GRAPH DATABASES AND Neo4j

What are Graph Databases?

A graph database is a database that identifies relationships between data entities, and treats these relationships with equal importance as it does the data itself. The primary intent behind graph databases is to provide a representation that is more efficient, and also highly intuitive in representing relationships. Graph databases hold data without constricting it to the definitions laid down by a pre-defined model. Instead, data is stored in a representational manner depicting the connection/relations of each individual entity with others.

Why use Graph Databases?

Every piece of information can be objectively interpreted to be intertwined and connected to other information in multiple rich, connected domains. Graph databases embrace the complexities of these relationships and provide a way for efficiently storing, processing, and querying these connections, independent of database size.

Highly connected data can be easily represented, and complex queries can be quickly executed. Since the database representation relies on the data, as well as the relationships between the data, a much larger scope of flexibility is also present. It is a common phenomenon for databases' structures to be altered, or for new attributes to be accommodated in some manner. Graph databases make this highly intuitive, and much simpler.

Graph databases allow for simpler, yet more expressive data models than those observed with conventional relational databases/NoSQL databases.

Further, the fundamental ideology behind graph databases align perfectly with today's agile, test-driven and strict deadline-restricted development practices, allowing the application to evolve with evolving business requirements.,

How can Graph Databases be used? How do they work?

Graph databases typically are built using what is known as the Property Graph Model.

A graph database is composed of two atomic components: a node and a relationship.

Each node represents an entity, and each relationship represents the association between two nodes by providing named, directed, semantically-relevant connections between the nodes.

In graph databases, relationships are of the highest priority, and connected data is of significant importance.

This *connections-first* approach to the data presented means that relationships and connections are persisted (and not temporarily computed) through every step in the data lifecycle - from idea, to a logical model design, to a physical model implementation, to operation using a query language and to persistence within a scalable, reliable database system.

What is Neo4j?

Neo4j is an open-source, NoSQL, native graph database that provides an ACID-compliant transactional backend for applications.

It is often referred to as a "native graph database" because of its ability in efficiently implementing the *Property Graph Model* down to the storage level, and it also provides full database characteristics; including but not limited to – ACID transaction compliance, cluster support and runtime failover.

Neo4j uses a declarative query language optimized for graphs named *Cypher*, provides constant support with drivers for popular programming languages too, in addition to providing all the benefits of using a graph database.

USE CASES OF GRAPHICAL DATABASES

Graphical databases are being applied and used with growing popularity.
A few of the most popular use cases for graphical databases are as follows.

1. Real-Time Recommendation Engines

Making effective real-time recommendations requires a database that understands and represents relationships between entities and the quality of these relationships. Recommendation engines backed with a graphical database can be based on either of two paradigms - identifying resources of interest to individuals; or identifying individuals likely to be interested in a given resource. With either approach, the graph databases make the necessary correlations and connections to serve up the most relevant results for the individual or resource in question.

2. Network and IT Security

Networks are inherently represented and perceived to be graphs. Therefore, graph databases prove to be an excellent fit for modelling, storing, and querying Network and IT Operational data, in all scenarios. Graph databases have been successfully employed in the domains of telecommunications, impact analysis, and cloud platform, data center and IT asset management.

3. Master Data Management

Master Data Management is the practice of identifying, cleaning, storing, and governing data being stored (generally on a relatively larger scale). MDM systems generally observe highly connected data, which result in severe business overhead if the database is poorly modelled. Graph databases allow for easy data modelling, cost lesser resources, provide more refined control over distributed data (preventing the requirement for migration), and allow establishment of relationships that connect the various data entities observed throughout the enterprise.

4. Fraud Detection

Traditional methods for fraud detection fail because they perform discrete analyses of data (which are susceptible to false positives and negatives), which is exploited by fraudsters who develop sophisticated techniques that exploit this weakness of discrete analysis. Graph Databases allow for advanced contextual link analysis, which helps in detecting complex scams and uncovering fraud rings in real time.

5. Identity and Access Management

Identity and Access Management solutions store information about various parties, and resources, along with the rules that govern the access levels that the parties have to respective resources (for access and manipulation of resources). Graph databases can store these complex, and densely connected access control structures; with its data model that supports both hierarchical and non-hierarchical structures, and with its property model that allows for capture of rich metadata concerning every element in the system.

