# 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:

```sql
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, we investigate this schema and some of the sample data in the project's SQL workspace. Three (3) specific things that could be improved about this schema are

```
1.     As the number of tables are less there is a lack of Foreign_Key
constraints this could lead to overloading of the database.

2.     There are no constraints on permitting a user to keep on voting;
this could lead to storage issues.

3.     The 'text_content' field in both the tables could also be a VARCHAR.

4.     'DOWNVOTES' and 'UPVOTES' should be of numeric/int datatype for easy
counting of likes and dislikes.
```

## Part II: Create the DDL for your new schema

Having done this initial investigation and assessment, our goal is to dive deep into the heart of the problem and create a new schema for Udiddit. *Our new schema should at least reflect fixes to the shortcomings we pointed to in the previous exercise.*

1. List of features and specifications that Udiddit needs in order to support its website and administrative interface:
    1. Allow new users to register:
        1. Each username has to be unique
        2. Usernames can be composed of at most 25 characters
        3. Usernames can't be empty
        4. We won't worry about user passwords for this project

```
DROP TABLE IF EXISTS  "users" ;
CREATE TABLE "users"
(
id SERIAL PRIMARY KEY,
username VARCHAR(25) UNIQUE NOT NULL,
last_login TIMESTAMP,
CONSTRAINT "username_not_empty" CHECK (LENGTH(TRIM("username")) > 0)
);
CREATE UNIQUE INDEX "unique_username_index" ON "users" ("username");
```

    2. Allow registered users to create new topics:
        1. Topic names have to be unique.
        2. The topic's name is at most 30 characters
        3. The topic's name can't be empty
        4. Topics can have an optional description of at most 500 characters.

```
DROP TABLE IF EXISTS "topics" ;
CREATE TABLE "topics"
(
id SERIAL PRIMARY KEY,
name VARCHAR(30) UNIQUE NOT NULL,
description VARCHAR(500),
CONSTRAINT "topic_name_not_empty" CHECK (LENGTH(TRIM("name")) > 0)
);
CREATE  INDEX "topic_name_index" ON "topics" ("name");
```

3. Allow registered users to create new posts on existing topics:
    1. Posts have a required title of at most 100 characters
    2. The title of a post can’t be empty.
    3. Posts should contain either a URL or a text content, **but not both**.
    4. If a topic gets deleted, all the posts associated with it should be automatically deleted too.
    5. If the user who created the post gets deleted, then the post will remain, but it will become dissociated from that user.

```
DROP TABLE IF EXISTS "posts" ;
CREATE TABLE "posts"
(
id SERIAL PRIMARY KEY,
title VARCHAR(100) NOT NULL,
created_on TIMESTAMP,
url VARCHAR(500),
text_content VARCHAR,
topic_id INTEGER REFERENCES "topics"("id") ON DELETE CASCADE,
user_id INTEGER REFERENCES "users"("id") ON DELETE SET NULL,
-- CONSTRAINT The title of a post can't be empty added.
CONSTRAINT "title_not_empty" CHECK (LENGTH(TRIM("title")) > 0),
CONSTRAINT "text_or_url" CHECK (("url" IS NOT NULL AND "text_content" IS
NULL) OR ("url" IS NULL AND "text_content" IS NOT NULL))
);
CREATE INDEX ON "posts" ("url");
```

4. Allow registered users to comment on existing posts:
    1. A comment’s text content can’t be empty.
    2. Contrary to the current linear comments, the new structure should allow comment threads at arbitrary levels.
    3. If a post gets deleted, all comments associated with it should be automatically deleted too.
    4. If the user who created the comment gets deleted, then the comment will remain, but it will become dissociated from that user.
    5. If a comment gets deleted, then all its descendants in the thread structure should be automatically deleted too.

