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

```sql
SELECT url, LENGTH(url)
FROM bad_posts
WHERE url IS NOT NULL
ORDER BY LENGTH(url) DESC
LIMIT 10;
```

2. [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.
3. [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.
4. [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:
    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:

```
--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
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 --
for new users sake when first created account will leave current date
);
--TOPICS
CREATE TABLE topics (
    id SERIAL
        CONSTRAINT pk_topics PRIMARY KEY,
    post_id INTEGER,
    topic_name VARCHAR(30) NOT NULL,
    topic_descr VARCHAR(500),
    topic_timestamp TIMESTAMP --
 using timestamp without time zone because given business context i think it's no
t required
);
--POSTS
CREATE TABLE posts (
    id SERIAL
        CONSTRAINT pk_posts PRIMARY KEY,
    topic_id INTEGER NOT NULL,
    user_id INTEGER,
    post_title VARCHAR NOT NULL,
    post_url VARCHAR(100),
    post_text TEXT,
    post_timestamp TIMESTAMP
);
--COMMENTS
CREATE TABLE comments (
```

```
    id SERIAL
        CONSTRAINT pk_comments PRIMARY KEY,
    parent_id INTEGER, --
 rule would be if parent_id = NULL then it's main comment, rest is cascading free
ley from children to parents
    user_id INTEGER,
    post_id INTEGER,
    comment_text TEXT NOT NULL,
    comment_timestamp TIMESTAMP
);
--VOTES
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), --
1 for upvote and -1 for downvote
    vote_timestamp TIMESTAMP
);

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

--TOPICS
ALTER TABLE topics
    ADD CONSTRAINT fk_topics_post FOREIGN KEY (post_id) REFERENCES posts(id)
    ON DELETE CASCADE;

--POSTS
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)
         ), --here we ensure only one of columns can be populated
    ADD CONSTRAINT fk_posts_comments FOREIGN KEY (id) REFERENCES comments(id)
        ON DELETE CASCADE,
    ADD CONSTRAINT fk_posts_votes FOREIGN KEY (id) REFERENCES votes(id)
        ON DELETE CASCADE,
    ALTER COLUMN post_title TYPE VARCHAR(120),--
varchar limitlifted to 120 chars due to some original titles exceeds business lim
it of 100 chars, I want to remain original titles intact
    ADD CONSTRAINT fk_posts_users FOREIGN KEY (user_id) REFERENCES users(id)
        ON DELETE SET NULL;
```

```
--COMMENTS
ALTER TABLE comments
    ADD CONSTRAINT fk_comments_users FOREIGN KEY (user_id) REFERENCES users(id)
        ON DELETE SET NULL;

--VOTES
ALTER TABLE votes
    ADD CONSTRAINT fk_votes_users FOREIGN KEY (user_id) REFERENCES users(id)
        ON DELETE SET NULL;

----3. INDEXES after migration
--USERS
CREATE INDEX find_username ON users(username);
CREATE INDEX find_last_login ON users(last_login_date);

--TOPICS
CREATE INDEX find_topic_name ON topics(LOWER(topic_name) VARCHAR_PATTERN_OPS);--
it will let incomplete, case insensitive search
CREATE INDEX find_posts_in_topic ON topics(post_id);

--POSTS
CREATE INDEX find_post_by_user ON posts(user_id);
CREATE INDEX find_post_with_url ON posts(post_url);

--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 --9984
    FROM bad_comments;

INSERT INTO temp(username)
    SELECT DISTINCT username --100
    FROM bad_posts;

INSERT INTO temp(username)
    SELECT regexp_split_to_table (upvotes, ',') --249799
    FROM bad_posts;

INSERT INTO temp(username)
    SELECT regexp_split_to_table (downvotes, ',') --249911
    FROM bad_posts; -- 509794 not distinct values

INSERT INTO users(username)
    SELECT DISTINCT username --11077
    FROM temp;

DROP TABLE temp;

/************************topic migration #1**********************/

INSERT INTO topics(topic_name)
    SELECT topic FROM bad_posts;

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

INSERT INTO posts(topic_id, post_title, post_url, post_text, user_id)
    SELECT t.id, title, url, text_content, u.id FROM bad_posts AS bp
    JOIN topics AS t
        ON t.topic_name = bp.topic
    JOIN users AS u
    ON bp.username = u.username;



/************************topic migration #2**********************/

INSERT INTO topics(topic_name, post_id)
    SELECT bp.topic, p.id as post_id FROM bad_posts AS bp
```

```
    JOIN posts AS p
        ON bp.title = p.post_title;


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

INSERT INTO comments(comment_text, post_id, user_id) --
100k comments inputted for 10k distinct post id's
    SELECT bc.text_content, p.id, u.id FROM bad_comments AS bc
    JOIN bad_posts AS bp
        ON bc.post_id = bp.id
    JOIN posts AS p
        ON bp.title = p.post_title
    JOIN users AS u
        ON bp.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;
```