

BCS 1510: Integration of Database

Student Names: Nitu Cristian(i6350367), Untesu David (i6357705), Group Number: I forgot

Friday, May 29, 2024

Table of contents

1. Database Description.....	2
1.1 Introduction to the Database and its Usages.....	2
1.2 Database ERD and description.....	3
2. Database structure breakdown.....	5
2.1 Example of Each Table and Functional Dependency breakdown.....	5
2.1.1 `Login_data` table.....	5
2.1.1 `user` table.....	6
2.1.3 `followers` table.....	8
2.1.4 `post` table.....	10
2.1.5 `likes_post` table.....	12
2.1.6 `record_post_interaction` table.....	14
2.1.7 `comments` table.....	16
3. Justification for views proposed.....	18
3.1 Possible bots view.....	18
3.2 Above average following count.....	19
3.3 View Daily interactions and Above average daily interactions.....	20
4. Stored procedures, functions and triggers justification.....	21
4.1 GetUserProfile() : JSON.....	21
4.2 CheckCommentLength() : CHAR_LENGTH, BEFORE_COMMENT_INSERT.....	21
4.3 LikesPost PROCEDURE (TRANSACTION) , BEFORE_LIKE_INSERT trigger.....	22
4.4 before_follow_insert TRIGGER.....	22
5. Query performance on index.....	23
6. Profit maximisation queries (with reference to 20Queries->20Queries.sql dir).....	25
6.1 Select more than ? followers.....	26
6.2 Show the total number of posts made by each user via user_id.....	26
6.3 Find all comments made on a particular user's post.....	27
6.4 Display the top X most liked posts.....	28
6.5 Count the number of posts each user has liked.....	29
6.6 List all users who haven't made a post yet.....	29
6.7 List users who follow each other.....	30
6.8 Show the user with the highest number of posts.....	30
6.9 List the top X users with the most followers.....	31

6.10 Find posts that have been liked by all users.....	31
6.11 Display the most active user (based on posts, comments (7), and likes).....	31
6.12 Find the average number of likes per post for each user.....	32
6.13 Show posts that have more comments than likes.....	33
6.14 List the users who have liked every post of a specific user.....	34
6.15 Display the most popular post of each user (based on likes).....	34
6.16 Find the user(s) with the highest ratio of followers to following.....	35
6.17 Show the month with the highest number of posts made.....	37
6.18 Identify users who have not interacted with a specific user's posts.....	38
6.20 Find users who are followed by more than X% of the platform users.....	38
7. Appendix.....	40

1. Database Description

1.1 Introduction to the Database and its Usages

Views

The Quackstagram Database is a Database used by the Quackstagram application. The database comes packed with 4 views that may be used in data analytics. These views include viewing daily interactions on the platform and a view that uses the daily interaction table to detect above average interaction days by calculating the average daily interactions on the platform and selecting the ones bigger than average. Other views include detecting bot accounts that have a high ratio in following to followers and 0 posts and above average following count views to help CSS detect 'popular' accounts.

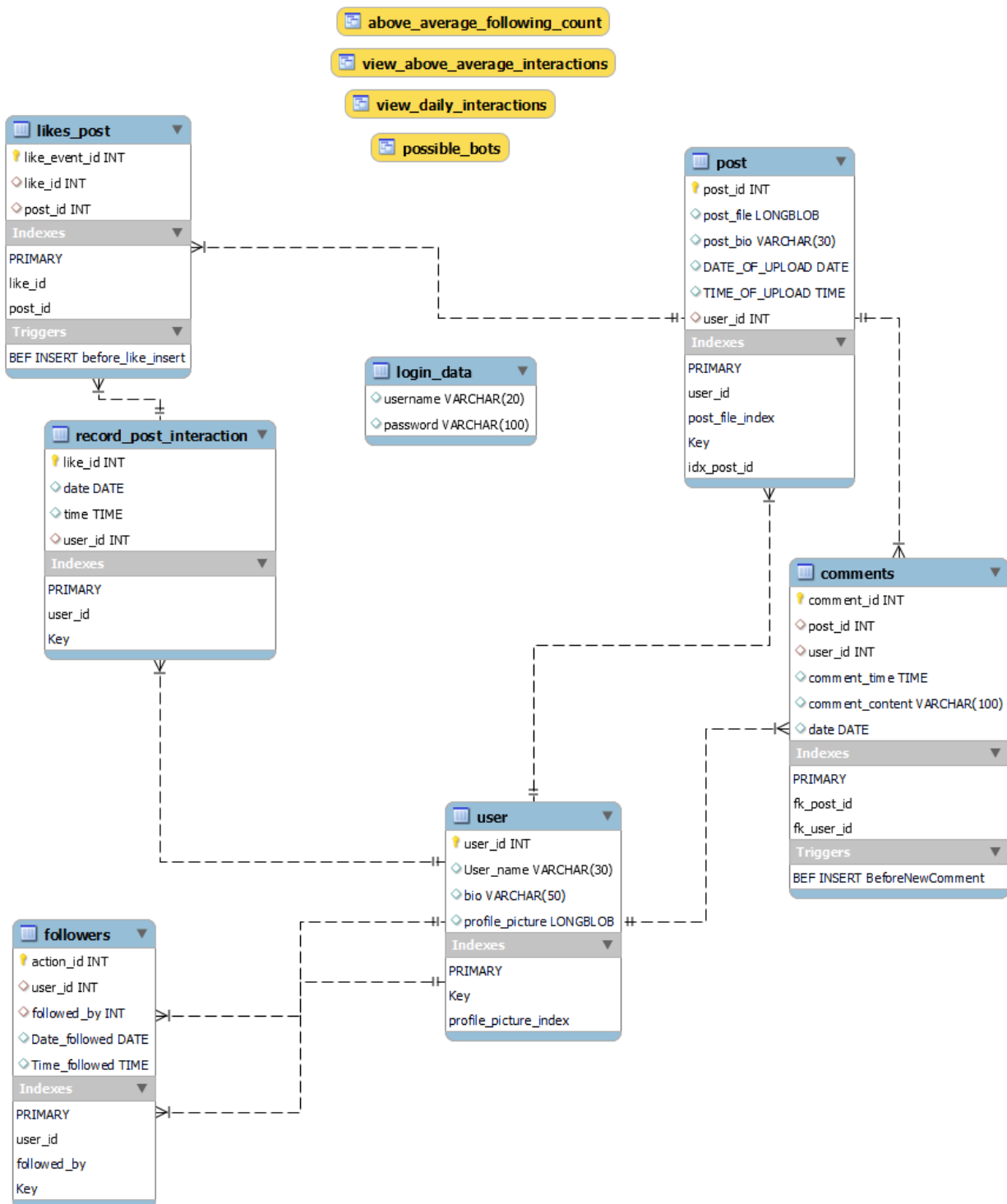
Integration and functionality

The Quackstagram application is now fully integrated with the database. The usages of the database includes storing hashed passwords, usernames, post data and interaction data. A hypothetical 'comments' table exists in the database which can be used by

developers in the latter development process of the application as currently the platform does not support comments. The database further supports post data including images and videos with the Binary Large Object data type which supports up to 4GB of media per tuple, however it is recommended that file compression happens on the application side. before releasing the final version of the application. User verification, update delete and insert queries happen using Prepared Statements which are more secure and frequented procedures such as liking a post are stored as stored procedures with triggers due to the complex nature of the operation. Furthermore the application fully uses identification numbers for each entity (users, posts, interaction, events) to optimise the read queries and to ensure $O(\log n)$ time complexity on searches by B- trees.

The application supports searching users (SearchExplore.java) by user names where the actual user has to be searched in the database with a LIKE sql function and the String username where it is the only place to which $O(n)$ time complexity is applied in the database as users have to be searched by String values. Modifications on the original version have been added such as to not break Design Principles such as the PostData.java class which loads data from the database and indexes on all IDs'.

