Maastricht University

Department of Advanced Computing Sciences

BCS1510 - Databases

Quackstagram - A Social Media Platform with SQL Backend

Professor :
Dr. Katharina Schneider
Dr. Tony Garnock-Jones

Grégoire MARIETTE
I6385935 BSc Computer Science, First Year

Academic Year 2024 - 2025

Contents

1 Introduction

I built this project as an enhancement to my Quackstagram application originally developed for the Object Oriented Modelling course. Having achieved a score of 29/30 on the OOM project, I had a solid foundation to build upon. This project focuses on migrating from file-based storage to a relational database to improve scalability, data integrity, and query capabilities.

2 Quackstagram Database Overview

The Quackstagram database serves as the data repository for an Instagram-inspired application. This MySQL system replaces the original file-based storage with a relational model consisting of seven tables: Users, Pictures, Follows, Notifications, Likes, Comments, and FollowerHistory.

The database design follows 3NF normalization principles while maintaining performance through well-defined relationships and strategic indexing. The implementation preserves all the application's original functionalities.

2.1 Table Structure

The database consists of the following tables:

- **Users**: Stores authentication data (username, passwordHash, salt), profile information (bio, profileImagePath)
- **Pictures**: Contains image metadata (imageId, imagePath), content information (caption), and timestamps
- **Follows**: Manages follower/following relationships between users with timestamps
- **Notifications**: Records interactions between users (likes, comments, follows)
- **Likes**: Tracks which users have liked which pictures
- **Comments**: Stores user comments on pictures with content and timestamps
- **FollowerHistory**: Track follower statistics for analysis

2.2 Database Integration

To integrate the database with the Quackstagram application, I implemented JDBC-based data access objects (DAOs) that replace file I/O operations with SQL queries.

1. A database connection manager to handle database connections
2. DAO implementations for each entity type (User, Picture, Notification, Follow)
3. Prepared SQL statement for increase security
4. Database exception handling
5. SQL views for analytical queries

2.2.1 Database Connection Management

The connection manager implements a simple singleton pattern:

Listing 1: Database Connection Manager

```java
public class DatabaseConnectionManager {
    private static DatabaseConnectionManager instance;
    private String jdbcUrl;
    private String username;
    private String password;

    private DatabaseConnectionManager() {
        loadConfiguration();
    }

    public static synchronized DatabaseConnectionManager getInstance() {
        if (instance == null) {
            instance = new DatabaseConnectionManager();
        }
        return instance;
    }

    private void loadConfiguration() {
        try {
            Properties props = new Properties();
            props.load(new FileInputStream("config/database.properties"));
            jdbcUrl = props.getProperty("jdbc.url");
            username = props.getProperty("jdbc.username");
            password = props.getProperty("jdbc.password");

            // Load driver
            Class.forName("com.mysql.cj.jdbc.Driver");
        } catch (Exception e) {
            System.err.println("Error loading configuration: " + e.getMessage());
            // Default values
            jdbcUrl = "jdbc:mysql://localhost:3306/quackstagram?useSSL=false";
            username = "user";
            password = "pass";
        }
    }

    public Connection getConnection() throws SQLException {
        return DriverManager.getConnection(jdbcUrl, username, password);
    }
}
```

2.2.2 DAO Implementation Example

The DAO implementations maintain the original interface while switching from file operations to database queries:

Listing 2: User DAO Implementation

```java
public class DatabaseUserDAO implements UserDAO {
    private final DatabaseConnectionManager connectionManager;

    public DatabaseUserDAO() {
        this.connectionManager = DatabaseConnectionManager.getInstance();
    }

    @Override
    public User findByUsername(String username) {
        try (Connection conn = connectionManager.getConnection();
             PreparedStatement stmt = conn.prepareStatement("SELECT * FROM Users
                 WHERE username = ?")) {

            stmt.setString(1, username);
            ResultSet rs = stmt.executeQuery();

            if (rs.next()) {
                String bio = rs.getString("bio");
                String passwordHash = rs.getString("passwordHash");
                String salt = rs.getString("salt");

                User user = new User(username, bio, passwordHash, salt);

                // Load additional statistics
                updateUserStats(conn, user);

                return user;
            }
            return null;
        } catch (SQLException e) {
            System.err.println("Error finding user: " + e.getMessage());
            return null;
        }
    }

    private void updateUserStats(Connection conn, User user) throws SQLException {
        // Count posts, followers, and following with separate queries
        try (PreparedStatement countStmt = conn.prepareStatement(
                "SELECT COUNT(*) FROM Pictures WHERE username = ?")) {
            countStmt.setString(1, user.getUsername());
            ResultSet countRs = countStmt.executeQuery();
            if (countRs.next()) {
                user.setPostCount(countRs.getInt(1));
            }
        }

        // Similar implementations for followers and following counts
    }
}
```

2.2.3 Database Views and Triggers

views for analytical queries and triggers for maintaining data consistency:

Listing 3: Database Views

```
CREATE VIEW user_engagement AS
SELECT
    u.username,
    (SELECT COUNT(*) FROM Pictures WHERE username = u.username) AS post_count,
    (SELECT COUNT(*) FROM Likes WHERE username = u.username) AS likes_given,
    (SELECT COUNT(*) FROM Comments WHERE username = u.username) AS comments_made,
    (SELECT COUNT(*) FROM Follows WHERE follower = u.username) AS following_count
FROM
    Users u
GROUP BY
    u.username
HAVING
    post_count > 0 OR likes_given > 0;
```

