Databases - Final Project

Group 66

Natalia Hadjisoteriou, João Tavares

Tasks	Designing Database	Normalisation	Views and Indexes	Database Implementation	Triggers and Procedures	SQL Queries
João	60%	0%	10%	80%	90%	10%
Natalia	40%	100%	90%	20%	10%	90%

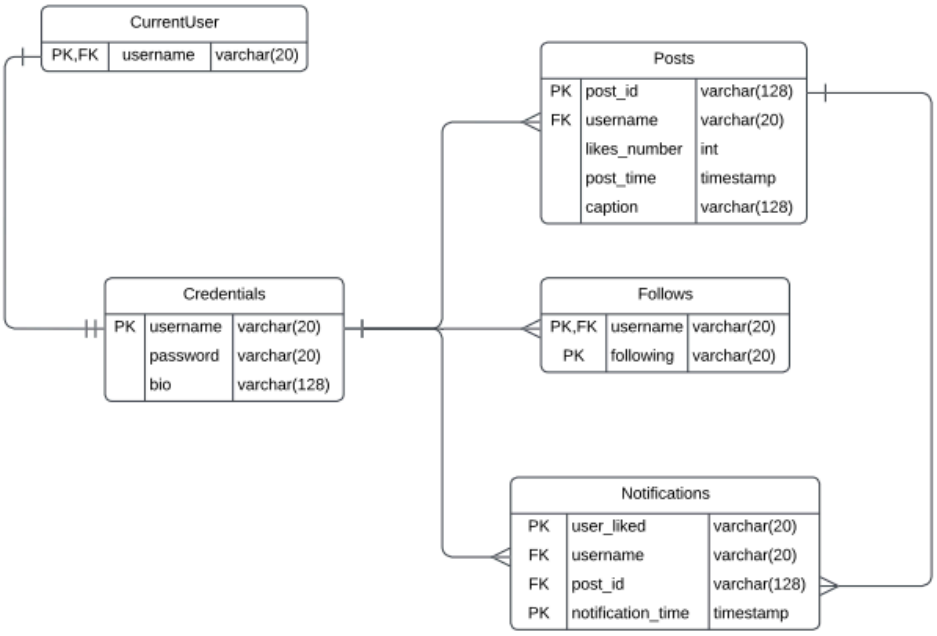
Table of Contents

1. Quackstagram Database

The **quackstagramdb** database, developed for this project, is designed to efficiently store and access data related to Quackstagram. The database consists of five tables that hold information about users, uploaded Posts, notification specifics, and the relationships among them logically and functionally. The tables are '**Credentials'**, **'CurrentUser'**, '**Follows'**, '**Notification'**, and '**Posts'**. The '**Credentials'** table contains the information a user provides during registration, including the **username**, **password**, and **bio**. This table stores new user information, validates existing users' credentials, and displays user profiles on the HomePage and ExplorePage. The '**CurrentUser**' table temporarily stores the username of the currently logged-in user, enabling the program to retrieve and display the appropriate user information on the profile page after switching between the different pages. The '**Follows**' table records the relationships between users and the people they follow, facilitating the display of "followers" and "following" information on each user's profile page by counting the tuples number appropriately. The '**Notification**' table manages all the data necessary to display notifications on the notifications page and tracks picture likes, as likes are the only actions

that generate notifications. The '**Posts**' table stores data about posts, including the user who uploaded the post, the time of upload, the number of likes, and the caption. This information is used to display the post along with its details. The Quackstagram database is designed to provide all the information required for the Quackstagram program to operate smoothly and efficiently without using text files as an unreliable method of storing data about each user and their relationships with others.

2. ERD diagram from before table normalisation



3. Functional Dependencies - Normalisation

Check if tables are in 3NF

FD A→B check if A is a superkey OR B is a prime attribute

Functional dependencies before normalisation (non-trivial):

Table 1 - Credentials

(PK) **username →password**, **bio**

'username' is a superkey = No violation

So, the table is in 3NF.

