Assignment 7: Database Design Patterns

This week you will explore two important topics in relational database usage. The first part focuses on implementing database authentication without storing passwords in plaintext. The second part focuses on working with hierarchies in two different representations.

A template archive is provided with this assignment, so that you can simply fill in the answers between the required annotations. When you complete the assignment, repackage this archive with the name **`cs121hw7-username`**, and submit it on the course website.

Part A: Secure Password Storage (30 points)

It is unfortunately all too common for database schemas to include a severe design defect: storing user passwords in plain text in the database. This is a catastrophic flaw for several reasons. First, once the database is compromised, the attacker can use users' passwords to access the system through normal channels. Second, most users use the same passwords for many different systems; once a user's password is known, that password can be used to compromise other systems that the user has access to.

A simple solution is to apply a cryptographically secure hash function to passwords. Instead of storing the password itself, a *hash* of the password is stored. Authentication simply involves applying the hash to the entered password, and then comparing the hash values. Unfortunately, this approach is susceptible to *dictionary attacks*; users are very predictable, and by computing the hash codes of the most common passwords (e.g. "abc123", "password", and "123456" are in the top 10 most common passwords), an attacker with access to the database can simply compare the stored hash codes to a dictionary of hash codes computed from common passwords. This won't compromise all user accounts, but almost always it will compromise at least a few accounts.

To prevent dictionary attacks, one can prepend a *salt* value to the password before applying the hash function; if the salt is large enough then the potential for dictionary attacks is eliminated. Of course, there is still the issue of users picking bad passwords, but several techniques can prevent brute-force attacks, such as imposing a time delay on authentication failure, and enforcing strict policies on password choices such as expiring passwords after a period of time, and disallowing users from using any common password.

For this section, you will implement a secure password storage mechanism using both hashing and salting. Note that such functionality is normally implemented in the application sitting on top of the database, or even in the web client. You, however, will be implementing these operations as stored routines in the database, simply to avoid the complexity of having to interface with the database from another programming language.

(The reason for implementing these operations in other parts of the system is to limit where the plaintext password is actually visible. For example, a more secure authentication mechanism for a client-server system would be for the server to pass the salt to the client once the username is known; then the client can salt and hash the password and send it back to the server. The password itself is simply never communicated from the client. Other similar mechanisms exist as well.)

All of your work for this problem will be in the `passwords.sql` file. Include your completed version of this file in your submission. Additionally, a helper function called **`make_salt`** is provided in **`make-salt.sql`**, which is able to generate a random salt of up to 20 characters. You can use it like this:

```
SELECT make_salt(10) AS salt;
```

Note: Do not include the definition of `make_salt` in your submission; the autograder will load it separately.

Scoring: Part 1 is 4pts, parts 2 and 3 are 8pts, part 4 is 10pts. Total is 30 points.

1. Create a table called **`user_info`** that will store the data for this password mechanism. There are three string values the table needs to store:

   - Usernames, in a column named **`username`**, will be up to 20 characters long
   - Salt values, in a column named **`salt`**
   - The hashed value of the password, called **`password_hash`**

   We will be using the SHA-2 function to generate the cryptographic hash. (MySQL also has the MD5 and SHA-1 hash functions, but both of these are now considered too weak to be secure.) You can try out the SHA-2 hash function in MySQL like this:

   ```
   SELECT SHA2('letmein', 256);
   ```

   This function can generate hashes that are 228 bits, 256 bits, 384 bits, or 512 bits. We will use 256-bit hashes. Note that the hash is returned as a sequence of hexadecimal characters, so you will need to choose a suitable string length to store the hexadecimal version of the 256-bit hash. Make your **`password_hash`** column a fixed-size column that is exactly the right size, no larger and no smaller.

   You should use a salt of at least 6 characters.

2. Create a stored procedure **`sp_add_user(new_username, password)`**. This procedure is very simple:

   - Generate a new salt.
   - Add a new record to your **`user_info`** table with the username, salt, and salted password.

   Make sure that you prepend the salt to the password before hashing the string. You can use the MySQL **`CONCAT(s1, s2, ...)`** function for this.

   Don't worry about handling the situation where the username is already taken; assume that the application has already verified whether the username is available.

3. Create a stored procedure **`sp_change_password(username, new_password)`**. This procedure is virtually identical to the previous procedure, except that an existing user record will be updated, rather than adding a new record.

   Make sure to generate a new salt value in this function!

4. Create a function (not a procedure) called **`authenticate(username, password)`**, which returns a **`BOOLEAN`** value of **`TRUE`** or **`FALSE`**, based on whether a valid username and password have been provided. The function should return **`TRUE`** iff:

   - The username actually appears in the **`user_info`** table, and
   - When the specified password is salted and hashed, the resulting hash matches the hash stored in the database.

