CS 6240: Assignment 5

Goal: Implement PageRank in MapReduce, but this time relying on a matrix-based approach. Compare performance to the adjacency-list based approach.

This homework is to be completed individually (i.e., no teams). You have to create all deliverables yourself from scratch. In particular, it is not allowed to copy someone else's code or text and modify it. (If you use publicly available code/text, you need to cite the source in your code and 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.) Always package all your solution files, including the report, into a single standard **ZIP** file. Make sure your report is a **PDF** file.

Please name your submitted ZIP file "**CS6240_first_last_HW_#**", where "first" is your first name, "last" your last name, and "#" the number of the HW assignment. For each program submission, include complete source code, build scripts, and small output files. Do not include input data, output data over 1 MB, or any sort of binaries such as JAR or class files.

To enable the graders to run your solution, make sure you include a standard Makefile with the same top-level targets (e.g., *alone* and *cloud*) as the one Joe presented in class (see the Extra Material folder in the Syllabus and Course Resources section). You may simply copy Joe's Makefile and modify the variable settings in the beginning as necessary. For this Makefile to work on your machine, you need Maven and make sure that the Maven plugins and dependencies in the pom.xml file are correct. Notice that in order to use the Makefile to execute your job elegantly on the cloud as shown by Joe, you also need to set up the AWS CLI on your machine. (If you are familiar with Gradle, you may also use it instead. However, we do not provide examples for Gradle.)

As with all software projects, you must include a README file briefly describing all of the steps necessary to build and execute both the standalone and AWS Elastic MapReduce (EMR) versions of your program. This description should start with unzipping the original submission, include the build commands, and fully describe the execution steps. This README will also be graded.

PageRank with Matrices

For simplicity, we first discuss the matrix-based computation without considering dangling nodes. Recall the definition of PageRank of node n as $P(n) = \alpha \frac{1}{|V|} + (1-\alpha) \sum_{m \in L(n)} \frac{P(m)}{C(m)}$, which resulted in an iterative algorithm that computes the values in iteration (t+1) from those in iteration t as

$$\forall n: P(n;t+1) = \alpha\frac{1}{|V|} + (1-\alpha)\sum_{m\in L(n)}\frac{P(m;t)}{C(m)}$$

In matrix notation, this can be written as

$$\mathbf{R}(t+1) = \frac{\alpha}{|V|}\mathbf{1} + (1-\alpha)\cdot\mathbf{M}\cdot\mathbf{R}(t)$$

With the following matrices:

$\mathbf{1} = \begin{bmatrix}1\\ \vdots \\ 1\end{bmatrix}$ is a $|V| \times 1$ column vector containing the value 1 for each entry.

$\mathbf{M}$ is a $|V| \times |V|$ matrix, such that $\mathbf{M}[i,j] = \begin{cases}1/C(j) & \text{if page } j \text{ links to page } i \\ 0 & \text{otherwise}\end{cases}$.

$\mathbf{R}$ is a $|V| \times 1$ column vector of PageRank values, such that $\mathbf{R}(t) = \begin{bmatrix}P(0;t)\\ P(1;t)\\ \vdots \\ P(|V|-1;t)\end{bmatrix}$.

Notice how the product of any row i of matrix $\mathbf{M}$ with vector $\mathbf{R}(t)$ aggregates the *incoming* contributions to page i, to compute its new PageRank $P(i;t+1)$.

The matrix representation also enables an elegant solution for the *dangling nodes* problem. Node j does not have any links to other pages if and only if column j in matrix $\mathbf{M}$ contains only zeros: Since j does not link to page 0, $\mathbf{M}[0,j] = 0$; since j does not link to page 1, $\mathbf{M}[1,j] = 0$, and so on. To distribute page j's PageRank evenly over all pages, we simply need to replace these zeros with $1/|V|$. More precisely, for each page j, if column vector $\mathbf{M}[*,j] = \begin{bmatrix}0\\0\\ \vdots \\ 0\end{bmatrix}$, set it to $\mathbf{M}[*,j] = \begin{bmatrix}1/|V|\\ 1/|V|\\ \vdots \\ 1/|V|\end{bmatrix}$. Let's refer to this modified matrix as $\mathbf{M}'$. Then iteration t should compute

$$\mathbf{R}(t+1) = \frac{\alpha}{|V|}\mathbf{1} + (1-\alpha)\cdot\mathbf{M}'\cdot\mathbf{R}(t)$$