1.2 Database ERD and description



This is a comprehensive Entity Relationship Diagram created using MySQL Workbench CE 8.0 See appendix for normalisation process (PART A)

An unrelated table login_data has been added for passwords and usernames used by the login and register processes in the application. The login_data stores hashed passwords using the apache java library. On login the application hashes the password and matches the digest with a pair of username and hashed_password.

The comments table is not integrated in the application however it still follows 3NF and BCNF form (If we don't consider an FD on time<-date).

Each entity in the database has a unique ID of type INT and it is AA as the application uses queries that perform JOINS on subqueries and therefore using IDs' ensures faster lookup times once the application has a lot of users. The comments table also uses a trigger to ensure the comment length isn't too large on Insert request.

The followers table has a M:M relation to users as users can follow each other, the action_id key is not necessarily used however it can be used in later development processes for auditing and tracking.

All user data such as posts, followers, following and likes per post are gathered from the database using COUNT and GROUP BY queries in order to ensure maximum data integrity as per the schema structure. Cascade ON DELETE methods were added for all foreign key references of user_id, post_id and like_id, which are used by all interaction tables, such that if a user/post/like is deleted, all its' children/identifying relations are removed for when CSS implements the UNFOLLOW option or DELETE POST/UNLIKE actions.

The multiple likes per post has also been fixed via the database side using a trigger and transaction type stored procedure to ensure that a user cannot like a post multiple times, and it is thus displayed in the application via error response code. Transaction type has been used as a procedure for this action due to the complex operations that needed and frequent and simultaneous use of the query from multiple users at once. The triggers are within bounds of trigger design principles. (maximum size of 32KB, Appendix A1)

Indexes are added by default on the LONGBLOB datatype. The BLOBS are parsed application side to byte arrays.

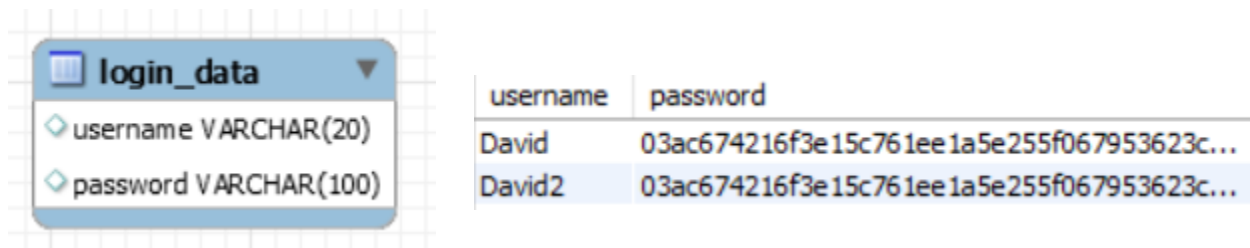
The database does not use TIMESTAMP for dates and times for data analytics purposes but also for more convenient parsing on the application layer between times of users from different time zones in future development processes.

2. Database structure breakdown

(The following section provides Rationale and a comprehensive breakdown of the database structure. Requirements 4 and 5 and 6 have been merged in 2.1)

2.1 Example of Each Table and Functional Dependency breakdown

2.1.1 `Login_data` table



username	password
David	03ac674216f3e15c761ee1a5e255f067953623c...
David2	03ac674216f3e15c761ee1a5e255f067953623c...

Fig. 1 Data stored

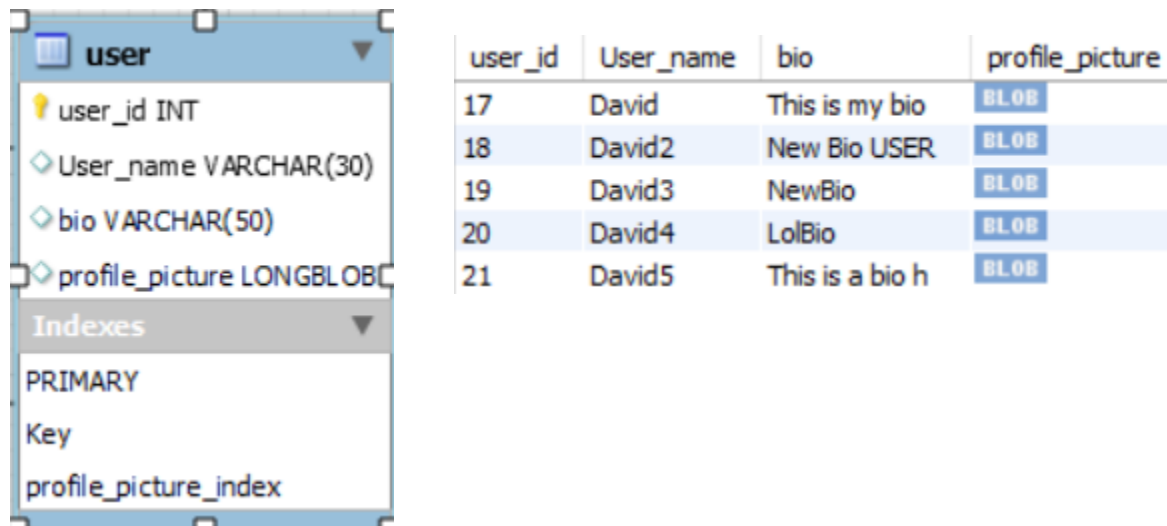
Login data table is a single independent table in the database used by login and sign up processes (Fig. 1). The table stores the digests of apache hashing functions (Fig. 2)

```
import org.apache.commons.codec.digest.DigestUtils;  
  
public class Hashing {  
    public static String hash(String password) {  
        return DigestUtils.sha256Hex(password);  
    }  
}
```

Fig. 2 apache Hash Function

The logic behind Sign In uses the hash function to check the digests match. On Match, the application retrieves the user with the user name and the application starts.

2.1.1 `user` table



user_id	User_name	bio	profile_picture
17	David	This is my bio	BLOB
18	David2	New Bio USER	BLOB
19	David3	NewBio	BLOB
20	David4	LolBio	BLOB
21	David5	This is a bio h	BLOB

Fig. 3 Data stored user

Data Types and functional dependencies (denoted by '<-'):

- *THE SET (U)*
- *user_id (A):* Primary key, AA, Unique, NN, INT
- *User_name (B):* VARCHAR(30), FD <- A
- *bio (C):* VARCHAR(50), FD <- A, DEFAULT NULL
- *profile_picture (D):* LONGBLOB (BLOB INDEX) FD <- A

Rationale:

The user table stores basic user information such as user name, profile picture and bio of the user. All dependencies are on user_id which is a unique Auto increment INT used to identify a specific user. All the data including user_id is loaded into the application as an object of type User identified by 'loggedUser' where all the queries in the application use the ID of the logged user to retrieve other information. The following count, like count and followers count are not included in this table as they would rely on triggers and stored procedures and complex logic which may deter data integrity due to the M:M relation of the mentioned data. Instead the follower table (discussed later) maps the M:M relation and on user login the data is counted and retrieved from the database. The Edit profile functionality can also be easily integrated by CSS with this design due to the FDs being on A so it supports UPDATE queries for any of the tuples without causing data integrity issues.

3NF PROOF (with reference to Data Types and functional dependencies):

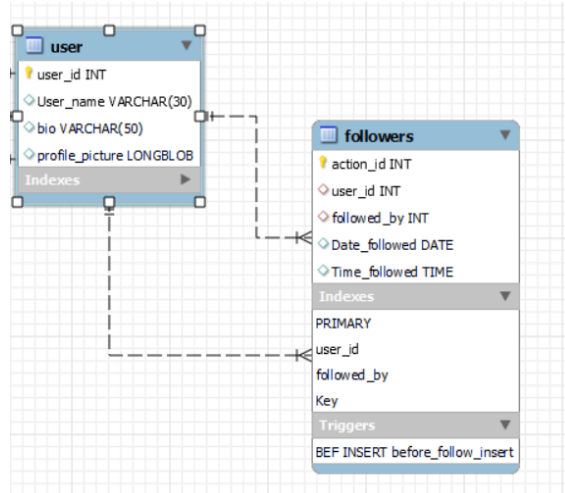
A Primary key (SUPER KEY)

$B \leftarrow A, C \leftarrow A, D \leftarrow A$

Thus, U : {A, B \leftarrow A, C \leftarrow A, D \leftarrow A}

- No transitive dependencies (3NF) All prime attributes on the right of ' \leftarrow '
- No Partial dependencies (2NF)
- Atomic data (1NF) Fig. 3

2.1.3 `followers` table



action_id	user_id	followed_by	Date_followed	Time_followed
1	18	19	2024-05-17	15:19:50
2	20	17	2024-05-22	14:05:59
3	19	18	2024-05-22	15:03:53
4	18	21	2024-05-22	15:07:17
5	19	21	2024-05-22	15:07:43
6	21	22	2024-05-22	15:19:46

Fig. 4 Data stored followers (In relation to `user`)

Data Types and functional dependencies (denoted by `<-`):

- *THE SET (F)*
- action_id (A): Primary key, AA, Unique, NN, INT
- user_id (B): INT FK, FD <- A, REF (U.user_id), NN
- followed_by (C): INT FK, FD <- A, NN, REF (U.user_id), NN
- Date_followed (D): DATE FD <- A
- Time_followed (T) : TIME FD <- A (CAN BE A FD - > D)

Rationale:

The followers table was created in a M:M (many-to-many) relation to the users' table. This is because users can follow each other and supposedly in future development processes can unfollow each other. The reason it is designed this way is its lack of reliability on a TRIGGER to enforce data integrity.

In this design, if users "unfollow", the statistics of the user are not affected and do not rely on a trigger to decrement the supposed statistics column that would be in the table `user` (e.g a following count, follower count column in the user table). It thus prevents synchronisation problems that may deter data integrity. A before insert trigger has still been implemented (even though this is handled on the application layer) to enforce unique

combinations of user_id and follower_id (follower_id referring to a user_id in users). Thus the user following and followers count are dynamically calculated by a COUNT and GROUP BY query due to the respective dynamic nature of the data THE SET. This is to ensure accuracy of data in a trade off for time-expensive queries. CSS can later implement caching methods in the quackstagram application to optimise efficiency.

3NF PROOF (with reference to Data Types and functional dependencies):

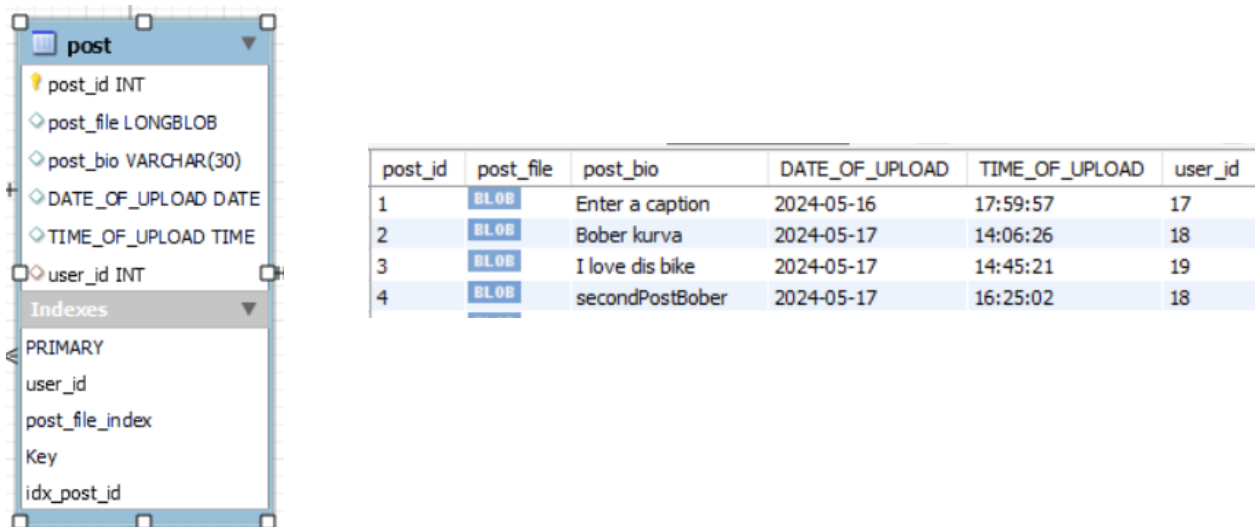
A Primary key (SUPER KEY)

$B \leftarrow A, C \leftarrow A, D \leftarrow A, D \leftarrow A, T \leftarrow A$

Thus, F : {A, B \leftarrow A, C \leftarrow A, D \leftarrow A, T \leftarrow A} OR {A, B \leftarrow A, C \leftarrow A, D \leftarrow A, D \leftarrow T, T \leftarrow A}

- Can be considered 3NF/BCNF if we say that T is not an FD on D (T \nrightarrow D)
- No Partial dependencies (2NF)
- Atomic data (1NF) Fig. 4

2.1.4 `post` table



post_id	post_file	post_bio	DATE_OF_UPLOAD	TIME_OF_UPLOAD	user_id
1	BLOB	Enter a caption	2024-05-16	17:59:57	17
2	BLOB	Bober kurva	2024-05-17	14:06:26	18
3	BLOB	I love dis bike	2024-05-17	14:45:21	19
4	BLOB	secondPostBober	2024-05-17	16:25:02	18

Fig. 5 Data stored post

Data Types and functional dependencies (denoted by '<-'):

- *THE SET (P)*
- post_id (A): Primary key, AA, Unique, NN, INT
- user_id (B): INT FK, FD <- A, REF (U.user_id), NN
- post_bio (E): varchar(30) , FD <- A
- post_file (C): LONGBLOB, FK <- A, DEFAULT NULL
- DATE_OF_UPLOAD (D): DATE FD <- A
- TIME_OF_UPLOAD (T) : TIME FD <- A (CAN BE A FD - > D)

Rationale:

Similar to the `user` entity table the `post` table in the database is a central table in the database used to identify a unique post in relation to a user and has an identifying relationship on other tables. The data type used for post_file is Binary Large Object which is used to store media files. Storing media in a database is not great practice, however CSS can use it until their platform grows and then take all the collected media and use web hosting services in the future for media. The post_id is the superkey identifying a unique post in the database, the application

mainly joins posts on post and user ids (example search explore). All unique ids in the database are indexed as they are usually never modified and used by read queries in JOIN operations. All functional dependencies being on the ID ensures no duplication of data..

3NF PROOF (with reference to Data Types and functional dependencies):

A Primary key (SUPER KEY)

$B \leftarrow A, C \leftarrow A, D \leftarrow A, D \leftarrow A, T \leftarrow A, E \leftarrow A$

Thus, P : {A, B \leftarrow A, C \leftarrow A, E \leftarrow A, D \leftarrow A, T \leftarrow A} OR {A, E \leftarrow A, B \leftarrow A, C \leftarrow A, D \leftarrow A, D \leftarrow T, T \leftarrow A}

- Can be considered 3NF/BCNF if we say that T is not an FD on D (T \nrightarrow D)
- No Partial dependencies (2NF)
- Atomic data (1NF) Fig. 5

