Graph Analytics Course

Graph Databases and Querying

Master Artificial Intelligence & Society
2026-2027

Dauphine UNIVERSITÉ PARIS SCIENCES & LETTRES

Agenda

- Lecture 1: Foundations of Graph Databases
- Lecture 2: Querying Graphs with Cypher
- Lecture 3: Advanced Querying with Cypher

General Information

- Graph Databases and Querying : 3 x lectures with a hands-on labs
  - Lecture 1: *Foundations of Graph Databases*. Introduces graph theory and positions graph databases within the broader landscape of database technologies (relational, document, key-value, column-family). Introduces Neo4j.
    Lab 1: instal Neo4j and explore a first graph dataset.
  - Lecture 2: *Querying Graphs with Cypher*. Introduces Cypher, Neo4j's declarative query language, covering data creation, pattern matching, and data modification.
    Lab2: build a graph and practice the full range of basic Cypher operations.
  - Lecture 3: *Advanced Querying with Cypher*. Covers advanced traversal techniques and query composition.
    Lab3: consolidate skills through applied exercises.

Graph Analytics Course

Graph Databases and Querying

Querying Graphs with Cypher

Master Artificial Intelligence & Society
2026-2027

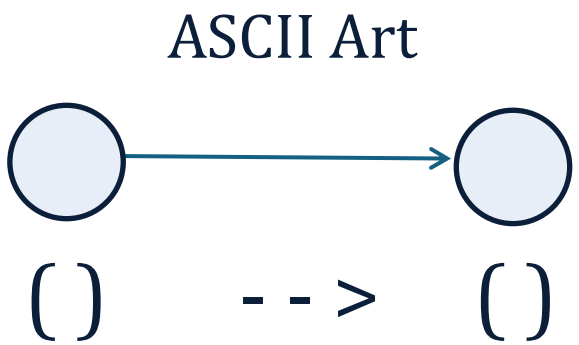
Dauphine UNIVERSITÉ PARIS SCIENCES & LETTRES

Agenda

1. What is Cypher ?
2. Creating data: CREATE
3. Querying data: MATCH, WHERE, RETURN
4. Sorting, limiting, and aggregating results
5. Updating data: SET, REMOVE, DELETE
6. Upserting with MERGE
7. Functions and parameters

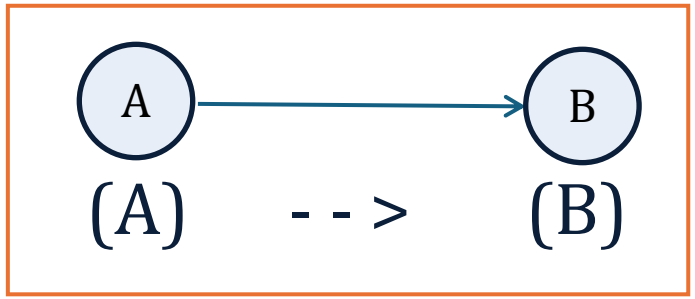
What is Cypher ?

- Neo4j's declarative query language: graph pattern matching (“SQL for graphs”)
- Similar in spirit to SQL but built around patterns of nodes and relationships instead of tables.
- Designed to be readable: queries look like ASCII-art drawings of the graph pattern being matched.

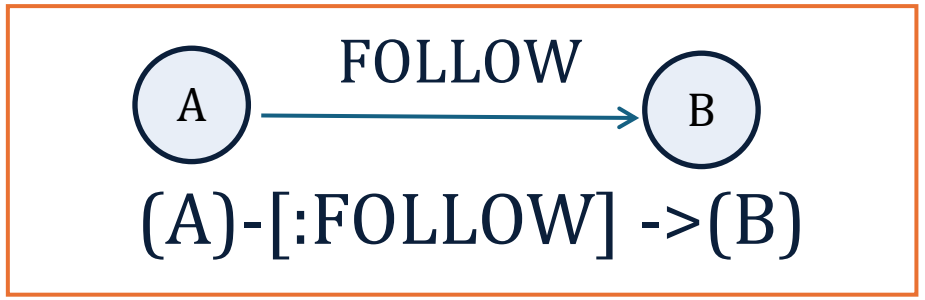


What is Cypher ?

