Assignment #2: Imperative SQL

For this assignment, you will be writing a few stored procedures in SQL to analyze a graph dataset. The dataset to analyze contains citation information for about 5,000 papers for the Arxiv high-energy physics theory paper archive. The dataset has around 14,400 citations between those papers.

The dataset is comprised of the following two tables:

```
nodes (paperID, paperTitle)
edges (paperID, citedPaperID)
```

The `nodes` table gives a unique paper identifier, as well as the paper's title. The `edges` table indicates citations between the papers (note that citations have a direction).

Tasks

Your job is to write stored procedures to analyze the aforementioned data.

2.1 Task 1 Connected Components

You will first write a stored procedure that treats the graph as being undirected (i.e., you do not need to worry about the direction of the citations) and find all connected components in the graph that have **more than four and at most ten papers**. You will then need to print the associated lists of paper titles.

To refresh your memory, a connected component is a subgraph such that there exists a path between each pair of nodes in the subgraph. Such a subgraph must be maximal in the sense that it is not possible to add any additional nodes that are connected to any node in the subgraph.

The standard method for computing a connected component is a simple *Breath-First Search (BFS)*: Pick a random starting node, and then search for all nodes reachable from the starting node, then search for all nodes reachable from all of *those* nodes, and then search for all the nodes reachable from *those* nodes, and so on, until no new nodes are found. The entire set of discovered nodes is a connected component. If there are any nodes that are not part of any connected component analyzed so far, then pick one of those nodes and restart the process. You are done when all of the nodes are part of exactly one connected component.

Your program should first compute all of the connected components, then print out the ones that are **larger in size than four, but no larger in size than ten**. When you print out the components, print each paperID, its title, and cluster identifier.

2.2 Task 2 PageRank

PageRank is a standard graph metric that is well-known as the basis for Google's original search engine. The idea behind PageRank is simple: We want a metric that rewards web pages (or, in our case, physics papers) that are often pointed to by other pages. The more popular the page, the greater its PageRank score.

To accomplish this, PageRank models a web surfer, starting at a random page, and randomly clicking links. The surfer simply goes to a page, sees the links, and picks one to follow. After each link clicked, there is a probability ***1 - d*** that the surfer will jump to a random page. This ***d*** parameter is called a damping factor and we will use a value of ***d = 0.85*** for this assignment.

Given this setup, the so-called PageRank of a web page (or a physics paper) is the probability that when the user stops clicking (or following citations), they will land on that page.

An important thing to consider is the existence of *"sinks"*, that is, pages that do not link anywhere else (or papers that do not cite other papers). These would accumulate all of this probability since they contain no outward links. In order to deal with this, we assume that sinks instead link with equal probability to every other page.

There are many ways to compute the PageRank for every paper in the data set. The simplest is the following iterative computation. Let $PR_i(paper_j)$ denote the estimated PageRank of $paper_j$ at iteration i; assume that there are n papers in all. We start out with $PR_0(paper_j) = 1/n$ for all j. Then, at iteration i, we simply set:

$$PR_i(paper_j) = \frac{1-d}{n} + d\left(\sum_{k \in \{papers\ citing\ paper_j\}} \frac{PR_{i-1}(paper_k)}{num\ citations\ in\ paper_k}\right)$$

This iterative process is continued until there is only a small movement in probability across iterations. In our case, we'll continue as long as:

$$0.01 < \sum_j |PR_i(paper_j) - PR_{i-1}(paper_{j)}|$$

Your goal for this problem is to write one, or more, stored procedures that together compute the PageRank for each of the papers in the graph. You should run your code and print out the 10 papers with the greatest PageRank (sorted), as well as their corresponding PageRank scores. Again, when you print out a paper, print both its paperID and title.

Note: The sum of all PageRanks should always equal 1 — but only if you handle sinks properly (by redistributing their rank mass).

Getting Started

First, log onto SQL server, go to your database and create two tables:

```
CREATE TABLE nodes (
      paperID INTEGER,
      paperTitle VARCHAR (100));

CREATE TABLE edges (
      paperID INTEGER,
      citedPaperID INTEGER));
```

Once you've done this, unzip the file that we've provided and use the `nodes.sql` and `edges.sql` scripts to load the data into the database.

One important thing: don't keep anything important on the database server. There's a reasonable chance that people are going to leave many gigabytes of "trash" (junky, leftover tables) on the server as a result of this assignment, which can cause it to become unstable. If this happens, I might have to go in and start removing trash-clogged databases so that people can continue working. Thus, it's important that you save your code somewhere else.

A Note on Speed

It is very important that you try to do as much as possible declaratively. Looping through the contents of a table using a cursor is necessarily going to be slow. You should try to do as much as possible using declarative SQL queries. Use loops and conditionals to guide the overall control flow, and when there's clearly no way to do what you want using declarative SQL.

On this assignment, there's often a 100X, or more, difference in performance between a well-written code that is mostly using declarative queries, and one written with a lot of loops. Speed does not matter for grading purposes, however, it is easy to write code that is so slow it will not complete in a reasonable amount of time. Not to mention that declarative queries are easier to code and debug!

Turn-in

Submit a single compressed ***netid_assignment2.zip*** file containing the following files:

1. A *netid_connected.sql* file (no image) with your code for Task 1 (Connected Components).
2. A *netid_pagerank.sql* file (no image) with your code for Task 2 (PageRank).
3. A *netid_results.pdf* file containing the results of running your code for both tasks.

Grading

Each problem is worth 50% of the overall grade. If you get the right answer and your code is correct, you will get full points. If you don't get the right answer or your code is not correct, you will not get all of the points. Partial credit may be given at the discretion of the grader.