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!

```
-- EXPLORE DATA:
-- show all tables
\dt

-- show datatypes of a single table
\d bad_posts
\d bad_comments
```

We should (1) split the current single table into multiple tables in order to normalize data (2) add the constraints in order to prevent the user from inputting wrong type of data (3) add indexing for quick search.

Suggestions on (1) data normalization and (2) constraints:
Store users in a separate table to avoid partial dependencies (username on user id).
Create table "users" which should contain
id SERIAL PRIMARY KEY
username VARCHAR(50) NOT NULL

In table posts keep only data directly related to a single post.
Table "posts" should contain
"id" SERIAL PRIMARY KEY
"topic_id" SMALLINT,
FOREIGN KEY ("topic_id") REFERENCES "topics" ("id")
"user_id" INTEGER,
FOREIGN KEY ("user_id") REFERENCES "users" ("id")
"title" VARCHAR(150),
"url" VARCHAR(4000) DEFAULT NULL
"text_content" TEXT DEFAULT NULL

In case that name of the topic changes or we add a new topic, store topics in a separate table
Create table "topics" which should contain
"id" SERIAL PRIMARY KEY
"topic" VARCHAR(50)

Create two tables for up and downvotes.
Create table "upvotes" which should contain
"post_id" INTEGER
FOREIGN KEY ("post_id") REFERENCES "posts" ("id")
"user_id" INTEGER

```
FOREIGN KEY ("user_id") REFERENCES "users" ("id")
PRIMARY KEY("post_id", "user_id")

Create table "downvotes" which should contain
"post_id" INTEGER
FOREIGN KEY ("post_id") REFERENCES "posts" ("id")
"user_id" INTEGER
FOREIGN KEY ("user_id") REFERENCES "users" ("id")
PRIMARY KEY("post_id", "user_id")

Table comments should contain references (foreign keys) to other tables
Table "comments" should include
"id" SERIAL PRIMARY KEY
"user_id" INTEGER,
FOREIGN KEY ("user_id") REFERENCES "users" ("id")
"post_id" INTEGER,
FOREIGN KEY ("post_id") REFERENCES "posts" ("id")
Text_content TEXT(4000)
```

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

```
CREATE TABLE "users"(
      "id" SERIAL PRIMARY KEY,
      "username" VARCHAR(25) NOT NULL,
      CHECK(length(regexp_replace("username", '\s', '', 'g')) > 0),
      -- ensures that the username is not only space character
      -- := length(trim("username") > 0)
      "last_login" TIMESTAMP
      -- Guideline #2: a) database needs to be able to list all users who
haven't logged in in the last year
);

CREATE UNIQUE INDEX "unique_username" ON "users"(LOWER(TRIM("username")));
-- lower assures zero case sensitivity (prevents us from creating both Lara
and lara), trim assures zero sensitivity to spacing before and at the end '
lara   ' is same as 'lara'
CREATE INDEX "by_username" ON "users" (LOWER("username")
VARCHAR_PATTERN_OPS);
-- Guideline #2: c) Find a user by their 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,
	CHECK(length(regexp_replace("topic_name", '\s', '', 'g')) > 0),
	"description" VARCHAR(500) DEFAULT NULL,
	"user_id" INTEGER,
	-- cannot use constraint NOT NULL as the user_id of topic is not
known
	FOREIGN KEY ("user_id") REFERENCES "users"("id")
);

CREATE UNIQUE INDEX "unique_topic" ON "topics"(LOWER(TRIM("topic_name")));
CREATE INDEX ON "topics"(LOWER("topic_name") VARCHAR_PATTERN_OPS); --
Guideline #2:: e) Find a topic by its name.
-- can be written together as CREATE UNIQUE INDEX "unique_topic" ON
"topics"(LOWER(TRIM("name")) VARCHAR_PATTERN_OPS);

CREATE INDEX "by_user_id" ON "topics"("user_id"); -- Guideline #2: b) List
all users who haven't created any post. WHERE id NOT IN user_id
```

