quackstagram report (databases course)

2025

Contents

1 Overview of who did what **3**

2 What my database is about and a description of how the Quackstagram project uses my database **3**

3 Database ERD diagram (before normalization) **4**

4 Functional Dependencies (Pre-Normalized Schema) **7**

5 Normalization and proof of 3NF **8**
5.1 USER 8
5.2 PICTURE 8
5.3 COMMENT 8
5.4 LIKE 9
5.5 FOLLOW 9
5.6 NOTIFICATION 9
5.7 ADMIN_USER and REGULAR_USER 10

6 Database ERD diagram (after normalization) **10**

7 An example of each table with some data and primary key attributes clearly identified **11**

8 Justification for the usefulness of the views within a scenario for possible use of the views in the Quackstagram Project **13**
8.1 a. User Activity Levels: Most Active Commenters 13
8.2 b. User Engagement Metrics: Engagement Score 14
8.3 c. Content Popularity: Most Liked Pictures 14
8.4 d. Content Popularity: Recent Popular Content 14
8.5 e. System Analytics: Daily User Registrations 15
8.6 f. System Analytics: Notification Trends 15

9 Analysis of the speed of the queries in your views and justification for the indexes 16
9.1 Index Design for Performance Optimization : Justification 16
9.1.1 Index 1: Optimize LIKE Queries by imagePath 16
9.1.2 Index 2: Optimize COMMENT Queries by username . . . 16
9.1.3 Index 3: Optimize PICTURE Queries by creation time . . 16
9.1.4 Index 4: Optimize NOTIFICATION composite query . . 17
9.1.5 Index 5: Optimize FOLLOW relationships 17
9.2 Index Design for Performance Optimization : Results 18
9.2.1 LIKE 18
9.2.2 PICTURE 19
9.2.3 COMMENT 20
9.2.4 FOLLOW 21
9.2.5 NOTIFICATION 22
9.2.6 Why Indexes Can Be Slower on Small Tables 22

10 Justification for the necessity of the triggers and stored procedure and function within a scenario for possible use of the triggers within Quackstagram 23
10.1 Stored Procedure: LogUserActivity 23
10.2 Function: CountUserPictures 23
10.3 Trigger: before_comment_insert 23
10.4 Trigger: after_user_insert 24

11 SQL queries with answers to all the questions asked in Part D 24
11.1 1. List all users who have more than X followers where X can be any integer value. 24
11.2 2. Show the total number of posts made by each user. 25
11.3 3. Find all comments made on a particular user's post 25
11.4 4. Display the top X most liked posts 25
11.5 5. Count the number of posts each user has liked 26
11.6 6. List all users who haven't made a post yet 26
11.7 7. List users who follow each other 26
11.8 8. Show the user with the highest number of posts 26
11.9 9. List the top X users with the most followers 27
11.1010. Find posts that have been liked by all users 27
11.1111. Display the most active user 27
11.1212. Find the average number of likes per post for each user. . . . 28
11.1313. Show posts that have more comments than likes 28
11.1414. List the users who have liked every post of a specific user. . . 28
11.1515. Display the most popular post of each user 29
11.1616. Find the user(s) with the highest ratio of followers to following. 29
11.1717. Show the month with the highest number of posts made . . . 30
11.1818. Identify users who have not interacted with a specific user's posts 30

11.1919. Display the user with the greatest increase in followers in the last X days 31
11.2020. Find users who are followed by more than X% of the platform users 32

1 Overview of who did what

Aditya Singha - complete project

2 What my database is about and a description of how the Quackstagram project uses my database

- **Purpose:**
  - Stores all data for Quackstagram, a simple social media app
  - Handles users, posts, comments, likes, follows, and notifications
- **Key Tables:**
  - `USER`: Usernames, bios, passwords
  - `PICTURE`: Posts with captions and timestamps
  - `COMMENT` & `LIKE`: User interactions
  - `FOLLOW`: Who follows whom
  - `NOTIFICATION`: Alerts for likes, comments, follows
- **Connection Design Choice:**
  - **Why Not Singleton:**
    - Singleton locks the connection, causing slowdowns with multiple users
    - Can create bottlenecks during high traffic
  - **Our Solution:**
    - `DatabaseConnector` creates new connections per request
    - Better performance under load
    - Isolated queries prevent system-wide crashes

How Quackstagram Uses the Database

- **User Actions:**
  - Posting a picture → Insert into `PICTURE`

  - Liking/commenting → Updates `LIKE`/`COMMENT` tables
  - Following someone → Adds record to `FOLLOW`

- **Content Display:**
  - Feed generation = Load from `PICTURE` + check `FOLLOW` relationships
  - Notifications = Pull from `NOTIFICATION` based on user activity

- **Admin Features:**
  - Special `ADMIN_USER` table for moderator accounts
  - Future expansion capabilities

Design Advantages

- **Performance:**
  - Indexes on `username`, `imagePath` etc. optimize searches

- **Organization:**
  - Clear table relationships (comments tied to posts and users)

- **Scalability:**
  - Connection-per-request model handles user growth
  - No single point of failure

3 Database ERD diagram (before normalization)

We examine the codebase initially, scan the various relations and their relationships. ERD diagram 1 shows the relations before any changes.

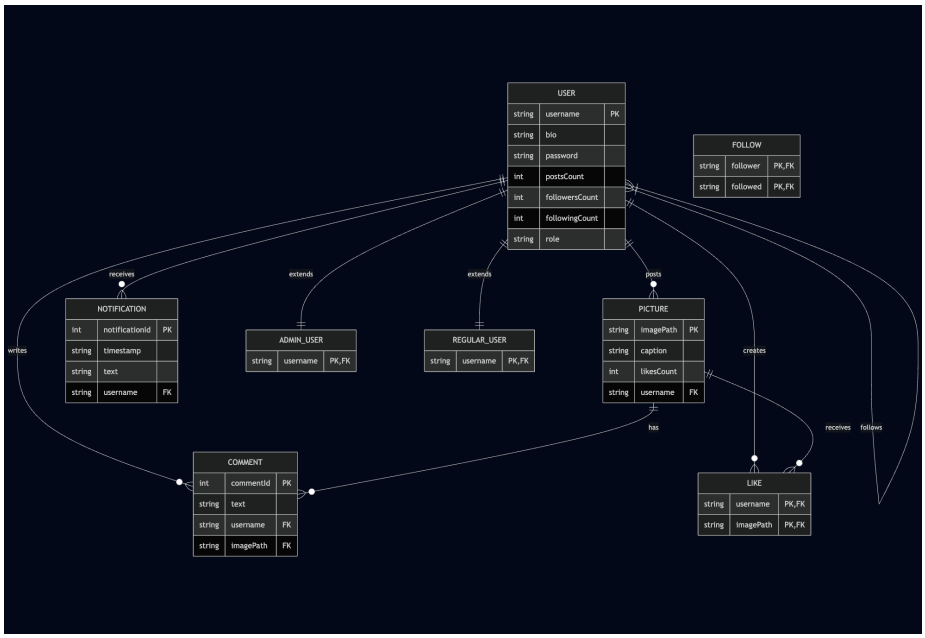


Figure 1: Initial ERD

We examine the codebase now, and then go over the questions which are to be answered later via SQL queries. Then we come up with a better ERD for our database schema. The project is a bit challenging because the codebase is poorly refactored, and thus the SQL queries will be substantially harder to execute because there are no explicit Like, Comment, Follow or Notification Classes, and also there is not enough time to refactor the codebase before starting the project.

2 is the updated ERD.

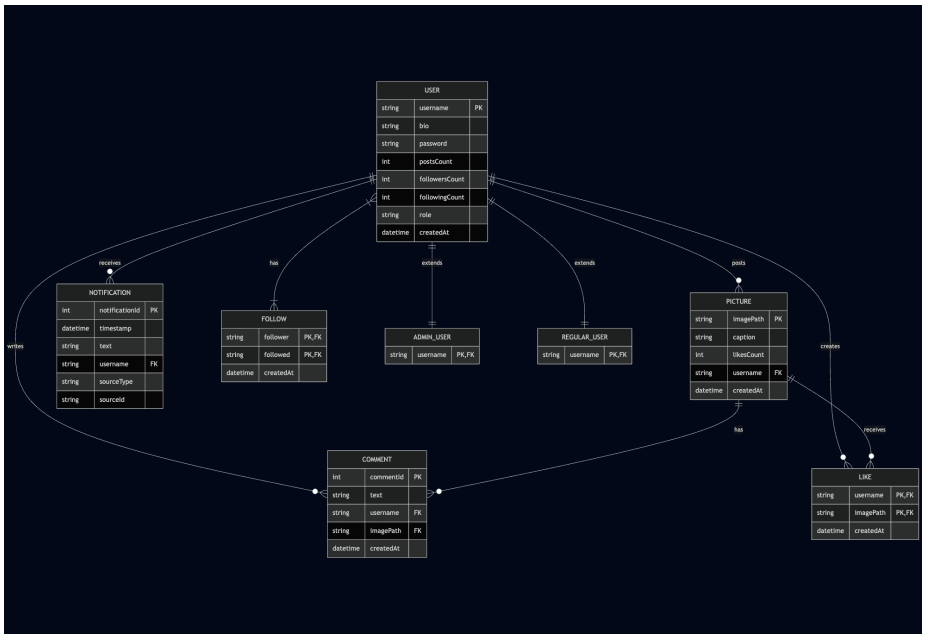


Figure 2: Revised ERD

Changes made and line of reasoning:

