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. A `username` field exists in both tables, but there is no user table and no primary keys associated with the username. Primary keys would ensure each username is uniquely identifiable and allow a username to be referenced across multiple tables. Without these constraints, more than one user can use the same username and data redundancies are likely.

2. In the `bad_comments` table, the `post_id` value is of data type `BIGINT` and likely should be changed to `INT` to save disk space as the number of posts have not exceeded the range of values allowed by `INT`.

3. In the `bad_posts` table, the `upvotes` and `downvotes` values are stored as text. While the values themselves are a list of usernames tied to votes, there are no constraints that point these to usernames (as there is also no `users` table), so the values could be anything.

4. In the `bad_posts` table, there is no requirement that a post features a `topic`, `username`, `title`, or `text_content`, all of which should be required to submit a post. The lack of requirements could result in empty posts that contain no data or metadata.

5. In the `bad_posts` table, the data featured in the `upvotes` and `downvotes` fields are lists of names rather than single names. This fails to conform to normalized data standards and should be normalized or converted into a count of upvotes and downvotes depending on the requirements.

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:

```
-----------------------------------------
-- USERS
-----------------------------------------
-- Create `users` table
CREATE TABLE users (
   id SERIAL PRIMARY KEY,
   username VARCHAR(25) NOT NULL UNIQUE,
   last_login TIMESTAMP
```

```
    -- Add constraint
    CONSTRAINT "username_not_empty" CHECK (LENGTH(TRIM(username)) >= 0)
);

-- Create index on username
CREATE INDEX idx_users_username ON users(username);


----------------------------------------
-- TOPICS
----------------------------------------
-- Create `topics` table
CREATE TABLE topics (
    id SERIAL PRIMARY KEY,
    name VARCHAR(30) UNIQUE NOT NULL,
    description VARCHAR(500)
    -- Add constraint
    CONSTRAINT "topic_name_not_empty" CHECK (LENGTH(TRIM(name)) >= 0)
);

-- Create index on topic names
CREATE INDEX idx_topics_name ON topics(name);


----------------------------------------
-- POSTS
----------------------------------------
-- Create `posts` table
CREATE TABLE posts (
    id SERIAL PRIMARY KEY,
    title VARCHAR(100) NOT NULL,
    url VARCHAR(4000) DEFAULT NULL,
    text_content TEXT DEFAULT NULL,
    topic_id INT NOT NULL,
    user_id INT DEFAULT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    -- Add constraints
    CONSTRAINT check_url_or_text CHECK (
        (url IS NOT NULL and text_content IS NULL) OR
```

```
        (url IS NULL and text_content IS NOT NULL)
    ),
    -- Add constraint
    CONSTRAINT "post_title_not_empty" CHECK (LENGTH(TRIM(title)) >= 0)
    -- Add foreign key references
    FOREIGN KEY (topic_id) REFERENCES topics(id) ON DELETE CASCADE,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE SET NULL
);

-- Create index on post URL
CREATE INDEX idx_posts_url ON posts(url);
-- Create index on post topic_id
CREATE INDEX idx_posts_topic_id ON posts(topic_id);
-- Create index on post user_id
CREATE INDEX idx_posts_user_id ON posts(user_id);

----------------------------------------
-- COMMENTS
----------------------------------------
-- Create `comments` table
CREATE TABLE comments (
    id SERIAL PRIMARY KEY,
    text_content TEXT NOT NULL,
    post_id INT NOT NULL,
    user_id INT DEFAULT NULL,
    parent_comment_id INT DEFAULT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    -- Add constraint
    CONSTRAINT "comment_text_not_empty" CHECK (LENGTH(TRIM(text_content)) >= 0),
    -- Add foreign key references
    FOREIGN KEY (post_id) REFERENCES posts(id) ON DELETE CASCADE,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE SET NULL,
    FOREIGN KEY (parent_comment_id) REFERENCES comments(id) ON DELETE CASCADE
);

-- Create index on comment post_id
CREATE INDEX idx_comments_post_id ON comments(post_id);
```

```sql
-- Create index on comment user_id
CREATE INDEX idx_comments_user_id ON comments(user_id);
-- Create index on comment parent_comment_id
CREATE INDEX idx_comments_parent_comment_id ON comments(parent_comment_id);

----------------------------------------
-- VOTES
----------------------------------------
-- Create `votes` table
CREATE TABLE votes (
   id SERIAL PRIMARY KEY,
   post_id INT NOT NULL,
   user_id INT DEFAULT NULL,
   vote_value INT NOT NULL,
   -- Add constraints
   CONSTRAINT "vote_value_valid" CHECK (vote_value IN (-1, 1)),
   CONSTRAINT "vote_limit" UNIQUE (post_id, user_id),
   -- Add foreign key references
   FOREIGN KEY (post_id) REFERENCES posts(id) ON DELETE CASCADE,
   FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE SET NULL
);

-- Create index on vote post_id
CREATE INDEX idx_votes_post_id ON votes(post_id);
-- Create index on vote user_id
CREATE INDEX idx_votes_user_id ON votes(user_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:

```
-- Populate `users` table with existing data
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;
```

```sql
-- Populate `topics` table with existing data
INSERT INTO topics ("name")
   SELECT DISTINCT topic
   FROM bad_posts;

-- Populate `posts` table with existing data
INSERT INTO posts ("id", "title", "url", "text_content", "topic_id", "user_id")
   SELECT
       bp.id,
       LEFT(bp.title, 100), -- Truncate to fit if necessary
       bp.url,
       bp.text_content,
       t.id,
       u.id
   FROM bad_posts AS bp
   INNER JOIN users AS u
   ON bp.username = u.username
   INNER JOIN topics AS t
   ON bp.topic = t.name;

-- Populate `comments` table with existing data
INSERT INTO comments ("text_content", "post_id", "user_id")
   SELECT
       bc.text_content,
       bc.post_id,
       u.id
   FROM bad_comments AS bc
   INNER JOIN users AS u
   ON bc.username = u.username;

-- Populate `votes` table with converted upvotes data
INSERT INTO votes ("post_id", "user_id", "vote_value")
   SELECT
       bp.id,
       u.id,
       1 AS upvote
   FROM (SELECT
             "id",
             REGEXP_SPLIT_TO_TABLE("upvotes", ',') AS upvote
```

```
        FROM bad_posts) bp
   JOIN users AS u
   ON bp.upvote = u.username;

-- Populate `votes` table with converted downvotes data
INSERT INTO votes ("post_id", "user_id", "vote_value")
   SELECT
       bp.id,
       u.id,
       -1 AS downvote
   FROM (SELECT
             "id",
             REGEXP_SPLIT_TO_TABLE("downvotes", ',') AS downvote
         FROM bad_posts) bp
   JOIN users AS u
   ON bp.downvote = u.username;
```