Neo4j CASE STUDY – Lyft and Data Discovery

Lyft is an app-based gig economy company that provides more than 50 million rides a month to customers through its network of contract drivers. Riders use the Lyft app, which generates data with every ride Lyft provides.

Lyft's growth exacerbated the challenge of data discovery. Lyft already had about 10 petabytes in thousands of tables across a variety of different data stores. Growth meant even more data generated by the mobile app and other services. All this only slowed down the process of data discovery, and exposed the inefficiency in the data discovery process. It was discovered that data discovery consumed a third of the time of the data scientists employed.

Lyft engineers ultimately decided to build a tool that would simplify the data discovery process. Their first target audience would be the most frequent users of data – data analysts and scientists.

The tool, named *Amundsen*, would offer 3 complementary ways to do data discovery – search-based, lineage-based, and network-based.

An effective search was a top priority, ranking results by popularity and relevance.
Lineage-based discovery traces connections among datasets.
Network-based data discovery connects data with people, particularly valuable for new team members.

Amundsen uses a microservice architecture. The *Databuilder* service ingests data into the search service, which is backed by *Elasticsearch*, and the metadata service, which is run by the *Neo4j* graph database. The connections that are first made in Neo4j are then used by Elasticsearch to power search by providing relevance based on search terms, the user's position in the company and the popularity of the tables.
This proved to be a good fit since Lyft's data ecosystem could be shaped and naturally expressed as a graph, and Neo4j provides a foundation for new projects like compliance and data quality.

Amundsen's adoption rate is significant. Lyft's choice to make the tool open source helped make its impact even broader, allowed an active community to grow around it, and also helped the tool immensely gain more popularity.

Neo4j – BASIC COMMANDS AND SYNTAX

Read Clauses

1. **MATCH**
   Search the data with a specified pattern.

   - Get all the nodes in the neo4j database

   ```
   MATCH (n) RETURN n
   ```

   - Get all the nodes under a specific label

   ```
   MATCH (node:label)
   RETURN node
   ```

   - Match by relationship

   ```
   MATCH (node:label)<-[: Relationship]-(n)
   RETURN n
   ```

   - Delete all nodes

   ```
   MATCH (n) detach delete n
   ```

2. **OPTIONAL MATCH**
   Same as match, with the only difference being it can use NULLs in case of missing parts of the pattern.

   - Optional Match with relationship

   ```
   MATCH (node:label {properties… })
   OPTIONAL MATCH (node)-->(x)
   RETURN x
   ```

3. **WHERE**
   Add contents to the CQL queries. This helps in filtering the results of a MATCH query.

   - Where Clause

   ```
   MATCH (label)
   WHERE label.property = "property"
   RETURN label
   ```

   Note: The **where** clause can be used in conjunction with the operators like the **AND** operator as well, for more refined and filtered results. The **where** clause can also be used to filter the nodes using the relationships.

4. **COUNT**
   Count the number of rows.
   - Count Function

```
MATCH (n { properties… })-->(x)
RETURN n, count(*)
```

5. **START**
   Starting points through the legacy indexes.

6. **LOAD CSV**
   Import data from CSV file

Write Clauses

1. **CREATE**

   Create nodes, relationships, and properties.

   Creating Nodes
   - Create a single node

```
CREATE (node_name)
```

   - Create multiple nodes

```
CREATE (node1), (node2)
```

   - Create a node with a label

```
CREATE (node:label)
```

   - Create a node with multiple labels

```
CREATE (node:label1:label2:label3: … :labeln)
```

   - Create a node with properties

```
CREATE (node:label { key1: value1, key2: value2, … })
```

   Creating Relationships
   - Create a relationship

```
CREATE (node1)-[:RelationshipType]->(node2)
```

   - Create a relationship between existing nodes

```
MATCH (a:LabeofNode1), (b:LabeofNode2)
   WHERE a.name = "nameofnode1" AND b.name = " nameofnode2"
```

```
CREATE (a)-[: Relation]->(b)
RETURN a,b
```

- Create a relationship with labels and properties

```
CREATE (node1)-[label:Rel_Type {key1:value1, key2:value2,
…}]-> (node2)
```

- Create a complete path

```
CREATE p = (Node1 {properties})-[:Relationship_Type]->
   (Node2 {properties})[:Relationship_Type]->(Node3 {
properties})
RETURN p
```

2. **MERGE**
Verifies whether the specified pattern exists in the graph. If not, it creates the pattern.

