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!

Should be created an table for the topics, containing, at least, an ID (using a surrogate key and being the primary key) and the topic names that would be unique and not null;
Having created the topics table, we should switch the topics in the bad_posts by its respective ID and constraint it as an foreign key.

Should be created an table for the users, containing, at least, an ID (using a surrogate key and being the primary key) and a username that would be unique and not null;
Having created the users table, we should switch the username in both tables (bad_posts and bad_comments) by its respective ID and make it an foreign key.

The up and downvotes should be a separate table containing the post id and the user id that up/downvoted the post.

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:

```
-- 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
CREATE TABLE "usuarios" (
  "usuario_id" SERIAL PRIMARY KEY,
  "nome_de_usuario" VARCHAR(25) CONSTRAINT "usuario_unico" UNIQUE
);

-- iii. Usernames can't be empty
```

```
ALTER TABLE "usuarios" ADD CONSTRAINT "usuario_nao_nulo" CHECK
(LENGTH(nome_de_usuario) > 0);

-----------------------------------------------------------------------------

-- b. Allow registered users to create new topics ->  a tabela deve ter,
pelo menos, o tópico e o usuário
-- i. Topic names have to be unique
-- ii. The topic’s name is at most 30 characters
CREATE TABLE "topicos" (
  "topico_id" SERIAL PRIMARY KEY,
  "nome_do_topico" VARCHAR(30) CONSTRAINT "nome_do_topico_unico" UNIQUE,
  "usuario_id" INTEGER CONSTRAINT "usuario_valido" REFERENCES
"usuarios"("usuario_id") ON DELETE SET NULL -- -> assim como foi feito no
resto do projeto, ao deletar o usuário, o topico continuará existindo
);

-- iii. The topic’s name can’t be empty
ALTER TABLE "topicos" ADD CONSTRAINT "topico_nao_nulo" CHECK
(LENGTH(nome_do_topico) > 0);

-- iv. Topics can have an optional description of at most 500 characters
ALTER TABLE "topicos" ADD COLUMN "descricao" VARCHAR(500);

-----------------------------------------------------------------------------

-- c. Allow registered users to create new posts on existing topics -> a
tabela deve ter, pelo menos, o post, o tópico e o usuário
-- i. Posts have a required title of at most 100 characters
-- 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" (
  "post_id" SERIAL PRIMARY KEY,
  "titulo_do_post" VARCHAR(100),
  "topico_id" INTEGER CONSTRAINT "topico_valido" REFERENCES
"topicos"("topico_id") ON DELETE CASCADE,
  "usuario_id" INTEGER CONSTRAINT "usuario_valido" REFERENCES
"usuarios"("usuario_id") ON DELETE SET NULL
);

ALTER TABLE “posts” ADD CONSTRAINT “topico_id_nao_nulo” CHECK (“topico_id”
IS NOT NULL);

-- ii. The title of a post can’t be empty
ALTER TABLE "posts" ADD CONSTRAINT "titulo_nao_nulo" CHECK
(LENGTH(titulo_do_post) > 0);

-- iii. Posts should contain either a URL or a text content, but not both.
ALTER TABLE "posts" ADD COLUMN "url" TEXT;

ALTER TABLE "posts" ADD COLUMN "conteudo" TEXT;
```

```
ALTER TABLE "posts" ADD CONSTRAINT "url_ou_conteudo" CHECK (("url" IS NULL
AND "conteudo" IS NOT NULL) OR ("url" IS NOT NULL AND "conteudo" IS NULL));

--------------------------------------------------------------------------
----

-- d. Allow registered users to comment on existing posts -> a tabela deve
ter, pelo menos, o comentário, o post e o usuário
-- 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
CREATE TABLE "comentarios" (
  "comentario_id" SERIAL PRIMARY KEY,
  "comentario" TEXT,
  "post_id" INTEGER CONSTRAINT "post_valido" REFERENCES "posts"("post_id")
ON DELETE CASCADE,
  "usuario_id" INTEGER CONSTRAINT "usuario_valido" REFERENCES
"usuarios"("usuario_id") ON DELETE SET NULL
);

ALTER TABLE “comentarios” ADD CONSTRAINT “post_id_nao_nulo” CHECK (“post_id”
IS NOT NULL);

-- i. A comment’s text content can’t be empty
ALTER TABLE "comentarios" ADD CONSTRAINT "comentario_nao_nulo" CHECK
(LENGTH(comentario) > 0);

-- ii. Contrary to the current linear comments, the new structure should
allow comment threads at arbitrary levels