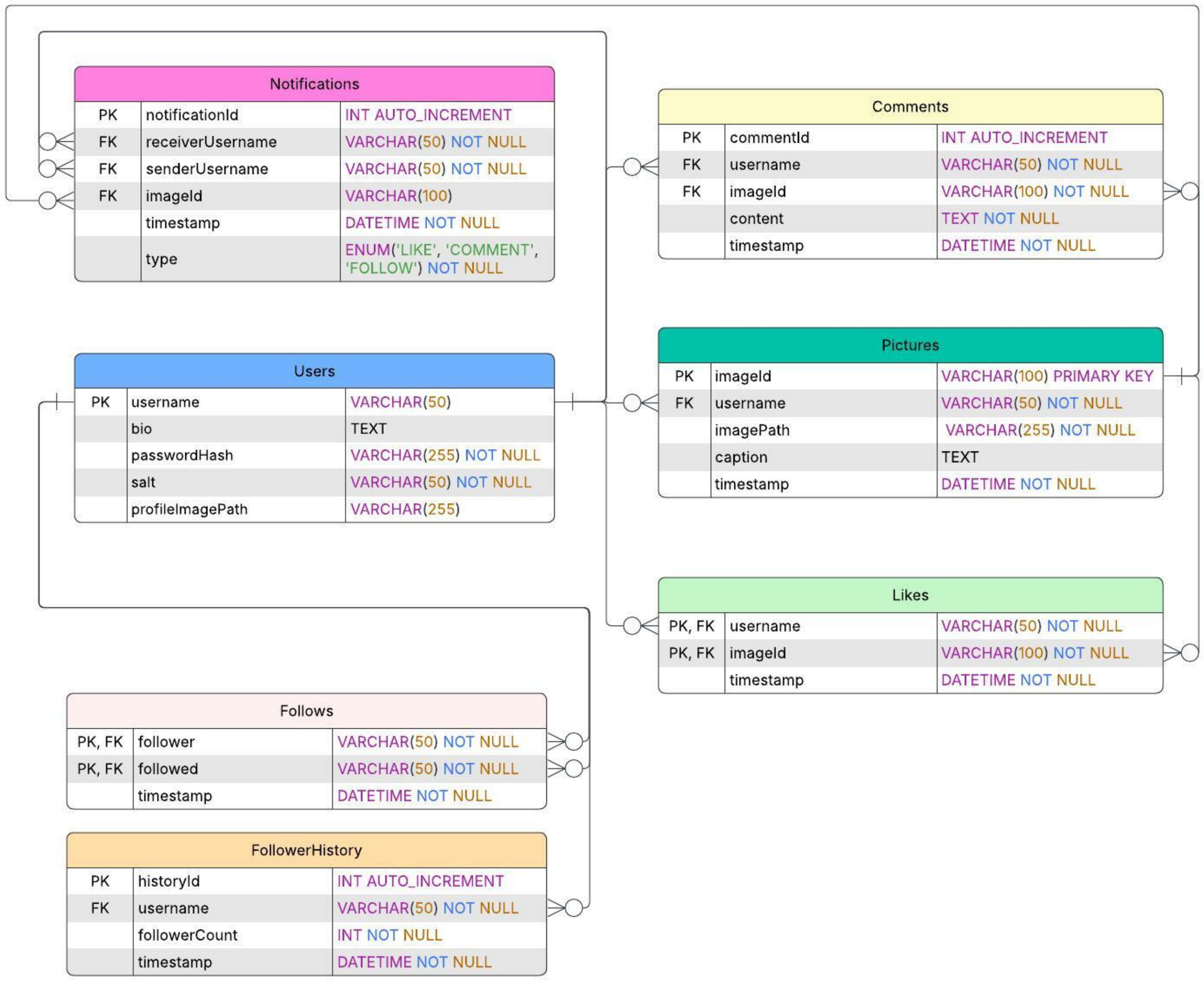
Listing 4: Database Triggers

```
CREATE TRIGGER after_like_insert
AFTER INSERT ON Likes
FOR EACH ROW
BEGIN
    -- Get the username of the picture owner
    DECLARE picture_owner VARCHAR(50);
    SELECT username INTO picture_owner
    FROM Pictures
    WHERE imageId = NEW.imageId;

    -- Create notification using the stored procedure
    CALL create_notification(picture_owner, NEW.username, NEW.imageId, 'LIKE');
END
```

By doing all these changes, I maintaining the same user experience and application structure while improving the data integrity and security, more efficient queries, and enhanced analytics capabilities.

3 Entity-relationship diagram



Notifications		
PK	notificationId	INT AUTO_INCREMENT
FK	receiverUsername	VARCHAR(50) NOT NULL
FK	senderUsername	VARCHAR(50) NOT NULL
FK	imageId	VARCHAR(100)
	timestamp	DATETIME NOT NULL
	type	ENUM('LIKE', 'COMMENT', 'FOLLOW') NOT NULL

Comments		
PK	commentId	INT AUTO_INCREMENT
FK	username	VARCHAR(50) NOT NULL
FK	imageId	VARCHAR(100) NOT NULL
	content	TEXT NOT NULL
	timestamp	DATETIME NOT NULL

Users		
PK	username	VARCHAR(50)
	bio	TEXT
	passwordHash	VARCHAR(255) NOT NULL
	salt	VARCHAR(50) NOT NULL
	profileImagePath	VARCHAR(255)

Pictures		
PK	imageId	VARCHAR(100) PRIMARY KEY
FK	username	VARCHAR(50) NOT NULL
	imagePath	VARCHAR(255) NOT NULL
	caption	TEXT
	timestamp	DATETIME NOT NULL

Likes		
PK, FK	username	VARCHAR(50) NOT NULL
PK, FK	imageId	VARCHAR(100) NOT NULL
	timestamp	DATETIME NOT NULL

Follows		
PK, FK	follower	VARCHAR(50) NOT NULL
PK, FK	followed	VARCHAR(50) NOT NULL
	timestamp	DATETIME NOT NULL

FollowerHistory		
PK	historyId	INT AUTO_INCREMENT
FK	username	VARCHAR(50) NOT NULL
	followerCount	INT NOT NULL
	timestamp	DATETIME NOT NULL

Figure 1: Entity-relationship diagram

4 Database Table Examples with Sample Data

This section presents examples of each table in the Quackstagram database with sample data. Primary key are **highlighted in bold**.

