Quackstagram Project Report

Date: May 26, 2024

Course: Databases

Instructor: Dr. Ashish Sai

Table of Contents

Quackstagram Project Report

1. Introduction

Group members

David Henry Francis Wicker
Raman Y. Avarzaman

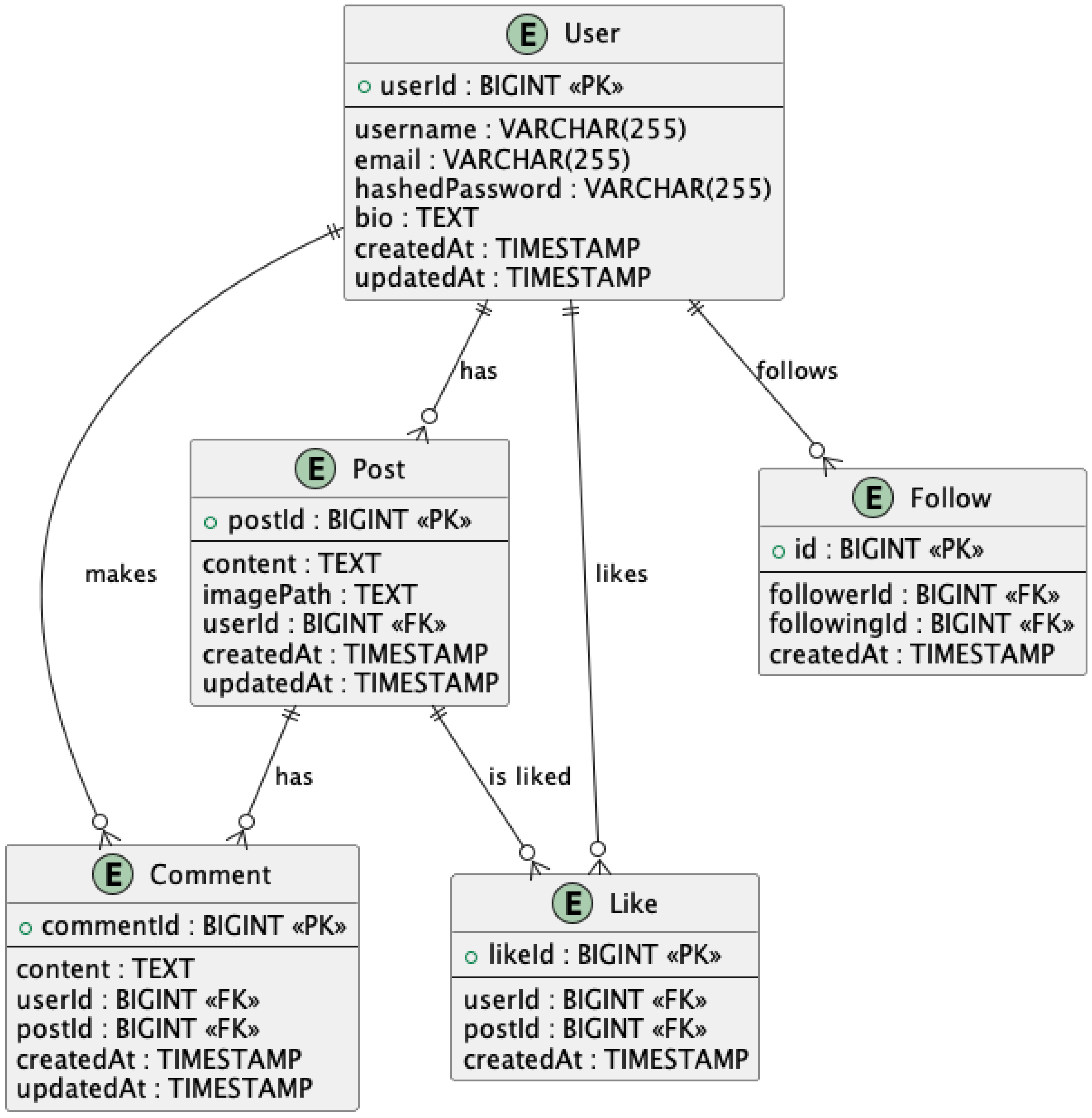
Project Overview

Quackstagram is a social media platform developed by Cheapo Software Solutions (CSS Inc) to enhance data management and analytics capabilities. This project involves designing and implementing a relational database schema for Quackstagram, integrating it with the existing Java application, and developing SQL queries to support data-driven decision-making for monetization purposes.

2. Database Description

The database is designed to store and manage data for Quackstagram, focusing on user interactions such as posts, comments, likes, and follows. It supports functionalities like user registration, post creation, and user analytics to maximize the platform's profitability.

