Data Warehousing

Lecture 9 – Graph Queries – Introduction to Cypher

CITS3401
CITS5504

Dr. Sirui Li

Computer Science and Software Engineering

School of Maths, Physics and Computing

Acknowledgement: The lecture slides are adopted from online sources.

Lecture Outline

- Introduction to Cypher
- CRUD Actions:
  - Creating
  - Reading
  - Updating
  - Deleting
- Connect Neo4j with Python

Introduction to Cypher

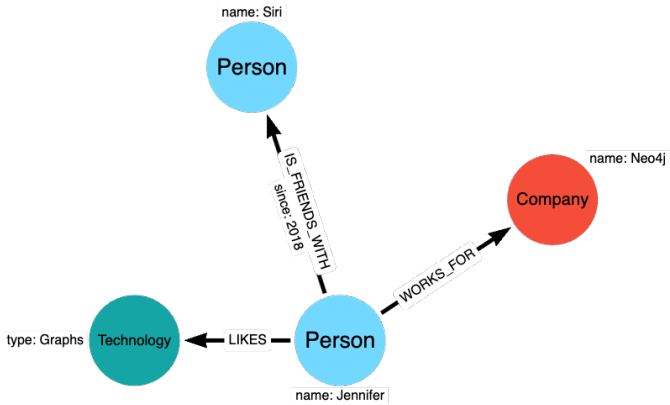
- Cypher is a graph database query language, specific to Neo4j.
- Other query languages exist for other graph databases, e.g.
  - Gremlin (Amazon Neptune)
  - SPARQL (Amazon Neptune)
  - PGQL (Oracle)
  - AQL (ArangoDB)

Introduction to Cypher

- Cypher is designed to be **easily read** and **understood**.
- The language is structured in a way that resembles the way we could describe graphs in plain English.
- Cypher allows us to ask the graph database to **find a specific pattern**, e.g. “find things like this”.

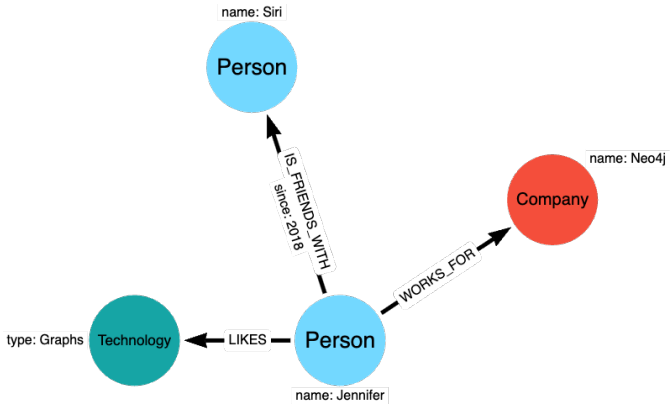
Describing a graph in plain English

How would you describe the graph below in plain English?



Describing a graph in plain English

Jennifer likes graphs. Jennifer is a friend with Siri. Jennifer works for Neo4j.



Representing Nodes in Cypher

- Nodes and relationships in Cypher are represented in ACSII (American Standard Code for Information Interchange). Nodes are surrounded by **round brackets**.
- A node is in the form:

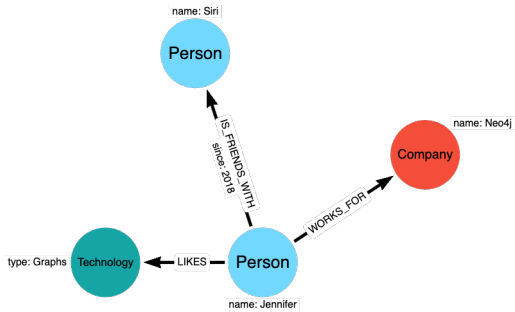
```
(Label {property: value})
```

- For example:

```
(Person {name: "Siri"})
(Person {name: "Siri", gender: "F"})
```

Just like SQL or Python, you need quotes around strings.

