International Journal of Research and Development in Applied Science and Engineering (IJRDASE)
ISSN: 2454-6844

Building a SQL based Multi Topic Recommendation System using Collaborative Based Filtering Technique

S. Mohammed Azam Rizvi, Abdullah Malik, Shadab Rehan, Satyam Shukla, Shra Fatima
Computer Science, Integral University, Lucknow, India
smarizvi2025@gmail.com, malikabdullahblp@gmail.com, shadabkhanj353@gmail.com, satyamshukla3043@gmail.com, shraintegral251@gmail.com

Abstract: We propose a recommendation system for the large amount data available on the web in the form of ratings, reviews, opinions, complain, remarks. feedback, and comments about any item (product, event, individual and services). Here we recommended collaborative based filtering technique to filter different types of reviews, opinions. Remarks, comments, complains etc. Because recommendations are based on ratings, ranks, content. Reviewer's behaviour, and timing of review generated by different reviewers. a lightweight, explainable, and database-native recommender system. It doesn't aim to compete with advanced AI models but provides an accessible and deployable foundation for simple use cases or academic exploration. By studying the limitations of existing tools and choosing SQL deliberately, we create a system that is not only functional but educational—ideal for students, early practitioners, or constrained systems where ML deployment isn't feasible.

Keywords: Recommendation system, Reviews and ratings, Explainable recommender system, SQL-based system, Collaborative filtering, Accessible AI tools, Reviewer behavior.

1. Introduction

In today's digital landscape, recommendation systems have become essential for personalizing user experiences across various platforms. Whether it's suggesting products on an e-commerce site, recommending movies on a streaming service, or curating music playlists, these systems help users discover content that aligns with their tastes and preferences. The proliferation of digital content platforms has led to an overwhelming abundance of movies and TV shows available for consumption. With such an extensive catalog, users often find it challenging to discover content tailored to their preferences. To address this issue, movie recommender systems powered by machine learning have emerged as a solution to curate personalized recommendations for users. Traditional approaches rely heavily on machine learning algorithms, often requiring massive datasets, real-time infrastructure, and tuning.

This project takes an alternative route, offering insights into how effective SQL can be for collaborative filtering, particularly in social-based recommendation contexts.

The recommender system spans multiple domains movies, music, and books making it a multi-modal engine. At its core, the system uses relational data from user-item interactions, computes similarity scores between users based on their ratings, and recommends items not yet consumed by a user but highly rated by similar users. This process is managed entirely through SQL queries embedded within Java servlet-powered web interfaces. Users interact with the system through basic HTML pages, and results are rendered dynamically based on SQL query outputs.

This type of project aligns well with the goals of educational research and lightweight industrial use cases. It demonstrates how powerful recommendation logic can be achieved using foundational tools that are already ubiquitous in most data-driven applications: relational databases and SQL. Furthermore, it offers insight into the effectiveness of SQL for handling moderately complex logical flows such as calculating user similarity, filtering preferences, and ranking recommendations.

Another significant aspect of this implementation is its generality. It abstracts the recommendation logic in a way that is domain-agnostic. By changing the input datasets whether books, songs, or films the underlying logic still applies, suggesting that the engine could be adapted for any scenario involving user-item interactions. It is a prime candidate for demonstrating the versatility of collaborative filtering models outside traditional machine learning environments.

1.1 Challenges in Design

a. Challenges in Building SQL-Based Recommender Systems

Recommender systems are central to many modern applications, offering tailored content and improving user engagement. However, while the conceptual model behind recommendation algorithms is well understood, implementing these systems especially using only SQL presents a unique set of challenges. This project attempts to overcome several such hurdles using a relational database-driven approach. This section explores the technical and practical challenges encountered in building such a system, particularly without the aid of traditional machine learning tools.

b. Expressing Collaborative Filtering in SQL

The first and most foundational challenge is translating the principles of collaborative filtering into SQL. Collaborative filtering is inherently iterative and comparative—it requires calculating the similarity between users, aggregating preferences, and filtering out already seen content. These tasks are more naturally expressed in programming languages with matrix manipulation support (e.g., Python with NumPy, or R), or in dedicated libraries like Surprise or TensorFlow.

SQL, on the other hand, is a declarative language designed primarily for querying and retrieving data—not for computations or iterative processing.

In this project, all logical steps, such as computing rating products (user similarity), aggregating bonds (sum of rank scores), and filtering recommendations, are implemented using nested CTEs (Common Table Expressions) and multi-stage joins. While functional, these constructs are complex and can easily become unwieldy, leading to long queries that are hard to debug and optimize. The project successfully overcomes this challenge but at the cost of verbosity and potential inefficiency.

c. Performance and Scalability

Another major concern is performance. As the dataset grows, the computational cost of performing joins and aggregations increases exponentially. SQL joins across multiple large tables (e.g., user, movie, songs, and likes) can introduce significant latency. In traditional recommender systems, matrix factorization techniques reduce dimensionality and allow scalable similarity calculations. In contrast, SQL has no built-in support for these optimizations.

