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!

|  |
| --- |
| **bad\_posts**   1. [url] column has VARCHAR(4000), looking at the data length vary from 13 to 24 character, I’d suggest VARCHAR (100) to give 4x more threshold for corner cases. Query I’ve uses to check that   SELECT url, LENGTH(url)  FROM bad\_posts  WHERE url IS NOT NULL  ORDER BY LENGTH(url) DESC  LIMIT 10;   1. [upvotes] and [downvotes] consist of comma delimited strings, this prevent clear association between post id and up or down voter. This also breaks First Normal Form due to multiple values in one column (atomicity). Solution here would be set up two additionam upvote and downvote tables with FOREIGN KEY Id pointing to PRIMARY KEY in bad\_posts (id). Also in new tables TYPE can be change to VARCHAR with sensible limit. TEXT indicates we could expect a large text blob. 2. [id] because comments in bad\_comments table shouldn’t exist without posts, there should be set a FOREIGN KEY on [id] REFERENCES bad\_comments [post\_id] and add clause ON DELETE CASCADE to prevent ‘ghost’ comments without posts. 3. [topic] in my opinion should have UNIQUE INDEX. Firstly it’d let for faster topic based search and same ensures uniqueness of topic which may be useful for further analysis.   **bad\_comments**   1. [post\_id] type is stated as BIGINT, this value takes 8 bytes of storage and make possible to numerate up to ridiculous high number. I believe In this case we can safely use INTEGER or better SMALLINT. It’s very rare likely we have over 32k comments under one post.   **both tables**   1. [username] should not be NULL if only business rules will state we should not have anonymous posts and/or comments. 2. [text\_content] should have CONSTRAINT that prevents empty inputs, which would be from business perspective useless. Possibly CHECK CONSTRAINT may be of use here. 3. [username] could be exported to separate table with assigned id. 4. There is a lack of timestamp for user inputs in categories like user login, posts and comments. It may be useful telemetry statistic. Solution would be to add couple of TIMESTAMP flags for appropriate tables. |

## 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:

--DDL Creating Tables, Constraints and Indexes

--NOTE: execution in psql:

--1. TABLE creation

--data migration (Part 3)

--2. ALTER TABLE additional constraints

--3. INDEXES

----1. TABLE creation

--USERS

--last \_login\_date it's for new users sake when first created account will leave current date

CREATE TABLE users (

    id SERIAL

        CONSTRAINT pk\_users PRIMARY KEY,

    username VARCHAR(25) NOT NULL

        CONSTRAINT unique\_user UNIQUE,

    last\_login\_date DATE NOT NULL DEFAULT CURRENT\_DATE

);

--TOPICS

--for topic\_timestamp using timestamp without time zone because given business context i think it's not required

CREATE TABLE topics (

    id SERIAL

        CONSTRAINT pk\_topics PRIMARY KEY,

    topic\_name VARCHAR(30) NOT NULL,

    topic\_descr VARCHAR(500),

    topic\_timestamp TIMESTAMP

);

--POSTS

CREATE TABLE posts (

    id SERIAL

        CONSTRAINT pk\_posts PRIMARY KEY,

    topic\_id INTEGER NOT NULL,

    user\_id INTEGER,

    post\_title VARCHAR(100) NOT NULL,

    post\_url VARCHAR(100),

    post\_text TEXT,

    post\_timestamp TIMESTAMP

);

--COMMENTS

--for parent\_id rule would be if parent\_id = NULL then it's main comment, rest is cascading freeley from children to parents

CREATE TABLE comments (

    id SERIAL

        CONSTRAINT pk\_comments PRIMARY KEY,

    parent\_id INTEGER,

    user\_id INTEGER,

    post\_id INTEGER,

    comment\_text TEXT NOT NULL,

    comment\_timestamp TIMESTAMP

);

--VOTES

--1 for upvote and -1 for downvote

CREATE TABLE votes (

    id SERIAL

        CONSTRAINT pk\_votes PRIMARY KEY,

    post\_id INTEGER,

    user\_id INTEGER,

    vote SMALLINT

        CONSTRAINT vote\_up\_down CHECK (vote = 1 OR vote = -1),

    vote\_timestamp TIMESTAMP

);

----2. ALTER TABLE additional constraints after migration

--USERS

ALTER TABLE users

    ADD CONSTRAINT username\_not\_empty CHECK (LENGTH(TRIM(username))>0);

--TOPICS

ALTER TABLE topics

    ADD CONSTRAINT topic\_not\_empty CHECK (LENGTH(TRIM(topic\_name))>0);

--POSTS

--check\_one\_value: here we ensure only one of columns can be populated

ALTER TABLE posts

    ADD CONSTRAINT chck\_one\_value CHECK (

        (post\_url IS NULL AND post\_text IS NOT NULL)

        OR

        (post\_text IS NULL AND post\_url IS NOT NULL)

         ),

    ADD CONSTRAINT fk\_posts\_users FOREIGN KEY (user\_id) REFERENCES users(id)

        ON DELETE SET NULL,

    ADD CONSTRAINT fk\_posts\_topics FOREIGN KEY (topic\_id) REFERENCES topics(id)

        ON DELETE CASCADE,

    ADD CONSTRAINT post\_title\_not\_empty CHECK (LENGTH(TRIM(post\_title))>0);

--COMMENTS

ALTER TABLE comments

    ADD CONSTRAINT fk\_comments\_users FOREIGN KEY (user\_id) REFERENCES users(id)