- Merge Clause

```
MERGE (node: label {properties…})
```

- Merge a node with a label

```
MERGE (node:label) RETURN node
```

- Merge a node with properties

```
MERGE (node:label {key1:value, key2:value, key3:value…})
```

- Merge a relationship

```
MATCH (a:label1), (b:label2)
   WHERE a.property1 = "value1" AND b.property2 = "value2"
   MERGE (a)-[r:relationship]->(b)
RETURN a, b
```

3. **FOREACH**
Update the data within a list.

- Foreach Clause

```
MATCH p = (start node)-[*]->(end node)
WHERE start.node = "node_name" AND end.node = "node_name"
FOREACH (n IN nodes(p)| SET n.marked = TRUE)
```

4. **CREATE UNIQUE**
Using the clauses CREATE and MATCH, you can get a unique pattern by matching the existing pattern and creating the missing one

5. **SET**
Update labels on nodes, properties on nodes and relationships.

- Set Clause

```
MATCH (node:label{properties…})
SET node.property = value
RETURN node
```

- Removing a property

```
MATCH (node:label {properties})
SET node.property = NULL
RETURN node
```

- Setting multiple properties

```
MATCH (node:label {properties})
SET node.property1 = value, node.property2 = value
RETURN node
```

- Setting a label on a node

```
MATCH (n {properties…})
SET n :label
RETURN n
```

- Setting multiple labels on a node

```
MATCH (n {properties…})
SET n :label1:label2
RETURN n
```

6. **DELETE**
Delete nodes, relationships, paths, etc. from the graph.

- Delete all nodes and relationships

```
MATCH (n) DETACH DELETE n
```

- Delete a particular node

```
MATCH (node:label {properties …})
DETACH DELETE node
```

7. **REMOVE**
   Remove properties and elements from nodes and relationships

   - Remove a property

   ```
   MATCH (node:label{properties …})
   REMOVE node.property
   RETURN node
   ```

   - Remove a label from a node

   ```
   MATCH (node:label {properties …})
   REMOVE node:label
   RETURN node
   ```

   - Remove multiple labels from a node

   ```
   MATCH (node:label1:label2 {properties …})
   REMOVE node:label1:label2
   RETURN node
   ```

General Clauses

1. **RETURN**
   Return nodes, relationships, and properties.

   - Return clause

   ```
   CREATE (node:label {properties})
   RETURN node
   ```

   Similarly, return clause can be used to return multiple nodes as well.

   - Returning properties

   ```
   MATCH (node:label {properties …})
   RETURN node.property
   ```

   - Returning a variable with column alias

   ```
   MATCH (node:label {properties …})
   RETURN node.property as Column Alias
   ```

   - Returning all the elements

   ```
   MATCH (node:label {properties …})
   RETURN *
   ```

2. **ORDER BY**
Arrange the output of a query in order along with the clauses **RETURN** or **WITH.**

- Order by clause

```
MATCH (n)
RETURN n.property1, n.property2 …
ORDER BY n.property
```

- Ordering nodes by multiple properties

```
MATCH (n)
RETURN n
ORDER BY n.property1, n.property2
```

- Ordering nodes in descending order

```
MATCH (n)
RETURN n
ORDER BY n.name DESC
```

3. **LIMIT**
Limit the rows in the result to a specific value.

- Limit clause

```
MATCH (n)
RETURN n.property1, n.property2 …
LIMIT limit_value
```

Note: The limit clause can be used with an expression as well.

4. **SKIP**
Define from which row to start including the rows in the output.

- Skip clause

```
MATCH (n)
RETURN n.property1, n.property2 …
LIMIT skip_value
```

Note: The skip clause can be used with an expression as well.

5. **WITH**
   Chain the query parts together.

   - With clause

   ```
   MATCH (n)
   WITH n
   ORDER BY n.property
   RETURN collect(n.property)
   ```

6. **UNWIND**
   Expand a list into a sequence of rows.

   - Unwind a list

   ```
   UNWIND [a, b, c, d] AS x
   RETURN x
   ```

7. **UNION**
   Combine the result of multiple queries.

8. **CALL**
   Invoke a procedure deployed in the database.