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(500) DEFAULT NULL,
	"text" VARCHAR(4000) DEFAULT NULL,
	-- either url or text should exist but not booth
	CHECK((nullif("url",'') is null or nullif("text",'') is null) and not
(nullif("url",'') is null and nullif("text",'') is null)),
	"last_login" TIMESTAMP,
	-- Guideline #2: f) List the latest 20 posts for a given topic.
	"topic_id" INTEGER NOT NULL, -- if the topic_id does not exist,
cannot enter the new post
	-- best practice is to constraint all foreign keys to NOT NULL (do
not set as NOT NULL if ON DELETE SET NULL)
	FOREIGN KEY ("topic_id") REFERENCES "topics"("id") ON DELETE CASCADE,
```

```
-- if the topics id gets deleted, delete all columns related to that id
      "user_id" INTEGER,
      FOREIGN KEY ("user_id") REFERENCES "users"("id") ON DELETE SET NULL
);

CREATE INDEX "by_topic_id" ON "posts"("topic_id"); -- Guideline #2: d) List
all topics that don't have any posts. WHERE id NOT IN topic_id & f) List
the latest 20 posts for a given topic. WHERE topic_id = n
CREATE INDEX "by_user_id" ON "posts"("user_id"); -- Guideline #2: g) List
the latest 20 posts made by a given user.
CREATE INDEX "by_url" ON "posts"("url"); -- Guideline #2: h) Find all posts
that link to a specific URL, for moderation purposes. WHERE usl = 'str'
```

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,
      "comment" VARCHAR(5000) NOT NULL,
      CHECK(length(regexp_replace("comment", '\s', '', 'g')) > 0),
      "parent_id" INTEGER DEFAULT NULL, -- allows to store hierarchy,
comment1 has no parent, comment2 commenting on comment1 has parent id=1
      FOREIGN KEY ("parent_id") REFERENCES "comments"("id") ON DELETE
CASCADE, -- delete all children if the parent is deleted
      "time" TIMESTAMP,
      "user_id" INTEGER,
      FOREIGN KEY ("user_id") REFERENCES "users"("id") ON DELETE SET NULL,
      "post_id" INTEGER NOT NULL,
      FOREIGN KEY ("post_id") REFERENCES "posts"("id") ON DELETE CASCADE
);

CREATE INDEX "by_parent" ON "comments"("parent_id"); -- Guideline #2: i)
List all the top-level comments (those that don't have a parent comment)
for a given post.
```

```
--WHERE "parent_id" IS NULL & Guideline #2: j) List all the direct children
of a parent comment. WHERE "parent_id" = n
CREATE INDEX "lastest_comments_of_user_n" ON "comments"("user_id", "time");
-- Guideline #2: k) List the latest 20 comments made by a given user. WHERE
user_id = n -> ORDER BY TIMESTAMP, LIMIT 20
--CREATE INDEX "lastest_comments_of_user_n" ON "comments"("user_id",
"time") could be used as well, but we prefer to use only part of the WHERE
clause
```

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.

```
-- how to store likes and dislikes in a single table?
https://dba.stackexchange.com/questions/51527/model-likes-dislikes-in-sql:
--The column Vote should be constrained to only allow the values Like and
Dislike or, if you prefer, 1 and -1 -- the latter having the advantage of a
simpler --possibility to do the sum() of the likes and dislikes to get an
overall vote.

CREATE TABLE "votes"(
      "id" SERIAL PRIMARY KEY,
      -- PRIMARY KEY ("user_id", "post_id") cannot be added because of
user_id & post_id ON DELETE SET NULL. PRIMARY KEY is UNIQUE and always NOT
NULL
      "vote" SMALLINT NOT NULL,
      CHECK("vote" = 1 OR "vote" = -1),
      "user_id" INTEGER,
      FOREIGN KEY ("user_id") REFERENCES "users"("id") ON DELETE SET NULL,
      "post_id" INTEGER,
      FOREIGN KEY ("post_id") REFERENCES "posts"("id") ON DELETE SET NULL,
      UNIQUE("user_id", "post_id") --A user can only cast one vote on a
given post
);

CREATE INDEX "post_score" ON "votes"("post_id");
-- Guideline #2: l) Compute the score of a post, defined as the difference
between the number of upvotes and the number of downvotes
```

```
-- SELECT SUM(votes) WHERE post_id = n;
```

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.