- **Added Creation Timestamps** (`createdAt`)
  - Essential for:
    - Sorting content chronologically
    - Activity tracking & analytics
    - "Time since" display (e.g., "Posted 2h ago")
- **Explicit `FOLLOW` Entity**
  - Replaced direct M:N with dedicated table
  - Clearer fields: `follower` (FK), `followed` (FK)
- **Enhanced `NOTIFICATION` Entity**
  - Added:
    - `sourceType` ("like", "comment", "follow")
    - `sourceId` (references triggering entity)
  - Enables better linking/filtering of notifications
- **Clearer Relationship Names**
  - E.g., `USER ''has'' FOLLOW` (vs. generic naming)
  - Improves readability & self-documentation

4 Functional Dependencies (Pre-Normalized Schema)

- **USER**
  - username → bio, password, postsCount, followersCount, following-Count, role, createdAt
  - role → {presence in ADMIN_USER or REGULAR_USER}
- **PICTURE**
  - imagePath → caption, likesCount, username, createdAt
  - username → multiple imagePaths (1:N relationship)
- **COMMENT**
  - commentId → text, username, imagePath, createdAt
  - imagePath → multiple commentIds (1:N)
- **LIKE**
  - (username, imagePath) → createdAt
  - imagePath → multiple usernames (many users can like one image)
- **FOLLOW**
  - (follower, followed) → createdAt
  - follower → multiple followed
  - followed → multiple followers
- **NOTIFICATION**
  - notificationId → timestamp, text, username, sourceType, sourceId
  - sourceType → {like, comment, follow} (domain constraint)
- **ADMIN_USER**
  - username → isAdmin = true
  - username ∈ USER(username)
- **REGULAR_USER**
  - username → isAdmin = false
  - username ∈ USER(username)

5 Normalization and proof of 3NF

5.1 USER

- **Primary Key:** username
- **Normalization Proof:**
  - All attributes are atomic (1NF)
  - No composite key (automatically 2NF)
  - Removed derived (postsCount, followersCount, followingCount) and redundant (role) attributes
  - All remaining attributes depend only on the primary key
- **Final Schema:** USER(username, bio, password, createdAt)

5.2 PICTURE

- **Primary Key:** imagePath
- **Normalization Proof:**
  - Atomic values (1NF)
  - Single-column PK (2NF)
  - Removed derived likesCount (count of LIKE records)
  - Attributes caption, username, createdAt depend only on imagePath
- **Final Schema:** PICTURE(imagePath, caption, username (FK), createdAt)

5.3 COMMENT

- **Primary Key:** commentId
- **Normalization Proof:**
  - Atomic values (1NF)
  - Single-column PK (2NF)
  - All attributes functionally depend on commentId
  - No transitive dependencies
- **Final Schema:** COMMENT(commentId, text, username (FK), imagePath (FK), createdAt)

5.4 LIKE

- **Primary Key:** (username, imagePath)
- **Normalization Proof:**
  - Atomic values (1NF)
  - No partial dependencies - createdAt depends on both PK components (2NF)
  - No non-key attributes causing transitive dependencies (3NF)
- **Final Schema:** LIKE(<u>username, imagePath</u>, createdAt)

5.5 FOLLOW

- **Primary Key:** (follower, followed)
- **Normalization Proof:**
  - Atomic values (1NF)
  - No partial dependencies - createdAt depends on both PK components (2NF)
  - No non-key attributes causing transitive dependencies (3NF)
  - Added business rule: CHECK(follower $\neq$ followed)
- **Final Schema:** FOLLOW(<u>follower, followed</u>, createdAt, CHECK(follower $\neq$ followed))

5.6 NOTIFICATION

- **Primary Key:** notificationId
- **Normalization Proof:**
  - Atomic values (1NF)
  - Single-column PK (2NF)
  - All attributes depend on notificationId
  - Added domain constraint for sourceType
- **Final Schema:** NOTIFICATION(<u>notificationId</u>, timestamp, text, username (FK), sourceType, sourceId, CHECK(sourceType $\in$ {like, comment, follow}))

5.7 ADMIN_USER and REGULAR_USER

- **Primary Key:** username (FK to USER)
- **Normalization Proof:**
  - Atomic values (1NF)
  - Single-column PK (2NF)
  - No non-key attributes (trivially 3NF)
  - Proper subtype implementation after removing role from USER
- **Final Schemas:**
  - ADMIN_USER(<u>username</u>) with FK to USER
  - REGULAR_USER(<u>username</u>) with FK to USER

6 Database ERD diagram (after normalization)

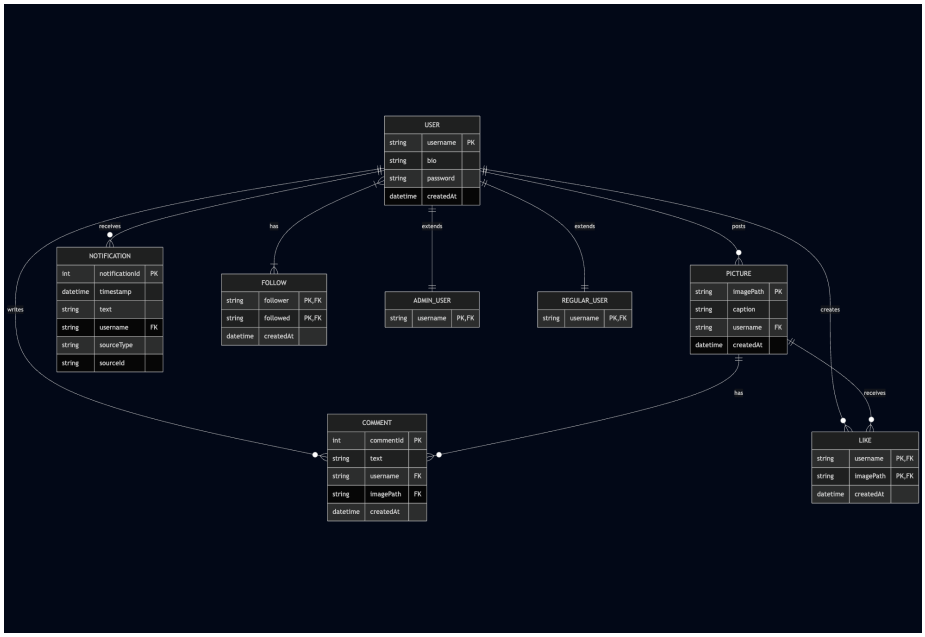


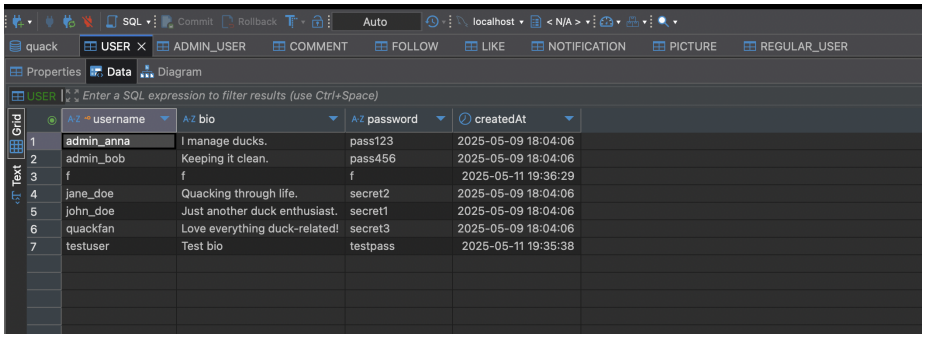
Figure 3: Final ERD

Constraints

- FOLLOW: CHECK (follower < > followed)
- NOTIFICATION: CHECK (sourceType IN ('like', 'comment', 'follow'))

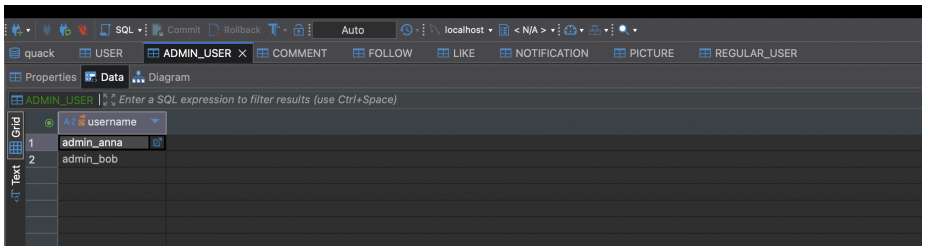
7 An example of each table with some data and primary key attributes clearly identified