2.1.5 `likes_post` table

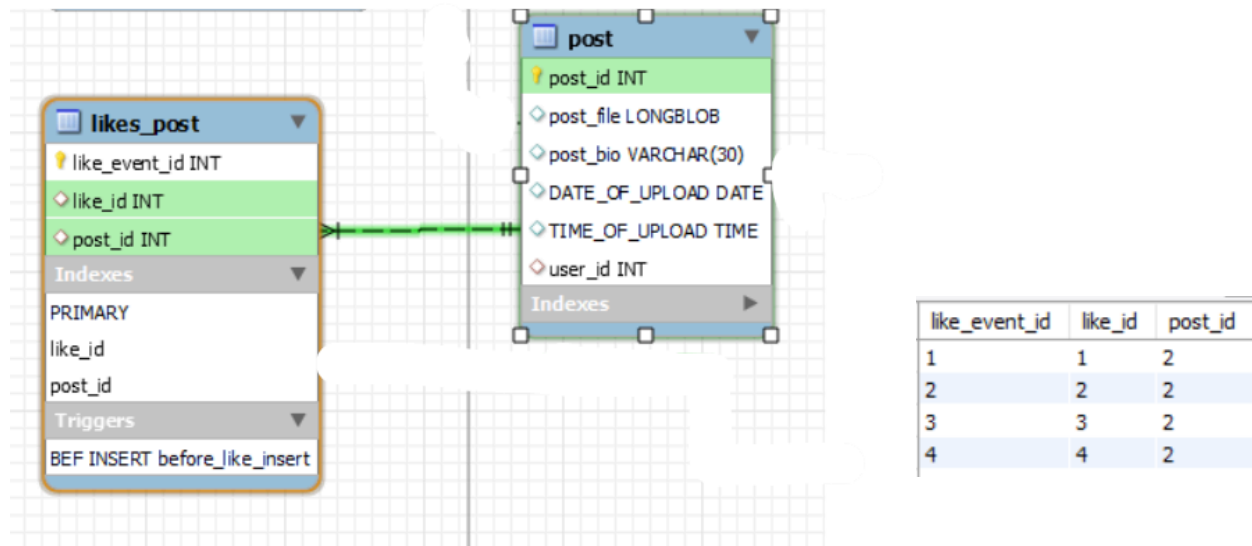


Fig. 6 Data stored likes_post

Data Types and functional dependencies (denoted by '<-'):

- *THE SET (LP)*
- *like Event id (A):* Primary key, AA, Unique, NN, INT
- *like_id (B):* INT FK, FD <- A, REF (RPI.like_id), NN
- *post_id (C):* INT FK, FD <- A REF(P.post_id), NN

Rationale:

The likes_post table represents the action of a user liking a post by like_id foreign key which has an identifying relationship in the table. All dependencies are on A instead of the candidate composite key {B,C} for later development processes where quackstagram may want to display which users liked the picture. For now the likes_post table has a 'before insert' trigger that checks for a unique combination of {B,C} to ensure data integrity for when such implementation may happen. The insert is also controlled by a Transaction stored procedure 'likePost' which uses record_post_interaction (RPI) to ensure maximum data integrity. The like_id FK pinpoints the date time and user_id of who liked the posts using the RPI table.

3NF PROOF (with reference to Data Types and functional dependencies):

A Primary key (SUPER KEY), {BC} (CANDIDATE COMPOSITE KEY)

A, B \rightarrow A C \rightarrow A, But can also be B \rightarrow C, C \rightarrow B, REMOVE A¹

Thus, LP : {A, BC \rightarrow A }

- No transitive dependencies (3NF) All prime attributes on the right of \rightarrow
- No Partial dependencies (2NF)
- Atomic data (1NF) Fig. 6

2.1.6 `record_post_interaction` table

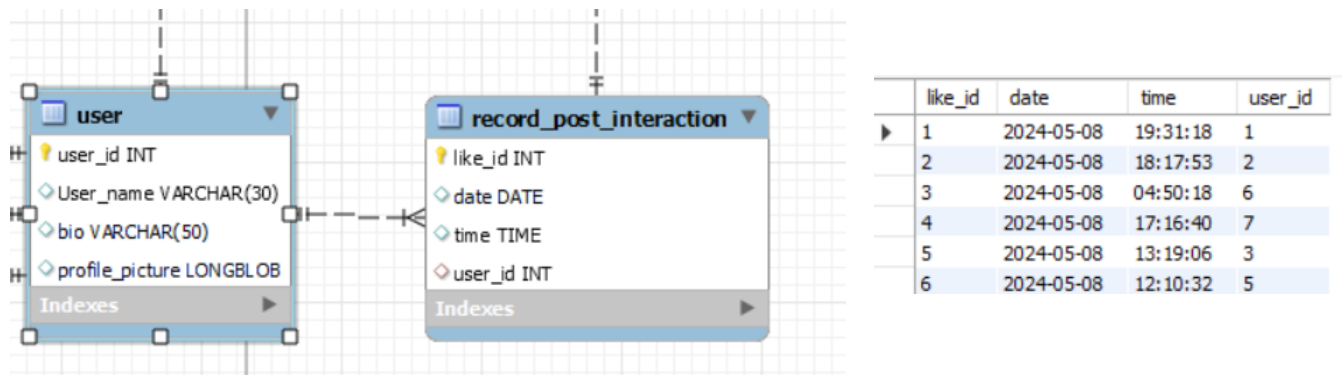


Fig. 7 Data stored record_post_interaction

Data Types and functional dependencies (denoted by ` \rightarrow `):

- *THE SET (RPI)*
- like_id (A) : Primary key, AA, Unique, NN, INT
- user_id (B) : INT FK, FD \rightarrow A, REF (U.user_id), NN
- DATE (C) : DATE, FD \rightarrow A DEF NULL
- TIME (D) : TIME, FD \rightarrow A DEF NULL

Rationale:

The record_post_interaction (RPI) table is an intermediate table that handles the action of liking a picture by a user and maps it to a certain post. The relationship between RPI and the user table is many-to-one as it is an identifying relationship from user to RPI. RPI has a Many to one relationship to likes_post and this table stores like_id FK to likes_post and data about when the like has happened through the date and time columns.

3NF PROOF (with reference to Data Types and functional dependencies):

A Primary key (SUPER KEY)

$A, B \leftarrow A \quad C \leftarrow A, D \leftarrow A$

Thus, RPI : $\{A, B \leftarrow A, C \leftarrow A, D \leftarrow A\}$

- No transitive dependencies (3NF) All prime attributes on the right of \leftarrow
- No Partial dependencies (2NF)
- Atomic data (1NF) Fig. 7

2.1.7 `comments` table

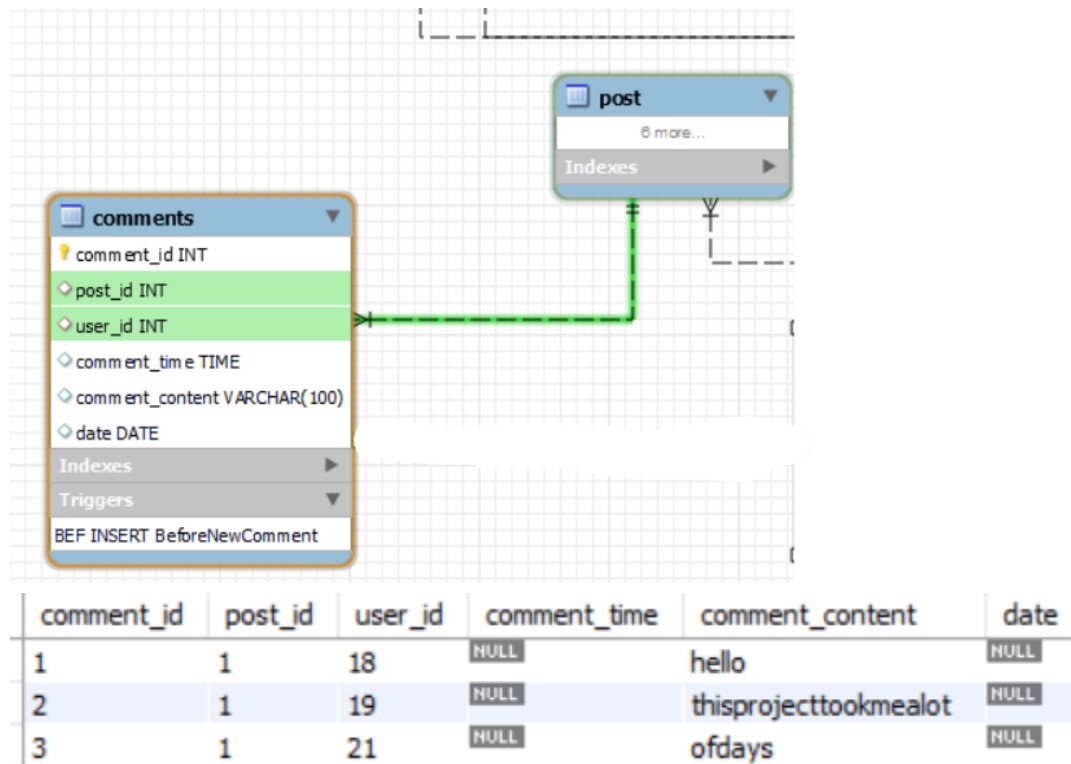


Fig. 8 Data stored comments

Data Types and functional dependencies (denoted by '<-'):

