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:

```sql
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 we need to do is normalize the data of the two tables bad_comments and bad_post. The data of these two tables has not been normalized.
2. There are no relationships between 2 tables. So, in bad_comments table, post_id column should be set as foreign key. 1 post may have many comments .
3. In bad_posts table, 2 column upvotes and downvotes should be integer datatype. Then we can use integers to replace the text data type, 1 for upvote and 0 for downvote.
4. Should create index to improve queries 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
BEGIN;
-- Part II: Create the DDL for your new schema
-- a. Allow new users to register:
CREATE TABLE users (
    user_id SERIAL PRIMARY KEY,
    username VARCHAR(25) UNIQUE NOT NULL,
    CONSTRAINT usernames_cant_be_empty CHECK(LENGTH(TRIM(username)) > 0),
    last_login TIMESTAMP
);
-- List all users who haven't logged in in the last year.
CREATE INDEX index_login ON users(last_login);
-- Find a user by their username.
CREATE INDEX find_by_user_name ON users(username);
-- b. Allow registered users to create new topics:
CREATE TABLE topics(
    topic_id SERIAL PRIMARY KEY,
    topic_name VARCHAR(30) UNIQUE NOT NULL,
    CONSTRAINT  topic_name_cant_be_empty CHECK(LENGTH(TRIM(topic_name)) > 0),
    desciption VARCHAR(500)
);
-- Find a topic by its name
CREATE INDEX find_topic_name ON topics(topic_name);
-- c. Allow registered users to create new posts on existing topics:
CREATE TABLE posts(
    post_id SERIAL PRIMARY KEY,
-- If a topic gets deleted, all the posts associated with it should be
automatically deleted too.
    topic_id INTEGER NOT NULL REFERENCES topics(topic_id) ON DELETE CASCADE,
-- If the user who created the post gets deleted, then the post will remain,
but it will become dissociated from that user
    user_id INTEGER REFERENCES users(user_id) ON DELETE SET NULL,
    title VARCHAR(100) NOT NULL,
-- The topic's name can't be empty
    CONSTRAINT title_of_post_cant_be_empty CHECK(LENGTH(TRIM(title)) > 0),
    url VARCHAR(5000),
    text_content VARCHAR,
-- Posts should contain either a URL or a text content, but not both.
    CONSTRAINT content_or_url CHECK((url IS NULL AND text_content IS NOT NULL)
OR (url IS NOT NULL AND text_content IS NULL)),
    date_create_post TIMESTAMP
);
-- Find all posts that link to a specific URL, for moderation purposes.
CREATE INDEX find_by_url ON posts(url);
```

```sql
CREATE INDEX find_post_by_user ON posts(user_id);
-- d. Allow registered users to comment on existing posts:
CREATE TABLE comments(
    cmt_id SERIAL PRIMARY KEY,
    parent_cmt_id INTEGER,
    post_id INTEGER REFERENCES posts ON DELETE CASCADE,
    user_id INTEGER REFERENCES users ON DELETE SET NULL,
    comment_time TIMESTAMP,
    content VARCHAR NOT NULL,
    CONSTRAINT comment_not_empty CHECK(LENGTH(TRIM(content))>0)
);
ALTER TABLE comments
    ADD CONSTRAINT parent_comment FOREIGN KEY(parent_cmt_id)
    REFERENCES comments(cmt_id) ON DELETE CASCADE;
-- List the latest comments made by a given user.
CREATE INDEX comments_by_user ON comments(user_id,comment_time);
-- e. Make sure that a given user can only vote once on a given post:
CREATE TABLE votes(
    vote_id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users ON DELETE SET NULL,
    post_id INTEGER REFERENCES posts ON DELETE CASCADE,
    post_vote INTEGER,
    CONSTRAINT set_value_for_vote CHECK(post_vote IN(-1,1)),
    CONSTRAINT user_only_one_vote UNIQUE (user_id, post_id)
);
COMMIT;
```

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:

```sql
BEGIN;
-- table users --
WITH list_user_from_old AS(
    SELECT username FROM bad_posts UNION
    SELECT username FROM bad_comments UNION
    SELECT REGEXP_SPLIT_TO_TABLE(upvotes,',') FROM bad_posts UNION
    SELECT REGEXP_SPLIT_TO_TABLE(downvotes,',') FROM bad_posts
)
INSERT INTO users(username)
    (SELECT DISTINCT username FROM list_user_from_old);
-- table topics --
INSERT INTO topics(topic_name)
    (SELECT DISTINCT topic FROM bad_posts);
-- table posts --
```

```sql
INSERT INTO posts(topic_id,user_id,title,url,text_content)
    (SELECT t.topic_id, u.user_id, LEFT(bad_posts.title, 100), bad_posts.url,
bad_posts.text_content
    FROM bad_posts
    INNER JOIN topics t ON bad_posts.topic = t.topic_name
    INNER JOIN users u ON bad_posts.username = u.username);
-- table comments --
WITH comment_from_badcomments AS(
    SELECT bc.text_content,posts.user_id,posts.post_id
    FROM bad_posts bp
    INNER JOIN posts ON bp.title = posts.title
    INNER JOIN users ON posts.user_id = users.user_id
    INNER JOIN bad_comments bc ON bc.username = users.username
)
INSERT INTO comments(content,user_id,post_id)
    (SELECT * FROM comment_from_badcomments);
-- table votes --
-- upvote --
WITH upvote_list AS(
    SELECT users.user_id, posts.post_id , 1 AS upvote
    FROM(
        SELECT title, REGEXP_SPLIT_TO_TABLE(upvotes,',') AS username FROM
bad_posts) bp
    INNER JOIN users
    ON users.username = bp.username
    INNER JOIN posts
    ON posts.title = bp.title
)
INSERT INTO votes(user_id,post_id,post_vote)
    (SELECT * FROM upvote_list);
-- downvote --
WITH downvote_list AS(
    SELECT users.user_id, posts.post_id , -1 AS downvote
    FROM(
        SELECT title, REGEXP_SPLIT_TO_TABLE(downvotes,',') AS username FROM
bad_posts) bp
    INNER JOIN users
    ON users.username = bp.username
    INNER JOIN posts
    ON posts.title = bp.title
)
INSERT INTO votes(user_id,post_id,post_vote)
    (SELECT * FROM downvote_list);
COMMIT;
```