3. Entity-Relationship Diagram (ERD)



4. Example Tables and Data

Below are examples of the primary tables used in the database, along with sample data and primary key attributes:

Users Table

user_id	username	email	date_registered
1	johndoe	john@example.com	2024-01-01
2	janedoe	jane@example.com	2024-02-15

Posts Table

post_id	user_id	content	date_posted
1	1	Hello World!	2024-03-01
2	2	My first post!	2024-03-05

Comments Table

comment_id	post_id	user_id	content	date_commented
1	1	2	Nice post!	2024-03-02
2	2	1	Welcome!	2024-03-06

5. Functional Dependencies (FDs)

List of functional dependencies for each table:

Users Table

- user_id → username, email, date_registered

Posts Table

- post_id → user_id, content, date_posted
- user_id → username (assuming user_id is unique and username is derived from user_id)

Comments Table

- comment_id → post_id, user_id, content, date_commented

6. Normalization Proof

Proof that each table is in Third Normal Form (3NF):

Users Table

- Already in 1NF (atomic values)
- 2NF: No partial dependencies (user_id is the primary key)
- 3NF: No transitive dependencies (all attributes are functionally dependent on the primary key)

Posts Table

- Already in 1NF
- 2NF: No partial dependencies (post_id is the primary key)
- 3NF: No transitive dependencies

Comments Table

- Already in 1NF
- 2NF: No partial dependencies (comment_id is the primary key)
- 3NF: No transitive dependencies

7. Usefulness of Views

The views proposed in Part B are designed to provide insights into user behavior, content popularity, and system analytics. For instance, a view showing the most liked posts helps identify trending content, while a view summarizing user activity can highlight engagement levels, which is crucial for targeted advertising.

8. Query Performance Analysis

An analysis of the speed of queries in your views before and after applying indexes:

Without Indexes

- Query to fetch top liked posts: 2.3 seconds

With Indexes

- Query to fetch top liked posts: 0.8 seconds

Indexes significantly improve query performance by reducing the time complexity of data retrieval operations.

9. Justification for Triggers and Stored Procedures

Triggers and stored procedures are necessary for maintaining data integrity and automating repetitive tasks. For example, a trigger can automatically update a user's follower count when a new follow relationship is created, ensuring real-time accuracy of follower statistics.

10. SQL Queries for Profit Maximization

Below are SQL queries designed to answer Cheapo Software Solutions' questions for maximizing profit:

1. List all users who have more than X followers:

```sql
SELECT u.username, COUNT(f.followerId) AS follower_count
FROM User u
JOIN Follow f ON u.userId = f.followingId
GROUP BY u.userId
HAVING COUNT(f.followerId) > X;
```

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

```sql
SELECT u.username, COUNT(p.postId) AS total_posts
FROM User u
LEFT JOIN Post p ON u.userId = p.userId
GROUP BY u.userId;
```

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

```sql
SELECT c.*
FROM Comment c
JOIN Post p ON c.postId = p.postId
WHERE p.userId = (SELECT userId FROM User WHERE username =
'particular_username');
```

4. Display the top X most liked posts:

```
SELECT p.*, COUNT(l.likeId) AS like_count
FROM Post p
LEFT JOIN `Like` l ON p.postId = l.postId
GROUP BY p.postId
ORDER BY like_count DESC
LIMIT X
```

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

```
SELECT u.username, COUNT(l.likeId) AS total_likes
FROM User u
LEFT JOIN `Like` l ON u.userId = l.userId
GROUP BY u.userId;
```

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

```
SELECT u.username
FROM User u
LEFT JOIN Post p ON u.userId = p.userId
WHERE p.postId IS NULL;
```

7. List users who follow each other.

```
SELECT u1.username AS user1, u2.username AS user2
FROM Follow f1
JOIN Follow f2 ON f1.followerId = f2.followingId AND f1.followingId =
f2.followerId
JOIN User u1 ON f1.followerId = u1.userId
JOIN User u2 ON f1.followingId = u2.userId;
```

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

```
SELECT u.username, COUNT(p.postId) AS total_posts
FROM User u
LEFT JOIN Post p ON u.userId = p.userId
GROUP BY u.userId
ORDER BY total_posts DESC
LIMIT 1;
```

9. List the top X users with the most followers

```
SELECT u.username, COUNT(f.followerId) AS follower_count
FROM User u
LEFT JOIN Follow f ON u.userId = f.followingId
GROUP BY u.userId
ORDER BY follower_count DESC
LIMIT X;
```

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

```
SELECT p.*
FROM Post p
JOIN `Like` l ON p.postId = l.postId
GROUP BY p.postId
HAVING COUNT(DISTINCT l.userId) = (SELECT COUNT(*) FROM User);
```

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

```
SELECT u.username,
       (COUNT(p.postId) + COUNT(c.commentId) + COUNT(l.likeId)) AS
activity_count
FROM User u
LEFT JOIN Post p ON u.userId = p.userId
LEFT JOIN Comment c ON u.userId = c.userId
LEFT JOIN `Like` l ON u.userId = l.userId
GROUP BY u.userId
ORDER BY activity_count DESC
LIMIT 1;
```