Table 2 - CurrentUser

no non-trivial functional dependencies

So, the table is in 3NF

Table 3 - Follows

no non-trivial functional dependencies

So, the table is in 3NF

Table 4 - Notifications

(PK) **notification_time**, **user_liked** → **username**, **post_id**

'**(notification_time, user_liked)**' is a superkey = no violation

post_id → **username**

VIOLATION: '**post_id**' is not a superkey, '**username**' is not a prime attribute

user_liked, **post_id** → **notification_time**

'**notification_time**' is a prime attribute = no violation

↪Candidate key: '**(notification_time, user_liked)**' (happens to be the primary key)

notification_time, **post_id** → **user_liked**

'**user_liked**' is a prime attribute = no violation

↪Candidate key: '**(notification_time, user_liked)**' (happens to be the primary key)

Since there is a violation, the table is not in 3NF.

Table 5 - Posts

(PK) **post_id** → **username**, **likes_number**, **post_time**, **caption**

'**post_id**' is a superkey = no violation

username, **post_time**→**caption**, **likes_number**, **post_id**

'**(username, post_time)**' is a superkey = no violation

So, the table is in 3NF

Normalise relation Notifications (user_liked, username, post_id, notification_time) - Decomposition:

Keys:

1. **(notification_time, user_liked)**

Prime attributes

1. **notification_time**
2. **user_liked**

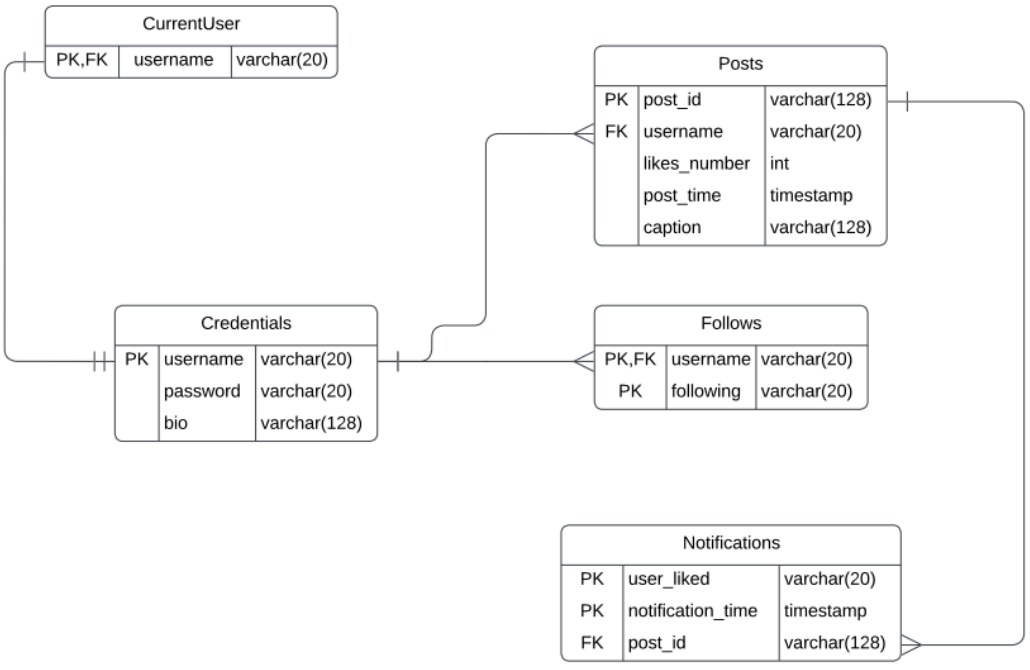
post_id → username **VIOLATION**

Step 1: Minimising the set of functional dependencies (Finding minimal basis G)

