# 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 all of this data in two tables is not very wise specially according to the amount of data that a social news aggregator has to store. For instance, users and topics at least should be considered separate tables according to me.

2) The upvotes and downvotes columns in “bad_posts” are text with the names of the usernames. Username should be defined as serial fields with its own primary key. Furthermore, upvotes and downvotes in the bad_posts table should refer to these values and username should be a FOREIGN KEY in the bad_posts table.

3) The upvotes and downvotes columns stores multiple values of usernames. Following the first normal form rule, the table should have several rows to define who has put an upvote or downvote. Furthermore, this functionality should be limited otherwise there can be infinite upvotes and downvotes per post which means it is not a robust database schema since we disk space can be saturated due to this.

4) Having 4000 character as a limit for URL is way too much, it means that we are allowing memory in a column that we know should be much smaller in terms of characters. (Normally URLs are much shorter)

5) The text_content within bad_comments has no limit of characters which is risky since it can saturate the disk space which is always limited in real world applications.

6) Add which username starts a topic since this can be information that is valuable but it is currently not contemplated in the data schema.

7) There are no Foreign Key or Custom constraints in this schema which key to making sure that the business rules are taking into account in the database and we have consistent data stored in the database and therefore a robust social new aggregator

## 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:

```
a) Users Table

CREATE TABLE "users"
```

```
(
	"id" BIGSERIAL CONSTRAINT "users_pk" PRIMARY KEY,
	"username" VARCHAR(25) CONSTRAINT "usernames_not_null" NOT NULL,
	"creation_date" TIMESTAMP,
	"last_login" TIMESTAMP,
	CONSTRAINT "usernames_not_empty" CHECK (LENGTH(TRIM("username")) > 0)
);

CREATE UNIQUE INDEX "unique_username" ON "users"(TRIM("username"));


b) Topics table

CREATE TABLE "topics"
(
	"id" BIGSERIAL CONSTRAINT "topics_pk" PRIMARY KEY,
	"user_id" BIGINT DEFAULT NULL,
	"topic" VARCHAR(30) CONSTRAINT "topic_not_null" NOT NULL,
	"topic_description" VARCHAR(500) DEFAULT NULL,
	"creation_date" TIMESTAMP,
	CONSTRAINT "topics_not_empty" CHECK (LENGTH(TRIM("topic")) > 0)
);

CREATE UNIQUE INDEX "unique_topic" ON "topics"(TRIM("topic"));

c) Posts table

CREATE TABLE "posts"
(
	"id" BIGSERIAL CONSTRAINT "posts_pk" PRIMARY KEY,
	"user_id" BIGINT,
	"topic_id" BIGINT CONSTRAINT "topic_id_not_null" NOT NULL,
	"title" VARCHAR(100) CONSTRAINT "title_not_null" NOT NULL,
	"url" VARCHAR(200) DEFAULT NULL,
	"text_content" VARCHAR(2000) DEFAULT NULL,
	"creation_date" TIMESTAMP,
	CONSTRAINT "title_not_empty" CHECK (LENGTH(TRIM("title")) > 0),
	CONSTRAINT "users_fk_on_posts" FOREIGN KEY ("user_id") REFERENCES "users" ON
DELETE SET NULL,
	CONSTRAINT "topics_fk_on_posts" FOREIGN KEY ("topic_id") REFERENCES "topics"
ON DELETE CASCADE,
	CONSTRAINT "url_or_text_content" CHECK (NOT ("url" IS NULL AND
"text_content" IS NULL) AND NOT (LENGTH("url") > 0 AND LENGTH("text_content") > 0))
);

CREATE INDEX "posts_user_id_index" ON "posts"("user_id");
CREATE INDEX "posts_title_index" ON "posts"("title");
CREATE INDEX "posts_topic_id_index" ON "posts"("topic_id");
CREATE INDEX "posts_url_index" ON "posts"("url");

d) Comments table
```

```
CREATE TABLE "comments"
(
	"id" BIGSERIAL CONSTRAINT "comments_pk" PRIMARY KEY,
	"user_id" BIGINT,
	"post_id" BIGINT CONSTRAINT "post_id_not_null" NOT NULL,
	"parent_comment_id" BIGINT,
	"text_content" VARCHAR(2000) CONSTRAINT "text_content_not_null" NOT NULL,
	"creation_date" TIMESTAMP,
	CONSTRAINT "text_content_not_empty" CHECK (LENGTH(TRIM("text_content")) >
0),
	CONSTRAINT "posts_fk_on_comments" FOREIGN KEY ("post_id") REFERENCES "posts"
ON DELETE CASCADE,
	CONSTRAINT "users_fk_on_comments" FOREIGN KEY ("user_id") REFERENCES "users"
ON DELETE SET NULL,
	CONSTRAINT "parent_fk_on_comments_to_child" FOREIGN KEY
("parent_comment_id") REFERENCES "comments" ON DELETE CASCADE
);

CREATE INDEX "comments_user_id_index" ON "comments"("user_id");
CREATE INDEX "comments_post_id_index" ON "comments"("post_id");
CREATE INDEX "comments_parent_comment_id_index" ON "comments"("parent_comment_id");

e) Votes table

CREATE TABLE "votes"
(
	"id" BIGSERIAL CONSTRAINT "votes_pk" PRIMARY KEY,
	"user_id" BIGINT,
	"post_id" BIGINT CONSTRAINT "post_id_not_null" NOT NULL,
	"vote" SMALLINT,
	"creation_date" TIMESTAMP,
	CONSTRAINT "vote_values_1_or_-1" CHECK ("vote" IN (-1,1)),
	CONSTRAINT "posts_fk_on_votes" FOREIGN KEY ("post_id") REFERENCES "posts" ON
DELETE CASCADE,
	CONSTRAINT "users_fk_on_votes" FOREIGN KEY ("user_id") REFERENCES "users" ON
DELETE SET NULL
);

CREATE INDEX "votes_user_id_index" ON "votes"("user_id");
CREATE INDEX "votes_post_id_index" ON "votes"("post_id");
CREATE INDEX "votes_vote_index" ON "votes"("vote");
```

