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!

To improve this schema, I recommend the following changes:

1. In order to eliminate partial dependencies and transitive dependencies to achieve 2NF and 3NF, respectively, these two tables should be broken up into 5 tables representing each entity: users, topics, posts, comments and votes each with its own id column as a BIGSERIAL PRIMARY KEY

2. In order to avoid having to make changes in the future to accommodate additional records, the id column in each table should be BIGSERIAL data type instead of SERIAL data type.

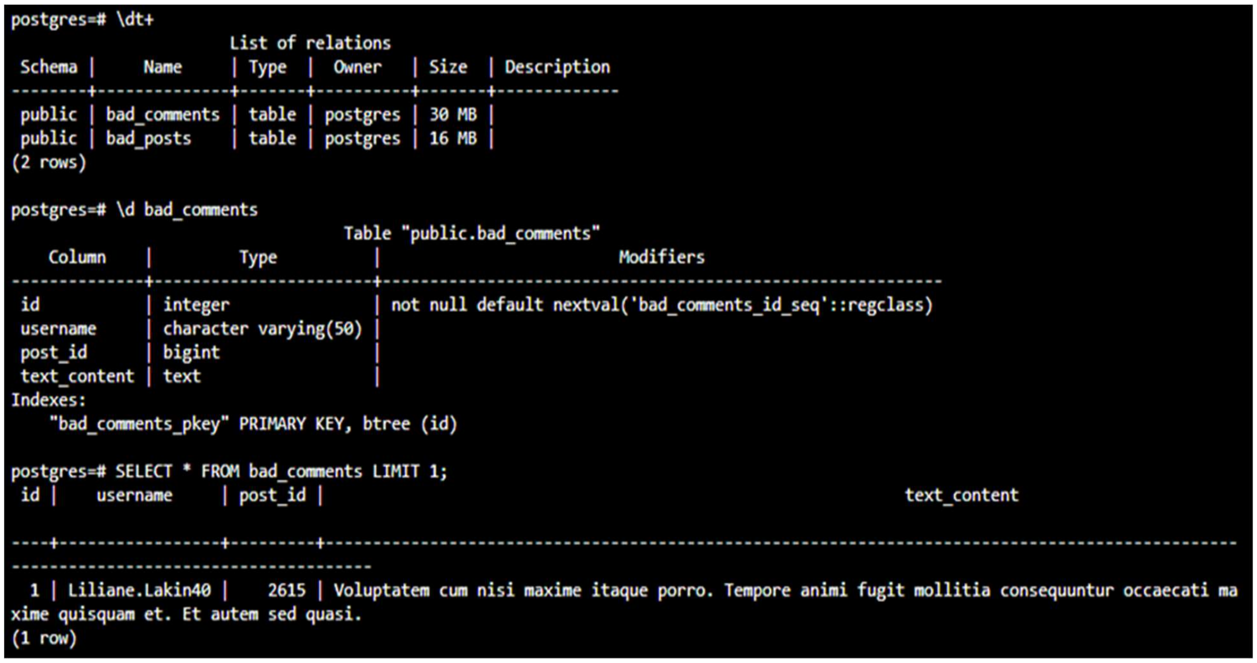
3. In order to enforce referential integrity between the tables and prevent invalid ids from being entered in dependent tables, a foreign key constraint should be added on the column “id” to establish the link between the id in the posts table and the post_id in the comments table. The same should be done for the tables created for each entity.

4. Vote columns should be integers to allow the counting of votes for each post.

5. UNIQUE constraints should be added on columns including “topic”, “username” and “title” to prevent duplication of these fields.

6. Timestamps should be added for the creation of posts and comments in order to be able to execute time-based queries and login dates for users as this will likely be information that will be used in analysis to make business decisions.

7. Indexes should be added to make queries more performant. The indexes should be added on columns that aren’t primary keys that will be executed frequently such as posts by a given user, posts by topic or the last time a user logged in.

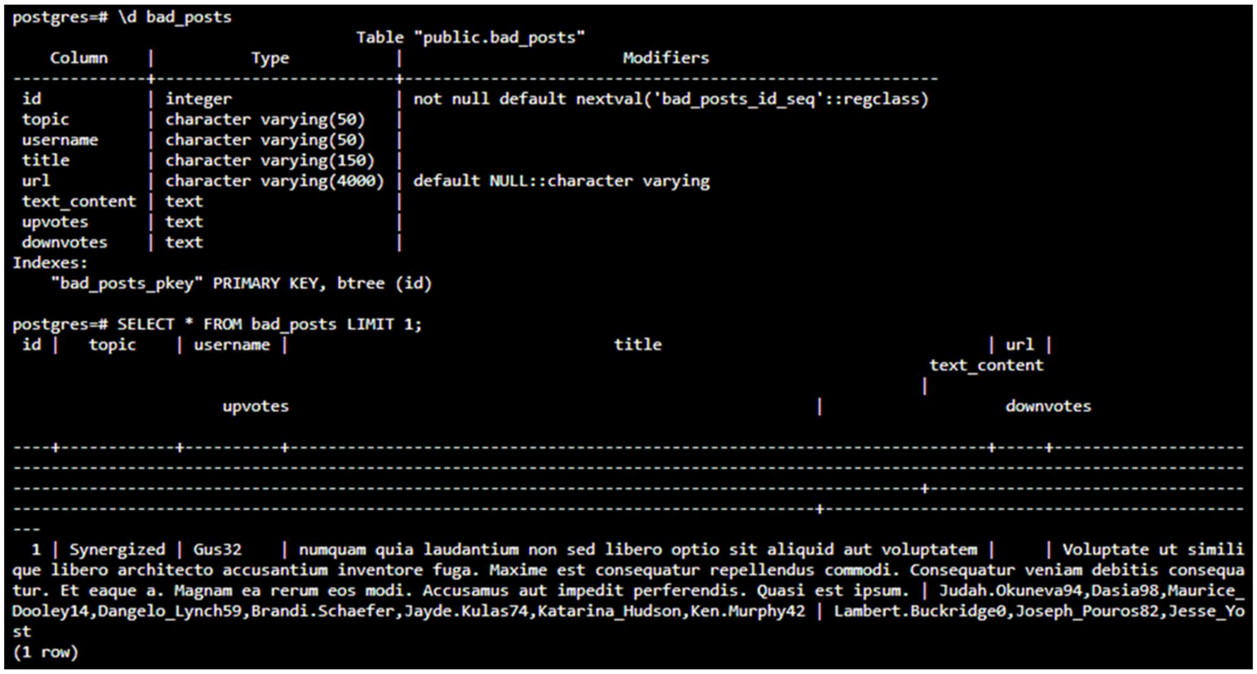


```
postgres=# \dt+
                       List of relations
 Schema |     Name     | Type  |  Owner   | Size  | Description
--------+--------------+-------+----------+-------+-------------
 public | bad_comments | table | postgres | 30 MB |
 public | bad_posts    | table | postgres | 16 MB |
(2 rows)

postgres=# \d bad_comments
                                   Table "public.bad_comments"
    Column    |         Type          |                         Modifiers
--------------+-----------------------+-----------------------------------------------------------
 id           | integer               | not null default nextval('bad_comments_id_seq'::regclass)
 username     | character varying(50) |
 post_id      | bigint                |
 text_content | text                  |
Indexes:
    "bad_comments_pkey" PRIMARY KEY, btree (id)

postgres=# SELECT * FROM bad_comments LIMIT 1;
 id |    username     | post_id |                                              text_content

----+-----------------+---------+------------------------------------------------------------------------------------------------
------------------------------------
  1 | Liliane.Lakin40 |    2615 | Voluptatem cum nisi maxime itaque porro. Tempore animi fugit mollitia consequuntur occaecati ma
xime quisquam et. Et autem sed quasi.
(1 row)
```



```
postgres=# \d bad_posts
                                   Table "public.bad_posts"
    Column    |          Type           |                       Modifiers
--------------+-------------------------+--------------------------------------------------------
 id           | integer                 | not null default nextval('bad_posts_id_seq'::regclass)
 topic        | character varying(50)   |
 username     | character varying(50)   |
 title        | character varying(150)  |
 url          | character varying(4000) | default NULL::character varying
 text_content | text                    |
 upvotes      | text                    |
 downvotes    | text                    |
Indexes:
    "bad_posts_pkey" PRIMARY KEY, btree (id)

postgres=# SELECT * FROM bad_posts LIMIT 1;
 id |   topic    | username |                                  title                                  | url |
                                                                                            text_content
                                                                                           |
                      upvotes                                                                      |              downvotes

----+------------+----------+-------------------------------------------------------------------------+-----+------------------
-------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------+-------------------------------------
-----------------------------------------------------------------------------------------+------------------------------------
---
  1 | Synergized | Gus32    | numquam quia laudantium non sed libero optio sit aliquid aut voluptatem |     | Voluptate ut simili
que libero architecto accusantium inventore fuga. Maxime est consequatur repellendus commodi. Consequatur veniam debitis consequa
tur. Et eaque a. Magnam ea rerum eos modi. Accusamus aut impedit perferendis. Quasi est ipsum. | Judah.Okuneva94,Dasia98,Maurice_
Dooley14,Dangelo_Lynch59,Brandi.Schaefer,Jayde.Kulas74,Katarina_Hudson,Ken.Murphy42 | Lambert.Buckridge0,Joseph_Pouros82,Jesse_Yo
st
(1 row)
```

