

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 variable username appears in both tables bad_posts and bad_comments - create a new table users with a dedicated user_id as primary key
- Indexes are missing from the existing tables - create these for some of the tables we want to create (e.g. index on the topicname in the topics table), to help with subsequent queries
- The current schema is not set up for avoiding duplicate entries. Putting the data in normal form would make it easier to write a schema that used the unique function to filter duplicates.
- Upvotes and downvotes can be easier encoded, e.g. by using numerical values such as 1 for upvotes and -1 for downvotes.
- Within bad_posts, the variables username and title can be currently NULL at the same time. One of the two should be not NULL at any given time, to allow registered users to post and allow searching by title.

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 Uddidit 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:

```
-- create user table based on guidelines and review #2 feedback

CREATE TABLE "user" (
  "id" SERIAL PRIMARY KEY,
  "username" TEXT NOT NULL CHECK (LENGTH (TRIM ("username"))) > 0),
  CONSTRAINT "username_length" CHECK (LENGTH("username") <= 25),
  CONSTRAINT "username_unique" UNIQUE ("username"),
  "last_login" TIMESTAMP
);
```

```

-- create topic table based on guidelines and review #2 feedback

CREATE TABLE "topic" (
  "id" SERIAL PRIMARY KEY,
  "topicname" TEXT UNIQUE NOT NULL CHECK (LENGTH (TRIM ("topicname"))) > 0),
  CONSTRAINT "topicname_length" CHECK (LENGTH("topicname") <= 30),
  "topicdesc" TEXT,
  CONSTRAINT "topicdesc_length" CHECK (LENGTH("topicdesc") <= 500),
  "topicdate" TIMESTAMP
);

CREATE INDEX "finding_latest_topic" ON "topic" ("topicdate");

-- create post table based on guidelines and review #2 feedback

CREATE TABLE "post" (
  "id" SERIAL PRIMARY KEY,
  "posttitle" TEXT NOT NULL CHECK (LENGTH (TRIM ("posttitle"))) > 0),
  CONSTRAINT "posttitle_length" CHECK (LENGTH("posttitle") <= 100),
  "url" TEXT,
  CONSTRAINT "url_length" CHECK (LENGTH("url") <= 2000),
  "text_content" TEXT,
  "user_id" INT REFERENCES "user" ("id") ON DELETE SET NULL,
  "topic_id" INT NOT NULL REFERENCES "topic" ("id") ON DELETE CASCADE,
  "last_post" TIMESTAMP
);

ALTER TABLE "post"
ADD CONSTRAINT "urlcontent" CHECK(
  ("url" IS NULL AND "text_content" IS NOT NULL)
  OR
  ("url" IS NOT NULL AND "text_content" IS NULL)
);

-- create comment table based on guidelines and review #2 feedback

CREATE TABLE "comment" (
  "id" SERIAL PRIMARY KEY,
  "text" TEXT NOT NULL CHECK (LENGTH (TRIM ( "text" )) > 0),
  "parent_comment_id" INT REFERENCES "comment" ("id") ON DELETE CASCADE,
  "user_id" INT REFERENCES "user" ("id") ON DELETE SET NULL,
  "post_id" INT NOT NULL REFERENCES "post" ("id") ON DELETE CASCADE,
  "date_commented" TIMESTAMP
);

-- create vote table based on guidelines and review #2 feedback

```

```

CREATE TABLE "vote" (
  "id" SERIAL PRIMARY KEY,
  "user_id" INT REFERENCES "user" ("id") ON DELETE SET NULL,
  "post_id" INT REFERENCES "post" ("id") ON DELETE CASCADE,
  "vote" SMALLINT CHECK ("vote" = 1 OR "vote" = -1),
  CONSTRAINT "no_repeat_votes" UNIQUE ("user_id","post_id")
);

-- create indices for new schema
CREATE INDEX "finding_user_by_username" ON "user" ("username");
CREATE INDEX "finding_user_last_login_date" ON "user" ("username", "last_login");
CREATE INDEX "finding_latest_posts_for_topic" ON "post" ("url", "text_content",
"last_post", "topic_id");
CREATE INDEX "finding_latest_posts_for_user" ON "post" ("url", "text_content",
"last_post", "user_id");
CREATE INDEX "finding_all_posts_with_url" ON "post" ("url");
CREATE INDEX "finding_latest_comments_for_user" ON "comment" ("date_commented",
"user_id");
CREATE INDEX "finding_post_score" ON "vote" ("post_id", "vote");

```

Notes on edit of DDL-tables, based on Udacity review #2 (review text copied in for reference):

1. You don't have to create DQL queries however at least your new schema fulfills the basic requirements of **Guideline #2**. Here are few guidelines that you make sure to fulfill for the newly created schema. Without creating them it's not possible to create a standard set of DQL queries.
 - a. List all users who haven't logged in in the last year
 - f. List the latest 20 posts for a given topic.
 - g. List the latest 20 posts made by a given user.
 - k. List the latest 20 comments made by a given user.

