# 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. There are not any foreign key constraints and the data type for the one field that could act as a foreign key in the design does not match the data type of the primary key in the “bad_posts” table. This creates a couple problems. If you tried to create a foreign key constraint, you would first have to type cast the field, which would severely slow or halt use of the database. Next, numerical data types such as integer or biginteger can also contain negative numbers, while auto-incrementing counterparts only contain positive numbers. This could cause additional problems with troubleshooting and correcting database structure.
2. The data is missing a critical piece of information and that is the user that created a given topic. I am going to add a default value of NULL for the foreign key in what will be the new “topics” table that references a “users” table.
3. The database doesn’t contain an option for users to register.
4. The current structure permits users to vote on a given post as much as they want, which would lead to excessive disk use. Somebody with malicious intent could use this as a target to launch a Denial of Service attack. This can be fixed with a bridge table using a composite primary key to only allow 1 vote per post per user.
5. The “text_content” field in both tables uses the “TEXT” data type which allows unlimited characters. A better option would be to use “VARCHAR” with a reasonable length.
6. The maximum length for the “url” field in “bad_posts” table is 4000 characters. This is not reasonable for a “url” length. A significantly smaller length is needed for this to again save disk space and prevent or slow down attacks to the system.
7. The “upvotes” and “downvotes” have a “TEXT” data type which is wrong. The fields should have a numeric data type (SMALLINT specifically) to allow various aggregations to be applied to voting data. The data provided in the table for the two fields contain multiple users, which also violates the rule of first normal form that states a field should contain only 1 value.
8. There are not any kind of constraints or indexes to validate data quality and improve query speeds.
9. The “topic” field in the “bad_posts” table does not depend on the primary key of the table. A separate “topics” table needs to be created.

## 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. Note: I wrote queries from this last as a bonus. They are at the end of the document.
    a. List all users who haven't logged in in the last year.
    I did not add any indexes for this query. Updating an index with a condition containing an expression would add considerable overhead to database operations. Updating this type of index has the potential to severely slow down or bring services to a halt for users of the platform and analysts/DBAs. I believe a sequential scan is the best choice to handle this type of query.

b. List all users who haven't created any post.
   I created an index on the foreign key "author_id" of the "posts" table that references the "users" table.
c. Find a user by their username.
   The unique index on the "username" field takes care of this.
d. List all topics that don't have any posts.
   I created an index on the foreign key "topic_id" of the "posts" table that references the "topics" table.
e. Find a topic by its name.
   I created an index using pattern_ops on the "name" field on "topics" table because a user might perform a partial match in a case insensitive way. There is also a unique index built for this field per requirements.
f. List the latest 20 posts for a given topic.
   This has been taken care of indexes created on the "name" field of "topics" and an index created on the "topic_id" foreign key of "posts" referencing the "topics" table.
g. List the latest 20 posts made by a given user.
   This has been taken care of by the unique index on the "username" field in the "users" table and the index on the foreign key in the "posts" table that references the "users" table.
h. Find all posts that link to a specific URL, for moderation purposes.
   I created an index on the "url" field of the "posts" table.
i. List all the top-level comments (those that don't have a parent comment) for a given post.
   I created an index on "parent_id" of the comments table.
j. List all the direct children of a parent comment.
   This is taken care of by the unique index built by the primary key in the "comments" table.
k. List the latest 20 comments made by a given user.
   I created an index on the primary key ("author_id") in the "comments" table that references the "users" table.
l. Compute the score of a post, defined as the difference between the number of upvotes and the number of downvotes.
   I helped boost the speed of this query by taking care of field order in the construction of the primary key for the "user_votes" table. I placed the "post_id" field that references the "posts" table **first** in the list of fields making up the primary key because **order does matter** in creation of unique indexes. This helped minimize the number of total indexes that needed to be created. I also created an index on the "vote" column.