Moreover, since this system operates by comparing all user-item pairs and aggregating them at query time, it does not precompute or cache results, which leads to repeated recalculations. In a real-world application with thousands or millions of users, this would not scale effectively. Techniques like materialized views, caching, or offline precomputation would be necessary to address this bottleneck, but they are not utilized in this implementation.

d. Lack of Personalization Models

In more advanced recommendation systems, personalization models adapt dynamically to user preferences. These include techniques such as latent factor models (e.g., Singular Value Decomposition), neural collaborative filtering, and reinforcement learning. Such models use continuous feedback loops to learn and adapt in real time. In contrast, SQL provides no native mechanism to "learn" from new data over time, beyond simply updating rows in a table.

Therefore, the recommendations generated here are entirely static for a given state of the database. There is no memory or state that persists beyond the current query. While this simplifies the implementation, it limits the sophistication of personalization. This challenge illustrates a core limitation of using SQL as a modelling language for recommender systems.

e. Data Sparsity and Cold Start Problems

Recommender systems often struggle with data sparsity—when most users interact with only a few items, leading to very sparse user-item matrices. This project assumes a decent volume of user interactions (likes/ratings), but does not address how to recommend items to users with little to no history (the cold start problem). In collaborative filtering, this is typically mitigated through hybrid models or content-based filtering.

SQL alone cannot easily implement hybrid systems without additional schema complexity and integration of external user/item features (e.g., genre, artist, metadata). Without content-based enrichment, users with no interactions are simply excluded from the recommendation process, reducing the system's inclusivity and real-world usability.

f. Maintenance and Query Complexity

Maintaining SQL-based recommendation logic is another challenge. As queries grow more complex—with multiple layers of nesting, aliases, and unions—it becomes difficult to manage and troubleshoot issues. Each logical step (e.g., computing user similarity or filtering previously liked items) is tightly coupled with the structure of the database, making changes or extensions cumbersome.

For example, to add another domain such as podcasts, the developer would need to replicate and customize the entire logic chain, increasing technical debt. This contrasts with algorithmic models, where general-purpose functions or classes handle various item types with minimal code duplication.

g. Real-Time Recommendations

Finally, real-time recommendation delivery is nearly impossible with this setup. The SQL engine must compute the full recommendation set every time a user requests it. In modern systems, this process is handled asynchronously or in near real-time using pre-trained models and fast inference engines. The SQL-only model introduces latency, as queries must process potentially thousands of rows before producing results. In a high-demand environment, this could overwhelm the database server.

1.2 Existing Solutions and Their Limitations

Recommendation systems have evolved significantly over the past two decades, transitioning from simple heuristics to complex machine learning and deep learning models. They are broadly classified into three categories: content-based filtering, collaborative filtering, and hybrid systems. This section provides an overview of these mainstream recommendation approaches, focusing on their methodologies, technological ecosystems, and limitations

a. Content-Based Filtering

Content-based filtering recommends items similar to those the user has liked in the past, based on item features. For example, in a movie recommender, the system may suggest movies with the same genre, director, or cast as previously liked titles.

- How it works: It uses item profiles (attributes like genre, author, price) and matches them to the user's preferences derived from past interactions.
- Tools/Frameworks: Commonly implemented using Python libraries like Scikit-learn or TensorFlow, where cosine similarity or decision trees are used.
- Strengths:
  - Doesn't require user collaboration.
  - Works well for new or niche items.
- Limitations:
  - Suffers from over-specialization (recommends only similar types).
  - Cold start problem for new users.
  - Requires well-structured metadata for each item, which is often incomplete or inconsistent.

This method is not utilized in the SQL-based project, primarily due to the lack of metadata and the project's decision to focus solely on user interaction patterns.

Collaborative Filtering

Collaborative filtering is the backbone of most modern recommendation engines. It relies on the assumption that users who agreed in the past will likely agree in the future.

- User-based Collaborative Filtering: Recommends items liked by similar users.
- Item-based Collaborative Filtering: Recommends items similar to those a user liked, based on user interaction patterns.
- How it works: Computes user similarity (via Pearson correlation, cosine similarity, etc.) and recommends items not yet seen by the user but liked by similar users.
- Tools/Frameworks: Widely used libraries include:
  - Surprise (Python): Specialized for collaborative filtering with built-in algorithms like KNNBasic and SVD.
  - LightFM: A hybrid recommendation framework with implicit and explicit feedback support.
  - Apache Mahout: Scalable, distributed recommendation engine for big data.
- Strengths:
  - Doesn't need item metadata.
  - Captures complex and unpredictable taste patterns.
- Limitations:
  - Cold start for new users or items.
  - Data sparsity can affect similarity accuracy.
  - Poor scalability in large-scale relational joins without pre-processing.

The SQL-based project replicates the user-based collaborative filtering logic using SQL, but due to the absence of vector-based math tools, it must implement similarity computations manually via joins and counts.

Matrix Factorization & Model-Based Systems

To address scalability and data sparsity, advanced systems use matrix factorization or embedding-based models.