```
DROP TABLE IF EXISTS "comments" ;
CREATE TABLE "comments"
(
id SERIAL PRIMARY KEY,
text_content VARCHAR(10000) NOT NULL,
comment_time TIMESTAMP,
old_comment_id INTEGER DEFAULT NULL,
post_id INTEGER REFERENCES "posts"("id") ON DELETE CASCADE,
user_id INTEGER REFERENCES "users"("id") ON DELETE SET NULL,
CONSTRAINT "new_comment" FOREIGN KEY ("old_comment_id")  REFERENCES
"comments" ON DELETE CASCADE,
CONSTRAINT "text_content_not_empty" CHECK(LENGTH(TRIM("text_content")) > 0)
);
```

5. Make sure that a given user can only vote once on a given post:
    1. Hint: we can store the (up/down) value of the vote as the values 1 and -1 respectively.
    2. If the user who cast a vote gets deleted, then all their votes will remain, but will become dissociated from the user.
    3. If a post gets deleted, then all the votes for that post should be automatically deleted too.

```
DROP TABLE IF EXISTS  "votes" ;
CREATE TABLE "votes"
(
user_id INTEGER REFERENCES "users"("id") ON DELETE SET NULL,
post_id INTEGER REFERENCES "posts"("id") ON DELETE CASCADE,
vote SMALLINT NOT NULL,
CONSTRAINT "vote_up_or_down" CHECK("vote"=1 OR "vote"=-1),
CONSTRAINT "unique_vote_per_user" PRIMARY KEY("user_id" , "post_id")
);
```

2. List of sample queries that Udiddit needs in order to support its website and administrative interface.
    1. List all users who haven't logged in in the last year.
    2. List all users who haven't created any post.
    3. Find a user by their username.
    4. List all topics that don't have any posts.
    5. Find a topic by its name.
    6. List the latest 20 posts for a given topic.
    7. List the latest 20 posts made by a given user.
    8. Find all posts that link to a specific URL, for moderation purposes.
    9. List all the top-level comments (those that don't have a parent comment) for a given post.

10. List all the direct children of a parent comment.
11. List the latest 20 comments made by a given user.
12. Compute the score of a post, defined as the difference between the number of upvotes and the number of downvotes

3. We will need to use normalization, various constraints, as well as indexes in our new database schema. We should use named constraints and indexes to make our schema cleaner

```
postgres=# \dt
             List of relations
 Schema |     Name      | Type  |  Owner
--------+--------------+-------+----------
 public | bad_comments | table | postgres
 public | bad_posts     | table | postgres
 public | comments      | table | postgres
 public | posts         | table | postgres
 public | topics        | table | postgres
 public | users         | table | postgres
 public | votes         | table | postgres
```

4. Our new database schema will be composed of five (5) tables that should have an auto-incrementing id as their primary key.

**DDL for new schema:**:

```
DROP TABLE IF EXISTS  "users" ;
CREATE TABLE "users"
(
id SERIAL PRIMARY KEY,
username VARCHAR(25) UNIQUE NOT NULL,
last_login TIMESTAMP,
CONSTRAINT "username_not_empty" CHECK (LENGTH(TRIM("username")) > 0)
);
CREATE UNIQUE INDEX "unique_username_index" ON "users" ("username");

DROP TABLE IF EXISTS "topics" ;
CREATE TABLE "topics"
(
id SERIAL PRIMARY KEY,
name VARCHAR(30) UNIQUE NOT NULL,
description VARCHAR(500),
CONSTRAINT "topic_name_not_empty" CHECK (LENGTH(TRIM("name")) > 0)
```

```sql
);
CREATE  INDEX "topic_name_index" ON "topics" ("name");

DROP TABLE IF EXISTS "posts" ;
CREATE TABLE "posts"
(
id SERIAL PRIMARY KEY,
title VARCHAR(100) NOT NULL,
created_on TIMESTAMP,
url VARCHAR(500),
text_content  VARCHAR,
topic_id INTEGER REFERENCES "topics"("id") ON DELETE CASCADE,
user_id INTEGER REFERENCES "users"("id") ON DELETE SET NULL,
--CONSTRAINT The title of a post can’t be empty added.
CONSTRAINT "title_not_empty" CHECK (LENGTH(TRIM("title")) > 0),
CONSTRAINT "text_or_url" CHECK (("url" IS NOT NULL AND "text_content" IS
NULL) OR ("url" IS NULL AND "text_content" IS NOT NULL))
);
CREATE INDEX ON "posts" ("url");