        ON DELETE SET NULL,

    ADD CONSTRAINT comment\_not\_empty CHECK (LENGTH(TRIM(comment\_text))>0),

    ADD CONSTRAINT fk\_comments\_posts FOREIGN KEY (post\_id) REFERENCES posts(id)

        ON DELETE CASCADE,

    ADD CONSTRAINT fk\_comments\_parent FOREIGN KEY (parent\_id) REFERENCES comments(id)

        ON DELETE CASCADE;

--VOTES

ALTER TABLE votes

    ADD CONSTRAINT fk\_votes\_users FOREIGN KEY (user\_id) REFERENCES users(id)

        ON DELETE SET NULL,

    ADD CONSTRAINT unique\_vote\_id UNIQUE (id),

    ADD CONSTRAINT fk\_votes\_posts FOREIGN KEY (post\_id) REFERENCES posts(id)

        ON DELETE CASCADE;

----3. INDEXES after migration

--USERS

CREATE INDEX find\_username ON users(username);

CREATE INDEX find\_last\_login ON users(last\_login\_date);

--TOPICS

--additional conditions on find\_topic\_name will let incomplete, case insensitive search

CREATE INDEX find\_topic\_name ON topics(LOWER(topic\_name) VARCHAR\_PATTERN\_OPS);

--POSTS

CREATE INDEX find\_post\_by\_user ON posts(user\_id);

CREATE INDEX find\_post\_with\_url ON posts(post\_url);

CREATE INDEX find\_posts\_in\_topic ON posts(topic\_id);

--COMMENTS

CREATE INDEX find\_parents\_only ON comments(parent\_id) WHERE parent\_id IS NULL;

CREATE INDEX find\_comment\_by\_user ON comments(user\_id);

CREATE INDEX find\_all\_children ON comments(id) WHERE parent\_id IS NOT NULL;

--VOTES

CREATE INDEX vote\_calc ON votes(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. 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:

--DML migrating data

/\*

CREATE TABLE bad\_posts (

    id SERIAL PRIMARY KEY, --redundant

    topic VARCHAR(50), --done in topics table

    username VARCHAR(50), --done 9984

    title VARCHAR(150), --post title done in posts table with topic id's assigned

    url VARCHAR(4000) DEFAULT NULL, -- post url

    text\_content TEXT DEFAULT NULL, --post text

    upvotes TEXT,--done users 249799

    downvotes TEXT --done users 249911

);

CREATE TABLE bad\_comments (

    id SERIAL PRIMARY KEY, -- redundant

    username VARCHAR(50), --distinct 100

    post\_id BIGINT, --migrated from new posts table

    text\_content TEXT -- migrated 100k

);

\*/

----NOTE: migration need to be done in folowing order:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*users migration\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

CREATE TABLE temp (

    username VARCHAR

);

INSERT INTO temp(username)

    SELECT DISTINCT username

    FROM bad\_comments;

INSERT INTO temp(username)

    SELECT DISTINCT username

    FROM bad\_posts;

INSERT INTO temp(username)

    SELECT DISTINCT(regexp\_split\_to\_table (upvotes, ','))

    FROM bad\_posts;

INSERT INTO temp(username)

    SELECT DISTINCT(regexp\_split\_to\_table (downvotes, ','))

    FROM bad\_posts;

INSERT INTO users(username)

    SELECT DISTINCT username

    FROM temp;

DROP TABLE temp;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*topic migration\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

INSERT INTO topics(topic\_name)

    SELECT DISTINCT topic FROM bad\_posts;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*posts migration\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

INSERT INTO posts(

    user\_id,

    topic\_id,

    post\_title,

    post\_url,

    post\_text

    )

SELECT u.id,

    t.id,

    LEFT(bp.title, 100),

    bp.url,

    bp.text\_content

FROM bad\_posts AS bp

JOIN topics AS t ON t.topic\_name = bp.topic

JOIN users AS u ON bp.username = u.username;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*comments migration\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

INSERT INTO comments(

    user\_id,

    post\_id,

    comment\_text

    )

SELECT u.id,

    bc.post\_id,

    bc.text\_content

FROM bad\_comments AS bc

JOIN users AS u ON bc.username = u.username;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*votes migration\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

CREATE TABLE temp2(

    post\_id INTEGER,

    vote\_name VARCHAR,

    vote\_name\_id INTEGER,

    up\_down SMALLINT

);

INSERT INTO temp2(post\_id, vote\_name, vote\_name\_id, up\_down)

    SELECT

    p.id as post\_id,

    regexp\_split\_to\_table (bp.upvotes, ','),

    u.id as user\_id,

    1

    FROM bad\_posts AS bp

    JOIN users AS u

        ON bp.username = u.username

    JOIN posts AS p

        ON p.post\_title = bp.title;

INSERT INTO temp2(post\_id, vote\_name, vote\_name\_id, up\_down)

    SELECT

    p.id as post\_id,

    regexp\_split\_to\_table (bp.downvotes, ','),

    u.id as user\_id,

    -1

    FROM bad\_posts AS bp

    JOIN users AS u

        ON bp.username = u.username

    JOIN posts AS p

        ON p.post\_title = bp.title;

INSERT INTO votes (post\_id, user\_id, vote)

    SELECT post\_id, vote\_name\_id, up\_down FROM temp2;

DROP TABLE temp2;