- *THE SET (C)*
- comment_id (A) : Primary key, AA, Unique, NN, INT
- user_id (B) : INT FK, FD <- A, REF (P.user_id), NN
- comment_time (C) : TIME, FD <- A DEF NULL
- date (D) : DATE, FD <- A DEF NULL
- post_id (E) : INT FK, FD <- A, REF (P.post_id), NN

Rationale:

The comments table is an intermediate table that handles the action of users commenting on a post and maps each comment to a specific post. The relationship between comments and the user table is many-to-one, as it is an identifying relationship from user to comments. Comments has a many-to-one relationship to post, storing the post_id FK to post and data about when the comment was made

through the date and time columns. The BEFORE INSERT trigger ensures proper handling before a new comment is recorded that checks the comment length. This THE SETup ensures each comment is uniquely associated with a specific user and post, providing clear traceability and data integrity. It is not implemented in quackstagram.

3NF PROOF (with reference to Data Types and functional dependencies):

A Primary key (SUPER KEY)

$A, B \leftarrow A, C \leftarrow A, D \leftarrow A$

Thus, LP : {A, B \leftarrow A, C \leftarrow A, D \leftarrow A }

- No transitive dependencies (3NF) All prime attributes on the right of \leftarrow
- No Partial dependencies (2NF)
- Atomic data (1NF) Fig. 7

3. Justification for views proposed

3.1 Possible bots view

```
CREATE VIEW `quackstagram`.`possible_bots` AS
SELECT
  `u`.`user_id` AS `user_id`,
  `u`.`User_name` AS `username`,
  COUNT(DISTINCT `f`.`followed_by`) AS `follower_count`,
  COUNT(DISTINCT `f2`.`user_id`) AS `following_count`
FROM
  (`quackstagram`.`user` `u`
  LEFT JOIN `quackstagram`.`followers` `f` ON (`u`.`user_id` = `f`.`user_id`)
  LEFT JOIN `quackstagram`.`followers` `f2` ON (`u`.`user_id` = `f2`.`followed_by`)
  LEFT JOIN `quackstagram`.`post` `p` ON (`u`.`user_id` = `p`.`user_id`))
GROUP BY
  `u`.`user_id`, `u`.`User_name`
HAVING
  (COUNT(DISTINCT `f2`.`user_id`) / COUNT(DISTINCT `f`.`followed_by`) > 7)
  AND (COUNT(`p`.`post_id`) = 0);
```

user_id	username	follower_count	following_count
3	Clari	1	9
13	Hayward	1	11
34	Anna-diana	1	15
158	Wilfred	1	8
161	Minna	1	17

Fig. 9 View output and QUERY

The possible bots view may be used as a table reference to detect possible bots in the application. The view groups user_ids and user names that have following to follower ratio more than 7 and 0 posts, these variables can of course change but it can be used by CSS to automatically restrict/delete accounts that meet this exceptional threshold. Detecting bottled accounts is crucial in a larger pool of users to prevent database stress.

3.2 Above average following count

```
CREATE VIEW above_average_following_count AS
SELECT
    u.user_id,
    u.User_name AS username,
    COUNT(f.followed_by) AS following_count
FROM
    user u
    LEFT JOIN
    followers f ON u.user_id = f.user_id
GROUP BY
    u.user_id, u.User_name
HAVING
    COUNT(f.followed_by) > (
        SELECT
            AVG(following_count)
        FROM (
            SELECT
                COUNT(followed_by) AS following_count
            FROM
                followers
            GROUP BY
                user_id
        ) AS avg_following
    );
```

	user_id	username	following_count
▶	1969	Zathin	441
	98	Giordano	20
	107	Dale	20

Fig. 10 View output and QUERY

The above average following count table is a view that calculates the average followers count for every user on the platform and only selects users that surpass that number. This can be useful for CSS for detecting `popular` or `celebrity` status accounts in analysing their behaviour.

3.3 View Daily interactions and Above average daily interactions

```
CREATE VIEW View_Daily_interactions AS
SELECT
    rpi.date AS action_date,
    COUNT(DISTINCT lp.like_event_id) AS total_likes,
    COUNT(DISTINCT p.post_id) AS total_posts
FROM
    record_post_interaction rpi
    LEFT JOIN
    likes_post lp ON rpi.like_id = lp.like_id
    LEFT JOIN
    post p ON rpi.user_id = p.user_id
GROUP BY
    rpi.date
ORDER BY rpi.date DESC;
```

	action_date	total_likes	total_posts
►	2024-05-25	2	0

Fig. 11 View output and QUERY

This view collects/counts the interactions with the platform by likes (like_id) and posts per day (post_id) and are grouped by date. The above average interactions view uses this table and calculates an average of daily interactions, where average interactions, $I = (\text{COUNT}(\text{total_likes} * 0.5) + \text{COUNT}(\text{totals_posts}) / 2)$ the total_likes were given a halved weight in accordance to posts and selected from the table (Fig. 11) and SELECT ALL the dates which have interactions > daily interactions. Due to the platform not currently having a lot of data this view cannot be shown in detail (fig. 12) (Also because the data dump used in the development process has been dropped by accident :). This is a practical view for the platform to detect patterns in high influxes of data. Usually posts are considered a higher weight of interaction due to more engaging nature.

	action_date	total_likes	total_posts	interactions
►	2024-05-28	2	2	1.50000

Fig. 12 View output on view_above_average_interactions

4. Stored procedures, functions and triggers justification

4.1 GetUserProfile() : JSON

getUserProfile() function (with reference to triggers.sql):

The data mentioned in Fig. 3, Fig. 4, Fig. 5 (User and follower table) above is a combination of expensive GROUP BY, COUNT subqueries that always give an output `N` on input `x` and therefore qualifies as a deterministic stored procedure stored in the database. `f() getUserProfile` returns a JSON file parsed on the application side. The reason for returning a JSON file is so that debugging and error handling is simplified, helps maintenance and therefore can be determined if it is a local problem or a server-side/DBMS problem and it will be frequently used as it is used to search users in search Explore class, it does not return sensitive data such as passwords therefore if intercepted no dangerous data is leaked. The size of the query also makes it impractical to include as source code.

4.2 CheckCommentLength() : CHAR_LENGTH, BEFORE_COMMENT_INSERT

CheckCommentLength() function (with reference to triggers.sql):

The CheckCommentLength() is always called by the BEFORE INSERT trigger in the comments table (Section 1.2). The `f()CheckCommentLength` returns the input length of a comment called by a trigger and even though comments are not implemented and such verifications usually happen application-side, when implemented by CSS having a server-side check is also practical for data integrity.

BEFORE_COMMENT_INSERT (with reference to triggers.sql):

The BeforeNewComment trigger ensures that new comments in the database are the correct length. Before a comment is added, it uses the CheckCommentLength() function to check its size. If the comment is too long, the trigger raises an error with SQLSTATE '45000' and the message 'Error2'. This server-side check can complement CSS's client-side validations, ensuring consistent enforcement of comment length rules and maintaining data integrity once comments feature are implemented.

4.3 LikesPost PROCEDURE (TRANSACTION) , BEFORE_LIKE_INSERT trigger

LikesPost PROCEDURE (with reference to triggers.sql):

The LikePost Procedure contains an inner transaction with on error rolling back and not committing the like. The reason it is a procedure is it involves several steps and the error state can be used application side to handle duplicate like bugs (which was given to us in previous versions by CSS). To maximise the enforcement of integrity rules, the stored procedure is also called by a trigger `before_like_insert`. The steps involve:

1. INSERTING values first into RPI table and getting the last auto incremented value (which represents like_id)
2. Getting the Last inserted ID from like_id in step 1 and inserting that into like_posts with the given post_id creating the relation mapping
3. COMMIT IF: NO SQL ERROR STATE, ELSE: signals '4500' and 'Transaction Failed' which are used for error handling application side.