- Core idea
  - Named nodes



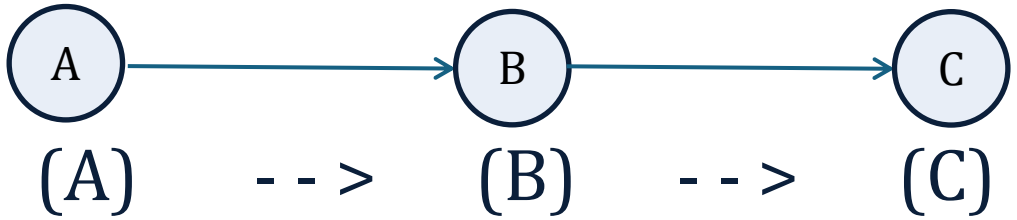
  - Types relationships



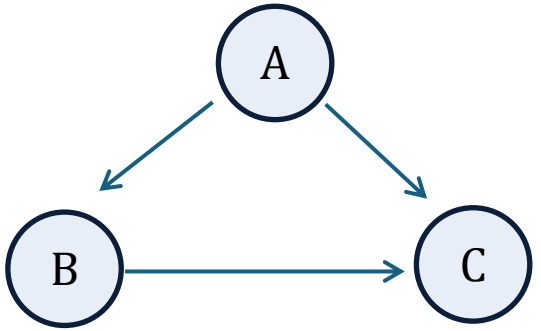


What is Cypher ?

- Core idea
  - Describing a path



  - Multiple paths



(A) - - > (B) - - > (C), (A) - - > (C)

(A)- - > (B) - - > (C), (A) - - > (C)
(A) - - > (B) - - > (C) <- - (A)

What is Cypher ?

- Case Sensitivity & Style Conventions
  - Keywords : UPPERCASE (e.g. MATCH, RETURN)
  - Labels : PascalCase (e.g. :Person, :Movie)
  - Relationship types : UPPER_SNAKE_CASE (e.g. :ACTED_IN)
  - Property keys and variables: camelCase (e.g. releaseYear, firstName)
  - Statements are terminated with a semicolon ;

What is Cypher ?

- Node syntax (variables p refer to the node in the query)

```
()                          // an anonymous node
(p)                         // a node, bound to variable p
(p:Person)                  // a node with label Person
(p:Person:Employee)         // a node with two labels
(p:Person {name:'Alice'}) // a node with a property filter
```

- Relationship Syntax

```
-->                                 // any relationship, any type
-[:FRIEND_OF]->                     // a specific type, directed
-[f:FRIEND_OF]->                    // bound to variable f
-[:FRIEND_OF {since:2021}]->        // with a property filter
-[:FRIEND_OF|COLLEAGUE_OF]->      // either of two types
-[*1..3]->                         // variable length, 1 to 3 hops
```

What is Cypher ?

- Combining nodes and relationships: a Pattern
  - Reads as: “a Person node named Alice, connected by a FRIEND_OF relationship to another Person node b”

```
(a:Person {name:'Alice'})-[:FRIEND_OF]->(b:Person)
```

  - This whole expression is a pattern : the fundamental unit of Cypher matching

Agenda

1. What is Cypher ?
2. Creating data: CREATE
3. Querying data: MATCH, WHERE, RETURN
4. Sorting, limiting, and aggregating results
5. Updating data: SET, REMOVE, DELETE
6. Upserting with MERGE
7. Functions and parameters

CREATE

- Single node
  - CREATE always adds new data : it never checks if something similar exists
  - Creates one new node with label Person and two properties.

```
CREATE (p:Person {name: 'Alice', age: 30})
RETURN p;
```

- Multiple nodes
  - Comma-separated patterns create several disconnected elements in one statement

```
CREATE (a:Person {name: 'Alice', age: 30}),
       (b:Person {name: 'Bob', age: 27});
```