4.1 Users Table

The Users table stores account information for all registered users on the platform.

username	bio	passwordHash	salt	profileImagePath
Xylo	Fierce warrior, not solo	Ywp+k...	J9F...	img/storage/profile/Xylo.png
Lorin	For copyright reasons, I am not Grogu	p+8+H...	DS...	img/storage/profile/Lorin.png
Zara	Humanoid robot much like the rest	bPTzvv...	/W...	img/storage/profile/Zara.png
Mystar	Xylo and I are not the same!	YDVdIu...	O4F+...	img/storage/profile/Mystar.png

Table 1: Sample data from the Users table

4.2 Pictures Table

The Pictures table contains all images posted by users with associated metadata.

imageId	username	caption	imagePath	timestamp
Xylo_1	Xylo	My tea strong as Force is.	img/uploaded/Xylo_1.png	2023-12-17 19:22:40
Lorin_1	Lorin	In the cookie jar my hand was not.	img/uploaded/Lorin_1.png	2023-12-17 19:07:43
Zara_1	Zara	Lost my map I have. Oops.	img/uploaded/Zara_1.png	2023-12-17 19:24:31
Mystar_1	Mystar	Cookies gone?	img/uploaded/Mystar_1.png	2023-12-17 19:26:50
Lorin_2	Lorin	Meditate I must.	img/uploaded/Lorin_2.png	2023-12-17 19:09:35

Table 2: Sample data from the Pictures table

4.3 Follows Table

The Follows table tracks social connections between users.

follower	**followed**	timestamp
Xylo	Lorin	2023-12-17 18:30:00
Xylo	Mystar	2023-12-17 18:31:00
Xylo	Zara	2023-12-17 18:35:00
Zara	Lorin	2023-12-17 18:32:00
Zara	Xylo	2023-12-17 18:33:00
Mystar	Lorin	2023-12-17 18:34:00
Mystar	Zara	2023-12-17 18:35:00
Mystar	Xylo	2023-12-17 18:36:00
Lorin	Mystar	2023-12-17 18:37:00
Lorin	Zara	2023-12-17 18:40:00

Table 3: Sample data from the Follows table with composite primary key of follower and followed

4.4 Notifications Table

The Notifications table stores alerts about user interactions.

notificationId	receiverUsername	senderUsername	imageId	timestamp	type
1	Lorin	Zara	Lorin_1	2023-12-17 19:29:41	LIKE
2	Lorin	Xylo	Lorin_1	2025-03-14 17:26:08	LIKE
3	Lorin	Xylo	Lorin_2	2025-03-14 17:26:11	LIKE
4	Mystar	Xylo	Mystar_2	2025-03-14 17:34:34	LIKE
5	Lorin	Xylo	NULL	2023-12-17 18:30:00	FOLLOW
6	Xylo	Zara	Xylo_2	2025-03-16 01:45:19	LIKE

Table 4: Sample data from the Notifications table

4.5 Likes Table

The Likes table records which users have liked which pictures.

username	**imageId**	timestamp
Xylo	Lorin_1	2023-12-17 19:40:00
Xylo	Lorin_2	2023-12-17 19:41:00
Xylo	Mystar_1	2023-12-17 19:49:00
Zara	Lorin_1	2023-12-17 19:42:00
Lorin	Xylo_1	2023-12-17 19:44:00
Lorin	Zara_1	2023-12-17 19:53:00
Mystar	Xylo_2	2023-12-17 19:46:00
Zara	Xylo_2	2023-12-17 19:47:00

Table 5: Sample data from the Likes table with composite primary key

4.6 Comments Table

The Comments table stores user-generated text responses to pictures.

commentId	username	imageId	content	timestamp
1	Xylo	Lorin_1	Very wise, you are.	2023-12-17 20:00:00
2	Zara	Lorin_1	LOL reminds me of someone!	2023-12-17 20:01:00
3	Mystar	Lorin_2	Teach me your ways	2023-12-17 20:02:00
4	Lorin	Xylo_1	Recipe, can you share?	2023-12-17 20:03:00
5	Mystar	Xylo_2	These are not the droids you are looking for	2023-12-17 20:04:00

Table 6: Sample data from the Comments table

4.7 FollowerHistory Table

The FollowerHistory table tracks changes in user follower counts over time.

historyId	username	followerCount	timestamp
1	Lorin	1	2023-12-17 18:30:00
2	Lorin	2	2023-12-17 18:32:00
3	Lorin	3	2023-12-17 18:34:00
4	Xylo	1	2023-12-17 18:33:00
5	Xylo	2	2023-12-17 18:36:00
6	Mystar	1	2023-12-17 18:37:00
7	Zara	1	2023-12-17 18:35:00

Table 7: Sample data from the FollowerHistory table

5 Functional Dependencies for Each Table

This section outlines the functional dependencies present in each table of the Quackstagram database.

5.1 Users Table

The Users table has the following functional dependency:

- `username` $\rightarrow$ {`bio`, `passwordHash`, `salt`, `profileImagePath`}

The username is the primary key and uniquely determines all other attributes in this table. Each user has exactly one bio, one password hash, one salt value, and one profile image path.

5.2 Pictures Table

The Pictures table has the following functional dependency:

- `imageId` $\rightarrow$ {`username`, `imagePath`, `caption`, `timestamp`}

The imageId is the primary key and uniquely determines all other attributes in this table. Each picture belongs to exactly one user, has one file path, one caption, and one posting timestamp.

5.3 Follows Table

The Follows table has the following functional dependency:

- `(follower, followed)` $\rightarrow$ {`timestamp`}

The composite primary key of follower and followed uniquely determines the timestamp. Each unique follower-followed pair can have only one timestamp when the follow action occurred.

5.4 Notifications Table

The Notifications table has the following functional dependency:

- `notificationId` $\rightarrow$ {`receiverUsername`, `senderUsername`, `imageId`, `timestamp`, `type`}

The notificationId is the primary key and uniquely determines all other attributes in this table. Each notification has exactly one receiver, one sender, potentially one related image (or null), one timestamp, and one type classification.

