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. There are multiple values of usernames in a row for the “bad_posts” (“upvotes”) and (“downvotes”) columns. This violates the First Normal Form for relational databases. The database should be amended to include only one value of username in each row for these columns.

2. The “bad_comments” (“post_id”) is NOT referenced to the “bad_posts”(“id”) column. A FOREIGN KEY reference should be made to ensure coherence in the post_id of the two tables.

3. The user details in both the “bad_posts” and “bad_comments” tables are given as complete usernames instead of a computer generated ID. This can lead to;
    a. Error in data entry
    b. Huge inconvenience when a username is changed.

    A separate table of users should be made where every user is assigned a unique ID. This ID can then be referenced in the posts and comments table.

4. Data type of “bad_comments” (“post_id”) is BIGINT. As this column should be referring to “bad_posts”(“id”), it should be the same data type i.e. INTEGER.

5. There is no separate table for Topics. They are included as a column of the “bad_posts” table. Having a separate table for Topics will help in including topics with no posts yet. The topics can then be referenced from their ID to the posts table.

6. There are no additional constraint checks on any column (other than Primary Keys) to ensure a check on the database level. Some checks on columns such as username, url etc. would help with having an additional level of checks on the data at the database level.

7. There is no indexing to speed up the searching of the database. Indexing can be added to different columns to meet search requirements for the business.

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:

```
-- creating tables DDL

CREATE TABLE "users" (

    "id" SERIAL PRIMARY KEY,

    "username" VARCHAR(25) UNIQUE,
```

```
    "last_login" TIMESTAMP
);

CREATE TABLE "topics" (
    "id" SERIAL PRIMARY KEY,
    "name" VARCHAR(30) UNIQUE,
    "description" VARCHAR(500),
    "user_id" INTEGER REFERENCES "users"
);

CREATE TABLE "posts" (
    "id" SERIAL PRIMARY KEY,
    "title" VARCHAR(100),
    "url" VARCHAR DEFAULT NULL,
    "text_content" TEXT DEFAULT NULL,
    "topic_id" INTEGER REFERENCES "topics" ON DELETE CASCADE,
    "user_id" INTEGER REFERENCES "users" ON DELETE SET NULL,
    "post_time" TIMESTAMP
    CONSTRAINT "content_check" CHECK (("url" IS NOT NULL AND "text_content" IS NULL)
OR ("url" IS NULL AND "text_content" IS NOT NULL))
);

CREATE TABLE "comments" (
    "id" SERIAL PRIMARY KEY,
    "user_id" INTEGER REFERENCES "users" ON DELETE SET NULL,
```

```
    "post_id" INTEGER REFERENCES "posts" ON DELETE CASCADE,
    "parent_comment" INTEGER REFERENCES "comments"("id") ON DELETE CASCADE,
    "content" TEXT
);

CREATE TABLE "user_votes" (
    "id" SERIAL PRIMARY KEY,
    "user_id" INTEGER REFERENCES "users" ON DELETE SET NULL,
    "post_id" INTEGER REFERENCES "posts" ON DELETE CASCADE,
    "vote" SMALLINT,
    CONSTRAINT "vote_only_once" UNIQUE ("user_id", "post_id")
);



-- adding constraints to the tables
ALTER TABLE "topics" ADD CONSTRAINT "topic_name_not_empty" CHECK
(LENGTH(TRIM("name"))>0),
    ALTER COLUMN "name" SET NOT NULL;
ALTER TABLE "topics" ALTER COLUMN "user_id" SET NOT NULL;
ALTER TABLE "topics" ALTER COLUMN "user_id" SET DEFAULT 1; -- doing this to allow
data entry in current format



ALTER TABLE "users" ADD CONSTRAINT "username_not_empty" CHECK
(LENGTH(TRIM("username"))>0),
     ALTER COLUMN "username" SET NOT NULL;
```

```
ALTER TABLE "posts" ADD CONSTRAINT "post_title_not_empty" CHECK
(LENGTH(TRIM("title"))>0),

    ALTER COLUMN "title" SET NOT NULL;

ALTER TABLE "posts" ALTER COLUMN "topic_id" SET NOT NULL;


ALTER TABLE "comments" ADD CONSTRAINT "comment_content_not_empty" CHECK
(LENGTH(TRIM("content"))>0),

    ALTER COLUMN "content" SET NOT NULL;

ALTER TABLE "comments" ALTER COLUMN "post_id" SET NOT NULL,

   ALTER COLUMN "parent_comment" SET NOT NULL;

ALTER TABLE "comments" ALTER COLUMN "parent_comment" SET DEFAULT 1; -- doing
this to allow data entry in current format


ALTER TABLE "user_votes" ADD CONSTRAINT "voting_system" CHECK ("vote" = 1 OR
"vote" = -1);

ALTER TABLE "user_votes" ALTER COLUMN "post_id" SET NOT NULL;


-- creating indexes

CREATE INDEX "search_post_topic_time" ON "posts" ("topic_id", "post_time");

CREATE INDEX "search_post_user_time" ON "posts" ("user_id", "post_time");

CREATE INDEX "search_child_comment" ON "comments" ("parent_comment");

CREATE INDEX "search_user_comment" ON "comments" ("user_id");

CREATE INDEX "search_url_posts" ON "posts" ("url");

CREATE INDEX "search_post_votes" ON "user_votes" ("post_id");
```

