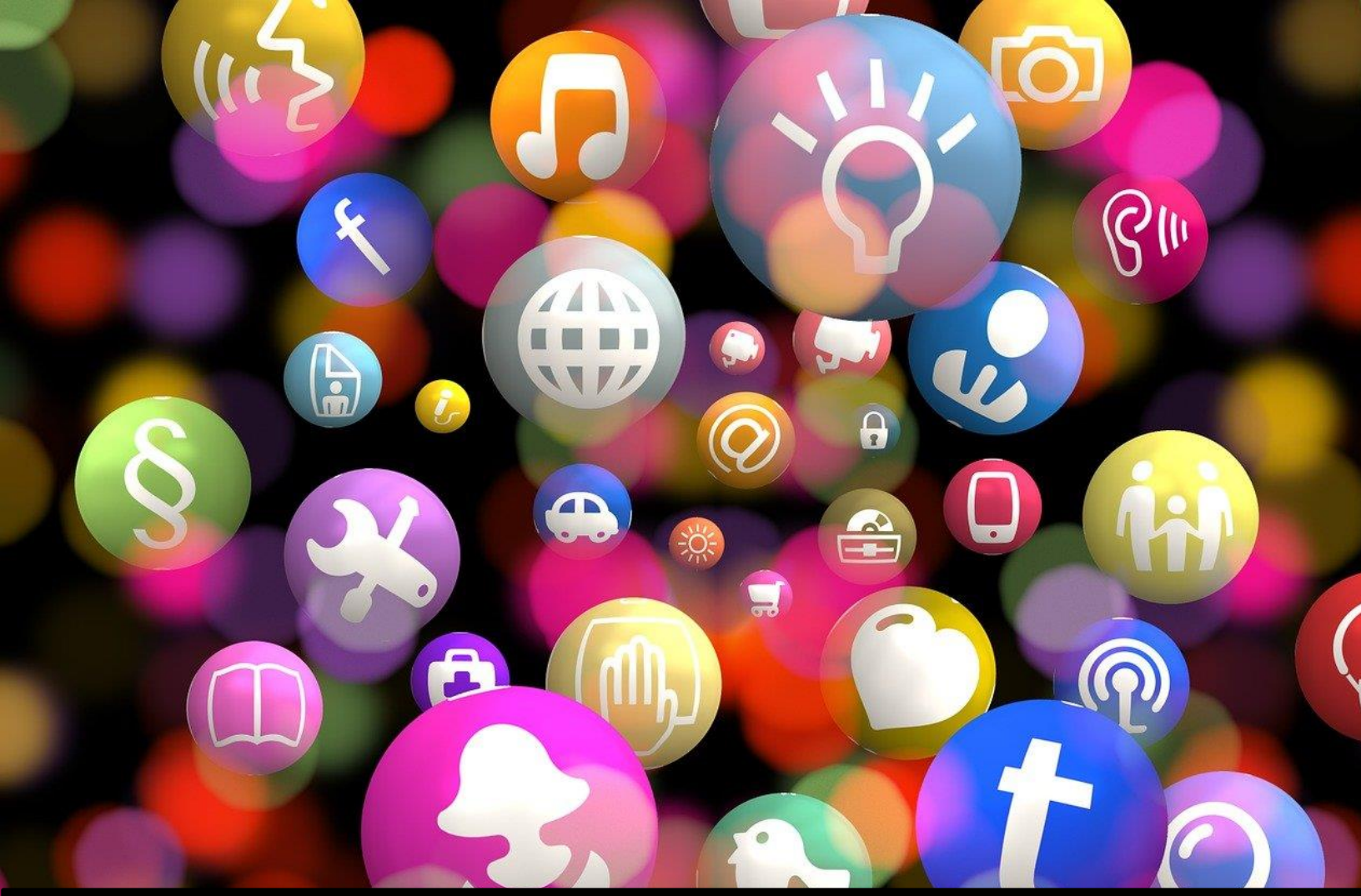

Udiddit, a social news aggregator

By Miroslava Chrienova

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!

Issues observed in the existing schema:

1. Existing schema is missing proper structure and data division in separate tables. It is suggested to add more tables e.g. for users, topics. Proper structure prevents database overloading.
2. Tables lack constraints:
    a. UNIQUE constraint which makes sure that values are not repeated,
    b. FOREIGN KEY constraint that makes sure that column contains only values present in other column (in the same or different table),
    c. CHECK constraints - allowing to implement custom business rules at the level of the database.
3. For quick search, adding a reasonable amount of indexing is advisable.
4. In bad_posts table, following changes are recommended:
    a. Change "upvotes" DATA TYPES from TEXT to NUMERIC.
    b. Change "downvotes" DATA TYPES from TEXT to NUMERIC.
5. In bad_comments table, following changes are recommended:
    a. "username" should be UNIQUE and NOT NULL.
    b. "text_content" should be changed from TEXT to VARCHAR with a limited number of characters.
    c. "post_id" should be UNIQUE and NOT NULL.

