

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. Storing votes as text: In the current schema, the upvotes and downvotes are stored as TEXT data types. This approach makes it difficult to perform operations and calculations on the votes. It would be better to store the votes as separate integer columns representing the count of upvotes and downvotes.
2. Lack of foreign key constraints: The existing schema does not define any foreign key constraints to establish relationships between tables. This can lead to inconsistencies and data integrity issues. For example, when a user or topic is deleted, there is no mechanism to automatically handle the associated posts or comments. Implementing foreign key constraints would help maintain data integrity and enforce referential integrity rules.
3. Linear comment structure: The schema currently supports linear comments where each comment is independent and doesn't have a hierarchical structure. This makes it difficult to represent threaded discussions and navigate through comment threads. To improve this, the schema should be modified to support a hierarchical comment structure, allowing comments to have parent-child relationships.

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 Uddidit. 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 Uddidit 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 Uddidit 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:

```

-----
-- DATABASE TABLES
-----

-- Create users table for new data
CREATE TABLE users (
    "id" SERIAL PRIMARY KEY,
    "username" VARCHAR(25) NOT NULL UNIQUE,
    "loggin_time" TIMESTAMP,
    CONSTRAINT "username_length" CHECK(
        Length(
            Trim("username")
        ) > 0
    )
);

-- Create topics table
CREATE TABLE topics (
    "id" SERIAL PRIMARY KEY,
    "name" VARCHAR(30) NOT NULL UNIQUE,
    "description" VARCHAR(500),
    "user_id" INT REFERENCES users(id) ON DELETE
    SET
        NULL,
    CONSTRAINT "topic_name_length" CHECK(
        Length(
            Trim("name")
        ) > 0
    )
);

-- Create posts table
CREATE TABLE posts (
    "id" SERIAL PRIMARY KEY,
    "title" VARCHAR(100) NOT NULL,
    "url" VARCHAR(4000),
    "text_content" TEXT,
    "created_at" TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    "user_id" INT REFERENCES users(id) ON DELETE
    SET
        NULL,
    "topic_id" INT REFERENCES topics(id) ON DELETE CASCADE NOT NULL,
    CONSTRAINT "text_content_or_url" CHECK(
        (
            ("url") IS NULL
            AND ("text_content") IS NOT NULL
        )
        OR (
            ("url") IS NOT NULL
            AND ("text_content") IS NULL
        )
    )
);

```

```

    )
),
CONSTRAINT "post_title_length" CHECK(
    Length(
        Trim("title")
    ) > 0
)
);
-- Create comments table
CREATE TABLE comments (
    "id" SERIAL PRIMARY KEY,
    "comment" TEXT NOT NULL,
    "created_at" TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    "user_id" INT REFERENCES users(id) ON DELETE
SET
    NULL,
    "post_id" INT REFERENCES posts(id) ON DELETE CASCADE,
    "parent_comment_id" INT REFERENCES comments(id) ON DELETE CASCADE,
    "topic_id" INT REFERENCES topics(id) ON DELETE CASCADE NOT NULL,
    CONSTRAINT "comment_not_empty" CHECK(
        Length(
            Trim("comment")
        ) > 0
    )
);
-- Create votes table
CREATE TABLE votes (
    "id" SERIAL PRIMARY KEY,
    "vote_value" INT CHECK (
        vote_value = 1
        OR vote_value = -1
    ),
    "created_at" TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    "user_id" INT REFERENCES users(id) ON DELETE
SET
    NULL,
    "post_id" INT REFERENCES posts(id) ON DELETE CASCADE,
    "topic_id" INT REFERENCES topics(id) ON DELETE CASCADE 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:

```
-----  
--  
--      DATA MIGRATION FROM BAD SCHEMAS TO NORMALIZED SCHEMAS  
-----  
--  
-- Insert all unique usernames from both bad_comments and bad_posts  
-- tables  
INSERT INTO users("username")  
SELECT  
    DISTINCT username  
FROM  
    bad_posts  
UNION
```

```

SELECT
    DISTINCT username
FROM
    bad_comments;
-- Insert distinct topic form bad_posts
INSERT INTO topics(name, description)
SELECT
    DISTINCT topic,
    NULL
FROM
    bad_posts;
-- Insert fields from the "bad_posts", "users", and "topics"
-- Using sunbstring for long title to take only 100 character to
store in table
INSERT INTO posts(
    "title", "url", "text_content", "user_id",
    "topic_id"
)
SELECT
    substring(bad_posts.title for 100),
    bad_posts.url,
    bad_posts.text_content,
    users.id AS user_id,
    topics.id AS topic_id
FROM
    "bad_posts"
    JOIN users ON users.username = bad_posts.username
    JOIN topics ON bad_posts.topic = topics.name;
-- Migrate comments (as top-level comments)
INSERT INTO comments (
    user_id, post_id, parent_comment_id,
    comment, topic_id
)
SELECT
    u.id,
    p.id,
    NULL,
    bc.text_content,
    p.topic_id
FROM
    bad_comments bc
    JOIN users u ON u.username = bc.username
    JOIN posts p ON p.id = bc.post_id;

```



```

-- Migrate votes
INSERT INTO votes (
    user_id, post_id, vote_value, topic_id
)
SELECT
    u.id,
    p.id,
    1,
    p.topic_id
FROM
    bad_posts bp
    JOIN users u ON u.username = bp.username
    JOIN posts p ON p.title = bp.title CROSS
    JOIN LATERAL regexp_split_to_table(bp.upvotes, ',') vote;
-- Migrate downvotes as negative votes
INSERT INTO votes (
    user_id, post_id, vote_value, topic_id
)
SELECT
    u.id,
    p.id,
    -1,
    p.topic_id
FROM
    bad_posts bp
    JOIN users u ON u.username = bp.username
    JOIN posts p ON p.title = bp.title CROSS
    JOIN LATERAL regexp_split_to_table(bp.downvotes, ',') vote;

```