Below attahced are images of sample data for each table in the database. You can view the attribute symbols to locate primary and foreign keys in each table



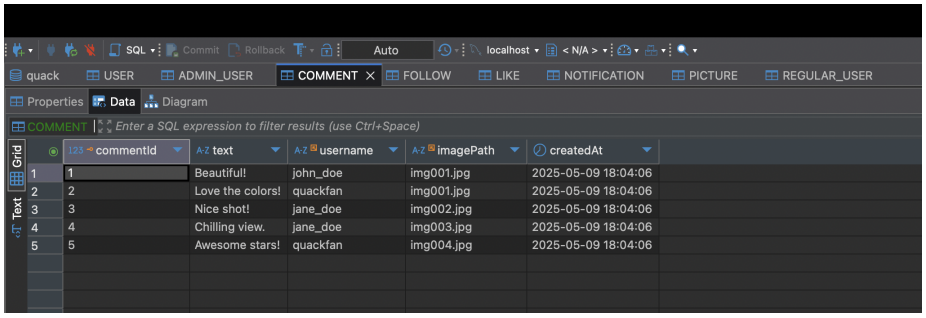
	username	bio	password	createdAt
1	admin_anna	I manage ducks.	pass123	2025-05-09 18:04:06
2	admin_bob	Keeping it clean.	pass456	2025-05-09 18:04:06
3	f	f	f	2025-05-11 19:36:29
4	jane_doe	Quacking through life.	secret2	2025-05-09 18:04:06
5	john_doe	Just another duck enthusiast.	secret1	2025-05-09 18:04:06
6	quackfan	Love everything duck-related!	secret3	2025-05-09 18:04:06
7	testuser	Test bio	testpass	2025-05-11 19:35:38

Figure 4: USER



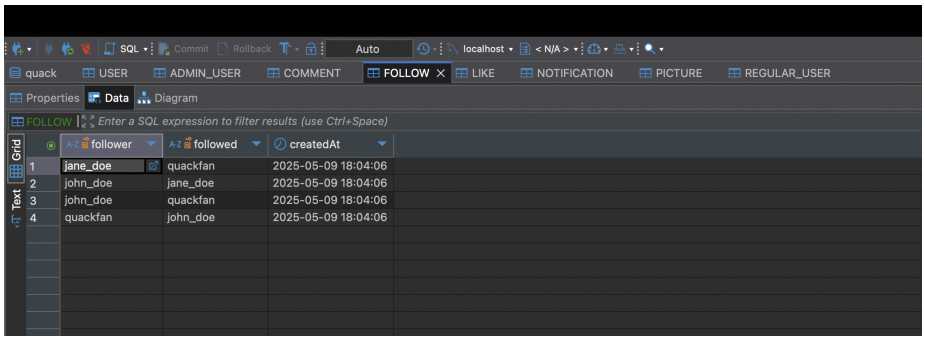
	username
1	admin_anna
2	admin_bob

Figure 5: ADMIN USER



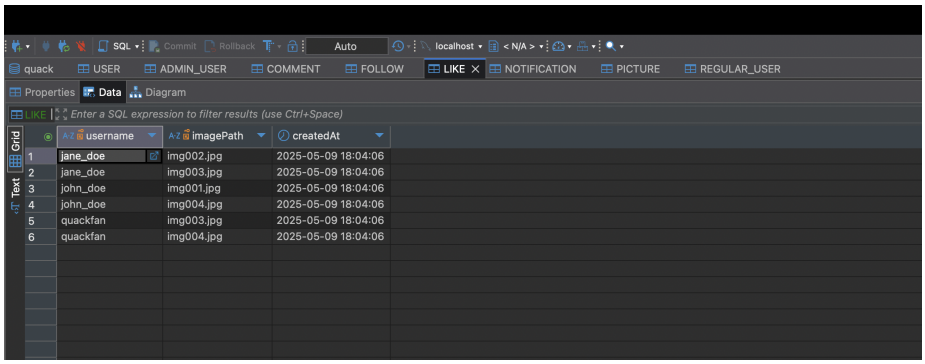
	commentId	text	username	imagePath	createdAt
1	1	Beautiful!	john_doe	img001.jpg	2025-05-09 18:04:06
2	2	Love the colors!	quackfan	img001.jpg	2025-05-09 18:04:06
3	3	Nice shot!	jane_doe	img002.jpg	2025-05-09 18:04:06
4	4	Chilling view.	jane_doe	img003.jpg	2025-05-09 18:04:06
5	5	Awesome stars!	quackfan	img004.jpg	2025-05-09 18:04:06

Figure 6: COMMENT



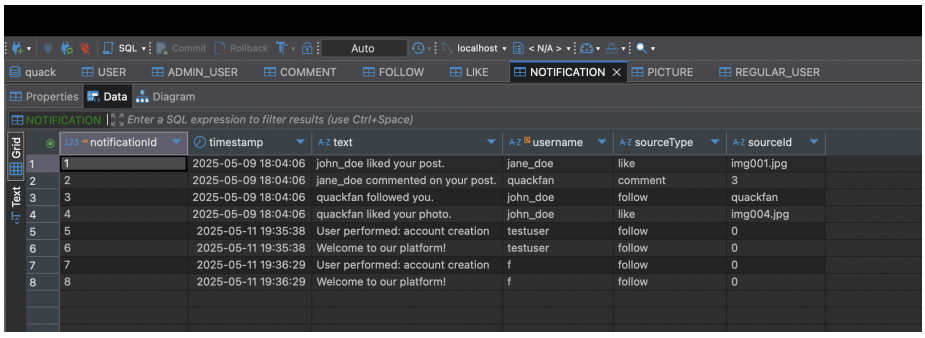
	follower	followed	createdAt
1	jane_doe	quackfan	2025-05-09 18:04:06
2	john_doe	jane_doe	2025-05-09 18:04:06
3	john_doe	quackfan	2025-05-09 18:04:06
4	quackfan	john_doe	2025-05-09 18:04:06

Figure 7: FOLLOW



	username	imagePath	createdAt
1	jane_doe	img002.jpg	2025-05-09 18:04:06
2	jane_doe	img003.jpg	2025-05-09 18:04:06
3	john_doe	img001.jpg	2025-05-09 18:04:06
4	john_doe	img004.jpg	2025-05-09 18:04:06
5	quackfan	img003.jpg	2025-05-09 18:04:06
6	quackfan	img004.jpg	2025-05-09 18:04:06

Figure 8: LIKE



	notificationId	timestamp	text	username	sourceType	sourceId
1	1	2025-05-09 18:04:06	john_doe liked your post.	jane_doe	like	img001.jpg
2	2	2025-05-09 18:04:06	jane_doe commented on your post.	quackfan	comment	3
3	3	2025-05-09 18:04:06	quackfan followed you.	john_doe	follow	quackfan
4	4	2025-05-09 18:04:06	quackfan liked your photo.	john_doe	like	img004.jpg
5	5	2025-05-11 19:35:38	User performed: account creation	testuser	follow	0
6	6	2025-05-11 19:35:38	Welcome to our platform!	testuser	follow	0
7	7	2025-05-11 19:36:29	User performed: account creation	f	follow	0
8	8	2025-05-11 19:36:29	Welcome to our platform!	f	follow	0

Figure 9: NOTIFICATION

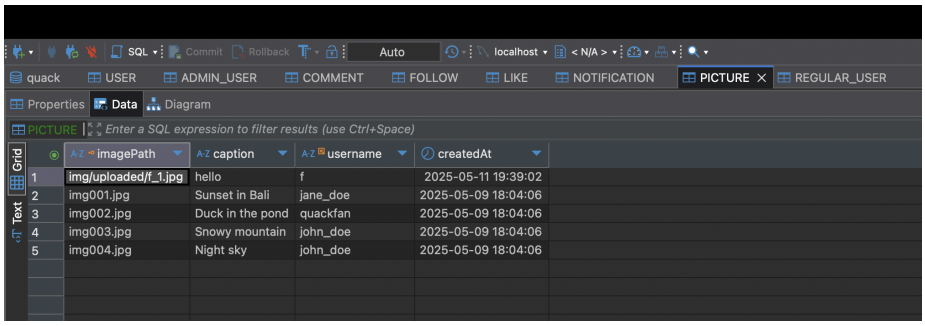


Figure 10: PICTURE

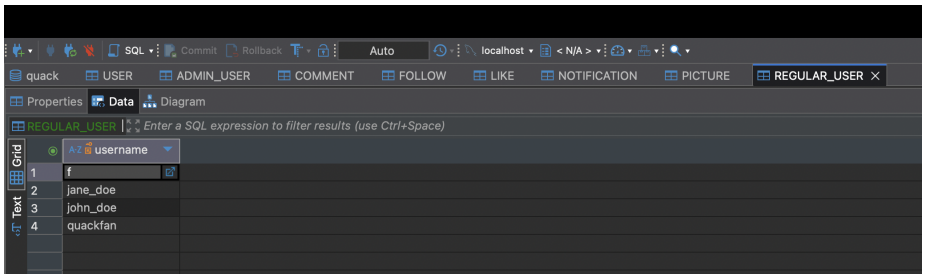


Figure 11: REGULAR USER

8 Justification for the usefulness of the views within a scenario for possible use of the views in the Quackstagram Project

8.1 a. User Activity Levels: Most Active Commenters

Justification: Identifies engaged community members who actively participate in discussions. This helps recognize valuable contributors and can inform influencer partnerships or moderation decisions.