5.5 Likes Table

The Likes table has the following functional dependency:

- `(username, imageId)` $\rightarrow$ {`timestamp`}

The composite primary key of username and imageId uniquely determines the timestamp. This enforces that each user can like a specific image only once, with the timestamp recording when that like occurred.

5.6 Comments Table

The Comments table has the following functional dependency:

- `commentId` $\rightarrow$ {`username`, `imageId`, `content`, `timestamp`}

The commentId is the primary key and uniquely determines all other attributes in this table. Each comment is made by exactly one user, relates to exactly one image, has one content text, and one timestamp.

5.7 FollowerHistory Table

The FollowerHistory table has the following functional dependencies:

- `historyId` $\rightarrow$ {`username`, `followerCount`, `timestamp`}
- `(username, timestamp)` $\rightarrow$ {`followerCount`}

The historyId is the primary key and uniquely determines all other attributes. Additionally, the combination of username and timestamp functionally determines the follower count, as there can only be one record of a user's follower count at any specific point in time.

6 Proof of Third Normal Form (3NF) for Each Table

This section provides formal proofs that each table in the Quackstagram database is in the Third Normal Form (3NF). A table is in 3NF if it satisfies all of the following conditions:

1. The table is in First Normal Form (1NF): All attributes contain atomic values.
2. The table is in Second Normal Form (2NF): All non-key attributes are fully functionally dependent on the primary key.
3. Every non-key attribute is non-transitively dependent on the primary key (i.e., there are no transitive dependencies).

6.1 Users Table

Proof for 1NF: All attributes in the Users table contain atomic values. The username, bio, passwordHash, salt, and profileImagePath fields each contain a single value per record.

Proof for 2NF: The primary key is `username`. Since this is a single-attribute key, there are no partial dependencies possible, and the table automatically satisfies 2NF.

Proof for 3NF: All non-key attributes (bio, passwordHash, salt, profileImagePath) depend directly on the primary key `username` and not on any other non-key attribute. There are no transitive dependencies where one non-key attribute determines another. Each attribute represents a distinct property of the user entity.

Conclusion: The Users table is in 3NF.

6.2 Pictures Table

Proof for 1NF: All attributes in the Pictures table contain atomic values. The imageId, username, imagePath, caption, and timestamp fields each contain a single value per record.

Proof for 2NF: The primary key is `imageId`. Since this is a single-attribute key, there are no partial dependencies possible, and the table automatically satisfies 2NF.

Proof for 3NF: All non-key attributes (username, imagePath, caption, timestamp) depend directly on the primary key `imageId` and not on any other non-key attribute. While the username is a foreign key to the Users table, it is a property of the picture entity and directly determined by imageId.

Conclusion: The Pictures table is in 3NF.

6.3 Follows Table

Proof for 1NF: All attributes in the Follows table contain atomic values. The follower, followed, and timestamp fields each contain a single value per record.

Proof for 2NF: The primary key is the composite key (`follower`, `followed`). The only non-key attribute is timestamp, which depends on the entire composite key. A specific follower-followed pair uniquely determines the follow timestamp.

Proof for 3NF: The only non-key attribute (timestamp) depends directly on the composite primary key (`follower`, `followed`) and not on any other attribute. Since there are no other non-key attributes, there cannot be any transitive dependencies.

Conclusion: The Follows table is in 3NF.

6.4 Notifications Table

Proof for 1NF: All attributes in the Notifications table contain atomic values. The notificationId, receiverUsername, senderUsername, imageId, timestamp, and type fields each contain a single value per record.

Proof for 2NF: The primary key is `notificationId`. Since this is a single-attribute key, there are no partial dependencies possible, and the table automatically satisfies 2NF.

Proof for 3NF: All non-key attributes (receiverUsername, senderUsername, imageId, timestamp, type) depend directly on the primary key `notificationId` and not on any other non-key attribute. While receiverUsername, senderUsername, and imageId are foreign keys to other tables, they are properties of the notification entity and directly determined by notificationId.

Conclusion: The Notifications table is in 3NF.

6.5 Likes Table

Proof for 1NF: All attributes in the Likes table contain atomic values. The username, imageId, and timestamp fields each contain a single value per record.

Proof for 2NF: The primary key is the composite key (`username`, `imageId`). The only non-key attribute is timestamp, which depends on the entire composite key. A specific username-imageId pair uniquely determines when the like occurred.

Proof for 3NF: The only non-key attribute (timestamp) depends directly on the composite primary key (`username`, `imageId`) and not on any other attribute. Since there are no other non-key attributes, there cannot be any transitive dependencies.

Conclusion: The Likes table is in 3NF.

6.6 Comments Table

Proof for 1NF: All attributes in the Comments table contain atomic values. The commentId, username, imageId, content, and timestamp fields each contain a single value per record.

Proof for 2NF: The primary key is `commentId`. Since this is a single-attribute key, there are no partial dependencies possible, and the table automatically satisfies 2NF.

Proof for 3NF: All non-key attributes (username, imageId, content, timestamp) depend directly on the primary key `commentId` and not on any other non-key attribute. While username and imageId are foreign keys to other tables, they are properties of the comment entity and directly determined by commentId.

Conclusion: The Comments table is in 3NF.

6.7 FollowerHistory Table

Proof for 1NF: All attributes in the FollowerHistory table contain atomic values. The historyId, username, followerCount, and timestamp fields each contain a single value per record.

Proof for 2NF: The primary key is `historyId`. Since this is a single-attribute key, there are no partial dependencies possible, and the table automatically satisfies 2NF.

Proof for 3NF: All non-key attributes (username, followerCount, timestamp) depend directly on the primary key `historyId` and not on any other non-key attribute. While there is a natural business relationship between username and followerCount, in this table design they are both dependent on the historyId.

Conclusion: The FollowerHistory table is in 3NF.

Overall, the Quackstagram database schema adheres to Third Normal Form.

