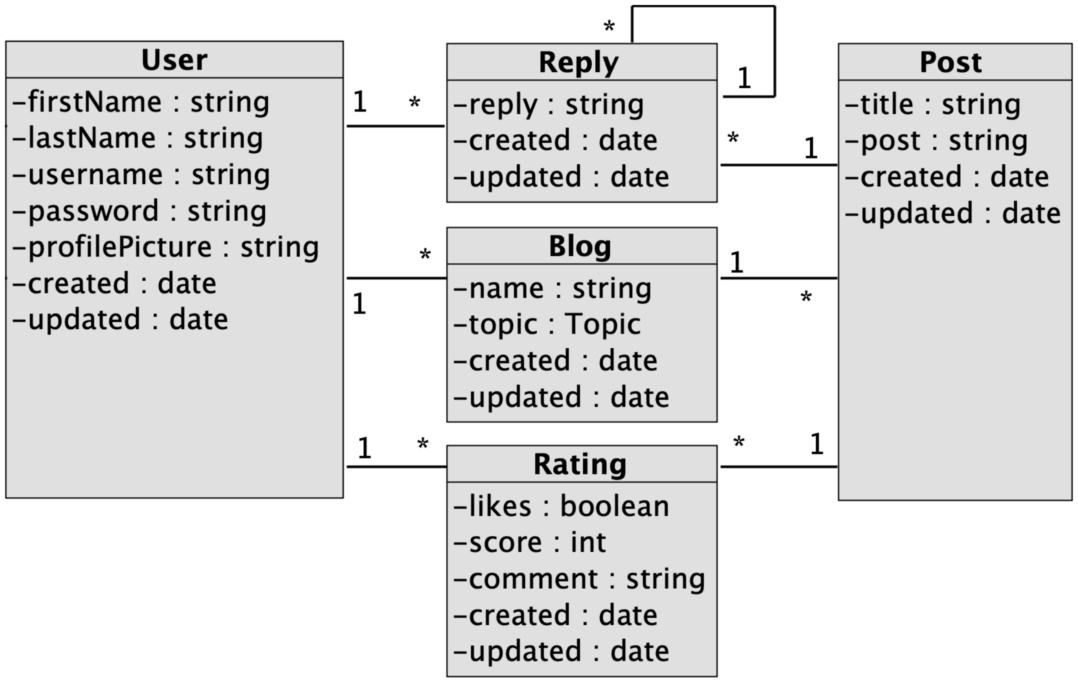
Querying Relational Databases

Introduction

In this assignment we're going to practice retrieving data from a database. We'll provide you with SQL scripts you can use to import the database and data. You will then write several SQL queries that will retrieve the data in various ways. We'll provide example queries 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.



Learning Objectives

- Importing databases
- Using ***compound primary keys***
- Implementing ***enumerated data types*** in SQL
- Retrieving data from relational databases with SQL's ***select*** statement
- Retrieving data from multiple tables by ***joining*** tables
- Grouping related records with SQL's ***group by*** clause

Importing a Database

In a previous assignment you've created a schema called ***db_design***. 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 and all commands will be based on that schema.

Retrieving All Records

Let's start with something simple. Let's retrieve all the records from the ***users*** table. In a new SQL screen, execute the following SQL code

```
-- select all records from the users table
SELECT * FROM users;
```

Notice the comment above the query starting with double dashes. Comments are useful to explain the intent of the query in plain language. Confirm all the users are displayed in the data grid below the query. ***Now you try it***. In the same SQL screen, a couple lines below the previous query, write SQL code that retrieves all the records from the ***blogs*** table. Add a comment above all your queries that explain the intent of the query, just like we did above. Make sure each SQL statement is terminated with a semicolon (";") and separated from other statements with at least one new line. To execute a specific SQL statement, highlight it with your mouse and click on the lightning bolt. Confirm that all the blogs are displayed. Save the SQL code in a file called ***querying-relational-databases.sql***. Implement the rest of the queries in the same SQL screen and save the file as you go. In the end the file should contain both the sample queries provided here as well as your own queries. Submit the file as the deliverable.