1.1 Can we remove attributes from the left-hand side of the functional dependency?
↳ NO, we only have one attribute

1.2 Can we remove the whole functional dependency?
↳ We can remove the functional dependency **post_id → username** and store the relationship between **post_id** and **username** in the relation '**Posts (post_id, username, likes_number, post_time, caption)**'
Therefore, we can remove the attribute '**username**' from the table '**Notification**'

ERD diagram from after table normalisation:



Example for each of the tables

Table 1 - Credentials

Describe table:

Field	Type	Null	Key	Default	Extra
username	varchar(20)	NO	PRI	NULL	
password	varchar(20)	NO		password	
bio	varchar(128)	YES		NULL	

Example table:

Primary key

username	password	bio
Lorin	Password	For copyright reasons, I am not Grogu
Mystar	Password	Xylo and I are not the same!
Natalia	xixi	bababababbab
Username	Password	Bio
Xylo	Password	Fierce warrior, not solo
Zara	Password	Humanoid robot much like the rest

Table 2 - CurrentUser

Describe table:

Field	Type	Null	Key	Default	Extra
username	varchar(20)	NO	PRI	NULL	

Example table:

Primary key

username
Lorin

Describe table:

Field	Type	Null	Key	Default	Extra
username	varchar(20)	NO	PRI	NULL	
following	varchar(20)	NO	PRI	NULL	

Example table:

Primary key + FK **Primary key**

username	following
Lorin	Mystar
Lorin	Xylo
Mystar	Lorin
Mystar	Zara
Username	Mystar
Username	Zara
Xylo	Lorin
Zara	Lorin

Foreign key: username referencing Credentials(username)

Table 4 - Notifications

Describe table:

Field	Type	Null	Key	Default	Extra
user_liked	varchar(20)	NO	PRI	NULL	
post_id	varchar(128)	NO	MUL	NULL	
notification_time	timestamp	NO	PRI	NULL	

Primary key **FK** **Primary key**

user_liked	post_id	notification_time
Xylo	Lorin_1	2024-03-23 19:44:59
Mystar	Lorin_2	2024-03-23 20:06:02
Xylo	Mystar_1	2024-03-23 19:45:11
Lorin	Mystar_2	2024-03-23 19:45:13
Zara	Xylo_1	2024-03-23 20:06:02

Foreign key: post_id
referencing
Posts (post_id)

Table 5 - Posts

Describe table:

Field	Type	Null	Key	Default	Extra
post_id	varchar(128)	NO	PRI	NULL	
username	varchar(20)	NO	MUL	NULL	
likes_number	int	NO		0	
post_time	timestamp	NO		CURRENT_TIMESTAMP	DEFAULT_GENERATED
caption	varchar(128)	YES		Caption	

Example table:

Primary key **FK**

post_id	username	likes_number	post_time	caption
Lorin_1	Lorin	1	2023-12-17 20:07:43	In the cookie jar my hand was not.
Lorin_2	Lorin	1	2023-12-17 20:09:35	Meditate I must.
Mystar_1	Mystar	1	2023-12-17 20:26:50	Cookies gone?.
Mystar_2	Mystar	1	2023-12-17 20:27:24	In my soup a fly is.
Xylo_1	Xylo	1	2023-12-17 20:22:40	My tea strong as Force is.
Xylo_2	Xylo	0	2023-12-17 20:23:14	Jedi mind trick failed.
Zara_1	Zara	0	2023-12-17 20:24:31	Lost my map I have. Oops.
Zara_2	Zara	1	2023-12-17 20:22:40	Yoga with Yoda

Foreign key: username
referencing
Credentials (username)

4. Views and Indexes

Views proposed:

User behaviour: View that displays how many posts each user has in descending order

Content popularity: View that displays the top 5 users with the most likes in all posts

System analytics: View that displays the monthly notifications per user

User behaviour:

```
CREATE VIEW PostsPerUser AS
    SELECT username, COUNT (username) AS num_posts
    FROM Posts
    GROUP BY username
    HAVING num_posts>0
    ORDER BY num_posts DESC;
```