More Examples




```
(Person {name: "Siri"})
(Person {name: "Jennifer"})
(Company {name: "Neo4j"})
(Technology {type: "Graphs"})
```

Representing Relationships in Cypher

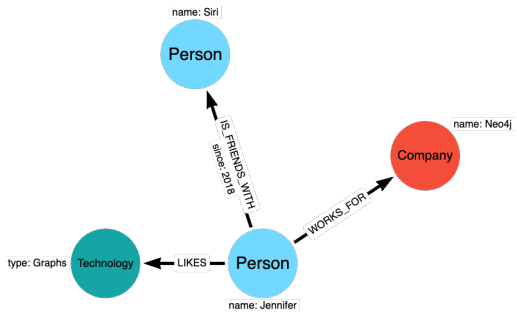
- Relationships are represented similarly but use **square brackets** and a colon before the relationship label.
- A relationship is in the form:

```
[:Label {property: value}]
```

- For example:

```
[:IS_FRIEND_WITH {since: 2015}]
```

More Examples




```
[:IS_FRIEND_WITH {since: 2015}]
[:LIKES]
[:WORKS_FOR]
```

Connecting them together

- We can use this notation to represent a pattern in a Cypher query.
- We can connect nodes and edges using **hyphens**, and specify the **direction** using an **accent**.

Connecting them together

```
(Person {name: "Jennifer"})-
[:WORKS_FOR]->
(Company {name: "Neo4j"})
```

Cypher Clauses

- Like most query languages, Cypher is composed of **clauses**.
- The simplest queries consist of a **MATCH** clause followed by a **RETURN** clause.

Cypher Clauses – Example Query

Here is a simple example query:

```
MATCH (Person {name: "Jennifer"})-[:WORKS_FOR]->(a:Company)
RETURN a
```

What does this mean in plain English?

"Find all companies that Jennifer works for"

Lecture Outline

- Introduction to Cypher
- CRUD Actions:
  - Creating
  - Reading
  - Updating
  - Deleting
- Connect Neo4j with Python

Creating

Generally speaking, there are two ways to create data in Neo4j:

- Creating new data using **CREATE** and/or **MERGE** clauses
- Importing an existing CSV-based dataset using **LOAD CSV**

Creating new data using CREATE - nodes

```
CREATE (n)
```

```
neo4j$ CREATE (n)
```

Table

Code

Created 1 node, completed after 12 ms.

Creating new data using CREATE - nodes

```
CREATE (p:Person {name: "Siri"})
```

```
neo4j$ CREATE (p:Person{name:"Siri"})
```

Table

Code

Added 1 label, created 1 node, set 1 property, completed after 2 ms.

Creating new data using CREATE - nodes

```
CREATE (p1:Person {name: "Siri"}), (p2:Person {name: "test user"})
```

```
neo4j$ CREATE (p1:Person{name:"Siri"}), (p2:Person{name:"test user"})
```

Table

Code

Added 2 labels, created 2 nodes, set 2 properties, completed after 3 ms.

Creating new data using CREATE - relationships

We can also create relationships between **existing** nodes, but to do this we must first use a **MATCH** clause:

```
MATCH (p:Person {name: "Siri"}),
      (b:Book {name: "Harry Potter"})
CREATE (p)-[r:READ]->(b)
RETURN p, r, b
```

The RETURN is optional

Using MERGE to avoid duplication

- The **MERGE** clause checks whether the specific pattern exists. If they do, they are matched. If they don't, they are created.
- It is essentially a combination of **MATCH** and **MERGE.**
- **MERGE** allows us to avoid duplication.

MERGE Example

```
MERGE (p:Person {name: "Siri"})
```

```
neo4j$ MERGE (p:Person{name:"Siri"})
```

Table

(no changes, no records)

Code

Because there is a Person node named "Siri" in the database.

