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. Each cell must contain atomic values. Table **bad_posts** *upvotes* & *downvotes* columns violate First Normal Form. Each cell contains multiple usernames. We must split the *upvotes* and *downvotes* usernames into multiple rows to bring the table into First Normal Form.

2. *Text_content* column of the **bad_posts** table uniquely identifies *upvotes* and *downvotes*, and *text_content* is not the primary key. Therefore, it violates the Third Normal Form which says 'No Transitive dependencies'. We will put them in separate tables.

3. *Username* column is used to reference bad_posts in the bad_comments table which creates update and delete anomalies. We should create a separate table for users and use the user_id foreign key in the bad_comments to reference bad_posts.

4. These tables do not use any constraints. We must use Foreign Key, Primary Key, Check & Unique Key constraints to maintain referential integrity, to avoid having multiple users with the same username and to maintain data consistency.

5. Multiple entities are merged into the **bad_posts** table. We must create separate tables for each entities to avoid redundancies, to streamline the processes and to make our database scalable and performant.

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:

```sql
-- 1. Users Table
CREATE TABLE "users" (
 "id" SERIAL PRIMARY KEY,
 "username" VARCHAR(25) NOT NULL,
 "last_session" TIMESTAMPTZ NOT NULL DEFAULT NOW(),
 CONSTRAINT "unique_username" UNIQUE ("username"),
 CONSTRAINT "username_must_exist" CHECK(LENGTH(TRIM("username")) > 0)
);


--- 2. Topics Table
CREATE TABLE "topics" (
 "id" SERIAL PRIMARY KEY NOT NULL,
 "name" VARCHAR(30) NOT NULL,
 "description" VARCHAR(500),
 "created_at" TIMESTAMPTZ NOT NULL DEFAULT NOW(),
 "user_id" INT,
 CONSTRAINT "topic_name_must_be_unique" UNIQUE ("name"),
 CONSTRAINT "topic_name_must_exist" CHECK(LENGTH(TRIM("name")) > 0),
 CONSTRAINT "fk_user" FOREIGN KEY("user_id") REFERENCES "users" ("id")
ON DELETE SET NULL
);


--- 3. Posts Table
CREATE TABLE "posts" (
 "id" SERIAL PRIMARY KEY NOT NULL,
 "title" VARCHAR(100) NOT NULL,
 "url" VARCHAR(500),
 "text" TEXT,
 "created_at" TIMESTAMPTZ NOT NULL DEFAULT NOW(),
 "topic_id" INT,
 "user_id" INT,
```

```
 CONSTRAINT "title_must_exist" CHECK(LENGTH(TRIM("title")) > 0),
 CONSTRAINT "url_or_text" CHECK(("text" IS NOT NULL AND "url" IS NULL)
OR ("url" IS NOT NULL AND "text" IS NULL)),
 CONSTRAINT "fk_topic" FOREIGN KEY("topic_id") REFERENCES "topics"
("id") ON DELETE CASCADE,
 CONSTRAINT "fk_user" FOREIGN KEY("user_id") REFERENCES "users" ("id")
ON DELETE SET NULL
);

--- Post table's index on title
CREATE INDEX "idx_post_title" ON "posts" ("title");


--- 4. Comments Table
CREATE TABLE "comments" (
 "id" SERIAL PRIMARY KEY NOT NULL,
 "content" TEXT NOT NULL,
 "posted_at" TIMESTAMPTZ NOT NULL DEFAULT NOW(),
 "user_id" INT,
 "post_id" INT,
 "p_comment_id" INT,
 CONSTRAINT "comment_must_exist" CHECK (LENGTH(TRIM("content")) > 0),
 CONSTRAINT "fk_user" FOREIGN KEY("user_id") REFERENCES "users" ("id")
ON DELETE SET NULL,
 CONSTRAINT "fk_post" FOREIGN KEY("post_id") REFERENCES "posts" ("id")
ON DELETE CASCADE,
 CONSTRAINT "fk_comment" FOREIGN KEY("p_comment_id") REFERENCES
"comments" ("id") ON DELETE CASCADE
);

--- Comments table's index on parent id (p_comment_id)
CREATE INDEX "idx_parent_comment" ON "comments" ("p_comment_id");
```

```sql
--- 5. Votes Table
CREATE TABLE "votes" (
 "id" SERIAL PRIMARY KEY NOT NULL,
 "value" SMALLINT NOT NULL,
 "user_id" INT,
 "post_id" INT,
 CONSTRAINT "value_can_be_1_or_minus1" CHECK(value in (1, -1)),
 CONSTRAINT "user_can_like_post_once" UNIQUE ("user_id", "post_id"),
 CONSTRAINT "fk_user" FOREIGN KEY("user_id") REFERENCES "users" ("id")
ON DELETE SET NULL,
 CONSTRAINT "fk_post" FOREIGN KEY("post_id") REFERENCES "posts" ("id")
ON DELETE CASCADE
);

--- Votes table's index on "value" column to calculate scores
CREATE INDEX "idx_vote_value" ON "votes" ("value");
```

