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. Both tables don’t show how usernames reference a registered list of users. We need to create a table of unique registered users, and then reference that within both these tables (e.g., username REFERENCES users (username)). This would ensure that only registered users are posting and commenting. 2. Upvotes and downvotes in the bad\_posts table include the names of the users who upvoted or downvoted respectively separated by a comma. This information needs to be placed in its own table(s) and split into rows in order to be normalized (e.g., column 1 = post\_id, column 2 = username). 3. Post\_id in bad\_comments should reference id in bad\_posts so that there is a connection between which post is being commented on and the post itself. |

## 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:
   1. Allow new users to register:
      1. Each username has to be unique
      2. Usernames can be composed of at most 25 characters
      3. Usernames can’t be empty
      4. We won’t worry about user passwords for this project
   2. Allow registered users to create new topics:
      1. Topic names have to be unique.
      2. The topic’s name is at most 30 characters
      3. The topic’s name can’t be empty
      4. Topics can have an optional description of at most 500 characters.
   3. Allow registered users to create new posts on existing topics:
      1. Posts have a required title of at most 100 characters
      2. The title of a post can’t be empty.
      3. Posts should contain either a URL or a text content, **but not both**.
      4. If a topic gets deleted, all the posts associated with it should be automatically deleted too.
      5. If the user who created the post gets deleted, then the post will remain, but it will become dissociated from that user.
   4. Allow registered users to comment on existing posts:
      1. A comment’s text content can’t be empty.
      2. Contrary to the current linear comments, the new structure should allow comment threads at arbitrary levels.
      3. If a post gets deleted, all comments associated with it should be automatically deleted too.
      4. If the user who created the comment gets deleted, then the comment will remain, but it will become dissociated from that user.
      5. If a comment gets deleted, then all its descendants in the thread structure should be automatically deleted too.
   5. Make sure that a given user can only vote once on a given post:
      1. Hint: you can store the (up/down) value of the vote as the values 1 and -1 respectively.
      2. If the user who cast a vote gets deleted, then all their votes will remain, but will become dissociated from the user.
      3. 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.
   1. List all users who haven’t logged in in the last year.
   2. List all users who haven’t created any post.
   3. Find a user by their username.
   4. List all topics that don’t have any posts.
   5. Find a topic by its name.
   6. List the latest 20 posts for a given topic.
   7. List the latest 20 posts made by a given user.
   8. Find all posts that link to a specific URL, for moderation purposes.
   9. List all the top-level comments (those that don’t have a parent comment) for a given post.
   10. List all the direct children of a parent comment.
   11. List the latest 20 comments made by a given user.
   12. 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: Allow new users to register: Each username has to be unique Usernames can be composed of at most 25 characters Usernames can't be empty We won't worry about user passwords for this project\*/   CREATE TABLE users (  id SERIAL PRIMARY KEY,  username VARCHAR(25) UNIQUE NOT NULL,  -- to users who haven't logged in in the last year:  last\_login DATE,  CONSTRAINT valid\_username CHECK (LENGTH(TRIM(username)) > 0) );  CREATE UNIQUE INDEX ON users (LOWER(username)); -- to prevent 'Bob' and 'bob' CREATE INDEX last\_login\_index ON users (last\_login); --to speed up date searches CREATE INDEX user\_inded ON users (username); --to speed up finding a user by username  --INSERT INTO users (username) VALUES ('abcdefghijklmnopqrstuvwxyz'); --checking length constraint (checked) --INSERT INTO users (username) VALUES ('Bob'), ('bob'); --checking case sensitivity (checked)  /\*Topic names have to be unique. The topic's name is at most 30 characters The topic's name can't be empty Topics can have an optional description of at most 500 characters.\*/  CREATE TABLE topics (  id SERIAL PRIMARY KEY,  name VARCHAR(30) UNIQUE NOT NULL,  description VARCHAR(500),  -- preventing spaces as name:  CONSTRAINT valid\_name CHECK (LENGTH(TRIM(name))>0) );  CREATE INDEX topic\_name\_index ON topics (name);  /\*Allow registered users to create new posts on existing topics: Posts have a required title of at most 100 characters The title of a post can't be empty. Posts should contain either a URL or a text content, but not both. If a topic gets deleted, all the posts associated with it should be automatically deleted too. 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 (  id SERIAL PRIMARY KEY,  title VARCHAR(500) NOT NULL, --post titles do not need to be unique  url VARCHAR(500),  text\_content VARCHAR,  topic\_id INTEGER REFERENCES topics (id) ON DELETE CASCADE,  user\_id INTEGER REFERENCES users (id) ON DELETE SET NULL,  --to find latest posts and latest posts by a given user:  post\_date DATE,  --either URL or text:  CONSTRAINT url\_or\_text CHECK (url IS NULL OR text\_content IS NULL),  --preventing spaces as title:  CONSTRAINT valid\_title CHECK (LENGTH(TRIM(title))>0) );  CREATE INDEX user\_posts\_index ON posts (user\_id); --speed up finding posts by a certain user CREATE INDEX post\_date\_index ON posts (post\_date); --speed up finding recent posts CREATE INDEX post\_url\_index ON posts (url); --speed up specific URL searches CREATE INDEX post\_user\_date\_index ON posts (user\_id, post\_date); -- speed up searches for latest posts by a specific user  CREATE INDEX post\_topic\_date\_index ON posts (topic\_id, post\_date); -- speed up searches for most recent posts for a given topic   --INSERT INTO posts (url) VALUES ('text'); -- checking title requirement (checked) --INSERT INTO posts (title, url, text\_content) VALUES ('title','text', 'text'); --checking url\_or\_text constraint. (checked)  /\*Allow registered users to comment on existing posts: A comment's text content can't be empty. Contrary to the current linear comments, the new structure should allow comment threads at arbitrary levels. If a post gets deleted, all comments associated with it should be automatically deleted too. If the user who created the comment gets deleted, then the comment will remain, but it will become dissociated from that user. If a comment gets deleted, then all its descendants in the thread structure should be automatically deleted too.\*/  CREATE TABLE comments (  id SERIAL PRIMARY KEY,  comment\_text VARCHAR NOT NULL,  post\_id INTEGER NOT NULL REFERENCES posts ON DELETE CASCADE,  parent\_id INTEGER REFERENCES comments (id) ON DELETE CASCADE,  comment\_date DATE, --to find latest posts by a given user  --referenced https://knowledge.udacity.com/questions/285776  user\_id INTEGER REFERENCES users ON DELETE SET NULL );  CREATE INDEX comments\_post\_id ON comments (post\_id); --to speed up searches for top-level comments CREATE INDEX comments\_parent\_id ON comments (parent\_id);--to speed up searches for top-level comments and direct children comments CREATE INDEX comment\_date\_index ON comments (user\_id, comment\_date); --to speed up searches for recent comments by a given user  /\*Make sure that a given user can only vote once on a given post: Hint: you can store the (up/down) value of the vote as the values 1 and -1 respectively. If the user who cast a vote gets deleted, then all their votes will remain, but will become dissociated from the user. If a post gets deleted, then all the votes for that post should be automatically deleted too.\*/  CREATE TABLE votes (  id SERIAL,  user\_id INTEGER REFERENCES users ON DELETE SET NULL,  post\_id INTEGER REFERENCES posts ON DELETE CASCADE,  vote SMALLINT NOT NULL,  CONSTRAINT valid\_vote CHECK (vote = 1 OR vote = -1),  PRIMARY KEY (id, user\_id) -- only one vote per user per post ); |