MERGE Example

```
MERGE (p:Person {name: "Siri", gender: "F"})
```

```
neo4j$ MERGE (p:Person{name:"Siri", gender:"F"})
```

Table

Added 1 label, created 1 node, set 2 properties, completed after 1 ms.

Code

MERGE Example

```
MATCH (p:Person {name: "Siri")
RETURN p
```

```
1 MATCH (p:Person{name:"Siri"})
2 RETURN p
```

Graph

Table

Text

Code

p
(:Person {name: "Siri"})
(:Person {gender: "F",name: "Siri"})

MERGE Example

```
MATCH (p:Person {name: "Siri"}),
      (b:Book {name: "Harry Potter"})
MERGE (p)-[r:READ]->(b)
RETURN p, r, b
```

The RETURN is optional

Creating new data via LOAD CSV

- Oftentimes we will have an **existing CSV dataset** that we want to import into Neo4j.
- We need to translate the CSV tables into nodes and relationships.

Read Week 9 lab for more information.

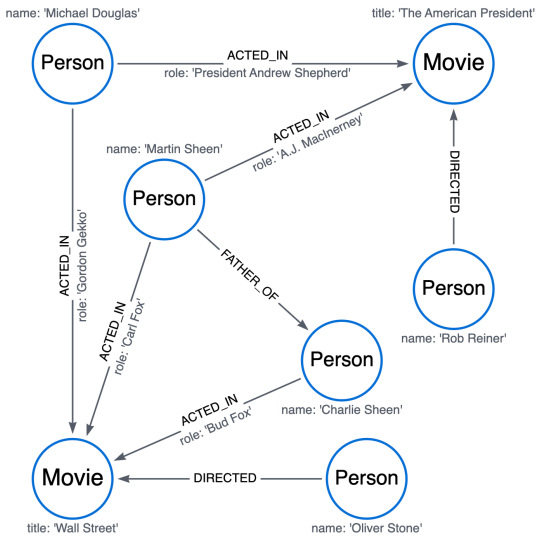
Lecture Outline

- Introduction to Cypher
- CRUD Actions:
  - Creating
  - Reading
  - Updating
  - Deleting
- Connect Neo4j with Python

Reading/Querying in Cypher

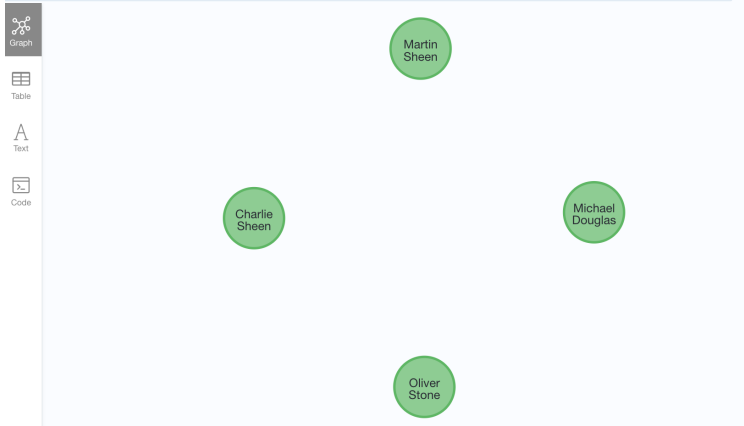
- The **MATCH** clause allows you to specify the patterns Neo4j will search for in the database.
- **MATCH** is often coupled to a **WHERE** part which adds restrictions, or predicates, to the **MATCH** patterns, making them more specific.
- The **RETURN** clause defines the parts of a pattern (nodes, relationships, and/or properties) to be included in the query result.

Reading/Querying in Cypher



MATCH and RETURN

```
MATCH (:Movie{title: 'Wall Street'})←[:ACTED_IN|DIRECTED]-(p:Person)
RETURN p
```



Find all people who have either acted in or directed the movie titled "Wall Street."

