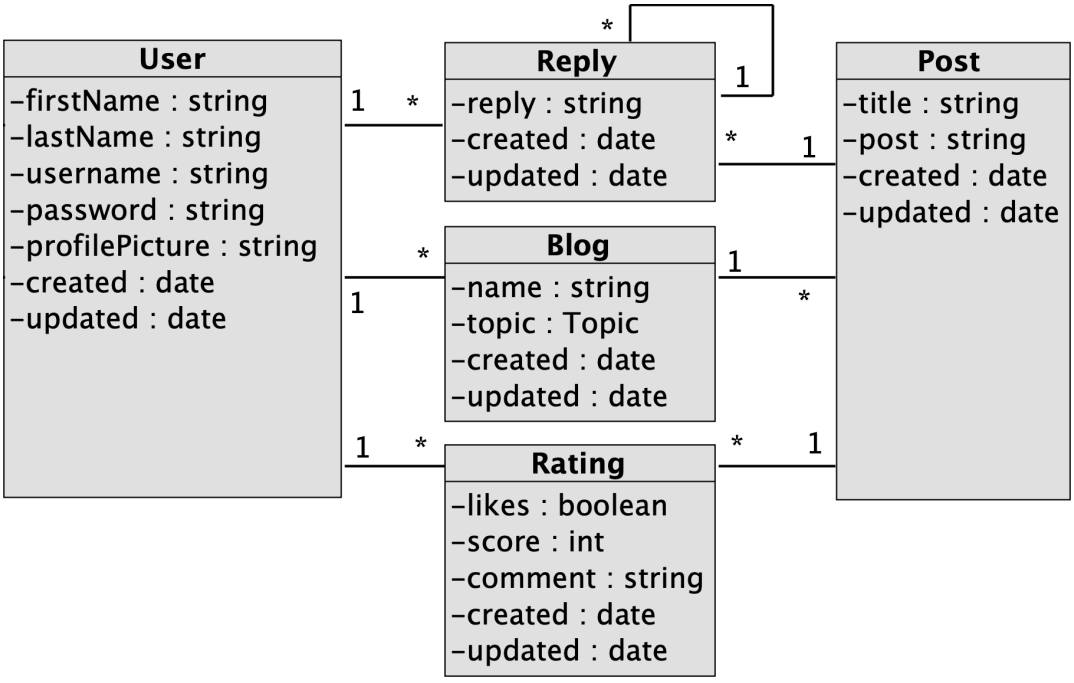
Creating Relational Databases

Introduction

In the previous assignment we designed a database you might use to implement a blogging or social networking application. We used ***UML*** (***Unified Modeling Language***) class diagrams to describe the datatypes and the relations between them. In particular we described classes ***User***, ***Blog***, ***Post***, ***Rating*** and ***Reply*** as shown below.



In this assignment we'll focus on converting this class diagram into an equivalent ***relational model*** using ***SQL***'s (***Structured Query Language***) ***DDL*** (***Data Definition Language***). We'll also add an additional class -- ***Follow*** -- to represent users following each other. Finally we'll add a self reference in the Reply class to represent users replying to other replies.

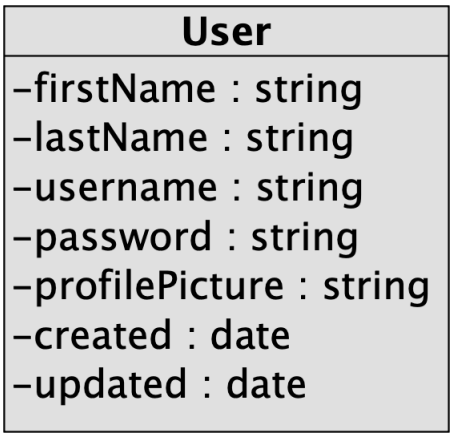
Learning objectives

- Converting conceptual data definitions documented in UML as relational models implemented with SQL
- Use SQL's CREATE statements to create tables
- Use Primary and Foreign keys to implement one to many and many to many relationships
- Use self reference to implement hierarchical data structures

Creating Tables

Let's start by creating a table that represents the users of the application: users. The class diagram describing the data related to users is shown below. You might have already created a table from previous assignments so we're going to remove that table and recreate it here. You should only have at most three (3) records in there so it shouldn't be too hard to recreate the data.

To remove a table use the ***DROP TABLE*** command and then create the table. In a new SQL window execute the following code. Save the code in a file called ***create-users-table.sql***

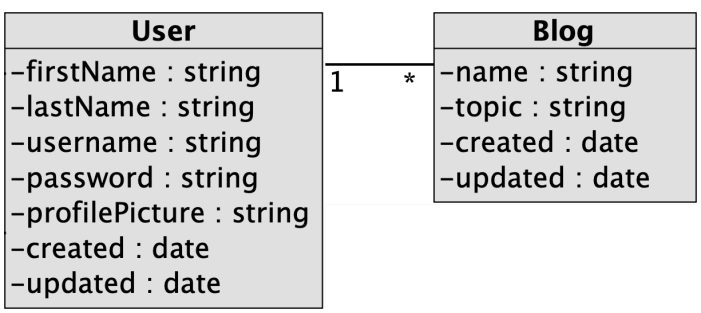



```
DROP TABLE IF EXISTS `db_design`.`users`;
CREATE TABLE `db_design`.`users` (
 `id` INT NOT NULL AUTO_INCREMENT,
 `first_name` VARCHAR(45) NULL,
 `last_name` VARCHAR(45) NULL,
 `username` VARCHAR(45) NOT NULL,
 `password` VARCHAR(45) NOT NULL,
 `profile_picture` VARCHAR(45) NULL,
 `created` DATETIME NULL DEFAULT CURRENT_TIMESTAMP,
 `updated` DATETIME NULL DEFAULT CURRENT_TIMESTAMP
    ON UPDATE CURRENT_TIMESTAMP,
 PRIMARY KEY (`id`));
```

A few notes on the ***User*** class diagram and its equivalent ***users*** table. First note that class names are typically *capitalized singular nouns*, e.g., ***User***, whereas in ***SQL*** table names are plural nouns, e.g., ***users***, so the ***User*** class becomes the ***users*** table. We could choose the table names as all caps or all lowercase, but we'll use lowercase in this course (more on casing in a bit). Also note that the table definition includes a field ***id*** which has no equivalent in the class diagram. This is because representing unique records in SQL is not enforced by default and needs explicit declaration of a field configured as ***UNIQUE*** or ***PRIMARY KEY***. ***UML*** is based on object oriented design where the instance objects are implicitly considered to be unique so an additional ***id*** field would be redundant. It would be acceptable to include the ***id*** field anyway in the UML class diagram especially if we want to make it clear that there's an ***SQL*** equivalent field. We have chosen ***not to include*** it in our UML class diagrams in this course and it is ***strongly discouraged*** unless otherwise stated. The fields ***first_name***, ***last_name***, and ***profile_picture*** are each composed of several words and we use the ***underscore*** ("_") to separate the words. Note that we did not use the camelcase naming convention we used in the equivalent class diagram where we did not use the underscore and instead capitalized each word except the first one, e.g., ***firstName***. The reason we did not use the same naming convention in the SQL code is that SQL is ***case insensitive*** whereas UML is ***case sensitive***, so that in SQL ***firstName***, ***firstname***, ***FIRSTNAME***, and ***fIrStNaMe*** are all consider the same as far as SQL is concerned. Therefore we choose to use underscores to separate the words that make up a field name. Finally note that we have declared the SQL ***id*** field as a ***PRIMARY KEY***. This enforces the ***id*** field as unique, and readies it to participate in relationships with other tables defined later. Finally note we've added fields ***created*** and ***updated*** which don't appear in the UML diagram, but help keep a history of when records were originally created or updated since. We'll do this with pretty much every table.

Implementing One to Many Relationships

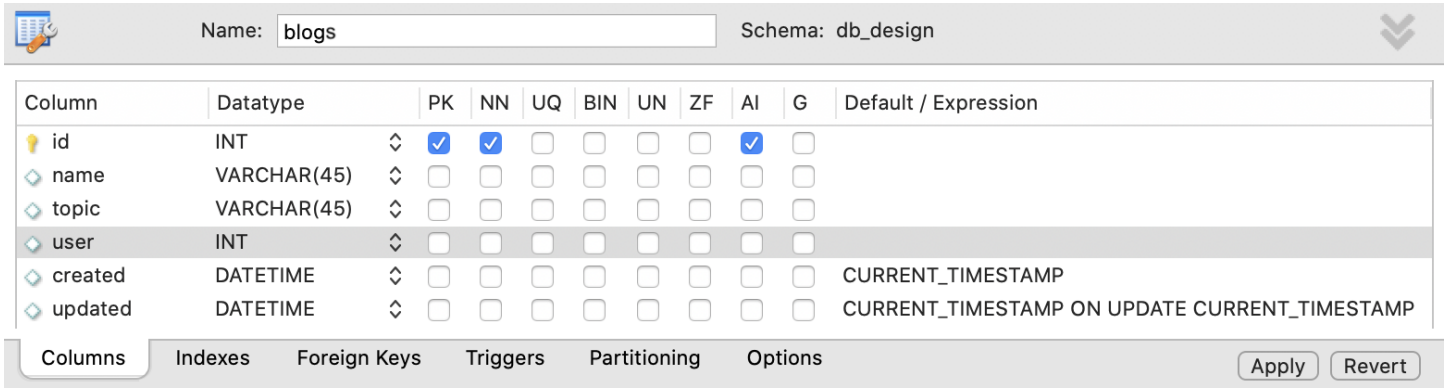
Creating the *blogs* table



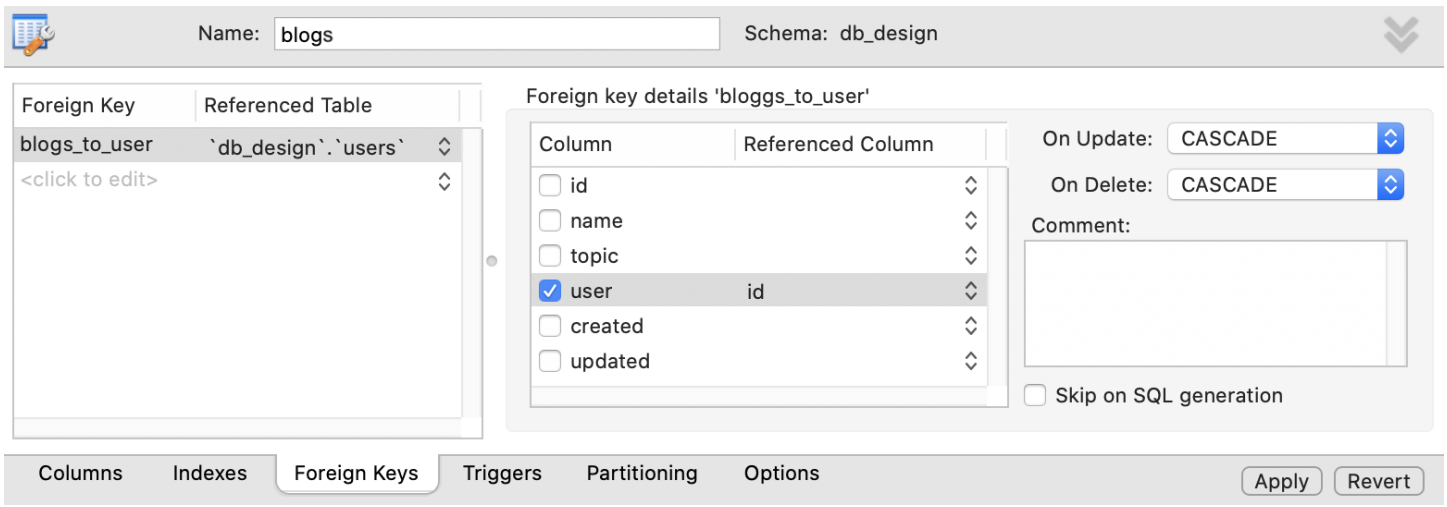
Let's now create the ***one to many*** relation between the ***User*** and ***Blog*** class. In SQL, relations are implemented with ***foreign keys*** referencing the ***primary key*** of other tables. In our case we want to establish that a user instance can be associated with many blog instances. We implement this by declaring a field in the many side (Blog) that references the ***id*** field on the one side (User). In a new SQL window execute the following:

```
DROP TABLE IF EXISTS `db_design`.`blogs`;                  -- removes blogs table if it already exists
CREATE TABLE `db_design`.`blogs` (                         -- declares the blogs table in db_design schema
 `id` INT NOT NULL AUTO_INCREMENT,                         -- unique identifier declared as PRIMARY KEY later
 `name` VARCHAR(45) NULL,                                  -- name of the blog declared as a string in UML
 `topic` VARCHAR(45) NULL,                                 -- topic of the blog declared as a string in UML
 `user_id` INT NULL,                                       -- user field will refer to blog's user's id
 `created` DATETIME NULL                                   -- records when record was created
           DEFAULT CURRENT_TIMESTAMP,
 `updated` DATETIME NULL                                   -- records when record was updated
           DEFAULT CURRENT_TIMESTAMP
           ON UPDATE CURRENT_TIMESTAMP,
 PRIMARY KEY (`id`),                                       -- constrains id field as unique
 INDEX `blogs_to_user_idx` (`user_id` ASC),                -- index on user field for faster retrieval
 CONSTRAINT `blogs_to_user`                                -- naming the constraint/configuration
   FOREIGN KEY (`user_id`)                                 -- declaring user field as foreign key
   REFERENCES `db_design`.`users` (`id`)                   -- that references the id field in the users table
   ON DELETE CASCADE                                       -- remove this blog if the user is removed
   ON UPDATE CASCADE);                                     -- update foreign key if referenced id field changes
```

Save the code in a file called ***create-blogs-table.sql***. Note that alternatively you could have created the table using Workbench's user interface as follows. ***SKIP*** this if you already know how to create tables using the Workbench user interface. The following is not required to complete the assignment. Clicking on the create new table icon opens the table definition screen below. Add fields id, name, topic, user, created, and updated with the data types shown below.



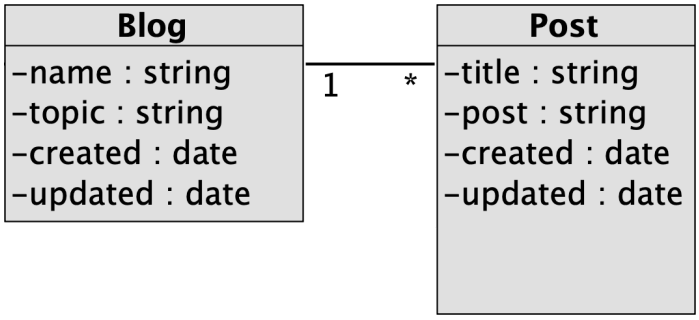
Add ***Default/Expression*** for ***created*** and ***updated*** fields as shown above. Click on ***Foreign Keys*** to configure the relation between the blogs table and the users table as shown below



Under the ***Foreign Key*** column provide a unique name, e.g., ***blogs_to_user***. Under the ***Referenced Table*** select the table being referenced, e.g., ***users***. Under the ***Column*** column, select the field referencing the ***users*** table, e.g., ***user***. Under the ***Referenced Column*** column select the field in the users table that the user field references, e.g., ***id***. Finally select ***CASCADE*** for ***On Update*** and ***On Delete*** so that both the record is removed if the referenced record is removed or the user field is updated if the referenced field is updated.

Creating the *posts* table

Now you try it. Having walked you through creating a ***one to many*** relationship, create a ***one to many*** relationship on your own between the ***Blog*** class and the ***Post*** class shown here on the right. Save the resulting SQL code in a file called ***create-posts-table.sql***

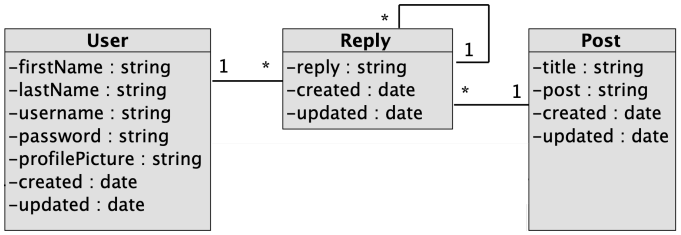


Implementing Many to Many Relationships

Let's now explore how to create a ***many to many*** relationship. The ***Reply*** class establishes a many to many relationship between the ***User*** and ***Post*** classes as shown below. Let's implement this with a ***replies*** table.

Creating the *replies* table

The ***replies*** table will capture a user's reply to a post. We'll need know who made the reply and the post the reply was made to. We'll capture this by recording the IDs of the related user and post records. In a new SQL screen execute the following code and save it to a file called ***create-replies-table.sql***




```
DROP TABLE IF EXISTS `db_design`.`replies`;          -- removes the table if it already exists
CREATE TABLE `db_design`.`replies` (                 -- creates the table
 `id` INT NOT NULL AUTO_INCREMENT,                   -- declares the unique identifier field
 `reply` VARCHAR(45) NULL,                           -- this is the main content
 `user_id` INT NULL,                                 -- references other record related to this record
 `post_id` INT NULL,                                 -- references other record related to this record
 `created` DATETIME NULL                             -- when was this record created
    DEFAULT CURRENT_TIMESTAMP,
 `updated` DATETIME NULL                             -- when was this record last modified
    DEFAULT CURRENT_TIMESTAMP
    ON UPDATE CURRENT_TIMESTAMP,
 PRIMARY KEY (`id`),                                 -- configures unique identifier as primary
 INDEX `replies_2_users_idx` (`user_id` ASC),        -- creates index for faster lookup
 INDEX `replies_2_posts_idx` (`post_id` ASC),        -- creates index for faster lookup
 CONSTRAINT `replies_2_users`
   FOREIGN KEY (`user_id`)                           -- configures reference field user_id
   REFERENCES `db_design`.`users` (`id`)             -- to reference a field in another table
   ON DELETE CASCADE                                 -- removes this record if referenced record removed
   ON UPDATE CASCADE,                                -- updates this field if referenced field updates
 CONSTRAINT `replies_2_posts`
   FOREIGN KEY (`post_id`)                           -- configures reference field post_id
   REFERENCES `db_design`.`posts` (`id`)             -- to reference a field in another table
   ON DELETE CASCADE                                 -- removes this record if referenced record removed
   ON UPDATE CASCADE);                               -- updates this field if referenced field updates
```

Creating self references

When a user replies to a post, other users should be able to reply to that reply, and then other users can reply to that reply to the reply. This allow users to enter into a threaded conversation. This recursive, or hierarchical data structure can be implemented with records that reference other records in the same table. We captured this in the UML class diagram for the ***Reply*** class as a self reference association. That's the loop at the top right corner in the ***Reply*** class. To implement this in SQL we could have added an additional field and constraint such as

```
...
`replies_id` INT NULL
 CONSTRAINT `replies_2_replies`
  FOREIGN KEY (`replies_id`)
  REFERENCES `db_design`.`replies` (`id`)
  ON DELETE CASCADE
  ON UPDATE CASCADE;
...
```

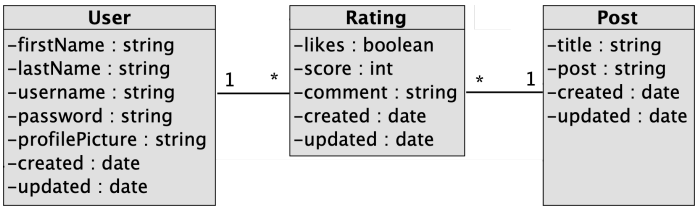
The reason we didn't do this was that the constraint was referencing a table that didn't yet exists, i.e., replies. To implement self referencing we're going to modify (alter) the table we just created using SQL's ***ALTER TABLE*** statement. In a new SQL screen, execute the code below and save it in ***alter-replies-add-self-reference.sql***

```
ALTER TABLE `db_design`.`replies`                -- modify existing table
ADD COLUMN `replies_id` INT NULL AFTER `post_id`, -- add new field after some existing field
ADD INDEX `replies_2_replies_idx` (`replies_id` ASC); -- creates index for faster retrieval

ALTER TABLE `db_design`.`replies`                -- modify existing table
ADD CONSTRAINT `replies_2_replies`
 FOREIGN KEY (`replies_id`)                      -- configures reference field replies_id
 REFERENCES `db_design`.`replies` (`id`)         -- to reference a field in the same table
 ON DELETE CASCADE                               -- removes this record if referenced record removed
 ON UPDATE CASCADE;                              -- updates this field if referenced field updates
```

