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. The `topic` column of the `bad_posts` table is inconsistently capitalized. It would be better to create a separate `topics` table with columns `id` and `topic_name`, with the `topic` column of the `bad_posts` table replaced by a `topic_id` column as a foreign key that references the `id` column of the `topics` table.
2. Similarly, rather than storing the `username` directly in the `bad_posts` table, it would be better to create a separate `users` table with columns `id` and `username`, with the `username` column of the `bad_posts` table replaced by a `user_id` column as a foreign key that references the `id` column of the `users` table.
3. The same concept also applies to storing the `username` directly in the `bad_comments` table. The `username` column of the `bad_comments` table should be replaced by a `user_id` column as a foreign key that references the `id` column of the same `users` table described above.
4. The `upvotes` and `downvotes` should not be represented directly in the `bad_posts` table. Instead, it would be better to create a separate `votes` table with three columns: `user_id`, `post_id`, and a column `is_upvote` specified as a `BOOLEAN` that is `TRUE` for an upvote and `FALSE` for a downvote (since a user presumably is not allowed to have the same post both upvoted and downvoted simultaneously). This table would have its primary key specified as `(post_id, user_id)`, since that order would probably make more sense for indexing purposes. (One might frequently wish to quickly find, or at least count, all votes for a particular post, whereas finding or counting all votes by a particular user seems like a less commonly required task.)
5. The `post_id` column of `bad_comments` is a `BIGINT`, but the `id` column of `bad_posts` that it is intended to reference was specified as `SERIAL`, which corresponds to `INTEGER` rather than `BIGINT`. These should be brought into concordance one way or the other. It would probably be wise to specify all id columns as `BIGINT` (or `BIGSERIAL`) to eliminate any chance of running out of ids if this social media site were to become wildly successful.
6. The reference of the `post_id` column of `bad_comments` to the `id` column of `bad_posts` should also be made explicit by setting it up as a foreign key.

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:

Note: For the reviewer's convenience, I have also included the DDL below with my project submission as the file `schema.sql` in the `sql` directory.

```sql
CREATE TABLE users (
  id BIGSERIAL PRIMARY KEY,
  -- uniqueness of username enforced by unique index
  username VARCHAR(25) NOT NULL,
  -- allow last_login to be null since a user could create an account
  -- (or have one created for them) without ever logging in
  last_login TIMESTAMP WITH TIME ZONE,
  -- the number_of_posts column is redundant, as it can be computed from
  -- the posts table whenever it is needed, but it is stored here so that
  -- it can be indexed as per item 2b of the project guidelines
  number_of_posts INTEGER DEFAULT 0,
  CONSTRAINT username_nonempty CHECK (LENGTH(TRIM(username)) > 0)
);
CREATE UNIQUE INDEX user_by_username ON users (username);
CREATE INDEX user_by_last_login ON users (last_login);
CREATE INDEX user_by_number_of_posts ON users (number_of_posts);

CREATE TABLE topics (
  -- an ordinary 4-byte integer should suffice for id here
  id SERIAL PRIMARY KEY,
  -- uniqueness of name enforced by unique index
  name VARCHAR(30) NOT NULL,
  description VARCHAR(500),
  -- the number_of_posts column is redundant, as it can be computed from
  -- the posts table whenever it is needed, but it is stored here so that
  -- it can be indexed as per item 2d of the project guidelines
  number_of_posts INTEGER DEFAULT 0,
  CONSTRAINT topic_name_nonempty CHECK (LENGTH(TRIM(name)) > 0)
);
CREATE UNIQUE INDEX topic_by_name ON topics (name);

CREATE TABLE posts (
  id BIGSERIAL PRIMARY KEY,
  user_id BIGINT REFERENCES users (id) ON DELETE SET NULL,
  topic_id INTEGER REFERENCES topics (id) ON DELETE CASCADE,
  title VARCHAR(100) NOT NULL,
  created_at TIMESTAMP WITH TIME ZONE,
  -- choose a reasonable limit for url length
  url VARCHAR(2000),
  content TEXT,
  CONSTRAINT title_nonempty CHECK (LENGTH(TRIM(title)) > 0),
  CONSTRAINT post_url_absent_or_nonempty CHECK (
       url IS NULL OR LENGTH(TRIM(url)) > 0
  ),
  CONSTRAINT post_content_absent_or_nonempty CHECK (
       content IS NULL OR LENGTH(TRIM(content)) > 0
  ),
  CONSTRAINT post_url_xor_content CHECK (
       -- exactly one of url or content must be present
       (url IS NULL AND content IS NOT NULL) OR
       (url IS NOT NULL AND content IS NULL)
  )
);
-- including created_at in these indexes makes it easier to retrieve
-- the last 20 posts by the specified user or for the specified topic
```

```sql
CREATE INDEX posts_by_user ON posts (user_id, created_at);
CREATE INDEX posts_by_topic ON posts (topic_id, created_at);
CREATE INDEX posts_by_url ON posts (url);

CREATE TABLE comments (
  id BIGSERIAL PRIMARY KEY,
  user_id BIGINT REFERENCES users (id) ON DELETE SET NULL,
  created_at TIMESTAMP WITH TIME ZONE,
  content TEXT NOT NULL,
  -- top-level comments have a parent_post_id; others have a parent_comment_id
  parent_post_id BIGINT REFERENCES posts (id) ON DELETE CASCADE,
  parent_comment_id BIGINT REFERENCES comments (id) ON DELETE CASCADE,
  CONSTRAINT comment_exactly_one_parent CHECK (
      -- exactly one of parent_post_id or parent_comment_id must be present
      (parent_post_id IS NULL AND parent_comment_id IS NOT NULL) OR
      (parent_post_id IS NOT NULL AND parent_comment_id IS NULL)
  ),
  CONSTRAINT comment_content_nonempty CHECK (LENGTH(TRIM(content)) > 0)
);
-- including created_at in this index makes it easier to retrieve
-- the last 20 comments by the specified user
CREATE INDEX comments_by_user ON comments (user_id, created_at);
CREATE INDEX comments_by_parent_post ON comments (parent_post_id);
CREATE INDEX comments_by_parent_comment ON comments (parent_comment_id);

CREATE TABLE votes (
  user_id BIGINT REFERENCES users (id) ON DELETE SET NULL,
  post_id BIGINT REFERENCES posts (id) ON DELETE CASCADE,
  -- a boolean would use less space for storing the value (one byte vs. two),
  -- but using a smallint with 1 and -1 allows finding a post's score by
  -- simply computing a sum
  value SMALLINT NOT NULL,
  CONSTRAINT vote_up_or_down CHECK (value IN (1, -1)),
  -- specifying the primary key in this order (with post_id first) allows for
  -- quick lookup of all votes for a particular post, which is required when
  -- computing the post's score (upvotes minus downvotes)
  PRIMARY KEY (post_id, 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:

Note: For the reviewer's convenience, I have also included the DML/DQL below with my project submission as the file `migration.sql` in the `sql` directory.

```
INSERT INTO users (username)
-- there is no need for SELECT DISTINCT due to the use of UNION
SELECT username FROM bad_posts
UNION SELECT REGEXP_SPLIT_TO_TABLE(upvotes, ',') FROM bad_posts
UNION SELECT REGEXP_SPLIT_TO_TABLE(downvotes, ',') FROM bad_posts
UNION SELECT username FROM bad_comments;

INSERT INTO topics (name)
-- standardize capitalization of topic names to title case
SELECT DISTINCT INITCAP(topic) FROM bad_posts;

-- Note 1: Leaving the timestamps as NULL for existing posts and comments
-- causes them to be sorted as earlier than all non-null timestamps
-- created later (after the migration), which is the correct behavior.
```

```sql
-- Note 2: The old INTEGER id values for posts and comments can be used as
-- the new BIGINT id values; there is no need to explicitly cast them.

INSERT INTO posts (id, user_id, topic_id, title, url, content)
-- keep only first 100 characters of title if it is longer than that
SELECT bp.id, u.id, t.id, LEFT(title, 100), url, text_content
FROM bad_posts bp
JOIN users u ON bp.username = u.username
JOIN topics t ON INITCAP(bp.topic) = t.name;

INSERT INTO comments (id, user_id, content, parent_post_id)
SELECT bc.id, u.id, bc.text_content, bc.post_id
FROM bad_comments bc JOIN users u ON bc.username = u.username;

WITH temp_votes AS (
  SELECT
       REGEXP_SPLIT_TO_TABLE(upvotes, ',') AS username,
       id AS post_id,
       1 AS value
  FROM bad_posts
  UNION ALL SELECT
       REGEXP_SPLIT_TO_TABLE(downvotes, ',') AS username,
       id AS post_id,
       -1 AS value
  FROM bad_posts
)
INSERT INTO votes (user_id, post_id, value)
SELECT u.id, tv.post_id, tv.value
FROM temp_votes tv JOIN users u ON tv.username = u.username;

UPDATE users u SET number_of_posts = (
  SELECT COUNT(*) FROM posts p WHERE p.user_id = u.id
);
UPDATE topics t SET number_of_posts = (
  SELECT COUNT(*) FROM posts p WHERE p.topic_id = t.id
);
```