Content Popularity:

```
CREATE VIEW Top5UsersMostLikes AS
    SELECT username, SUM (likes_number) AS total_likes
    FROM Posts
    GROUP BY username
    HAVING total_likes>0
    ORDER BY total_likes DESC
    LIMIT 5;
```

System analytics:

```
CREATE VIEW MonthlyNotificationsPerUser AS
SELECT
SUBSTRING_INDEX(post_id, '_', 1) AS username,
DATE_FORMAT(notification_time, '%Y-%m') AS month,  *isolating year and month from timestamp*
COUNT(*) AS monthly_notifications
FROM Notifications
GROUP BY username, month;
```

Justification for the choice of the views:

PostsPerUser: We can use this view to easily display the number of posts of each user on their profile without having to perform the "select" statement again, just by using the view and identifying the current username to display the number of posts. This view is implemented in the new version of the Quackstagram code, replacing the old functionality of identifying the number of posts for each of the users.

Top5UsersMostLikes: We could potentially use this information to display the top 5 most liked pictures on the top of the Explore page.

MonthlyNotificationsPerUser: This view displays the monthly notifications of each user of Quackstagram, which could be useful when getting insights about how users interact with each other, which users are more popular when it comes to notifications received and how the month affects the users' interactions.

Runtimes of views before implementing indexes:

To compare runtimes before and after indexes, we filled all the tables with 500 rows of dummy data to check how well the database handles a larger amount of data.

Query_ID	Duration	Query
1	0.00148850	select * from PostsPeruser
2	0.00096800	select * from Top5UsersMostLikes
3	0.00222525	select * from MonthlyNotificationsPerUser

Indexes:

To create two efficient indexes for these views, we need to identify attributes that are frequently used in each one.

```
CREATE VIEW PostsPerUser AS
    SELECT username, COUNT (username) AS num_posts
    FROM Posts
    GROUP BY username
    HAVING num_posts>0
    ORDER BY num_posts DESC;
```

From this view, we identify the columns from the table '**Posts**' that would be suitable for creating indexes. The attribute most frequently used in this view is **username**. In fact, it is the only one that's being used so it is clear that an attribute should be included.

CREATE INDEX idx_username ON Posts (username);

Content Popularity:

```
CREATE VIEW Top5UsersMostLikes AS
    SELECT username, SUM (likes_number) AS total_likes
    FROM Posts
    GROUP BY username
    HAVING total_likes>0
    ORDER BY total_likes DESC
    LIMIT 5;
```

From this view, we identify the columns from table '**Posts**' that would be suitable for creating indexes. In this case, the column **username** is again mostly used, meaning the index created from the first view will increase the efficiency on the second view as well.

```
CREATE VIEW MonthlyNotificationsPerUser AS
SELECT
SUBSTRING_INDEX(post_id, '_', 1) AS username,
DATE_FORMAT(notification_time, '%Y-%m') AS month,  *isolating year and month from timestamp*
COUNT(*) AS monthly_notifications
FROM Notifications
GROUP BY username, month;
```

From this view, we identify the columns from table '**Notification**' that would be suitable for creating indexes. From this table, the two attributes used are **post_id** used to identify who the post belongs to and **notification_time**. In this case, the attribute being manipulated the most is **notification_time**, meaning it would be efficient to create the second index on it.

CREATE INDEX idx_time ON Notifications(notification_time);

Runtimes of views after implementing indexes:

```
+----------+------------+-----------------------------------------+
| Query_ID | Duration   | Query                                   |
+----------+------------+-----------------------------------------+
|        1 | 0.00123350 | select * from PostsPerUser              |
|        2 | 0.00055300 | select * from Top5UsersMostLikes        |
|        3 | 0.00191075 | select * from MonthlyNotificationsPerUser |
+----------+------------+-----------------------------------------+
```

Table 1:

	Before	*After*
1	*0.00148850 seconds*	*0.00123350 seconds*
2	*0.00096800 seconds*	*0.00055300 seconds*
3	*0.00222525 seconds*	*0.00191075 seconds*

1st view: 17.1313% runtime decrease
2nd view: 42.8719% decrease
3rd view: 14.1332% decrease

The implementation of the two indexes efficiently decreases the runtime of all the views, proving their usefulness. The data proves that the indexes optimise the runtime by quite a lot (look at Table 1).

