Raghad Alamoudi

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. The provided schema is currently denormalized, where the columns from both tables can be segregated into smaller tables. For instance, tables could have been designed for the "topic" and "title" columns in the "bad_posts" table.
2. The table could utilize constraints more efficiently, namely: UNIQUE constraint to avoid repeating values within a column, CHECK constraint to implement any business rules, and more importantly, FOREIGN KEY constraint to provide links between related tables.
3. Some data types in the schema can be improved, such as: converting the "upvotes" and "downvotes" columns from TEXT to NUMERIC, where "upvotes" can be valued at 1 and "downvotes" at 0. The "text_content" column's data type can be changed from TEXT to VARCHAR(n) for the sake of limiting the length of characters per cell/row.
4. Each row should have a value for "username." Therefore, in the DDL, it should be set to NOT NULL.
5. The table does not utilize any form of indexing for the sake of optimizing queries. It would have been worthwhile creating indexes for columns such as: "username," "topic," "title," and so-on-soforth.

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_id 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 subjects:
        i. Subjects names have to be unique.
        ii. The subject's name is at most 30 characters
        iii. The subject's name can't be empty
        iv. Subjects can have an optional description of at most 500 characters.
    c. Allow registered users_id to create new reports on existing topics:
        i. Reports have a required title of at most 100 characters
        ii. The title of a reports can't be empty.
        iii. Reports should contain either a URL or a text content, **but not both**.
        iv. If a subjects gets deleted, all the posts associated with it should be automatically deleted too.
        v. If the user who created the subjects 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 Statement'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 reports gets deleted, all comments associated with it should be automatically deleted too.
        iv. If the user_id who created the statements gets deleted, then the statements will remain, but it will become dissociated from that user.
        v. If a statements 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_id 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:

```sql
-- Part II: Create the DDL for your new schema

-- Drop tables if exists
DROP TABLE IF EXISTS "users_id";

DROP TABLE IF EXISTS "subjects";

DROP TABLE IF EXISTS "reports";

DROP TABLE IF EXISTS "statements";

DROP TABLE IF EXISTS "report_votes";

-- Users table
-- another name for username is login_name
-- ttemp stands for times_temp
CREATE TABLE "users"
  ("users_id" SERIAL PRIMARY KEY, "login_name"  VARCHAR(50) UNIQUE NOT NULL,
    CONSTRAINT "check_users_length_not_zero" CHECK (Length(Trim("login_name")) > 0),
     "login_ttamp" TIMESTAMP);

-- Index for a list of all users who haven't logged in in the last year.
-- another name for username is login_name
-- ttemp stands for times_temp
CREATE INDEX "find_users_not_logged_in_last_year" ON "users" ("login_name",
"login_ttamp");

-- Index for finding a user by their login name.
-- another name for username is login_name
CREATE INDEX "find_users_by_their_login_name" ON "users" ("login_name");

-- subjects table
-- topic name another word for it is subjects name
CREATE TABLE "subjects"
  ("id" SERIAL PRIMARY KEY, "subjects_name"  VARCHAR(30) UNIQUE NOT NULL,
  "description" VARCHAR(500),"user_id" INTEGER, CONSTRAINT
  "check_topics_length_not_zero" CHECK (Length(Trim("subjects_name")) > 0));

-- Indexes to list all statements that don't have any reports.
CREATE INDEX "find_statements_ids_in_statements" ON "statements" ("id");

CREATE INDEX "find_statemenets_ids_in_report" ON "report" ("topic_id");

-- Index for finding a statements by its name.
CREATE INDEX "find_statements_name_in_statements" ON "statements" ("subjects_name");
```

```sql
-- reports table
-- ttontent stands for text content
-- ttemp stands for times_temp
CREATE TABLE "reports"
(
    "id"             SERIAL PRIMARY KEY,
    "title"          VARCHAR(100) NOT NULL,
    "url"            TEXT,
    "tcontent"       TEXT,
    "users_id"       INTEGER,
    "subject_id"       INTEGER,
        CONSTRAINT "check_reports_length_not_zero" CHECK (Length(Trim("title"))
    > 0),
        CONSTRAINT "fk_user" FOREIGN KEY ("users_id") REFERENCES "users" ("id")
    ON
        DELETE SET NULL,
        CONSTRAINT "fk_subjects" FOREIGN KEY ("subjects_id") REFERENCES "subjects" (
    "users_id")
        ON DELETE CASCADE,
        CONSTRAINT "check_text_or_URL_isexist." CHECK ((("url") IS NULL AND (
        "tcontent") IS NOT NULL) OR (("url") IS NOT NULL AND (
    "tcontent")
        IS NULL)),
    "reports_ttamp" TIMESTAMP
);

-- Indexes for a list all users who haven't created any reports.
CREATE INDEX "find_user_ids_in_users" ON "users_id" ("id");

CREATE INDEX "find_user_ids_in_report" ON "reports" ("user_id");

-- Index for a list of latest reports for a given subject.
CREATE INDEX "find_reports_with_timestamp_and_topic" ON "reports" ("URL",
"text_content", "subjects_id", "reports_times_tamp");

-- Index for a list of latest reports for a given user.
CREATE INDEX "find_reports_with_timestamp_and_user" ON "reports" ("URL",
"tcontent", "user_id", "reports_ttamp");

-- Index to find reports with URL.
CREATE INDEX "find_reports_with_URL" ON "reports" ("URL");
```

```sql
-- Comments table
-- pstatements stands for parent_statements
CREATE TABLE "statements"
  (
    "id"                SERIAL PRIMARY KEY,
    "statements"           TEXT NOT NULL,
    "users_id"           INTEGER,
    "subjects_id"       INTEGER,
    "reports_id"           INTEGER,
    "pstatements_id" INTEGER DEFAULT NULL, -- Look here for the i, j queries
    CONSTRAINT "check_reports_length_not_zero" CHECK (Length(Trim("statements")) > 0
    ),
    CONSTRAINT "fk_user" FOREIGN KEY ("users_id") REFERENCES "users" ("id") ON
    DELETE SET NULL,
    CONSTRAINT "fk_subject" FOREIGN KEY ("subjects_id") REFERENCES "subject" ("id")
    ON DELETE CASCADE,
    CONSTRAINT "fk_report" FOREIGN KEY ("reports_id") REFERENCES "reports" ("id") ON
    DELETE CASCADE,
    CONSTRAINT "parent_child_statements_thread" FOREIGN KEY ("pstatements_id")
    REFERENCES "statements" ("users_id") ON DELETE CASCADE
  );

-- Index for all the top-level statements for a given reports.
-- pstatments stands for parent_statements
CREATE INDEX "find_top_level_statements_for_a_report" ON "statements" ("statments",
"post_id", "pstatements_id") WHERE "pstatements_id" = NULL;

-- Index for all the direct children of a parent statements.
-- pstatemenets stands for parent_statements
CREATE INDEX "find_all_the_direct_children_a_parent_statements" ON "statements" (
  "statements", "pcomment_id");

-- Index to list the latest statements made by a given user.
CREATE INDEX "find_latest_statements_by_user" ON "statements" ("statements", "users_id");

-- Votes on reports table
CREATE TABLE "post_votes"
  (
    "id"            SERIAL PRIMARY KEY,
    "reports_vote"     INTEGER NOT NULL,
    "voter_user_id" INTEGER,
    "reports_id"       INTEGER,
    CONSTRAINT "set_values_for_votes" CHECK ("reports_vote" = 1 OR "reports_vote" =
    -1),
    CONSTRAINT "fk_user" FOREIGN KEY ("voter_user_id") REFERENCES "users" ("id"
    ) ON DELETE SET NULL,
    CONSTRAINT "fk_post" FOREIGN KEY ("reports_id") REFERENCES "reports" ("id") ON
    DELETE CASCADE
  );

-- Index to find score of reports.
CREATE INDEX "find_score_of_repots" ON "reports_votes" ("reports_vote", "post_id");
```