Code solutions:

- a. For 2a., added "last_login" timestamp to "user" table: `"last_login" TIMESTAMP`
 - b. For 2f. and 2g., added "last_post" to "post" table: `"last_post" TIMESTAMP`
 - c. For 2k., added "date_commented" to "comment" table: `"date_commented" TIMESTAMP`
2. Most commonly FOREIGN KEYS not allowed NULL values. However, when we use the ON DELETE SET NULL constraint then we can add the NULL values for the column. For which column we didn't use the ON DELETE SET NULL constraint make sure that

column uses the NOT NULL constraint to ensure that it should not any **non-existent IDs** to be inserted in related tables.

For example:

- Posts table has two FOREIGN KEY references **one** is for the user_id another one is for the topic_id.
- user_id column uses the ON DELETE SET NULL constraint. So we cannot use the NOT NULL constraint for the user_id column. In simple words, we can say that ON DELETE SET NULL and NOT NULL cannot work at the same time.
- However, the topic_id column uses the ON DELETE CASCADE constraint. For this, we can ensure that it does not allow the NULL to be inserted for the column to do this we will use the NOT NULL constraint for the column.

Code solution:

- added NOT NULL constraint to
 - Table post, topic_id: `"topic_id" INT NOT NULL REFERENCES "topic" ("id") ON DELETE CASCADE`
 - Table comment, post_id: `"post_id" INT NOT NULL REFERENCES "post" ("id") ON DELETE CASCADE`

3. Please create an **index** for your newly **created schema** to speed up the query execution. If you don't know how to create an **index** you can revisit the lesson [Performance with Indexes](#).

Code solution: multiple indices were created at the end of the code above:

```
-- create indices for new schema
CREATE INDEX "finding_user_by_username" ON "user" ("username");
CREATE INDEX "finding_user_last_login_date" ON "user" ("username", "last_login");
CREATE INDEX "finding_latest_posts_for_topic" ON "post" ("url", "text_content", "last_post", "topic_id");
CREATE INDEX "finding_latest_posts_for_user" ON "post" ("url", "text_content", "last_post", "user_id");
CREATE INDEX "finding_all_posts_with_url" ON "post" ("url");
CREATE INDEX "finding_latest_comments_for_user" ON "comment" ("date_commented", "user_id");
CREATE INDEX "finding_post_score" ON "vote" ("post_id", "vote");
```

4. There are a few small improvements required for the users table as per the requirements:
 - Usernames can't be empty. To achieve this you will be going to use the TRIM() function to remove the while space. Also, check the length of the characters

using the CHECK constraint and LENGTH function. You can take a look at the below screenshot I'm able to add an **empty username** in the username column.

```
postgres=# INSERT INTO "user"(username) VALUES (' ');
INSERT 0 1
postgres=#
```

Code solution: added trimming and length check to username

```
"username" TEXT NOT NULL CHECK (LENGTH (TRIM ("username"))) > 0),
```

5. There are a few small improvements required for the topics table as per the requirements
 - The topic's name can't be empty. To achieve this you will be going to use the TRIM() function to remove the while space. Also, check the length of the characters using the CHECK constraint and LENGTH function. You can take a look at the below screenshot I'm able to add an **empty topic** in the topic column.

```
postgres=#
postgres=# INSERT INTO "topic"(topicname) VALUES (' ');
INSERT 0 1
postgres=#
```

Code solution: added trimming and length check to topicname

```
"topicname" TEXT UNIQUE NOT NULL CHECK (LENGTH (TRIM ("topicname"))) > 0),
```

- Topic names have to be unique. You have to use the UNIQUE constraint to ensure the topic name are inserted are unique. You can take a look at the below screenshot I'm able to add the **same topic name** multiple times in the name column.

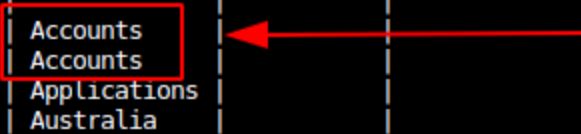

```

postgres=#
postgres=#
postgres=# SELECT * FROM "topic"
postgres-# ORDER BY topicname ASC
postgres-# LIMIT 5;
 id |      topicname      | topicdesc | topicdate
-----+-----+-----+-----
 90 |                      |           |
 69 | Accounts             |           |
 29 | Applications          |           |
 31 | Australia             |           |
  7 | Azerbaijanian_Manat  |           |
(5 rows)

postgres=#
postgres=# INSERT INTO "topic"(topicname) VALUES ( 'Accounts');
INSERT 0 1
postgres=#
postgres=# SELECT * FROM "topic"
postgres-# ORDER BY topicname ASC
postgres-# LIMIT 5;
 id | topicname | topicdesc | topicdate
-----+-----+-----+-----
 90 |           |           |
 91 | Accounts  |           |
 69 | Accounts  |           |
 29 | Applications |           |
 31 | Australia |           |
(5 rows)

postgres=#
postgres=#

```



Code solution: added a UNIQUE statement to topicname

```
"topicname" TEXT UNIQUE NOT NULL CHECK (LENGTH (TRIM ("topicname"))) > 0),
```

