# 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!

the table bad_posts should be about post mainly. so everything else should be an id or in a new table.
- 1/ a table user should exist
- 2/ a table vote should exist
- 3/ foreign_key are missing so we can link user table to post table and comment table. foreign key are missing so we can link vote to comments table.
- 4/ This is not mandatory but a table topic can be another new table
- 5/ vote should be specific number or boolean but not text , so we can count them.

## 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

```
-- create users table
CREATE TABLE "users" (
  "id" SERIAL PRIMARY KEY,
  "username" VARCHAR(25) NOT NULL,
  "last_login_at" TIMESTAMP DEFAULT CURRENT_TIMESTAMP
  );

-- Ensure usernames are unique
ALTER TABLE "users" ADD CONSTRAINT "username_unique"
    UNIQUE ("username");

-- ensure username can't be empty\blank.
ALTER TABLE "users"
ADD CONSTRAINT "chk_username_text" CHECK (trim("username") <> '');
```

    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.

```
CREATE TABLE topics (
    id SERIAL PRIMARY KEY,
```

```
    name VARCHAR(30) NOT NULL,
    description VARCHAR(500)

);

-- Ensure topic names are unique
ALTER TABLE "topics"
ADD CONSTRAINT "unique_topic_name" UNIQUE ("name");

-- Ensure topic names cannot be empty (NOT NULL)
ALTER TABLE "topics"
ADD CONSTRAINT "chk_topic_name" CHECK (trim("name") <> '');
```

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.

```
CREATE TABLE posts (
    id SERIAL PRIMARY KEY,
    topic_id INT ,
    user_id INT,
    title VARCHAR(100) NOT NULL,
    url VARCHAR(400) ,
    text_content TEXT ,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- A comment's text content can't be empty.
ALTER TABLE "posts"
ADD CONSTRAINT "chk_title" CHECK (trim("text_content") <> '');


-- Ensure posts contain either a URL or text content, but not both
```

```
ALTER TABLE "posts"
ADD CONSTRAINT "chk_url_or_text" CHECK (
    (url IS NOT NULL AND "text_content" IS NULL) OR
    (url IS NULL AND "text_content" IS NOT NULL)
);

-- Automatically delete posts if the topic is deleted
ALTER TABLE "posts"
ADD CONSTRAINT "fk_posts_topic" FOREIGN KEY ("topic_id")
REFERENCES "topics"("id") ON DELETE CASCADE;

-- If a user is deleted, keep the post but dissociate it from the user
ALTER TABLE posts
ADD CONSTRAINT "fk_posts_user" FOREIGN KEY ("user_id")
REFERENCES "users"("id") ON DELETE SET NULL;
```

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.

```
CREATE TABLE comments (
    id SERIAL PRIMARY KEY,
    user_id INT,         -- ID of the user who made the comment
    post_id INT,        -- ID of the post to which the comment belongs
    parent_comment_id INT ,  -- ID of the parent comment (NULL for
top-level comments)
    text_content, TEXT NOT NULL -- The comment text (cannot be empty)
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- A comment's text content can't be empty.
```

```
ALTER TABLE "comments"
ADD CONSTRAINT "chk_comment_text" CHECK (trim("text_content") <> '');

-- If a post gets deleted, all comments associated with it should be
automatically deleted too.
ALTER TABLE "comments" ADD CONSTRAINT "fk_comments_post"
FOREIGN KEY ("post_id") REFERENCES "posts"(id) ON DELETE CASCADE;

-- If the user who created the comment gets deleted, then the comment will
remain, but it will become dissociated from that user.
ALTER TABLE "comments" ADD CONSTRAINT "fk_comments_user"
FOREIGN KEY ("user_id") REFERENCES "users"("id") ON DELETE SET NULL;

-- If a comment gets deleted, then all its descendants in the thread
structure should be automatically deleted too.

ALTER TABLE "comments" ADD CONSTRAINT "fk_comments_parent"
FOREIGN KEY ("parent_comment_id") REFERENCES "comments"("id") ON DELETE
CASCADE;
```