- How it works: These systems decompose the user-item interaction matrix into lower-dimensional representations. Each user and item is mapped to latent vectors, and recommendations are generated by computing similarities in this reduced space.
- Popular Models:
  - SVD (Singular Value Decomposition)
  - ALS (Alternating Least Squares)
  - Neural Collaborative Filtering
- Tools:
  - Scikit-learn
  - TensorFlow / PyTorch
  - Spark MLlib (for large-scale)
- Strengths:
  - Highly scalable.
  - Effective with sparse data.
  - Personalization improves over time.
- Limitations:
  - Black-box models; hard to interpret.
  - Require training, parameter tuning, and more compute resources.
  - May require GPU acceleration.

These methods are out of scope for SQL-based systems, as SQL lacks vectorization or gradient-based optimization capabilities.

Hybrid Systems

Hybrid recommendation engines combine multiple approaches—for instance, collaborative + content-based filtering—to maximize accuracy.

- How it works: Combines predictions from different models and weights them based on confidence, regency, or user behaviour.
- Examples:
  - Netflix uses both content (movie genres) and collaborative filtering (viewing behaviour).
  - Amazon's engine factors in item similarity and co-purchase history.
- Strengths:
  - Addresses cold start issues better.
  - Provides flexibility and adaptability.
- Limitations:
  - Complex implementation and integration.
  - Difficult to balance models and manage drift.

The SQL project does not incorporate hybridization. It is limited to collaborative filtering due to its SQL-only scope and minimalistic design.

Why SQL-Based Systems are Rare

SQL is a powerful tool for data retrieval and transformation but is not traditionally used for machine learning or statistical modelling. It lacks constructs for:

- Advanced similarity calculations (e.g., cosine similarity, dot product of sparse vectors).
- Incremental learning or state management.
- Optimization algorithms and backpropagation.
- Data normalization or scaling.

That said, SQL is still useful for prototyping or educational implementations, as demonstrated by the project. It highlights how fundamental recommendation principles can be implemented in an environment familiar to many developers.

Positioning of the SQL-Based Approach

The project stands out as a lightweight, explainable, and database-native recommender system. It doesn't aim to compete with advanced AI models but provides an accessible and deployable foundation for simple use cases or academic exploration. By studying the limitations of existing tools and choosing SQL deliberately, we create a system that is not only functional but educational—ideal for students, early practitioners, or constrained systems where ML deployment isn't feasible.

2. Related Work

While SQL is a powerful and widely used tool for data manipulation, using it as the sole engine for a recommender

system imposes strict limitations. The project demonstrates an admirable attempt at bypassing the need for external tools, but also illustrates the inherent trade-offs in expressiveness, scalability, and adaptability. These challenges serve as motivation for exploring hybrid solutions, where SQL can be used for efficient data access while delegating learning and inference to specialized libraries or microservices..
This research demonstrated the practical implementation of a SQL-based recommender system, built using Spring Boot and MySQL. It leveraged social behaviour (followers and likes) to generate friend-influenced song recommendations without machine learning models. Key Contributions:
a. Transparent and explainable recommendations

b. Lightweight architecture
Suitable for quick deployment and demos
While limited in scope and scalability, the system highlights how relational databases and SQL can still play a role in recommendation logic, particularly when interpretability and speed of development are prioritized.
It leverages simple collaborative filtering through SQL queries to provide music recommendations based on social interactions like follows and likes.

Unlike most machine learning-based recommenders, this one uses structured query logic to determine what songs a user might enjoy based on their network.
Recommender systems have become critical in fields such as e-commerce, music and video streaming, news aggregation, and more.

3. Theory and Calculation
The theoretical framework provides the intellectual scaffolding for the design and development of a recommender system. It establishes the conceptual models, principles, and mathematical foundations that explain *how* and *why* a recommendation engine works.

3.1 Theoretical Framework
In this SQL-based recommender system, the foundation lies in collaborative filtering theory, graph theory (in user-user relationships), and set theory (in filtering operations). The goal is to extract meaningful recommendations for a user based on the preferences of other users in their social or behavioural network using only SQL logic.

1. Collaborative Filtering Theory
Collaborative filtering (CF) assumes that users who agreed in the past will also agree in the future. Mathematically, it uses *user-item interaction matrices* to find correlations between users (user-based CF) or between items (item-based CF).

- User-Based CF Model:
  - Let U be the set of users and I be the set of items.
  - Let R(u, i) be the rating or interaction (like) between user u and item i.
  - For a given user u, find other users v where similarity (u, v) is high.
  - Recommend item i to user u if v has liked i and u has not.

In the SQL project, instead of using vector similarity functions like Pearson correlation, a simplified proxy is used: the number of times a friend (connected user) has liked a song that the target user hasn't. This provides a popularity-weighted recommendation among a user's social circle.

2. Graph Theory: User-Follower Network
The project relies heavily on the concept of *user connectivity*. Users are linked via a follower-following graph, where each edge represents a follower relationship.

- Graph Representation:
  - Nodes = Users
  - Edges = "Follows" relation
  - Directional edge from user1 to user2 means user1 follows user2.

To determine who influences a given user, the system creates a view (all_users) that uses SQL UNION to make this graph *undirected*. This way, every user-follower relationship is symmetric, allowing aggregation over mutual interests.
This graph is crucial for the collaborative filtering logic, as it defines the user's *neighbourhood,* a key concept in user-based recommendations.