DROP TABLE IF EXISTS "comments" ;
CREATE TABLE "comments"
(
id SERIAL PRIMARY KEY,
text_content VARCHAR(10000) NOT NULL,
comment_time TIMESTAMP,
old_comment_id INTEGER DEFAULT NULL,
post_id INTEGER REFERENCES "posts"("id") ON DELETE CASCADE,
user_id INTEGER REFERENCES "users"("id") ON DELETE SET NULL,
CONSTRAINT "new_comment" FOREIGN KEY ("old_comment_id")  REFERENCES "comments" ON
DELETE CASCADE,
CONSTRAINT "text_content_not_empty" CHECK(LENGTH(TRIM("text_content")) > 0)
);

DROP TABLE IF EXISTS  "votes" ;
CREATE TABLE "votes"
(
user_id INTEGER REFERENCES "users"("id") ON DELETE SET NULL,
post_id INTEGER REFERENCES "posts"("id") ON DELETE CASCADE,
vote SMALLINT NOT NULL,
CONSTRAINT "vote_up_or_down" CHECK("vote"=1 OR "vote"=-1),
CONSTRAINT "unique_vote_per_user" PRIMARY KEY("user_id" , "post_id")
);
```

## Part III: Migrate the provided data

Now that our new schema is created, it's time to migrate the data from the provided schema in the project's SQL Workspace to our own schema. This will allow us to review some DML and DQL concepts, as we will be using INSERT...SELECT queries to do so.

Topic descriptions can all be empty

1. Since the bad_comments table doesn't have the threading feature, we can migrate all comments as top-level comments, i.e. without a parent
2. We can use the Postgres string function **regexp_split_to_table** to unwind the comma-separated votes values into separate rows
3. Don't forget that some users only vote or comment, and haven't created any posts. We'll have to create those users too.
4. The order of our migrations matter! For example, since posts depend on users and topics, we'll have to migrate the latter first.
5. Tip: We can start by running only SELECTs to fine-tune our queries, and use a LIMIT to avoid large data sets. Once we know we have the correct query, we can then run our full INSERT...SELECT query.
6. **NOTE**: The data in our SQL Workspace contains thousands of posts and comments. The DML queries may take at least 10-15 seconds to run.

**DML to migrate the current data in bad_posts and bad_comments to our new database schema:**

```
-- migrate to 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, ',') AS "username"   FROM
bad_posts
  UNION
  SELECT DISTINCT regexp_split_to_table(downvotes, ',')AS "username"   FROM
bad_posts
  order by username;
```

```sql
-- migrate to topics table
INSERT INTO "topics"("name")
 SELECT DISTINCT topic FROM bad_posts;

-- migrate to posts table
INSERT INTO "posts"("title","url","text_content","topic_id","user_id")
SELECT LEFT(bp.title,
100),bp.url::VARCHAR(500),bp.text_content::VARCHAR,t.id,u.id
FROM topics  t
INNER JOIN bad_posts bp ON bp.topic = t.name
INNER JOIN users u ON bp.username = u.username;

-- migrate to comments table
INSERT INTO "comments"( "post_id", "user_id", "text_content")
SELECT  p.id,  u.id,  bc.text_content::VARCHAR
FROM bad_comments bc
JOIN users u ON bc.username = u.username
JOIN posts p ON p.id = bc.post_id;

-- migrate to votes table
INSERT INTO "votes" ("post_id","user_id","vote")
SELECT bp.id, u.id,1 AS vote_up
FROM (SELECT id, REGEXP_SPLIT_TO_TABLE(upvotes,',')
  AS upvote_users FROM bad_posts) bp
JOIN users u ON u.username=bp.upvote_users;

INSERT INTO "votes"("post_id","user_id","vote")
SELECT bp.id, u.id,-1 AS vote_down
FROM (SELECT id, REGEXP_SPLIT_TO_TABLE(downvotes,',')
  AS downvote_users FROM bad_posts) bp
JOIN users u ON u.username=bp.downvote_users;

--drop  tables
DROP TABLE bad_comments;
DROP TABLE bad_posts;
```