# 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. Insufficient constraint
   There is no constraint for most of the columns which may cause data duplication and inconsistency. For example, in “bad_comments” table, the “post_id” column do not have a foreign key constraint to reference the “id” column from “bad_posts” table.
2. Choice of datatype
   Some columns would have a better choice of datatype. For example, in “bad_comments” table, the datatype of “post_id” column is BIGINT which can represent a large range of numbers (-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807), it would be excessive in this use case, datatype INT would be a better choice.
3. Denormalized data
   This schema can be better normalized through "normal forms". For example, we can create a username table which each username can be represented by an unique id, then, we replace both “username” column in “bad_posts” and “bad_comments” by “username_id” to make it easier to manage.
4. Index Creation
   Currently, there is only the primary key constraints index created for each table. If there is a need on querying data frequently, the execution time of queries might be improved by creating more indexes.

## 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:

```
--1. users table--
CREATE TABLE "users" (
    "id" SERIAL PRIMARY KEY,
    "username" VARCHAR(25) UNIQUE NOT NULL CHECK (LENGTH(TRIM("username")) > 0 ),
    "last_logged_in" TIMESTAMP
);

CREATE INDEX "username_search" ON "users" ("username");

--2. topics table--
CREATE TABLE "topics" (
    "id" SERIAL PRIMARY KEY,
    "name" VARCHAR(30) UNIQUE NOT NULL CHECK (LENGTH(TRIM("name")) > 0 ),
    "description" VARCHAR(500)
);

CREATE INDEX "topic_name_search" ON "topics" ("name");

--3. posts table--
CREATE TABLE "posts" (
    "id" SERIAL PRIMARY KEY,
    "title" VARCHAR(100) NOT NULL CHECK (LENGTH(TRIM("title")) > 0 ),
    "url" VARCHAR,
    "text_content" VARCHAR,
    "topic_id" INTEGER NOT NULL REFERENCES "topics" ("id") ON DELETE CASCADE,
```

```
    "user_id" INTEGER REFERENCES "users" ("id") ON DELETE SET NULL,
    "created_at" TIMESTAMP
);

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

CREATE INDEX "url_search" ON "posts" ("url");
CREATE INDEX "latest_user_post_search" ON "posts" ("user_id", "created_at");
CREATE INDEX "topic_post_search" ON "posts" ("topic_id", "created_at");

--4. comments table--
CREATE TABLE "comments" (
    "id" SERIAL PRIMARY KEY,
    "text_content" VARCHAR NOT NULL CHECK (LENGTH(TRIM("text_content")) > 0 ),
    "post_id" INTEGER NOT NULL REFERENCES "posts" ("id") ON DELETE CASCADE,
    "user_id" INTEGER REFERENCES "users" ("id") ON DELETE SET NULL,
    "parent_comment_id" INTEGER,
    "created_at" TIMESTAMP
);

CREATE INDEX "parent_comment_search" ON "comments" ("parent_comment_id");
CREATE INDEX "latest_user_comment_search" ON "comments" ("user_id", "created_at");

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

--5. votes table--
CREATE TABLE "votes" (
    "id" SERIAL PRIMARY KEY,
    "user_id" INTEGER REFERENCES "users" ("id") ON DELETE SET NULL,
    "post_id" INTEGER NOT NULL REFERENCES "posts" ("id") ON DELETE CASCADE,
    "vote" SMALLINT CHECK (("vote" = 1) OR ("vote" = -1))
);

ALTER TABLE "votes" ADD CONSTRAINT "vote_once" UNIQUE("user_id", "post_id");

CREATE INDEX "vote_compute" ON "votes" ("post_id", "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:

```
--1. users table--
INSERT INTO "users" ("username")
SELECT DISTINCT "username" FROM "bad_posts"
UNION
SELECT DISTINCT "username" FROM "bad_comments"
UNION
SELECT DISTINCT regexp_split_to_table(upvotes,',') FROM "bad_posts"
UNION
SELECT DISTINCT regexp_split_to_table(downvotes,',') FROM "bad_posts";

--2. topics table--
INSERT INTO "topics" ("name")
SELECT DISTINCT "topic" FROM "bad_posts";

--3. posts table--
INSERT INTO "posts" ("title", "url", "text_content","topic_id","user_id")
SELECT left("title",100), "url", "text_content", "topics"."id", "users"."id"
FROM "bad_posts"
JOIN "topics"
ON "topics"."name" = "bad_posts"."topic"
JOIN "users"
ON "users"."username" = "bad_posts"."username";

--4. comments table--
INSERT INTO "comments" ("text_content","post_id","user_id")
SELECT "bad_comments"."text_content", "posts"."id", "users"."id"
FROM "bad_comments"
JOIN "posts"
```

```
ON "bad_comments"."post_id" = "posts"."id"
JOIN "users"
ON "bad_comments"."username" = "users"."username";

--5. votes table--
INSERT INTO "votes" ("vote","user_id","post_id")
SELECT 1, "users"."id", "t1"."id"
FROM "users"
JOIN (SELECT "id",regexp_split_to_table(upvotes,',') AS "username"
      FROM "bad_posts") AS "t1"
ON "t1"."username" = "users"."username";

INSERT INTO "votes" ("vote","user_id","post_id")
SELECT -1, "users"."id", "t1"."id"
FROM "users"
JOIN (SELECT "id",regexp_split_to_table(downvotes,',') AS "username"
      FROM "bad_posts") AS "t1"
ON "t1"."username" = "users"."username";
```