Udiddit, a social news aggregator

Introduction About Udiddit

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, 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. In an overview of the original database schema, it feels like both tables are lack of logic and clarity. The structure among different features (fields) is badly organized and is hard to mainitain, let alone serve for different business purposes. One way to fix this is to split them into more than two tables. For example, we can create tables such as `users`, `votes`, `comments`, `posts` etc. to classify, and store and manage these data separately. The reason for doing so is also to conform to the **1NF**, **2NF** and **3NF** of designing databases.
2. The data types for some fields are not reasonable. For example, the `upvotes` and `downvotes` in `bad_posts` should be `INTEGER` like `+1` or `-1`. The reason for doing so is for convenient arithmetic calculations in analytical process.
3. The two tables in the original schema don't have any other `CONSTRAINT` other than `PRIMARY KEY`.
    - `REFERENCES` or `FOREIGN KEY` on some of the common fields held by each table shold be added for effective interactivity. Meanwhile, some business scenarios such as what if we delete a post or a user or a comment, should be considered and planned carefully.
    - `UNIQUE` constraint should be added in order to prevent repetition. For example, the `username` should be `UNIQUE`.
    - `CHECK` constraint should be added for more complex and flexible business purposes.
    - Other `CONSTRAINT` such as `NULL` or `NOT NULL`, case-insensitivity ect. should also be considered. For example, `username` should be `NOT NULL`; a post `title` should be case-insensitive.
4. For better performance, we may also consider adding `INDEX` on some of the fields for fast query according to our query plan e.g. `user_name`, `topic_name`etc..

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.

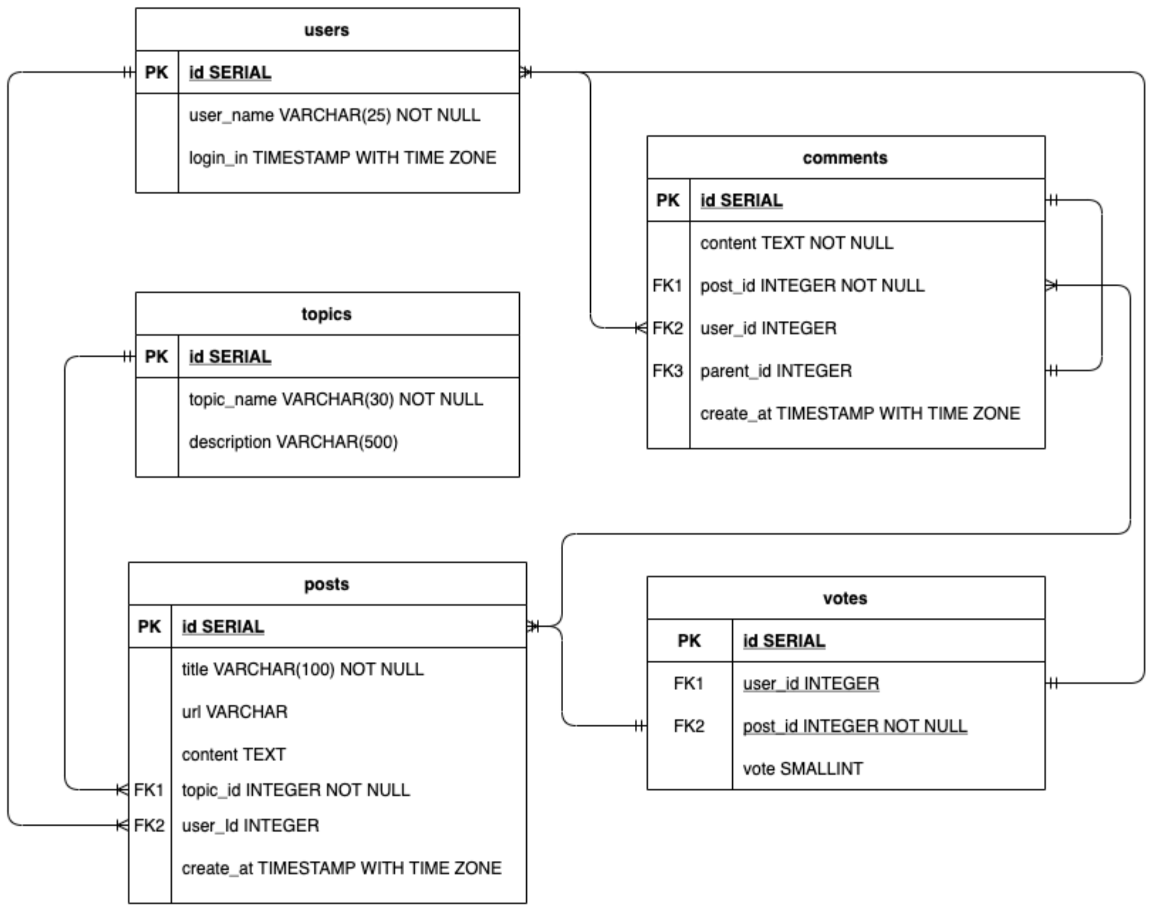
   d. Allow registered users to comment on existing posts:
      i. A comment's text content can't be empty.
      ii. Contrary to the current linear comments, the new structure should allow comment threads at arbitrary levels.
      iii. If a post gets deleted, all comments associated with it should be 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.

