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. The first thing that would be done is to normalize the data from the two tables.

   In the "bad_comments" table the "username" would be an **id** with reference to a **users** table. With that, we can better manage the posts with bad comments.

```
CREATE TABLE bad_comments (
      id SERIAL PRIMARY KEY,
      user_id INTEGER,
      post_id BIGINT,
      text_content TEXT
);
```

   For the "bad_posts", something similar to the example above would be necessary in order to normalize the data. "Topics" and "username" columns could reference **topics** and **users** tables respectively, just referencing their ids. Also, if a bad_post could be classified with different kinds of topic types (sport, tech, music, etc.), a new table must be created to manage that variety.

   Lastly, for the "up" or "down" votes I would suggest something different like a **BOOLEAN** column. If a user hits like (**up**), then it assumes 1 or true; otherwise, it assumes 0 or false for dislike (**down**) votes.

```
CREATE TABLE bad_posts (
      id SERIAL PRIMARY KEY,
      topic_id INTEGER,
      user_id INTEGER,
      title VARCHAR(150),
      url VARCHAR(4000) DEFAULT NULL,
      text_content TEXT DEFAULT NULL
);

CREATE TABLE bad_posts_votes (
      id SERIAL PRIMARY KEY,
      user_id INTEGER,
      bad_post_id INTEGER,
      vote BOOLEAN
      PRIMARY KEY (user_id, bad_post_id)
);
```

2. The second one, the **users** table suggested above would have **UNIQUE** usernames in order to prevent duplicated ones.
3. Third and last one, a post would have a "title" and a "topic" assigned to it, they must need a **NOT NULL** constraint.

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:

```
/*
    Student name: Mathaus Vila Nova
    LinkedIn: https://www.linkedin.com/in/mathausvilanova/
    Course: Udacity SQL Nanodegree Program
```

```
      Udiddit Project - New Table Schema
*/

CREATE TABLE users (
      id SERIAL PRIMARY KEY,
      username VARCHAR(25) NOT NULL UNIQUE CHECK(LENGTH(TRIM("username")) >
0),
      register_date DATE NOT NULL DEFAULT CURRENT_DATE,
      last_login_date DATE DEFAULT NULL
);

CREATE TABLE topics (
      id SERIAL PRIMARY KEY,
      topic_name VARCHAR(30) NOT NULL UNIQUE CHECK(LENGTH(TRIM("topic_name"))
> 0),
      topic_desc VARCHAR(500) DEFAULT NULL,
      created_by INTEGER,
      CONSTRAINT "user_id_fk"
            FOREIGN KEY ("created_by")
                  REFERENCES users("id") ON DELETE SET NULL
);

CREATE TABLE posts (
      id SERIAL PRIMARY KEY,
      title VARCHAR(100) NOT NULL CHECK(LENGTH(TRIM("title")) > 0),
      url VARCHAR(4000) DEFAULT NULL,
      content TEXT DEFAULT NULL,
      topic_id INTEGER,
      created_by INTEGER,
      CONSTRAINT "post_url_or_content"
            CHECK (
                  (NULLIF (TRIM("url"), '') IS NULL OR NULLIF
(TRIM("content"), '') IS NULL)
                  AND
                  NOT (NULLIF (TRIM("url"), '') IS NULL AND NULLIF
(TRIM("content"), '') IS NULL)
            ),
      CONSTRAINT "topic_id_fk"
            FOREIGN KEY ("topic_id")
                  REFERENCES topics("id") ON DELETE CASCADE,
      CONSTRAINT "user_id_fk"
            FOREIGN KEY ("created_by")
```

```
                  REFERENCES users("id") ON DELETE SET NULL
);

CREATE TABLE post_comments (
      id SERIAL PRIMARY KEY,
      comment TEXT NOT NULL CHECK(LENGTH(TRIM("comment")) > 0),
      post_id INTEGER,
      parent_comment_id INTEGER DEFAULT NULL,
      created_by INTEGER,
      CONSTRAINT "post_id_fk"
            FOREIGN KEY ("post_id")
                  REFERENCES posts("id") ON DELETE CASCADE,
      CONSTRAINT "post_comment_id_fk"
            FOREIGN KEY ("parent_comment_id")
                  REFERENCES post_comments("id") ON DELETE CASCADE,
      CONSTRAINT "user_id_fk"
            FOREIGN KEY ("created_by")
                  REFERENCES users("id") ON DELETE SET NULL
);

CREATE TABLE post_votes (
      id SERIAL PRIMARY KEY,
      vote BOOLEAN,
      post_id INTEGER,
      created_by INTEGER,
      UNIQUE (post_id,created_by),
      CONSTRAINT "post_id_fk"
            FOREIGN KEY ("post_id")
                  REFERENCES posts("id") ON DELETE CASCADE,
      CONSTRAINT "user_id_fk"
            FOREIGN KEY ("created_by")
                  REFERENCES users("id") ON DELETE SET NULL
);

/* Indexes */

CREATE INDEX "post_url_idx" ON posts("url");
```