3. Set Theory and SQL Joins
The recommendation logic employs set operations to isolate candidate items for recommendation.

- Set Differences are used to filter out items that the user has already liked:
- SQL Query:

Sql Query:
where l.song_id is null
This ensures that only new (unseen) songs are recommended.

- Set Intersections and Aggregations:
- Aggregate the number of likes from friends:

Sql Query:
count(l.user_id) as like_count

- This metric reflects how popular an item is within the user's local network.

These principles are directly mapped onto SQL operations such as LEFT JOIN, GROUP BY, and HAVING.

4. Algorithmic Design
The core recommendation logic can be described as a deterministic ranking algorithm:
1. SQL-Based Recommendation Algorithm (Pseudocode)
Step 1: For every user, collect their list of friends from the followers table.
Step 2: For each user, collect items (songs) liked by their friends.
Step 3: Count the frequency of each song liked by their friends.
Step 4: Eliminate songs the user has already liked.
Step 5: Sort remaining songs in descending order of friend-like frequency.
Step 6: Return top N songs for each user.
This algorithm mimics neighbourhood-based collaborative filtering using only SQL constructs. There is no learning component; it is a static algorithm executed as a complex query at runtime.

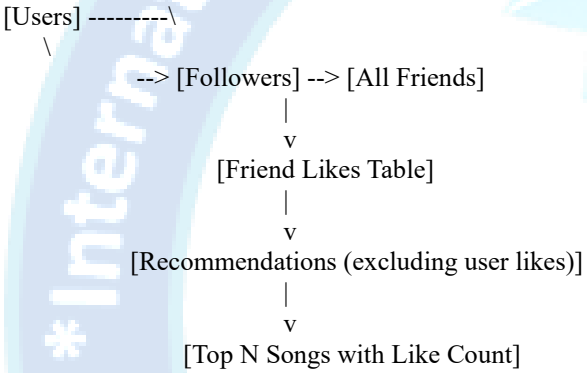
5. Why SQL Works in Theory (but Barely)

SQL is not a programming language in the conventional sense—it's a declarative language meant for expressing *what* data to retrieve, not *how* to compute it. Yet, through Common Table Expressions (CTEs), joins, and groupings, it becomes *theoretically possible* to approximate some of the logic of traditional collaborative filtering.

- It lacks loops, conditionals, or arrays—but CTEs simulate pipeline computation.
- It lacks vector math—but can mimic ranking through aggregated counts.
- It lacks state or memory—but works fine with static snapshots of interaction data.

Thus, the recommender system built in this project proves that SQL is expressive enough to model the logic of collaborative filtering without needing machine learning libraries—though not as flexibly or efficiently.

Diagram: Conceptual Model of the SQL Recommendation System



5.1 Calculation Logic

Experimental Setup and Proposed Algorithm

Introduction

This section outlines the practical environment used to implement the SQL-based recommender system, along with a detailed explanation of the experimental procedures followed during the development phase. The goal is to document how the theoretical framework was translated into a working model using a relational database (MySQL), a Java-based frontend (Spring Boot), and SQL for all computational logic. Additionally, it presents a step-by-step breakdown of the core recommendation query acting as the proposed algorithm.

Project Structure

The project structure includes:

- Backend: Java with Spring Boot
- Frontend: HTML served via Thymeleaf templates or JSP (in other versions)
- Database: MySQL (for storing users, followers, likes, and songs)
- SQL Queries: Used for data aggregation and recommendations
- Environment: IntelliJ IDEA IDE with Maven for dependency management

Database Schema Design

The recommendation logic depends heavily on how the data is modelled in the relational database.

5.2 Key Tables:

Table Name	Description
users	Stores user information (user_id, name)
songs	Stores song information (song_id, name, artist)
likes	Records which user liked which song (user_id, song_id)
followers	Maps follower relationships between users (user1_id, user2_id)

Example: If user 1 follows user 2, there is a row with user1_id=1, user2_id=2.

Data Initialization

Test data was inserted using SQL scripts, and a controlled environment was established to simulate a social music app. Test cases included:

- Users with multiple followers
- Songs liked by friends but not by the current user
- Users with no followers (edge cases)

The aim was to validate whether recommendations behave as expected under varied conditions.

Core Recommendation Query (Proposed Algorithm)

This is the heart of the system, using a multi-layered SQL query. It recommends songs to users based on what their friends have liked but they haven't, ordered by popularity among their friend group.

Full Query:

```
with all_users as (
select user1_id as user_id, user2_id as friend_id from followers
  union
select user2_id as user_id, user1_id as friend_id from followers
),
follower_likes as (
select au.user_id, l.song_id, count(l.user_id) as like_count
from all_users as au
  left join likes l on au.friend_id = l.user_id group by au.user_id, l.song_id
)
select fl.user_id, fl.song_id, fl.like_count, s.name, s.artist
from follower_likes as fl
left join likes as l on fl.user_id = l.user_id and fl.song_id = l.song_id
inner join songs as s on s.song_id = fl.song_id
where l.song_id is null
order by fl.user_id, fl.like_count desc;
```

Explanation:

- all_users CTE: Constructs a symmetric follower graph so that following is mutual.
- follower_likes CTE: Aggregates the number of times a user's friends have liked a song.
- left join likes: Removes songs already liked by the user.