Quackstagram Scenario: The marketing team wants to identify power users for a "Featured Commenter" program. They use this view to find users with consistent engagement to highlight in their "Community Spotlight" section.

```
CREATE VIEW MostActiveCommenters AS
SELECT username, COUNT(*) AS commentCount
FROM COMMENT
GROUP BY username
HAVING COUNT(*) > 2
ORDER BY commentCount DESC;
```

8.2 b. User Engagement Metrics: Engagement Score

Justification: Provides a composite metric weighing both likes and comments (with comments weighted higher as they indicate deeper engagement). Essential for measuring user participation holistically.

Quackstagram Scenario: When selecting users for early access to new features, the product team prioritizes those with high engagement scores from this view, ensuring feedback comes from active community members.

```
CREATE VIEW UserEngagementScores AS
SELECT u.username,
       COUNT(DISTINCT l.imagePath) AS likeCount,
       COUNT(DISTINCT c.commentId) AS commentCount,
       (COUNT(DISTINCT l.imagePath) + (COUNT(DISTINCT c.commentId) * 2)) AS engagementScore
FROM USER u
LEFT JOIN `LIKE` l ON u.username = l.username
LEFT JOIN COMMENT c ON u.username = c.username
GROUP BY u.username
HAVING engagementScore > 0
ORDER BY engagementScore DESC;
```

8.3 c. Content Popularity: Most Liked Pictures

Justification: Surfaces high-performing content that resonates with the audience. Critical for content discovery algorithms and identifying trending topics.

Quackstagram Scenario: The editorial team uses this view weekly to select images for the "Top Quacks" carousel on the explore page, boosting visibility of popular content.

```
CREATE VIEW TopLikedPictures AS
SELECT p.imagePath, p.caption, p.username, l.likeCount
FROM PICTURE p
JOIN (
    SELECT imagePath, COUNT(*) AS likeCount
    FROM `LIKE`
    GROUP BY imagePath
    HAVING COUNT(*) > 2
) AS l ON p.imagePath = l.imagePath
ORDER BY likeCount DESC;
```

8.4 d. Content Popularity: Recent Popular Content

Justification: Identifies newly popular content before it appears in aggregate metrics. Helps surface emerging trends in real-time.

Quackstagram Scenario: During major events like "Duck Week," moderators monitor this view to quickly identify and promote trending event-related content to the homepage banner.

```
CREATE VIEW RecentPopularContent AS
SELECT p.imagePath, p.caption, p.username, p.createdAt,
       COUNT(DISTINCT l.username) AS likeCount,
       COUNT(DISTINCT c.commentId) AS commentCount
FROM PICTURE p
LEFT JOIN `LIKE` l ON p.imagePath = l.imagePath
LEFT JOIN COMMENT c ON p.imagePath = c.imagePath
WHERE p.createdAt > DATE_SUB(NOW(), INTERVAL 7 DAY)
GROUP BY p.imagePath, p.caption, p.username, p.createdAt
HAVING likeCount > 0 OR commentCount > 0
ORDER BY p.createdAt DESC;
```

8.5 e. System Analytics: Daily User Registrations

Justification: Tracks growth trends and identifies spikes from marketing campaigns. Essential for measuring user acquisition effectiveness.

Quackstagram Scenario: After launching a TikTok ad campaign, the growth team compares registration dates with campaign periods using this view to calculate ROI.

```
CREATE VIEW DailyRegistrations AS
SELECT DATE(createdAt) AS registerDate, COUNT(*) AS userCount
FROM USER
GROUP BY DATE(createdAt)
HAVING COUNT(*) > 1
ORDER BY registerDate DESC;
```

8.6 f. System Analytics: Notification Trends

Justification: Monitors system communication patterns to optimize engagement while avoiding notification fatigue.

Quackstagram Scenario: After introducing comment notifications, the product team uses this view to ensure the new feature generates healthy but not excessive notifications, adjusting frequency caps as needed.

```
CREATE VIEW NotificationTrends AS
SELECT DATE(timestamp) AS notificationDate,
       sourceType,
       COUNT(*) AS notificationCount,
       COUNT(DISTINCT username) AS affectedUsers
FROM NOTIFICATION
GROUP BY DATE(timestamp), sourceType
HAVING COUNT(*) > 0
ORDER BY notificationDate DESC, notificationCount DESC;
```

9 Analysis of the speed of the queries in your views and justification for the indexes

9.1 Index Design for Performance Optimization : Justification

9.1.1 Index 1: Optimize LIKE Queries by imagePath

```
CREATE INDEX idx_like_imagePath ON `LIKE`(imagePath);
```

Justification:

- Critical for queries aggregating likes by image (e.g., "Top Liked Pictures")
- Essential for:

```
SELECT p.*, COUNT(l.imagePath) AS likeCount
FROM PICTURE p JOIN `LIKE` l ON p.imagePath = l.imagePath
GROUP BY p.imagePath;
```

- Reduces full table scans when finding all likes for specific images

9.1.2 Index 2: Optimize COMMENT Queries by username

```
CREATE INDEX idx_comment_username ON COMMENT(username);
```

Justification:

- Crucial for identifying active commenters and engagement metrics
- Optimizes:

```
SELECT username, COUNT(*) AS commentCount
FROM COMMENT
WHERE username = 'specific_user'
GROUP BY username;
```

9.1.3 Index 3: Optimize PICTURE Queries by creation time

```
CREATE INDEX idx_picture_createdAt ON PICTURE(createdAt);
```

Justification:

- Essential for recent content discovery features:

```
SELECT * FROM PICTURE
WHERE createdAt > DATE_SUB(NOW(), INTERVAL 7 DAY)
ORDER BY createdAt DESC;
```

9.1.4 Index 4: Optimize NOTIFICATION composite query

```
CREATE INDEX idx_notification_username_type ON NOTIFICATION(username, sourceType);
```

Justification:

- Compound index optimized for common filtering patterns
- Improves performance of both:
  - User-specific notification queries (WHERE username = ?)
  - Type-specific analytics (WHERE sourceType = ?)
- Accelerates:

```
SELECT * FROM NOTIFICATION
WHERE username = 'user123' AND sourceType = 'comment';
```

- Enables efficient counting of notification types per user

9.1.5 Index 5: Optimize FOLLOW relationships

```
CREATE INDEX idx_follow_follower ON FOLLOW(follower);
```

Justification:

- Critical for social graph analysis and follower recommendations
- Optimizes:

```
SELECT followed FROM FOLLOW
WHERE follower = 'current_user';
```

