Programming Relational Databases

Introduction

In this assignment we're going to practice programming a database with functions, stored procedures and triggers. We'll provide you with SQL scripts you can use to import the database and data. You will then write several functions, stored procedures and triggers to automate some common tasks. We'll provide detailed examples and then ask you to write your own. Look for "***Now you try it***". The database we're using is described in the following UML class diagram. The learning objectives of this assignment are creating views, writing functions, stored procedures

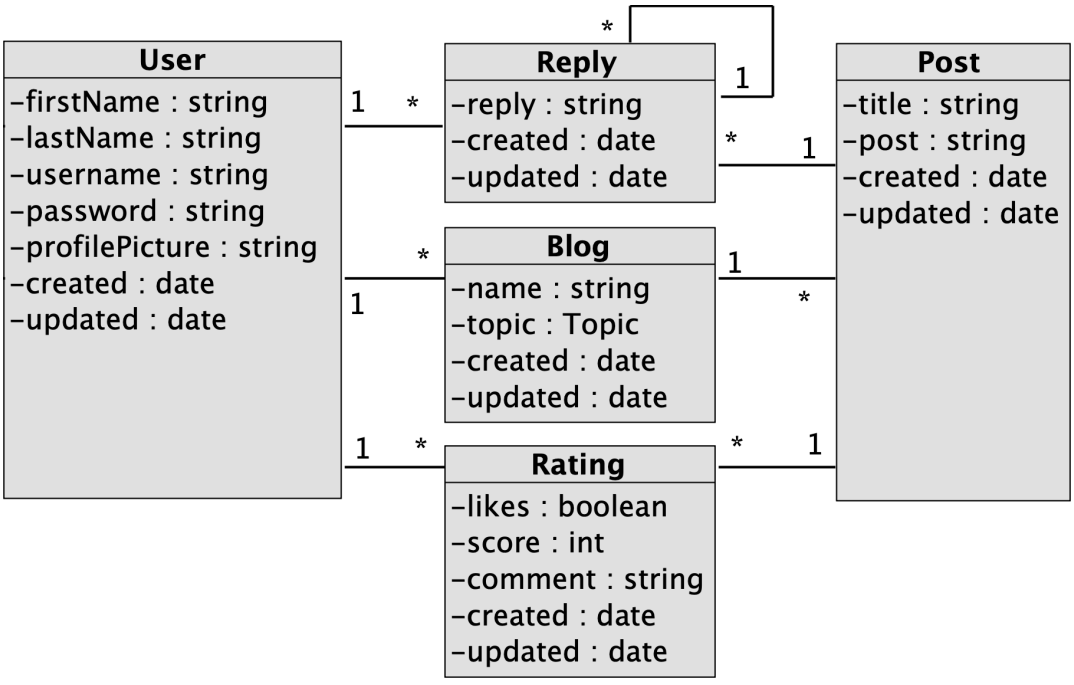
Importing a Database

In a previous assignment you created a schema called ***db_design***. Double click the schema ***db_design*** so that it is the default schema and all commands will be based on that schema. Download the data from my GitHub and import it into your ***db_design*** schema. Unzip the downloaded file into a folder. To import the data, from ***MySQL Workbench***, from the main menu, select ***Server***, ***Data Import***. In the ***Data Import*** screen, click the ***Import from Disk*** tab, and then click the ***Import from Dump Project Folder***. Click the browse button all the way on the right to select the directory you unzipped earlier. Select the folder and then click on ***Start Import***. Close the screen when done. Double click the ***db_design*** schema on the left hand side to make it the default schema.

Problem Statement

Given the UML class diagram below, consider the following two questions:

1. Which blogs are popular?
2. Which posts are popular?



These would be reasonable questions we should be able to answer with a social networking Web site where users are posting content to various blogs and rating each other's content. We could use such data to determine trends and suggest content to other users. We will be implementing SQL queries and functions to answer these two questions. We'll show you how to answer the first question, and then you can try answering the second question on your own. To complete this assignment, do all your work in these two files:

1. ***popular-blogs.sql*** - includes SQL queries we provide in this assignment
2. ***popular-posts.sql*** - includes SQL queries you write on your own to complete this assignment

Export the database so that it includes all tables, data, views, functions, and stored procedures and submit a zipped file that includes all files you worked on for this assignment.