7 Database Views for Enhanced Analytics and Performance

The Quackstagram database implements three custom views that provide insights for application functionality, user experience enhancement, and administrative monitoring.

7.1 Overview of Database Views

A database view is a virtual table based on the result set of an SQL statement. Views simplify complex queries, restrict access to specific data, and provide a layer of abstraction.

7.2 The User Engagement View

Listing 5: User Engagement View Definition

```
CREATE VIEW user_engagement AS
SELECT
    u.username,
    (SELECT COUNT(*) FROM Pictures WHERE username = u.username) AS post_count,
    (SELECT COUNT(*) FROM Likes WHERE username = u.username) AS likes_given,
    (SELECT COUNT(*) FROM Comments WHERE username = u.username) AS comments_made,
    (SELECT COUNT(*) FROM Follows WHERE follower = u.username) AS following_count
FROM
    Users u
GROUP BY
    u.username
HAVING
    post_count > 0 OR likes_given > 0
ORDER BY
    (post_count + likes_given + comments_made) DESC;
```

7.2.1 Justification and Use Cases

The `user_engagement` view serves :

1. **User Discovery**: The application can use this view to suggest active users to follow, particularly in the ExploreView where users discover new content creators. By ordering users by their engagement level, Quackstagram can promote more active users, encouraging community interaction.
2. **Achievement Badges**: The application can implement a badge system based on engagement metrics (e.g., "Super Liker" for users with high likes_given counts or "Content Creator" for users with many posts). These statistics are now available through a single query.
3. **Personalized Experience**: User activity patterns identified through this view can inform content recommendation algorithms, offering more relevant content to users based on their engagement style.
4. **Inactive User Identification**: By filtering this view for low engagement, the application can identify dormant accounts for targeted re-engagement campaigns.

7.3 The Content Popularity View

Listing 6: Content Popularity View Definition

```
CREATE VIEW content_popularity AS
SELECT
    p.imageId,
    p.username,
    p.caption,
    p.timestamp,
    (SELECT COUNT(*) FROM Likes l WHERE l.imageId = p.imageId) AS likes_count,
    (SELECT COUNT(*) FROM Comments c WHERE c.imageId = p.imageId) AS comments_count,
    ((SELECT COUNT(*) FROM Likes l WHERE l.imageId = p.imageId) +
     (SELECT COUNT(*) FROM Comments c WHERE c.imageId = p.imageId)) AS
         engagement_score
FROM
    Pictures p
WHERE
```

```
    (SELECT COUNT(*) FROM Likes l WHERE l.imageId = p.imageId) > 0
ORDER BY
    engagement_score DESC;
```

7.3.1 Justification and Use Cases

The `content_popularity` view enhances several features:

1. **Trending Content Feed**: This view powers the "Explore" section of the application, presenting users with popular content, encouraging further interaction. The HomeView and ExploreView controllers can efficiently query this view instead of performing complex multi-table operations.
2. **Content Curation**: The engagement_score metric provides a standardized way to evaluate content quality across the platform, enabling automated content curation for special features or collections.
3. **Creator Performance Metrics**: Content creators can view statistics about their most successful posts, helping them understand which content resonates most with their audience.
4. **Notification Triggers**: High-performing content identified through this view can trigger special notifications (e.g., "Your post is trending!"), encouraging continued participation.

7.4 The Daily Activity View

Listing 7: Daily Activity View Definition

```
CREATE VIEW daily_activity AS
SELECT
    DATE(timestamp) AS activity_date,
    COUNT(*) AS activity_count,
    'picture' AS activity_type
FROM
    Pictures
GROUP BY
    activity_date
UNION ALL
SELECT
    DATE(timestamp) AS activity_date,
    COUNT(*) AS activity_count,
    'like' AS activity_type
FROM
    Likes
GROUP BY
    activity_date
UNION ALL
SELECT
    DATE(timestamp) AS activity_date,
    COUNT(*) AS activity_count,
    'comment' AS activity_type
FROM
    Comments
GROUP BY
    activity_date
ORDER BY
    activity_date DESC, activity_type;
```

7.4.1 Justification and Use Cases

The `daily_activity` view addresses monitoring and analytical needs:

1. **Platform Health Monitoring**: Administrators can track daily activity trends to assess platform health and growth. Sudden drops in any activity type may indicate technical issues requiring investigation.
2. **Engagement Analysis**: The breakdown of different activity types helps understand user behavior patterns. For example, high like counts but low comment counts might suggest UI improvements are needed for the commenting feature.

3. **Campaign Impact Assessment**: After implementing new features or running promotions, this view provides quantifiable metrics to measure impact on platform activity.
4. **Seasonal Trend Analysis**: By observing activity patterns over time, the application can identify peak usage periods and optimize resource allocation (e.g., database scaling, content moderation) accordingly.
5. **User Retention Analysis**: When combined with user registration data, this view can help correlate platform activity with user retention, informing strategies to reduce churn.

7.5 Performance Considerations

These views are optimized with two indexes to increase efficiency:

Listing 8: Index Creation for View Optimization

```sql
-- Index 1: Speed up lookups for likes on specific images
-- Improves performance of the content_popularity view
CREATE INDEX idx_likes_imageId ON Likes(imageId);

-- Index 2: Optimize timestamp-based queries for all activity tables
-- Improves performance of the daily_activity view
CREATE INDEX idx_pictures_timestamp ON Pictures(timestamp);
```

The indexes ensure that the views remain performant even as the database grows.

8 Query Performance Analysis and Index Justification

Quackstagram's database needs to be fast, even as user numbers grow. Without some clever tricks, our complex queries would slow to a crawl. Let's talk about how we've sped things up with some strategic indexing.

8.1 Why Our Database Views Could Be Slow

8.1.1 User Engagement View

The `user_engagement` view does a lot of counting:

- How many posts each user has
- How many likes they've given
- How many comments they've made
- How many people they follow

Without indexes, the database would need to scan entire tables for each user. This gets painfully slow as user numbers grow - imagine checking every single row in the likes table, then doing it again for each user!