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. subjects descriptions can all be empty
2. Since the bad_statements table doesn't have the threading feature, you can migrate all comments as top-level statements, 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 statements, and haven't created any reports. You'll have to create those users too.
5. The order of your migrations matter! For example, since posts depend on users and subjects, 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_reports and bad_statements to your new database schema:

```sql
-- Part III: Migrate the provided data

-- Insert all unique login_name from both initial tables.
INSERT INTO "users"  ("login_name")
WITH unique_usernames
     AS (SELECT login_name
         FROM   "bad_repots"
         UNION ALL
         SELECT login_name
         FROM   "bad_statements"),
     distinct_login_name
     AS (SELECT login_name
         FROM   unique_login_name)
SELECT DISTINCT login_name
FROM   distinct_login_name
ORDER  BY 1 ASC;

-- Insert distinct topics from "bad_reports"
INSERT INTO "subjects" ("subjects_name")
SELECT DISTINCT subjects
FROM   "bad_reports";

-- Insert fields from the "bad_reports", "users" and "reports".

-- usu stands for user
-- bad stands for bad reports
-- sub stands for subject
INSERT INTO "reports_votes"
INSERT INTO "reports"
            ("subjects","url", "text_content", "user_id", "_id")
SELECT bad.subjects, bad.url, bad.text_content,
       usu.id AS user_id,
       sub.id AS subjects_id
FROM   "bad_reports" bad
       join "users" usu
         ON bad.login_name = usu.login_name
       join "subjects" sub
         ON bad.subjects = sub.subjects_name
WHERE  Left(bad.title,100);
```

```
-- Insert statements and ids in "statements".

-- rep stands for reports
-- usu stands for user
-- bad stands for bad reports
-- bst stand for bad_statements
INSERT INTO "reports_votes"
INSERT INTO "statements"
            ("statements", "user_id", "subjects_id", "reports_id")
SELECT bst.text_content AS STATEMENT,
       rep.user_id,
       rep.subject_id,
       rep.id           AS post_id
FROM   "bad_reports" bad
       join "reports" rep
         ON bad.title = pos.title
       join "users_id" usu
         ON rep.user_id = usu.id
       join "bad_statements" bst
         ON usu.login_name = bst.login_name;

-- Insert upvotes & downvotes in "reports_votes".

-- rep stands for reports
-- usu stands for user
-- bad stands for bad reports
INSERT INTO "reports_votes"
            ("reports_vote", "voter_user_id", "reports_id")
WITH "bad_posts_upvotes"
     AS (SELECT title,
                Regexp_split_to_table(bad.upvotes, ',') AS login_name_upvotes
         FROM   "bad_reports" bad),
     "bad_reports_downvotes"
     AS (SELECT title,
                Regexp_split_to_table(bad.downvotes, ',') AS login_name_downvotes
         FROM   "bad_reports" bad) SELECT 1     AS reports_vote,
       usu.id AS voter_user_id,
       rep.id AS post_id
FROM   "bad_reports_upvotes" bpu
       join "reports" rep
         ON bpu.title = po.title
       join "users" usu
         ON bpu.login_name_upvotes = usu.login_name
UNION ALL
SELECT -1    AS reports_vote,
       usu.id AS voter_user_id,
       reports.id AS reports_id
FROM   "bad_reports_downvotes" bpd
       join "reports" rep
         ON bpd.title = rep.title
       join "users" usu
         ON bpd.login_name_downvotes = usu.login_name;
```