CS 6240: Assignment 2

Goals: Implement non-trivial joins in MapReduce, 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 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 Canvas 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 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.

Joins in MapReduce

In many courses, homework assignments are designed so that they can be solved in the "obvious" way as taught in class. Unfortunately, that is not how the real world tends to function. We therefore decided to give you a slightly more realistic challenge: find 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 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**.

We want 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 (Week 1)

We will solve the triangle counting problem with joins. For simplicity, assume our input is a table called "Edges" with columns "from" and "to," i.e., tuple (X, Y) represents the followership from X to Y. The solution for triangle counting has 3 *conceptual* steps (your implementation may combine some of these into fewer jobs and you may be able to count the triangles without producing them):

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. 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.

Since all join conditions are equalities, we can use any equi-join algorithm. The lecture material introduces Reduce-side join (hash+shuffle) and Replicated join (partition+broadcast). 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. Enter the numbers you calculate, estimate, or measure 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 (join of Edges with itself)	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 (join of Path2 with Edges)	Total cardinality and volume of input	Total cardinality and volume of data sent from Mappers to Reducers	Good upper bound for total cardinality and volume of output	Total cardinality and volume of input	Total cardinality and volume of data broadcast to all machines	Good upper bound for total cardinality and volume of output

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,

determine 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 here):

1. Estimating the number of tuples in Path2: The simplest solution would be to execute the join from step 1 and look at its output. 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 or S3 could take too long, and the result may not fit on your laptop's hard drive. 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 $m \cdot n$ 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 determine the desired count statistics for intermediate result Path2. Think about a MapReduce program to do this—it suffices to run it *locally* to save money (using "make local", not the IDE), but you can also use AWS.
2. Estimating the total number of triangles: This is more challenging, because we need to know the actual endpoints of each length-2 path 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 there cannot be more triangles than length-2 paths! Stated differently, we at least know an upper bound on the final output size and can focus on the intermediate result as the bottleneck of the computation. Hence it is acceptable to enter the cardinality and volume estimate of Path2 as a good upper bound 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 a problem. Then you will have to resort to a universal fail-safe strategy for Big Data: data reduction and approximation. This often 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 about 100 outgoing edges are sampled. This results in output size underestimation, because for Y we estimate zero results, even though it contributes 1000 paths of length 2. (Random sampling over joins is known as a difficult problem. For more information 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.

You can do the following on your *local machine* to find a good value for MAX. (For accurate estimates you do not want it too small; for avoiding performance problems you do not want it too large.) Start with a small MAX value and then systematically try larger ones. We recommend a geometric sequence, e.g., double or quadruple the MAX value each time, to get an idea how quickly computation cost and result size for that MAX value grow. See if you can spot a trend and leverage your insight to estimate the total result size if we had not used the MAX threshold to remove edges.

Design and implement the following program in MapReduce:

MAX-filter: Write a simple filter program that 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.

Bonus challenge (5 extra points): Implement a smarter MAX-filter approach by compacting and shuffling the range of user IDs. The MAX threshold is simple, but potentially problematic because it depends on user IDs, which are artificial values. For example, assume the data has users with IDs 1-10 and 1000-1100. Setting MAX=20, we get the edges for the first 10 users. Doubling MAX to 40 does not add any edges, and this remains unchanged until we reach MAX = 1280 when suddenly a lot of new edges for users 1000-1100 are included. The corresponding trend graph with MAX on the x-axis and the number of included edges on the y-axis would look like an extreme step function. Now assume the IDs were modified by replacing all numbers in range 1000-1100 with numbers in range 11-111. Then the trend would look very different, even though the graph did not change. This can be addressed by range-compaction and randomization: (1) map the existing user IDs to consecutive numbers in range [1,number_of_users], (2) shuffle the positions in this range, and (3) apply MAX based on the shuffled positions.

Example: Assume we are given user IDs 1, 7, 1000, and 1500. To implement the above idea, we can create an array A = [1,7,1000,1500] and then shuffle it to obtain, say, B = [1000,1,1500,7]. Then MAX=2 would select only the first entry in B, i.e., user 1000. Similarly, MAX=3 would select users 1000 and 1, and so on.

Join-Program Design and Implementation in MapReduce (MR) (Week 2)

Design and implement the following two join programs in MapReduce.

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 lower 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.

Before running the programs on the full Edges dataset, use the table you prepared earlier to predict their efficiency and feasibility. 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 m5.xlarge worker machines?

If you believe you cannot make your program work on the full input, use MAX-filter to reduce the number of edges. Try to integrate the functionality of MAX-filter directly into the Map phase of the join, instead of running it as a separate job.

If you use MAX-filter, pick a MAX number where you get "reasonable" job running time on 5 cheap EMR worker machines like m5.xlarge. Here reasonable means between 15 min and 1 hour. Use your local machine as much as possible to understand the relationship between the value of MAX filter and key statistics like (intermediate) result sizes and running time. Run on EMR only for a few "interesting" MAX values to get into the desired range. (Note: It is fine to submit results for a program running longer than 1 hour. We set the upper limit to protect you from spending too much money: If your job is still not done after 1 hour, how long are you willing to wait? If you are not sure, terminate it and set a smaller MAX value.)

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 Github repository for this homework. (4 points)**

Problem Analysis (20 points total)

Show the MapReduce pseudo-code for the program you used to determine the cardinality (and maybe data volume) of Path2. If you did not use a program, show the steps of the analysis you performed to estimate the number. (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 Path2, 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. If you solved the bonus challenge, then also report the corresponding numbers for the compacted and reshuffled user IDs. 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)

Join Implementation (48 points total)

Show the Map-Reduce pseudo-code for MAX-filter, RS-join, and Rep-join. If you found a way to merge multiple program 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 do not need to show the separate MAX-filter pseudo-code. (30 points)

Pick the MAX threshold for each program separately, so that it finishes on the smaller cluster in 15-60 minutes. Run each program with the corresponding MAX value on AWS on the full Twitter edges dataset. (If you solved the bonus challenge, make sure MAX-filter is applied to the compacted and shuffled range of user IDs, not the original ones.) Use the following two configurations:

- 6 cheap machines (1 master and 5 workers)
- 11 cheap machines (1 master and 10 workers)

Use the same machine type for all experiments. Report your results in a table like this, indicating clearly if you applied MAX-filter to the original user IDs or the compacted and shuffled version from the bonus challenge: (10 points)

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

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 Canvas; for the files on Github it is the time the files were pushed to Github, according to Github. We recommend the following approach:

1. Push all files to Github and make sure everything is there (see deliverables below). (Optional: Create a version number for this snapshot of your repository.)
2. Submit the report on Canvas. 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 on Canvas. 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 Canvas; 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 **Github** repository:

2. Log file for each job you used to fill in the corresponding running-time cells in the above table. (syslog or similar, 4 points)
3. All output files produced by those same successful runs on AWS (4 sets of part-r-… or similar files). (4 points)
4. The MapReduce 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. 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.