CREATING A SAMPLE GRAPH DATABASE USING Neo4j

Consider a very simple graph database representing a social media platform (for image sharing).

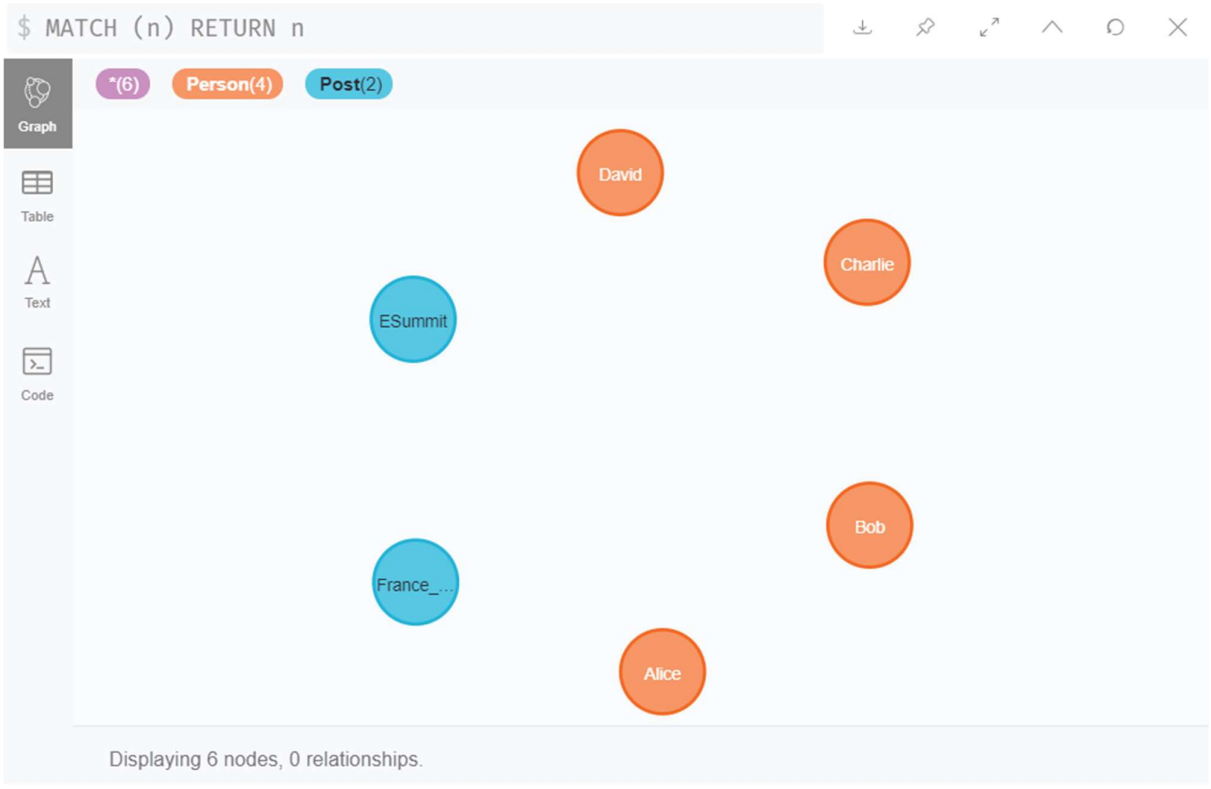
Here, the nodes will be of types "***Person***" and "***Post***". The relationships would be **FRIEND_OF** (between ***Person*** and ***Person***), and **CREATES, COMMENTS**, and **LIKES** (between ***"Person"*** and ***"Post"***).

Consider 4 ***Persons*** - Alice, Bob, Charlie, and David; and 2 ***Posts*** –ESummit and France_Trip.

Each ***Person*** has *name*, and *dob* as two attributes.
Each ***Post*** has *title, location, upload_date, image_location,* and *caption* as its attributes.