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:

Guideline #1 Solutions

5. **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 TABLE "users"(
"id" SERIAL PRIMARY KEY,
"username" VARCHAR(25) UNIQUE NOT NULL,
"last_login" TIMESTAMP,
CONSTRAINT "no_empty_username" CHECK (LENGTH(TRIM("username")) > 0)
);

CREATE INDEX "username_index" ON "users" ("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,
"topic_name" VARCHAR(30) NOT NULL,
"description" VARCHAR(500)  DEFAULT NULL
);

CREATE UNIQUE INDEX "unique_topics" ON "topics"( TRIM("topic_name"));
```

```
CREATE INDEX ON "topics" (LOWER("topic_name") VARCHAR_PATTERN_OPS);
```

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,
"title" VARCHAR(100) NOT NULL,
"url" VARCHAR(1000) DEFAULT NULL,
"text_content" VARCHAR(5000) DEFAULT NULL,
"created_on" TIMESTAMP,
"topic_id" INTEGER,
"user_id" INTEGER,
CONSTRAINT "text_or_url" CHECK (("url" IS NOT NULL AND "text_content" IS
NULL) OR ("url" IS NULL AND "text_content" IS NOT NULL)),
CONSTRAINT "fk_topic_id" FOREIGN KEY ("topic_id") REFERENCES "topics"
("id") ON DELETE CASCADE,
CONSTRAINT "fk_user_id" 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" INTEGER,
"post_id" INTEGER,
"parent_id" INTEGER DEFAULT NULL,
"comment_text" VARCHAR(10000) NOT NULL,
"comment_time" TIMESTAMP,
CONSTRAINT "fk_post_id" FOREIGN KEY ("post_id") REFERENCES "posts" ("id")
ON DELETE CASCADE,
CONSTRAINT "fk_user_id" FOREIGN KEY ("user_id") REFERENCES "users" ("id")
ON DELETE SET NULL,
CONSTRAINT "child_comment" FOREIGN KEY ("parent_id") REFERENCES "comments"
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"(
"vote" SMALLINT CHECK ("vote" = '1' OR "vote" = '-1'),
"user_id" INTEGER,
"post_id" INTEGER,

CONSTRAINT "pk_user_votes" PRIMARY KEY ("post_id", "user_id"),
CONSTRAINT "fk_user_id" FOREIGN KEY ("user_id") REFERENCES "users" ("id")
ON DELETE SET NULL,
CONSTRAINT "kf_post_id" FOREIGN KEY ("post_id") REFERENCES "posts" ("id")
ON DELETE CASCADE
);
```

Guideline #4

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

```
postgres=# \dt
          List of relations
 Schema |   Name   | Type  |  Owner
--------+----------+-------+----------
 public | comments | table | postgres
 public | posts    | table | postgres
 public | topics   | table | postgres
 public | users    | table | postgres
 public | votes    | table | postgres
(5 rows)
```

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:

8. 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

```
-- migration of users from bad_posts, bad_comments, upvotes and downvotes
INSERT INTO users("username") (SELECT DISTINCT regexp_split_to_table
(upvotes, ',' ) as usernames FROM bad_posts
UNION
SELECT DISTINCT regexp_split_to_table (downvotes, ',' ) as usernames FROM
bad_posts
UNION
SELECT DISTINCT “username” from bad_posts
UNION
SELECT DISTINCT “username” from bad_comments
);

-- migrating comments
INSERT INTO comments ("comment_text") SELECT text_content FROM
bad_comments;

-- insert data into topics table
INSERT INTO "topics" ("topic_name") SELECT DISTINCT topic FROM bad_posts;
```

```
--migration of posts

--adding extra columns for topic and users
ALTER TABLE posts ADD COLUMN "topic_from_bad_posts" VARCHAR;
ALTER TABLE posts ADD COLUMN "user_from_bad_posts" VARCHAR;

--updating ids
INSERT INTO "posts" ("title", "url", "text_content",
"topic_from_bad_posts", "user_from_bad_posts")SELECT substring(title, 1,
100), url, text_content, topic, username FROM bad_posts;

UPDATE posts SET "topic_id" = (SELECT "id" FROM "topics" WHERE
```

```
"topics"."topic_name" = "posts"."topic_from_bad_posts");
UPDATE posts SET "user_id" = (SELECT "id" FROM "users" WHERE
"users"."username" = "posts"."user_from_bad_posts");

--alter table by dropping columns
ALTER TABLE posts DROP COLUMN "topic_from_bad_posts";
ALTER TABLE posts DROP COLUMN "user_from_bad_posts";


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


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

```
--migrate comments (joining tables posts, users, bad_comments)
INSERT INTO "comments"(user_id, post_id, comment_text)
SELECT p.user_id, p.id, bc.text_content
FROM bad_comments bc
JOIN posts p
ON p.id = bc.post_id
JOIN users u
ON p.user_id = u.id;

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