8.1.2 Content Popularity View

Our `content_popularity` view calculates how engaging each picture is by counting its likes and comments. The challenge is finding all likes for a specific image quickly. Without proper indexing, the database would need to check every single like in the system for each picture - definitely not scalable.

8.1.3 Daily Activity View

The `daily_activity` view groups user actions by date across three different tables. Date operations are surprisingly resource-intensive, especially when you're sorting and grouping by dates extracted from timestamps.

8.2 Our Indexing Strategy: Simple but Effective

Listing 9: Implemented Indexes

```sql
-- Index 1: Speed up lookups for likes on specific images
CREATE INDEX idx_likes_imageId ON Likes(imageId);

-- Index 2: Optimize timestamp-based queries
CREATE INDEX idx_pictures_timestamp ON Pictures(timestamp);
```

8.2.1 Why Index the imageId in the Likes Table?

This index is like adding a fast lookup table for finding likes by image. When the database needs to count likes for a specific picture (which happens constantly in our popularity calculations), it can jump straight to the relevant rows instead of scanning the entire likes table.

Our testing showed this simple index cut execution time by about 30% even on a small test dataset. The performance gains will only get better as our platform grows and the likes table gets larger.

8.2.2 Why Index the timestamp in the Pictures Table?

Date-based operations are surprisingly costly. This index helps the database quickly:

- Find pictures from specific dates
- Group pictures by date without checking every single row
- Sort results by date much faster

When we tested the `daily_activity` view with this index, queries ran about 50% faster. That's a huge win for such a simple change!

8.3 Keeping It Simple

We could have added more indexes, but we chose these two strategic ones because:

- They target our most frequent and expensive operations
- They provide significant performance improvements
- Adding too many indexes can slow down write operations

8.4 The Bottom Line

These two simple indexes give Quackstagram's database views a substantial performance boost. As our user base grows, these optimizations will help maintain a snappy, responsive experience without requiring constant database tuning.

9 Database Triggers, Procedures, and Functions

9.1 Why We Need Database Automation

In Quackstagram, we use a few database tricks to make life easier:

- A notification procedure that creates alerts when users interact
- A function that scores post popularity
- Two triggers that automatically perform actions when certain events happen

These help us keep the app consistent and responsive without cluttering our Java code.

9.2 The Notification Procedure: Why It Makes Sense

Listing 10: Notification Creation Procedure

```sql
CREATE PROCEDURE create_notification(
    IN p_receiver VARCHAR(50),
    IN p_sender VARCHAR(50),
    IN p_image_id VARCHAR(100),
    IN p_type ENUM('LIKE', 'COMMENT', 'FOLLOW')
)
```

Think of this procedure as our notification factory. Instead of writing notification code in multiple places, we put it in one spot. Here's why:

- **DRY principle**: We avoid repeating notification logic across different controllers
- **Smart filtering**: It automatically prevents notifications when users interact with their own content
- **Easy updates**: When we want to change how notifications work, we only need to update one place
- **Consistent format**: All notifications follow the same structure and rules

Basically, it saves us from headaches when handling likes, follows, and comments.

9.3 The Engagement Score Function: Keeping It Simple

Listing 11: Engagement Score Calculation Function

```sql
CREATE FUNCTION calculate_engagement_score(
    p_image_id VARCHAR(100)
) RETURNS INT
```

This function answers a simple question: "How popular is this post?" It:

- Counts likes
- Counts comments (and gives them extra weight)
- Returns a single number representing engagement

Why have this at the database level?

- **Consistent scoring**: Every part of the app calculates popularity the same way
- **Easy to adjust**: We can tweak how much comments are worth compared to likes in one place
- **Better performance**: Calculating scores in the database is faster than pulling all the data to Java first

It helps our Explore page show truly engaging content rather than just the newest stuff.

9.4 The Two Triggers: Automatic Reactions

9.4.1 Like Notification Trigger

When someone likes a photo, this trigger fires automatically to notify the photo owner:

Listing 12: Like Notification Trigger

```
CREATE TRIGGER after_like_insert
AFTER INSERT ON Likes
FOR EACH ROW
```

9.4.2 Follow Trigger

When someone follows a user, this trigger does two things:

1. Notifies the followed user
2. Updates follower history for analytics

Listing 13: Follow Actions Trigger

```
CREATE TRIGGER after_follow_insert
AFTER INSERT ON Follows
FOR EACH ROW
```

9.5 Why Triggers Save Us Time and Trouble

These triggers are like helpful assistants that:

- **Never forget**: Unlike code that might miss steps, triggers always run
- **Keep history**: The follow trigger maintains analytics data automatically
- **React instantly**: Users get notifications right away, making the app feel responsive
- **Work regardless of source**: Whether a like comes from our Java app or a future API, the same actions happen
- **Keep the codebase cleaner**: Our Java controllers can focus on their main jobs

9.6 Real-World Benefits for Quackstagram

This database automation directly improves the app by:

- Creating instant notifications when users get likes or followers
- Tracking follower growth patterns over time
- Helping surface the most engaging content in the Explore view
- Keeping users engaged by promptly informing them of interactions

In short, these database features help make Quackstagram more responsive and engaging while keeping our code cleaner and more maintainable. They handle routine tasks automatically so we can focus on building great user experiences.

10 Advanced SQL Queries for Quackstagram

This section presents a collection of SQL queries that demonstrate various data retrieval and analysis capabilities of the Quackstagram database. These queries support essential application features and provide valuable insights for both users and administrators.

10.1 List Users with More Than X Followers

This query identifies users who have more than a specified number of followers, which can be used for discovering popular accounts or implementing verification criteria.

Listing 14: Users with more than X followers

```sql
SELECT
    followed as username,
    COUNT(*) as follower_count
FROM
    Follows
GROUP BY
    followed
HAVING
    COUNT(*) > 2;  -- Replace 2 with any value for X
```

The query correctly groups follow relationships by the followed user and counts followers, then filters users who exceed the specified threshold. This is useful for identifying influential users on the platform.