CREATE

- Nodes and a relationship together
  - Creates two nodes and the relationship between them in a single statement

```
CREATE (a:Person {name: 'Alice'})
       -[:FRIEND_OF {since: 2021}]->
       (b:Person {name: 'Bob'});
```

Agenda

1. What is Cypher ?
2. Creating data: CREATE
3. Querying data: MATCH, WHERE, RETURN
4. Sorting, limiting, and aggregating results
5. Updating data: SET, REMOVE, DELETE
6. Upserting with MERGE
7. Functions and parameters

MATCH

- The Core Query Clause
  - MATCH searches the existing graph; it never creates anything
  - Finds every node with label Person and returns it

```
MATCH (p:Person)
RETURN p;
```

- Filtering by Inline Property
  - Inline { } filters restrict the match to nodes with exactly that property value.

```
MATCH (p:Person {name: 'Alice'})
RETURN p;
```

MATCH

- Traversing a Relationship
  - Finds all people that Alice has a FRIEND_OF relationship to
  - b.name accesses the name property of node b

```
MATCH (a:Person {name: 'Alice'})-[:FRIEND_OF]->(b:Person)
RETURN b.name;
```

- Ignoring Direction
  - Omitting the arrowhead matches the relationship in either direction
  - Useful when the relationship is conceptually symmetric (e.g. friendship)

```
MATCH (a:Person {name: 'Alice'})-[:FRIEND_OF]-(b:Person)
RETURN b.name;
```

MATCH

- Multi-Hop Patterns
  - Anonymous node ( ) in the middle: we don't care about its identity, only that it exists
  - Chaining relationship patterns is how you express multi-hop traversals

```
MATCH (a:Person {name:'Alice'})-[:FRIEND_OF]->()
      -[:FRIEND_OF]->(fof:Person)
RETURN DISTINCT fof.name;
```

MATCH

- Connecting existing nodes
  - First MATCH the existing nodes, then CREATE a relationship between them
  - Standard way to link data created in separate steps

```
MATCH (a:Person {name: 'Alice'}), (b:Person {name: 'Bob'})
CREATE (a)-[:FRIEND_OF]->(b);
```

WHERE

- WHERE Clause
  - Filters rows after the pattern in MATCH has been found
  - Can express conditions that inline { } property filters cannot (ranges, comparisons, boolean logic).
  - Syntax: MATCH (...) WHERE <condition> RETURN ...
  - Comparison operators: =, <>, <, >, <=, >=
  - Boolean combinators: AND, OR, NOT, XOR

```
MATCH (p:Person)
WHERE p.age > 25 AND p.age <= 40
RETURN p.name, p.age;
```

WHERE

- String Predicates

```
MATCH (p:Person)
WHERE p.name STARTS WITH 'A'
   OR p.name ENDS WITH 'e'
   OR p.name CONTAINS 'ic'
RETURN p.name;
```

- Checking for NULL and Labels
  - IS NULL / IS NOT NULL check whether a property exists
  - p:Employee in a WHERE clause checks whether the node carries that label

```
MATCH (p:Person)
WHERE p.age IS NOT NULL
  AND p:Employee
RETURN p.name;
```

- Lists and IN

```
MATCH (p:Person)
WHERE p.name IN ['Alice', 'Bob','Carla']
RETURN p;
```

RETURN

- RETURN Clause
  - Chooses what data flows out of the query.
  - Can return whole nodes/relationships, individual properties, or computed expressions
  - Supports aliasing with AS for friendlier column names

- Aliasing and Expressions

```
MATCH (p:Person)
RETURN p.name AS name,
       p.age AS age,
       p.age >= 18 AS isAdult;
```

- RETURN DISTINCT
  - Removes duplicate rows
    (essential once traversals can revisit the same node via multiple paths)

```
MATCH (a:Person)-[:FRIEND_OF]->(:Person)-
[:FRIEND_OF]->(fof)
RETURN DISTINCT fof.name;
```