6. There is a small improvement required for the posts table as per the requirements:
 - The title of a post can't be empty. To achieve this you will be going to use the TRIM() function to remove the while space. Also, check the length of the characters using the CHECK constraint and LENGTH function. You can take a look at the below screenshot I'm able to add an **empty title** in the title column.

```

postgres=#
postgres=# INSERT INTO "post" ("id", "posttitle", "user_id", "topic_id", "url", "text_content")
postgres-# VALUES (500001, ' ', 5, 3, 'www.google.com', NULL);
INSERT 0 1
postgres=#
postgres-#

```

Code solutions:

- a. added trimming and length check to posttitle

```
"posttitle" TEXT NOT NULL CHECK (LENGTH (TRIM ("posttitle"))) > 0),
```

- b. Added a length constraint of 2000 to url, based on [link](#) provided by reviewer

```
CONSTRAINT "url_length" CHECK (LENGTH("url") <= 2000),
```

7. There is a small improvement required for the comments table as per the requirements:

- A comment's text content can't be empty. To achieve this you will be going to use the TRIM() function to remove the while space. Also, check the length of the characters using the CHECK constraint and LENGTH function. You can take a look at the below screenshot I'm able to add an **empty text content** in the text_content column.

Code solution: added trimming and length check to text: `"text" TEXT NOT NULL CHECK (LENGTH (TRIM ("text")) > 0),`

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:

```

-- migration of usernames from bad_posts and bad_comments to table user
-- adapted based on review #2 feedback
INSERT INTO "user" ("username")
SELECT DISTINCT REGEXP_SPLIT_TO_TABLE(upvotes, ',')
FROM "bad_posts"
UNION
SELECT DISTINCT REGEXP_SPLIT_TO_TABLE(downvotes, ',')
FROM "bad_posts"
UNION
SELECT DISTINCT "username"
FROM "bad_posts"
UNION
SELECT DISTINCT "username"
FROM "bad_comments";

-- migration of distinct topics from bad_posts into table topic
INSERT INTO "topic" ("topicname")
SELECT DISTINCT "topic"
FROM "bad_posts";

-- migrate information from bad_posts into table post
INSERT INTO "post" ("posttitle", "url", "text_content", "user_id",
"topic_id")
SELECT LEFT("bp"."title", 100) AS "posttitle",
       "bp"."url",
       "bp"."text_content",
       "u"."id" AS "user_id",
       "tp"."id" AS "topic_id"
FROM "user" AS "u"
JOIN "bad_posts" AS "bp"
ON "u"."username" = "bp"."username"
JOIN "topic" AS "tp"
ON "bp"."topic" = "tp"."topicname";

-- migrate information from bad_comments into table comment
INSERT INTO "comment" ("text", "user_id", "post_id")
SELECT "bc"."text_content" AS "text",
       "u"."id" AS "user_id",
       "p"."id" AS "post_id"

```

```

FROM "bad_comments" AS "bc"
JOIN "user" AS "u"
ON "u"."username" = "bc"."username"
JOIN "post" AS "p"
ON "p"."id" = "bc"."post_id";

-- migrate information from bad_posts into table vote
INSERT INTO "vote" ("user_id", "vote")
WITH "downvote" AS (
    SELECT "id", REGEXP_SPLIT_TO_TABLE("downvotes", ',') AS "downvotes"
    FROM "bad_posts" AS "bp"),
    "upvote" AS (
    SELECT "id", REGEXP_SPLIT_TO_TABLE("upvotes", ',') AS "upvotes"
    FROM "bad_posts" AS "bp")
SELECT "u"."id", -1 AS "vote"
FROM "downvote" AS "dv"
JOIN "user" AS "u"
ON "dv"."downvotes" = "u"."username"
UNION ALL
SELECT "u"."id", 1 AS "vote"
FROM "upvote" AS "uv"
JOIN "user" AS "u"
ON "uv"."upvotes" = "u"."username";

-- drop initial tables after successful data migration
DROP TABLE "bad_posts";
DROP TABLE "bad_comments";

```

Notes on edit of Data Migration, based on Udacity review #2 (review text copied in for reference):

1. You didn't migrate all user's into the users table. In the upvotes and downvotes columns, some unique users never **create a post** and **comment on any post**. So please migrate all user's into the users table. **Remember:** You can use the Postgres string function `regexp_split_to_table` to unwind the comma-separated votes values into separate rows.

Code solution: added `regexp_split_to_table` functions to user table migration in code above

```
-- migration of usernames from bad_posts and bad_comments to table user
-- adapted based on review #2 feedback
INSERT INTO "user" ("username")
SELECT DISTINCT REGEXP_SPLIT_TO_TABLE(upvotes, ',')
FROM "bad_posts"
UNION
SELECT DISTINCT REGEXP_SPLIT_TO_TABLE(downvotes, ',')
FROM "bad_posts"
UNION
SELECT DISTINCT "username"
FROM "bad_posts"
UNION
SELECT DISTINCT "username"
FROM "bad_comments";
```

2. Due to the missing of **users** for the users table, the votes table didn't have enough upvotes and downvotes into it. Once you migrate all the user's to the users table, you will be able to migrate the upvotes and downvotes to the votes table, so please fix the user table issues.

Comment: users table fixed according to feedback above.