5. Triggers, Function and Procedure

In our database implementation, we have integrated two triggers and procedures to manage our database:

1. **Triggers:**

- AfterLikesInsert Trigger:

This trigger is activated after each insertion into the `Notification` table, specifically when a new like is recorded. It calls the procedure `**UpdatePostLikes(image_id VARCHAR(128))**` to update the total number of likes for the post associated with the inserted notification. Takes the `**NEW.post_id**` parameter to pass the **post_id** to the `**UpdatePostLikes**` procedure.

```
delimiter //
CREATE TRIGGER AfterLikeInsert
AFTER INSERT ON notifications
FOR EACH ROW
BEGIN
      CALL UpdatePostLikes(NEW.post_id);
END//
```

- AfterLikesDelete Trigger:

This trigger is executed after each deletion from the `Notification` table, aka when a like is removed. Similar to the previous trigger, it calls the `**UpdatePostLikes**` procedure to decrement the total number of likes for the post linked with the deleted notification. It takes the `**OLD.post_id**` parameter to pass the **post_id** to the `**UpdatePostLikes**` procedure.

```
delimiter //
CREATE TRIGGER AfterLikeDelete
AFTER DELETE ON notifications
FOR EACH ROW
BEGIN
   CALL UpdatePostLikes(OLD.post_id);
END//
```

2. **Procedure:**

- UpdatePostLikes Procedure:

This procedure accepts a `**post_id**` parameter **(VARCHAR(128))** and updates the total likes count for the post associated with the provided **post_id**. It retrieves the likes count for the specified post from the `**Notification**` table and updates the **likes_number** field in the `**Posts**` table with the retrieved count.

```
delimiter //
CREATE PROCEDURE UpdatePostLikes(IN image_id VARCHAR(128))
BEGIN
   UPDATE posts
   SET likes_number = (SELECT COUNT(*) FROM notifications WHERE
notifications.post_id = image_id)
   WHERE posts.post_id = image_id;
END//
```

3. **Function:**

- CalculateTotalLikesForUser Function:

Although not yet implemented due to the lack of a `total_likes` field in the `**Credentials**` table, this function is supposed to calculate the total number of likes for a specific user. It could be called within triggers or queries to determine the total likes count for a given user. In the triggers '**AfterLikesInsert**' or '**AfterLikesDelete**', the `**CalculateTotalLikesForUser**` function would be called to compute the total likes for the user whose post was liked, while the `UpdatePostLikes` procedure would update the total likes count for the post itself. In the code below, there is the function '**CalculateTotalLikesForUser**', as well as an example of how the trigger '**AfterLikesInsert**' would look like if it were to be implemented.

```
delimiter //
CREATE FUNCTION CalculateTotalLikesForUser(username VARCHAR(20)) RETURNS
INT
BEGIN
   DECLARE total_likes INT;
   SELECT SUM(likes_number) INTO total_likes FROM posts WHERE username =
username;
   RETURN total_likes;
END//

CREATE TRIGGER AfterLikeInsert
AFTER INSERT ON notifications
```

```
FOR EACH ROW
BEGIN
   DECLARE total_likes INT;
   SET total_likes = CalculateTotalLikesForUser(
      (SELECT username FROM posts WHERE post_id = NEW.post_id)
   );
   UPDATE credentials
   SET total_likes = total_likes
   WHERE username = (
      SELECT username
      FROM posts
      WHERE post_id = NEW.post_id
   );
   CALL UpdatePostLikes(NEW.post_id);
END//
```

