

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!

The database is not well normalized...

bad_posts table:

- Both columns **upvotes** and **downvotes** violates the 1st Normal Form on storing a list of comma separated values on a single column.
- There is a violation of the 2nd Normal Form also there is a transitive dependencies between title and post topic.
- The **bad_posts** must be splitted, and new tables must be created, **Users, Posts, Topics, Votes, Comments, Post_Comments;**
- Username must be replaced with user_id from bad_comments.

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 Uddidit. 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 Uddidit 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:

```
BEGIN;  
-- DDL  
-- This table <PROFILES> stores the users data as in PostgreSQL  
-- user is a reserved word I rather use profile  
CREATE TABLE "profiles" (  
  "id" SERIAL,
```

```
"username" VARCHAR(25) NOT NULL,  
CONSTRAINT "profile_pk" PRIMARY KEY ("id"),  
CONSTRAINT "unique_user_name" UNIQUE("username"),  
CONSTRAINT "not_empty_username" CHECK(LENGTH("username") > 0)  
);
```

```
-- For each sessions on the application the user information will be holded  
-- Into this table. This way we'll be able to query the DB to  
-- List all users who haven't logged in the last year.
```

```
CREATE TABLE "profile_sessions" (  
"id" SERIAL,  
"profile_id" INTEGER,  
"session_date_time" TIMESTAMP,  
CONSTRAINT "profile_sessions_pk" PRIMARY KEY ("id"),  
CONSTRAINT "profiles_fk" FOREIGN KEY ("profile_id") REFERENCES "profiles"("id") ON  
DELETE  
SET NULL  
);
```

```
-- This Index will prevent from taking long time queryng for a specific user  
CREATE INDEX IF NOT EXISTS "user_name_idx" ON "profiles" ("username"  
VARCHAR_PATTERN_OPS);
```

```
-- Each topic must be on a separated table so when  
-- a new post is created topic ID can be used as Foreign key
```

```
CREATE TABLE "topics" (  
"id" SERIAL,  
"name" VARCHAR(30) NOT NULL,  
"description" VARCHAR(500),  
"profile_id" INTEGER,  
CONSTRAINT "unique_topic_name" UNIQUE("name"),  
CONSTRAINT "topic_pk" PRIMARY KEY ("id"),  
CONSTRAINT "profiles_fk" FOREIGN KEY ("profile_id") REFERENCES "profiles"("id"),  
CONSTRAINT "not_empty_topic_name" CHECK(LENGTH("name") > 0)  
);
```

```
-- In case topic table gets heavier this index will make it quick  
-- to query for each topic by it's name
```

```
CREATE INDEX IF NOT EXISTS "topic_name_idx" ON "topics" ("name"  
VARCHAR_PATTERN_OPS);
```

```
-- This table stores all posts by storing it's title and url also the date in which  
-- the post was created along with the user who created it and the topic it belongs to.  
CREATE TABLE "posts" (  

