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 FOREIGN KEY constraint might be added to the "bad_comments" table's "post_id" column to verify that each "post_id" in the "bad_comments" table matches an existing id in the "bad_posts" table, which would improve the bad posts table. The referential integrity between the two tables will be preserved as a result.
2. The "bad_posts" table presently has "TEXT" as the data type for the upvotes and downvotes columns. Given that these values are likely to be numerical, it could be preferable to store them using an integer data type (i.e INT). The use of a text data type could cause performance problems and make it more challenging to conduct mathematical operations on these values.
3. To enhance the efficiency of queries that filter or join on the "post_id" column in the "bad_comments" table, it would be useful to build an index on this column. When running these kinds of queries, the database will be able to locate rows in the table more quickly as a result.

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
-----------------------------------------------------------------------------------------
-- Creating users database and adding index
-----------------------------------------------------------------------------------------
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(25) NOT NULL UNIQUE,
    date_created TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    recent_login TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    CONSTRAINT username_not_empty CHECK (username <> '')
);
CREATE INDEX idx_recent_login ON users (recent_login);
CREATE UNIQUE INDEX idx_username ON users (username);
-----------------------------------------------------------------------------------------
-- Creating topics database and adding index
-----------------------------------------------------------------------------------------
CREATE TABLE topics (
    id SERIAL PRIMARY KEY,
    name VARCHAR(30) UNIQUE NOT NULL,
    description VARCHAR(500),
    date_created TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    made_by INTEGER,
    FOREIGN KEY (made_by) REFERENCES users(id) ON DELETE SET NULL,
    CONSTRAINT name_not_empty CHECK (name <> '')
);
CREATE INDEX idx_made_by ON topics (made_by);
CREATE UNIQUE INDEX idx_name ON topics (name);
-----------------------------------------------------------------------------------------
-- Creating post database and adding index
-----------------------------------------------------------------------------------------
CREATE TABLE posts (
    id SERIAL PRIMARY KEY,
    title VARCHAR(100) UNIQUE NOT NULL,
    url VARCHAR(255),
    text_content TEXT,
    date_created TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    made_by INTEGER,
    topic_id INTEGER NOT NULL,
    FOREIGN KEY (made_by) REFERENCES users(id) ON DELETE SET NULL,
    FOREIGN KEY (topic_id) REFERENCES topics(id) ON DELETE CASCADE,
    CONSTRAINT title_not_empty CHECK (title <> ''),
    CONSTRAINT only_one_content CHECK (
        (url IS NOT NULL AND text_content IS NULL)
        OR (url IS NULL AND text_content IS NOT NULL))
);
CREATE INDEX idx_topic_id ON posts (topic_id);
CREATE UNIQUE INDEX idx_title ON posts (title);
```

```sql
-----------------------------------------------------------------------------------------
-- Creating comments database and adding index
-----------------------------------------------------------------------------------------
CREATE TABLE comments (
    id SERIAL PRIMARY KEY,
    text_content TEXT UNIQUE NOT NULL,
    date_created TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    made_by INTEGER,
    post_id INTEGER NOT NULL,
    parent_id INTEGER NOT NULL,
    FOREIGN KEY (made_by) REFERENCES users(id) ON DELETE SET NULL,
    FOREIGN KEY (post_id) REFERENCES posts(id) ON DELETE CASCADE,
    FOREIGN KEY (parent_id) REFERENCES comments(id) ON DELETE CASCADE,
    CONSTRAINT parent_exists CHECK (parent_id IS NULL OR comments.id = parent_id)
);
CREATE INDEX idx_post_id ON comments (post_id);
CREATE INDEX idx_parent_id ON comments (parent_id);
-----------------------------------------------------------------------------------------
-- Creating votes database
-----------------------------------------------------------------------------------------
CREATE TABLE votes (
    id SERIAL PRIMARY KEY,
    value INTEGER NOT NULL,
    date_created TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    made_by INTEGER,
    post_id INTEGER NOT NULL,
    FOREIGN KEY (made_by) REFERENCES users(id) ON DELETE SET NULL,
    FOREIGN KEY (post_id) REFERENCES users(id) ON DELETE CASCADE,
    CONSTRAINT value_not_null CHECK (value IS NOT NULL)
);
```

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:

```
--------------------------------------------------------------------------------
-- Migrate data from bad_post and bad comments to users table
--------------------------------------------------------------------------------

-- Users who've made posts
INSERT INTO users (username)
SELECT DISTINCT username
FROM bad_posts;

-- Users who've made only commented
INSERT INTO users (username)
SELECT DISTINCT bc.username
FROM bad_comments bc
LEFT JOIN users u
ON bc.username = u.username
WHERE u.username IS NULL;
```

```sql
-- Users who've only downvoted
INSERT INTO users (username)
  WITH table1 AS (
    SELECT regexp_split_to_table(downvotes, ',') AS downvote
    FROM bad_posts)
  SELECT downvote
  FROM table1
  LEFT JOIN users u
  ON table1.downvote = u.username
  WHERE u.username IS NULL;

-- Users who've only upvoted
INSERT INTO users (username)
  WITH table1 AS (
    SELECT regexp_split_to_table(upvotes, ',') AS upvote
    FROM bad_posts)
  SELECT DISTINCT upvote
  FROM table1
  LEFT JOIN users u
  ON table1.upvote = u.username
  WHERE u.username IS NULL;

-------------------------------------------------------------------------------
-- Migrate data from bad_post to topics table
-------------------------------------------------------------------------------

INSERT INTO topics (name)
  SELECT DISTINCT topic
  FROM bad_posts;

-------------------------------------------------------------------------------
-- Migrate data from bad_post to post table
-------------------------------------------------------------------------------

INSERT INTO posts (id, title, url, text_content, date_created, made_by, topic_id)
  SELECT bp.id, SUBSTRING(bp.title, 1, 100), bp.url, bp.text_content,
         u.date_created, u.id, t.id
  FROM bad_posts bp
  JOIN users u ON bp.username = u.username
  JOIN topics t ON bp.topic = t.name;

-------------------------------------------------------------------------------
-- Migrate data from bad comments to comments table
-------------------------------------------------------------------------------

INSERT INTO comments (id, text_content, date_created, made_by, post_id, parent_id)
  SELECT bc.id, bc.text_content, u.date_created, u.id, p.id, NULL
  FROM bad_comments bc
  JOIN users u ON bc.username = u.username
  JOIN posts p ON bc.post_id = p.id;
```

```sql
-----------------------------------------------------------------------------------------
-- Migrate data from bad post to votes table
-----------------------------------------------------------------------------------------

-- For the downvotes
INSERT INTO votes (made_by, post_id, value)
  WITH table1 AS (
    SELECT id, REGEXP_SPLIT_TO_TABLE(downvotes, ',') AS downvote
    FROM bad_posts)
  SELECT u.id, table1.id, -1 AS value
  FROM table1
  JOIN users u
  ON u.username = table1.downvote;

-- For the upvotes
INSERT INTO votes (made_by, post_id, value)
  WITH table1 AS (
    SELECT id, REGEXP_SPLIT_TO_TABLE(upvotes, ',') AS upvote
    FROM bad_posts)
  SELECT u.id, table1.id, 1 AS value
  FROM table1
  JOIN users u
  ON u.username = table1.downvote;
```