Creating the *ratings* table

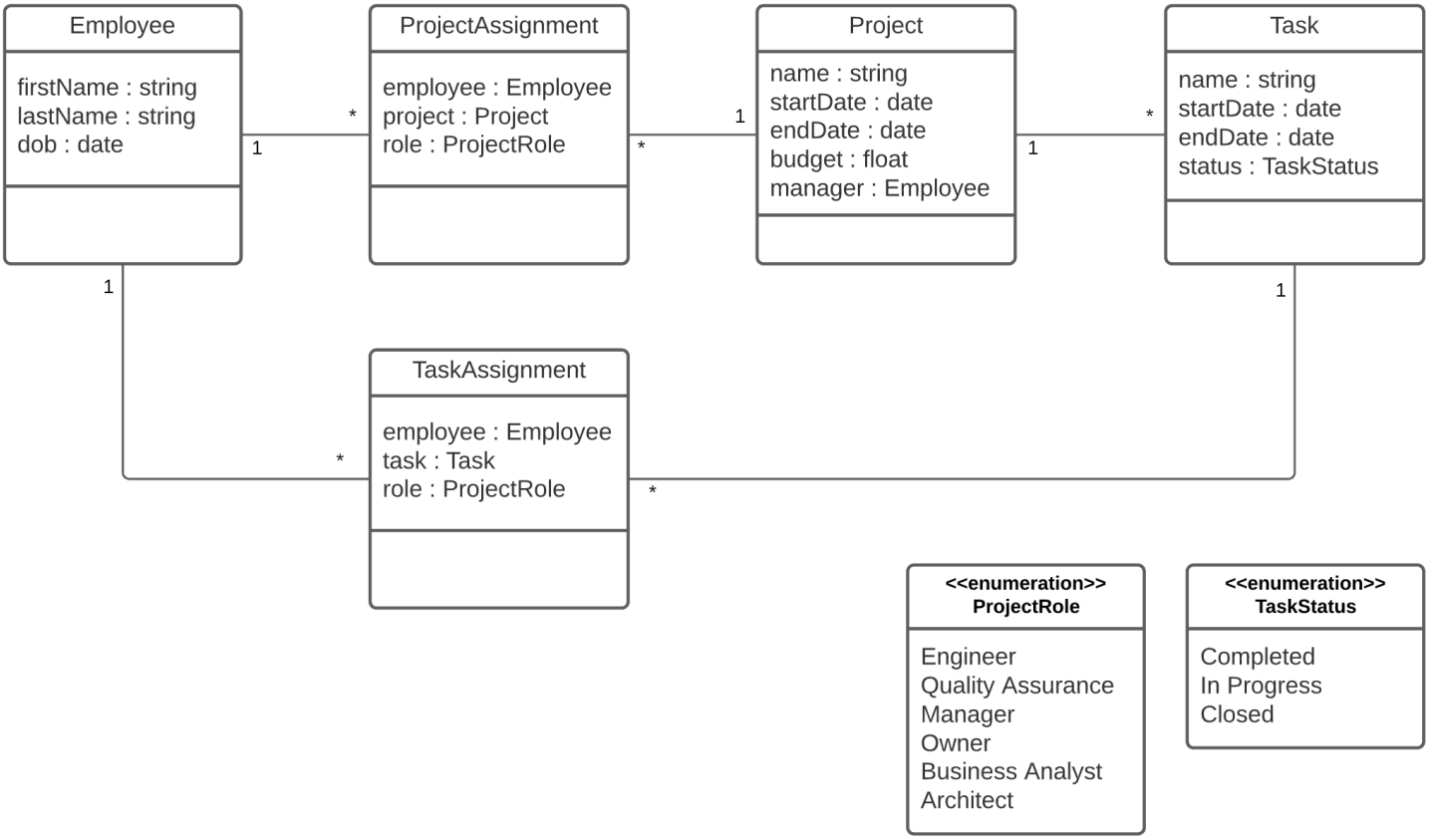
Now that we've walked you through creating a ***many to many*** relationship, create a ***many to many*** relationship on your own between the ***User*** class and the ***Post*** class shown here on the right. Save the resulting SQL code in a file called ***create-ratings-table.sql***. The ***ratings*** table has no self reference, so no need to alter.



Challenge (required for graduates)

The following are a set of practice sections exploring some additional database design topics. They are optional or extra credit for undergraduate students, but are required for graduate students. For extra credit policy reach out to your instructor.

Consider the following class diagram



Using SQL, implement an equivalent relational model in WorkBench or an equivalent MySQL client. Export the database and submit the resulting files.

Deliverables

On Canvas or Blackboard, submit all the SQL files generated throughout this assignment:

1. create-users-table.sql
2. create-blogs-table.sql
3. create-posts-table.sql
4. create-ratings-table.sql
5. alter-replies-add-self-reference.sql
6. create-replies-table.sql

Graduate students will submit additional SQL files that implements the relational model equivalent to the class diagram provided.