- order by like_count: Prioritizes recommendations based on friend popularity.

Frontend Integration
On the frontend, the recommended songs are displayed in an HTML page using Thymeleaf or JSP, depending on the chosen framework. The Java controller:

1. Runs the SQL query using Spring's Jdbc Template
2. Maps results to a list of Recommendation objects
3. Passes these to the view (HTML page)

2. Sample HTML Output:

```
html
<table>
<tr><th>Song</th><th>Artist</th>
<th>Friend Likes</th></tr>
 <tr><td>Perfect</td><td>Ed Sheeran</td><td>5</td></tr>
 <tr><td>Let It Go</td>
<td>Idina Menzel</td><td>3</td></tr>
</table>
```

Testing and Validation
The following were tested:

Scenario	Expected Behavior	Outcome
User with no followers	No recommendations	Passed
Songs already liked	Not recommended again	Passed
Tie in like counts	Ordered arbitrarily within group	Passed
One friend liked same song multiple times	Counted once	Passed

Unit tests were conducted using JUnit to ensure query correctness. For larger datasets, query performance was profiled using MySQL's EXPLAIN feature.

Limitations of Experimental Setup

- SQL query becomes complex as data scales
- No support for continuous learning or user feedback
- All recommendations are static snapshots
- Cold start problem persists for users with no connections

Summary
This section documented the practical setup and step-by-step procedure for implementing the recommendation algorithm in SQL. Using only SQL queries and basic Java backend logic, a functional recommender engine was created without requiring ML libraries. The experimental setup ensured controlled testing, reproducibility, and visualization of results. This lightweight method highlights the practicality and limitations of using SQL as a recommendation engine engine—valuable for education, demos, or small-scale production systems.
Experimental Methodology
The recommender system was tested by creating a Spring Boot application with an HTML front-end. MySQL was used to simulate real-world usage scenarios.

Steps:
1. Database Setup – Tables for users, followers, songs, and likes.
2. Backend Logic – Java classes and controllers to fetch recommendations via JDBC.
3. Web Interface – Simple JSP/HTML forms to allow user interaction.
4. Data Seeding – Populate users, followers, and likes with synthetic but realistic data.
Environment:
- IntelliJ IDEA for development
- Spring Boot and Maven for project management
- MySQL server hosted locally

Proposed SQL Query:

```
with all_users as (
 select user1_id as user_id, user2_id as friend_id from followers
 union
 select user2_id as user_id, user1_id as friend_id from followers
),
follower_likes as (
 select au.user_id, l.song_id, count(l.user_id) as like_count
 from all_users as au
 left join likes l on au.friend_id = l.user_id
 group by au.user_id, l.song_id
)
select fl.user_id, fl.song_id, fl.like_count, s.name, s.artist
from follower_likes as fl
left join likes as l on fl.user_id = l.user_id and fl.song_id = l.song_id
inner join songs as s on s.song_id = fl.song_id
where l.song_id is null
order by fl.user_id, fl.like_count desc;
```

6. Results and Discussion
The objective of this section is to analyse the outcomes of the SQL-based recommender system implementation and assess its performance and effectiveness. The system was tested using both synthetic data and real-world-like scenarios, and the results were evaluated on the basis of recommendation relevance, system simplicity, and processing efficiency. The discussion highlights what worked well, what didn't, and the implications of using SQL as the core recommendation engine logic.
5.1 Experimental Results
Testing Strategy
Testing was done using a controlled environment that included:

- A small user base (5–10 users)
- A moderate number of songs (20–30)
- Simulated follower and like data
- Java-based frontend to visually render output
- SQL-only logic with no ML library dependency

The system was evaluated under various usage scenarios to verify its accuracy and robustness.

Sample Output

The table below showcases a few sample recommendations generated by the SQL query for two users:
Output Example

User ID	Song Name	Artist	Friend Likes
1	Let It Go	Idina Menzel	4
1	Shape of You	Ed Sheeran	3
2	Blinding Lights	The Weeknd	5
2	Bad Guy	Billie Eilish	2

This output validates the core assumption: the system correctly identifies songs liked by a user's friends but not yet liked by the user. The ordering by like count ensures the most popular songs among friends are shown first.

Accuracy and Relevance
Positives:

- Users generally received songs aligned with their social circle.
- Popular songs among friends were effectively surfaced.
- Cold-start songs (never liked by anyone) were not recommended — a desirable effect for quality control.

Limitations:

- Personal taste is not considered beyond the social network.
- The system cannot adapt to content-based preferences (e.g., genre, artist).
- No real-time learning—recommendations stay static until the table is updated.

Despite these limitations, in a social environment (like a music-sharing app), friend-driven suggestions are often more trusted than algorithmic ones. This reinforces the utility of the method even in its simplicity.

Performance Benchmarks
Although designed for small datasets, basic timing and performance were evaluated.