## 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:

|  |
| --- |
| /\*MIGRATING DATA \*/  /\*Topic descriptions can all be empty 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 You can use the Postgres string function regexp\_split\_to\_table to unwind the comma-separated votes values into separate rows Don't forget that some users only vote or comment, and haven't created any posts. You'll have to create those users too. The order of your migrations matter! For example, since posts depend on users and topics, you'll have to migrate the latter first. 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. 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.\*/  /\*user table will require a join of all users who created posts or commented\*/ INSERT INTO users (username) SELECT username FROM bad\_posts UNION SELECT regexp\_split\_to\_table(upvotes,',') FROM bad\_posts UNION SELECT regexp\_split\_to\_table(downvotes, ',') FROM bad\_posts UNION SELECT username FROM bad\_comments;   /\*topics table, no descriptions\*/ INSERT INTO topics (name) SELECT DISTINCT topic FROM bad\_posts;  /\*posts table\*/ INSERT INTO posts (id, title, url, text\_content, topic\_id, user\_id) SELECT bad\_posts.id, bad\_posts.title, bad\_posts.url, bad\_posts.text\_content, topics.id as topic\_id, users.id as user\_id FROM bad\_posts JOIN topics ON bad\_posts.topic = topics.name JOIN users ON bad\_posts.username = users.username;  /\*comments table\*/ INSERT INTO comments (id, comment\_text, post\_id, user\_id) SELECT bad\_comments.id, bad\_comments.text\_content, bad\_comments.post\_id, users.id FROM bad\_comments JOIN users ON bad\_comments.username = users.username;   /\*votes table\*/ INSERT INTO votes (user\_id, post\_id, vote)  SELECT users.id as user\_id, posts.id as post\_id, -1 AS down\_vote FROM (SELECT REGEXP\_SPLIT\_TO\_TABLE(downvotes, ',') AS username FROM bad\_posts ) t1 JOIN users ON users.username = t1.username JOIN posts ON posts.user\_id = users.id  UNION  SELECT users.id as user\_id, posts.id as post\_id, 1 AS down\_vote FROM (SELECT REGEXP\_SPLIT\_TO\_TABLE(upvotes, ',') AS username FROM bad\_posts ) t1 JOIN users ON users.username = t1.username JOIN posts ON posts.user\_id = users.id --referenced https://knowledge.udacity.com/questions/425750  /\*taking a final look at the migrated data\*/ SELECT \* FROM votes LIMIT 10; SELECT \* FROM comments LIMIT 10; SELECT \* FROM topics LIMIT 10; SELECT \* FROM users LIMIT 10; SELECT \* FROM posts LIMIT 10; |