BEFORE_LIKE_INSERT (with reference to triggers.sql):

The likes_post table has a `before insert` trigger that checks for a unique combination of LP: {B,C}. The triggers' purpose is to signal the application of duplicate likes in case the transaction `likePost` has a false positive and tries inserting the values.

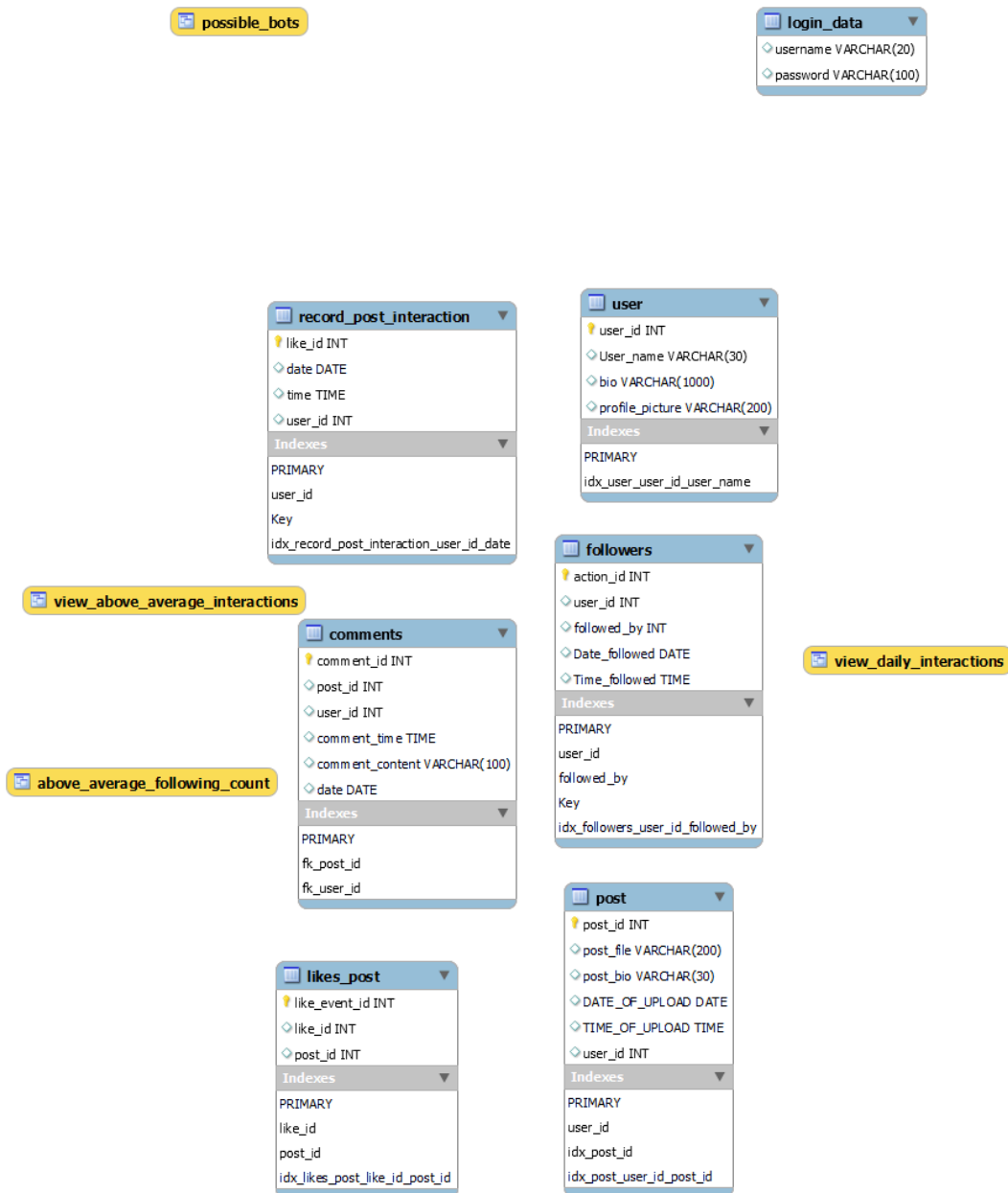
4.4 before_follow_insert TRIGGER

before_follow_insert (with reference to triggers.sql):

The before_follow_insert trigger is designed to maintain the integrity of the followers table in the quackstagram database. Before a new follower relationship is added, this trigger checks if the combination of user_id and followed_by already exists in the table. If such a relationship is found, the trigger raises an error with SQLSTATE '45000' and the message 'Already following'. This server-side check ensures that duplicate follow relationships are prevented, complementing any client-side validations and helping to maintain the consistency and reliability of CSS' follower data.

5. Query performance on index

(A different version of the database has been used for the data dump, all these queries are in reference to quackstagramdd.sql which contains the data used to complete the 20 queries and test index performance. This database does not have foreign key constraints, triggers have been removed for the sake of testing on a large data set and post.post_file(LONGBLOB) has been changed to varchar(300). Database design is maintained quackstagram DOES NOT use this version)



View performance texting (With reference to Fig. A3 Fig A4.)

View Name	Non-index performance (seconds)	Indexed Performance(seconds)
above_average_following_count	0.0780	0.094
view_above_average_interactions	0.0750	0.0719
possible_bots	3.703	2.292
view_daily_interactions	0.375	0.359

Conclusion:

The testing has been done through quackstagramdd.sql there is a 2000 user set. Post-indexing has significantly improved the possible_bots view which contains a very expensive query and given the relatively small dataset.

Controlled variables:

- *User Count: 2000*
- *Post count: 21726*
- *Like events: 31791*
- *Follow relations: 23352*

6. Profit maximisation queries (with reference to 20Queries->20Queries.sql dir)

(A different version of the database has been used for the data dump, all these queries are in reference to quackstagramdd.sql which contains the data used to complete the 20 queries and test index performance. This database does not have foreign key constraints, triggers have been removed for the sake of testing on a large data set and post.post_file(LONGBLOB) has been changed to varchar(300). Database design is maintained quackstagram DOES NOT use this version)

6.1 Select more than ? followers

USE quakcsgatramdd;

SELECT u.user_id, u.User_name, COUNT(f.followed_by) AS follower_count

FROM user u

JOIN followers f ON u.user_id = f.user_id

GROUP BY u.user_id, u.User_name

HAVING COUNT(f.followed_by) > ?; - - for the output below: 50

	user_id	User_name	follower_count
▶	72	Gearard	59
	699	Cale	53
	1969	Zathin	466

6.2 Show the total number of posts made by each user via user_id

USE quakcsgatramdd;

SELECT u.user_name, COUNT(p.post_id) AS `Number of posts`

FROM user u

LEFT JOIN post p ON u.user_id = p.user_id

GROUP BY u.user_name

ORDER BY COUNT(p.post_id) DESC;

user_name	Number of posts
Viva	223
Bitchip	219
Namfix	213
Biodex	209
Vagram	203
Transcof	202
Aerified	195
Alphazap	195
Home Ing	194

6.3 Find all comments made on a particular user's post.

USE quakcsgatramdd;

select * from comments

where user_id = ?; -- 18 for output below

```
-- Use this query to put comments, use for previous comments also --  
  
INSERT INTO comments (comment_id, post_id, user_id, comment_time,  
comment_content, date)  
VALUES  
  
    (DEFAULT, 1, 18, DEFAULT, 'hello', DEFAULT),  
  
    (DEFAULT, 1, 19, DEFAULT, 'thisprojecttookmealot', DEFAULT),  
  
    (DEFAULT, 1, 21, DEFAULT, 'ofdays', DEFAULT);  
  
-- hypothetical insert query not implemented due to no comments implementation
```

	comment_id	post_id	user_id	comment_time	comment_content	date
▶	1	1	18	NULL	hello	NULL
	4	2	18	NULL	hello	NULL