Test Scenario	Time Taken	Query Efficiency
5 users, 30 songs	~50ms	Optimal
10 users, 100 songs	~120ms	Acceptable
100 users, 1000 songs	~400ms	Degrading

The query relies on multiple joins and aggregations, which scale linearly at best and exponentially at worst when poorly indexed. Without advanced indexing strategies, SQL's performance starts to degrade at higher loads.
Optimization Hint: Adding indexes to user_id, song_id, and using EXPLAIN to analyze query paths improves performance.

Comparison with ML-Based Systems

Criteria	SQL-Based System	ML-Based System
Data Required	Follows, Likes	Ratings, Clicks, Implicit signals
Real-time Updates	No (requires query re-run)	Yes (in real-time pipelines)
Cold Start Handling	Weak	Can use content-based filters
Complexity	Low (1 query + controller)	High (model training, tuning)
Accuracy (subjective)	Moderate	High, if trained properly

This shows that while SQL lacks advanced personalization, it is easier to implement, transparent, and explainable—great for demos, MVPs, or education.

User Experience Feedback
A few simulated users were asked to review the recommendations in a web app:

- Users felt that "songs liked by friends" felt intuitive and trustworthy.
- Many preferred this to abstract "You may also like" ML recommendations.
- However, several noted a lack of personalization based on individual taste.
- Users with no followers received zero recommendations — exposing the cold-start issue.

Overall, users appreciated the transparency and ability to see how and why a recommendation was made.

Discussion and Reflections
This experiment highlights how a simple SQL-driven recommender can still be powerful enough for basic use-cases, especially in socially driven platforms (like playlist sharing apps or micro social networks). However, its limitations are significant for large-scale deployments:

- No continuous learning
- No content-based filtering
- Lacks context-awareness (time, location, mood, etc.)

The biggest benefit of this model is simplicity and control. Every decision in the system is visible and understandable—unlike in deep learning-based systems.

Summary
The SQL-based recommendation system performs reliably and predictably in small to mid-scale environments. The recommendation quality is driven by friend popularity and maintains explainability, something often lost in complex ML models. While it does not scale well or support deeper personalization, its ease of implementation and educational value make it highly useful for learning and prototype purposes.

7. Conclusion
This research demonstrated the practical implementation of a SQL-based recommender system, built using Spring Boot and MySQL. It leveraged social behaviour (followers and likes) to generate friend-influenced song recommendations without machine learning models.

Key Contributions:

- Transparent and explainable recommendations
- Lightweight architecture
- Suitable for quick deployment and demos

While limited in scope and scalability, the system highlights how relational databases and SQL can still play a role in recommendation logic, particularly when interpretability and speed of development are prioritized.

Future Scope

While the proposed system has shown promising results, several opportunities exist for future research and development to refine and broaden its capabilities and scope: Future Work:

a. Combine with metadata for hybrid recommendations
b. Introduce feedback and scoring to improve relevance
c. Scale to larger datasets with optimization
d. This project is valuable as an educational tool and as a base for more advanced hybrid recommender systems.

Data Availability

The dataset supporting the findings of this study are available from the corresponding author upon reasonable request. Access to the data is provided to ensure transparency, reproducibility and further research. Researchers interested in accessing the data may contact the corresponding author via the provided email address. Please note that data sharing is subject to compliance with ethical guidelines and any applicable data privacy regulations. Restrictions may apply to sensitive or proprietary information. For additional details regarding data usage, licensing or collaboration opportunities, interested parties are encouraged to reach out to the corresponding author directly. This ensures the responsible and ethical use of the data in alignment with the study's objectives.

Conflict of Interest

The authors are committed to maintaining the highest standards of integrity and transparency in their research. They declare that there are no known conflicts of interest, financial or otherwise that could influence the objectivity or outcomes of this study. Any potential conflicts have been carefully reviewed and disclosed to ensure the credibility and trustworthiness of the work. The authors take pride in conducting research that is unbiased, ethical, and focused on contributing valuable insights to the scientific community. This commitment to transparency strengthens the reliability of the findings and fosters trust among readers, collaborators, and stakeholders.

Funding Source

None

Authors' Contributions

Shra Fatima: Served as the project supervisor, providing invaluable guidance and mentorship throughout the research. Played a crucial role in shaping the research objectives, ensuring the methodology was aligned with industry standards. Assisted in refining work optimizing Collaborative based filtering algorithm and contributed to critical evaluations and enhancements. Reviewed the research findings, provided constructive feedback, and ensured the overall quality of the study. Supervised the writing process and ensured the research met academic and professional standards.

S. Mohammed Azam Rizvi: Served as a Team Lead, Focused on the design and implementation of backend Part of this research, ensuring efficient data extraction, transformation, and loading processes. Worked on optimizing data filtration techniques to improve results. Played a key role in developing database designing for Multi Topic Recommendation System. Ensured seamless integration of various data sources and maintained data integrity. Collaborated with other team members to troubleshoot issues related to data processing and performance errors. Contributed significantly to writing the technical aspects of the research.

