# 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!

Improvement on bad_posts table

1. Bad_posts table is difficult to read. I would split it into multiple tables which need to follow the three normalization forms.
2. There is no constraint on the username which allows anyone to register under the same name. I would include the unique and not null constraint on username
3. There is also no constraint on who can post and/or create topics. I would add that only if a username exists then a topic or a port can be created.
4. The URL is limited to 4000 characters which probably should be enough but I would remove the limit just in case.
5. The upvote and downvote column is an accumulated number of users who liked or disliked a particular post. This violates the first normalization form. I would create a separate table just to keep track of how many registered users liked or disliked a post
6. There are 50 000 rows in the table I would also certainly include indexing for columns like upvote/downvote, username and topic.

Improvement on bad_comments table

1. I would again include indexing because there are 100 000 rows so which are currently missing from username for example
2. I would also put constraint on who can comment and make sure that there are no empty comments left on any posts.
3. I would also put a limitation on how long each comment can be. I think 5000 characters will be a limit.

## 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:

```
CREATE TABLE "users" (
  "id" SERIAL PRIMARY KEY,
  "username" VARCHAR(25) UNIQUE NOT NULL,
    "login_date" DATE
);

CREATE TABLE "topic" (
  "id" SERIAL PRIMARY KEY,
  "topic_name" VARCHAR(30) UNIQUE NOT NULL,
  "description" VARCHAR(500)
```

```
);

CREATE TABLE "post" (
  "id" SERIAL PRIMARY KEY,
   "user_id" INTEGER
   CONSTRAINT "valid_post_on_user" REFERENCES "users" ("id") ON DELETE SET NULL,
   "topic_id" INTEGER
   CONSTRAINT "valid_post_on_topic" REFERENCES "topic" ("id") ON DELETE CASCADE ,
   "title" VARCHAR(100),
   "content" VARCHAR
   CONSTRAINT "valid_post_content"
   CHECK ("content"  LIKE 'http%' OR "content" IS NOT NULL)
);

CREATE TABLE "comment" (
    "id" SERIAL PRIMARY KEY,
    "user_id" INTEGER,
    "post_id" INTEGER ,
    "thread_id" INTEGER,
    "comment_content" VARCHAR(5000) NOT NULL,
    "comment_date" DATE

);

ALTER TABLE "comment"
  ADD CONSTRAINT "valid_postID_on_post"
  FOREIGN KEY ("post_id") REFERENCES "post" ("id") ON DELETE CASCADE,

  ADD CONSTRAINT "valid_postID_on_user"
  FOREIGN KEY ("user_id") REFERENCES "users" ("id") ON DELETE SET NULL,

  ADD CONSTRAINT "valid_threadID_on_post"
  FOREIGN KEY ("thread_id") REFERENCES "comment" ("id") ON DELETE CASCADE;


CREATE TABLE "votes" (
    "post_id" INTEGER REFERENCES "post" ("id") ON DELETE CASCADE ,
    "user_id" INTEGER REFERENCES "users" ("id") ON DELETE SET NULL,
    "vote" SMALLINT NOT NULL CHECK ("vote" = -1 OR "vote" = 1) ,
    PRIMARY KEY("user_id", "post_id")
);


CREATE INDEX "find_no_logged" ON "users" ("login_date");
CREATE INDEX "no_topic" ON "topic" ("user_id");
CREATE INDEX "no_post" ON "post" ("topic_id");
CREATE INDEX "find_user" ON "users" ("username");
```

```
CREATE INDEX "find_topic" ON "topic" ("topic_name");

CREATE INDEX "url_posts" ON "post" ("content" );
CREATE INDEX "find_comment_thread" ON "comment" ("thread_id", "post_id" );

CREATE INDEX "votes_score" ON "votes" ("vote");
```

## 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:

```
INSERT INTO "users" ("username")
  SELECT DISTINCT(username)
  FROM "bad_posts"

  UNION

  SELECT DISTINCT(username)
  FROM "bad_comments"

  UNION

  SELECT REGEXP_SPLIT_TO_TABLE(bad_posts.upvotes, ',')
  FROM "bad_posts"

  UNION
```

```
  SELECT REGEXP_SPLIT_TO_TABLE(bad_posts.downvotes, ',')
  FROM "bad_posts"

  ORDER BY 1 DESC;



INSERT INTO "topic" ("topic_name")
  SELECT DISTINCT("topic")  FROM "bad_posts"
  JOIN "users"
  ON users.username = bad_posts.username;



INSERT INTO "post" ("user_id", "topic_id", "title", "content")

  SELECT users.id ,  topic.id , LEFT(bad_posts.title, 100) ,
  CASE WHEN bad_posts.text_content IS NULL
            THEN bad_posts.url
       WHEN bad_posts.url IS NULL
            THEN bad_posts.text_content END
  FROM "bad_posts"
  JOIN "users"
  ON bad_posts.username = users.username
  JOIN "topic"
  ON bad_posts.topic = topic.topic_name
  WHERE bad_posts.text_content IS NOT NULL OR bad_posts.url IS NOT NULL;



INSERT INTO "comment" ("user_id", "post_id", "comment_content")

  SELECT users.id, post.id, bad_comments.text_content FROM   bad_posts
  JOIN bad_comments
  ON bad_posts.id = bad_comments.post_id
  JOIN users
  ON bad_comments.username = users.username
  JOIN post
  ON post.title = LEFT(bad_posts.title, 100)
  ORDER BY 1;
```

```
INSERT INTO "votes" ("post_id", "user_id", "vote")

    WITH t1 AS(
      SELECT REGEXP_SPLIT_TO_TABLE(bad_posts.upvotes, ',') upvotes,
REGEXP_SPLIT_TO_TABLE(bad_posts.downvotes, ',') downvotes, post.id post_id,
bad_posts.username username
      FROM "bad_posts"
      JOIN post
      ON post.title = LEFT(bad_posts.title, 100))

    SELECT t1.post_id, users.id, 1 AS vote
    FROM t1
    JOIN users
    ON users.username = t1.upvotes

    UNION

    SELECT  t1.post_id, users.id, -1 AS vote
    FROM t1
    JOIN users
    ON users.username = t1.downvotes

    ORDER BY 1;
```

I think the most challenging DQLs in Part II were I had to import data into table “comment” and “votes”. Table “votes” required to create a subtable, JOIN the subtable ON user names taking into account that each table has to be joined separately and then united into one. I also had to modify the users table multiple times to make sure I included all the users, the ones who create topics, posts and also voted. Table “comment” required to JOIN on multiple tables which was not trivial in the beginning.