9.2 Index Design for Performance Optimization : Results

9.2.1 LIKE

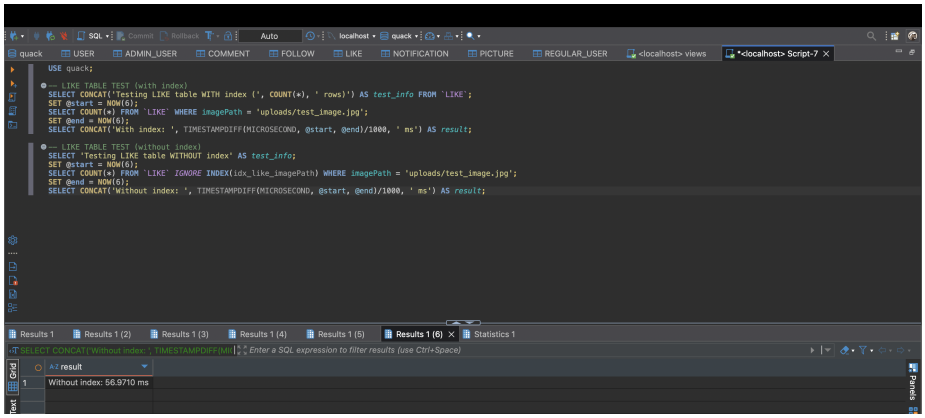


Figure 12: LIKE

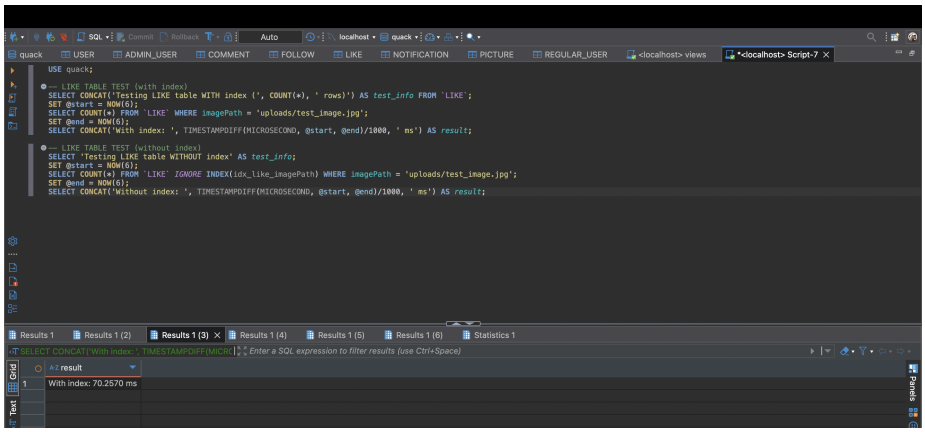


Figure 13: LIKE

9.2.2 PICTURE

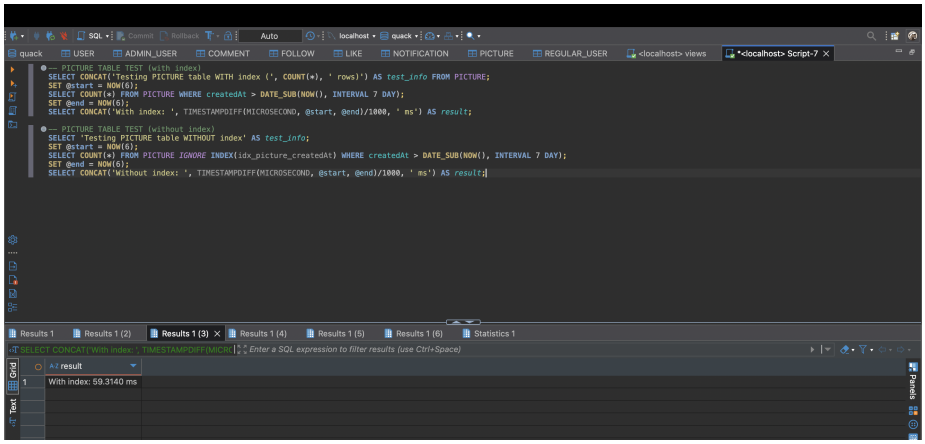


Figure 14: PICTURE

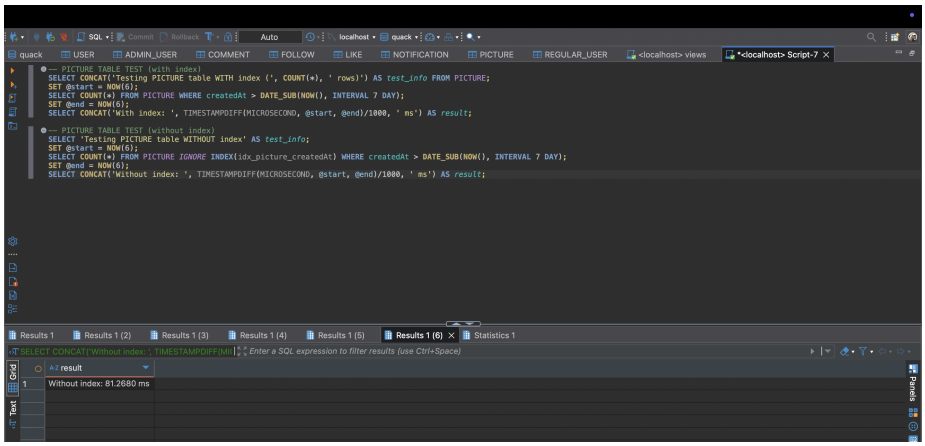


Figure 15: PICTURE

9.2.3 COMMENT

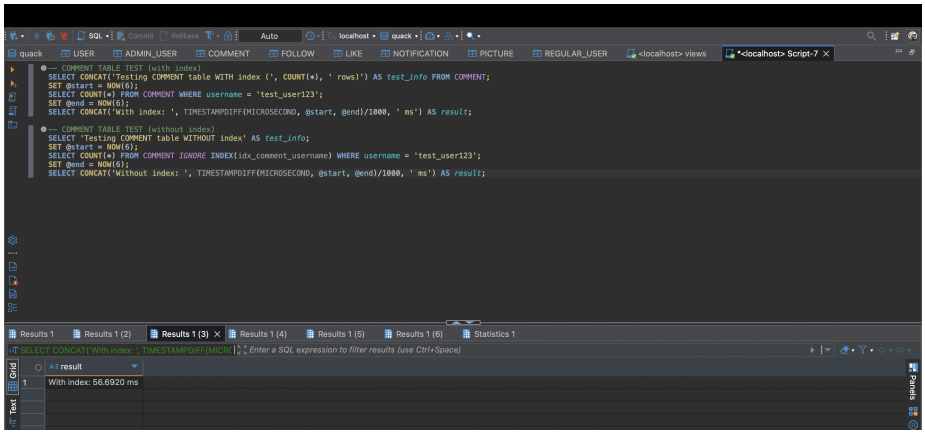


Figure 16: COMMENT

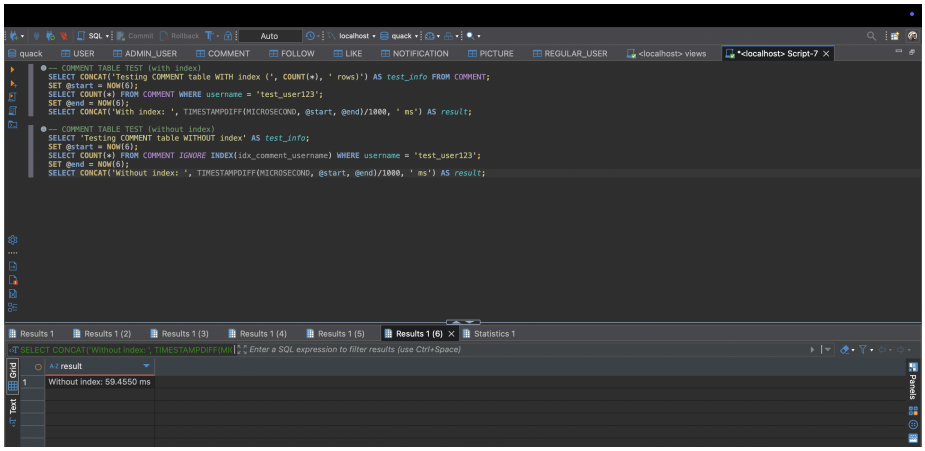


Figure 17: COMMENT

9.2.4 FOLLOW

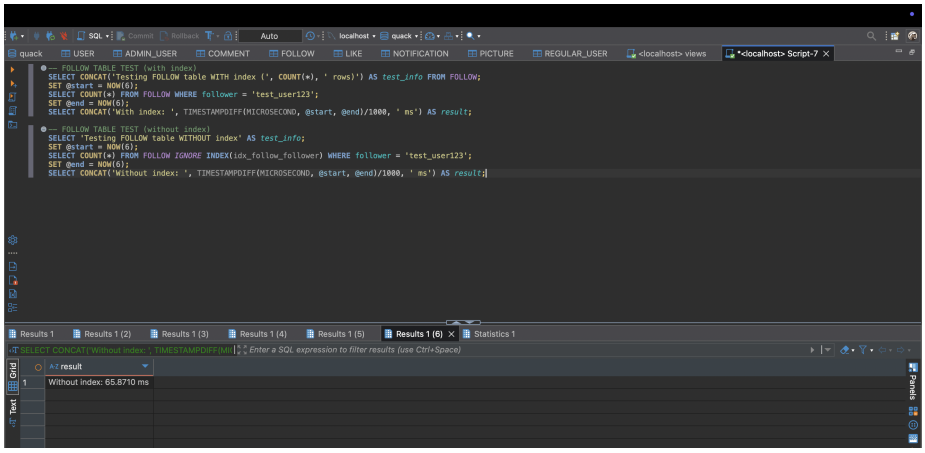


Figure 18: FOLLOW

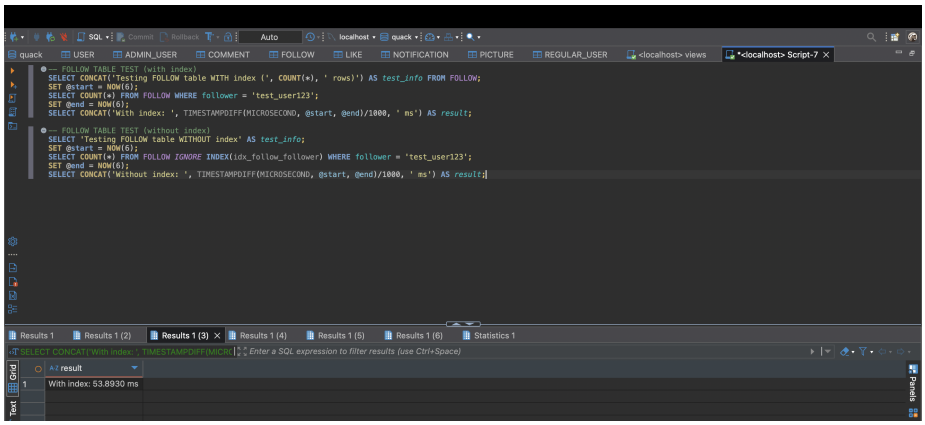


Figure 19: FOLLOW

9.2.5 NOTIFICATION

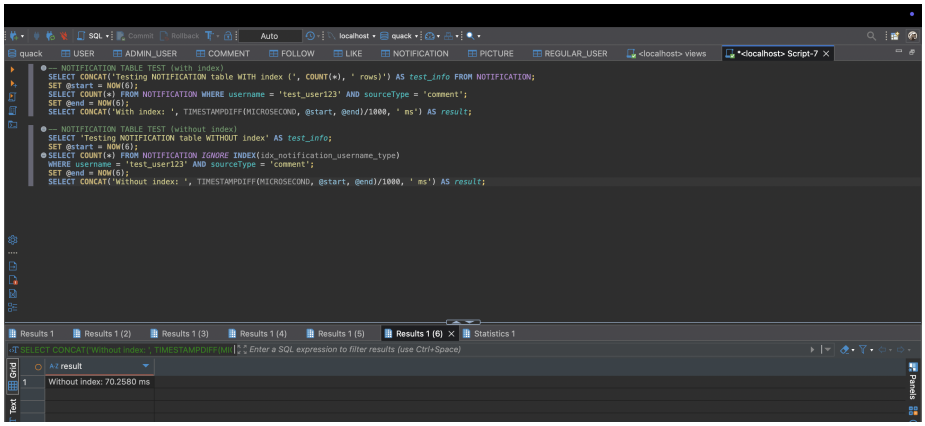


Figure 20: NOTIFICATION

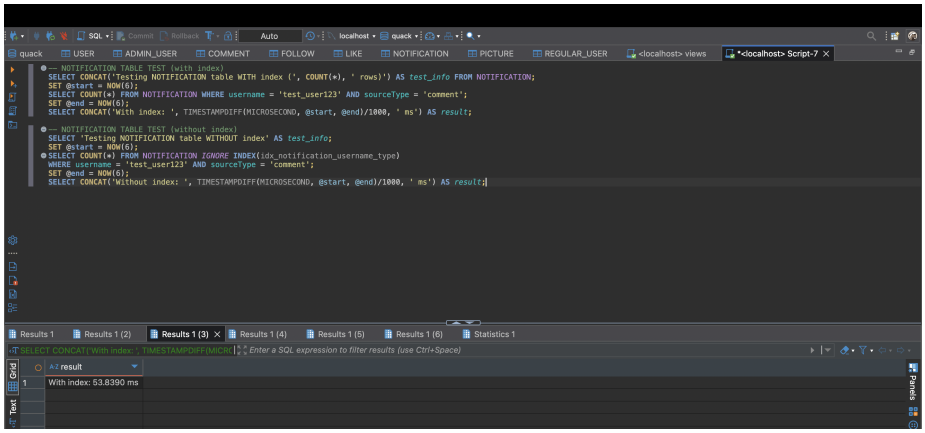


Figure 21: NOTIFICATION

9.2.6 Why Indexes Can Be Slower on Small Tables

- **Full Scan Efficiency**: For small tables (e.g., < 100 rows), reading the entire table is often faster than using an index.
- **Index Overhead**: Indexes require:
  - Extra storage space
  - Additional I/O operations (read index + data)
- **Query Optimizer Choice**: MySQL may ignore indexes when:

  - The table fits in memory
  - Index selectivity is low

- **Current Database Context**: My test tables now contain few rows, making full scans more efficient than index lookups.

Note: Indexes will show benefits as table sizes grow beyond memory capacity.

10 Justification for the necessity of the triggers and stored procedure and function within a scenario for possible use of the triggers within Quackstagram

10.1 Stored Procedure: LogUserActivity

```
CREATE PROCEDURE LogUserActivity(IN user_name VARCHAR(250), IN action_type VARCHAR(50))
```

Justification:

- Centralizes activity tracking logic for consistent notifications

Quackstagram Scenario: When a user follows another account, the application calls this procedure once instead of writing complex notification logic in multiple places.