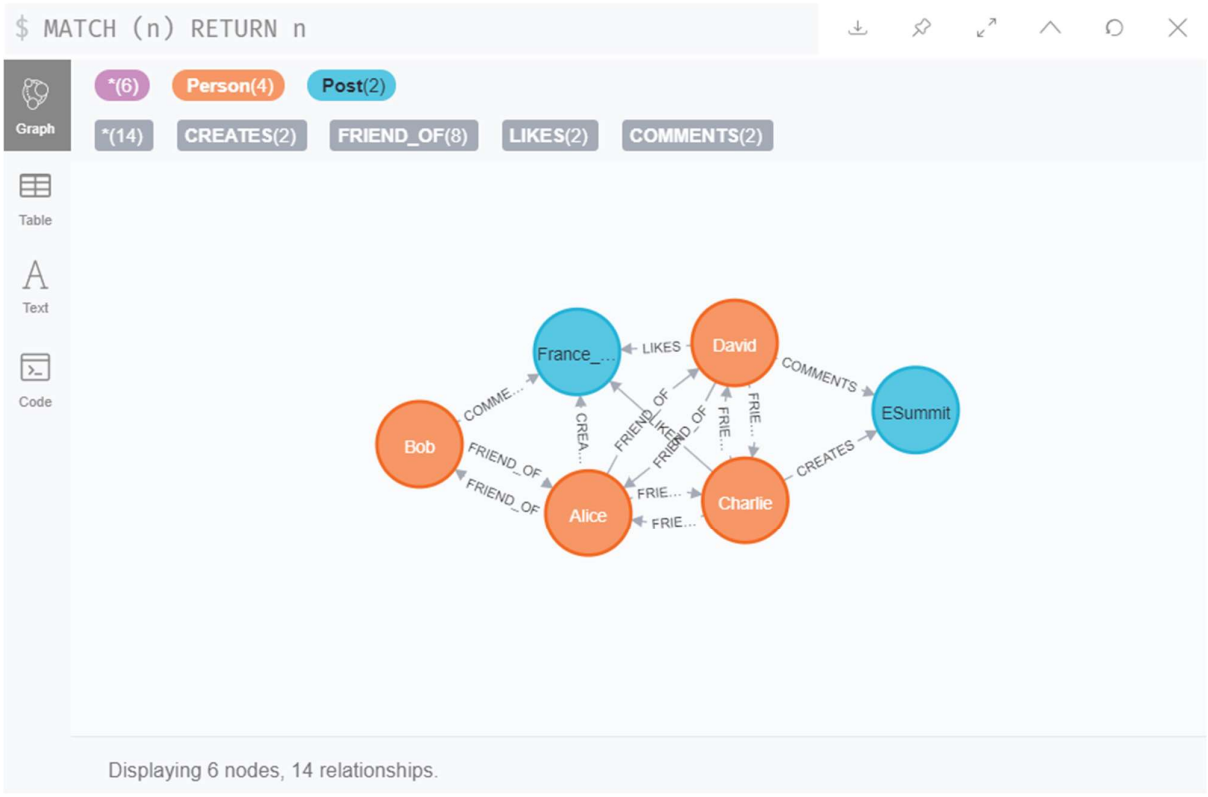
Creating Nodes:

```
CREATE (Alice:Person {name:'Alice', dob:1990})
CREATE (Bob:Person {name:'Bob', dob:1980})
CREATE (Charlie:Person {name:'Charlie', dob:1988})
CREATE (David:Person {name:'David', dob:1970})
CREATE (ESummit:Post {caption:'Pictures from the ESummit!',
title:'ESummit', location:'Home', upload_date:'02/13/2020',
image_location:'img007'})
CREATE (France_Trip:Post {caption:'Pictures from the France trip',
title:'France_Trip', location:'France', upload_date:'07/23/2020',
image_location:'https://www.tripimage.org/'})
```