Algorithm Summary

To implement PageRank using an adjacency-matrix representation, you need to implement the following steps:

1. Create matrix $\mathbf{M}'$ from either the original input or the adjacency-list representation from a previous assignment. (But also see the discussion about sparse matrix representation below!)
2. Create vector $\mathbf{R}(0)$ with the initial PageRank values $1/|V|$.
3. Run 10 iterations to compute $\mathbf{R}(10)$.
4. Return the top-100 pages from $\mathbf{R}(10)$.

Notes about these steps:

Step 1: The matrix should not store the page names. Instead, each row and column of $\mathbf{M}'$ corresponds to a unique page. This implies that you need to map each page 1-to-1 to a unique row and/or column number. This mapping will need to be inversed for step 4. Make sure you properly handle pages that are referenced by links, but do not show up themselves in the data.

Step 3: The main operation in an iteration is the computation of the product between matrix $\mathbf{M}'$ and vector $\mathbf{R}(t)$. Once the resulting vector with $|V|$ elements is obtained, you can simply multiply each of its elements by $(1-\alpha)$ and add $\alpha/|V|$ to it. There is no need to create vector $\mathbf{1}$ explicitly.

Step 4: Finding the top-100 PageRanks in $\mathbf{R}(10)$ should be straightforward, but recall that the vector does not tell you the page names. Make sure that the final result shows the *names* of the top-ranked pages, not just their index values in the vector.

Implement this algorithm in MapReduce. Think about how to effectively partition the problems to achieve good speedup. For the matrix-with-vector product in step 3, design and implement **two different versions**: version A partitions the matrix horizontally (i.e., into blocks of rows), while version B should partition the matrix vertically (i.e., into blocks of columns). You need to decide (i) how many rows or columns should make up a block and (ii) how to assign blocks to worker machines.

Dense Matrix versus Sparse Matrix

The straightforward implementation of the algorithm described above might not be feasible for very large matrices. Fortunately, in practice large matrices tend to be sparse, which can be exploited. Estimate how much data would be transferred through the network and how much work each machine would have to do. Based on this analysis, decide if you need a sparse matrix representation or not. (This includes vectors, which are special cases of matrices.) It is allowed to choose different representations for different matrices and vectors.

Matrix cells can be laid out in many different ways, the most well-known being row-major order and column-major order. Think carefully, which one fits your algorithm. In particular, determine what data a single Map or Reduce call should process and design the matrix serialization accordingly. You are allowed to use different representations for version A and B.

Make sure you understand how big (number of rows and columns) a matrix is. Then try to determine how many of its entries will be non-zero. Matrices that overwhelmingly contain zeros are called *sparse*. Choose an appropriate matrix representation based on its size and sparseness, taking into account how you will access its elements during PageRank computation. Here are three typical options, illustrated for example matrix $\mathbf{E} = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 0 & 0 \\ 3 & 4 & 0 \end{bmatrix}$ to be stored in row-major order (the approach for column-major order is analogous):

Possible dense representation: Each element is explicitly stored in a comma-separated format, where the semi-colon separates different rows. For the example: [1, 0, 2; 0, 0, 0; 3, 4, 0].

Compression: One can apply standard compression algorithms to the dense representation. This requires (i) generating the dense representation, (ii) compression, and (iii) decompression whenever the matrix is loaded for computation.

Possible sparse representation: Only rows with non-zero elements are stored, and for those rows the columns with non-zero values are explicitly enumerated as (column, value) pairs. For the example: [0: (0,1), (2,2); 2: (0,3), (1,4)]. All other elements are implicitly known to have value zero. Note the similarity to the adjacency-list representation of a graph.

