ZIP file** (**not** rar, gzip, tar etc!). To simplify the grading process, please name this file yourFirstName_yourLastName_HW#.zip. 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.zip:

1. The report as discussed above. (1 PDF file)
2. Log files describing the overall job execution for the run you used to fill in the corresponding cell in the above table. (syslog or similar, 6 points)

Make sure the following is easy to find in your **CCIS Github** repository:

3. The MapReduce projects, including source code and build scripts. (7 points)
4. The Spark Scala projects, including source code and build scripts. (7 points)

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.