6.4 Display the top X most liked posts.

Use quackstagramdd;

```
SELECT *  
FROM post p  
    JOIN (  
        SELECT lp.post_id, COUNT(lp.post_id) AS like_count  
        FROM likes_post lp  
        GROUP BY lp.post_id  
        ORDER BY COUNT(lp.post_id) DESC  
    )  
    AS `Select Most Liked Posts` ON p.post_id = `Select Most Liked Posts`.post_id  
ORDER BY `Select Most Liked Posts`.like_count DESC  
LIMIT ?; -- put 10 for output below
```

post_id	post_file	post_bio	DATE_OF_UPLOAD	TIME_OF_UPLOAD	user_id	post_id	like_count
1	file_1192_0	Bio for post 0	2024-05-08	20:08:22	1192	1	1957
2	file_1003_1	Bio for post 1	2024-05-08	17:05:39	1003	2	1946
3	file_1949_2	Bio for post 2	2024-05-08	18:23:57	1949	3	1924
4	file_588_3	Bio for post 3	2024-05-08	06:31:46	588	4	1907
5	file_1936_4	Bio for post 4	2024-05-08	06:03:27	1936	5	1876
6	file_1087_5	Bio for post 5	2024-05-08	19:20:50	1087	6	1860
7	file_604_6	Bio for post 6	2024-05-08	04:27:05	604	7	1826
8	file_409_7	Bio for post 7	2024-05-08	17:26:55	409	8	1802
9	file_734_8	Bio for post 8	2024-05-08	17:22:30	734	9	1743
10	file_1730_9	Bio for post 9	2024-05-08	20:18:27	1730	10	1700

6.5 Count the number of posts each user has liked.

```
SELECT u.user_id, u.user_name, COUNT(like_id) AS `Number_of_likes_from_user`  
from record_post_interaction rpi RIGHT JOIN user u  
ON rpi.user_id = u.user_id  
GROUP BY u.user_id  
ORDER BY `Number_of_likes_from_user` DESC;
```

user_id	user_name	Number_of_likes_from_user
72	Gearard	23
70	Melodee	22
420	Davey	22
1310	Stim	22
1410	Cardify	22
1945	Kanlam	22
8	Faber	21
71	Matthieu	21
80	Rebbeca	21
96	Tybie	21

6.6 List all users who haven't made a post yet.

```
SELECT u.user_id, u.user_name  
FROM user u  
WHERE u.user_id NOT IN(  
SELECT u.user_id  
FROM user u join post p  
ON u.user_id=p.user_id  
);
```

user_id	user_name
8	Faber
70	Melodee
72	Gearard
80	Rebbeca
420	Davey
699	Cale
NULL	NULL

6.7 List users who follow each other.

use quackstagramdd;

```
SELECT a.user_id, a.followed_by
FROM followers a
WHERE a.followed_by IN (
  SELECT b.user_id
  FROM followers b
  WHERE a.user_id = b.followed_by
ORDER BY user_id ASC
);
```

user_id	followed_by
4	652
8	1301
8	1519
14	1322
34	335
64	547
70	1470
71	1937
72	80
72	108

6.8 Show the user with the highest number of posts.

```
SELECT get_posts.posts_count, username
FROM(
  SELECT
    COUNT(DISTINCT p.post_id) AS posts_count, u.user_name AS username
  FROM
    user u
    LEFT JOIN
    post p ON u.user_id = p.user_id
  WHERE
    u.user_name = u.user_name
  GROUP BY
    u.user_id
  ORDER BY posts_count DESC
) AS get_posts
LIMIT 1; -- GET ONLY THE FIRST ONE SINCE THE count is already sorted desc
```

	posts_count	username
▶	23	Jarret

6.9 List the top X users with the most followers.

```
SELECT user_name, Followers_count
FROM (
    SELECT u.User_name, COUNT(f.user_id) AS Followers_Count
    FROM user u JOIN followers f ON u.user_id = f.user_id
    GROUP BY u.user_id, u.User_name
    ORDER BY Followers_Count DESC
)
AS Top_FOLLOWED
LIMIT ?; -- INser desired X val output uses 3
```

	user_name	Followers_count
▶	Zathin	466
	Gearard	59
	Cale	53

6.10 Find posts that have been liked by all users.

- For this query make 2 accounts in the application and use the quackstagram.sql

USE quackstagram;

```
SELECT p.post_id, p.post_file, GROUP_CONCAT(DISTINCT u.user_name SEPARATOR ' , ') AS
usernames -- in case of error 1308, change memory alloc in your DBMS
```

```
FROM post p
    JOIN likes_post lp ON p.post_id = lp.post_id
    JOIN record_post_interaction rpi ON lp.like_id = rpi.like_id
    JOIN user u ON rpi.user_id = u.user_id
GROUP BY p.post_id, p.post_file
HAVING COUNT(DISTINCT rpi.user_id) = (SELECT COUNT(*) FROM user);
```

6.11 Display the most active user (based on posts, comments (7), and likes).

Use quackstagramd;

```
SELECT u.user_id, (postCount.post_count + countL.like_count) AS `interaction_count`
FROM user u
```



```

LEFT JOIN (
SELECT
user_id, COUNT(*) AS post_count
FROM post
GROUP BY user_id
) AS postCount ON u.user_id = postCount.user_id -- Subquery select post count
LEFT JOIN (
SELECT user_id, COUNT(*) AS like_count
FROM record_post_interaction
GROUP BY user_id
) AS countL ON u.user_id = countL.user_id -- count amount of likes by each user
ORDER BY interaction_count DESC
LIMIT 1; -- omit likes as per question specification, not implemented in the application

```

	user_id	interaction_count
▶	1731	42

6.12 Find the average number of likes per post for each user.

```

use quackstagramdd;

SELECT u.user_name, AVG(lop.`likes of post`)
FROM user u LEFT JOIN
(
SELECT p.user_id, p.post_id, l.`Likes of post`
FROM post p RIGHT JOIN(
SELECT post_id, COUNT(like_Event_id) as `Likes of post`
FROM likes_post
group by post_id -- query to count the number of likes per post
) l

```

```

ON l.post_id=p.post_id ) lop -- QUERY GET USER id with post_id and likes of post
ON u.user_id = lop.user_id
GROUP BY u.user_id
ORDER BY AVG(lop.`likes of post`) DESC
;

```

user_name	AVG(lop.`likes of post`)
Biodex	1957.0000
Asoka	1946.0000
Lotstring	1924.0000
Gaelan	1907.0000
Tresom	1876.0000
Lotlux	1860.0000
Angelico	1826.0000
Brocky	1802.0000
Allard	1743.0000

6.13 Show posts that have more comments than likes.

Use quackstagramdd; - - in this none will

```

SELECT DISTINCT comm.post_id, comm.`number_of_comments`, likes.likesPost
FROM likes_post lp JOIN
(SELECT post_id, COUNT(comment_id) AS `number_of_comments`
from comments
GROUP by post_id) AS comm
JOIN
(SELECT post_id, COUNT(like_Event_id) AS likesPost
FROM likes_post
GROUP BY post_id) as likes

```

WHERE comm.`number_of_comments`>likes.likesPost;

	post_id	number_of_comments	likesPost
--	---------	--------------------	-----------

6.14 List the users who have liked every post of a specific user.

Use quackstagram; - - use this version make an account and like from another

```
SELECT DISTINCT u.user_name AS `person that liked`
FROM user u JOIN record_post_interaction rpi JOIN likes_post lp
WHERE
rpi.like_id = lp.like_id
AND u.user_id=rpi.user_id
AND lp.post_id IN(
SELECT idu.post_id
FROM
(SELECT p.post_id, u.user_name
FROM post P JOIN user u
WHERE p.user_id=u.user_id) idu -- postid username query
WHERE idu.user_name = ? -- select post of specific user name
);
```