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:

```
CREATE SCHEMA new_udiddit;

SET search_path TO new_udiddit;

CREATE TABLE "users" (
  "id" BIGSERIAL,
  "username" VARCHAR(25) NOT NULL,
  "last_login" TIMESTAMP,
  CONSTRAINT "users_pk" PRIMARY KEY ("id"),
  CONSTRAINT "username_not_empty" CHECK (LENGTH(TRIM("username"))>0)
);

CREATE UNIQUE INDEX "lower_unique_username" ON "users" (LOWER("username"));
CREATE INDEX "last_login" ON "users" ("username", "last_login");
```

```sql
CREATE TABLE "topics" (
  "id" BIGSERIAL,
  "topic_name" VARCHAR(30) NOT NULL,
  "description" VARCHAR(500),
  CONSTRAINT "topics_pk" PRIMARY KEY ("id"),
  CONSTRAINT "topic_unique" UNIQUE ("topic_name"),
  CONSTRAINT "topic_name_not_empty" CHECK (LENGTH(TRIM("topic_name"))> 0)
);


CREATE TABLE "posts" (
  "id" BIGSERIAL,
  "topic_id" BIGINT NOT NULL,
  "user_id" BIGINT,
 "title" VARCHAR(100) NOT NULL,
  "url" VARCHAR(4000),
  "text_content" TEXT,
  "creation_date" TIMESTAMP,
  CONSTRAINT "posts_pk" PRIMARY KEY ("id"),
  CONSTRAINT "posts_users_fk" FOREIGN KEY ("user_id") REFERENCES "users" ("id") ON DELETE SET NULL,
  CONSTRAINT "posts_topics_fk" FOREIGN KEY ("topic_id") REFERENCES "topics" ("id") ON DELETE CASCADE,
  CONSTRAINT "title_not_empty" CHECK (LENGTH(TRIM("title")) >0),
   CONSTRAINT "url_or_text_content" CHECK (("url" IS NOT NULL AND "text_content" IS NULL) OR ("url" IS NULL
AND "text_content" IS NOT NULL))
);
CREATE INDEX "posts_by_given_user" ON "posts" ("user_id", "id");
CREATE INDEX "posts_by_url" ON "posts" ("url", "id");
CREATE INDEX "posts_by_topic" ON "posts" ("topic_id", "id");
CREATE INDEX "recent_user_posts" ON "posts" ("creation_date" DESC, "user_id");


CREATE TABLE "comments" (
  "id" BIGSERIAL,
  "post_id" BIGINT NOT NULL,
  "user_id" BIGINT,
  "text_content" TEXT NOT NULL,
  "parent_comment_id" INTEGER,
  "level" INTEGER,
  "creation_date" TIMESTAMP,
  CONSTRAINT "comments_pk" PRIMARY KEY ("id"),
  CONSTRAINT "users_comments_fk" FOREIGN KEY ("user_id") REFERENCES "users" ("id") ON DELETE SET
NULL,
  CONSTRAINT "posts_comments_fk" FOREIGN KEY ("post_id") REFERENCES "posts" ("id") ON DELETE CASCADE,
  CONSTRAINT "parent_comment_id_fk" FOREIGN KEY ("parent_comment_id") REFERENCES "comments" ("id")
ON DELETE CASCADE,
  CONSTRAINT "comment_not_empty" CHECK (LENGTH(TRIM("text_content")) >0)
);

CREATE INDEX "parent_child" ON "comments" ("parent_comment_id", "id");
CREATE INDEX "comments_by_given_user" ON "comments" ("user_id", "id");
CREATE INDEX "recent_comments_by_user" ON "comments" ("creation_date" DESC, "user_id");



CREATE TABLE "votes" (
  "id" BIGSERIAL,
  "post_id" BIGINT NOT NULL,
  "user_id" BIGINT,
  "vote" SMALLINT,
  CONSTRAINT "votes_pk" PRIMARY KEY ("id"),
  CONSTRAINT "users_votes_fk" FOREIGN KEY ("user_id") REFERENCES "users" ("id") ON DELETE SET NULL,
  CONSTRAINT "posts_votes_fk" FOREIGN KEY ("post_id") REFERENCES "posts" ("id") ON DELETE CASCADE,
  CONSTRAINT "one_vote_per_post_per_user" UNIQUE ("post_id", "user_id"),
  CONSTRAINT "valid_votes" CHECK (vote = 1 OR vote = -1)
);

CREATE INDEX "computing_post_score" ON "votes" ("post_id", "vote");
```

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 usernames for users who created posts on exisiting topics
INSERT INTO new_udiddit.users (username)
SELECT DISTINCT username FROM public.bad_posts
ON CONFLICT DO NOTHING;



--migrating usernames for users who commented on posts
INSERT INTO new_udiddit.users (username)
SELECT DISTINCT username FROM public.bad_comments
ON CONFLICT DO NOTHING;

--migrating usernames for users who upvoted or downvoted posts
INSERT INTO new_udiddit.users (username)
SELECT DISTINCT regexp_split_to_table("upvotes",',') FROM public.bad_posts

UNION

SELECT DISTINCT regexp_split_to_table("downvotes",',') FROM public.bad_posts
ON CONFLICT DO NOTHING;

--migrating topics
```

```sql
INSERT INTO new_udiddit.topics (topic_name)
SELECT DISTINCT topic
  FROM public.bad_posts;

--migrating data from posts table
INSERT INTO new_udiddit.posts ("id", "topic_id", "user_id", "title", "url", "text_content")
SELECT DISTINCT p.id, nt.id, nu.id, LEFT("title", 100), "url", "text_content"
                FROM public.bad_posts p
                         JOIN new_udiddit.topics nt
                           ON p.topic = nt.topic_name
                         JOIN new_udiddit.users nu
                           ON p.username = nu.username
        ON CONFLICT DO NOTHING;

-- migrating data from comments table
INSERT INTO new_udiddit.comments("post_id", "user_id", "text_content")
SELECT DISTINCT np.id, nu.id, t1.text_content
                FROM
                         (SELECT DISTINCT "post_id", "username", "text_content"
                                        FROM public.bad_comments) t1
                                                 JOIN new_udiddit.users nu
                                                   ON nu.username = t1.username
                                                 JOIN new_udiddit.posts np
                                                   ON np.id = t1.post_id
                      ON CONFLICT DO NOTHING;

--migrating votes
INSERT INTO "votes" ("post_id", "user_id", "vote")
SELECT DISTINCT np.id, nu.id, 1 AS vote
                FROM
                          (SELECT DISTINCT topic, title, regexp_split_to_table("upvotes",',') AS upvotes
                                         FROM public.bad_posts p) t1
                                                    JOIN new_udiddit.users nu
                                                      ON t1.upvotes = nu.username
                                                    JOIN public.bad_posts p
                                                      ON t1.upvotes = p.upvotes
                                                    JOIN new_udiddit.posts np
```

```
                                                ON t1.title = np.title


UNION
SELECT DISTINCT np.id, nu.id, -1 AS vote
 FROM
        (SELECT DISTINCT topic, title, regexp_split_to_table("downvotes",',') AS downvotes
                    FROM public.bad_posts p) t1
                            JOIN new_udiddit.users nu
                             ON t1.downvotes = nu.username
                            JOIN public.bad_posts p
                             ON t1.downvotes = p.downvotes
                            JOIN new_udiddit.posts np
                             ON t1.title = np.title
    ON CONFLICT DO NOTHING;
```

Part IV: (OPTIONAL) Suggestions to make your project stand out

- Outline more than three things that you find wrong with the initially provided, denormalized Udiddit schema
- Write clean and legible DDL with self-documenting table, column, constraint, and index names.
- Add a section containing all or some of the most challenging DQLs for the queries and features of Part 2

```
-- a. List all users who haven't logged in the last year.


SELECT t1.id,
        t1.username,
        t1.last_login
 FROM
        (SELECT u.id,
                u.username,
                u.last_login
          FROM new_udiddit.users AS u) t1
WHERE last_login < (CURRENT_DATE - 365)
 ORDER BY 1;
```

```sql
-- b. List all users who haven't created any post.

SELECT nu.id,
       nu.username
 FROM new_udiddit.users AS nu
WHERE nu.id NOT IN
            (SELECT DISTINCT np.user_id
                     FROM new_udiddit.posts AS np)
 ORDER BY 1;
-- f. List the latest 20 posts for a given topic.

WITH recent_posts_by_topic AS
                      (SELECT np.id AS post_id,
                              np.topic_id,
                              nt.topic_name,
                              np.user_id,
                              np.title,
                              np.url,
                              np.text_content,
                              np.creation_date
                         FROM new_udiddit.topics nt
                              JOIN new_udiddit.posts np
                                ON nt.id = np.topic_id)

SELECT rp.creation_date,
       rp.topic_id,
       rp.topic_name
 FROM recent_posts_by_topic AS rp
WHERE rp.topic_name = 'Synergized'
 ORDER BY 1 DESC
   LIMIT 20;



-- j. List all the direct children of a parent comment.

 WITH RECURSIVE t1 AS
```

```
                    (SELECT nc1.id,
                             nc1.user_id,
                             nc1.post_id,
                             nc1.text_content,
                             nc1.parent_comment_id,
                             nc1.level,
                             nc1.creation_date
                      FROM new_udiddit.comments nc1
                     WHERE nc1.id = 100001

                     UNION

                     SELECT nc2.id,
                             nc2.user_id,
                             nc2.post_id,
                             nc2.text_content,
                             nc2.parent_comment_id,
                             nc2.level,
                             nc2.creation_date
                      FROM new_udiddit.comments nc2
                           JOIN t1
                             ON t1.id = nc2.parent_comment_id)
SELECT *
  FROM t1;



- k. List the latest 20 comments made by a given user.

WITH recent_posts_by_user AS
                        ( SELECT nc.user_id,
                                 nu.username,
                                  nc.id,
                                  nc.text_content,
                                  nc.creation_date
                             FROM new_udiddit.comments nc
                                  JOIN new_udiddit.users nu
                                   ON nc.user_id = nu.id
```

```
                                    GROUP BY 1, 2, 3, 4, 5
                                     ORDER BY 5 DESC, 1)
SELECT rp.creation_date,
            rp.username,
            rp.text_content
  FROM recent_posts_by_user AS rp
 WHERE rp.username = 'Luz42'
 ORDER BY 1 DESC
    LIMIT 20;




- I. Compute the score of a post, defined as the difference between the number of upvotes and the number of
downvotes

SELECT t1.post_id,
            t1.title AS post_title,
            (t1.upvotes - t2.downvotes) AS score
 FROM
             (SELECT DISTINCT nv1.post_id,
                                    np1.title,
                                    COUNT(vote) AS upvotes
                           FROM new_udiddit.posts np1
                                     JOIN new_udiddit.votes nv1
                                       ON np1.id = nv1.post_id
                             WHERE vote = 1
                             GROUP BY 1, 2)  t1

  JOIN

              (SELECT DISTINCT nv2.post_id,
                                     np2.title,
                                     COUNT(vote) AS downvotes
                           FROM new_udiddit.posts np2
                                    JOIN new_udiddit.votes nv2
                                      ON np2.id = nv2.post_id
```

```
                                WHERE vote = -1
                                 GROUP BY 1, 2) t2
       ON t1.post_id = t2.post_id
GROUP BY 1, 2, 3
 ORDER BY 1;
-
```

- Using WITH and JSON_AGG, devise a query that will retrieve a JSON document with three levels of nested comments for a given post

```
WITH RECURSIVE t1 AS
                         (SELECT nc.id,
                                     nc.user_id,
                                     nc.post_id,
                                     nc.text_content,
                                     nc.parent_comment_id,
                                     nc.level,
                                     nc.creation_date
                            FROM new_udiddit.comments nc
                                     JOIN new_udiddit.posts np
                                       ON np.id = nc.post_id
                          WHERE np.id = 9987 AND nc.id = 100001

                          UNION


                          SELECT nc2.id,
                                     nc2.user_id,
                                     nc2.post_id,
                                     nc2.text_content,
                                     nc2.parent_comment_id,
                                     nc2.level,
                                     nc2.creation_date
                            FROM new_udiddit.comments nc2
                                     JOIN t1
                                       ON t1.id = nc2.parent_comment_id)
```

```
SELECT t1.post_id,
             t1.level,
             JSON_AGG(text_content) AS JSON_doc
  FROM t1
 WHERE LEVEL < 4
 GROUP BY 1, 2
 ORDER BY 2;
```