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:

```
/*
      Student name: Mathaus Vila Nova
      LinkedIn: https://www.linkedin.com/in/mathausvilanova/
      Course: Udacity SQL Nanodegree Program

      Udiddit Project - Data Migration
*/

/* 1. Populating users table */
INSERT INTO users (username) (
      (SELECT DISTINCT username FROM bad_posts WHERE username IS NOT NULL)
      UNION
      (SELECT DISTINCT username FROM bad_comments WHERE username IS NOT NULL)
      UNION
      (SELECT DISTINCT regexp_split_to_table(downvotes, ',') AS username FROM
```

```
bad_posts)
      UNION
      (SELECT DISTINCT regexp_split_to_table(upvotes, ',') AS username FROM
bad_posts)
      ORDER BY username
);

/* 2. Populating topics table */
INSERT INTO topics (topic_name) (
      SELECT DISTINCT INITCAP(bp.topic) AS topic_name
      FROM bad_posts bp
      ORDER BY 1
);

/* 3. Populating posts table */
INSERT INTO posts (title, url, content, topic_id, created_by) (
      SELECT
            CONCAT(LEFT(bp.title, 97), '...') AS title,
            bp.url AS url,
            bp.text_content AS content,
            tp.id AS topic_id,
            u.id AS created_by
      FROM
            bad_posts bp
            LEFT JOIN topics tp ON tp.topic_name = INITCAP(bp.topic)
            LEFT JOIN users u ON u.username = bp.username
);

/* 4. Populating post_comments table */
INSERT INTO post_comments (comment, post_id, created_by) (
      SELECT
            bc.text_content AS comment,
            bc.post_id AS post_id,
            u.id AS created_by
      FROM
            bad_comments bc
            INNER JOIN users u ON u.username = bc.username
            INNER JOIN posts p ON p.id = bc.post_id
);

/* 5. Populating post_votes table */
INSERT INTO post_votes (vote, post_id, created_by) (
      /* SELECT statement considering 'dislikes' votes */
```

```
      (SELECT
            t1.vote AS vote,
            t1.post_id AS post_id,
            u.id AS created_by
      FROM (
            SELECT
                  p.id AS post_id,
                  regexp_split_to_table(bp.downvotes, ',') AS username,
                  false AS vote
            FROM
                  bad_posts bp
                  LEFT JOIN posts p ON p.title = CONCAT(LEFT(bp.title, 97),
'...')
      ) AS t1
      LEFT JOIN users u ON t1.username = u.username)

      UNION

      /* SELECT statement considering 'likes' votes */
      (SELECT
            t1.vote AS vote,
            t1.post_id AS post_id,
            u.id AS created_by
      FROM (
            SELECT
                  p.id AS post_id,
                  regexp_split_to_table(bp.upvotes, ',') AS username,
                  true AS vote
            FROM
                  bad_posts bp
                  LEFT JOIN posts p ON p.title = CONCAT(LEFT(bp.title, 97),
'...')
      ) AS t1
      LEFT JOIN users u ON t1.username = u.username)
);
```