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. Normalization and Data Redundancy
Issue: The current schema may lead to data redundancy, particularly with user information. For instance, the username is stored in both the bad_posts and bad_comments tables. If a user's information changes, it would require updates in multiple places, increasing the risk of inconsistencies.
Improvement: Normalize the schema by creating a separate users table to store user information. This table would include unique usernames and any other relevant user details. Other tables can then reference this user table via foreign keys, ensuring data consistency and reducing redundancy.
2. Data Type Optimization
Issue: The upvotes and downvotes fields are currently defined as TEXT, which is inefficient for storing numeric values. This can lead to performance issues and complicate calculations for vote counts.
Improvement: Change the upvotes and downvotes fields to INTEGER types. This allows for efficient storage and enables straightforward arithmetic operations to calculate total votes. Additionally, consider using a separate votes table to track individual votes, which would allow for more detailed analytics and prevent users from voting multiple times on the same post.
3. Lack of Referential Integrity
Issue: The bad_comments table references post_id as a BIGINT, but there is no foreign key constraint linking it to the bad_posts table. This can lead to orphaned comments if a post is deleted.
Improvement: Implement foreign key constraints in the bad_comments table to ensure referential integrity. This would prevent comments from being associated with non-existent posts and automatically delete comments when their corresponding posts are removed.
Additional Suggestions
Timestamp Fields: Adding created_at and updated_at timestamp fields to both tables would help track when posts and comments are created or modified.
Text Content Length Limits: Consider adding constraints on the length of text_content to prevent excessively long entries that could affect performance.

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:

```sql
-- Users table
CREATE TABLE users (
user_id SERIAL PRIMARY KEY,
username VARCHAR(25) UNIQUE NOT NULL CHECK (LENGTH(TRIM(username)) > 0),
last_login TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Topics Table
CREATE TABLE topics (
topic_id SERIAL PRIMARY KEY,topic_name VARCHAR(30) UNIQUE NOT NULL CHECK
(LENGTH(TRIM(topic_name)) > 0),
description VARCHAR(500),
user_id INT,
FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE SET NULL
);

-- Posts Table
CREATE TABLE posts (
post_id SERIAL PRIMARY KEY,
title VARCHAR(100) NOT NULL CHECK (LENGTH(TRIM(title)) > 0),
url VARCHAR(4000),
text_content TEXT,
user_id INT,
topic_id INT NOT NULL,
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE SET NULL,
FOREIGN KEY (topic_id) REFERENCES topics(topic_id) ON DELETE CASCADE,
CHECK ((url IS NOT NULL AND text_content IS NULL)
OR (url IS NULL AND text_content IS NOT NULL))
);

-- Index for the Posts table
CREATE INDEX idx_post_user ON posts (user_id);
CREATE INDEX idx_post_topic ON posts (topic_id);
CREATE INDEX idx_post_url ON posts (url);

-- Comments Table
CREATE TABLE comments (
comment_id SERIAL PRIMARY KEY,
text_content TEXT NOT NULL CHECK (LENGTH(TRIM(text_content)) > 0),
user_id INT,
post_id INT NOT NULL,
parent_comment_id INT NULL,
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE SET NULL,
FOREIGN KEY (post_id) REFERENCES posts(post_id) ON DELETE CASCADE,
FOREIGN KEY (parent_comment_id) REFERENCES comments(comment_id)
ON DELETE CASCADE
);
```

```sql
-- Index for the Comments table
CREATE INDEX idx_comment_post ON comments (post_id);
CREATE INDEX idx_comment_user ON comments (user_id);

-- Votes Table
CREATE TABLE votes (
vote_id SERIAL PRIMARY KEY,
vote_value INT CHECK (vote_value IN (1,
-1)),
user_id INT,
post_id INT NOT NULL,
UNIQUE (user_id, post_id),
FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE SET NULL,
FOREIGN KEY (post_id) REFERENCES posts(post_id) ON DELETE CASCADE
);

-- Index for the Votes table
CREATE INDEX idx_vote_user ON votes (user_id);
CREATE INDEX idx_vote_post ON votes (post_id);
CREATE OR REPLACE FUNCTION update_timestamp()
RETURNS TRIGGER AS $$
BEGIN
NEW.updated_at = CURRENT_TIMESTAMP;
RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Trigger for the posts table
CREATE TRIGGER set_timestamp
BEFORE UPDATE ON posts
FOR EACH ROW
EXECUTE FUNCTION update_timestamp();

-- Trigger for the comments table
CREATE TRIGGER set_timestamp
BEFORE UPDATE ON comments
FOR EACH ROW
EXECUTE FUNCTION update_timestamp();
```

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:

```
- Insert users who created posts
INSERT INTO users (username)
SELECT DISTINCT username FROM bad_posts
ON CONFLICT (username) DO NOTHING;

-- Insert users who commented
INSERT INTO users (username)
SELECT DISTINCT username FROM bad_comments
ON CONFLICT (username) DO NOTHING;
INSERT INTO users (username)
SELECT DISTINCT REGEXP_SPLIT_TO_TABLE(upvotes, ',') FROM bad_posts
ON CONFLICT (username) DO NOTHING;
INSERT INTO users (username)
SELECT DISTINCT REGEXP_SPLIT_TO_TABLE(downvotes, ',') FROM bad_posts
ON CONFLICT (username) DO NOTHING;
```

```sql
-- Insert topics with NULL descriptions
INSERT INTO topics (topic_name, description)
SELECT DISTINCT topic, NULL FROM bad_posts;-- Insert posts
INSERT INTO posts (
title, url, text_content,
user_id, topic_id
)
SELECT
SUBSTR(b.title, 1, 100),
b.url,
b.text_content,
u.user_id,
t.topic_id
FROM
bad_posts b
JOIN users u ON b.username = u.username
JOIN topics t ON b.topic = t.topic_name;

-- Insert comments as top-level comments
INSERT INTO comments (
text_content, user_id, post_id
)
SELECT
bc.text_content,
u.user_id,
bc.post_id
FROM
bad_comments bc
JOIN users u ON bc.username = u.username;

-- Insert votes using regexp_split_to_table(comma-separated)
INSERT INTO votes (vote_value, user_id, post_id)
SELECT 1, u.user_id, b.id
FROM
(
SELECT id, REGEXP_SPLIT_TO_TABLE(upvotes, ',') AS username FROM bad_posts
) b
JOIN users u ON u.username = b.username;
INSERT INTO votes (vote_value, user_id, post_id)
SELECT -1, u.user_id, b.id
FROM
(
SELECT id, REGEXP_SPLIT_TO_TABLE(downvotes, ',') AS username FROM bad_posts
) b
JOIN users u ON u.username = b.username;
```