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:

Notes: I set the size of the "url", "text_content" fields based on maximum lengths of the data in the provided tables and on research. The maximum length of the "url" field in existing data was 24 characters. I performed an internet search on this topic and found a post on StackOverflow (https://stackoverflow.com/a/25731768) and went along with its suggestion for a maximum length of 500 characters. Internet browsers allow for much longer lengths, but many websites have embedded software to shorten the length of a url to share with others.

Both "text_content" fields in the existing data had maximum lengths of just over 600 characters. I chose 1500 maximum characters for these fields as this is in the 2-3 times more range compared to existing data. This allows some flexibility in length of content but at the same time provides reasonable limits to help conserve disk space.

I chose to use all BIGINT for primary key and foreign key data types as an application of this type could become very large.

```
CREATE TABLE users
(
  id BIGSERIAL CONSTRAINT "users_pk" PRIMARY KEY,
  username VARCHAR(25) CONSTRAINT "username_not_null" NOT NULL,
  last_login TIMESTAMP,
  created_date TIMESTAMP,
  created_by VARCHAR(25),
  updated_date TIMESTAMP,
  updated_by VARCHAR(25),
  CONSTRAINT "empty_usernames_not_allowed" CHECK (LENGTH(TRIM("username")) > 0)
);

CREATE UNIQUE INDEX "unique_username" ON "users"(TRIM("username"));
```

```
CREATE TABLE topics
(
  id BIGSERIAL CONSTRAINT "topics_pk" PRIMARY KEY,
  user_id BIGINT DEFAULT NULL,
  name VARCHAR(30) CONSTRAINT "topic_name_not_null" NOT NULL,
  description VARCHAR(500) DEFAULT NULL,
  created_date TIMESTAMP,
  created_by VARCHAR(25),
  updated_date TIMESTAMP,
  updated_by VARCHAR(25),
  CONSTRAINT "empty_names_not_allowed" CHECK (LENGTH(TRIM("name")) > 0)
);

CREATE UNIQUE INDEX "unique_topic" ON "topics"(TRIM("name"));
CREATE INDEX ON "topics"(LOWER("name") VARCHAR_PATTERN_OPS);

CREATE TABLE posts
(
  id BIGSERIAL CONSTRAINT "posts_pk" PRIMARY KEY,
  author_id BIGINT,
  topic_id BIGINT CONSTRAINT "topic_required" NOT NULL,
  title VARCHAR(100) CONSTRAINT "title_not_null" NOT NULL,
  url VARCHAR(500) DEFAULT NULL,
  text_content VARCHAR(1500) DEFAULT NULL,
  created_date TIMESTAMP,
  created_by VARCHAR(25),
  updated_date TIMESTAMP,
  updated_by VARCHAR(25),
  CONSTRAINT "empty_titles_not_allowed" CHECK (LENGTH(TRIM("title")) > 0),
  CONSTRAINT "url_or_txtContent" CHECK
  (
    (NULLIF(url,'') IS NULL OR NULLIF(text_content,'') IS NULL)
    AND NOT
    (NULLIF(url,'') IS NULL AND NULLIF(text_content,'') IS NULL)
  ),
  CONSTRAINT "posts_to_topics_fk" FOREIGN KEY ("topic_id")
    REFERENCES "topics" ON DELETE CASCADE,
  CONSTRAINT "posts_to_users_fk" FOREIGN KEY ("author_id")
    REFERENCES "users" ON DELETE SET NULL
);

CREATE INDEX "posts_by_user" ON "posts"("author_id");
CREATE INDEX "topics_post_matching" ON "posts"("topic_id");
CREATE INDEX "find_post_with_url" ON "posts"("url");
```

```
CREATE TABLE comments
(
  id BIGSERIAL CONSTRAINT "comments_pk" PRIMARY KEY,
  author_id BIGINT,
  post_id BIGINT,
  text_content VARCHAR(1500) CONSTRAINT "comment_not_null" NOT NULL,
  parent_id BIGINT DEFAULT NULL,
  created_date TIMESTAMP,
  created_by VARCHAR(25),
  updated_date TIMESTAMP,
  updated_by VARCHAR(25),
  CONSTRAINT "empty_comments_not_allowed" CHECK (LENGTH(TRIM("text_content"))>0),
  CONSTRAINT "parent_child_comments_fk" FOREIGN KEY ("parent_id")
    REFERENCES "comments" ON DELETE CASCADE,
  CONSTRAINT "comment_to_post_fk" FOREIGN KEY ("post_id")
    REFERENCES "posts" ON DELETE CASCADE,
  CONSTRAINT "comment_to_users_fk" FOREIGN KEY ("author_id")
    REFERENCES "users" ON DELETE SET NULL
);

CREATE INDEX "find_parent_comments" ON "comments"("parent_id");
CREATE INDEX "find_comments_by_user" ON "comments"("author_id");

CREATE TABLE user_votes
(
  user_id BIGINT,
  post_id BIGINT,
  vote SMALLINT CONSTRAINT "valid_votes" CHECK ("vote" IN (-1, 1)),
  created_date TIMESTAMP,
  created_by VARCHAR(25),
  updated_date TIMESTAMP,
  updated_by VARCHAR(25),
  CONSTRAINT "user_votes_pk" PRIMARY KEY ("post_id","user_id"),
  CONSTRAINT "votes_to_users_fk" FOREIGN KEY ("user_id")
    REFERENCES "users" ON DELETE SET NULL,
  CONSTRAINT "votes_to_posts_fk" FOREIGN KEY ("post_id")
    REFERENCES "posts" ON DELETE CASCADE
);

CREATE INDEX "compute_votes" ON "user_votes"("vote");
```

## 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:

**STARTS ON NEXT PAGE**

```sql
/*
    Part 2 - Data Migration
*/

/* Step 1 - Migrate data to "users" table
   I used "UNION" to remove all duplicates. */

INSERT INTO "users"("username")
  SELECT username
  FROM   bad_posts

  UNION

  SELECT regexp_split_to_table(upvotes, ',')::VARCHAR AS username
  FROM   bad_posts

  UNION

  SELECT regexp_split_to_table(downvotes, ',')::VARCHAR AS username
  FROM   bad_posts

  UNION

  SELECT (username::VARCHAR) AS username
  FROM   bad_comments;

/* Step 2 - Migrate data to "topics" table
   "DISTINCT" used to remove duplicates */

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

```sql
/* Step 3 - Migrate data to "posts" table */

INSERT INTO
"posts"("id","author_id","topic_id","title","url","text_content")
  SELECT  bp.id::BIGINT AS posts_pk,
          u.id AS author_id,
          t.id AS topic_id,
          bp.title::VARCHAR(100),
          bp.url::VARCHAR(500),
          bp.text_content::VARCHAR
  FROM topics t
       INNER JOIN
       bad_posts bp
       ON
       t.name = bp.topic
       INNER JOIN
       users u
       ON
       bp.username = u.username;

/* Step 4 - Migrate data to "comments" table */

INSERT INTO "comments"("author_id","post_id","text_content")
  SELECT  u.id AS author_id,
          bc.post_id,
          bc.text_content::VARCHAR(1500)
  FROM users u
       INNER JOIN
       bad_comments bc
       ON u.username = bc.username;
```

```sql
/* Step 5 - Migrate data to "user_votes" table */

INSERT INTO "user_votes"("user_id","post_id", "vote")
 SELECT  t2.user_id,
         t1.post_id,
         t1.vote
 FROM (
       (
        SELECT  bp.id::BIGINT AS post_id,
                regexp_split_to_table(bp.upvotes,',')::VARCHAR AS username,
                1::SMALLINT AS vote
        FROM    bad_posts bp
       ) AS t1
       INNER JOIN
       (
        SELECT  u.id AS user_id,
                u.username
        FROM users u
       ) AS t2
       ON t1.username = t2.username
     )