Agenda

1. What is Cypher ?
2. Creating data: CREATE
3. Querying data: MATCH, WHERE, RETURN
4. Sorting, limiting, and aggregating results
5. Updating data: SET, REMOVE, DELETE
6. Upserting with MERGE
7. Functions and parameters

Sorting, Limiting

- Sorting : ORDER BY
  - Default order is ascending (ASC), DESC for descending

```
MATCH (p:Person)
RETURN p.name, p.age
ORDER BY p.age DESC;
```

- Limiting : LIMIT and SKIP
  - LIMIT : caps the number of returned rows, important for performance on large graphs
  - SKIP : discards the first n rows

```
MATCH (p:Person)
RETURN p.name
ORDER BY p.name
SKIP 10
LIMIT 5;
```

Aggregation

- Aggregating functions: count( ), sum( ), avg( ), min( ), max( ), collect( )
- Unlike SQL, there is no explicit GROUP BY (any non-aggregated expression in RETURN automatically becomes the grouping key).

- p.name is the implicit grouping key because it's the only non-aggregated return item.

```
MATCH (p:Person)-[:FRIEND_OF]->()
RETURN p.name, count(*) AS
numberOfFriends
ORDER BY numberOfFriends DESC;
```

- collect( ) gathers values into a list — handy to bring graph results back as arrays.

```
MATCH (p:Person)
RETURN avg(p.age)    AS avgAge,
       min(p.age)    AS youngest,
       max(p.age)    AS oldest,
       collect(p.name) AS allNames;
```

Agenda

1. What is Cypher ?
2. Creating data: CREATE
3. Querying data: MATCH, WHERE, RETURN
4. Sorting, limiting, and aggregating results
5. Updating data: SET, REMOVE, DELETE
6. Upserting with MERGE
7. Functions and parameters

Updating data

- SET: updates (or adds) a property on a matched node or relationship

```
MATCH (p:Person {name: 'Alice'})
SET p.age = 31;
```

- Multiple properties and labels

```
MATCH (p:Person {name: 'Alice'})
SET p.age = 31, p.city = 'Paris';
```

```
MATCH (p:Person {name: 'Alice'})
SET p:Employee;  // add a label
```

- Replacing all properties

```
MATCH (p:Person {name: 'Alice'})
SET p = {name: 'Alice', age: 31};
```

  - SET p = { … } replaces all existing properties with the given map
  - SET p += { … } merge in new properties without deleting the others

Updating data

- REMOVE: deletes a property or a label from a node/relationship.

```
MATCH (p:Person {name: 'Alice'})
REMOVE p.city;
```

```
MATCH (p:Person {name: 'Alice'})
REMOVE p:Employee; //label
```

- DELETE: deletes the matched node(s)
  - fails if the node still has relationships attached

```
MATCH (p:Person {name: 'Alice'})
DELETE p;
```

Updating data

- DETACH DELETE: deletes the node and all its relationships in one go

```
MATCH (p:Person {name: 'Alice'})
DETACH DELETE p;
```

- Deleting only a relationship: removes just the relationship, keeping both nodes intact

```
MATCH (:Person {name:'Alice'})-[f:FRIEND_OF]->(:Person {name:'Bob'})
DELETE f;
```

Agenda

1. What is Cypher ?
2. Creating data: CREATE
3. Querying data: MATCH, WHERE, RETURN
4. Sorting, limiting, and aggregating results
5. Updating data: SET, REMOVE, DELETE
6. Upserting with MERGE
7. Functions and parameters

Why MERGE?

- CREATE always inserts new data (running the same CREATE twice creates duplicates)

- MERGE works like “find this pattern; if it doesn't exist, create it”

- Example :
  - first run creates Alice, then matches the existing Alice (no duplicate created)

```
MERGE (p:Person {name: 'Alice'})
RETURN p;
```

- MERGE: ON CREATE / ON MATCH
  - run different SET logic depending on whether the node was just created or already existed.

```
MERGE (p:Person {name: 'Alice'})
ON CREATE SET p.createdAt=timestamp()
ON MATCH  SET p.lastSeen =timestamp();
```

Why MERGE?