From Guideline #1, we can get a brief ERD as follows,

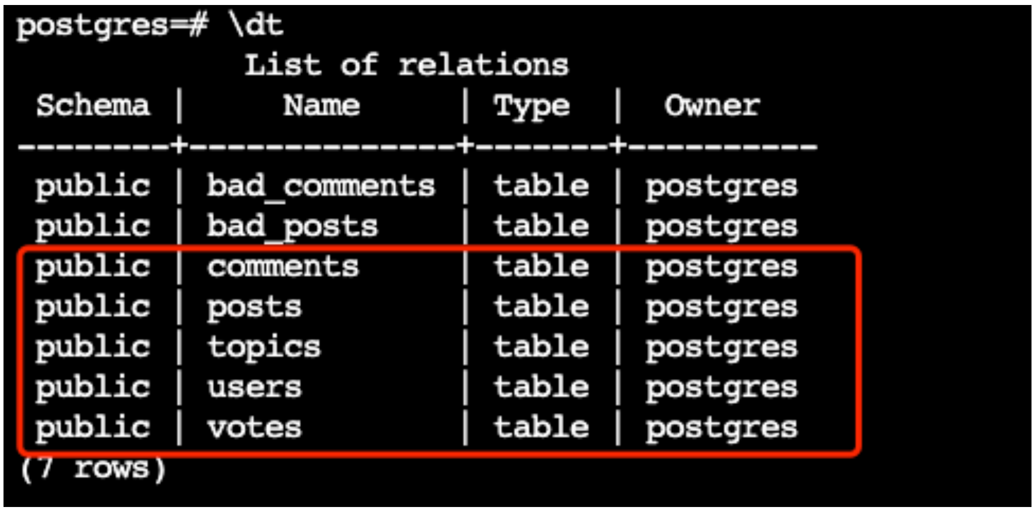

users
PK id SERIAL
user_name VARCHAR(25) NOT NULL
login_in TIMESTAMP WITH TIME ZONE
comments
PK id SERIAL
content TEXT NOT NULL
FK1 post_id INTEGER NOT NULL
FK2 user_id INTEGER
FK3 parent_id INTEGER
create_at TIMESTAMP WITH TIME ZONE
topics
PK id SERIAL
topic_name VARCHAR(30) NOT NULL
description VARCHAR(500)
posts
PK id SERIAL
title VARCHAR(100) NOT NULL
url VARCHAR
content TEXT
FK1 topic_id INTEGER NOT NULL
FK2 user_Id INTEGER
create_at TIMESTAMP WITH TIME ZONE
votes
PK id SERIAL
FK1 user_id INTEGER
FK2 post_id INTEGER NOT NULL
vote SMALLINT

Once you've taken the time to think about your new schema, write the DDL for it in the space provided here:

```
DROP TABLE IF EXISTS "users" CASCADE;
DROP TABLE IF EXISTS "topics" CASCADE;
DROP TABLE IF EXISTS "posts" CASCADE;
DROP TABLE IF EXISTS "comments" CASCADE;
DROP TABLE IF EXISTS "votes" CASCADE;

-- Guidline #1: a. Create "users" table
CREATE TABLE "users" (
  "id" SERIAL PRIMARY KEY,
  "user_name" VARCHAR(25) NOT NULL,
  "login_in" TIMESTAMP WITH TIME ZONE,
  CONSTRAINT "unique_user_name" UNIQUE ("user_name"),
  CONSTRAINT "non_empty_user_name" CHECK(LENGTH(TRIM("user_name"))>0)
 );
CREATE INDEX "user_name_index" ON "users" ("user_name");


-- Guidline #1: b. Create "topics" table
CREATE TABLE "topics" (
  "id" SERIAL PRIMARY KEY,
```

```sql
  "topic_name" VARCHAR(30) NOT NULL,
  "description" VARCHAR(500),
  CONSTRAINT "unique_topic_name" UNIQUE ("topic_name"),
  CONSTRAINT "non_empty_topic_name"
CHECK(LENGTH(TRIM("topic_name"))>0)
 );
CREATE INDEX "topic_name_index" ON "topics" ("topic_name");


-- Guidline #1: c.
CREATE TABLE "posts" (
  "id" SERIAL PRIMARY KEY,
  "title" VARCHAR(100) NOT NULL,
  "url" VARCHAR,
  "content" TEXT,
  "topic_id" INTEGER NOT NULL,
  "user_id" INTEGER,
  "created_at" TIMESTAMP WITH TIME ZONE,

  CONSTRAINT "non_empty_title" CHECK(LENGTH(TRIM("title"))>0),

  CONSTRAINT "url_or_text_check"
    CHECK ((LENGTH(TRIM("url"))=0 AND LENGTH(TRIM("content"))!=0) OR
           (LENGTH(TRIM("url"))!=0 AND LENGTH(TRIM("content"))=0)),

  CONSTRAINT "fkey_topic_id"
    FOREIGN KEY ("topic_id") REFERENCES "topics" ON DELETE CASCADE,

  CONSTRAINT "fkey_user_id"
    FOREIGN KEY ("user_id") REFERENCES "users" ON DELETE SET NULL
);
CREATE INDEX "latest_posts_given_topic" ON "posts" ("topic_id",
"created_at");
CREATE INDEX "latest_posts_given_user" ON "posts" ("user_id",
"created_at");
CREATE INDEX "url_moderation" ON "posts" ("url");


-- Guidline #1: d. Create "comments" table
CREATE TABLE "comments" (
  "id" SERIAL PRIMARY KEY,
  "content" TEXT NOT NULL,
  "post_id" INTEGER NOT NULL,
  "user_id" INTEGER,
  "parent_id" INTEGER,
  "created_at" TIMESTAMP WITH TIME ZONE,

  CONSTRAINT "non_empty_comment_text" CHECK (LENGTH(TRIM("content"))>0),

  CONSTRAINT "fkey_post_id"
    FOREIGN KEY ("post_id") REFERENCES "posts" ON DELETE CASCADE,

  CONSTRAINT "fkey_user_id"
    FOREIGN KEY ("user_id") REFERENCES "users" ON DELETE SET NULL,

  CONSTRAINT "fkey_comment_id"
    FOREIGN KEY ("parent_id") REFERENCES "comments" ON DELETE CASCADE
 );
CREATE INDEX "comments_given_parent" ON "comments" ("parent_id", "id");
CREATE INDEX "comments_given_user" ON "comments" ("user_id", "created_at");


 -- Guidline #1: e. Create "votes" table
CREATE TABLE "votes" (
  "id" SERIAL PRIMARY KEY,
  "user_id" INTEGER,
  "post_id" INTEGER NOT NULL,
  "vote" SMALLINT,

  CONSTRAINT "uniqe_votes" UNIQUE ("user_id", "post_id"),

  CONSTRAINT "fkey_user_id"
    FOREIGN KEY ("user_id") REFERENCES "users" ON DELETE SET NULL,

  CONSTRAINT "fkey_post_id"
    FOREIGN KEY ("post_id") REFERENCES "posts" ON DELETE CASCADE,

  CONSTRAINT "vote_value_check" CHECK ("vote"= 1 OR "vote"= -1)
);
```