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.

```
CREATE TABLE votes (
    id SERIAL PRIMARY KEY,
    user_id INT,        -- ID of the user who voted
    post_id INT ,  -- ID of the post being voted on
    vote_value SMALLINT NOT NULL  -- Vote value: 1 or -1
);

-- you can store the (up/down) value of the vote as the values 1 and -1
respectively
ALTER TABLE "votes"
```

```
ADD CONSTRAINT "chk_vote_value" CHECK ("vote_value" IN (-1, 1));

-- given user can only vote once
ALTER TABLE "votes"
ADD CONSTRAINT "unique_user_post_vote" UNIQUE ("user_id", "post_id");

-- If the user who cast a vote gets deleted, then all their votes will
remain, but will become dissociated from the user.
ALTER TABLE "votes"  ADD CONSTRAINT "fk_votes_user"
FOREIGN KEY ("user_id") REFERENCES "users"(id) ON DELETE SET NULL;

-- If a post gets deleted, then all the votes for that post should be
automatically deleted too.

ALTER TABLE "votes"  ADD CONSTRAINT "fk_votes_post"
FOREIGN KEY ("post_id") REFERENCES "posts"(id) ON DELETE CASCADE;
```

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.

```
-- Indexes to improve join performance on foreign keys:
-- Indexes to improve join performance on foreign keys:
CREATE INDEX idx_posts_topic_id ON posts(topic_id);
CREATE INDEX idx_posts_user_id  ON posts(user_id);


-- Indexes for quick lookups on foreign key columns:
CREATE INDEX idx_comments_post_id           ON comments(post_id);
CREATE INDEX idx_comments_parent_comment_id ON
comments(parent_comment_id);
CREATE INDEX idx_comments_user_id           ON comments(user_id);


-- Indexes for performance:
CREATE INDEX idx_votes_post_id ON votes(post_id);
CREATE INDEX idx_votes_user_id ON votes(user_id);
```

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:

```
```

## 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:

### 1. Migrate Users

```
INSERT INTO users(username)
SELECT DISTINCT trim(username)
FROM (
  SELECT username
    FROM bad_posts
  UNION
  SELECT username
    FROM bad_comments
  UNION
  -- Unwind comma-separated upvotes
  SELECT regexp_split_to_table(upvotes, ',') AS username
```

```
    FROM bad_posts
    WHERE upvotes IS NOT NULL
  UNION
  -- Unwind comma-separated downvotes
  SELECT regexp_split_to_table(downvotes, ',') AS username
    FROM bad_posts
    WHERE downvotes IS NOT NULL
) AS all_users
WHERE trim(username) <> '';
```

### 2. Migrate Topics

```
INSERT INTO topics(name, description)
SELECT DISTINCT trim(topic) AS name,
       '' AS description
FROM bad_posts
WHERE topic IS NOT NULL AND trim(topic) <> '';
```

### 3. Migrate Posts

```
INSERT INTO posts(id, topic_id, user_id, title, url, text_content)
SELECT b.id,
       t.id,
       u.id,
       left(b.title,100),
       b.url,
       b.text_content
FROM bad_posts b
JOIN topics t ON t.name = trim(b.topic)
JOIN users u ON u.username = trim(b.username);
```

### 4. Migrate Comments

```
INSERT INTO comments(user_id, post_id, parent_comment_id, text_content)
SELECT u.id,
       p.id,
       NULL,               -- No parent since legacy comments are not
threaded
```

```
       bc.text_content

FROM bad_comments bc
JOIN posts p ON p.id = bc.post_id
JOIN users u ON u.username = trim(bc.username);
```

### 5. Migrate Votes

```
INSERT INTO votes(user_id, post_id, vote_value)
SELECT u.id,
       b.id,
       1 AS vote_value

FROM bad_posts b
CROSS JOIN LATERAL regexp_split_to_table(b.upvotes, ',') AS vote_username
JOIN users u ON u.username = trim(vote_username)
WHERE b.upvotes IS NOT NULL AND trim(b.upvotes) <> '';


INSERT INTO votes(user_id, post_id, vote_value)
SELECT u.id,
       b.id,
       -1 AS vote_value
FROM bad_posts b
CROSS JOIN LATERAL regexp_split_to_table(b.downvotes, ',') AS
vote_username
JOIN users u ON u.username = trim(vote_username)
WHERE b.downvotes IS NOT NULL AND trim(b.downvotes) <> '';
```