MATCH and RETURN

```
MATCH (p1:Person)-[r1:ACTED_IN]→(m:Movie)←[r2:ACTED_IN]-(p2:Person)
RETURN p1, p2
```

p1	p2
(:Person {name: "Michael Douglas"})	(:Person {name: "Charlie Sheen"})
(:Person {name: "Martin Sheen"})	(:Person {name: "Charlie Sheen"})
(:Person {name: "Michael Douglas"})	(:Person {name: "Martin Sheen"})
(:Person {name: "Charlie Sheen"})	(:Person {name: "Martin Sheen"})
(:Person {name: "Martin Sheen"})	(:Person {name: "Michael Douglas"})
(:Person {name: "Charlie Sheen"})	(:Person {name: "Michael Douglas"})
(:Person {name: "Michael Douglas"})	(:Person {name: "Martin Sheen"})
(:Person {name: "Martin Sheen"})	(:Person {name: "Michael Douglas"})

To return pairs of actors who have acted in the same movie together.

The WHERE CLAUSE

Extending the previous query using **WHERE:**

```
MATCH (p1:Person)-[r1:ACTED_IN]→(m:Movie)←[r2:ACTED_IN]-(p2:Person)
WHERE p1.name = "Martin Sheen"
RETURN p1,p2
```

p1	p2
(:Person {name: "Martin Sheen"})	(:Person {name: "Michael Douglas"})
(:Person {name: "Martin Sheen"})	(:Person {name: "Michael Douglas"})
(:Person {name: "Martin Sheen"})	(:Person {name: "Charlie Sheen"})

We first find all pairs of actors who have acted in the same movie (p1 and p2).
Then, we filter the results to only include pairs where the first actor (p1) is Martin Sheen.

The WHERE CLAUSE

```
MATCH (p1:Person{name: "Martin Sheen"})-[r1:ACTED_IN]→(m:Movie)←[r2:ACTED_IN]-(p2:Person)
RETURN p1,p2
```

p1	p2
(:Person {name: "Martin Sheen"})	(:Person {name: "Michael Douglas"})
(:Person {name: "Martin Sheen"})	(:Person {name: "Michael Douglas"})
(:Person {name: "Martin Sheen"})	(:Person {name: "Charlie Sheen"})

We start by finding Martin Sheen (p1), then we traverse through movies he has acted in, and finally, we find other actors (p2) who have also acted in those same movies.

This returns all pairs of actors who have acted in the same movie as Martin Sheen.

The WHERE CLAUSE

WHERE can be used with the following boolean operators: AND, OR, XOR, and NOT.

```
MATCH (p1:Person)-[r1:ACTED_IN]->(m:Movie)<-[r2:ACTED_IN]-(p2:Person)
WHERE p1.name="Martin Sheen" AND p2.age<30
RETURN p1, p2
```

https://neo4j.com/docs/cypher-manual/current/syntax/operators/#query-operators-boolean

The WHERE CLAUSE

WHERE can also be used to perform string matching, using STARTS WITH, ENDS WITH and CONTAINS

```
MATCH (p1:Person)-[r1:ACTED_IN]→(m:Movie)←[r2:ACTED_IN]-(p2:Person)
WHERE p1.name STARTS WITH "Martin" AND p2.name ENDS WITH "Sheen"
RETURN p1, p2
```

p1	p2
(:Person {name: "Martin Sheen"})	(:Person {name: "Charlie Sheen"})

https://neo4j.com/docs/cypher-manual/current/clauses/where/#query-where-string

The WITH CLAUSE

WITH can be used to manipulate the output before it is passed on to the following query parts.

```
MATCH (p1:Person)-[r1:ACTED_IN]→(m:Movie)←[r2:ACTED_IN]-(p2:Person)
WITH p1, p2, toUpper(p1.name) AS upperCaseName
WHERE upperCaseName STARTS WITH "MARTIN"
RETURN p1,p2
```

p1	p2
(:Person {name: "Martin Sheen"})	(:Person {name: "Charlie Sheen"})
(:Person {name: "Martin Sheen"})	(:Person {name: "Michael Douglas"})
(:Person {name: "Martin Sheen"})	(:Person {name: "Michael Douglas"})

