Assignment 5 Description

JDBC – Functions – Recursive Queries

This assignment description refers to **Java JDBC** programs and JDBC classes and object. However, any other equivalent high-level language database API (e.g. C++/C# ODBC, Python, etc) is acceptable so long as it allows you to write standard SQL queries and access the database through its API. You will not be submitting computer code.

Assignment Overview

This assignment has 3 parts and one extra credit part. East part builds on the one previous to it. They are briefly described below. Following the overview, each part is described in detail.

In **Part I** of this this assignment you will be creating a simple database from a supplied data set. The data set consists of two comma-separated files which describe a set of people and their hierarchical relationship to each other. You will create tables (described below) and insert the data from the files into the tables. You will answer a few questions about the data that will require you to write and execute a few SQL queries.

In **Part II**, you will implement a **FindAllAncestor** function four ways –

1) As a **Java JDBC** (or equivalent) program, implementing the pattern demonstrated in class and found in the **FindAllPrereq** Java class supplied with the assignment. (See notes at the end of the assignment description[i])
2) As an **SQL stored function** in your database using the pattern shown in class and found in the **find-all-pre-req-f.sql** file.
3) As an **SQL recursive query**, following the pattern shown in class and found in the **find-all-prereq-rq.sql** file.
4) And finally as an **SQL stored function** which uses the recursive query to simplify the code, as briefly shown in class and found in the **find-all-prereq-f-rq.sql** file.

In **Part III** of the assignment, you will modify your **SQL stored function** to include one more piece of information about the data.

Part IV is optional extra credit. It describes an advanced way for implementing what amounts to a depth-first search of the relationship hierarchy which provides deeper insight.

Part I – Prepare the Database

You are given two comma separated value (.csv) files that contain person identifiers, gender identifiers, and parent-child relationship pairs.

- **id-gender-list.csv** – comma-separated-value file with two columns
    - First line is a column header
    - Column 1: person identifier
    - Column 2: single character gender identifier; 'f' = female, 'm' = male
- **parent-child-list.csv** – comma-separated-value file with two columns
    - First line is a column header
    - Column 1: child person id
    - Column 2: parent person id

These files are found with the accompanying assignment material.

Your assignment in Part I is to write a Java JDBC (or equivalent) program to

1. create two tables in your default database schema
2. insert the data from the lists using JDBC **PreparedStatement** objects
    a. insert the data from **id-gender-list.csv** file into the *person* table
    b. insert the data from **parent-child-list.csv** file into the *parent_child* table

Table Schemas

```
create table person(
      id varchar(10) primary key,
      gender varchar(1),
      unique(id, gender));

create table parent_child(
      child_id varchar(10) references person,
      parent_id varchar(10) references person,
      unique(child_id, parent_id));
```

After creating and loading the data, write and execute SQL queries to answer assignment questions 1 – 4. Use any means you choose to execute the SQL statements; Java JDBC, pgAdmin, pgsql console, etc. Do not submit the text of your queries, only the answers.

Part II – Create “FindAllAncestor” Programs and Queries

In this part of the assignment, you will write a Java JDBC program, a stored SQL function and a recursive query to “find all ancestors” given a person id as a seed to start from. The pattern is identical to the Java program, SQL function and recursive query for finding course perquisites that was demonstrated in class.

Java JDBC Program

Write a Java JDBC program to traverse the *parent_child* table using the same algorithm we used to find all the prerequisite courses for a given course. You may copy and modify the **FindAllPrereq.java** program supplied with this assignment.

The question you will answer **asks for a count** of ancestors. You do not need to print out or list all the ancestors. You can leave the final query as it is and count the results in your program or simply modify the final query to return a count directly.

SQL Function

Write and install an SQL function that is the equivalent to the Java program. You may modify the **find-all-prereq-f.sql** function that is supplied with this assignment. You will be asked to use this function to answer a few questions.

Recursive Query

Write a recursive query to find all ancestors for a given person. You may modify the **find-all-prereq-rq.sql** query that is supplied with this assignment. You will use this query with various modifications to questions 5 through 10 in the assignment.