Determining Popular Blogs

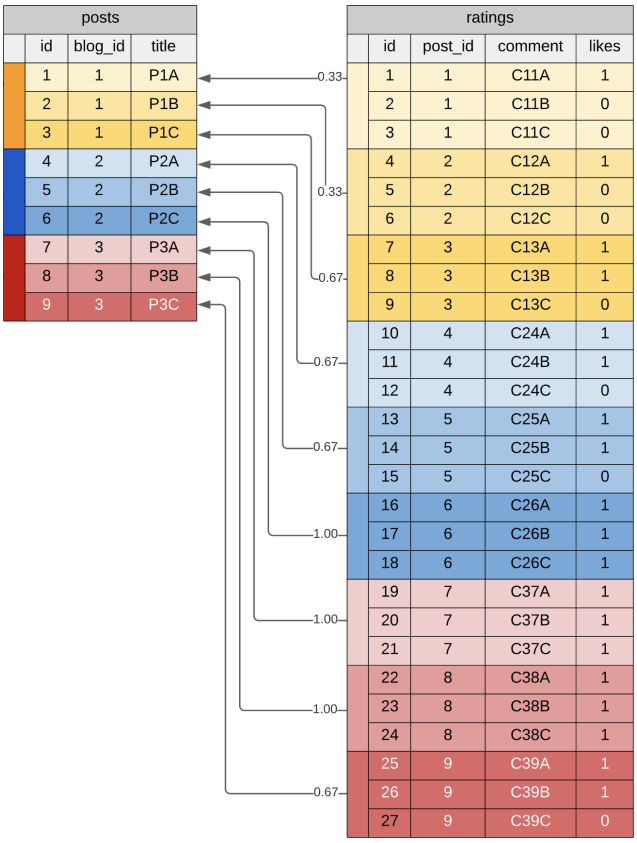
Let's consider the first question: "***Which blogs are popular?***" This question is somewhat vague since there's no criteria for popularity. Let's choose the criteria of popularity as ***above average number of likes***. We can now rewrite our question as

1. "Which blogs have an average number of likes higher than the average number of likes for all blogs?"

To answer this question we'll need to answer the following two related questions

A. "What is the average number of likes for a given blog?"
B. "What is the average number of likes for all blogs?"

If we can answer these two questions, then question (1) is just comparing which of (A) above are higher than (B).



Likes for each post

Average number of likes for a given blog

Analyzing the class diagram we realize that the ***likes*** are stored in the ***ratings*** table and they are for a given record in the ***posts*** table as illustrated here on the right

Since posts are then related to some blog, to figure out the average likes per blog, we first have to figure out what is the average likes per posts. We can then roll those up per blog.

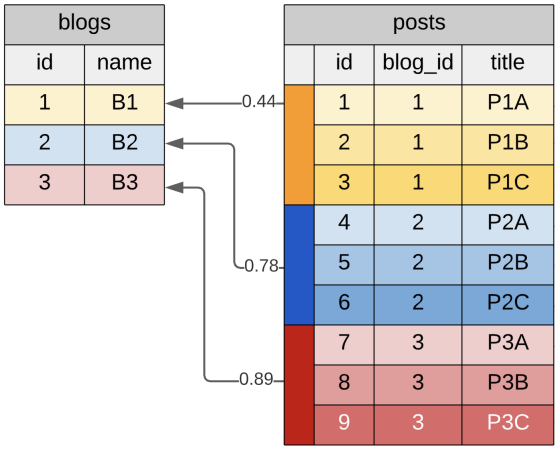
To determine popular posts we'll calculate the average likes per posts. We'll group all posts and average out all the likes. Here's a query that calculates the popularity of posts

```
-- average likes per post
SELECT r.post_id, AVG(likes) AS 'avg_likes_per_post'
FROM ratings r
GROUP BY r.post_id;
```