Retrieving a Record by its Primary Key

The previous queries retrieved all the records from a table. Let's now retrieve a single record knowing its primary key. In the same SQL screen you've been writing your queries, execute the following query that retrieves a blog record whose ***id*** is equal to 2. Confirm a single blog record displays in the ***Results Grid*** below the queries

```
-- select a blog record by its primary key
SELECT * FROM blogs
WHERE blogs.id = 2;
```

Now you try it. Write a SQL query that retrieves a record from the ***posts*** table whose ***id*** is equal to 1. Confirm that a single post record is displayed. Add a comment. Save the SQL file.

Retrieving Records by a Predicate

Instead of using the primary key in the ***WHERE*** clause, let's now use one of the other non key fields. Let's retrieve a user by their ***username***. Execute the following SQL in the same SQL screen. Confirm that a record from the users table is retrieved

```
-- select a user by their username
SELECT * FROM users
WHERE users.username = 'elonmusk';
```

Now you try it. Write a SQL query that retrieves a record from the ***blogs*** table whose name is ***'SpaceX'***. Confirm that a record from the blogs table is retrieved. Add a comment. Save the SQL file.

Joining Two Tables

All previous queries have been on a single table. Let's now retrieve records from two different tables that are related by their primary and foreign key values. We refer to this as ***joining*** tables. Execute the following SQL which retrieves all the ***usernames*** and their blog's ***name***

```
-- select usernames and their blog's name
SELECT users.username, blogs.name
FROM users JOIN blogs ON users.id = blogs.user;
```

Note that in the ***SELECT*** clause we are retrieving the ***username*** from the ***users*** table and the blog's ***name*** from the ***blogs*** table. We do this by retrieving records whose foreign keys match the primary keys of the corresponding records, i.e., we ***join the tables*** to find the records that are related. ***Now you try it***. Write SQL code that will retrieve all the blog names and the related posts' title in the related records in the posts table. Add a comment. Save the SQL file.

Implicit Joins

The queries above were using ***explicit join*** statements which can become hard to read when joining more than just two tables. Using ***implicit joins*** is much easier to read, which consists of moving the joining criteria to the ***WHERE*** clause as shown below. Let's rewrite the explicit joins above using an equivalent ***implicit join***. Execute the following implicit join which retrieves all the usernames and their blogs

```
-- select usernames and their blog's name
SELECT users.username, blogs.name
FROM users, blogs
WHERE users.id = blogs.user;
```

Confirm that you retrieve the same records as before. ***No you try it***. Write an implicit join that retrieves all the blog names and the related posts' titles. Add a comment. Save the SQL file.

Combining Implicit Joins and Non Key Fields

The previous example used keys in the WHERE clause to join tables. We can mix join criteria as well as non key criteria in the WHERE clause. Let's mix in an additional criteria in the ***WHERE*** clause. Execute the following SQL query which retrieves all the blogs from elonmusk

```
-- select usernames and their blog's name for a particular username
SELECT users.username, blogs.name
FROM users, blogs
WHERE users.id = blogs.user
AND users.username = 'elonmusk';
```

Confirm that all blogs belonging to ***elonmusk*** are displayed. ***Now you try it***. White SQL code to retrieve a blog's name and all the posts' titles from a blog whose name is ***'Everyday Astronaut'***. Confirm that all the posts are displayed from the Everyday Astronaut blog. Add a comment. Save the SQL file.

Joining Three Tables

All the exercises so far have joined two tables. Let's now try joining three tables. Let's retrieve the username, blog's name, and posts by 'erdayastronaut'. Execute the following query

```
-- select username, blog's name, and the blog's posts for a particular username
SELECT users.username, blogs.name, posts.post
FROM users, blogs, posts
WHERE users.id = blogs.user
AND blogs.id = posts.blog
AND users.username = 'erdayastronaut';
```

Confirm that all the posts from ***"erdayastronaut'*** are displayed. ***Now you try it***. Write an SQL statement that retrieves all the blog's names, posts, and replies to those posts for a blog whose name is 'Everyday Astronaut'. Confirm that the posts are displayed, add a comment and save the SQL file.