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:

```
WITH "users_collection" AS (
   SELECT "username"
   FROM "bad_posts"
   UNION
   SELECT "username"
   FROM "bad_comments"
   UNION
   SELECT regexp_split_to_table("upvotes", ',') AS "username"
   FROM "bad_posts"
   UNION
   SELECT regexp_split_to_table("downvotes", ',') AS "username"
   FROM "bad_posts"
)
   INSERT INTO "users" ("username") SELECT DISTINCT "username" FROM
"users_collection";
```

```sql
INSERT INTO "topics" ("name", "user_id") (
   SELECT DISTINCT ON (b.topic) b.topic AS "name", u.id AS "user_id"
   FROM bad_posts b
   JOIN users u
   ON u.username = b.username
);



INSERT INTO "posts" ("title", "url", "text", "topic_id", "user_id") (
   SELECT b.title, b.url, b.text_content AS text, t.id AS topic_id, u.id
AS user_id
   FROM bad_posts b
   JOIN users u
   ON b.username = u.username AND LENGTH(TRIM(b.title)) <= 100
   JOIN topics t
   ON b.topic = t.name
);



INSERT INTO "comments" ("content", "user_id", "post_id") (
   SELECT b.text_content, u.id AS user_id, p.id AS post_id
   FROM bad_comments b
   JOIN users u
   ON u.username = b.username
   JOIN bad_posts bp
   ON b.post_id = bp.id
   JOIN posts p
   ON p.title = bp.title
);
```

```
INSERT INTO "votes" ("value", "user_id", "post_id")(
   WITH upvotes AS (
     SELECT DISTINCT ON(username) title,
regexp_split_to_table("upvotes", ',') AS username
     FROM bad_posts
   ),
   downvotes AS (
     SELECT DISTINCT ON (username) title,
regexp_split_to_table("downvotes", ',') AS username
     FROM bad_posts
   )(
     SELECT 1 AS value, u.id AS user_id, p.id AS post_id
     FROM upvotes uv
     JOIN users u
     ON u.username = uv.username
     JOIN posts p
     ON p.title = uv.title
   )
   UNION ALL
   (
     SELECT -1 AS value, u.id AS user_id, p.id AS post_id
     FROM downvotes dv
     JOIN users u
     ON u.username = dv.username
     JOIN posts p
     ON p.title = dv.title
   )
);
```

Part IV: Some sample queries for testing.

```sql
--- 1. List all users who haven't created any post.
SELECT username, p.title, p.text, p.url
FROM users u
LEFT JOIN posts p
ON p.user_id = u.id
WHERE p.user_id IS NULL;


--- 2. Find a user by their username.
SELECT *
FROM users
WHERE username = 'Rickie0';


--- 3. List all topics that don't have any posts.
SELECT name, COUNT(*)
FROM topics t
JOIN posts p
ON p.topic_id = t.id
GROUP BY 1
HAVING COUNT(*) < 1;


--- 4. Find a topic by its name.
SELECT *
FROM topics t
WHERE t.name = 'Car';
```

```sql
--- 5. List the latest 20 posts for a given topic.
SELECT *
FROM posts p
WHERE p.topic_id = (
 SELECT id
 FROM topics t
 WHERE t.name = 'Car'
)
LIMIT 20;


--- 6. List the latest 20 posts made by a given user.
SELECT *
FROM posts p
WHERE p.user_id = (
 SELECT id
 FROM users u
 WHERE u.username = 'Keagan_Howell'
)
ORDER BY p.created_at DESC
LIMIT 20;
```