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 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.

Joins in MapReduce

In many classes, 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 with the resources available. 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)

One could solve the triangle-counting problem by first finding and outputting all triangles and then counting them. **But is this solution really feasible?** Let us find out by considering the following solution—let's call it "Naïve"—assuming our input is a table called "Edges" with columns "from" and "to," i.e., tuple (X, Y) represents that X follows Y:

1. Join the Edges table with itself to find paths of length 2. Tuples (X, Y) and (A, B) join to produce output tuple (X, Y, B) if and only if the edges connect on the intermediate node, i.e., 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).
2. Next, we join Path2 with Edges to close the triangle. This step checks for each path (X, Y, Z) if (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.

To determine if the Naïve solution is feasible, it helps to know *important statistics* like input and output size of each computation step. Some statistics are easy to obtain, e.g., the input size of step 1, which is the original input graph. For others, we can derive estimates or upper bounds analytically. For example, notice that the number of triangles cannot exceed the number of length-2 paths: for each unique triangle (X, Y, Z), there is a unique path (X, Y, Z). This implies that the crucial unknown statistic in the Naïve program is the number of length-2 paths, i.e., the *cardinality* of Path2. We now discuss two approaches for determining the cardinality of Path2. The first computes it accurately and the second estimates it.

1. **Exact approach for determining the cardinality of 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 be infeasible, and the result may not even 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. 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 are exactly m·n length-2 paths through Y—one for each combination (x_i, Y, z_j). All we need to do is add up those products over all possible user IDs Y. In HW 1 you already determined m, hence now you only need to extend that program to also determine n and compute the product of m and n.

2. **Approximating the cardinality of Path2**: Sometimes it may not be possible or feasible to determine the exact count statistics for a problem. Then you 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 means that we severely underestimate output size as 0.[1] To obtain a better estimate, use the following 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. For example, given edges (1, 2), (1, 100), (50, 2), (60, 40) and MAX=10, only edge (1, 2) should be kept. Then execute step 1 (the join to create Path2) on this 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.

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

Armed with a good understanding of important statistics, in particular the cardinality of Path2, the next goal is to implement the triangle-counting algorithm. 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. Design and implement the following two join programs in MapReduce.

RS-join: Implement the Reduce-side join to compute Path2 (step 1). 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" (step 2). Recall that we want only the number of triangles, not the triangles themselves. Exploit this to lower execution cost.

Rep-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.

---

[1] 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.

Report

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

1. [10 points] Write a MapReduce program that uses a single job to determine the *exact* cardinality of Path2, i.e., the number of length-2 paths in the Twitter graph, as discussed above. Instead of using a second job for adding up the products of the number of incoming and outgoing edges over all nodes, use a global counter. Show the pseudo-code for this program.
2. [10 points] Show the link to the source code for this program in your Github Classroom repository.
3. [2 points] Run this program locally (no need to spend money on AWS) and report the cardinality of Path2 it computed on the full Twitter edges set.
4. [10 points] Write a MapReduce program that uses a single job to determine the *approximate* cardinality of Path2, i.e., the number of length-2 paths in the Twitter graph, as discussed above. Recall that this program uses a parameter MAX to remove edges and then performs the join of step 1 on the reduced edge set. Show the pseudo-code for this program.
5. [10 points] Show the link to the source code for this program in your Github Classroom repository.
6. [2 points] Run this program locally (no need to spend money on AWS) for a geometric sequence of MAX values, e.g., 100, 500, 2500, and so on. Plot your observations in a graph with MAX on the x-axis and the corresponding Path2 cardinality on the y-axis. As you increase MAX, look carefully at the increasing Path2 cardinality, and stop increasing MAX when you believe the next larger value will overwhelm your local resources (CPU, memory, or disk space).
7. [2 points] Look at the above graph and try to find the trend (manually "fit a curve" to the graph). Then use this trend to estimate the y-value when setting the x-value to the largest user ID in nodes.csv. Report this estimate and compare it to the Path2 cardinality you obtained from the exact approach. Is it close?
8. [2 points] 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 are 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 Twitter graph did not change. Find out the situation for the Twitter dataset by creating a plot with MAX on the x-axis and the *number of edges that remain after applying the MAX filter* on the y-axis. Use the same MAX values as in the previous question. (You can run this program locally, not on AWS.)
9. [10 points] Show the pseudo-code for your triangle-counting program using RS-join. It is allowed to use more than 1 MR job. Make sure your program has MAX as a parameter to control job cost.
10. [10 points] Show the link to the source code for this program in your Github Classroom repository.

11. [4 points] Run RS-join on EMR using 1 master and 4 worker nodes (all cheap machines as for HW 1). Based on your analysis, decide if you need to use the MAX filter for the RS-join program or if you can run it on the full Twitter edge set. If you use MAX filter, set it to a value for which you estimate that your program will finish in 15 min to 1 hour. You may need to try 2-3 "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.)
    a. Report the MAX-filter value you used.
    b. Report the triangle count obtained for this MAX-filter value.
    c. Report the running time of your program on EMR.
    d. Show a link to your syslog file created by the run for which you reported these numbers.
12. [2 points] Run the same RS-join program for the same MAX value on a larger cluster with 1 master and 8 workers (same instance type as above).
    a. Report the running time on the larger cluster.
    b. Compute the ratio between running time on the small vs the large cluster. Is this a good speedup?
13. [10 points] Show the pseudo-code for your triangle-counting program using Rep-join. It is allowed to use more than 1 MR job but try to solve the problem with a single job. Make sure your program has MAX as a parameter to control job cost.
14. [10 points] Show the link to the source code for this program in your Github Classroom repository.
15. [4 points] Run Rep-join on EMR using 1 master and 4 worker nodes (the same instance type as for RS-join). Based on your analysis, decide if you need to use the MAX filter for the Rep-join program or if you can run it on the full Twitter edge set. If you use MAX filter, set it to a value for which you estimate that your program will finish in 15 min to 1 hour. You may need to try 2-3 "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.)
    a. Report the MAX-filter value you used.
    b. Report the triangle count obtained for this MAX-filter value.
    c. Report the running time of your program on EMR.
    d. Show a link to your syslog file created by the run for which you reported these numbers.
16. [2 points] Run the same Rep-join program for the same MAX value on a larger cluster with 1 master and 8 workers (same instance type as above).
    a. Report the running time on the larger cluster.
    b. Compute the ratio between running time on the small vs the large cluster. Is this a good speedup?

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 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.