Udiddit, a social news aggregator

Introduction

Udiddit, a social news aggregation, web content rating, and discussion website, is currently using a risky and unreliable Postgres database schema to store the forum posts, discussions, and votes made by their users about different topics.

The schema allows posts to be created by registered users on certain topics, and can include a URL or a text content. It also allows registered users to cast an upvote (like) or downvote (dislike) for any forum post that has been created. In addition to this, the schema also allows registered users to add comments on posts.

Here is the DDL used to create the schema:

```
CREATE TABLE bad_posts (
      id SERIAL PRIMARY KEY,
      topic VARCHAR(50),
      username VARCHAR(50),
      title VARCHAR(150),
      url VARCHAR(4000) DEFAULT NULL,
      text_content TEXT DEFAULT NULL,
      upvotes TEXT,
      downvotes TEXT
);

CREATE TABLE bad_comments (
      id SERIAL PRIMARY KEY,
      username VARCHAR(50),
      post_id BIGINT,
      text_content TEXT
);
```

Part I: Investigate the existing schema

As a first step, investigate this schema and some of the sample data in the project's SQL workspace. Then, in your own words, outline three (3) specific things that could be improved about this schema. Don't hesitate to outline more if you want to stand out!

1. There should be a Users table. The current schema is unfortunately not normalized which is due to the lack of this Users table. There should be an ID associated with each username and should also be unique.
2. The lack of normalization means that there are no foreign keys. In order for the data to be normalized, we not only need to split up the schema into multiple tables, but there should be some foreign key relationships to relate to the main table. For example, a comment can't exist without a post.
3. Upvotes and downvotes should definitely not be of type TEXT. They should be of type INTEGER. A user should be able to select between +1 or -1. Under the current schema, a user can set anything in the upvotes and downvotes fields when they should specifically be INTEGER instead.

Part II: Create the DDL for your new schema

Having done this initial investigation and assessment, your next goal is to dive deep into the heart of the problem and create a new schema for Udiddit. Your new schema should at least reflect fixes to the shortcomings you pointed to in the previous exercise. To help you create the new schema, a few guidelines are provided to you:

1. Guideline #1: here is a list of features and specifications that Udiddit needs in order to support its website and administrative interface:
    a. Allow new users to register:
        i. Each username has to be unique
        ii. Usernames can be composed of at most 25 characters
        iii. Usernames can't be empty
        iv. We won't worry about user passwords for this project
    b. Allow registered users to create new topics:
        i. Topic names have to be unique.
        ii. The topic's name is at most 30 characters
        iii. The topic's name can't be empty
        iv. Topics can have an optional description of at most 500 characters.
    c. Allow registered users to create new posts on existing topics:
        i. Posts have a required title of at most 100 characters
        ii. The title of a post can't be empty.
        iii. Posts should contain either a URL or a text content, **but not both**.
        iv. If a topic gets deleted, all the posts associated with it should be automatically deleted too.
        v. If the user who created the post gets deleted, then the post will remain, but it will become dissociated from that user.
    d. Allow registered users to comment on existing posts:
        i. A comment's text content can't be empty.
        ii. Contrary to the current linear comments, the new structure should allow comment threads at arbitrary levels.
        iii. If a post gets deleted, all comments associated with it should be automatically deleted too.
        iv. If the user who created the comment gets deleted, then the comment will remain, but it will become dissociated from that user.
        v. If a comment gets deleted, then all its descendants in the thread structure should be automatically deleted too.
    e. Make sure that a given user can only vote once on a given post:
        i. Hint: you can store the (up/down) value of the vote as the values 1 and -1 respectively.
        ii. If the user who cast a vote gets deleted, then all their votes will remain, but will become dissociated from the user.

iii. If a post gets deleted, then all the votes for that post should be automatically deleted too.