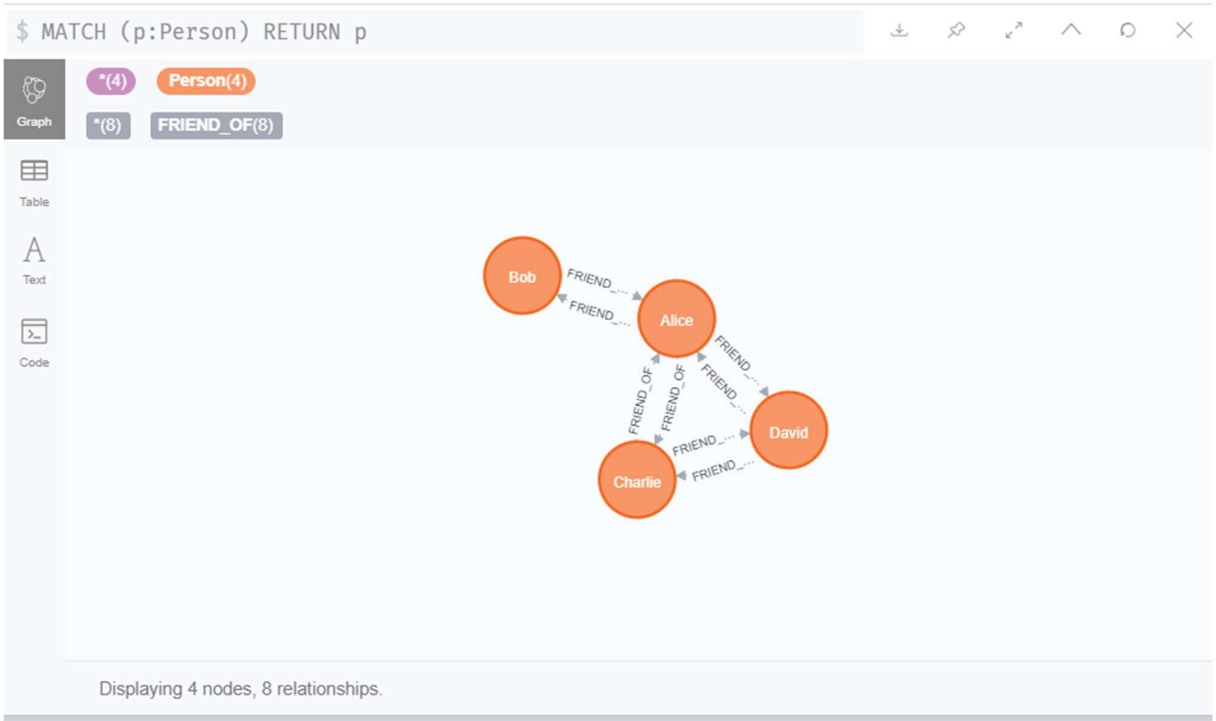


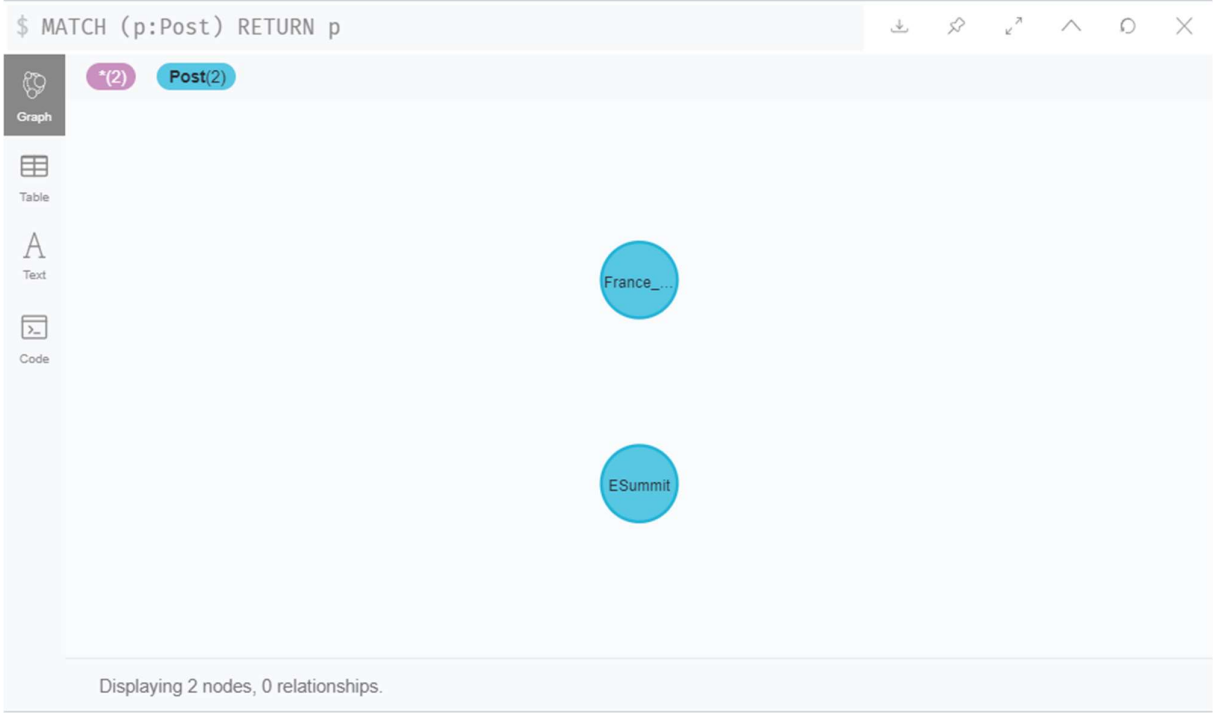
Creating Relationships:

```
CREATE
(Alice)-[:FRIEND_OF]->(Charlie),
(Alice)-[:FRIEND_OF]->(David),
(Alice)-[:FRIEND_OF]->(Bob),
(Bob)-[:FRIEND_OF]->(Alice),
(Charlie)-[:FRIEND_OF]->(Alice),
(Charlie)-[:FRIEND_OF]->(David),
(David)-[:FRIEND_OF]->(Charlie),
(David)-[:FRIEND_OF]->(Alice),
(Alice)-[:CREATES {created_date:['23/07/2020 14:00']}]->(France_Trip),
(Charlie)-[:CREATES {created_date:['23/07/2020 14:00']}]->(ESummit),
(Charlie)-[:LIKES {created_date:['23/07/2020 15:00']}]->(France_Trip),
(David)-[:LIKES {created_date:['23/07/2020 17:30']}]->(France_Trip),
(David)-[:COMMENTS {created_date:['14/02/2020 10:45'],comment:'Looks like
you had a good time!'}]->(ESummit),
(Bob)-[:COMMENTS {created_date:['23/07/2020 14:45'],comment:'Wow. This
looks amazing.'}]->(France_Trip)
```