Since we need to roll up these popularity scores to their corresponding blogs, we'll wrap the query in a view so that we can use the results to calculate a blog's popularity. Here's the view

```
-- average likes per post -- VIEW
CREATE VIEW `avg_likes_per_post` AS
SELECT r.post_id, AVG(likes) AS 'avg_likes_per_post'
FROM ratings r
GROUP BY r.post_id;
```

Let's use the new view to roll up post popularity up to their blog by grouping the popularities by the blogs they belong to, e.g., grouping by the blog's ID as shown below




```
-- average likes per blog
SELECT b.id, AVG(alp.avg_likes_per_post) AS 'avg_likes_per_blog'
FROM blogs b, posts p, avg_likes_per_post alp
WHERE b.id=p.blog
AND p.id=alp.post_id
GROUP BY b.id;
```

Let's wrap the above query in a view so we can use it later

```
-- average likes per blog -- VIEW
CREATE VIEW `avg_likes_per_blog` AS
SELECT b.id AS 'blog_id', AVG(alp.avg_likes_per_post) AS 'avg_likes_per_blog'
FROM blogs b, posts p, avg_likes_per_post alp
WHERE b.id=p.blog AND p.id=alp.post_id
GROUP BY b.id;
```

With the ***avg_likes_per_blog*** we can now determine the average likes for a specific blog, e.g., blog with ID = 2

```
-- average likes for specific blog, e.g., blog_id=2
SELECT * FROM avg_likes_per_blog
WHERE blog_id=2;
```

Let's put the query in a function so we can parameterize it for any blog ID

```
CREATE FUNCTION avg_likes_for_blog (blog_id INT)
RETURNS float
BEGIN
  DECLARE avg_likes FLOAT;
  SELECT avg_likes_per_blog AS `avg_likes_for_blog`
  FROM avg_likes_per_blog alpb  WHERE alpb.blog_id=blog_id
   LIMIT 1  INTO avg_likes;
RETURN avg_likes;
END
```

This answers part A of our question:

"What is the average number of likes for a given blog?"

If you get the following error:

```
Operation failed: There was an error while applying the SQL script to the database.
ERROR 1418: This function has none of DETERMINISTIC, NO SQL, or READS SQL DATA in its declaration and binary logging is enabled (you *might* want to use the less safe log_bin_trust_function_creators variable)
```

You can fix this by executing the following from an SQL window:

```
SET GLOBAL log_bin_trust_function_creators = 1;
```

Average number of likes for all blogs

Let's now focus on part B of our question: "What is the average number of likes for all blogs?"
We can group ***avg_likes_per_blog*** and average out the likes per blog into a single average as follows:

```
-- average likes for all blogs
SELECT AVG(avg_likes_per_blog) AS 'avg_likes_all_blogs'
FROM avg_likes_per_blog alb;
```

We can wrap the query into a reusable function as follows

```
-- average likes for all blogs -- FUNCTION
CREATE FUNCTION `avg_likes_all_blogs` ()
RETURNS FLOAT
BEGIN
  DECLARE avg_likes FLOAT;
  SELECT AVG(avg_likes_per_blog) AS 'avg_likes_all_blogs'
  FROM avg_likes_per_blog alb INTO avg_likes;
  RETURN avg_likes;
END
```

This answers our part B of our question:

We can now use the two functions to answer our final question: Which blogs are popular? Which translates to which blogs have average likes greater than the average likes across all blogs:

```
-- blogs with higher likes than average, i.e., popular blogs
SELECT b.name, avg_likes_for_blog(b.id)
FROM blogs b
WHERE avg_likes_for_blog(b.id) > avg_likes_all_blogs();
```