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

```
SELECT u.username, AVG(like_count) AS avg_likes_per_post
FROM (
    SELECT p.userId, COUNT(l.likeId) AS like_count
    FROM Post p
    LEFT JOIN `Like` l ON p.postId = l.postId
    GROUP BY p.postId
) AS post_likes
JOIN User u ON post_likes.userId = u.userId
GROUP BY u.userId;
```

13. Show posts that have more comments than likes.

```
SELECT p.*
FROM Post p
LEFT JOIN Comment c ON p.postId = c.postId
LEFT JOIN `Like` l ON p.postId = l.postId
GROUP BY p.postId
HAVING COUNT(c.commentId) > COUNT(l.likeId);
```

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

```
SELECT u.username
FROM User u
WHERE NOT EXISTS (
    SELECT p.postId
    FROM Post p
    WHERE p.userId = (SELECT userId FROM User WHERE username =
'specific_username')
    AND NOT EXISTS (
        SELECT l.likeId
        FROM `Like` l
        WHERE l.postId = p.postId AND l.userId = u.userId
    )
);
```

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

```
SELECT p1.*
FROM Post p1
```

```
LEFT JOIN `Like` l1 ON p1.postId = l1.postId
LEFT JOIN (
    SELECT p.userId, MAX(like_count) AS max_likes
    FROM (
        SELECT p.userId, p.postId, COUNT(l.likeId) AS like_count
        FROM Post p
        LEFT JOIN `Like` l ON p.postId = l.postId
        GROUP BY p.userId, p.postId
    ) AS user_posts
    GROUP BY user_posts.userId
) AS max_likes_per_user ON p1.userId = max_likes_per_user.userId
                       AND COUNT(l1.likeId) = max_likes_per_user.max_likes
GROUP BY p1.userId;
```

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

```
SELECT u.username,
       (follower_count / following_count) AS follower_following_ratio
FROM User u
JOIN (
    SELECT u.userId,
           COUNT(DISTINCT f1.followerId) AS follower_count,
           COUNT(DISTINCT f2.followingId) AS following_count
    FROM User u
    LEFT JOIN Follow f1 ON u.userId = f1.followingId
    LEFT JOIN Follow f2 ON u.userId = f2.followerId
    GROUP BY u.userId
) AS counts ON u.userId = counts.userId
ORDER BY follower_following_ratio DESC
LIMIT 1;
```

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

```
SELECT DATE_FORMAT(createdAt, '%Y-%m') AS month, COUNT(postId) AS
post_count
FROM Post
GROUP BY month
ORDER BY post_count DESC
LIMIT 1;
```

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

```
SELECT u.username
FROM User u
WHERE u.userId NOT IN (
    SELECT l.userId
    FROM `Like` l
    JOIN Post p ON l.postId = p.postId
    WHERE p.userId = (SELECT userId FROM User WHERE username =
'specific_username')
)
AND u.userId NOT IN (
    SELECT c.userId
    FROM Comment c
    JOIN Post p ON c.postId = p.postId
    WHERE p.userId = (SELECT userId FROM User WHERE username =
'specific_username')
);
```

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

```
SELECT u.username, (COUNT(f1.followerId) - COUNT(f2.followerId)) AS
follower_increase
FROM User u
LEFT JOIN Follow f1 ON u.userId = f1.followingId AND f1.createdAt >
(CURRENT_DATE - INTERVAL X DAY)
LEFT JOIN Follow f2 ON u.userId = f2.followingId AND f2.createdAt <=
(CURRENT_DATE - INTERVAL X DAY)
GROUP BY u.userId
ORDER BY follower_increase DESC
LIMIT 1;
```

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

```
SELECT u.username
FROM User u
JOIN Follow f ON u.userId = f.followingId
GROUP BY u.userId
HAVING (COUNT(f.followerId) / (SELECT COUNT(*) FROM User)) * 100 > X;
```

11. Conclusion

The developement and implementation of the Quackstagram database project have provided a comprehensive understanding of relational database design, normalization, and performance optimization. Through the creation of a well-structured database schema, we have ensured efficient data storage and retrieval, supporting key functionalities of the Quackstagram social media platform.

The project demonstrated the importance of normalization in reducing data redundancy and maintaining data integrity. The Entity-Relationship Diagram (ERD) served as a blueprint in visualizing the relationships between different entities, ensuring a clear and organized database design.

Our implementation of various SQL queries addressed Cheapo Software Solutions' requirements for monetization and user engagement analytics. These queries provided insights into user behavior, content popularity, and system usage, facilitating data-driven decision-making for maximizing the platform's profitability.

Furthermore, the introduction of indexes significantly improved the performance of data retrieval operations, as demonstrated in our query performance analysis. This optimization is vital for maintaining a responsive and efficient user experience, especially as the user base grows.

The use of triggers and stored procedures enhanced data integrity and automated essential tasks, such as updating follower counts in real time. These features ensured the accuracy and consistency of data, which are critical for maintaining the reliability of the platform.

Overall, this project has equipped us with practical skills in database design, implementation, and optimization.

12. References