6.15 Display the most popular post of each user (based on likes).

Use quackstagramdd;

```
SELECT u.user_name, p.post_id, p.post_file, p.post_bio, nlp.`Number of likes`
FROM user u
      JOIN (
      SELECT lp.post_id, COUNT(lp.like_event_id) AS `Number of likes`
      FROM likes_post lp
      GROUP BY lp.post_id
```

```

) nlp ON nlp.post_id = (
    SELECT p.post_id
    FROM post p
        JOIN likes_post lp ON p.post_id = lp.post_id
    WHERE p.user_id = u.user_id
    GROUP BY p.post_id
    ORDER BY COUNT(lp.like_event_id) DESC
    LIMIT 1
)
JOIN post p ON p.post_id = nlp.post_id
ORDER BY nlp.`Number of likes` DESC; -- must have at least 2 posts

```

	user_name	post_id	post_file	post_bio	Number of likes
►	Biodex	1	file_1192_0	Bio for post 0	1957
	Asoka	2	file_1003_1	Bio for post 1	1946
	Lotstring	3	file_1949_2	Bio for post 2	1924
	Gaelan	4	file_588_3	Bio for post 3	1907
	Tresom	5	file_1936_4	Bio for post 4	1876
	Lotlux	6	file_1087_5	Bio for post 5	1860
	Angelico	7	file_604_6	Bio for post 6	1826
	Brocky	8	file_409_7	Bio for post 7	1802
	Allard	9	file_734_8	Bio for post 8	1743

6.16 Find the user(s) with the highest ratio of followers to following.

```

SELECT
    u.user_id,
    u.User_name,
    IFNULL(follower_counts.total_followers, 0) AS followers,
    IFNULL(following_counts.total_following, 0) AS following,
    IF(following_counts.total_following = 0, 'Infinity', -- Handle division by zero

```

```

        follower_counts.total_followers / following_counts.total_following) AS ratio
FROM
`user` u
LEFT JOIN (
SELECT
f.user_id,
COUNT(*) AS total_followers
FROM
`followers` f
GROUP BY
f.user_id
) follower_counts ON u.user_id = follower_counts.user_id -- Subquery to count how many
followers each user has
LEFT JOIN (
SELECT
f.followed_by,
COUNT(*) AS total_following
FROM
`followers` f
GROUP BY
f.followed_by
) following_counts ON u.user_id = following_counts.followed_by -- Subquery to count how
many users each user is following
ORDER BY
    ratio DESC
LIMIT ?; -- 'Highest insert here an INT' use 10 for output

```

user_id	User_name	followers	following	ratio
1323	Voyatouch	16	2	8.0000
477	Paulette	14	2	7.0000
1934	Quo Lux	20	3	6.6667
819	Yoshiko	13	2	6.5000
106	Kylie	19	3	6.3333
1423	Temp	19	3	6.3333
1109	Biodex	18	3	6.0000
162	Brittaney	17	3	5.6667
1409	Vagram	21	4	5.2500
84	Rosy	22	5	4.4000

6.17 Show the month with the highest number of posts made.

SELECT

YEAR(DATE_OF_UPLOAD) AS Year,

MONTH(DATE_OF_UPLOAD) AS Month,

COUNT(*) AS NumberOfPosts

FROM

post

GROUP BY

YEAR(DATE_OF_UPLOAD), MONTH(DATE_OF_UPLOAD)

ORDER BY

NumberOfPosts DESC

LIMIT 1; - - can be more

Year	Month	NumberOfPosts
2024	5	21726

6.18 Identify users who have not interacted with a specific user's posts

use quackstagramdd;

```
SELECT u.user_name AS `person that hasn't interacted`
```

```
FROM user u
```

```
WHERE NOT EXISTS (
```

```
SELECT 1
```

```
FROM post p
```

```
JOIN likes_post lp ON p.post_id = lp.post_id
```

```
JOIN record_post_interaction rpi ON lp.like_id = rpi.like_id
```

```
WHERE p.user_id = (
```

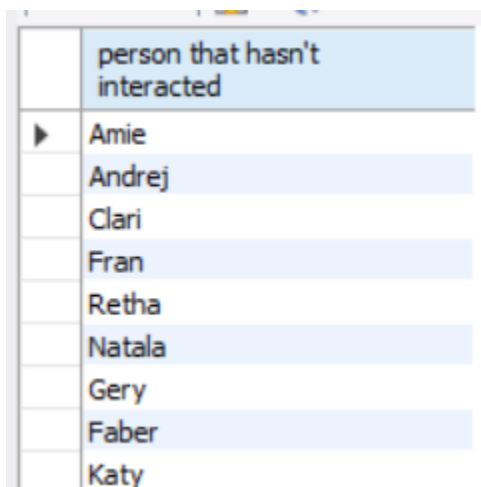
```
SELECT user_id
```

```
FROM user
```

```
WHERE user_name = ? -- use rosy for ourput
```

```
) AND rpi.user_id = u.user_id
```

```
);
```



	person that hasn't interacted
▶	Amie
	Andrej
	Clari
	Fran
	Retha
	Natala
	Gery
	Faber
	Katy

6.20 Find users who are followed by more than X% of the platform users

Use quackstagramdd;

SET @X = ?; -- for this dataset maximum is 23%, FORMAT @X = ?, where ? = INT, use 23 for output

-- Calculate the total number of users on the platform

SET @total_users = (SELECT COUNT(*) FROM `user`);

-- Calculate the threshold number of followers

SET @threshold = @total_users * (@X / 100);

SELECT u.user_name, COUNT(f.followed_by) AS followers_count

FROM `user` u

JOIN `followers` f ON u.user_id = f.user_id

GROUP BY u.user_id

HAVING COUNT(f.followed_by) > @threshold;

-- Query to find users who are followed by more than X% of the platform users

	user_name	followers_count
▶	Zathin	466

7. Appendix

	TRIGGER_NAME	EVENT	TIMING	SIZE_BYTES	SIZE_KB
▶	BeforeNewComment	INSERT	BEFORE	138	0.13
	before_follow_insert	INSERT	BEFORE	430	0.42
	before_like_insert	INSERT	BEFORE	699	0.68

Fig. A1 Trigger information.

	procedure_name	size_kb
▶	likePost	0.44

Fig. A2 Procedure information.

Overview Who Did What :

Name:	Functionality integration	Database design, query writing	Debugging/error handling implementation
R.D.M Untesu	60 %	60 %	50 %
Nitu Cristian	40 %	40 %	50 %

Fig. A3 Non-indexed log

2104	01:12:44	select * from above_average_following_count LIMIT 0, 1000000	1001 row(s) returned	0.078 sec / 0.000 sec
2105	01:18:17	select * from view_above_average_interactions LIMIT 0, 1000000	9 row(s) returned	0.750 sec / 0.000 sec
2106	01:18:37	possible_bots	Error Code: 1064. You have an error in your SQL syntax; check the manual that corresponds to your MySQL ...	0.000 sec
2107	01:18:41	select * from possible_bots LIMIT 0, 1000000	5 row(s) returned	3.703 sec / 0.000 sec
2108	01:18:52	select * from view_daily_interactions LIMIT 0, 1000000	10 row(s) returned	0.375 sec / 0.000 sec

Fig. A4 Post-indexing log

#	Time	Action	Message	Duration / Fetch
2161	01:37:50	select * from above_average_following_count LIMIT 0, 1000000	1001 row(s) returned	0.094 sec / 0.000 sec
2162	01:37:57	use quackstagramdd	0 row(s) affected	0.000 sec
2163	01:37:57	select * from view_above_average_interactions LIMIT 0, 1000000	9 row(s) returned	0.688 sec / 0.000 sec
2164	01:38:00	use quackstagramdd	0 row(s) affected	0.000 sec
2165	01:38:00	select * from possible_bots LIMIT 0, 1000000	5 row(s) returned	3.062 sec / 0.000 sec
2166	01:38:08	use quackstagramdd	0 row(s) affected	0.000 sec
2167	01:38:08	select * from view_daily_interactions LIMIT 0, 1000000	10 row(s) returned	0.359 sec / 0.000 sec