```

```
"id" SERIAL,  
"title" VARCHAR(100) NOT NULL,  
"url" TEXT,  
"created_at" TIMESTAMP,  
"topic_id" INTEGER,  
"profile_id" INTEGER,  
CONSTRAINT "posts_pk" PRIMARY KEY ("id"),  
CONSTRAINT "topics_fk" FOREIGN KEY ("topic_id") REFERENCES "topics"("id") ON  
DELETE CASCADE,  
CONSTRAINT "profiles_fk" FOREIGN KEY ("profile_id") REFERENCES "profiles"("id") ON  
DELETE  
SET NULL,  
CONSTRAINT "not_empty_post_title" CHECK(LENGTH("title") > 0)  
);
```

```
-- This index will make it quick to search each topic by its title
```

```
-- It will also allow for pattern search
```

```
CREATE INDEX IF NOT EXISTS "post_title_idx" ON "posts" ("title"  
VARCHAR_PATTERN_OPS);
```

```
-- This table stores the user votes for each post
```

```
CREATE TABLE "post_likes" (  
"id" SERIAL,  
"profile_id" INTEGER NOT NULL,  
"post_id" INTEGER NOT NULL,  
"vote" SMALLINT,  
CONSTRAINT "post_likes_pk" PRIMARY KEY ("id"),  
CONSTRAINT "profiles_fk" FOREIGN KEY ("profile_id") REFERENCES "profiles"("id") ON  
DELETE  
SET NULL,  
CONSTRAINT "posts_fk" FOREIGN KEY ("post_id") REFERENCES "posts"("id") ON  
DELETE CASCADE,  
CONSTRAINT "check_vote" CHECK(  
"vote" = 1  
OR "vote" = -1  
),  
CONSTRAINT "unique_vote" UNIQUE("profile_id", "post_id")  
);
```

```
-- This table stores all comments for each post and the comment owner.
```

```
CREATE TABLE "comments" (  
"id" SERIAL,  
"text_comment" TEXT NOT NULL,  
"created_at" TIMESTAMP,
```

```

"profile_id" INTEGER,
"post_id" INTEGER,
CONSTRAINT "comments_pk" PRIMARY KEY ("id"),
CONSTRAINT "profiles_fk" FOREIGN KEY ("profile_id") REFERENCES "profiles"("id") ON
DELETE
SET NULL,
CONSTRAINT "posts_fk" FOREIGN KEY ("post_id") REFERENCES "posts"("id") ON
DELETE CASCADE,
CONSTRAINT "not_empty_comment_text" CHECK(LENGTH("text_comment") > 0)
);

```

-- This table stores the user votes for each comment in a specific post

```

CREATE TABLE "comment_likes" (
"id" SERIAL,
"profile_id" INTEGER,
"comment_id" INTEGER,
"vote" SMALLINT,
CONSTRAINT "comments_likes_pk" PRIMARY KEY ("id"),
CONSTRAINT "profiles_fk" FOREIGN KEY ("profile_id") REFERENCES "profiles"("id") ON
DELETE
SET NULL,
CONSTRAINT "comments_fk" FOREIGN KEY ("comment_id") REFERENCES
"comments"("id") ON DELETE CASCADE,
CONSTRAINT "check_vote" CHECK(
"vote" = 1
OR "vote" = -1
),
CONSTRAINT "comment_unique_vote" UNIQUE("profile_id", "comment_id")
);

```

-- This table will store comment threads for each comment

```

CREATE TABLE "comment_threads" (
"id" SERIAL,
"text_comment" TEXT NOT NULL,
"created_at" TIMESTAMP,
"profile_id" INTEGER,
"comment_id" INTEGER,
CONSTRAINT "comments_threads_pk" PRIMARY KEY ("id"),
CONSTRAINT "profiles_fk" FOREIGN KEY ("profile_id") REFERENCES "profiles"("id") ON
DELETE
SET NULL,
CONSTRAINT "comments_fk" FOREIGN KEY ("comment_id") REFERENCES
"comments"("id") ON DELETE CASCADE,
CONSTRAINT "not_empty_comment_thread_text" CHECK(LENGTH("text_comment") >

```

```
0)
```

```
);
```

```
-- This table will store votes for each comment thread
```

```
CREATE TABLE "comment_thread_likes" (
```

```
"id" SERIAL,
```

```
"profile_id" INTEGER,
```

```
"comment_thread_id" INTEGER,
```

```
"vote" SMALLINT,
```

```
CONSTRAINT "comments_thread_pk" PRIMARY KEY ("id"),
```

```
CONSTRAINT "profiles_fk" FOREIGN KEY ("profile_id") REFERENCES "profiles"("id") ON  
DELETE
```

```
SET NULL,
```

```
CONSTRAINT "comment_thread_fk" FOREIGN KEY ("comment_thread_id")
```

```
REFERENCES "comment_threads"("id") ON DELETE CASCADE,
```

```
CONSTRAINT "check_vote" CHECK(
```

```
"vote" = 1
```

```
OR "vote" = -1
```

```
),
```

```
CONSTRAINT "comment_thread_unique_vote" UNIQUE("profile_id",
```

```
"comment_thread_id")
```

```
);
```

```
COMMIT;
```


