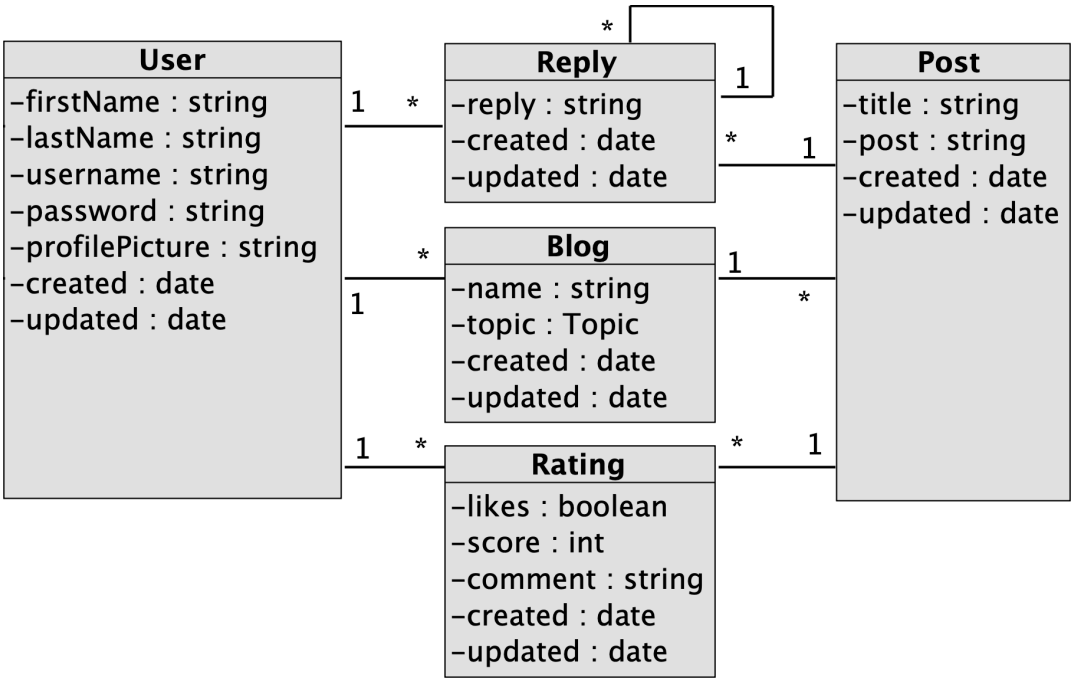
Updating Relational Databases

Introduction

In this assignment we're going to practice updating data in 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 update 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
- Inserting records into a relational database
- Updating records in a relational database
- Deleting records from a relational database

Importing a Database

In a previous assignment you've 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.

Inserting Into a Database

SQL's ***INSERT*** statement can be used to create or copy records into tables.

Inserting Specific Columns

The following syntax for the ***INSERT*** statement creates a new record in ***table_name*** where the columns listed right after of ***table_name*** are set to the values listed in the ***VALUES*** clause

INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);

Only the columns listed will be set to the values provided. If the table has other columns then they will be set to either ***NULL*** (no value) or some default value as configured by their ***DEFAULT*** clause. If a column is configured as ***NOT NULL***, and is not in the new record, then the ***INSERT*** statement will fail. The order of the columns match the order of the values, so ***column1*** gets ***value1***, ***column2*** gets ***value2***, etc. The relative order between the columns is not relevant.

Let's use this syntax to insert a couple of users. In a new SQL screen execute the following INSERT statements. Save the SQL in a file called ***updating-databases.sql***. Do all your work for this assignment in the same file and submit it as the deliverable.

```
-- find all users before inserting
SELECT * FROM users;

-- insert edward
INSERT INTO users (first_name, last_name, username, password)
VALUES ('Edward', 'Scissorhands', 'edward', 'p@ssw0rd');

-- insert frank
INSERT INTO users (first_name, last_name, username, password)
VALUES ('Frank', 'Herbert', 'frank', 'p@007rd');
```

Retrieve all the users to confirm the new users were inserted

```
-- find all users to confirm users were inserted
SELECT * FROM users;
```

Now you try it. In the same SQL screen, create two new blogs about ***space***: one called ***'The Angry Astronaut'*** and the other ***'NASA Space Flight'***. Both by ***Edward Scissorhands***. Save the SQL code into the same file ***updating-databases.sql***. Add comments before each statement describing them briefly as shown above.