Determining Popular Posts

Now you try it. Use the sample code we used to calculate popular blogs as a guide to calculate popular posts. Let's define popular posts as those posts with an average ***score*** higher than the average ***score*** across all posts. Create views and functions analogous to the ones we created. Write all your queries in a file called ***popular-posts.sql***. In the end you should have the following views and functions:

- ***avg_score_per_post*** - view containing average scores per post
- ***avg_score_for_post*** - function takes post ID as parameter and returns the average score for that post
- ***avg_score_all_posts*** - function calculates single float with average score across all posts

Stored Procedures

Stored procedures are a convenient way of encapsulating useful SQL scripts we can parameterize and reuse. Create, read, update and delete (CRUD) scripts come in handy to easily interact with the records in a table.

Create, Read, Update, and Delete Blogs (CRUD)

Let's create several CRUD stored procedures to interact with the blogs table. We'll create the following procedures:

- ***find_all_blogs*** - retrieves all the records in table blogs
- ***find_blog_by_id*** - retrieves a single record whose blog ID matches the parameter
- ***find_blog_for_user*** - retrieves a single record fromthe blogs table whose user_id matches the parameter
- ***create_blog_for_user*** - inserts a new record in the blogs table with the values in the parameters
- ***update_blog*** - updates an existing record for the blogs ID parameter setting new values based on the parameters
- ***delete_blog*** - deletes a record from the blogs table for the blogs ID parameter
- ***create_random_blogs*** - creates a number of random blogs

Here's the implementation of the ***find_all_blogs*** stored procedure

```
CREATE PROCEDURE `find_all_blogs` ()
BEGIN
  SELECT * FROM blogs;                    -- returns all the records in the blogs table
END
```

Here's the implementation of the ***find_blog_by_id*** stored procedure

```
CREATE PROCEDURE `find_blog_by_id` (blog_id INT)
BEGIN
  SELECT * FROM blogs WHERE blogs.id = blog_id;
END
```

Here's the implementation of the ***find_blogs_for_user*** stored procedure

```
CREATE PROCEDURE `find_blogs_for_user` (user_id INT)
BEGIN
  SELECT * FROM blogs WHERE blogs.user = user_id;
END
```

Here's the implementation of the ***create_blog_for_user*** stored procedure

```
CREATE PROCEDURE `create_blog_for_user` (
  user_id INT,
  blog_name VARCHAR(45),
  topic VARCHAR(45))
BEGIN
 INSERT INTO blogs (`user`, `name`, `topic`)
  VALUES (user_id, blog_name, topic);
END
```

Here's the implementation of the ***update_blog*** stored procedure

```
CREATE PROCEDURE `update_blog` (
  blog_id INT,
  new_name VARCHAR(45),
  new_topic VARCHAR(45))
BEGIN
 UPDATE blogs
  SET name=new_name, topic=new_topic
 WHERE blogs.id = blog_id;
END
```

Here's the implementation of the ***delete_blog*** stored procedure

```
CREATE PROCEDURE `delete_blog` (blog_id INT)
BEGIN
 DELETE FROM blogs WHERE blogs.id = blog_id;
END
```

Here's the implementation of the ***create_random_blogs*** stored procedure

```
CREATE PROCEDURE `create_random_blogs` (count INT, user_id INT)
BEGIN
 DECLARE i INT DEFAULT 0;
 DECLARE random_post_fix INT;
 DECLARE random_blog_name VARCHAR(45);

 WHILE i < count DO
  SET i = i + 1;
  SET random_post_fix = RAND() * 100;
  SET random_blog_name = CONCAT('RANDOM BLOG ', random_post_fix);
  INSERT INTO blogs (`name`, `topic`, `user`)
   VALUES   (random_blog_name, 'SPACE', user_id);
 END WHILE;
END;
```

Now You Try it

Using the stored procedures for interacting with the blogs table as guide, write similar store procedures to interact with the posts table. Implement the following store procedures:

- ***find_post_by_id*** - retrieves a single post record for a given post ID
- ***find_posts_for_blog*** - retrieves all post records for a given blog ID
- ***create_post_for_blog*** - creates a new post record for a given post title, post text, and blog ID
- ***update_post*** - updates an existing post's title and post's text, for a given post ID
- ***delete_post*** - removes an existing post record for a given post ID
- ***create_random_posts*** - creates a given number of random posts including a random title and random post text, for a given blog ID

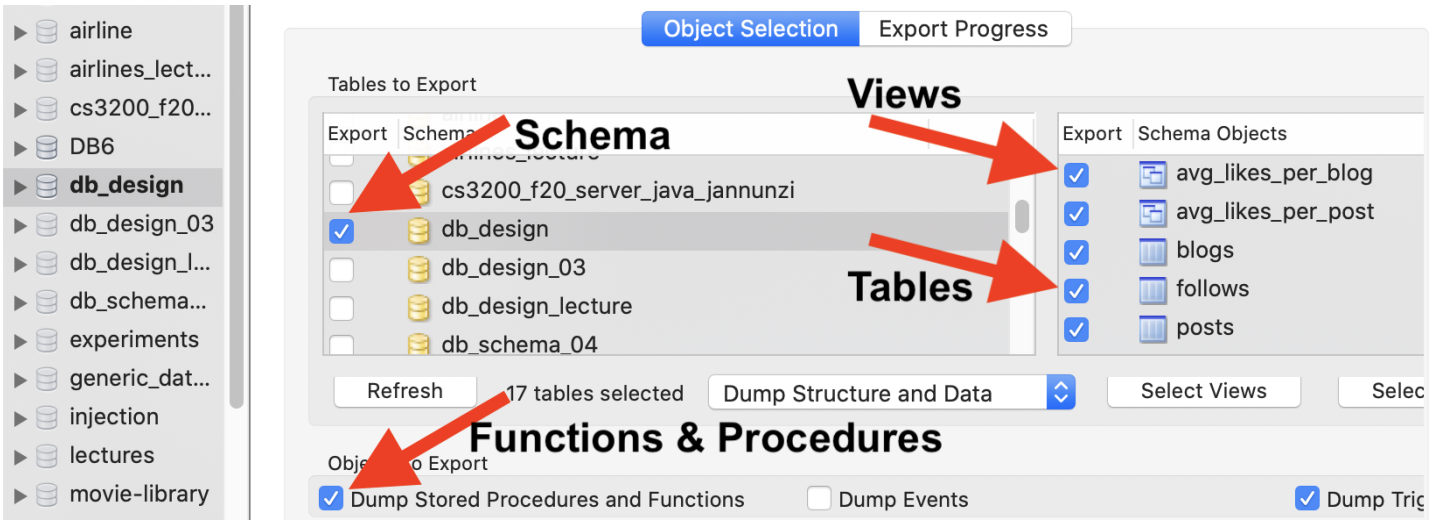
Deliverables

As a deliverable provide the following

1. ***popular-blogs.sql*** - contains SQL queries provided in the ***Determining Popular Blogs*** section. It does not include views and functions. These will be part of the database export listed below
2. ***popular-posts.sql*** - contains SQL queries you wrote yourself in the ***Determining Popular Posts*** section. It does not include views and functions. These will be part of the database export listed below
3. ***Database export*** - export all the tables, views, functions, and store procedures into an export directory, zip it up, and submit it along with the SQL files listed above.

To export the tables, views, functions, and store procedures follow these steps:

1. Double click the schema you've been working on, i.e., db_design
2. From the main menu select Server, Data Export
3. In the Data Export window shown beloa, select Object Selection, the schema (db_design), select all the tables and views, and the stored procedures and functions
4. Select Export to Dump Project Folder. Navigate to an alternate folder if necessary
5. Click Start Export
6. Wait until export is complete
7. Navigate to the folder listed in Export to Dump Project Folder to find the folder where all the files were exported to



On Canvas or Blackboard, submit the SQL file you've been working with throughout this assignment and a zip file containing the database.