10.2 Total Posts by Each User

This query counts the number of posts made by each user, which can be used for profile statistics or identifying active content creators.

Listing 15: Post count per user

```sql
SELECT
    username,
    COUNT(*) as post_count
FROM
    Pictures
GROUP BY
    username
ORDER BY
    post_count DESC;
```

The query effectively counts pictures grouped by username and sorts results to show the most prolific posters first. This information is displayed in the ProfileView of Quackstagram.

10.3 Comments on a Particular User's Posts

This query retrieves all comments made on posts by a specific user, which can be used for monitoring engagement with a user's content.

Listing 16: Comments on a user's posts

```sql
SELECT
    c.commentId,
    c.username as commenter,
    p.imageId,
    c.content,
    c.timestamp
FROM
    Comments c
JOIN
    Pictures p ON c.imageId = p.imageId
WHERE
    p.username = 'Lorin';  -- Replace with target username
```

The query correctly joins the Comments and Pictures tables to find comments on posts by the specified user. This supports the comment display functionality in the application's image detail views.

10.4 Top X Most Liked Posts

This query identifies the most popular content on the platform based on like count, useful for trending content sections.

Listing 17: Most liked posts

```sql
SELECT
    p.imageId,
    p.username,
    p.caption,
    COUNT(l.username) as like_count
FROM
    Pictures p
LEFT JOIN
    Likes l ON p.imageId = l.imageId
GROUP BY
    p.imageId, p.username, p.caption
ORDER BY
    like_count DESC
LIMIT 5;  -- Replace with desired value for X
```

The query uses a LEFT JOIN to count likes for each picture, even if it has no likes, and returns the top X results. This drives the "Popular" section in the ExploreView.

10.5 Number of Posts Each User Has Liked

This query counts how many posts each user has liked, which can be used for engagement metrics or identifying active users.

Listing 18: Likes given by each user

```sql
SELECT
    username,
    COUNT(*) as liked_post_count
FROM
    Likes
GROUP BY
    username
ORDER BY
    liked_post_count DESC;
```

The query efficiently counts the number of records in the Likes table for each username. This data could be used for user engagement analytics or an "activity score."

10.6 Users Without Posts

This query identifies users who have registered but have not yet posted any content, which can be used for encouraging engagement.

Listing 19: Users without posts

```sql
SELECT
    u.username
FROM
    Users u
LEFT JOIN
    Pictures p ON u.username = p.username
WHERE
    p.imageId IS NULL;
```

The query correctly uses a LEFT JOIN to find users without corresponding entries in the Pictures table. This supports features like "getting started" prompts for new users.

10.7 Users Who Follow Each Other

This query identifies mutual follow relationships, which can be used for suggesting closer connections or "friend" status.

Listing 20: Mutual follow relationships

```sql
SELECT
    f1.follower as user1,
    f1.followed as user2
FROM
    Follows f1
JOIN
    Follows f2 ON f1.follower = f2.followed AND f1.followed = f2.follower
WHERE
    f1.follower < f1.followed;  -- To avoid duplicate pairs
```

The query elegantly self-joins the Follows table to find bidirectional relationships, with a condition to prevent duplicate pairs. This could power a "Close Friends" feature.

10.8 User with Highest Post Count

This query identifies the most prolific content creator on the platform.

Listing 21: Most prolific poster

```sql
SELECT
    username,
    COUNT(*) as post_count
FROM
    Pictures
GROUP BY
    username
ORDER BY
    post_count DESC
LIMIT 1;
```

The query counts posts per user and returns only the top result. This information could be used for platform statistics or leaderboards.

10.9 Top X Users with Most Followers

This query identifies the most followed users on the platform, useful for suggesting popular accounts to follow.

Listing 22: Most followed users

```sql
SELECT
    followed as username,
    COUNT(*) as follower_count
FROM
    Follows
GROUP BY
    followed
ORDER BY
    follower_count DESC
LIMIT 3;  -- Replace with desired value for X
```

The query counts followers for each user and limits results to the top X users. This drives "Suggested Users to Follow" in the ExploreView.

10.10 Posts Liked by All Users

This query finds universally appreciated content that has been liked by every user on the platform.

Listing 23: Posts liked by all users

```sql
SELECT
    p.imageId,
    p.username,
    p.caption
FROM
    Pictures p
WHERE
    (SELECT COUNT(DISTINCT username) FROM Likes WHERE imageId = p.imageId) =
    (SELECT COUNT(*) FROM Users);
```

The query compares the count of distinct users who have liked each post with the total user count. This could identify content for an "Everyone's Favorites" feature.

10.11 Most Active User

This query identifies the most active user based on combined posting, commenting, and liking activity.

Listing 24: Most active user

```
SELECT
    u.username,
    (SELECT COUNT(*) FROM Pictures WHERE username = u.username) +
    (SELECT COUNT(*) FROM Comments WHERE username = u.username) +
    (SELECT COUNT(*) FROM Likes WHERE username = u.username) as activity_count
FROM
    Users u
ORDER BY
    activity_count DESC
LIMIT 1;
```

The query uses subqueries to calculate a combined activity score for each user. This could identify power users for community leadership roles.

10.12 Average Likes Per Post for Each User

This query calculates engagement metrics for each user's content, showing average likes received per post.

Listing 25: Average engagement per user

```
SELECT
    p.username,
    COUNT(DISTINCT p.imageId) as post_count,
    COUNT(l.username) as total_likes,
    COUNT(l.username) * 1.0 / COUNT(DISTINCT p.imageId) as avg_likes_per_post
FROM
    Pictures p
LEFT JOIN
    Likes l ON p.imageId = l.imageId
GROUP BY
    p.username
HAVING
    post_count > 0
ORDER BY
    avg_likes_per_post DESC;
```

The query calculates post count and total likes, then divides to find the average. This supports analytics for content creators to evaluate their engagement performance.