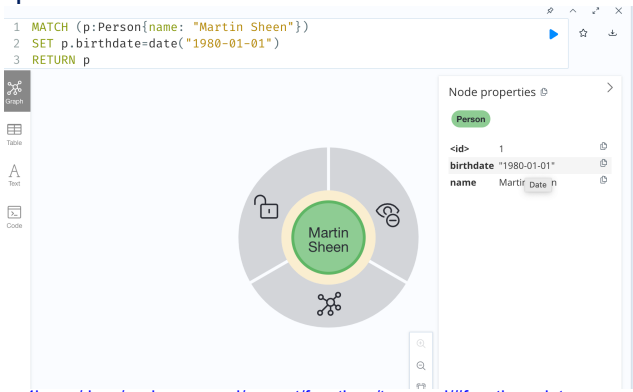
https://neo4j.com/docs/cypher-manual/current/clauses/with/

Lecture Outline

- Introduction to Cypher
- CRUD Actions:
  - Creating
  - Reading
  - Updating
  - Deleting
- Connect Neo4j with Python

Updating data via SET

- To update the data, we can use SET clause.
- We must first use a MATCH statement to match the pattern we want to find.



https://neo4j.com/docs/cypher-manual/current/functions/temporal/#functions-date

Lecture Outline

- Introduction to Cypher
- CRUD Actions:
  - Creating
  - Reading
  - Updating
  - Deleting
- Connect Neo4j with Python

Deleting data via DELETE

- Similarly to updating data, we must first query to find the data we want to delete, and then delete it via DELETE.

```
MATCH (p:Person {name: "Siri"})-[r:READ]->
      (b:Book {name: "Harry Potter"})
DELETE r
```

Deleting data via DELETE

If the node has any relationships, then Neo4j won't let us delete that node if any such relationship exist.

```
MATCH (p:Person {name:"Martin Sheen"})
DELETE p
```

ERROR **Neo.ClientError.Schema.ConstraintValidationFailed**

```
Cannot delete node<1>, because it still has relationships. To delete this node, you must
first delete its relationships.
```

Deleting data via DETACH DELETE

- We can delete both nodes and all outgoing/incoming relationships from that node using **DETACH DELETE**

```
MATCH (p:Person {name:"Martin Sheen"})
DETACH DELETE p
```

Table

Code

Deleted 1 node, deleted 3 relationships, completed after 1 ms.

Removing properties via REMOVE

- We can remove properties using **REMOVE:**

```
MATCH (p:Person {name:"Martin Sheen"})
REMOVE p.age
```

Lecture Outline

- Introduction to Cypher
- CRUD Actions:
  - Creating
  - Reading
  - Updating
  - Deleting
- Connect Neo4j with Python

Connect Neo4j with Python

1. **Seamless Data Pipeline:**
   Fetch data from Neo4j, process it using Python, and push results back to Neo4j, enabling end-to-end data workflows.
2. **Advanced Analytics:**
   Perform complex graph analytics, machine learning, and statistical analysis on Neo4j data using Python's libraries.
3. **Enhanced Visualization:**
   Visualize graph data and analysis results effectively using Python libraries like Matplotlib, Plotly, and NetworkX.

Python packages

- neo4j: https://neo4j.com/docs/python-manual/current/
- py2neo: https://pypi.org/project/py2neo/

Demo code is on LMS.

Summary

- Cypher: A powerful query language for Neo4j.
- Functionality: Enable CRUD operations: Create, Read, Update and Delete data.
- Learn More: Explore Cypher clauses in https://neo4j.com/docs/cypher-manual/current/clauses/
- Similar to relational databases, Neo4j can be seamlessly integrated with Python, allowing for enhanced data manipulation and analysis capabilities.

Copyright Notice