There is another subtlety to be aware of, because matrix $\mathbf{M}'$ will be denser than $\mathbf{M}$. Determine how many zeros in $\mathbf{M}$ are turned into non-zero values by the construction described above for handling dangling nodes. If you believe that the higher density might cause a serious problem for the algorithm, try to find a solution. In particular, you could deal with dangling nodes separately as was done in the previous PageRank assignments. For example, matrix product can be decomposed as $\mathbf{M}' \cdot \mathbf{R}(t) = (\mathbf{M} + \mathbf{D}) \cdot \mathbf{R}(t) = \mathbf{M} \cdot \mathbf{R}(t) + \mathbf{D} \cdot \mathbf{R}(t)$. The first term uses the original (sparser) matrix. Matrix $\mathbf{D}$ in the second term has a special structure: each of its columns either contains only zero or only 1/|V|. This could be exploited for computing product $\mathbf{D} \cdot \mathbf{R}(t)$, by encoding $\mathbf{D}$ very compactly as an index of those column numbers that have value 1/|V|. The actual value 1/|V| does not have to be stored repeatedly. Depending on the partitioning (version A versus B), the algorithm needs to make sure that the right information from $\mathbf{D}$ is sent to the workers.

The above illustrate typical options; and you need to make your own choice. Depending on the matrix representation, design an efficient algorithm for the corresponding matrix multiplication.

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.

Design Discussion and Pseudo-Code (30 points total)

For each step of the algorithm, briefly explain the main idea for how you partition the problem. Show compact pseudo-code for it. It is up to you to decide how to map the computation steps to MapReduce jobs. For example, you can merge multiple steps into a single job, or you can use multiple jobs to implement a single step. Make sure you discuss and show pseudo-code for both versions of step 3. (20 points)

Briefly explain how each matrix and vector is stored (serialized) for versions A and B. Do not just paste in source code. Explain it in plain English, using a carefully chosen example. See the above discussion about dense and sparse matrices for an example how to do this. (5 points)

Discuss how you handle dangling nodes. In particular, do you simply replace the zero-columns in $\mathbf{M}$ and then work with the corresponding $\mathbf{M}'$, or do you deal with the dangling nodes separately? Make sure you mention if your solution for dangling nodes introduces an additional MapReduce job during each iteration. (5 points)

Performance Comparison (18 points total)

Run your program in Elastic MapReduce (EMR) on the full English Wikipedia data set from 2006, using the following two configurations:

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

Report for both configurations the total execution time, as well as the times for (i) completion of steps 1 and 2, (ii) time for an iteration of step 3, and (iii) time for step 4. Make sure you clarify if your program works with the original input files or with the parsed adjacency lists from a previous assignment. Since there are two versions (A and B) for step 3, you need to report 4 sets of running-time numbers. (4 points)

For each configuration, report the running time of an iteration executed by the *adjacency-list* based Hadoop implementation in the earlier HW assignment. How do these numbers compare to the adjacency-matrix based version? Briefly discuss if you find the result surprising and explain possible reasons for it. Make sure you back your arguments with concrete data, e.g., from the log files. (10 points)

Report the top-100 Wikipedia pages with the highest PageRanks, along with their PageRank values, sorted from highest to lowest, for both the simple and full datasets. Are the results the same as in the adjacency-list based Hadoop implementation? If not, try to find possible explanations. (4 points)

Deliverables

1. The report as discussed above. (1 PDF file)
2. The source code of your programs, including an easily-configurable Makefile that builds your programs for local execution. **Make sure your code is clean and well-documented. Messy and hard-to-read code will result in point loss.** In addition to correctness, efficiency is also a criterion for the code grade. (45 points)
3. The syslog (or similar) files for a successful EMR run for both system configurations. (4 points)
4. *Final* output files from EMR execution only, i.e., only the top-100 pages and their PageRank values. (3 points)