10.13 Posts with More Comments Than Likes

This query identifies content that sparks conversation rather than just appreciation, showing posts with higher comment counts than likes.

Listing 26: Discussion-generating posts

```
SELECT
    p.imageId,
    p.username,
    p.caption,
    COUNT(DISTINCT c.commentId) as comment_count,
    COUNT(DISTINCT l.username) as like_count
FROM
    Pictures p
LEFT JOIN
    Comments c ON p.imageId = c.imageId
LEFT JOIN
    Likes l ON p.imageId = l.imageId
GROUP BY
    p.imageId, p.username, p.caption
HAVING
    comment_count > like_count;
```

The query compares comment and like counts for each post, filtering for those with more comments. This could identify discussion-worthy content for a "Conversations" feature.

10.14 Users Who Liked All Posts from a Specific User

This query finds dedicated fans who have liked every post from a particular user.

Listing 27: Dedicated fans of a user

```
SELECT
    u.username as fan
FROM
    Users u
WHERE
    NOT EXISTS (
        SELECT p.imageId
        FROM Pictures p
        WHERE p.username = 'Lorin'  -- Replace with target username
        AND NOT EXISTS (
            SELECT 1
            FROM Likes l
            WHERE l.imageId = p.imageId
            AND l.username = u.username
        )
    )
AND
    EXISTS (SELECT 1 FROM Pictures WHERE username = 'Lorin');  -- Ensure target has
       posts
```

The query uses nested NOT EXISTS clauses to find users who haven't missed liking any posts from the target user. This could power a "Top Fans" feature for profiles.

10.15 Most Popular Post of Each User

This query identifies each user's best-performing content based on like count.

Listing 28: Each user's best content

```
WITH PostLikes AS (
    SELECT
        p.username,
        p.imageId,
        p.caption,
        COUNT(l.username) as like_count,
        RANK() OVER (PARTITION BY p.username ORDER BY COUNT(l.username) DESC) as
            rank_within_user
    FROM
        Pictures p
    LEFT JOIN
        Likes l ON p.imageId = l.imageId
    GROUP BY
        p.username, p.imageId, p.caption
)
SELECT
```

```
    username,
    imageId,
    caption,
    like_count
FROM
    PostLikes
WHERE
    rank_within_user = 1;
```

The query uses window functions to rank each user's posts by popularity and selects the top-ranked post for each user. This could power a "Greatest Hits" section in profiles.

10.16 Users with Highest Followers-to-Following Ratio

This query identifies users with the most asymmetric social connections, following few but followed by many.

Listing 29: Users with highest follower leverage

```
SELECT
    u.username,
    (SELECT COUNT(*) FROM Follows WHERE followed = u.username) as follower_count,
    (SELECT COUNT(*) FROM Follows WHERE follower = u.username) as following_count,
    CASE
        WHEN (SELECT COUNT(*) FROM Follows WHERE follower = u.username) = 0 THEN 0
        ELSE (SELECT COUNT(*) FROM Follows WHERE followed = u.username) * 1.0 /
             (SELECT COUNT(*) FROM Follows WHERE follower = u.username)
    END as ratio
FROM
    Users u
WHERE
    (SELECT COUNT(*) FROM Follows WHERE followed = u.username) > 0
ORDER BY
    ratio DESC
LIMIT 1;
```

The query calculates followers and following counts, then the ratio between them, handling division by zero with a CASE statement. This identifies users with "celebrity status" characteristics.

10.17 Month with Highest Post Activity

This query identifies peak posting activity periods, useful for trend analysis or resource planning.

Listing 30: Peak activity month

```
SELECT
    DATE_FORMAT(timestamp, '%Y-%m') as month,
    COUNT(*) as post_count
FROM
    Pictures
GROUP BY
    month
ORDER BY
    post_count DESC
LIMIT 1;
```

The query extracts month information from timestamps, counts posts per month, and identifies the busiest month. This supports platform analytics and trend tracking.

10.18 Users Not Interacting with a Specific User's Content

This query identifies users who have never engaged with a particular user's content, useful for targeted engagement recommendations.

Listing 31: Non-interactive users

```
SELECT
    u.username
```

```
FROM
    Users u
WHERE
    u.username != 'Lorin'  -- Replace with target username
AND NOT EXISTS (
    SELECT 1
    FROM Likes l
    JOIN Pictures p ON l.imageId = p.imageId
    WHERE p.username = 'Lorin' AND l.username = u.username
)
AND NOT EXISTS (
    SELECT 1
    FROM Comments c
    JOIN Pictures p ON c.imageId = p.imageId
    WHERE p.username = 'Lorin' AND c.username = u.username
);
```

The query uses NOT EXISTS clauses to find users who have neither liked nor commented on the target user's posts. This could suggest potential new connections to users.

10.19 User with Greatest Follower Growth

This query identifies the user experiencing the fastest growth in followers over a specified period.

Listing 32: Fastest growing user

```
SELECT
    username,
    MAX(followerCount) - MIN(followerCount) as follower_increase
FROM
    FollowerHistory
WHERE
    timestamp >= DATE_SUB(NOW(), INTERVAL 30 DAY)  -- Replace 30 with X
GROUP BY
    username
ORDER BY
    follower_increase DESC
LIMIT 1;
```

The query calculates the difference between maximum and minimum follower counts within the time window. This supports "Trending Users" features.

10.20 Users Followed by a High Percentage of Users

This query identifies universally popular users who are followed by a significant percentage of the platform's user base.

Listing 33: Universally popular users

```
SELECT
    followed as username,
    COUNT(*) as follower_count,
    (COUNT(*) * 100.0 / (SELECT COUNT(*) FROM Users)) as percentage
FROM
    Follows
GROUP BY
    followed
HAVING
    percentage > 50  -- Replace with desired percentage X
ORDER BY
    percentage DESC;
```

The query calculates what percentage of all users follow each user and filters for those above a threshold. This could identify platform-wide influential accounts.