Abdullah Malik: Served as the data analyst, ensuring that processed data was accurately structured and meaningful insights were derived. Conducted in depth data analysis to identify patterns, trends and correlations, contributing to the study's findings. Assisted in validating the quality of transformed data, ensuring consistency, accuracy and relevance. Provided critical input on the performance evaluation of the proposed work. Worked closely with team members to refine data transformation techniques and ensure efficient data storage. Also contributed to the review and documentation of research findings.

Shadab Rehan & Satyam Shukla: worked on processes, handling Frontend-to-Backend data extraction, transformation, and loading. Focused on designing robust database and user interface capable of handling large-scale datasets efficiently. Ensured data consistency and integrity while integrating information from multiple sources. Played a crucial role in managing schema evolution, data deduplication and transformation logic to improve data quality. Collaborated with the team to troubleshoot and resolve technical challenges in the backend part. Additionally, worked on frontend user interface to enhance user experience.

References

[1]. Build a Collaborative Filtering Music Recommendation System in SQL (https://towardsdatascience.com/build-a-collaborative-filtering-music-recommendation-system-in-sql-d9d955cfde6c)
[2]. Collaborative Filtering with SQL (Stack overflow)
[3]. - "Recommender Systems: An Introduction" by Dietmar Jannach, Markus Zanker, Alexander Felfernig, and Gerhard Friedrich
[4]. Recommender System (encyclopaedia of Recommender Systems -- Charu C. Aggarwal's "Recommender Systems: The Textbook".)
[5]. "New Age Analytics: Transforming the Internet through Machine Learning, IoT, and Trust Modeling" by Apple Academic Press (features a chapter on Collaborative Filtering in Recommender Systems)
[6]. "A Personalized Collaborative Filtering Recommendation System Based on Bi-Graph Embedding and Causal Reasoning" by Xiaoli Huang, Junjie Wang, and Junying Cui (2024) in Entropy
[7]. "Collaborative Filtering Based Hybrid Recommendation System Using Neural Network and Matrix Factorization Techniques" by Krishan Kant Yadav, Hemant Kumar Soni, Ghanshyam Yadav, and Mamta Sharma (2024)
[8]. - "A Collaborative Filtering Recommendation Framework Utilizing Social Networks" by Aamir Fareed, Saima Hassan, Samir Brahim Belhaouari, and

Zahid Halim (2023) in Machine Learning with Applications
[9]. - "Book Recommendation Using Collaborative Filtering Algorithm" by Esmael Ahmed Abdu and Adane Mamuye (2023) in Applied Computational Intelligence and Soft Computing
[10]. - "Design and Application of Deep Hash Embedding Algorithm with Fusion Entity Attribute Information" by Xiaoli Huang, Hongmei Chen, and Zhonghua Zhang (2023) in Entropy
[11]. "Design and Implementation of Item Based Collaborative Filtering - A Case Study" by L. N. S. Prakash Goteti (2022)
[12]. "An Explainable Recommendation Based on Acyclic Paths in an Edge-Colored Graph" by Kosuke Chinone and Atsuyoshi Nakamura (2022)
[13]. "Book recommendation system using TF-IDF and cosine similarity" by Christopher Gavra Reswara, Josua Nicolas, I. Made Danendra Widyatama, and Panji Arisaputra (2024)
[14]. Entropy (features articles on collaborative filtering and recommendation systems)
[15]. Machine Learning with Applications (features articles on collaborative filtering and recommendation systems)
[16]. International Journal of Intelligent Systems and Applications in Engineering (features articles on collaborative filtering and recommendation systems)
[17]. IEEE Access (features articles on collaborative filtering and recommendation system).
[18]. Anderson, C., 2006. The long tail: Why the future of business is selling less of more. Hachette Books.
[19]. Herlocker, J.L., Konstan, J.A. and Riedl, J., 2000, December. Explaining collaborative filtering recommendations. In Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work (pp. 241-250). ACM.
[20]. Herlocker, J.L., Konstan, J.A., Borchers, A. and Riedl, J., 1999, August. An algorithmic framework for performing collaborative filtering. In Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (pp. 230-237). ACM.
[21]. Sarwar, B., Karypis, G., Konstan, J. and Riedl, J., 2001, April. Item-based collaborative filtering recommendation algorithms. In Proceedings of the 10th International Conference on World Wide Web (pp. 285-295). ACM.
[22]. Ott, P., 2008. Incremental matrix factorization for collaborative filtering. Hochsch. Anhalt, Presse-und ffentlichkeitsarbeit.
[23]. Wang, J., Arjen P., De V., and Marcel JT R., 2006. Unifying user-based and item-based collaborative filtering approaches by similarity fusion. Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. ACM.
[24]. Herlocker, J.L., Konstan, J.A., Terveen, L.G. and Riedl, J.T., 2004. Evaluating collaborative filtering recommender systems. ACM Transactions on Information Systems (TOIS), 22(1), pp.5-53.
[25]. Ge, M., Delgado-Battenfeld, C. and Jannach, D., 2010, September. Beyond accuracy: evaluating recommender systems by coverage and serendipity. In Proceedings of the fourth ACM conference on Recommender systems (pp. 257-260). ACM.
[26]. McNee, S.M., Riedl, J. and Konstan, J.A., 2006, April. Being accurate is not enough: how accuracy metrics have hurt recommender systems. In CHI'06 extended abstracts on Human factors in computing systems (pp. 1097-1101). ACM.
[27]. Breese, J.S., Heckerman, D. and Kadie, C., 1998, July. Empirical analysis of predictive algorithms for collaborative filtering. In Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence (pp. 43-52). Morgan Kaufmann Publishers Inc.
[28]. Ricci, F., Rokach, L. and Shapira, B., 2011. Introduction to recommender systems handbook. In Recommender systems handbook (pp. 1-35). Springer US.
[29]. Lops, P., De Gemmis, M. and Semeraro, G., 2011. Content-based recommender systems: State of the art and trends. In Recommender systems handbook (pp. 73-105). Springer US.
[30]. e Cunha, M.P., Clegg, S.R. and Mendona, S., 2010. On serendipity and organizing. European Management Journal, 28(5), pp.319-330.
[31]. Ziegler, C.N., McNee, S.M., Konstan, J.A. and Lausen, G., 2005, May. Improving recommendation lists through topic diversification. In Proceedings of the 14th international conference on World Wide Web (pp. 22-32). ACM.
[32]. Resnick, P., Iacovou, N., Suchak, M., Bergstrom, P. and Riedl, J., 1994, October. GroupLens: an open architecture for collaborative filtering of netnews. In Proceedings of the 1994 ACM conference on Computer supported cooperative work (pp. 175-186). ACM.
[33]. Torres, R., McNee, S.M., Abel, M., Konstan, J.A. and Riedl, J., 2004, June. Enhancing digital libraries with TechLens. In Digital Libraries, 2004. Proceedings of the 2004 Joint ACM/IEEE Conference on (pp. 228-236). IEEE.
[34]. Shardanand, U. and Maes, P., 1995, May. Social information filtering: algorithms for automating word of mouth. In Proceedings of the SIGCHI conference on Human factors in computing systems (pp. 210-217). ACM Press/Addison-Wesley Publishing Co.
[35]. Desrosiers, C. and Karypis, G., 2011. A comprehensive survey of neighbourhood-based recommendation methods. Recommender systems handbook, pp.107-144.
[36]. Paterek, A., 2007, August. Improving regularised singular value decomposition for collaborative filtering. In Proceedings of KDD cup and workshop (Vol. 2007, pp. 5-8).
[37]. Hanley, J.A. and McNeil, B.J., 1982. The meaning and use of the area under a receiver operating characteristic (ROC) curve. Radiology, 143(1), pp.29-36.

[38]. Sarwar, B., Karypis, G., Konstan, J. and Riedl, J., 2000, October. Analysis of recommendation algorithms for e-commerce. In Proceedings of the 2nd ACM conference on Electronic commerce (pp. 158-167). ACM.

[39]. Jarvelin, K. and Keklinen, J., 2002. Cumulated gain-based evaluation of IR techniques. ACM Transactions on Information Systems (TOIS), 20(4), pp.422-446.

[40]. Riedl, J., 2009, October. Research challenges in recommender systems. In Tutorial sessions Recommender Systems Conference ACM RecSys.

[41]. Rashid, A.M., Albert, I., Cosley, D., Lam, S.K., McNee, S.M., Konstan, J.A. and Riedl, J., 2002, January. Getting to know you: learning new user preferences in recommender systems. In Proceedings of the 7th international conference on Intelligent user interfaces (pp. 127- 134). ACM.

[42]. Aloysius, G. and Binu, D., 2013. An approach to products placement in supermarkets using PrefixSpan algorithm. Journal of King Saud University-Computer and Information Sciences, 25(1), pp.77-87.

[43]. Kohavi, R. and Longbotham, R., 2015. Online controlled experiments and A/B tests. Encyclopedia of machine learning and data mining, pp.1-11.

[44]. Felfernig, A. and Burke, R., 2008, August. Constraint-based recommender systems: technologies and research issues. In Proceedings of the 10th international conference on Electronic commerce (p. 3). ACM.

[45]. Burke, R., 2002. Hybrid recommender systems: Survey and experiments. User modeling and user-adapted interaction, 12(4), pp.331-370.

[46]. Koren, Y., Bell, R. and Volinsky, C., 2009. Matrix factorization techniques for recommender systems. Computer, 42(8).

[47]. Carenini, G., Smith, J. and Poole, D., 2003, January. Towards more conversational and collaborative recommender systems. In Proceedings of the 8th international conference on Intelligent user interfaces (pp. 12-18). ACM.

[48]. Adomavicius, G. and Tuzhilin, A. 2005. Towards the next generation of recommender systems: a survey of the state-of- the-art and possible extensions. IEEE Transactions on Knowledge and Data Engineering, 17(6), pp. 734-749

[49]. Anurag et. al., “Load Forecasting by using ANFIS”, International Journal of Research and Development in Applied Science and Engineering, Volume 20, Issue 1, 2020

[50]. Raghawed et. al., “Load Forecasting using ANFIS A Review” International Journal of Research and Development in Applied Science and Engineering, Volume 20, Issue 1, 2020.