SELECT bad_posts.topic FROM “bad_posts”; With this query we can see that there are repeated topics and therefore suggest to add the username that first creates the topic since this is useful information.

SELECT url, length(url) as url_length from “bad_posts WHERE url IS NOT NULL ORDER BY url_length DESC; The longest URL is 24 characters so we will put a maximum limit of 200 just in case

SELECT text_content, length(text_content) as text_content _length from “bad_posts WHERE text_content IS NOT NULL ORDER BY text_content _length DESC; The longest URL is 632 characters so we will put a maximum limit of 2000 just in case.

## 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 b_p.username
      FROM "bad_posts" AS b_p

      UNION

      SELECT regexp_split_to_table(b_p.upvotes,',')
      FROM "bad_posts" AS b_p

      UNION

      SELECT regexp_split_to_table(b_p.downvotes,',')
      FROM "bad_posts" AS b_p

      UNION
```

```
      SELECT b_c.username
      FROM "bad_comments" AS b_c;


INSERT INTO "topics"("topic")

      SELECT DISTINCT topic
      FROM bad_posts;

INSERT INTO "posts"("id", "user_id", "topic_id", "title", "url",
"text_content")

      SELECT b_p.id, u.id, t.id, b_p.title::VARCHAR(100),
b_p.url::VARCHAR(200), b_p.text_content::VARCHAR(2000)
      FROM bad_posts AS b_p
      INNER JOIN topics AS t
      ON b_p.topic = t.topic
      INNER JOIN users AS u
      ON b_p.username = u.username;


INSERT INTO "comments"("id", "user_id", "post_id", "text_content")

      SELECT b_c.id, u.id, b_c.post_id, b_c.text_content::VARCHAR(2000)
      FROM bad_comments AS b_c
      INNER JOIN users AS u
      ON b_c.username=u.username;


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

      SELECT users_select.user_id, split_username_votes.post_id,
split_username_votes.vote

      FROM (
            (
            SELECT b_p.id AS post_id,
                  regexp_split_to_table(b_p.upvotes,',') AS vote_usernames,
                  1::SMALLINT AS vote
            FROM bad_posts AS b_p

            UNION ALL

            SELECT b_p.id AS post_id,
                  regexp_split_to_table(b_p.downvotes,',') AS
```

```
vote_usernames,
                -1::SMALLINT AS vote
            FROM bad_posts AS b_p
            ) AS split_username_votes


            INNER JOIN
            (
            SELECT u.id AS user_id, u.username
            FROM users AS u
            ) AS users_select

            ON users_select.username = split_username_votes.vote_usernames
            );
```

Since based on this:
https://stackoverflow.com/questions/35627923/union-versus-select-distinct-and-union-all-performance

SELECT & UNION performs better than SELECT DISTINCT & UNION ALL and these two both do the same as what we need we will use the first combination for filling in the users table based on the union of, usernames in.

**Queries written in Postgres:**

1) SELECT topic FROM “bad_posts” where topic=”Outdoors”

Here came to realase that topic is a repeated column and therefore, it would be interesting to add in the data schema for instance which User started a topic. This is not included in the current schema given in part 1)

2) SELECT url FROM “bad_posts”;

We can see that URLs don’t need 4000 Characters