Inserting Fields in Order

Another variation on the syntax of the ***INSERT*** statement allows omitting the columns as shown below

INSERT INTO table_name
VALUES (value1, value2, value3, ...);

So how does the ***INSERT*** statement know which columns to use? Which values map to which columns? The syntax relies on the order of declaration of the columns in the original ***CREATE TABLE*** statement. So if the ***CREATE STATEMENT*** was

```
CREATE TABLE table_name(
  columnA TYPE1,
  columnB TYPE2,
  columnC TYPE3,
  columnD TYPE4,
  columnE TYPE5
);
```

Then the new record would have ***columnA*** getting ***value1***, ***columnB*** getting ***value2***, and so forth. Note that in the example above ***columnD*** might not be getting a value. Not all columns need to get a value in the new record, but if a value configured as ***NOT NULL*** doesn't get a value, or doesn't have a ***DEFAULT***, the ***INSERT*** statement will fail. Also you can't skip a column, so say you want to provide values for ***columnA***, ***B***, and ***D***, but skip ***C***, then the following statement would fail because we skipped the value between ***value2*** and ***value3***:

```
INSERT INTO table_name
VALUES (value1, value2, , value3, ...);
```

Instead you would provide a ***NULL*** for columns with no values, if the column is not ***NOT NULL***

```
INSERT INTO table_name
VALUES (value1, value2, NULL, value3, ...);
```

Finally, if ***table_name***'s primary key is configured as ***AUTO_INCREMENT***, you can provide a ***NULL*** for the column and the database will calculate the correct value for the column.

Let's now use this new syntax to insert a couple of records into the blogs table. In the same SQL screen and file, ***updating-databases.sql***, execute the following ***INSERT*** statements

```
-- find all the blogs before inserting new ones
SELECT * FROM blogs;

-- insert a blog about tesla news
INSERT INTO blogs
VALUES (NULL, 'E for Electric', 'CARS', current_date(), current_date(), 6);

-- insert a blog about space
INSERT INTO blogs
VALUES (NULL, 'SpaceXcentric', 'SPACE', current_date(), current_date(), 6);

-- find all the blogs
SELECT * FROM blogs;
```

If the ID 6 above does not work for you, feel free to use a different ID. Note the use of ***current_date()*** to provide a default value for the ***created*** and ***updated*** columns. ***Now you try it***. Find some content about ***Tesla*** and

SpaceX on the Web and create a couple of ***posts*** for each of the two new ***blogs*** (four in total). Provide values for columns ***post***, ***created***, ***updated***, and ***blog*** (or ***blog_id***). You can leave the value of the ***id*** column as ***NULL*** so the database auto increments them. The two Tesla related posts should reference the ***'E for Electric'*** blog and the other two posts related to SpaceX should reference the ***'SpaceXcentric'*** blog. Write and save the SQL ***INSERT*** statements in the same SQL file you've been using throughout this assignment.

Inserting Records From Another Table (Copying)

The ***INSERT*** statement can be used to copy records from one table to another table using the following syntax:

INSERT INTO table2
SELECT * FROM table1
WHERE condition;

Here the content from ***table1*** is copied into ***table2***. The records in ***table1*** identified in the ***SELECT*** statement are inserted into ***table2***. To practice this we'll create a new table called ***unliked_posts*** and we'll copy those posts that have a value of 0 ***likes***. The ***unliked_posts*** table we'll need to have fields compatible with the table and columns we are copying from. We can decide which columns we want to copy. We'll just copy the original post ID, who made the post, and for which blog. Let's create the ***unliked_posts*** as shown below. Execute and save the following SQL in the same file you've been working throughout this assignment

```
-- create table unliked_posts to hold all unliked posts
DROP TABLE IF EXISTS `db_design`.`unliked_posts`;
CREATE TABLE `db_design`.`unliked_posts` (
 post_id INT NOT NULL,                                  -- references the original post record
 user_id INT NOT NULL,                                  -- references the user who made the post
 blog_id INT NOT NULL,                                  -- references the blog where the post appeared
 username VARCHAR(45),                                  -- the username of who made the post
 post VARCHAR(140),                                     -- the actual post text
 blog_name VARCHAR(45),                                 -- the name of the blog
 PRIMARY KEY(post_id, user_id, blog_id),                -- combination of post, user, and blog must be unique
 FOREIGN KEY (post_id) REFERENCES posts(id),            -- post_id references posts's primary key
 FOREIGN KEY (user_id) REFERENCES users(id),            -- user_id references users's primary key
 FOREIGN KEY (blog_id) REFERENCES blogs(id)             -- blog_id references blog's primary key
);
```

We'll find all the records for the unliked posts and then copy them to the ***unliked_posts*** table as shown below

```
-- find all unlinked posts and copy them to the unliked_posts table
INSERT INTO unliked_posts               -- insert the following records into unliked_posts
SELECT posts.id as post_id,             -- rename posts's id to post_id, as expected by unliked_posts
    users.id as user_id,                -- rename users's id to user_id, as expected by unliked_posts
    blogs.id as blog_id,                -- rename blogs's id to blog_id, as expected by unliked_posts
    users.username,                     -- username field already as expected by unliked_posts
    posts.post,                         -- post field already as expected by unliked_posts
    blogs.name as blog_name             -- rename blogs's name to blog_name, as expected by unliked_posts
FROM users, blogs, posts, ratings       -- join 4 tables to retrieve corresponding columns from each
WHERE users.id = ratings.user_id        -- join ratings with users
AND blogs.id = posts.blog               -- join blogs with posts
AND posts.id = ratings.post_id          -- join posts with replies
AND ratings.likes = 0;                  -- we only want the unliked posts
```

Now you try it. Create a table called ***liked_posts*** and populate it with the posts that are liked. Use a similar table definition as the ***unliked_posts*** table. Here's the table definition:

```
DROP TABLE IF EXISTS `db_design`.`liked_posts`;
CREATE TABLE `db_design`.`liked_posts` (
 post_id INT NOT NULL,
 user_id INT NOT NULL,
 blog_id INT NOT NULL,
 username VARCHAR(45),
 post VARCHAR(140),
 blog_name VARCHAR(45),
 PRIMARY KEY(post_id, user_id, blog_id),
 FOREIGN KEY (post_id) REFERENCES posts(id),
 FOREIGN KEY (user_id) REFERENCES users(id),
 FOREIGN KEY (blog_id) REFERENCES blogs(id)
);
```

Write an insert statement that populates the ***liked_posts*** table with posts that are liked. Execute and save the SQL in the same file you've been working all this assignment.

Updating Records in a Database

Let's now practice modifying, or updating, existing records. The UPDATE SQL statement is as follows:

UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;

Let's take a look at all the records in ratings for the post's whose ID is 1

```
-- get all the ratings for blog 1
SELECT *
FROM ratings
WHERE post_id=1;
```

Let's update the likes of a record whose id is 5. We'll change it from 0 to 1. If ID 1 above doesn't work for you, feel free to fix it to one that works.

```
-- update the likes for record 5
UPDATE ratings
SET likes=1
WHERE ratings.id=5;
```

If ID 5 above doesn't work for you, feel free to fix it to one that works. ***Now you try it***. Update the score for one of the records in the ratings table. Set the score to 3 for the record whose ID is 9. If ID 9 doesn't work for you, feel free to choose another ID.

Deleting Records from a Database

Finally let's practice removing data from a database. The ***DELETE*** statement shown below removes records from ***table_name*** which meet the ***condition*** in the ***WHERE*** clause.

DELETE FROM table_name
WHERE condition;

Let's remove all ratings from user 2

```
DELETE FROM ratings
WHERE user_id=2;
```

If ID 2 above doesn't work for you, feel free to fix it to one that works. ***Now you try it***. Remove all ratings whose score is less than or equal to 3. If IDs don't work for you, feel free to fix them to ones that works.

Deliverables

On Canvas or Blackboard, submit the SQL file you've been working with throughout this assignment: ***updating-databases.sql***. It should include the statements listed below. It is OK if you had to use different IDs than the ones shown here or used throughout the assignment.

1. Inserting edward
2. Inserting frank
3. Finding all users
4. Inserting 'The Angry Astronaut'
5. Inserting 'NASA Space Flight'
6. Inserting 'E for Electric'
7. Inserting 'SpaceXcentric'
8. Finding all blogs
9. Inserting two posts for the 'E for Electric' blog
10. Inserting two posts for the 'SpaceXcentric' blog
11. Creating the unliked_posts table
12. Copying unliked posts into the unliked_posts
13. Creating the liked_posts table
14. Copying liked posts into the liked_posts
15. Getting all the ratings for blog 1
16. Updating the likes for record 5
17. Updating the score for 9
18. Removing ratings from user 2
19. Removing ratings with scores less than or equal to 3