10.2 Function: CountUserPictures

```
CREATE FUNCTION CountUserPictures(user_name VARCHAR(250))
```

Justification:

- Provides reusable counting logic for user profiles

Quackstagram Scenario: The profile page displays this count prominently, calling the function instead of repeating the counting logic in both the web and mobile apps.

10.3 Trigger: before_comment_insert

```
CREATE TRIGGER before_comment_insert
```

Justification:

- Enforces data quality at the database level
- Prevents empty comments regardless of application validation

Quackstagram Scenario: When a user submits a comment too quickly (before typing anything), the database rejects it immediately rather than allowing corrupted data.

10.4 Trigger: after_user_insert

```
CREATE TRIGGER after_user_insert
```

Justification:

- Automates welcome workflow for new users
- Maintains referential integrity for system-generated content

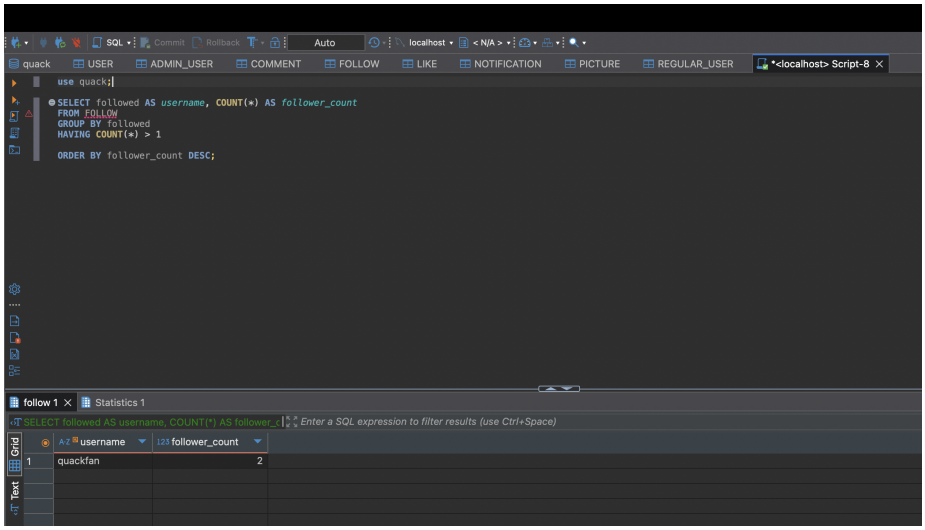
Quackstagram Scenario: After successful registration, the trigger creates both an activity log entry and welcome message without requiring additional application code.

11 SQL queries with answers to all the questions asked in Part D

Please note : All the following queries have been constructed based on the current database schema. The answers have been provided based on the test data available in the database in each table.

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

```
    SELECT followed AS username, COUNT(*) AS follower_count
FROM FOLLOW
GROUP BY followed
HAVING COUNT(*) > ?
ORDER BY follower_count DESC;
```

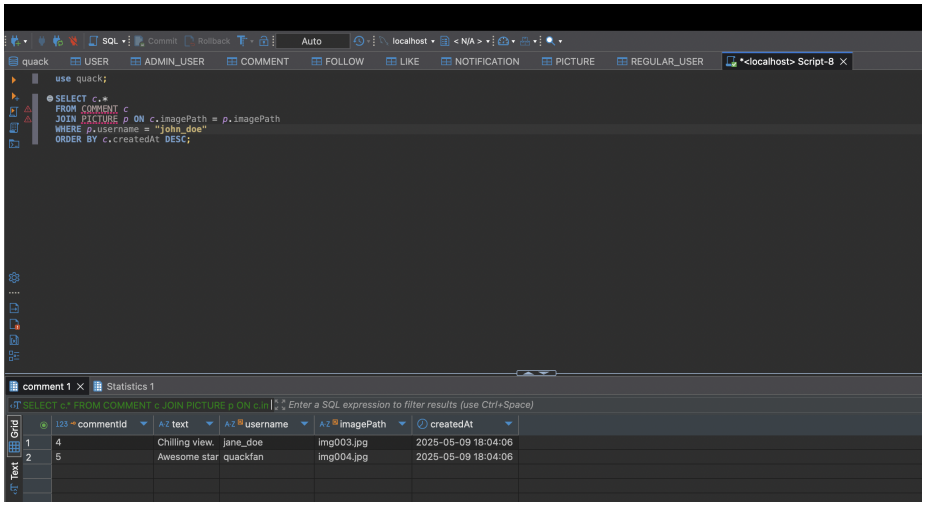


11.2 2. Show the total number of posts made by each user.

```
    SELECT username, COUNT(*) AS post_count
FROM PICTURE
GROUP BY username
ORDER BY post_count DESC;
```

