Alessia Taboga, SQL Udacity Nanodegree, project 2 — August 2022

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!

After investigating the current schema, these are my observations for possible improvements:

- Upvotes and downvotes columns contain more values in a single cell (a list of usernames). These cells break one of the rules of first normalisation form. Only a single value per cell should be allowed. This problem could be solved adding more rows or as in my solution, creating a separate entity table for the post_votes.

- The two existing tables do not reference each other. For example, there is no foreign key connecting the bad_comments.post_id column with the bad_posts.id column. Foreign keys must be added to guarantee referential integrity.

- Every entity (ex. users, topics, posts) should have its own table so that can be created/modified/managed independently from the other entities.

- The same values may appear many times in one column. For example the same topic name may appear many times in the bad_posts.topic column. In these cases, it is better to create a topic table with “id” and “topic_name” and use the topic_id in the posts table.

- Username should have a unique constrain as we need users to register/log in with different names.

- The maximum limit of characters for some columns is probably too high (ex 50 for username and 150 for post title). There could be also a discussion with the business owner about the allowed number of characters for posts and comments.

- The bad_comments table allows only comments on existing posts, not on other comments. It contains a column post_id, but not for example a comment_id, indicating a parent comment.

- The current schema hasn’t got any columns with dates/times, so no queries can be carried out to extract information based on these information (ex. which are the latest posts and 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 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:

```
--users table
--with decision of creating unique index on username, case-insensitive

CREATE TABLE "users" (
  "id" SERIAL,
  "username" VARCHAR(25) NOT NULL,
  "date" DATE,
  CONSTRAINT "users_pk" PRIMARY KEY ("id"),
  CONSTRAINT "username_not_empty" CHECK (LENGTH(TRIM("username"))>0)
);
```

```
CREATE UNIQUE INDEX "username_index" ON "users" (LOWER("username"));
```

```
--topics table
--with additional index on topic, case-insensitive pattern

CREATE TABLE "topics" (
  "id" SERIAL,
  "topic_name" VARCHAR(30) NOT NULL,
  "description" VARCHAR(500) DEFAULT NULL,
  CONSTRAINT "topics_pk" PRIMARY KEY ("id"),
  CONSTRAINT "unique_topicname" UNIQUE ("topic_name"),
  CONSTRAINT "topicname_not_empty" CHECK (LENGTH(TRIM("topic_name"))>0)
);

CREATE INDEX "topic_name_pattern_idx"
  ON "topics" (LOWER("topic_name") VARCHAR_PATTERN_OPS);
```

```
--posts table
--with index to search posts by title, in a pattern case-insensitive manner
-and index to speed up search on url

CREATE TABLE "posts" (
  "id" SERIAL,
  "post_title" VARCHAR(100) NOT NULL,
  "url_content" VARCHAR(4000) DEFAULT NULL,
  "text_content" TEXT DEFAULT NULL,
  "user_id" INTEGER,
  "topic_id" INTEGER NOT NULL,
  "timestamp" TIMESTAMP WITH TIME ZONE,
  CONSTRAINT "post_pk" PRIMARY KEY ("id"),
  CONSTRAINT "posts_valid_user" FOREIGN KEY ("user_id")
    REFERENCES "users" ("id") ON DELETE SET NULL,
  CONSTRAINT "posts_valid_topic" FOREIGN KEY ("topic_id")
    REFERENCES "topics" ("id") ON DELETE CASCADE,
  CONSTRAINT "check_url_or_text" CHECK (("url_content" IS NULL)
                                       OR ("text_content" IS NULL)),
  CONSTRAINT "post_title_not_empty" CHECK (LENGTH(TRIM("post_title"))>0)
);

CREATE INDEX "post_title_pattern_idx"
  ON "posts" (LOWER("post_title") VARCHAR_PATTERN_OPS);
```

```
CREATE INDEX "post_url_idx" ON "posts" ("url_content");
```

```
--comments table

CREATE TABLE "comments" (
  "id" SERIAL,
  "comment_text" VARCHAR(2000) NOT NULL,
  "user_id" INTEGER,
  "post_id" INTEGER NOT NULL,
  "comment_id" INTEGER,
  "timestamp" TIMESTAMP WITH TIME ZONE,
  CONSTRAINT "comments_pk" PRIMARY KEY ("id"),
  CONSTRAINT "comm_valid_user" FOREIGN KEY ("user_id")
    REFERENCES "users" ("id") ON DELETE SET NULL,
  CONSTRAINT "comm_valid_post" FOREIGN KEY ("post_id")
    REFERENCES "posts" ("id") ON DELETE CASCADE,
  CONSTRAINT "comm_valid_comment" FOREIGN KEY ("comment_id")
    REFERENCES "comments" ("id") ON DELETE CASCADE,
  CONSTRAINT "comm_text_not_empty" CHECK (LENGTH(TRIM("comment_text"))>0)
);
```

```
--post_votes table

CREATE TABLE "post_votes" (
  "id" SERIAL,
  "user_id" INTEGER,
  "post_id" INTEGER NOT NULL,
  "vote" SMALLINT,
  CONSTRAINT "votes_pk" PRIMARY KEY ("id"),
  CONSTRAINT "votes_valid_user" FOREIGN KEY ("user_id")
    REFERENCES "users" ("id") ON DELETE SET NULL,
  CONSTRAINT "votes_valid_post" FOREIGN KEY ("post_id")
    REFERENCES "posts" ("id") ON DELETE CASCADE,
  CONSTRAINT "unique_user_post" UNIQUE ("user_id","post_id"),
  CONSTRAINT "vote_values" CHECK ("vote" IN (-1,1))
);
```