6. Queries and answers

All the queries are in a Queries.sql file, which has been included in the java code, giving you the possibility to run the queries through the terminal and interacting with the program by specifying which query you want to run and what the value of variable X should be. The java code accesses the .sql file and runs the queries on the spot. The java class that handles this in the Quackstagram project is **SqlTesting.java**

1. List all users who have more than X followers where X can be any integer value.

```
SELECT following, COUNT(username) AS number_followers
FROM Follows
GROUP BY following
HAVING number_followers > @X;
```

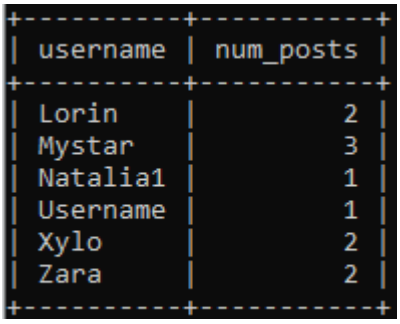
Example Output:

```
+-----------+------------------+
| following | number_followers |
+-----------+------------------+
| Lorin     |                4 |
+-----------+------------------+
```

2. Show the total number of posts made by each user.

```
SELECT username, COUNT(username) AS num_posts
FROM Posts
GROUP BY username;
```

Example Output:

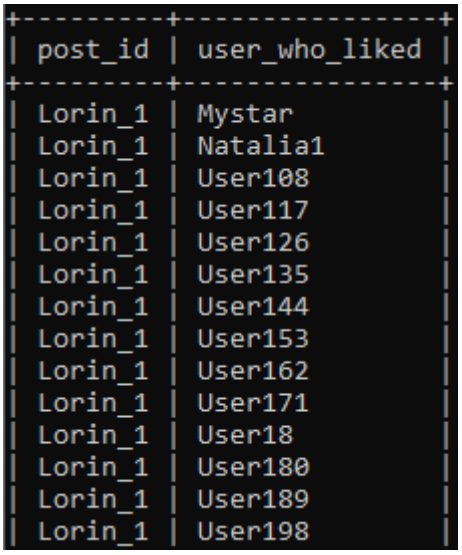


username	num_posts
Lorin	2
Mystar	3
Natalia1	1
Username	1
Xylo	2
Zara	2

3. Find all comments (**likes**) made on a particular user's post.
(Haven't implemented comments so we consider likes. due to the nature of "likes" we can only consider displaying the number of likes of a particular post X)

```
SELECT post_id, user_liked AS user_who_liked
FROM posts JOIN notifications USING(post_id)
WHERE posts.post_id = '@X'';
```

Example Output:



post_id	user_who_liked
Lorin_1	Mystar
Lorin_1	Natalia1
Lorin_1	User108
Lorin_1	User117
Lorin_1	User126
Lorin_1	User135
Lorin_1	User144
Lorin_1	User153
Lorin_1	User162
Lorin_1	User171
Lorin_1	User18
Lorin_1	User180
Lorin_1	User189
Lorin_1	User198

4. Display the top X most liked posts.

```
SELECT post_id, likes_number
FROM Posts
ORDER BY likes_number DESC
LIMIT @X;
```

Example Output for X = 5:

post_id	likes_number
Lorin_1	58
Mystar_1	58
Mystar_2	58
Username_2	58
Lorin_2	57

5 rows in set (0.00 sec)

5. Count the number of posts each user has liked.

```
SELECT n.user_liked, COUNT(n.post_id) AS number_likes
FROM Notifications n
JOIN Posts p ON n.post_id = p.post_id
GROUP BY n.user_liked;
```

