CS157C: NoSQL Database Systems

Final Project Report

Social Network Graph Application

Team Members

Aditya Dawadikar <aditya.dawadikar@sjsu.edu>

Timothy <timothy@sjsu.edu>

Jakob <jakob@sjsu.edu>

December 2024

Contents

1 Team Information

Name	Email	Responsibilities
Aditya Dawadikar	aditya.dawadikar@sjsu.edu	Dataset, Schema, Search & Explore
Timothy	timothy@sjsu.edu	User Management (UC-1 to UC-4)
Jakob	jakob@sjsu.edu	Social Graph Features (UC-5 to UC-9)

Table 1: Team Member Information

2 Property Graph Schema

2.1 Node: User

The `User` node represents a user in the social network.

Property	Type	Description
userId	STRING	Zero-padded unique ID (0001...5000)
username	STRING	Generated unique username
email	STRING	User email address
name	STRING	Full display name
bio	STRING	Short profile biography
passwordHash	STRING	bcrypt hashed password

Table 2: User Node Properties

2.2 Relationship: FOLLOWS

The `FOLLOWS` relationship represents a directed edge from User A to User B, indicating that User A follows User B.

```
(UserA:User)-[:FOLLOWS]->(UserB:User)
```

3 Dataset Information

3.1 Dataset Source

- **Name:** Synthetic Social Network Dataset
- **Type:** Generated using Python scripts
- **Size:** 5,000 users, 15,000+ FOLLOWS relationships

3.2 Dataset Description

The dataset is synthetically generated to simulate a realistic social network with:

- **User Generation:** 5,000 unique users with randomized usernames (Adjective + Noun + Number format)
- **Cluster Behavior:** Users are organized into 20 clusters for realistic community structure
- **Influencers:** 20 users with 1,000-3,500 followers each
- **Ghost Accounts:** 800 users with 0 followers (inactive accounts)
- **Normal Users:** Follow 3-50 users, primarily within their cluster

3.3 Data Processing and Loading

The data was generated and loaded using the following Python scripts:

1. `generate_users.py` - Creates 5,000 unique user records
2. `generate_graph.py` - Creates FOLLOWS relationships with realistic patterns
3. `ingest_graph.py` - Loads CSV data into Neo4j

3.4 Cypher Statements for Data Creation

Create Constraints:

```
CREATE CONSTRAINT user_id_unique IF NOT EXISTS
FOR (u:User) REQUIRE u.userId IS UNIQUE;

CREATE CONSTRAINT username_unique IF NOT EXISTS
FOR (u:User) REQUIRE u.username IS UNIQUE;
```

Load Users:

```
UNWIND $rows AS row
MERGE (u:User {userId: row.userId})
SET u.username = row.username,
    u.email = row.email,
    u.name = row.name,
    u.bio = row.bio,
    u.passwordHash = row.passwordHash;
```

Load FOLLOWS Relationships:

```
UNWIND $rows AS row
MATCH (f:User {userId: row.followerId})
MATCH (t:User {userId: row.followeeId})
MERGE (f)-[:FOLLOWS]->(t);
```

4 Use Case Evidence

4.1 UC-1: User Registration

[Timothy's implementation]

Screenshot:

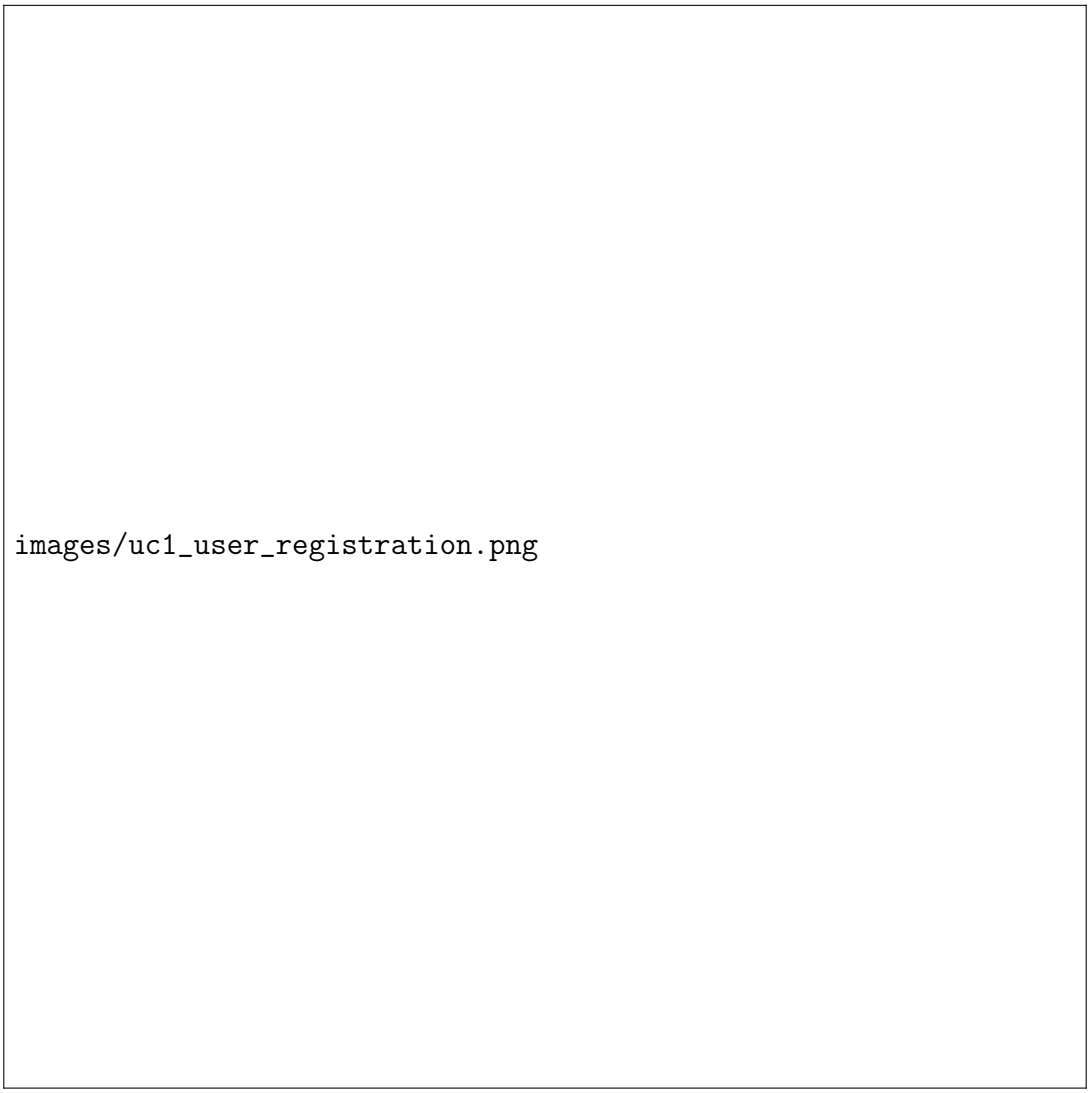

Figure 1: UC-1: User Registration

4.2 UC-2: User Login

[Timothy's implementation]

Screenshot:

images/uc2_user_login.png

Figure 2: UC-2: User Login

4.3 UC-3: View Profile

[Timothy's implementation]

Screenshot:

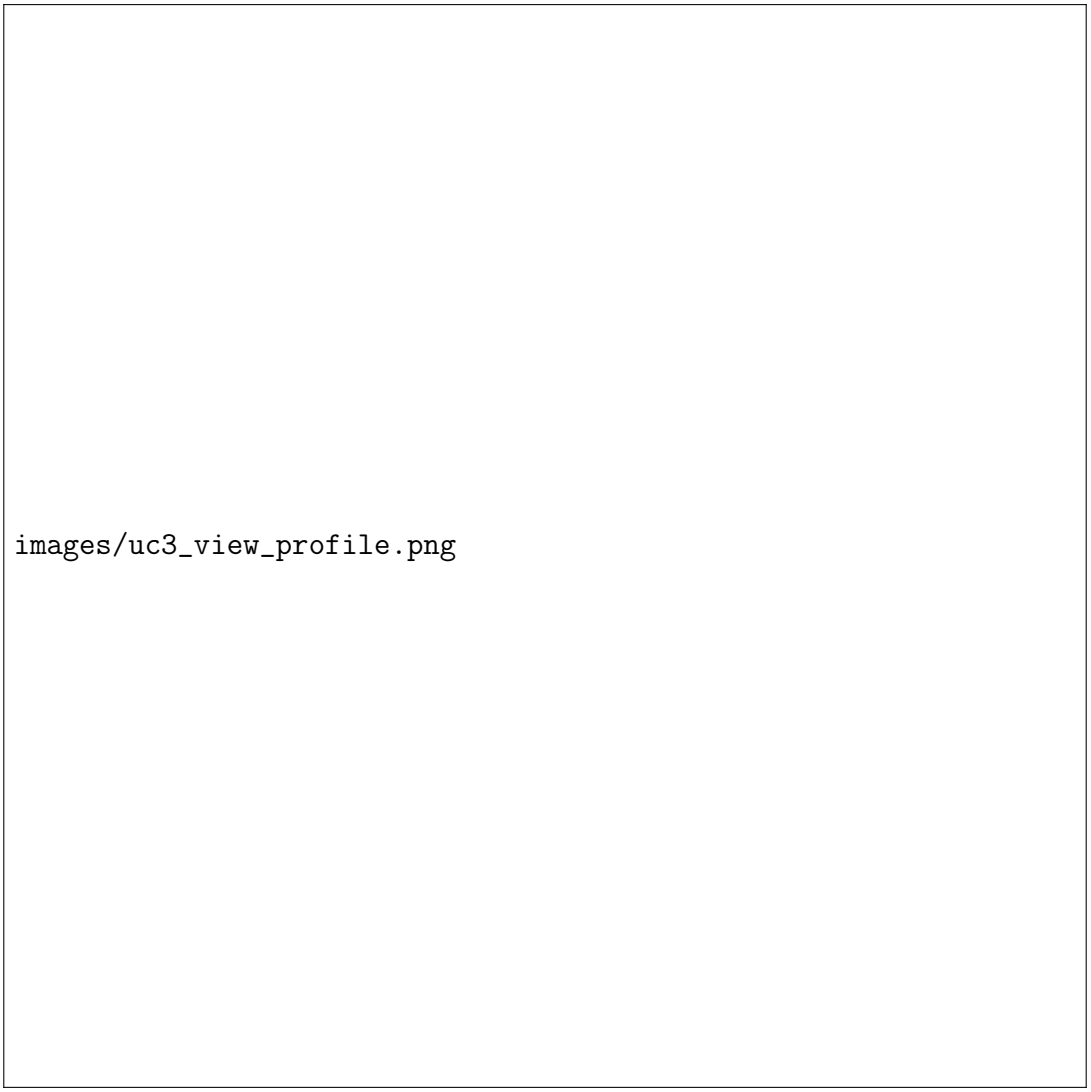

Figure 3: UC-3: View Profile

4.4 UC-4: Edit Profile

[Timothy's implementation]

Screenshot:

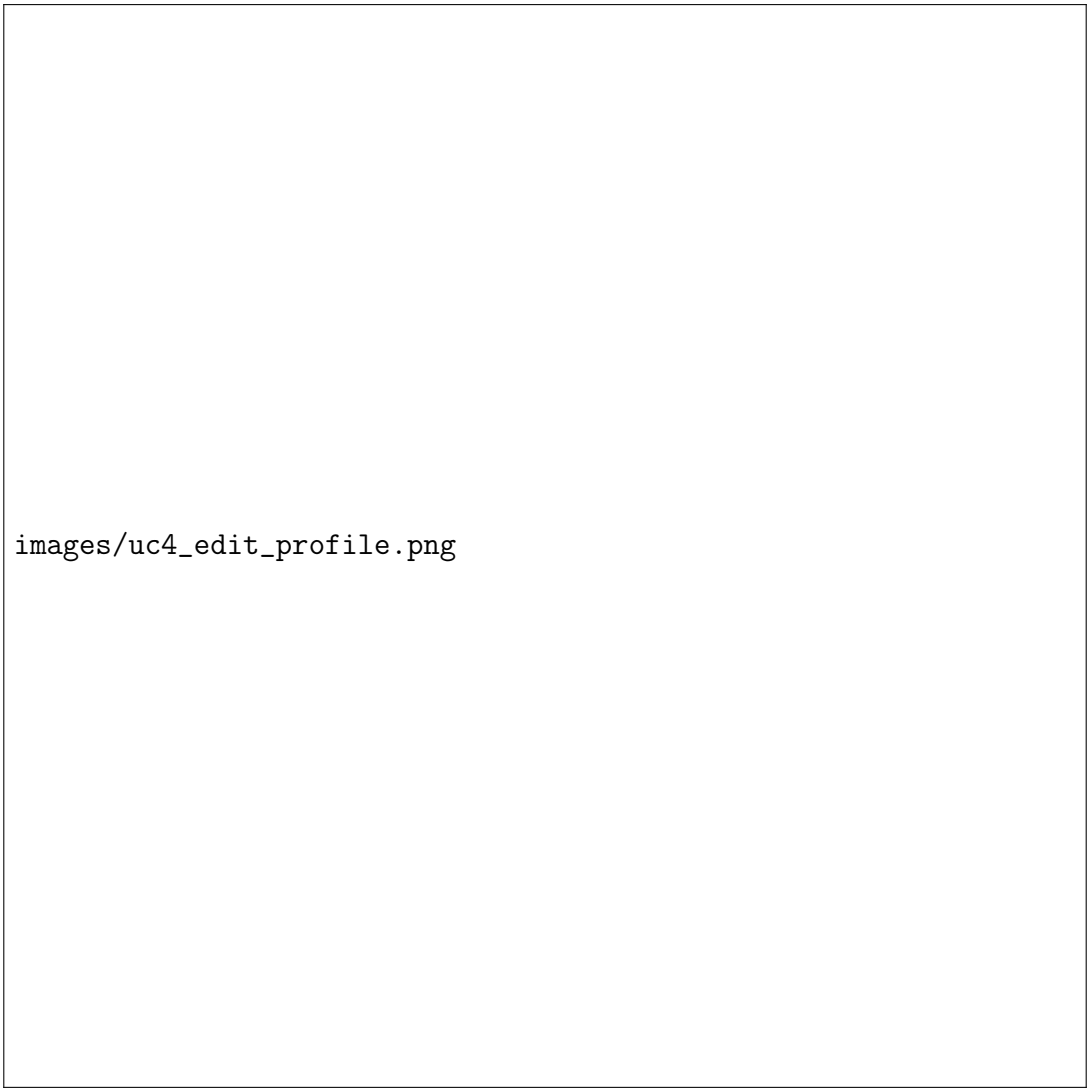

Figure 4: UC-4: Edit Profile

4.5 UC-5: Follow Another User

Description: A user can follow another user, creating a `FOLLOWS` relationship in Neo4j. The relationship is stored as a directed edge in the graph database, representing a one-way connection from the follower to the user being followed.

Cypher Query:

```
// UC-5: Follow Another User
// Creates a FOLLOWS relationship between two users

MATCH (follower:User {userId: $followerId})
MATCH (target:User {userId: $targetId})
MERGE (follower)-[r:FOLLOWS]->(target)
RETURN
    follower.userId AS followerId,
    follower.username AS followerUsername,
    target.userId AS targetId,
    target.username AS targetUsername,
    type(r) AS relationship
```

Implementation Notes:

- Uses `MERGE` to prevent duplicate relationships
- Validates that a user cannot follow themselves
- Checks if relationship already exists before creating
- Returns confirmation of the created relationship

Screenshot:

Figure 5: UC-5: Follow Another User - Creating a FOLLOWS relationship

4.6 UC-6: Unfollow a User

Description: A user can unfollow another user, removing the `FOLLOWS` relationship from the graph database. This operation deletes the directed edge between the two users.

Cypher Query:

```
// UC-6: Unfollow a User
// Removes the FOLLOWS relationship between two users
```

```
3
4  MATCH (follower:User {userId: $followerId})
5        -[r:FOLLOWS]->
6        (target:User {userId: $targetId})
7  DELETE r
8  RETURN
9      follower.userId AS followerId,
10     follower.username AS followerUsername,
11     target.userId AS targetId,
12     target.username AS targetUsername,
13     'DELETED' AS status
```

Implementation Notes:

- Uses pattern matching to find the exact relationship
- Validates relationship exists before deletion
- Only deletes the specific `FOLLOWS` edge, not the nodes
- Provides feedback on successful deletion

Screenshot:

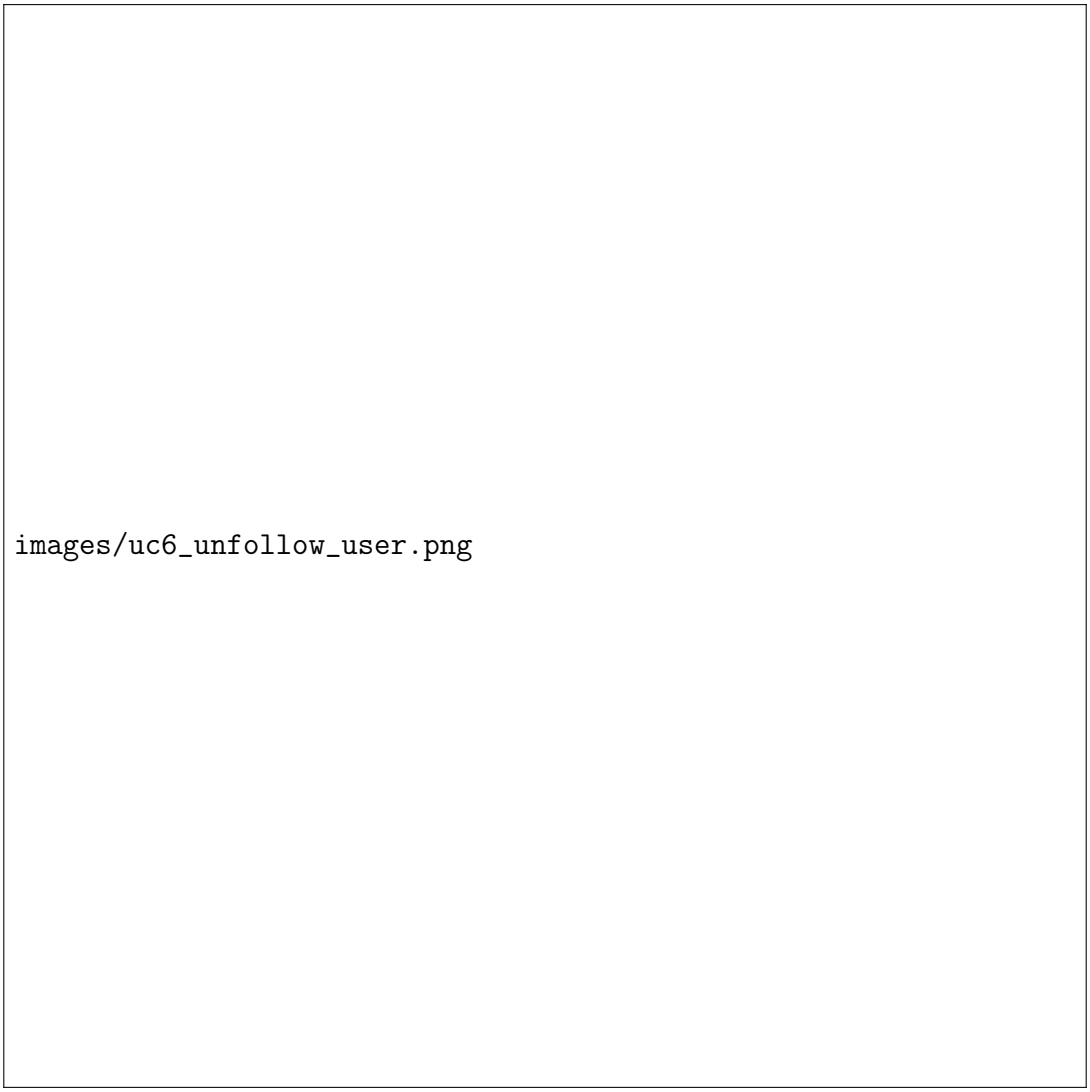

Figure 6: UC-6: Unfollow a User - Removing a FOLLOWS relationship

4.7 UC-7: View Friends/Connections

Description: A user can see a list of people they are following (outgoing `FOLLOWS` relationships) and who follow them (incoming `FOLLOWS` relationships). This provides a complete view of a user's social connections.

Cypher Query - View Followers:

```
// UC-7: View Followers
// Returns all users who follow the selected user

MATCH (u:User {userId: $userId})<-[:FOLLOWS]-(f:User)
RETURN
    f.userId AS id,
    f.username AS username,
    f.name AS name,
    f.bio AS bio
ORDER BY f.username
LIMIT 100
```

Cypher Query - View Following:

```
// UC-7: View Following
// Returns all users that the selected user follows

MATCH (u:User {userId: $userId})-[:FOLLOWS]->(t:User)
RETURN
    t.userId AS id,
    t.username AS username,
    t.name AS name,
    t.bio AS bio
ORDER BY t.username
LIMIT 100
```

Implementation Notes:

- Separates followers and following into distinct queries for clarity
- Uses directional relationship patterns (`<-[:FOLLOWS]-` vs `-[:FOLLOWS]->`)
- Includes user details (name, bio) for display
- Limited to 100 results for performance

Screenshots:

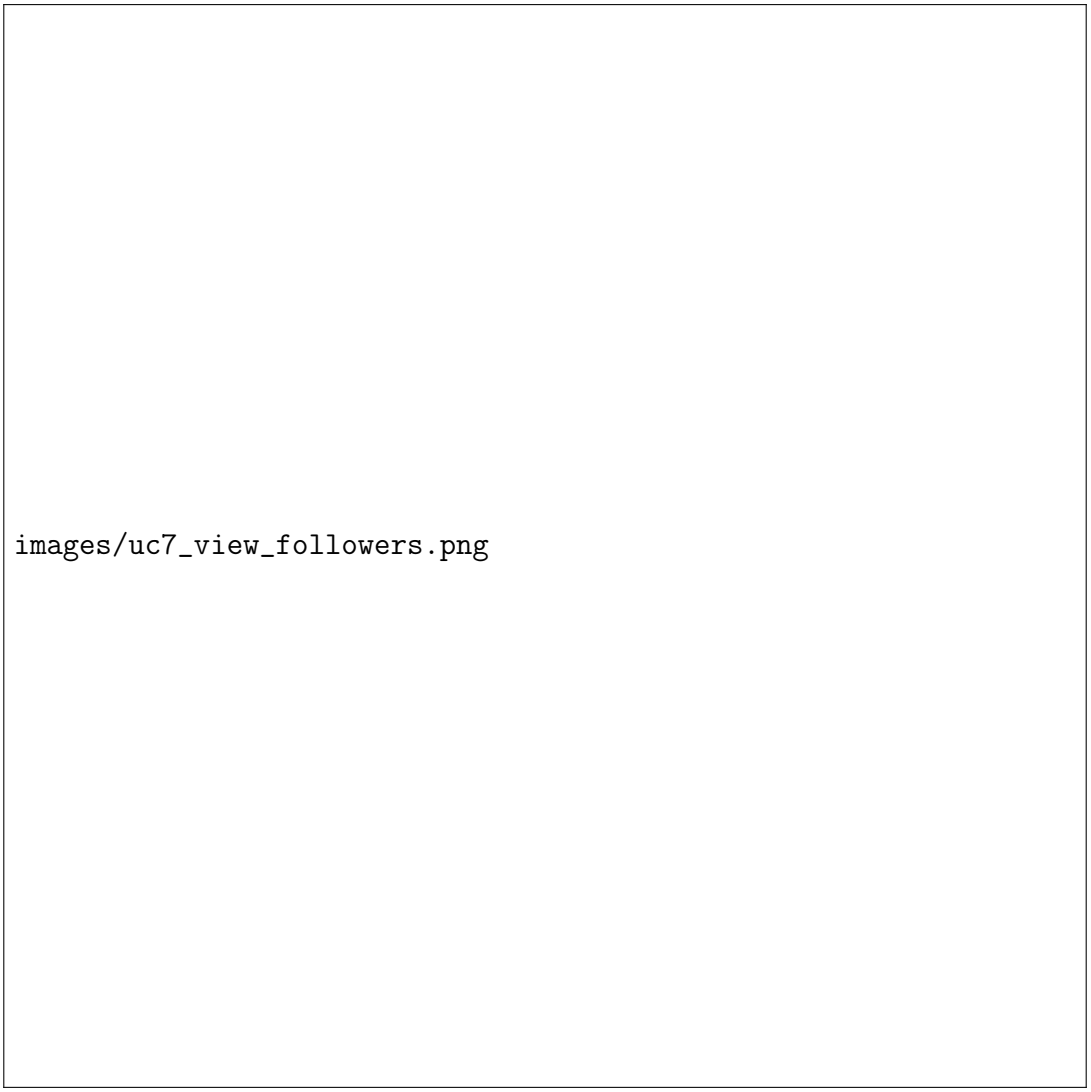

Figure 7: UC-7: View Followers - Users who follow the selected user

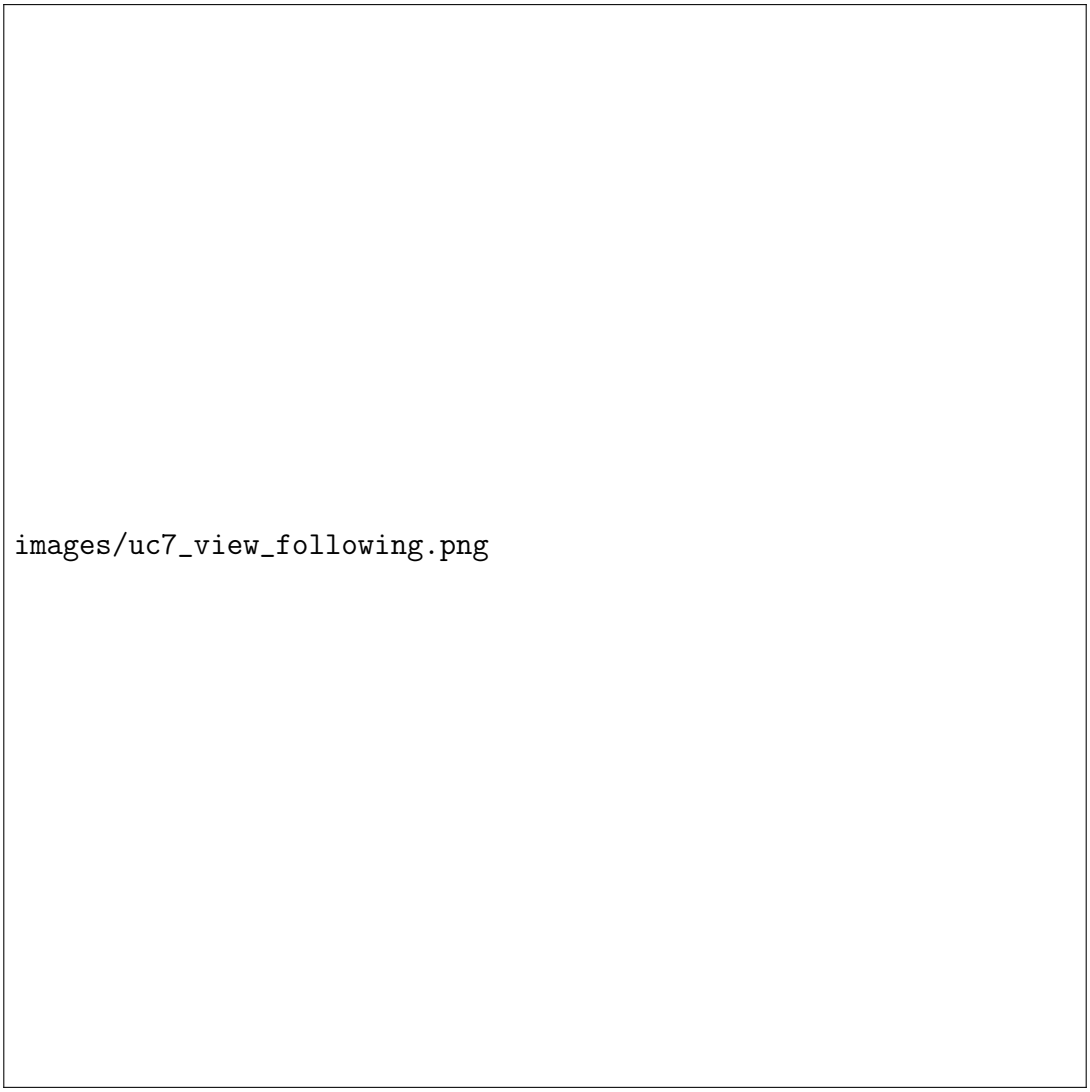

Figure 8: UC-7: View Following - Users the selected user follows

4.8 UC-8: Mutual Connections

Description: A user can see mutual friends—users who are followed by both User A and User B. This feature demonstrates Neo4j's pattern matching capability to find intersection patterns in the graph.

Cypher Query:

```
// UC-8: Mutual Connections
// Finds users that BOTH User A and User B follow
// Uses graph pattern matching to find intersection

MATCH (a:User {userId: $userAId})
      -[:FOLLOWS]->
      (mutual:User)
      <-[:FOLLOWS]-
      (b:User {userId: $userBId})
WHERE a <> b
RETURN DISTINCT
    mutual.userId AS id,
    mutual.username AS username,
```

```
    mutual.name AS name,
    mutual.bio AS bio
ORDER BY mutual.username
LIMIT 50
```

Implementation Notes:

- Uses a single pattern to match both relationships simultaneously
- The pattern `(a)-[:FOLLOWS]->(mutual)<-[:FOLLOWS]-(b)` finds users followed by both A and B
- `WHERE a <> b` ensures we're comparing two different users
- `DISTINCT` prevents duplicate results
- Efficient graph traversal compared to SQL JOIN operations

Screenshot:

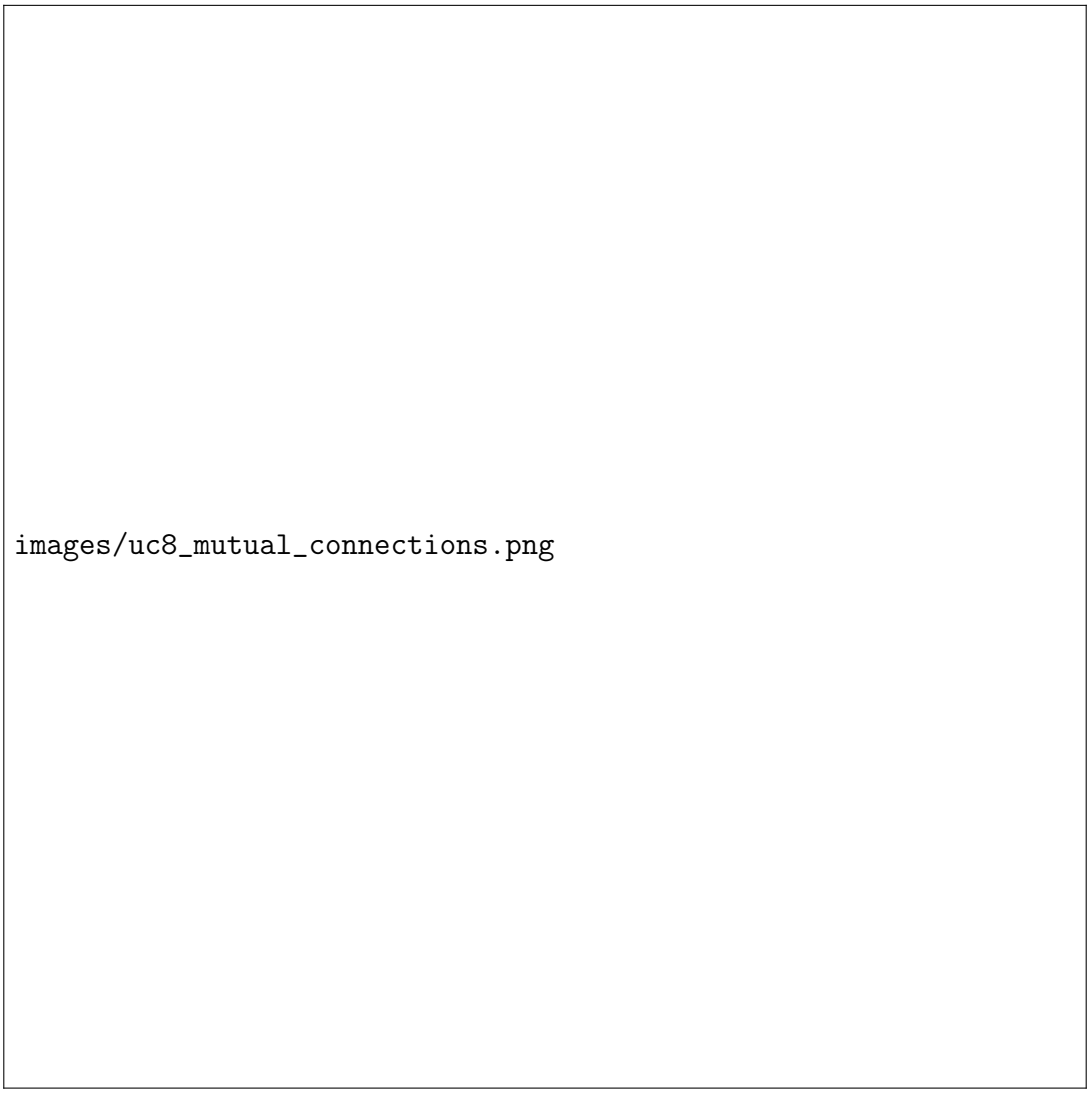

Figure 9: UC-8: Mutual Connections - Users followed by both User A and User B

4.9 UC-9: Friend Recommendations

Description: The system suggests new people to follow based on common connections using graph traversal queries. This implements a "friends-of-friends" algorithm that recommends users followed by people you follow, but whom you don't follow yet.

Cypher Query:

```
// UC-9: Friend Recommendations
// Uses 2-hop graph traversal to find friends-of-friends
// Excludes users already followed and the user themselves
// Ranks by number of mutual connections

MATCH (u:User {userId: $userId})
      -[:FOLLOWS]->(friend)
      -[:FOLLOWS]->(recommended)
WHERE NOT (u)-[:FOLLOWS]->(recommended)
  AND u <> recommended
WITH recommended, count(DISTINCT friend) AS mutualCount
RETURN
    recommended.userId AS id,
    recommended.username AS username,
    recommended.name AS name,
    recommended.bio AS bio,
    mutualCount
ORDER BY mutualCount DESC, recommended.username
LIMIT $limit
```

Implementation Notes:

- **2-hop traversal:** The pattern finds users two steps away in the graph
- **Exclusion filter:** Ensures we don't recommend users already followed
- **Self-exclusion:** Prevents recommending the user to themselves
- **Ranking:** Results are ordered by `mutualCount`, giving priority to stronger recommendations
- **Aggregation:** Uses `count(DISTINCT friend)` to count mutual connections
- This query demonstrates the power of graph databases for social network analytics

Screenshot:

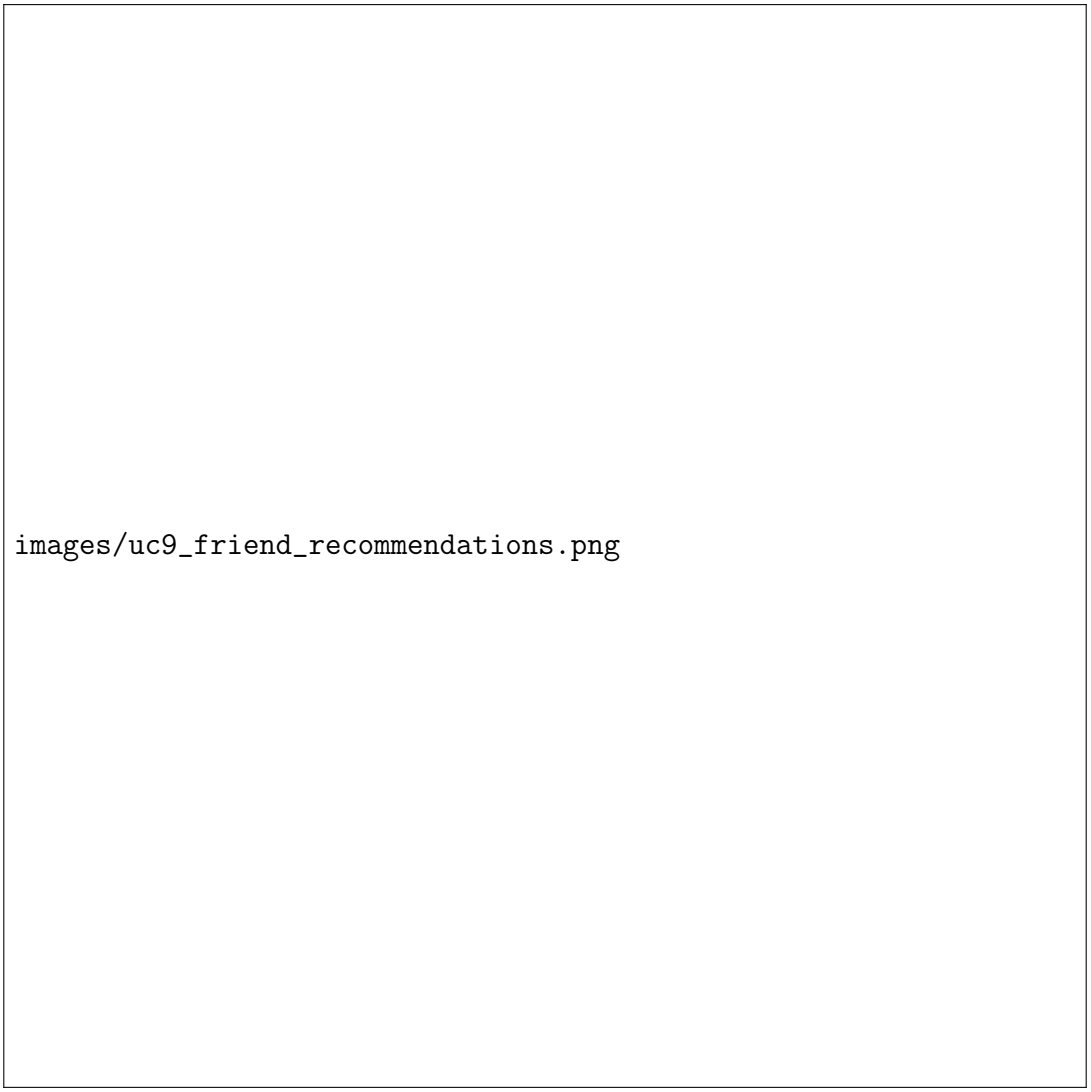

Figure 10: UC-9: Friend Recommendations - Suggested users based on mutual connections

4.10 UC-10: Search Users

[Aditya's implementation]

Screenshot:

images/uc10_search_users.png

Figure 11: UC-10: Search Users

4.11 UC-11: Explore Popular Users

[Aditya's implementation]

Screenshot:

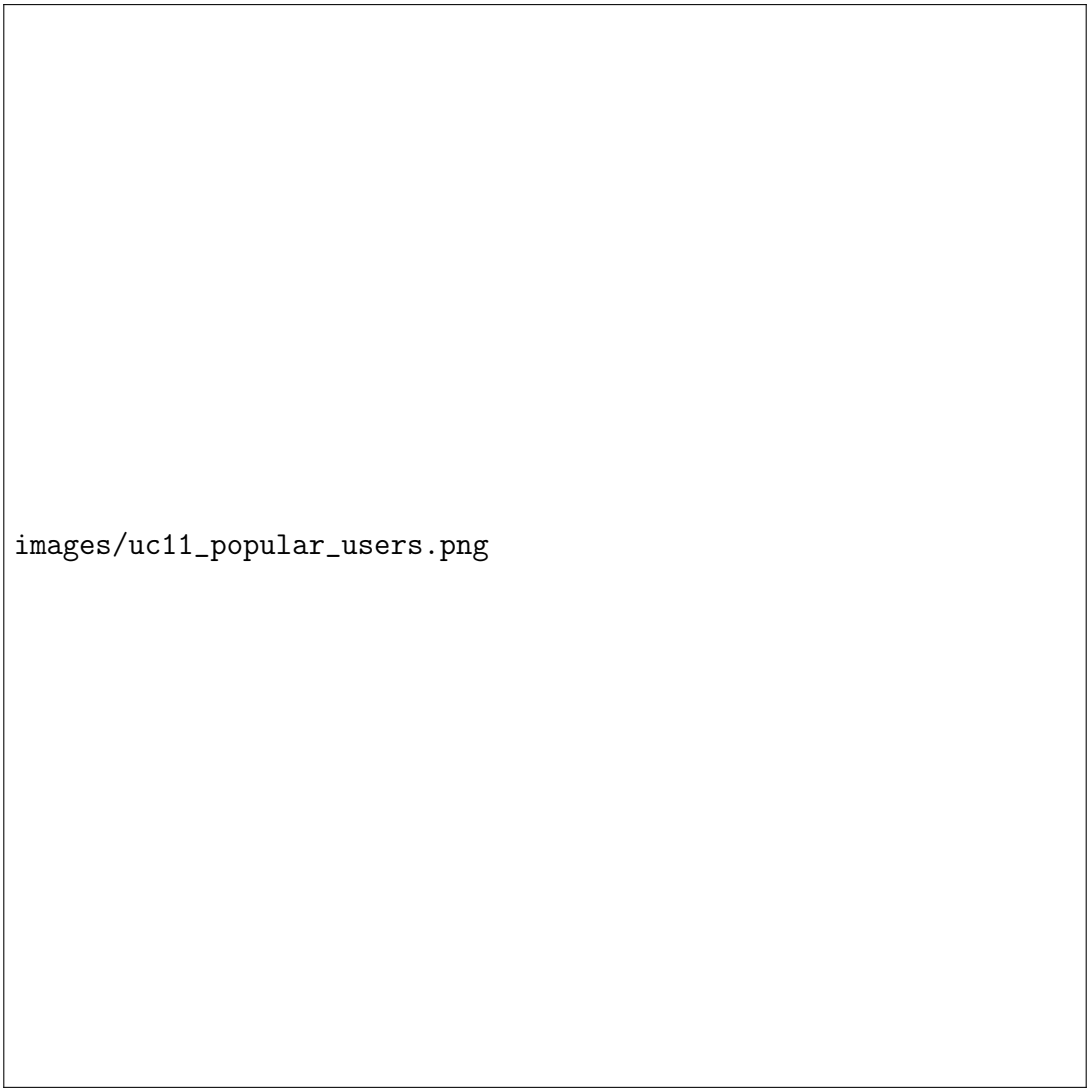

Figure 12: UC-11: Explore Popular Users