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 is a username column in both bad\_comments and bad\_posts tables and also usernames have been used in upvotes and downvotes columns in the bad\_comments table. User is an independent entity and based on the second normal form, it should be separated, and a new table, users, created separately, then instead of using the username, use the user's ID in the bad\_comments table and bad\_posts table to decrease disk space consumption and also decrease error probability during data entry.   * Creating a new table of users from all usernames * Using the user\_id column instead of the username column in the bad\_posts table * Using the user\_id column instead of the username column in the bad\_comments table * Removing the upvotes column from the bad\_posts table * Removing the downvotes column from the bad\_posts table * Creating a new table of votes, to retain the user\_ids with their up and down votes   2- Based on the first normal form, goal #2 (single value in a cell) and goal #3 (no repeating columns), using a list of usernames or even user IDs in upvotes and downvotes columns is sort of denormalized data, and we should separate them. It's better to use a votes table that has post\_id and user\_id columns instead and remove the two upvotes and downvotes columns from the bad\_posts table.   * Creating a new table of topics * Using the topic\_id column instead of the topic column in the bad\_posts table   3- We have 89 distinct values in the Topic column of the bad\_posts table, based on the second normal form, it's better to have a topics table separately, and instead of using the topic's name, use the topic's id in comments table to decrease disk space consumption and also decrease error probability during data entering. |

## 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:

|  |
| --- |
| -- Creates a database schema for Udiddit, a social news aggregator.  ---------------------------------------------------------------  -- Creates table users with its constraints and indexes.  CREATE TABLE users (      id SERIAL PRIMARY KEY,      username VARCHAR(25) NOT NULL,      user\_password VARCHAR(50) DEFAULT NULL,      last\_login\_ts TIMESTAMP,        CONSTRAINT "unique\_users\_username" UNIQUE ("username"),      CONSTRAINT "check\_users\_username" CHECK (LENGTH(TRIM("username")) > 0 )  );  CREATE INDEX "idx\_users\_login\_date" ON users ("last\_login\_ts");  ---------------------------------------------------------------  -- Creates table topics with its constraints and indexes.  CREATE TABLE topics (      id SMALLSERIAL PRIMARY KEY,      topic\_name VARCHAR(30) NOT NULL,      description VARCHAR(500) DEFAULT NULL,      created\_user\_id INTEGER,      created\_ts TIMESTAMP,        CONSTRAINT "unique\_topics\_name" UNIQUE ("topic\_name"),      CONSTRAINT "check\_topics\_name" CHECK (LENGTH(TRIM("topic\_name")) > 0 ),      CONSTRAINT "fk\_topics\_user\_id"        FOREIGN KEY ("created\_user\_id") REFERENCES "users" ("id")        ON DELETE SET NULL  );  CREATE UNIQUE INDEX  "idx\_topics\_topic\_name" ON topics (LOWER("topic\_name"));  ---------------------------------------------------------------  -- Creates table posts with its constraints and indexes.  CREATE TABLE posts (      id BIGSERIAL PRIMARY KEY,      topic\_id SMALLINT NOT NULL,      user\_id INTEGER,      title VARCHAR(100) NOT NULL,      url VARCHAR(4000),      text\_content TEXT,      posted\_ts TIMESTAMP,        CONSTRAINT "fk\_posts\_user\_id"        FOREIGN KEY ("user\_id") REFERENCES "users" ("id")        ON DELETE SET NULL,      CONSTRAINT "fk\_posts\_topic\_id"        FOREIGN KEY ("topic\_id") REFERENCES "topics" ("id")        ON DELETE CASCADE,      CONSTRAINT "check\_posts\_title" CHECK (LENGTH("title") > 0 ),      CONSTRAINT "check\_posts\_xor\_url\_content"        CHECK ((url IS NOT NULL AND text\_content is NULL) OR               (url IS NULL AND text\_content is NOT NULL))  );  CREATE INDEX "idx\_posts\_user\_posted" ON posts ("user\_id","posted\_ts");  CREATE INDEX "idx\_posts\_topic\_posted" ON posts ("topic\_id","posted\_ts");  CREATE INDEX "idx\_posts\_url" ON posts ("url");  ---------------------------------------------------------------  -- Creates table all\_comments with its constraints and indexes.  CREATE TABLE all\_comments (      id BIGSERIAL PRIMARY KEY,      user\_id INTEGER,      post\_id BIGINT NOT NULL,      parent\_comment\_id BIGINT,      commented\_ts TIMESTAMP,      text\_content TEXT NOT NULL,        CONSTRAINT "fk\_all\_comments\_user\_id"        FOREIGN KEY ("user\_id") REFERENCES "users" ("id")        ON DELETE SET NULL,      CONSTRAINT "fk\_all\_comments\_post\_id"        FOREIGN KEY ("post\_id") REFERENCES "posts" ("id")        ON DELETE CASCADE,      CONSTRAINT "fk\_all\_comments\_comment\_id"        FOREIGN KEY ("parent\_comment\_id") REFERENCES "all\_comments" ("id")        ON DELETE CASCADE,      CONSTRAINT "check\_all\_comments\_content"        CHECK (LENGTH(TRIM("text\_content")) > 0 )  );  CREATE INDEX "idx\_all\_comments\_post\_parent\_comment"    ON all\_comments ("post\_id","parent\_comment\_id");  CREATE INDEX "idx\_all\_comments\_parent\_comment"    ON all\_comments ("parent\_comment\_id");  CREATE INDEX "idx\_all\_comments\_user\_ts"    ON all\_comments ("user\_id","commented\_ts");  ---------------------------------------------------------------  -- Creates table votes with its constraints and indexes.  CREATE TABLE votes (      id BIGSERIAL PRIMARY KEY,      user\_id INTEGER,      post\_id BIGINT NOT NULL,      up\_down SMALLINT NOT NULL,        CONSTRAINT "fk\_votes\_user\_id"      FOREIGN KEY ("user\_id") REFERENCES "users" ("id")      ON DELETE SET NULL,      CONSTRAINT "fk\_votes\_post\_id"      FOREIGN KEY ("post\_id") REFERENCES "posts" ("id")      ON DELETE CASCADE,      CONSTRAINT "unique\_votes\_user\_post" UNIQUE ("user\_id", post\_id),      CONSTRAINT "check\_votes\_up\_down" CHECK (up\_down in (1,-1))  );  CREATE INDEX "idx\_votes\_post\_up\_down" ON votes ("post\_id","up\_down"); |