2. Guideline #2: here is a list of queries that Udiddit needs in order to support its website and administrative interface. Note that you don't need to produce the DQL for those queries: they are only provided to guide the design of your new database schema.
    a. List all users who haven't logged in in the last year.
    b. List all users who haven't created any post.
    c. Find a user by their username.
    d. List all topics that don't have any posts.
    e. Find a topic by its name.
    f. List the latest 20 posts for a given topic.
    g. List the latest 20 posts made by a given user.
    h. Find all posts that link to a specific URL, for moderation purposes.
    i. List all the top-level comments (those that don't have a parent comment) for a given post.
    j. List all the direct children of a parent comment.
    k. List the latest 20 comments made by a given user.
    l. Compute the score of a post, defined as the difference between the number of upvotes and the number of downvotes

3. Guideline #3: you'll need to use normalization, various constraints, as well as indexes in your new database schema. You should use named constraints and indexes to make your schema cleaner.

4. Guideline #4: your new database schema will be composed of five (5) tables that should have an auto-incrementing id as their primary key.

Once you've taken the time to think about your new schema, write the DDL for it in the space provided here:

```
-- Part II
-- Drop any of the new tables we propose if they exist
DROP TABLE IF EXISTS
"users",
"topics",
"posts",
"comments",
"votes";
```

```sql
-- a. Allow new users to register:
/* Design Guidelines:
   - The ID should autoincrement and should be unique upon insertion
   - The username should be at most 25 characters and should not be NULL
   - We should be able to record when they last logged in
*/
CREATE TABLE "users"
(
id SERIAL PRIMARY KEY,
username VARCHAR(25) NOT NULL,
last_login TIMESTAMP,
CONSTRAINT "unique_usernames" UNIQUE ("username"),
CONSTRAINT "non_empty_username" CHECK (LENGTH(TRIM("username")) > 0)
);
-- b. Allow registered users to create new topics:
/* Design guidelines
   - Each topic should have a unique ID and autoincrement upon insertion
   - The name of the topic should be at most 30 characters and is not null
   - The description of the topic should be at most 500 characters
*/
CREATE TABLE "topics"
(
id SERIAL PRIMARY KEY,
name VARCHAR(30) NOT NULL,
description VARCHAR(500),
CONSTRAINT "unique_topics" UNIQUE ("name"),
CONSTRAINT "non_empty_topic_name" CHECK (LENGTH(TRIM("name")) > 0)
);
-- c. Allow registered users to create new posts on existing topics:
/* Design guidelines
   - Each post should have a unique ID and should autoincrement upon insertion
   - The title of the post should be at most 100 characters and is not null
   - We should be able to record when the post was created
   - The URL should consist of at most 400 characters
   - The post should reference a particular topic we are discussing, and so we
     should have a foreign-key / primary-key relationship where the topic ID
     references the ID of the topics table.
   - If the topic is deleted from the topics table, we should remove all posts
```

```
     that share the same topic.
   - The post should correspond to the user who authored it, and so we should
     have a foreign-key / primary-key relationship where the user ID references
     the ID of the users table.
   - If the user is deleted from the users table, we should still keep the
     comments from this user, so we should simply set the user ID to null in
     the posts table.