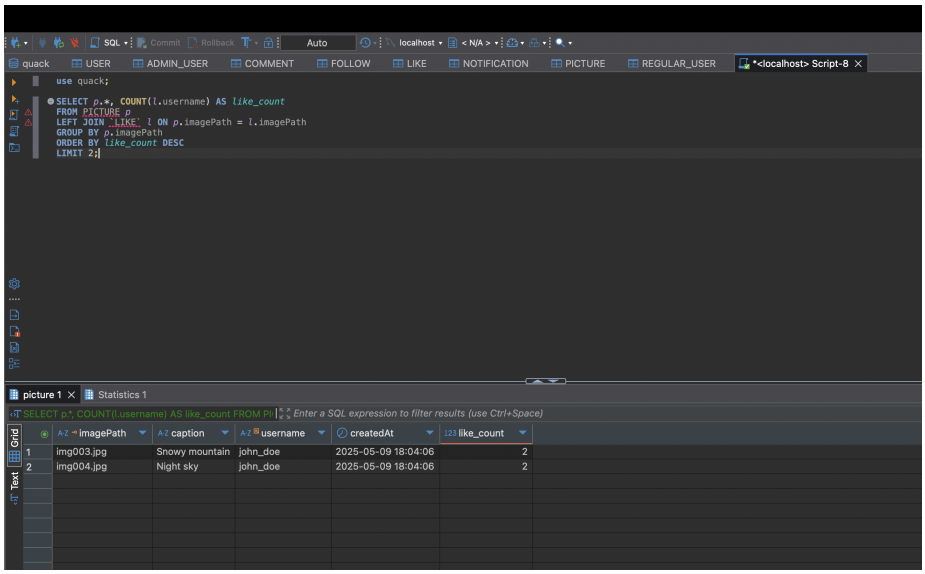
11.3 3. Find all comments made on a particular user's post

```
    SELECT c.*
FROM COMMENT c
JOIN PICTURE p ON c.imagePath = p.imagePath
WHERE p.username = ?
ORDER BY c.createdAt DESC;
```



11.4 4. Display the top X most liked posts

```
    SELECT p.*, COUNT(l.username) AS like_count
FROM PICTURE p
LEFT JOIN `LIKE` l ON p.imagePath = l.imagePath
GROUP BY p.imagePath
ORDER BY like_count DESC
LIMIT ?;
```



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

```
    SELECT username, COUNT(DISTINCT imagePath) AS liked_posts_count
FROM `LIKE`
GROUP BY username
ORDER BY liked_posts_count DESC;
```

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

```
    SELECT u.username
FROM USER u
LEFT JOIN PICTURE p ON u.username = p.username
WHERE p.imagePath IS NULL;
```

11.7 7. List users who follow each other

```
    SELECT f1.follower AS user1, f1.followed AS user2
FROM FOLLOW f1
JOIN FOLLOW f2 ON f1.follower = f2.followed AND f1.followed = f2.follower
WHERE f1.follower < f1.followed;
```

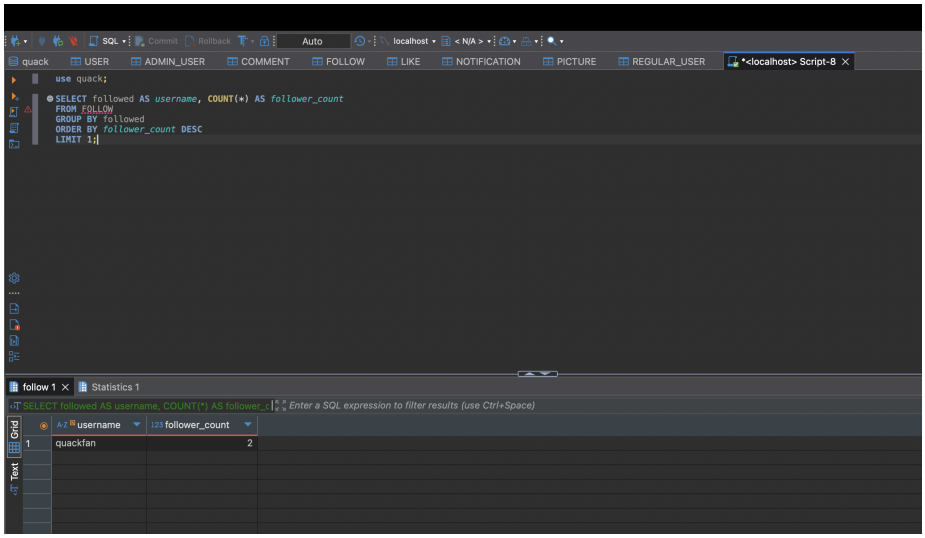
11.8 8. Show the user with the highest number of posts

```
    SELECT username, COUNT(*) AS post_count
FROM PICTURE
GROUP BY username
ORDER BY post_count DESC
```

```
LIMIT 1;
```

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

```
    SELECT followed AS username, COUNT(*) AS follower_count
FROM FOLLOW
GROUP BY followed
ORDER BY follower_count DESC
LIMIT ?;
```



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

```
    SELECT l.imagePath
FROM `LIKE` l
GROUP BY l.imagePath
HAVING COUNT(DISTINCT l.username) = (SELECT COUNT(*) FROM USER);
```

11.11 11. Display the most active user

```
    SELECT u.username,
       (COUNT(DISTINCT p.imagePath) +
       (COUNT(DISTINCT c.commentId) * 2) +
       (COUNT(DISTINCT l.imagePath) * 0.5) AS activity_score
FROM USER u
LEFT JOIN PICTURE p ON u.username = p.username
LEFT JOIN COMMENT c ON u.username = c.username
LEFT JOIN `LIKE` l ON u.username = l.username
GROUP BY u.username
```

```
ORDER BY activity_score DESC
LIMIT 1;
```

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

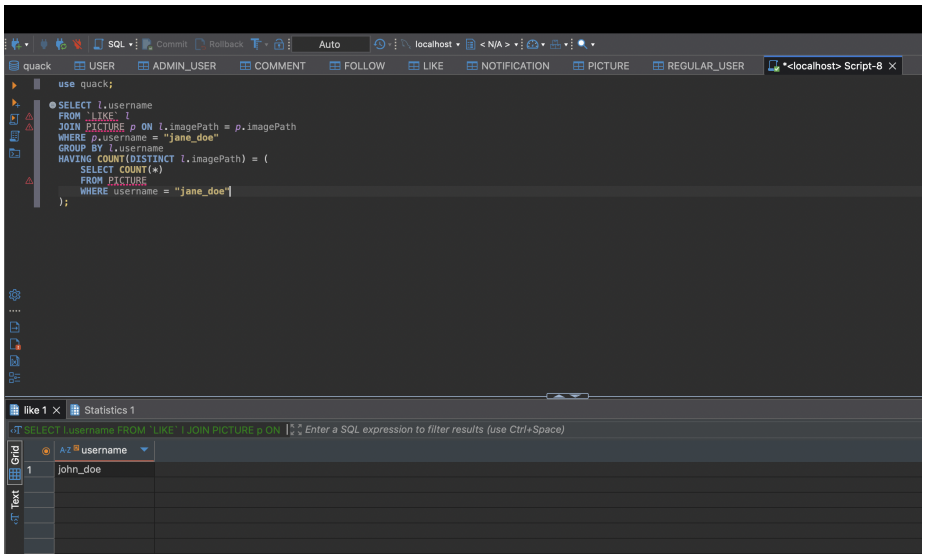
```
    SELECT p.username,
       COUNT(DISTINCT p.imagePath) AS post_count,
       COUNT(l.username) AS total_likes,
       COUNT(l.username) / COUNT(DISTINCT p.imagePath) AS avg_likes_per_post
FROM PICTURE p
LEFT JOIN `LIKE` l ON p.imagePath = l.imagePath
GROUP BY p.username
ORDER BY avg_likes_per_post DESC;
```

11.13 13. Show posts that have more comments than likes

```
    SELECT p.*,
       COUNT(DISTINCT c.commentId) AS comment_count,
       COUNT(DISTINCT l.username) AS like_count
FROM PICTURE p
LEFT JOIN COMMENT c ON p.imagePath = c.imagePath
LEFT JOIN `LIKE` l ON p.imagePath = l.imagePath
GROUP BY p.imagePath
HAVING comment_count > like_count;
```