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:

```
-- explore data in the tables bad_comments and bad_posts

SELECT * FROM bad_posts LIMIT 5;
SELECT * FROM bad_comments LIMIT 5;

-- migrate bad_posts(username), bad_comments(username)[--some users only
comment], bad_posts(upvotes,downvotes) [-- some users only vote] into
users(username)

INSERT INTO users(username)
SELECT DISTINCT regexp_split_to_table(upvotes, ',') as username
FROM bad_posts
UNION
SELECT DISTINCT regexp_split_to_table(downvotes, ',') as username
FROM bad_posts
UNION
SELECT DISTINCT username
FROM bad_posts
UNION
```

```sql
SELECT DISTINCT username
FROM bad_comments
ORDER BY username;

-- verify the existence of duplicate bad_posts(topic)
SELECT topic, COUNT(*)
FROM bad_posts
GROUP BY topic
HAVING COUNT(*) > 1;

-- migrate bad_posts(topic) into topics(topic_name)

INSERT INTO topics(topic_name)
SELECT DISTINCT topic
FROM bad_posts
ORDER BY topic;


-- prepare bad_posts(title,url,text_contet) for migrating into
posts(title,url,text)
-- remove rows where title IS NULL, remove rows where both the url and text
ARE NULL
CREATE VIEW clean_bad_posts
AS
SELECT id, topic, username, url,
CASE WHEN url IS NOT NULL THEN NULL
      ELSE text_content END AS text_content2,
 -- keep the url, if the url does not exist, keep the text
CASE WHEN LENGTH(title) > 100 THEN SUBSTRING(title, 1, 100)
      ELSE title END AS title_short
-- if the title is longer than 100 characters, cut it to 100 characters
-- alternative way to write it: SELECT SUBSTRING(title, 1, 100)
FROM bad_posts
WHERE title IS NOT NULL
-- if the title does not exist, drop the post
AND NOT (url IS NULL AND text_content IS  NULL)
-- if both url and text_content are null, drop the post
;

-- retrieve the users(id) that belongs to bad_posts(username) and
topics(id) that belongs to bad_posts(topic)
CREATE VIEW clean_bad_posts_with_ids
AS
```

```sql
SELECT a2.user_id, a2.username, t.id as topic_id, a2.topic, a2.title_short,
a2.url, a2.text_content2, a2.id
FROM
(SELECT u.id as user_id, c.*
FROM users u
JOIN clean_bad_posts c
ON u.username = c.username) a2
JOIN topics t
ON t.topic_name = a2.topic
;

-- migrate clean_bad_posts(title,url,text_content,topic_id,user_id) into
posts(title,url,text,topic_id,user_id)
INSERT INTO posts(id,title,url,"text",topic_id,user_id)
SELECT id, title_short, url, text_content2, topic_id, user_id
FROM clean_bad_posts_with_ids
;

-- check if the migration was successful
SELECT id, topic_id  FROM clean_bad_posts_with_ids LIMIT 5;
SELECT id, topic_id FROM posts LIMIT 5;
      -- should result in the same output


-- migrate bad_comments(username,post_id, text_content) into
comments(user_id, post_id, comment)

INSERT INTO comments(id, user_id, post_id, comment)
SELECT b.id as comment_id, u.id as user_id, b.post_id, b.text_content
FROM bad_comments b
JOIN users u
ON b.username = u.username
-- include the user_id instead of username
;

-- migrate bad_posts(upvotes, downvotes) to votes(vote)

SELECT upvotes FROM bad_posts LIMIT 1; -- first topic was upvoted by this
usernames:
--
Judah.Okuneva94,Dasia98,Maurice_Dooley14,Dangelo_Lynch59,Brandi.Schaefer,Ja
yde.Kulas74,Katarina_Hudson,Ken.Murphy42
```

```sql
-- migrate upvotes as +1

INSERT INTO votes(user_id, post_id, vote)
SELECT DISTINCT u.id as user_id, post_id, 1 AS vote
-- single user_id can vote only one time on each topic_id
FROM
(SELECT id as post_id, regexp_split_to_table(upvotes, ',') as users_upvotes
 -- write each username in its row
FROM bad_posts
) t2
JOIN users u
ON u.username = t2.users_upvotes
;

-- migrate downvotes as -1

INSERT INTO votes(user_id, post_id, vote)
SELECT DISTINCT u.id as user_id, post_id, -1 AS vote
-- single user_id can vote only one time on each topic_id
FROM
(SELECT id as post_id, regexp_split_to_table(downvotes, ',') as
users_downvotes
 -- write each username in its row
FROM bad_posts
) t2
JOIN users u
ON u.username = t2.users_downvotes
;
```