Aggregating Data with Group By

SQL's ***GROUP BY*** clause allows aggregating data by grouping records by fields. For instance, the following query groups the records in the ***ratings*** table by their ***post_id***. Once the groups are created, the ***likes*** are aggregated by adding them up with the ***sum()*** function. The resulting sum value is then labeled as ***'likes'***

```
-- calculate total likes per post
SELECT post_id, sum(ratings.likes) as `likes`
FROM ratings
GROUP BY ratings.post_id;
```

Execute the above query in the SQL screen and save the file. Confirm that the likes are added up. ***Now you try it***. Aggregate the scores in the ***ratings*** table by averaging their values using the ***avg()*** aggregation function. Label the resulting average as ***'scores'***. Use the query above as an example. Add a comment. Save the SQL file.

Inline Views

GROUP BY is very useful for aggregating records into single values, but they have their limitations. Since grouping occurs after the ***WHERE*** clause, but before the ***SELECT***, the individual values of the fields are lost. That means that the ***SELECT*** can only retrieve aggregated values of fields, and the fields being grouped. To get around this limitation we can use ***inline views***, which consists of including an inner query inside an outer query. The results of the inner query serves as a data source, as if it were a table, for the outer query. Here's an example where we use the ***GROUP BY*** statement used earlier embedded in another query (highlighed yellow)

```
-- retrieve blog's name, posts, and total likes per post
SELECT blogs.name, posts.post, inline_view.likes
FROM blogs, posts, (
  SELECT post_id, sum(ratings.likes) as `likes`
  FROM ratings
  GROUP BY ratings.post_id) as inline_view
WHERE inline_view.post_id = posts.id
AND posts.blog = blogs.id;
```

The inner query (highlighted yellow) is the same query we used earlier to calculate the total likes for each post. We name the resulting records as ***inline_view*** (any name will do). The ***inline_view*** can then be treated just like any other table as part of the SELECT, FROM, and WHERE clauses. In the ***FROM*** clause it's treated as a table from which to draw data from. In the WHERE clause it's used to join its ***post_id*** field withe ***posts.id***. In the SELECT clause it's used to retrieve the aggregated likes. ***Now you try it***. Use the query you wrote earlier to aggregate the average scores in from the ratings table. Name the results as ***inline_view*** like we did above. Display the blog's name, post's post, and the inline_view's aggregated score. Add a comment. Save the SQL file.

Challenge (required for graduates)

Consider the class diagram provided in the previous assignment. Using the relational schema you created, write SQL queries to answer the following questions:

1. What is the full name of the employee whose primary key is 123?
2. What are the names of the projects assigned to an employee whose primary key is 234?
3. What are the names of the tasks assigned to an employee whose primary key is 345 and project's primary key is 456?
4. How many projects is each employee assigned to?
5. How many tasks does each employee have?
6. What are the average number of tasks per project?
7. What is the average number of tasks for all employees across all projects?
8. Which employees have more tasks assigned to them than the average number of tasks per employee?

Save your queries in a file called ***challenge-db4-fa21.sql*** and submit it as part of your deliverables. Feel free to insert dummy data into the relational model you created to test your queries.

Deliverables

On Canvas or Blackboard, submit the SQL file you've been working with throughout this assignment: ***querying-relational-databases.sql***. It should include: retrieving all users and blogs, retrieving a single blog and post, retrieving a single user by username and single blog by name, joining the users and blogs tables, and joining the blogs and posts tables, using both explicit and implicit joins, combining joins and non key fields, joining three tables, aggregating with ***GROUP BY*** and inline views.