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:

```
BEGIN;
-- DML
-- INSERT INTO PROFILES
INSERT INTO "profiles" (username)
SELECT DISTINCT "username"
FROM "bad_posts"
WHERE "username" IS NOT NULL
AND "username" != "";

-----
-- To make sure also the user who voted, commented and didn't post
-- any article also get registered, the next three insert queries will retrieve data from
-- both upvotes and downvotes columns using REGEXP_SPLIT_TO_TABLE function
-- to convert the comma separated usernames list into a table and then check some
-- constraints and insert data into profiles table. The same applies for bad_comments
-- table usernames.
```

```

INSERT INTO "profiles" (username)
SELECT u.usernames
FROM
(SELECT DISTINCT REGEXP_SPLIT_TO_TABLE(upvotes,
',') AS usernames
FROM "bad_posts") u
WHERE u.usernames IS NOT NULL
AND u.usernames != ""
AND u.usernames NOT IN
(SELECT DISTINCT username
FROM "profiles");

-----

INSERT INTO "profiles" (username)
SELECT u.usernames
FROM
(SELECT DISTINCT REGEXP_SPLIT_TO_TABLE(downvotes,
',') AS usernames
FROM "bad_posts") u
WHERE u.usernames IS NOT NULL
AND u.usernames != ""
AND u.usernames NOT IN
(SELECT DISTINCT username
FROM "profiles");

-----

INSERT INTO "profiles" (username)
SELECT DISTINCT username
FROM "bad_comments" bc
WHERE bc.username IS NOT NULL
AND bc.username != ""
AND bc.username NOT IN
(SELECT DISTINCT username
FROM "profiles");

-----

-- INSERT INTO TOPICS
INSERT INTO "topics" (name)
SELECT DISTINCT topic
FROM "bad_posts"
WHERE "topic" IS NOT NULL
AND "topic" != "";

-----

-- INSERT INTO POSTS
INSERT INTO "posts" ( title,

```

```

url,
created_at,
topic_id,
profile_id ) SELECT
CASE
WHEN LENGTH(bp.title) > 100 THEN
CONCAT(LEFT(bp.title, 95), '...')
ELSE bp.title END,
bp.url, CURRENT_TIMESTAMP, t.id, p.id
FROM "bad_posts" bp
INNER JOIN "profiles" p
ON p.username = bp.username
INNER JOIN "topics" t
ON t.name = bp.topic
WHERE title IS NOT NULL
AND title != '';
-----
-- INSERT INTO POST LIKES UPVOTES
WITH T1 AS
(SELECT username,
title,
REGEXP_SPLIT_TO_TABLE(upvotes,
',') AS upvotes
FROM "bad_posts"), T2 AS (
SELECT pf.id, pf.username
FROM "profiles" pf
), T3 AS (
SELECT ps.id, ps.title AS title
FROM "posts" ps
)
INSERT INTO "post_likes" (profile_id, post_id, vote)
SELECT T2.id, T3.id, 1
FROM T1
INNER JOIN T2
ON T2.username=T1.upvotes
INNER JOIN T3
ON T3.title=T1.title
ORDER BY T1.username ASC;
-----
-- INSERT INTO POST LIKES DOWNVOTES
WITH T1 AS
(SELECT username,
title,
REGEXP_SPLIT_TO_TABLE(downvotes,

```

```

',') AS downvotes
FROM "bad_posts"), T2 AS (
SELECT pf.id, pf.username
FROM "profiles" pf
), T3 AS (
SELECT ps.id, ps.title AS title
FROM "posts" ps
)
INSERT INTO "post_likes" (profile_id, post_id, vote)
SELECT T2.id, T3.id, -1
FROM T1
INNER JOIN T2
ON T2.username=T1.downvotes
INNER JOIN T3
ON T3.title=T1.title
ORDER BY T1.username ASC;
-----
-- INSERT INTO COMMENTS
INSERT INTO "comments" (text_comment, created_at, profile_id, post_id)
SELECT bc.text_content,
NOW(),
pf.id,
ps.id
FROM "bad_comments" bc
INNER JOIN "profiles" pf
ON bc.username=pf.username
INNER JOIN "posts" ps
ON bc.post_id=ps.id
WHERE bc.text_content IS NOT NULL
AND bc.text_content != ""
AND bc.post_id IS NOT NULL;
COMMIT;

```