Examples of DQL

```
-- examples of DQL (Data Query Language)


-- List all users who haven’t logged in in the last year
SELECT *
FROM users
WHERE CURRENT_TIMESTAMP - last_login > INTERVAL '1 year';


-- List all users who haven’t created any post
SELECT u.id
FROM users u
LEFT JOIN posts p
ON u.id = p.user_id
WHERE p.user_id = NULL;


-- Find a user by their username.
SELECT u.username
FROM users u
WHERE u.username LIKE '%user%';


-- List all topics that don’t have any posts
SELECT t.name
FROM topics t
LEFT JOIN posts p
ON t.id = p.topic_id
WHERE p.topic_id = NULL;
```

```
-- Find a topic by its name
SELECT *
FROM topics t
WHERE t.name LIKE '%topic%';

-- List the latest 20 posts for a given topic
SELECT p.id, p.title, t.name, p.post_time
FROM posts p
JOIN topics t
ON p.topic_id = t.id
WHERE t.name LIKE '%e' -- example
ORDER BY p.post_time DESC
LIMIT 20;

-- List the latest 20 posts made by a given user.
SELECT p.id, p.title, u.username, p.post_time
FROM posts p
JOIN users u
ON p.user_id = u.id
WHERE u.username LIKE '%e' -- example
ORDER BY p.post_time DESC
LIMIT 20;

-- Find all posts that link to a specific URL, for moderation purposes.
SELECT *
FROM posts
```

```
WHERE url LIKE '%www.specific_URL%';


-- List all the top-level comments (those that don’t have a parent comment) for a given post.

SELECT *

FROM comments

WHERE parent_comment IS NULL;


-- List all the direct children of a parent comment.

SELECT *

FROM comments

WHERE parent_comment = 2 -- example comment_id;


-- List the latest 20 comments made by a given user.

SELECT u.username, c.post_id, c.parent_comment, c.content

FROM comments c

JOIN users u

ON c.user_id = u.id

WHERE u.username = '%example%';


-- Compute the score of a post, defined as the difference between the number of upvotes and the number of downvotes

SELECT post_id, SUM(vote) tot_score

FROM user_votes

GROUP BY 1

ORDER BY 2 DESC;
```

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:

```
-- adding data into the tables DML


-- adding ("usernames") in "users" table
INSERT INTO "users" ("username")
    SELECT regexp_split_to_table(upvotes,',') UID
    FROM bad_posts
    UNION
    SELECT regexp_split_to_table(upvotes,',') UID
    FROM bad_posts
```

```
    UNION
    SELECT username UID
    FROM bad_posts
    UNION
    SELECT username UID
    FROM bad_comments;

-- adding ("name") to "topics" table (there are no descriptions available)
INSERT INTO "topics" ("name")
    SELECT DISTINCT(topic)
    FROM bad_posts;

-- adding data into "posts"
INSERT INTO "posts" ("id", "title", "url", "text_content", "topic_id", "user_id")
    SELECT bp.id, bp.title::VARCHAR(100), bp.url, bp.text_content, t.id topic_id, u.id UID
    FROM topics t
    JOIN bad_posts bp
    ON t.name = bp.topic
    JOIN users u
    ON bp.username = u.username;

-- adding data into "comments" table
INSERT INTO "comments" ("id", "user_id", "post_id", "content")
    SELECT bc.id, u.id UID, bc.post_id, bc.text_content
    FROM users u
```

```
    JOIN bad_comments bc
    ON u.username = bc.username
    JOIN posts p
    ON bc.post_id = p.id;



-- adding data into "user_votes" table (by creating 2 dummy tables)


--dummy upvotes
CREATE TABLE "dummy_upvote" (
    "user_id" INTEGER,
    "post_id" INTEGER,
    "vote" INTEGER
);


INSERT INTO "dummy_upvote" ("user_id", "post_id")
    WITH t1 AS (
        SELECT bp.id post_id, regexp_split_to_table(bp.upvotes,',') uname_upvote
        FROM bad_posts bp
        )
    SELECT u.id UID, t1.post_id
    FROM t1
    JOIN users u
    ON t1.uname_upvote = u.username;
```

```
UPDATE "dummy_upvote" SET "vote" = 1;


--dummy downvotes
CREATE TABLE "dummy_downvote" (
    "user_id" INTEGER,
    "post_id" INTEGER,
    "vote" INTEGER
);


INSERT INTO "dummy_downvote" ("user_id", "post_id")
    WITH t1 AS (
        SELECT bp.id post_id, regexp_split_to_table(bp.downvotes,',') uname_downvote
        FROM bad_posts bp
        )
    SELECT u.id UID, t1.post_id
    FROM t1
    JOIN users u
    ON t1.uname_downvote = u.username;


UPDATE "dummy_downvote" SET "vote" = -1;


-- adding the dummy data into actual table of user_votes
INSERT INTO "user_votes" ("user_id", "post_id", "vote")
    SELECT *
    FROM dummy_upvote
```

```
    UNION ALL
    SELECT *
    FROM dummy_downvote;

-- deleting the dummy tables from the system
DROP TABLE "dummy_upvote";
DROP TABLE "dummy_downvote";
```