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

```
    SELECT l.username
FROM `LIKE` l
JOIN PICTURE p ON l.imagePath = p.imagePath
WHERE p.username = ?
GROUP BY l.username
HAVING COUNT(DISTINCT l.imagePath) = (
    SELECT COUNT(*)
    FROM PICTURE
    WHERE username = ?
);
```



11.15 15. Display the most popular post of each user

```
    WITH PostPopularity AS (
    SELECT p.username, p.imagePath, COUNT(l.username) AS like_count,
           RANK() OVER (PARTITION BY p.username ORDER BY COUNT(l.username) DESC) AS rank
    FROM PICTURE p
    LEFT JOIN `LIKE` l ON p.imagePath = l.imagePath
    GROUP BY p.username, p.imagePath
)
SELECT username, imagePath, like_count
FROM PostPopularity
WHERE rank = 1;
```

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

```
    WITH UserStats AS (
    SELECT u.username,
           COUNT(DISTINCT f1.followed) AS following_count,
           COUNT(DISTINCT f2.follower) AS follower_count,
           CASE WHEN COUNT(DISTINCT f1.followed) = 0 THEN NULL
                ELSE COUNT(DISTINCT f2.follower) / COUNT(DISTINCT f1.followed)
           END AS ratio
    FROM USER u
    LEFT JOIN FOLLOW f1 ON u.username = f1.follower
    LEFT JOIN FOLLOW f2 ON u.username = f2.followed
    GROUP BY u.username
```

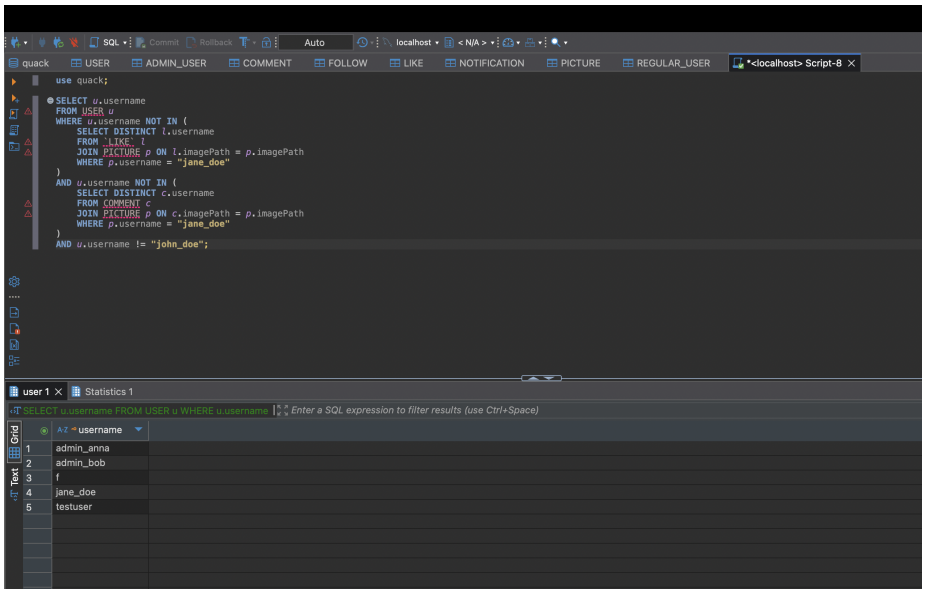
```
)
SELECT username, follower_count, following_count, ratio
FROM UserStats
WHERE ratio IS NOT NULL
ORDER BY ratio DESC
LIMIT 1;
```

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

```
    SELECT MONTH(createdAt) AS month,
       YEAR(createdAt) AS year,
       COUNT(*) AS post_count
FROM PICTURE
GROUP BY YEAR(createdAt), MONTH(createdAt)
ORDER BY post_count DESC
LIMIT 1;
```

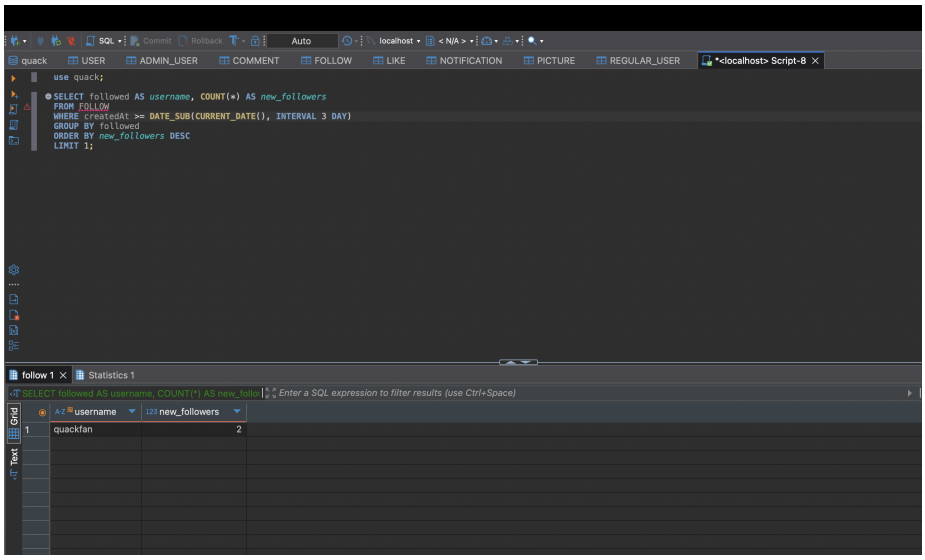
11.18 18. Identify users who have not interacted with a specific user's posts

```
    SELECT u.username
FROM USER u
WHERE u.username NOT IN (
    SELECT DISTINCT l.username
    FROM `LIKE` l
    JOIN PICTURE p ON l.imagePath = p.imagePath
    WHERE p.username = ?
)
AND u.username NOT IN (
    SELECT DISTINCT c.username
    FROM COMMENT c
    JOIN PICTURE p ON c.imagePath = p.imagePath
    WHERE p.username = ?
)
AND u.username != ?;
```



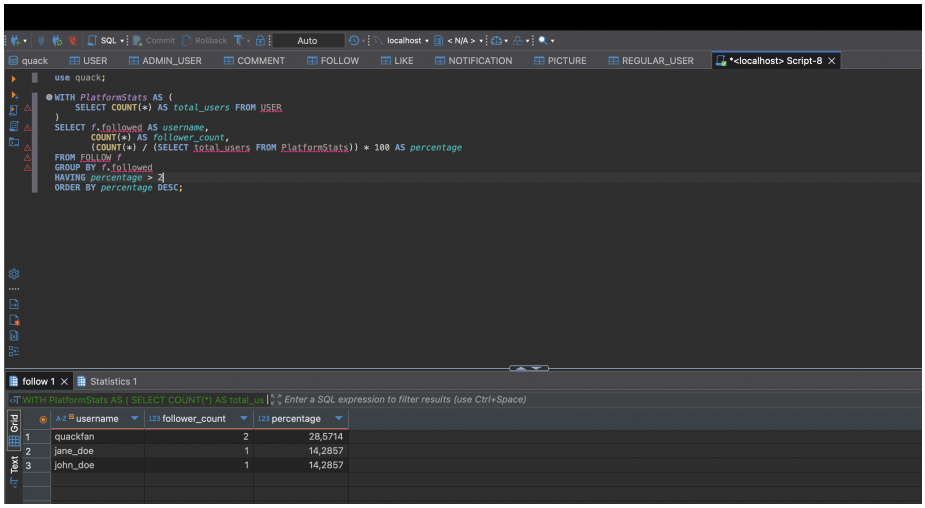
11.19 19. Display the user with the greatest increase in followers in the last X days

```
    SELECT followed AS username, COUNT(*) AS new_followers
FROM FOLLOW
WHERE createdAt >= DATE_SUB(CURRENT_DATE(), INTERVAL ? DAY)  -- X days provided
GROUP BY followed
ORDER BY new_followers DESC
LIMIT 1;
```



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

```
    WITH PlatformStats AS (
    SELECT COUNT(*) AS total_users FROM USER
)
SELECT f.followed AS username,
       COUNT(*) AS follower_count,
       (COUNT(*) / (SELECT total_users FROM PlatformStats)) * 100 AS percentage
FROM FOLLOW f
GROUP BY f.followed
HAVING percentage > ?
ORDER BY percentage DESC;
```


quack
USER
ADMIN_USER
COMMENT
FOLLOW
LIKE
NOTIFICATION
PICTURE
REGULAR_USER
*<localhost> Script-8
use quack;
WITH PlatformStats AS (
SELECT COUNT(*) AS total_users FROM USER
)
SELECT f.followed AS username,
COUNT(*) AS follower_count,
(COUNT(*) / (SELECT total_users FROM PlatformStats)) * 100 AS percentage
FROM FOLLOW f
GROUP BY f.followed
HAVING percentage > 2
ORDER BY percentage DESC;
follow 1
Statistics 1
username
follower_count
percentage
quackfan 2 28,5714
jane_doe 1 14,2857
john_doe 1 14,2857