GRAPHICAL OUTPUTS PRODUCED UPON RUNNING QUERIES ON THE GRAPH DATABASE CREATED



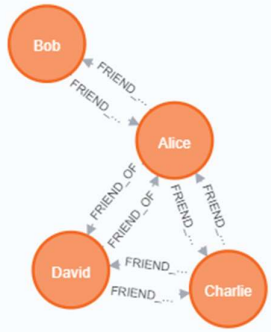


```
$ MATCH p=()-[r: FRIEND_OF]→() RETURN p
```

Graph | Table | Text | Code

*(4) Person(4)

*(8) FRIEND_OF(8)



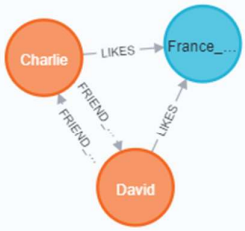
Displaying 4 nodes, 8 relationships.

```
$ MATCH p=()-[r: LIKES]→() RETURN p
```

Graph | Table | Text | Code

*(3) Person(2) Post(1)

*(4) LIKES(2) FRIEND_OF(2)



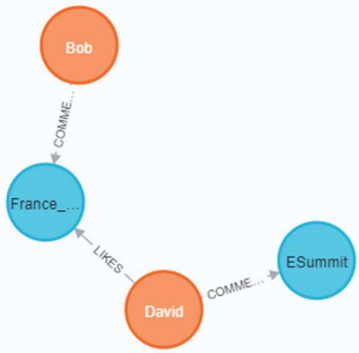
Displaying 3 nodes, 4 relationships.

```
$ MATCH p=()-[r: COMMENTS]→() RETURN p
```

Graph | Table | Text | Code

*(4) Person(2) Post(2)

*(3) COMMENTS(2) LIKES(1)



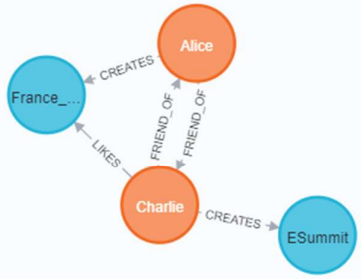
Displaying 4 nodes, 3 relationships.

```
$ MATCH p=()-[r: CREATES]→() RETURN p
```

Graph | Table | Text | Code

*(4) Person(2) Post(2)

*(5) CREATES(2) FRIEND_OF(2) LIKES(1)



Displaying 4 nodes, 5 relationships.