-- v. If a comment gets deleted, then all its descendants in the thread
structure should be automatically deleted too
ALTER TABLE "comentarios" ADD COLUMN "nivel_do_comentario" INTEGER CHECK
("nivel_do_comentario" >= 1);

-- o comentário, seja ele o pai, seja ele o descendente, não deixa de ser
somente um comentário e, por isso, eu decidi por adicionar somente um nível
a ele e a pessoa que de fato for
-- fazer o design do website separa esses comentários pelo seu nível. Assim
como foi feito VÁRIAS VEZES durante o curso, quando tínhamos, por exemplo,
um gerente. Ele não deixa de ser um
-- funcionário, só tem um nível diferente dos outros.

-- a minha ideia aqui é que o primeiro comentário tenha o nível 1, as
respostas a esse comentário tenham o nível 2 e assim sucessivamente

--------------------------------------------------------------------------

-- e. Make sure that a given user can only vote once on a given post -> a
tabela deve ter, pelo menos, o usuário, o voto e o post
-- 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
CREATE TABLE "votos" (
  "usuario_id" INTEGER CONSTRAINT "usuario_valido"
    REFERENCES "usuarios"("usuario_id") ON DELETE SET NULL,
  "post_id" INTEGER CONSTRAINT "post_valido"
    REFERENCES "posts"("post_id") ON DELETE CASCADE,
  "voto" SMALLINT,
  CONSTRAINT "id" PRIMARY KEY ("usuario_id", "post_id"),
  CONSTRAINT "voto_unico" UNIQUE ("post_id", "usuario_id") -- -> para cada
post, o usuário não pode se repetir
);

ALTER TABLE “votos” ADD CONSTRAINT “post_id_nao_nulo” CHECK (“post_id” IS
NOT NULL);

-- i. Hint: you can store the (up/down) value of the vote as the values 1
and -1 respectively
ALTER TABLE "votos" ADD CONSTRAINT "voto_valido" CHECK ("voto" = '-1' OR
"voto" = '1');

--------------------------------------------------------------------------------

-- 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.
CREATE TABLE "log_in" (
  "usuario_id" INTEGER CONSTRAINT "usuario_valido" REFERENCES
"usuarios"("usuario_id"),
  "quando" TIMESTAMP WITH TIME ZONE -- uma rede social, normalmente, tem
escala global e, por isso, decidi manter os fuso-horários
);

-- b. List all users who haven’t created any post
-- Já é possível ter essa resposta ao usar, por exemplo, a função SELECT
u.usuario_id FROM posts AS p RIGHT JOIN usuarios AS u WHERE u.usuario_id IS
NOT IN posts

-- c. Find a user by their username
-- Já é possível fazer isso, pois o nome de usuário é único, mas podemos
facilitar essa busca ao criar um index
CREATE INDEX "achar_usuario" ON "usuarios"("nome_de_usuario");

-- d. List all topics that don’t have any posts
-- Assim como na letra b, já é possível fazer isso por meio de DQLs

-- e. Find a topic by its name
-- Assim como na letra c, já é possível fazer isso, pois o nome do tópico é
único, mas podemos facilitar essa busca ao criar um index
CREATE INDEX "achar_topico" ON "topicos"("nome_do_topico");
```

```
-- f. List the latest 20 posts for a given topic
-- g. List the latest 20 posts made by a given user
ALTER TABLE "posts" ADD COLUMN "quando" TIMESTAMP WITH TIME ZONE;

-- h. List all the top-level comments (those that don’t have a parent
comment) for a given post
-- i. List all the direct children of a parent comment
-- Assim como em questões anteriores, já é possível fazer isso com, por
exemplo, SELECT * FROM comentarios WHERE nivel_do_comentario = 1 (no caso de
comentários originais), mas é possível
-- criar um index para facilitar essa busca
CREATE INDEX "comentario_original" ON "comentarios"("nivel_do_comentario");

-- j. List the latest 20 comments made by a given user
ALTER TABLE "comentarios" ADD COLUMN "quando" TIMESTAMP WITH TIME ZONE;

-- l. Compute the score of a post, defined as the difference between the
number of upvotes and the number of downvotes
-- Já é possível fazer isso por meio de DQLs. Como, por exemplo, SELECT
SUM(voto) FROM votos WHERE post_id = 123, mas, novamente, pode-se criar um
index para facilitar a busca pelo post
CREATE INDEX "achar_post" ON "votos"("post_id");

-- 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.

-- 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:

```
-- EXPLORANDO O BANCO DE DADOS INICIAL

-- TABELA "bad_comments"
-- COLUNAS "id" SERIAL PRIMARY KEY,
--         "username" VARCHAR(50),
--         "post_id" BIGINT,
--         "text_content" TEXT

-- TABELA "bad_posts"
-- COLUNAS "id" SERIAL PRIMARY KEY,
--         "topic" VARCHAR(50),
--         "username" VARCHAR(50),
--         "title" VARCHAR(150),
--         "url" VARCHAR(4000) DEFAULT NULL,
--         "text_content" TEXT,
--         "upvotes" TEXT,
--         "downvotes" TEXT
```

```
-- Primeiro, vou migrar os dados referentes ao usuário, pois sem eles
não é possível interagir com nada da rede social, e vou fazendo as
migrações como se fosse uma árvore e suas
-- ramificações ---> 5. The order of your migrations matter! For
example, since posts depend on users and topics, you'll have to
migrate the latter first.

-- 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
CREATE TABLE "usuarios_tabela_provisoria" (
  "id" SERIAL PRIMARY KEY,
  "usuario" VARCHAR(25)
);

INSERT INTO "usuarios_tabela_provisoria"("usuario") (
  SELECT username
    FROM bad_comments
  );

INSERT INTO "usuarios_tabela_provisoria"("usuario") (
  SELECT username
    FROM bad_posts
);

INSERT INTO "usuarios_tabela_provisoria"("usuario") (
  SELECT regexp_split_to_table("upvotes", ',')
    FROM bad_posts
);

INSERT INTO "usuarios_tabela_provisoria"("usuario") (
  SELECT regexp_split_to_table("downvotes", ',')
    FROM bad_posts
);

INSERT INTO "usuarios"("nome_de_usuario") (
  SELECT DISTINCT usuario
    FROM usuarios_tabela_provisoria
);

DROP TABLE "usuarios_tabela_provisoria";

-- Tópicos
-- 1.Topic descriptions can all be empty
INSERT INTO "topicos"("nome_do_topico", "descricao")
  SELECT DISTINCT topic,
                  NULL AS descricao
```

```
            FROM bad_posts;

-- Posts
INSERT INTO "posts"("post_id", "titulo_do_post", "url", "conteudo",
"topico_id", "usuario_id")
  SELECT bp.id,
         LEFT(bp.title, 100),
         bp.url,
         bp.text_content,
         t.topico_id AS topico_id,
         u.usuario_id AS usuario_id
    FROM bad_posts AS bp
    JOIN topicos AS t
      ON bp.topic = t.nome_do_topico
    JOIN usuarios AS u
      ON bp.username = u.nome_de_usuario;

INSERT INTO "posts"("post_id", "titulo_do_post", "url", "conteudo",
"topico_id", "usuario_id")
  SELECT bp.id,
         bp.title,
         bp.url,
         bp.text_content,
         t.topico_id AS topico_id,
         u.usuario_id AS usuario_id
    FROM bad_posts AS bp
    JOIN topicos AS t
      ON bp.topic = t.nome_do_topico
    JOIN usuarios AS u
      ON bp.username = u.nome_de_usuario
   WHERE LENGTH(bp.title) <= 100;

-- Comentarios
-- 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
INSERT INTO "comentarios"("comentario", "post_id", "usuario_id",
"nivel_do_comentario")
  SELECT bc.text_content,
         bc.post_id,
         u.usuario_id,
         '1'
    FROM bad_comments AS bc
    JOIN usuarios AS u
      ON bc.username = u.nome_de_usuario;

-- Votos
```

```
INSERT INTO "votos"("usuario_id", "post_id", "voto")
  SELECT u.usuario_id,
         vp.post_id,
         '1'
    FROM (SELECT id AS post_id,
                 regexp_split_to_table("upvotes", ',') AS usuario
            FROM bad_posts) AS vp
    JOIN usuarios AS u
      ON vp.usuario = u.nome_de_usuario;

INSERT INTO "votos"("usuario_id", "post_id", "voto")
  SELECT u.usuario_id,
         vn.post_id,
         '-1'
    FROM (SELECT id AS post_id,
                 regexp_split_to_table("downvotes", ',') AS usuario
            FROM bad_posts) AS vn
    JOIN usuarios AS u
      ON vn.usuario = u.nome_de_usuario;
```