If the username is not present, or the hashed password doesn't match the value stored in the database, authentication fails and **`FALSE`** should be returned.

Note that we don't distinguish between an invalid username and an invalid password; again, this is to keep attackers from identifying valid usernames from outside the system.

Once you have completed these operations, make sure you test them to verify that they are working properly. For example:

```
CALL sp_add_user('alice', 'hello');
CALL sp_add_user('bob', 'goodbye');

SELECT authenticate('carl', 'hello');       -- Should return 0/FALSE
SELECT authenticate('alice', 'goodbye');    -- Should return 0/FALSE
SELECT authenticate('alice', 'hello');      -- Should return 1/TRUE
SELECT authenticate('bob', 'goodbye');      -- Should return 1/TRUE

CALL sp_change_password('alice', 'greetings');

SELECT authenticate('alice', 'hello');      -- Should return 0/FALSE
SELECT authenticate('alice', 'greetings');  -- Should return 1/TRUE
SELECT authenticate('bob', 'greetings');    -- Should return 0/FALSE
```

Part B: Employee Hierarchies *(70 points)*

In class we discussed three common ways of representing hierarchies in a database. In this section you will explore two of them, the adjacency-list representation and the nested-sets representation.

The data you will use for this section is in the file **`make-emptree.sql`**, provided. Loading this file will create a table **`employee_hierarchy`**, which actually includes both representations of the hierarchy. From this table, two other tables are generated; **`employee_adjlist`** which includes only the adjacency-list representation, and **`employee_nestset`** which includes only the nested-sets representation. The hierarchies are identical. Every employee also has a salary, which is randomly generated, and unlike real life, it has nothing to do with where they appear in the hierarchy.

For these problems, you may not use **`employee_hierarchy`** directly; use **`employee_adjlist`** and **`employee_nestset`**. Some problems specify which one of these tables you are required to use.

Make sure you write clear comments describing your answers for each of these problems. A lack of commenting will result in point deductions.

You might also again benefit from the use of the **`ROW_COUNT()`** function, to tell whether an iterated query has actually made any changes.

Scoring: problem 1 is 14pts, 2 is 10pts, 3 is 8pts, 4 is 10pts, 5 is 14pts, 6 is 14pts.

Implement your answers to these questions in a file **`emptree.sql`**.

1. Write a user-defined function **`total_salaries_adjlist(emp_id)`** that uses **`employee_adjlist`** to compute the sum of all employee salaries in a particular subtree of the hierarchy. The specified employee's salary should also be included. For this problem, do not use the **`WITH`** clause, so you can see how temporary tables are used to compute such results. You may use the **`WITH`** clause on any subsequent problem that might benefit from it.

2. Write another user-defined function **`total_salaries_nestset(emp_id)`** that uses **`employee_nestset`** to compute the sum of all employee salaries in a particular subtree of the hierarchy. The specified employee's salary should also be included.

3. Using the **`employee_adjlist`** table, write the SQL query to find all employees that are leaves in the hierarchy; the result should include the employee ID, name, and salary, in that order.

   Note: In MySQL, the expression "***`attr`*** **`IN (`*****`subquery`*****`)`**" will always evaluate to false if ***`subquery`*** produces any **`NULL`** values! This is also the case for "***`attr`*** **`NOT IN (`*****`subquery`*****`)`**". The reason is the same as always: anything compared to **`NULL`** is always *unknown*.

4. Implement the same operation as in #3, but this time using the **`employee_nestset`** table.

5. Write a function **`tree_depth()`** that finds the maximum depth of the tree. You can use the tree representation of your own choosing to implement this function; however, explain why you chose the representation you chose in a comment before the function, and also make sure to explain how your implementation works. **A tree with a single node has a depth of 1.**

6. Given a particular employee ID *emp_id*, it should be obvious how one would retrieve that node's immediate children from the **`employee_adjlist`** representation. However, how would you do this for the **`employee_nestset`** representation? Implement a function **`emp_reports(emp_id)`** that uses the **`employee_nestset`** representation to compute how many "direct reports" (i.e. immediate children) the specified employee has in the organization.

Important Note about Hierarchical Schemas!

The example schemas given for representing hierarchies are simply the minimal schemas necessary for representing the hierarchies. In general, a hierarchical schema could also include other values that would make specific operations against the hierarchy much easier. For example, each node could also specify its depth in the tree, a relative index among its siblings, etc. A single schema may even combine multiple hierarchy representations, if they are useful in different situations.

The main cost of including such additional information is simply maintaining it as the hierarchy changes structure. However, if structural changes occur infrequently and queries would benefit greatly from the additional information, it would well be worth it.