After executing the codes above, we can get five new table schema as follows,



```
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
(7 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 `SELECT`s 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:

```sql
-- I. Migrate data into "users" from "bad_posts", "bad_comments"
INSERT INTO "users" ("user_name")
  (SELECT DISTINCT REGEXP_SPLIT_TO_TABLE("upvotes", ',') "user_name"
   FROM "bad_posts"

   UNION

   SELECT DISTINCT REGEXP_SPLIT_TO_TABLE("downvotes", ',') "user_name"
   FROM "bad_posts"

   UNION

   SELECT DISTINCT "username"
   FROM "bad_posts"

```

```
    UNION

    SELECT DISTINCT "username"
    FROM "bad_comments"
     );



-- II. Migrate data into "topics" table from "bad_posts"
INSERT INTO "topics" ("topic_name")
  SELECT DISTINCT topic FROM bad_posts;



-- III. Migrate data into "posts" table from "bad_posts"
INSERT INTO "posts" ("title",
                     "url",
                     "content",
                     "topic_id",
                     "user_id")
  SELECT SUBSTR("title",1,100),
         "bp"."url",
         "bp"."text_content",
         "tp"."id",
         "u"."id"
  FROM "bad_posts" "bp"
  JOIN "topics" "tp" ON "tp"."topic_name"="bp"."topic"
  JOIN "users" "u" ON "u"."user_name"="bp"."username";



-- IV. Migrate data into "votes" from "bad_posts" table
-- Extract all users who voted 'like'
INSERT INTO "votes" ("user_id", "post_id", "vote")
  SELECT "users"."id",
         "sub1"."post_id",
         1
  FROM (
        SELECT REGEXP_SPLIT_TO_TABLE("upvotes", ',') upvoters,
               "id" AS "post_id"
        FROM "bad_posts") sub1
  JOIN "users" ON "sub1"."upvoters"="users"."user_name";

-- Extract all users who voted 'dislike'
INSERT INTO "votes" ("user_id", "post_id", "vote")
  SELECT "users"."id",
         "sub1"."post_id",
```

```sql
          -1
  FROM (
        SELECT REGEXP_SPLIT_TO_TABLE("downvotes", ',') downvoters,
               "id" AS "post_id"
        FROM "bad_posts") sub1
  JOIN "users" ON "sub1"."downvoters"="users"."user_name";

-- V. Migrate data into "comments" from "bad_comments"
INSERT INTO "comments" ("content", "post_id", "user_id")
  SELECT "bad_comments"."text_content",
         "posts"."id",
         "users"."id"
  FROM "bad_comments"
  JOIN "posts" ON "posts"."id"="bad_comments"."post_id"
  JOIN "users" ON "users"."user_name"="bad_comments"."username";

-- Drop the original bad schema
DROP TABLE IF EXISTS "bad_posts";
DROP TABLE IF EXISTS "bad_comments";
```