   - There is an additional intricacy where we can either have a URL or we can
     have text content.  We can only have one or the other, so we must set up
     constraints to check to see if we have either a URL that's part of the
     post, or text content itself.  We can check for one or the other by
     checking the length of the fields to ensure that at most only one of them
     has a non-zero length.
*/
CREATE TABLE "posts"
(
id SERIAL PRIMARY KEY,
title VARCHAR(100) NOT NULL,
created_on TIMESTAMP,
url VARCHAR(400),
text_content TEXT,
topic_id INTEGER REFERENCES "topics" ON DELETE CASCADE,
user_id INTEGER REFERENCES "users" ON DELETE SET NULL,
CONSTRAINT "non_empty_title" CHECK (LENGTH(TRIM("title")) > 0),
CONSTRAINT "url_or_text" CHECK (
  (LENGTH(TRIM("url")) > 0 AND LENGTH(TRIM("text_content")) = 0) OR
  (LENGTH(TRIM("url")) = 0 AND LENGTH(TRIM("text_content")) > 0)
)
);
-- Set up an index so that we can quickly search for posts by URL.
CREATE INDEX ON "posts" ("url" VARCHAR_PATTERN_OPS);

-- d. Allow registered users to comment on existing posts:
/* Design guidelines:
   - Each comment should have a unique ID and should autoincrement upon
     insertion
   - We should not have any empty comments
   - We should be able to track when the comment was created
```

```
   - When commenting, we should know which post we commented on as well as
     which user wrote the comment
   - The comment should correspond to the post it was written to, so we should
     have a foreign-key / primary-key relationship where the post ID references
     the ID of the posts table.
   - Should the post get deleted (removed from the posts table), we should
     delete all comments associated with this post.
   - The comment should also correspond to the user who authored it, so we
     should have a foreign-key / primary-key relationship where the user ID
     references the ID of the users table.
   - Should the user get deleted (removed from the users table), we should be
     able to keep all of the comments, but simply set the author of the
     comments to null
*/
CREATE TABLE "comments"
(
id SERIAL PRIMARY KEY,
text_content TEXT NOT NULL,
created_on TIMESTAMP,
post_id INTEGER REFERENCES "posts" ON DELETE CASCADE,
user_id INTEGER REFERENCES "users" ON DELETE SET NULL,
parent_comment_id INTEGER REFERENCES "comments" ON DELETE CASCADE
CONSTRAINT "non_empty_text_content" CHECK(LENGTH(TRIM("text_content")) > 0)
);
-- e. Make sure that a given user can only vote once on a given post:
/* Design guidelines:
   - Each vote should have a unique ID and should autoincrement upon insertion
   - We should keep track of which user voted for which post
   - The vote should either be +1 for upvote or -1 for downvote and should
     never be null
   - The vote should also correspond to the user who voted for the post, so we
     should have a foreign-key / primary-key relationship where the user ID
     references the ID of the users table.
   - Should we delete the user from the users table, we should still keep the
     votes made by this user intact, so simply set the user ID in the votes
     table to null
   - There should only be one vote per user for a post, so to ensure this we
     will enforce a unique composition of user ID and post ID so that we should
```

```
    not see more than one vote for any one user and any one post
*/
CREATE TABLE "votes"
(
id SERIAL PRIMARY KEY,
user_id INTEGER REFERENCES "users" ON DELETE SET NULL,
post_id INTEGER REFERENCES "posts" ON DELETE CASCADE,
vote SMALLINT NOT NULL,
CONSTRAINT "vote_plus_or_min" CHECK("vote" = 1 OR "vote" = -1),
CONSTRAINT "one_vote_per_user" UNIQUE (user_id, post_id)
);
```