Part III

Modify your “find all ancestors” SQL stored function to not only find all ancestors, but also what generation the ancestors belong to. Here are some guidelines –

- Add an integer column to your local tables to keep track of the generation of the tuple.
- Do not use a global variable in your function to keep track of the generation. Use the value of the generation column, incrementing by one each time you add the next generation of parents.
- The root/seed person is considered to be generation 1. So the initialization of the iteration table should begin with generation 2 representing the seed’s parent generation.
- **You will have to modify the core select query that joins the iteration table with the parent_child table to find the next set of parents and selects into the temporary table.** Because we’ve added a generation column to the tables, the **except** clause will no longer function to eliminate duplications and eliminate endless loops. To understand this, consider the case where we encounter ancestor ‘X’ in generation 3 and then again in generation 7. When eliminating duplicates, the **except** clause considers all the columns we have selected which includes the generation count. Because the generation in the first case was 3 and in the second encounter was 7, the **except** clause does not consider these to be duplicates and will include them both in the select even though the person id was the same. We will continue to encounter ancestor ‘X’ every few generations in an endless loop using the **except** clause.
- We can accomplish the desired result (eliminating duplications) by doing two things:
    - Replace the **except** clause with a **subquery in the where** clause of the first select. Use the subquery to test that the parent_id that is being selected is not already in the result table. This will guarantee that if we have seen a parent before (i.e. they already exist in the result table) we won’t add them again.
    - The second fix is a little more subtle. Even if we haven’t encountered a parent before, It is possible that the same parent will be selected twice (or more) in the **current** join. This is not necessarily an error in the data. It is ~~possible~~ ~~common~~ ~~likely~~ certain that in every real pedigree, a person will have the same ancestor at the same generation level through multiple ancestor paths. To avoid adding the same ancestor during the current generation selection of parents, you will need to add **distinct** to the select statement.

Make the modifications to the SQL function and make sure it returns a table with both the ancestor id and their generation. You will use this function to questions 11 – 15 in the assignment.

Part IV – Extra Credit for the Truly Curious and Motivated

You were not asked to modify the recursive query to keep track of generations. Following the pattern of Part III is problematic when it comes to the recursive query because the recursive query does not allow subqueries of the **recursion table** in the iteration section so there isn't a simple way to prevent cycles (infinite loops).

There is another way to prevent cycles, but it will not directly produce the same results as the function you created. The reason is that even though the "fix" does prevent cycles, it does not prevent duplicate person ids. This is actually a desirable feature. As mentioned in Part III, the same ancestor may legitimately show up in different generations. The function you created in part III keeps the first occurrence but discards future encounters of the same ancestor. The scheme used with the recursive query will keep all occurrences, provided they are arrived at on a different path.

This is the difference between a **breadth-first** and a **depth-first** search. The Part III function implements a breadth-first search – moving one generation at a time through the relationships and keeping track of all the ancestors in a common table.

The recursive query (shown below) also moves through the hierarchy one generation at a time but it keeps track of ancestors that have been visited in a path variable (an array of ancestor ids) that is part of the table row attributes. The effect is that each tuple in the result table (that is accumulating with each iteration) has its own list of visited ancestors. A new parent will not be selected if it shows up in the path array for the child that is being selected against.

Here is the modified recursive query –

```
with recursive ancestor(ancestor_id, gen, path) as (
        select distinct PC.parent_id, 2,
               cast(ARRAY[] as varchar(10)[]) || PC.parent_id
        from parent_child PC
        where PC.child_id = 'LFDN-3X3'
    union
        select distinct PC.parent_id, A.gen + 1, A.path || PC.parent_id
        from ancestor A join parent_child PC on A.ancestor_id = PC.child_id
        where (PC.parent_id is not null) and (PC.parent_id != ALL(path))
    )
select distinct ancestor_id, gen from ancestor;
```

The detailed explanation is left as an exercise. However it is quickly noticed that an extra column has been added to the recursive table. It has an ARRAY type and keeps track of the "path" (i.e. list of ancestors) that have been traversed to get to an ancestor in the hierarchy.

In its current form, it will return the same information as the function you implemented in Part III. For extra credit, modify the final select statement to answer extra credit questions 16-18 in the assignment.