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 topic_name
INSERT INTO "topics" ("topic_name")
  SELECT DISTINCT "topic" FROM "bad_posts";
```

```
--users table: migrate username from username of both original tables
-- (bad_posts and bad_comments), avoiding repetitions
--UNION should drop repeated values

INSERT INTO "users" ("username")
  SELECT "username" FROM "bad_posts"
  UNION
  SELECT "username" FROM "bad_comments"
  UNION
  SELECT REGEXP_SPLIT_TO_TABLE ("bad_posts"."upvotes",',') FROM "bad_posts"
  UNION
```

```
  SELECT REGEXP_SPLIT_TO_TABLE ("bad_posts"."downvotes",',') FROM
"bad_posts";
```

```
--MIGRATION OF POSTS
--firstly add user_id and topic_id to original bad_posts table

ALTER TABLE "bad_posts"
  ADD COLUMN "user_id" INTEGER,
  ADD COLUMN "topic_id" INTEGER;

UPDATE "bad_posts" SET "user_id"=(
  SELECT "id"
  FROM "users"
  WHERE "users"."username"="bad_posts"."username"
);

UPDATE "bad_posts" SET "topic_id"=(
  SELECT "id"
  FROM "topics"
  WHERE "topics"."topic_name"="bad_posts"."topic"
);
```

```
--then migrate the information of bad_posts.
-- keeping the id of the original table.
-- Note that I need to cut the title to avoid error for not fitting
-- into the required VARCHAR(100)
-- (something that should be discussed with business owner)

INSERT INTO "posts" (
  "id","post_title","url_content","text_content","user_id","topic_id")
  SELECT "id",LEFT("title",100),"url","text_content","user_id","topic_id"
  FROM "bad_posts";
```

```
--MIGRATING COMMENTS
--I add user_id to the original bad_comments table to aid migration
--I have to assume that although not Foreign Key, the post_id
--of the table is correct
ALTER TABLE "bad_comments" ADD COLUMN "user_id" INTEGER;

--update the user_id in bad_comments using a subselect
UPDATE "bad_comments" SET "user_id"=(
  SELECT "id"
  FROM "users"
  WHERE "users"."username"="bad_comments"."username"
);

--migrating the data now
INSERT INTO "comments" ("id","comment_text","user_id","post_id")
  SELECT "id","text_content","user_id","post_id"
  FROM "bad_comments";
```

```
-- migrating votes: upvotes (+1)

INSERT INTO "post_votes"("user_id", "post_id", "vote")
SELECT "users"."id", "uv"."post_id", 1
FROM (
    SELECT "id" AS "post_id", REGEXP_SPLIT_TO_TABLE("upvotes",',') AS
"upv_user"
    FROM "bad_posts") uv
JOIN "users"
ON "users"."username"="uv"."upv_user";

-- migrating votes: downvotes (-1)

INSERT INTO "post_votes"("user_id", "post_id", "vote")
SELECT "users"."id", "dv"."post_id", -1
FROM (
    SELECT "id" AS "post_id", REGEXP_SPLIT_TO_TABLE("downvotes",',') AS
"downv_user"
    FROM "bad_posts") dv
JOIN "users"
ON "users"."username"="dv"."downv_user";
```

```
--RESTORE provided ORIGINAL TABLES
-- dropping the columns I have added to aid migration
ALTER TABLE "bad_posts" DROP COLUMN "user_id";
ALTER TABLE "bad_posts" DROP COLUMN "topic_id";
ALTER TABLE "bad_comments" DROP COLUMN "user_id";
```