Part III: Migrate the provided data

Now that your new schema is created, it's time to migrate the data from the provided schema in the project's SQL Workspace to your own schema. This will allow you to review some DML and DQL concepts, as you'll be using INSERT…SELECT queries to do so. Here are a few guidelines to help you in this process:

1. Topic descriptions can all be empty
2. Since the bad_comments table doesn't have the threading feature, you can migrate all comments as top-level comments, i.e. without a parent
3. You can use the Postgres string function **regexp_split_to_table** to unwind the comma-separated votes values into separate rows
4. Don't forget that some users only vote or comment, and haven't created any posts. You'll have to create those users too.
5. The order of your migrations matter! For example, since posts depend on users and topics, you'll have to migrate the latter first.
6. Tip: You can start by running only SELECTs to fine-tune your queries, and use a LIMIT to avoid large data sets. Once you know you have the correct query, you can then run your full INSERT…SELECT query.
7. **NOTE**: The data in your SQL Workspace contains thousands of posts and comments. The DML queries may take at least 10-15 seconds to run.

Write the DML to migrate the current data in bad_posts and bad_comments to your new database schema:

```
-- Part III
/* To insert the usernames from the bad schema into the good schema, we should
  have a look at all of the usernames from the bad posts, bad comments,
  as well as all of the users who ever voted for any posts, both upvotes
  and downvotes and insert all of them into the new users database.  This can
  be achieved by concatenating all of the users together into a single column
  and choosing the unique ones from this column.
*/
INSERT INTO "users"("username")
 SELECT DISTINCT username
 FROM bad_posts
 UNION
 SELECT DISTINCT username
 FROM bad_comments
 UNION
```

```
 SELECT DISTINCT regexp_split_to_table(upvotes, ',')
 FROM bad_posts
 UNION
 SELECT DISTINCT regexp_split_to_table(downvotes, ',')
 FROM bad_posts;

/* To insert the topics from the bad schema into the good schema, simply look
  at all of the topics from the bad posts table and put them into the good
  posts table - make sure the topics are all distinct (no duplicates).
*/
INSERT INTO "topics"("name")
SELECT DISTINCT topic FROM bad_posts;

/* To insert into the new posts table, we will identify the user ID and topic
  ID from the users and topic tables after we join them with the current
  bad posts so that we can consolidate all of the information into one
  temporary table.  This will allow us to uniquely identify which user and
  which topic we should assign for each post, and we will select the first
  100 characters of the title, followed by the URL or the text content and
  we will insert all of these fields into the posts table.  Note that in order
  to join successfully, what will be common between the users table and
  posts table are the usernames, and what's common between the topics table
  and the posts table are the topic names.  The IDs do not exist in the
  old schema.
*/
INSERT INTO "posts"
(
 "user_id",
 "topic_id",
 "title",
 "url",
 "text_content"
)

SELECT
 users.id,
 topics.id,
 LEFT(bad_posts.title, 100),
 bad_posts.url,
```

```
 bad_posts.text_content
FROM bad_posts
JOIN users ON bad_posts.username = users.username
JOIN topics ON bad_posts.topic = topics.name;

/* To insert into the comments table, we need to identify which user provided
  a comment on which post, so we must collect this information from the bad
  comments table, but we will need to identify which user and which post
  the comment came from, thus we join based on both the username and the
  post id from the old schema.  Thankfully we do have post ids from the old
  schema
*/
INSERT INTO "comments"
(
 "post_id",
 "user_id",
 "text_content"
)

SELECT
 posts.id,
 users.id,
 bad_comments.text_content
FROM bad_comments
JOIN users ON bad_comments.username = users.username
JOIN posts ON posts.id = bad_comments.post_id;

/* To insert into the votes table, we must do this for both the upvotes and
  downvotes separately.  Recall that only one user and one vote can be assigned
  to one post, so first let's take a look at all of the users who upvoted for
  those posts.  We will need to build a subquery which consists of just the
  users who upvoted, but note that the list of upvoters for a post is comma
  separated so use regexp_split_to_table to provide multiple users assigned
  to a single post.  Once we have this, take the users from the upvoted
  list and join with the usernames from our current schema, then insert
  their corresponding post ids, user ids and their upvotes.
*/
INSERT INTO "votes"
(
```

```
 "post_id",
 "user_id",
 "vote"
)

SELECT t1.id, users.id,
1 AS vote_up
FROM (SELECT id, REGEXP_SPLIT_TO_TABLE(upvotes,',')
 AS upvote_users FROM bad_posts) t1
JOIN users ON users.username=t1.upvote_users;

/* Do the same but for downvotes
*/
INSERT INTO "votes"
(
 "post_id",
 "user_id",
 "vote"
)

SELECT t1.id, users.id,
-1 AS vote_down
FROM (SELECT id, REGEXP_SPLIT_TO_TABLE(downvotes,',')
 AS downvote_users FROM bad_posts) t1
JOIN users ON users.username=t1.downvote_users;
```