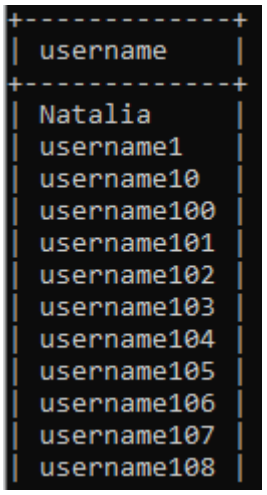
Example Output:

user_liked	number_likes
Mystar	4
Natalia1	3
User108	1
User117	1
User126	1
User135	1
User144	1
User153	1
User162	1
User171	1
User18	1
User180	1
User189	1
User198	1

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

```
SELECT username
FROM Credentials
WHERE username NOT IN (SELECT username FROM Posts);
```

Example Output:

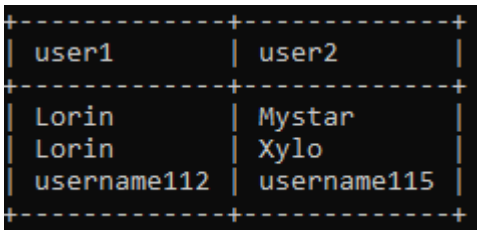


username
Natalia
username1
username10
username100
username101
username102
username103
username104
username105
username106
username107
username108

7. List users who follow each other.

```
SELECT a.username AS user1, a.following AS user2
FROM follows a
JOIN follows b ON a.username = b.following AND a.following = b.username
WHERE a.username <b.username;
```

Example Output:

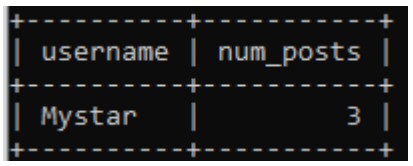


user1	user2
Lorin	Mystar
Lorin	Xylo
username112	username115

8. Show the user with the highest number of posts.

```
SELECT username, COUNT(*) AS num_posts
    FROM Posts
    GROUP BY username
    ORDER BY num_posts DESC
    LIMIT 1;
```

Example Output:



username	num_posts
Mystar	3

9. List the top X users with the most followers.

```
SELECT username, COUNT(following) AS followers
FROM Follows
GROUP BY username
ORDER BY followers DESC
LIMIT @X;
```

Example Output for X = 5:

```
+-----------+-----------+
| username  | followers |
+-----------+-----------+
| Lorin     |         4 |
| Natalia1  |         3 |
| Username  |         3 |
| Mystar    |         2 |
| username1 |         1 |
+-----------+-----------+
5 rows in set (0.00 sec)
```

10. Find posts that have been liked by all users.

```
SELECT post_id
FROM Posts
WHERE likes_number = (SELECT COUNT(*)
FROM Credentials);
```

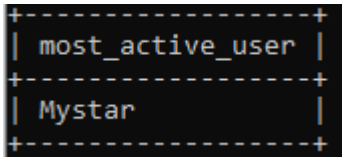
Example Output: (no post has around 500 likes)

```
Empty set (0.00 sec)
```

11. Display the most active user (based on posts, comments , and likes).
(We don't have a comments functionality in our code so we will base it on posts and likes)

```
SELECT username AS most_active_user
FROM posts, notifications
WHERE posts.username = notifications.user_liked
GROUP BY username
ORDER BY SUM(posts.post_id) + COUNT(notifications.user_liked) DESC
LIMIT 1;
```

Example Output:

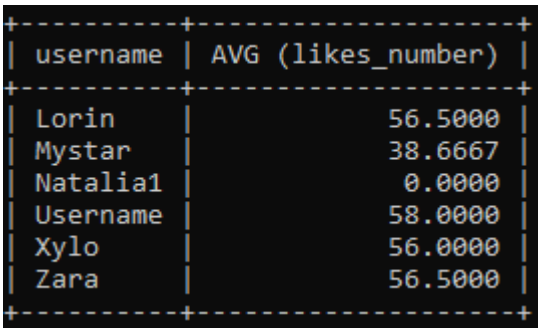


most_active_user
Mystar

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

```
SELECT username, AVG (likes_number)
FROM Posts
GROUP BY username;
```

Example Output:



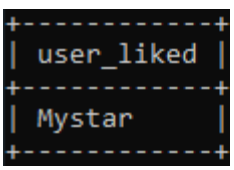
username	AVG (likes_number)
Lorin	56.5000
Mystar	38.6667
Natalia1	0.0000
Username	58.0000
Xylo	56.0000
Zara	56.5000

13. Show posts that have more comments than likes.
(We do not have comment implemented in our code therefore we cannot execute this query)

14. List the users who have liked every post of a specific user (X).

```
SELECT user_liked
FROM notifications
WHERE post_id IN (
   SELECT post_id
   FROM posts
   WHERE username = '@X')
GROUP BY user_liked
HAVING COUNT(DISTINCT post_id) = (
   SELECT COUNT(*)
   FROM posts
   WHERE username = '@X');
```

Example Output:



user_liked
Mystar

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

```
SELECT post_id
FROM posts
WHERE (username, likes_number) IN (SELECT  username,  MAX(likes_number)
 FROM posts
 GROUP BY  username
  );
```

Example Output:

```
+------------+
| post_id    |
+------------+
| Lorin_1    |
| Mystar_1   |
| Natalia1_1 |
| Username_2 |
| Xylo_1     |
| Zara_2     |
+------------+
```

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

```
SELECT following.username,
      COUNT(follower.username) / NULLIF(COUNT(following.following), 0) AS ratio
FROM follows AS following
LEFT JOIN follows AS follower ON following.following = follower.username
GROUP BY following.username
ORDER BY ratio DESC;
```

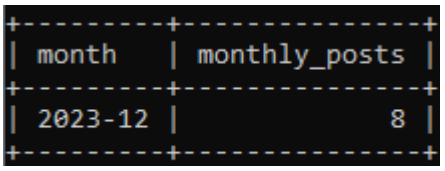
Example Output:

```
+-------------+--------+
| username    | ratio  |
+-------------+--------+
| username16  | 0.0000 |
| username261 | 0.0000 |
| Lorin       | 0.8333 |
| Mystar      | 1.0000 |
| Natalia1    | 1.0000 |
| Username    | 1.0000 |
| username1   | 1.0000 |
| username10  | 1.0000 |
| username100 | 1.0000 |
| username101 | 1.0000 |
| username102 | 1.0000 |
| username103 | 1.0000 |
```

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

```
SELECT
DATE_FORMAT(post_time, '%Y-%m') AS month,
COUNT(*) AS monthly_posts
FROM
Posts
GROUP BY
month
ORDER BY
 monthly_posts DESC
LIMIT 1;
```

Example Output:

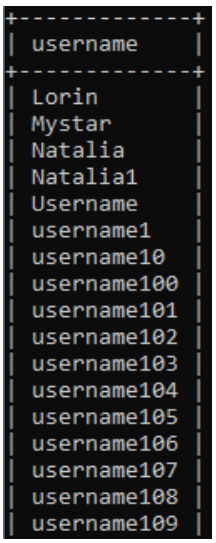


month	monthly_posts
2023-12	8

18. Identify users who have not interacted with a specific user's posts (X).

```
SELECT username
FROM Credentials
WHERE username NOT IN (SELECT DISTINCT user_liked
FROM
(SELECT * FROM posts JOIN notifications USING (post_id)
WHERE posts.post_id = notifications.post_id) AS subquery
WHERE post_id = '@X');
```

Example Output:



username
Lorin
Mystar
Natalia
Natalia1
Username
username1
username10
username100
username101
username102
username103
username104
username105
username106
username107
username108
username109

19. Display the user with the greatest increase in followers in the last X days.

There is no timestamp attribute in the table ‘follows’. Hence, it is impossible to determine when users gained followers.

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

```
SELECT following AS user
FROM Follows
GROUP BY following
HAVING COUNT(*) >
(SELECT (@X * COUNT(*) / 100)
   FROM Credentials);
```

Example Output for X = 1: