**Subject: Big Data Analytics Lab (CSL702)**

**Department: Computer Engg.**

**Year: B.E.**

**Name:** Abu Sufiyan Khan

**Roll No:** 221259

**Date:** 28/10/2025

**Problem Statement :-** Create a social network graph in Neo4j where users are connected by relationships like "FRIEND" and "FOLLOW." Insert nodes and relationships for 5 users. Write a query to find all "FRIEND" connections of a specific user.

**DataSet:** N/A

**Code/Steps:**

**Step 1 :-** Create 5 user nodes

CREATE

(u1:User {name: 'Alice', age: 25, city: 'Mumbai'}),

(u2:User {name: 'Bob', age: 27, city: 'Pune'}),

(u3:User {name: 'Charlie', age: 22, city: 'Delhi'}),

(u4:User {name: 'David', age: 30, city: 'Bengaluru'}),

(u5:User {name: 'Eve', age: 24, city: 'Kolkata'});

**To Confirm,**

MATCH (u:User)

RETURN u.name AS name, u.age AS age, u.city AS city;

**Step 2 :-** Create relationships (FRIEND and FOLLOW)

We’ll create some FRIEND (mutual) and FOLLOW (directed) relationships. Two ways: create directional only or mutual by creating two directed relationships.

Example: create both mutual FRIENDs for some pairs:

MATCH

(a:User {name:'Alice'}),

(b:User {name:'Bob'}),

(c:User {name:'Charlie'}),

(d:User {name:'David'}),

(e:User {name:'Eve'})

CREATE

// -- mutual friendship between Alice and Bob (two directed edges)

(a)-[:FRIEND {since: 2023}]->(b),

(b)-[:FRIEND {since: 2023}]->(a),

// -- Alice friends with Charlie (one direction shown, we can also make mutual)

(a)-[:FRIEND {since: 2024}]->(c),

(c)-[:FRIEND {since: 2024}]->(a),

// -- Bob friends with David (mutual)

(b)-[:FRIEND {since: 2022}]->(d),

(d)-[:FRIEND {since: 2022}]->(b),

// -- Following relationships (directed)

(c)-[:FOLLOW {since: 2024}]->(e),

(e)-[:FOLLOW {since: 2024}]->(a);

**Step 3 :-** Visualize the graph

MATCH (n) RETURN n;

**Step 4 :-** Queries — find all FRIEND connections of a specific user

a) Friends that Alice has an outgoing FRIEND relationship to:

MATCH (a:User {name:'Alice'})-[:FRIEND]->(friend)

RETURN friend.name AS FriendName, friend.age AS Age, friend.city AS City;

This returns nodes that Alice explicitly *has a FRIEND relationship to*.

b) Friends connected in either direction (mutual or undirected view)

If you modeled friendship as mutual (two directed edges), you can ignore direction:

MATCH (a:User {name:'Alice'})-[:FRIEND]-(friend)

RETURN friend.name AS FriendName;

Here -[:FRIEND]- matches relationships in either direction.

c) Incoming friendships (who added Alice as friend)

MATCH (someone)-[:FRIEND]->(a:User {name:'Alice'})

RETURN someone.name AS WhoFriendedAlice;

**Models:**

(User)-[:FRIEND]->(User)

(User)-[:LIKES]->(Interest)

**Cypher Implementation:**

CREATE

(a:User {name:'Alice'}),

(b:User {name:'Bob'}),

(i1:Interest {name:'Music'}),

(i2:Interest {name:'Sports'}),

(a)-[:LIKES]->(i1),

(b)-[:LIKES]->(i1),

(b)-[:LIKES]->(i2);

**Friend Recommendation Query:**

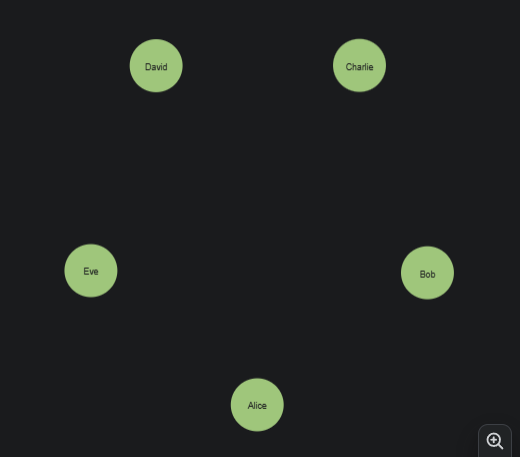
MATCH (u:User {name:'Alice'})-[:LIKES]->(interest)<-[:LIKES]-(other:User)

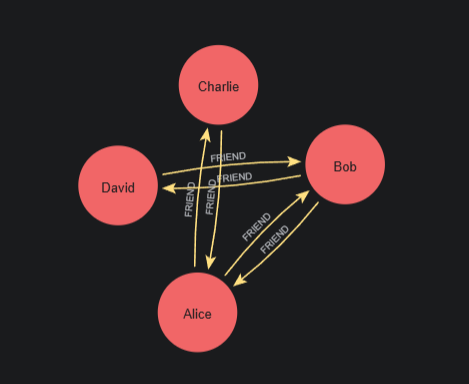
WHERE u <> other

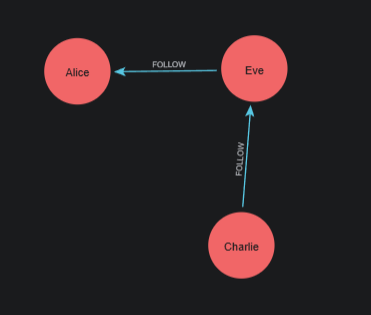
RETURN other.name AS RecommendedFriend, collect(interest.name) AS SharedInterests

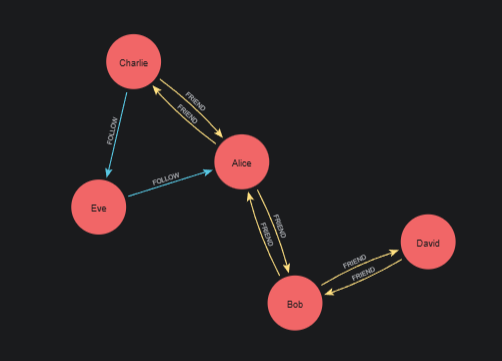
ORDER BY size(SharedInterests) DESC;

**Output :-**

****

****

****

****

**POSTLAB**

Q1) Apply: What is the difference between nodes and relationships in Neo4j, and how do they store data?

* In **Neo4j**, both **nodes** and **relationships** are the fundamental building blocks of a graph database.

| **Aspect** | **Nodes** | **Relationships** |
| --- | --- | --- |
| **Definition** | Represent **entities** or **objects** in the graph (e.g., a User, Product, or City). | Represent **connections** or **associations** between nodes (e.g., FRIEND, FOLLOWS, LIVES\_IN). |
| **Label** | Nodes can have one or more **labels** (e.g., :User, :Movie). | Relationships have **types** (e.g., :FRIEND, :ACTED\_IN). |
| **Direction** | Nodes are **direction-independent**. | Relationships are **directed** — have a start node and an end node. |
| **Data Storage** | Nodes store **properties** in key–value pairs (e.g., {name:'Alice', age:25}). | Relationships can also store **properties** (e.g., {since:2023}). |
| **Identification** | Each node has a unique internal ID. | Each relationship also has its own unique ID and references the IDs of its start and end nodes. |

**Example:**

CREATE (a:User {name:'Alice'})-[:FRIEND {since:2022}]->(b:User {name:'Bob'});

Here:

* a and b are **nodes** storing data (name).
* FRIEND is a **relationship** storing data (since:2022).

<https://chatgpt.com/c/68fefbaf-d3fc-8322-b407-eac3d15ecc09> link

Q2) Analyze: How does Neo4j handle traversal queries efficiently in large graphs?

* Neo4j is optimized for **graph traversal** — the process of exploring nodes and relationships.

**Mechanism:**

1. **Native Graph Storage:**

* Neo4j stores data as **linked nodes and relationships** directly in memory/disk.
* Each relationship stores **pointers** (references) to its start and end nodes, enabling direct access without costly joins.

1. **Index-Free Adjacency:**

* Every node directly knows its adjacent nodes.
* When traversing, Neo4j simply follows these links in constant time — O(1) per hop.
* No need for index lookups or joins like in relational databases.

1. **Traversal Framework:**

* Neo4j uses depth-first or breadth-first traversal strategies internally.
* Optimized algorithms and caching make traversals fast even in graphs with **millions of nodes**.

1. **Query Optimization:**

* Cypher queries are converted into efficient execution plans.
* Neo4j uses indexes only to find starting nodes — traversal after that is pointer-based.

**Example:**

MATCH (a:User {name:'Alice'})-[:FRIEND\*1..3]->(friends)

RETURN friends;

This query can traverse up to 3 levels of friendships efficiently using index-free adjacency.

Q3) Evaluate: Assess the advantages and limitations of using Neo4j for social network analysis compared to relational databases.

**Advantages of Neo4j:**

1. **Natural Representation:**

* Graph model directly maps to real-world social networks (users, friends, followers).

1. **Fast Relationship Queries:**

* Traversal queries like “friends of friends” are **much faster** than SQL joins.

1. **Scalability for Connectivity:**

* Handles **highly connected** data efficiently.

1. **Flexible Schema:**

* Schema-less — easy to evolve model (add properties, relationships).

1. **Cypher Query Language:**

* Intuitive and expressive for pattern matching.

**Limitations of Neo4j:**

1. **Not Ideal for Tabular Aggregations:**

* SQL databases outperform Neo4j in heavy numeric or tabular aggregation workloads.

1. **Memory Intensive:**

* Graph traversal requires more RAM for large connected datasets.

1. **Limited Horizontal Scaling:**

* Neo4j clustering is improving, but massive horizontal scalability is still more complex than SQL partitioning.

1. **Learning Curve:**

* Requires understanding of graph modeling and Cypher syntax.

**Conclusion:**

Neo4j excels at **relationship-centric analysis** (social, recommendation, fraud networks), while relational databases remain better for **transactional or analytical tabular data**.

Q4) Create: Propose an enhancement to the social network model to include user interests and recommend friends based on shared interests.

**Models:**

(User)-[:FRIEND]->(User)

(User)-[:LIKES]->(Interest)

**Cypher Implementation:**

CREATE

(a:User {name:'Alice'}),

(b:User {name:'Bob'}),

(i1:Interest {name:'Music'}),

(i2:Interest {name:'Sports'}),

(a)-[:LIKES]->(i1),

(b)-[:LIKES]->(i1),

(b)-[:LIKES]->(i2);

**Friend Recommendation Query:**

MATCH (u:User {name:'Alice'})-[:LIKES]->(interest)<-[:LIKES]-(other:User)

WHERE u <> other

RETURN other.name AS RecommendedFriend, collect(interest.name) AS SharedInterests

ORDER BY size(SharedInterests) DESC;

**Result:**

Recommends users who share common interests — just like “People You May Know” in Facebook.

<https://chatgpt.com/share/68ff6ad8-13ac-8003-8017-f940da4b8789> LINK

Q5) Analyze: Compare Neo4j with other graph databases such as Amazon Neptune and ArangoDB in terms of scalability and performance.

| **Feature** | **Neo4j** | **Amazon Neptune** | **ArangoDB** |
| --- | --- | --- | --- |
| **Model Type** | Property Graph | Property Graph + RDF | Multi-model (Graph + Document + Key/Value) |
| **Query Language** | Cypher | Gremlin + SPARQL | AQL (Arango Query Language) |
| **Performance** | Excellent for single-machine, highly connected data | Highly scalable (managed AWS service) | Good for mixed workloads |
| **Scalability** | Vertical (cluster for read replicas) | Horizontal (auto-scaled by AWS) | Scales well with sharding |
| **Deployment** | Self-managed or Aura Cloud | Fully managed on AWS | Self-hosted or Managed Cloud |
| **Traversal Efficiency** | Index-free adjacency (very fast) | Optimized with distributed memory graph | Slightly slower for deep traversal due to multi-model design |
| **Use Case Fit** | Deep graph analytics, recommendation | Large-scale enterprise graphs | Hybrid data use (documents + graph) |

**Conclusion:**

* **Neo4j:** Best for deep, real-time relationship analysis.
* **Amazon Neptune:** Best for large-scale, cloud-native graph applications.
* **ArangoDB:** Best for hybrid use cases needing both document and graph features.