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. Data is denormalized. We should apply the 3FN and split the data into different tables.
2. Upvotes and Downvotes columns should use a numeric approach. INT data type and make it so we only use one column.
3. We need to add some CONSTRAINTS to keep the data consistent.
4. Indexing the data will help our data run the queries faster.

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
-- Allow new users to register
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  username VARCHAR(25) NOT NULL, -- Usernames can be composed of at most 25
characters
  login TIMESTAMP,
  CONSTRAINT username_not_empty CHECK (LENGTH(TRIM(username )) > 0) --
Usernames cant be empty
);
CREATE UNIQUE INDEX unique_user ON users (LOWER(username)); -- Each username
has to be unique
CREATE INDEX last_login ON users(login); -- Login index

-- Allow registered users to create new topics
CREATE TABLE topics (
  id SERIAL PRIMARY KEY,
  topic_name VARCHAR(30) UNIQUE NOT NULL, -- The topics name is at most 30
characters, have to be unique
  description VARCHAR(500), -- Topics can have an optional description of at
most 500 characters
  CONSTRAINT topic_not_empty CHECK (LENGTH(TRIM(topic_name)) > 0) -- The
topics name cant be empty
);
CREATE INDEX topic_index ON topics (topic_name VARCHAR_PATTERN_OPS); -- Find
a topic by its name

-- Allow registered users to create new posts on existing topics
CREATE TABLE posts (
  id SERIAL PRIMARY KEY,
  title VARCHAR(100) NOT NULL, -- Posts have a required title of at most 100
characters
  url VARCHAR(2000) DEFAULT NULL,
  text_content text DEFAULT NULL,
  topic_id INT REFERENCES topics(id) ON DELETE CASCADE NOT NULL, -- If a
topic gets deleted, cascade
  user_id INT REFERENCES users(id) ON DELETE SET NULL, -- If a user gets
deleted, null
  post_date TIMESTAMP,
  CONSTRAINT url_or_text CHECK ((url IS NULL AND text_content IS NOT NULL)
OR (url IS NOT NULL AND text_content IS NULL)),  -- Posts should contain
either a URL or a text content, **but not both**
  CONSTRAINT title_not_empty CHECK (LENGTH(TRIM(title)) > 0) -- The title of
a post cant be empty.
);
CREATE INDEX post_creator ON posts(user_id); -- Posts by a given user
CREATE INDEX latest_post ON posts(post_date); -- Posts by date
CREATE INDEX post_by_topic ON posts(topic_id); -- Posts by topic
```

```sql
-- Allow registered users to comment on existing posts:
-- Contrary to the current linear comments, the new structure should allow
comment threads at arbitrary levels.
CREATE TABLE comments (
  id SERIAL PRIMARY KEY,
  content TEXT NOT NULL,
  user_id INT REFERENCES users(id) ON DELETE SET NULL, -- If the user who
created the comment gets deleted, null.
  post_id INT REFERENCES posts(id) ON DELETE CASCADE NOT NULL, -- If a post
gets deleted, cascade.
  parent_comment_id INT REFERENCES comments(id) ON DELETE CASCADE, -- If a
comment gets deleted, cascade thread, this field can contain NULL values.
  CONSTRAINT comment_not_empty CHECK (LENGTH(TRIM(content)) > 0) -- A
comments text content cant be empty.
);
CREATE INDEX parent_comments ON comments(id) WHERE parent_comment_id = NULL;
-- top-level comments
CREATE INDEX children_comments ON comments(parent_comment_id); -- direct
children of a parent comment
CREATE INDEX comments_by_user ON comments(user_id); -- comments made by a
given user

-- Votes
CREATE TABLE votes (
  id SERIAL PRIMARY KEY,
  post_id INT REFERENCES posts(id) ON DELETE CASCADE NOT NULL, -- If a post
gets deleted, cascade.
  user_id INT REFERENCES posts(id) ON DELETE SET NULL, -- If the user who
cast a vote gets deleted, null.
  vote INT CHECK (vote = 1 OR vote = -1),
  UNIQUE (post_id, user_id) -- Make sure that a given user can only vote
once on a given post
);
CREATE INDEX score ON votes(vote); -- Score of a post
```

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:

```
-- Inserting data into the users table
INSERT INTO users (username)
  WITH all_usernames AS (
    SELECT username FROM bad_posts
  UNION
  SELECT username FROM bad_comments
  UNION
  SELECT DISTINCT regexp_split_to_table(upvotes, ',') AS username FROM
bad_posts
  UNION
  SELECT DISTINCT regexp_split_to_table(downvotes, ',') AS username FROM
bad_posts
  )
  SELECT DISTINCT username
  FROM all_usernames;
```

```sql
-- Inserting data into the topics table
INSERT INTO topics (topic_name)
  SELECT DISTINCT topic FROM bad_posts;

-- Inserting data into the posts table
INSERT INTO posts (title, topic_id, user_id, url, text_content)
  SELECT LEFT(bp.title, 100), t.id, u.id, bp.url, bp.text_content
  FROM bad_posts bp
  JOIN users u
  ON u.username = bp.username
  JOIN topics t
  ON t.topic_name = bp.topic;

-- Inserting data into the comments table
INSERT INTO comments (content, user_id, post_id)
  SELECT bc.text_content, u.id, p.id
  FROM bad_comments bc
  JOIN users u
  ON u.username = bc.username
  JOIN posts p
  ON p.id = bc.post_id;

-- Inserting data into the votes table
-- Upvote
INSERT INTO votes (post_id, user_id, vote)
  WITH upvote AS (
    SELECT id, regexp_split_to_table(upvotes, ',') AS username FROM
bad_posts
  )
  SELECT p.id, u.id, 1
  FROM upvote uv
  JOIN users u
  ON u.username = uv.username
  JOIN posts p
  ON p.id = uv.id;

-- Downvote
INSERT INTO votes (post_id, user_id, vote)
  WITH downvote AS (
    SELECT id, regexp_split_to_table(downvotes, ',') AS username FROM
bad_posts
  )
  SELECT p.id, u.id, -1
  FROM downvote dv
  JOIN users u
  ON u.username = dv.username
  JOIN posts p
  ON p.id = dv.id;
```