## 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 a transaction  BEGIN;  -- Extracts usernames from the tables bad\_posts and bad\_comments and  -- inserts them into a new table, users, with assigning new IDs.  INSERT INTO users (username)  SELECT DISTINCT uu.username  FROM (      SELECT REGEXP\_SPLIT\_TO\_TABLE(bp.upvotes,',') username FROM bad\_posts bp      UNION ALL      SELECT REGEXP\_SPLIT\_TO\_TABLE(bp.downvotes,',') username FROM bad\_posts bp      UNION ALL      SELECT bp.username username FROM bad\_posts bp      UNION ALL      SELECT bc.username username FROM bad\_comments bc     ) uu;  ---------------------------------------------------------------  -- Extracts topics from the bad\_posts table and  -- inserts them into a new table, topics, with assigning new IDs.  INSERT INTO topics (topic\_name)  SELECT DISTINCT bp.topic FROM bad\_posts bp ORDER BY bp.topic;  ---------------------------------------------------------------  -- Extracts posts from the bad\_posts table and  -- inserts them into a new table, posts, with using the old IDs.  INSERT INTO posts (id, topic\_id, user\_id, title, url, text\_content)  SELECT bp.id post\_id, top.id topic\_id, u.id user\_id,         SUBSTRING(bp.title,1,100) title,         bp.url, bp.text\_content  FROM bad\_posts bp  JOIN topics top ON bp.topic = top.topic\_name  JOIN users u ON bp.username = u.username;  ---------------------------------------------------------------  -- Extracts posts from the bad\_comments table and  -- inserts them into a new table, all\_comments, with using the old IDs.  INSERT INTO all\_comments (id, user\_id, post\_id, text\_content)  SELECT bc.id, u.id user\_id, bc.post\_id, bc.text\_content  FROM bad\_comments bc  JOIN users u ON bc.username = u.username;  ---------------------------------------------------------------  -- Extracts usernames from the upvotes column of the bad\_posts table and  -- inserts equivalent user\_ids with vote 1 as the new rows  -- into a new table, votes.  INSERT INTO votes (user\_id, post\_id, up\_down)  SELECT u.id user\_id, upu.post\_id, 1  FROM      (SELECT bp.id post\_id, REGEXP\_SPLIT\_TO\_TABLE(bp.upvotes,',') upvote\_username       FROM bad\_posts bp      ) upu  JOIN users u    ON upu.upvote\_username = u.username;  ---------------------------------------------------------------  -- Extracts usernames from the downvotes column of the bad\_posts table and  -- inserts equivalent user\_ids with vote -1 as the new rows  -- into a new table, votes.  INSERT INTO votes (user\_id, post\_id, up\_down)  SELECT u.id user\_id, dou.post\_id, -1  FROM      (SELECT bp.id post\_id,              REGEXP\_SPLIT\_TO\_TABLE(bp.downvotes,',') downvote\_username       FROM bad\_posts bp      ) dou  JOIN users u    ON dou.downvote\_username = u.username;  ---------------------------------------------------------------  -- Changes the next value for sequence of posts table  -- because we used old IDs.  SELECT setval(pg\_get\_serial\_sequence('posts', 'id'), 50000, true);  ---------------------------------------------------------------  -- Changes the next value for sequence of all\_comments table  -- because we used old IDs.  SELECT setval(pg\_get\_serial\_sequence('all\_comments', 'id'), 100000, true);  ---------------------------------------------------------------  -- Commit the transaction  COMMIT;  ---------------------------------------------------------------  SELECT COUNT(\*) FROM users; -- 11077  SELECT COUNT(\*) FROM topics; -- 89  SELECT COUNT(\*) FROM posts; -- 50000  SELECT COUNT(\*) FROM bad\_posts; --50000  SELECT COUNT(\*) FROM all\_comments; -- 100000  SELECT COUNT(\*) FROM bad\_comments; -- 100000  SELECT COUNT(\*) FROM votes; -- 499710 |