 UNION ALL

 SELECT  t4.user_id,
         t3.post_id,
         t3.vote
 FROM (
      (
       SELECT bp.id::BIGINT AS post_id,
              regexp_split_to_table(bp.downvotes,',')::VARCHAR AS username,
              -1::SMALLINT AS vote
       FROM    bad_posts bp
      ) AS t3
      INNER JOIN
      (
       SELECT u.id AS user_id,
              u.username
       FROM users u
      )AS t4
      ON t3.username = t4.username
);
```

```sql
/*
  A. List all users who haven't logged in the last year.
*/

SELECT  username
FROM    users
WHERE   last_login < NOW() - '1 year'::INTERVAL;

/*
  B. List all users who haven't created any post.
*/

SELECT  u.username
FROM    users u
        LEFT JOIN
        posts p
        ON u.id = p.author_id
WHERE   p.author_id IS NULL;

/*
  C. Find a user by their username.
*/

SELECT *
FROM users
WHERE username = 'Zula71';

/*
  D. List all topics that don't have any posts.
*/

SELECT  t.name
FROM    topics t
        LEFT JOIN
        posts p
        ON t.id = p.topic_id
WHERE   p.topic_id IS NULL;
```

```sql
/*
  E. Find a topic by its name.
*/

SELECT  *
FROM    topics
WHERE   name = 'Beauty';

/*
  F. List the latest 20 posts for a given topic
*/

SELECT  t.name,
        p.title,
        u.username AS author,
        p.url,
        p.text_content
FROM    topics t
        INNER JOIN
        posts p
        ON t.id = p.topic_id
        INNER JOIN
        users u
        ON u.id = p.author_id
WHERE   t.name = 'calculate'
ORDER BY  p.created_date DESC
LIMIT 20;

/*
  G.  List the latest 20 posts for a given user
*/

SELECT  p.title,
        u.username AS author,
        p.url,
        p.text_content
FROM    posts p
        INNER JOIN
        users u
        ON u.id = p.author_id AND u.username = 'Dora55'
ORDER BY p.created_date DESC
LIMIT 20;
```

```sql
/*
  H. Final all posts that link to a specific URL, for moderation purposes.
*/

SELECT p.id,
       p.title,
       u.username AS author,
       u.id AS user_id,
       p.url
FROM   posts p
       INNER JOIN
       users u
       ON u.id = p.author_id  AND  p.url = 'http://vivien.org'
ORDER BY p.id ASC;

/*
  I. List all the top-level comments (those that don't have a parent
comment) for a given post.
*/

SELECT  p.title post,
        u.username AS author,
        c.text_content
FROM    posts p
        INNER JOIN
        comments c
        ON p.id = c.post_id AND c.parent_id IS NULL AND p.id = 10000
        INNER JOIN
        users u
        ON u.id = c.author_id;
```

```sql
/*
  J. List all the direct children of a parent comment.
*/

SELECT  u.username AS author,
        parent.id AS parent_comment_id,
        child.id AS child_comment_id,
        child.created_date AS comment_date,
        child.text_content
FROM    comments parent
        INNER JOIN
        comments child
        ON child.parent_id = parent.id AND parent.id = 1
        INNER JOIN
        users u
        ON child.author_id = u.id
ORDER BY 4 ASC;

/*
  K.  List the latest 20 comments made by a given user.
*/

SELECT  u.username AS author,
        c.text_content
FROM    comments c
        INNER JOIN
        users u
        ON c.author_id = u.id AND u.username = 'Kolby.Langosh'
ORDER BY c.created_date DESC
LIMIT 20;
```

```sql
/*
  L. Compute the score of a post, defined as the difference between the
number of upvotes and the number of downvotes.
*/

SELECT  p.title,
        SUM(uv.vote) AS post_score
FROM    posts p
        INNER JOIN
        user_votes uv
        ON p.id = uv.post_id
GROUP BY p.title
ORDER BY 2 DESC, 1 ASC;
```