- MERGE on relationships: ensures the relationship exists exactly once, without duplicating it on re-runs

```
MATCH (a:Person {name:'Alice'}), (b:Person {name:'Bob'})
MERGE (a)-[:FRIEND_OF]->(b);
```

- MERGE: Key Pitfall
  - MERGE matches on the entire pattern you specify.
  - MERGE (p:Person {name:'Alice', age:30}) will create a NEW node if a Person named Alice exists but with a different age.
  - Best practice: MERGE on a minimal, unique key, then use SET to add other properties.

Agenda

1. What is Cypher ?
2. Creating data: CREATE
3. Querying data: MATCH, WHERE, RETURN
4. Sorting, limiting, and aggregating results
5. Updating data: SET, REMOVE, DELETE
6. Upserting with MERGE
7. Functions and parameters

Built-in Function Categories

- String: toUpper( ), toLower( ), …
- Numeric: abs(), round(), ceil(), floor(), rand().
- List: size( ), head( ), tail( ),…
- Temporal: date( ), duration( )
- Node/relationship: id( ), labels( ), type( ), properties( ), keys( ).
- Example: size((p)-[:FRIEND_OF]->( )) counts matches of a pattern directly inline.

```
MATCH (p:Person)
RETURN toUpper(p.name) AS shoutName,
       size((p)-[:FRIEND_OF]->()) AS numFriends,
       labels(p) AS allLabels;
```

Built-in Function Categories

- Why Use Parameters?
  - Hard-coding values ('Alice') inside a query is fine for exploration, but bad practice for applications.
  - Parameters separate the query structure from the data
  - Example:

```
:param name => 'Alice'

MATCH (p:Person {name: $name})
RETURN p;
```

In Neo4j Browser, “:param” sets a parameter and “$name” references it in the query

Built-in Function Categories

- Putting it together: A Mini Script

```
CREATE (a:Person {name:'Alice', age:30}),
       (b:Person {name:'Bob',   age:27}),
       (c:Person {name:'Carla', age:35}),
       (a)-[:FRIEND_OF {since:2021}]->(b),
       (b)-[:FRIEND_OF {since:2022}]->(c);

MATCH (p:Person)-[:FRIEND_OF]->(f)
RETURN p.name, collect(f.name) AS friends;
```

Other queries

- Visualizing a pattern




```
MATCH (a:Person)-[:FRIEND_OF]->(b:Person)
```

- Comments in Cypher
  - // starts a line comment and /* ... */ for block comments

- Multiple MATCH clauses combine results as a cartesian product unless the patterns share a variable

```
MATCH (a:Person {name:'Alice'})
MATCH (b:Person {name:'Bob'})
RETURN a, b;
```

Other queries

- Checking relationship existence in WHERE
  - A pattern inside WHERE acts as an existence check (true/false), without binding new rows for it

```
MATCH (p:Person)
WHERE (p)-[:FRIEND_OF]->(:Person {name:'Bob'})
RETURN p.name;
```

- Working with maps
  - properties( ) returns the node's key-value pairs as a map.
  - keys( ) returns just the property names as a list.

```
MATCH (p:Person {name:'Alice'})
RETURN properties(p) AS allProps, keys(p) AS
propertyNames;
```

Other queries

- A first look at CASE
  - Simple conditional expression, usable inside RETURN, SET, WHERE.

```
MATCH (p:Person)
RETURN p.name,
  CASE
    WHEN p.age < 18 THEN 'minor'
    WHEN p.age < 65 THEN 'adult'
    ELSE 'senior'
  END AS ageGroup;
```

Summary

- Cypher patterns visually mirror the graph: (node)-[:REL]->(node)
- CREATE adds data; MATCH/WHERE/RETURN query it; SET/REMOVE/DELETE update it
- MERGE is the safe way to avoid duplicating data
- Aggregation has no explicit GROUP BY. It's implicit from non-aggregated RETURN items
- Parameters keep queries reusable, cacheable, and safe

What's Next

- Lab 1: install Neo4j
- Lab 2: hands-on Cypher practice with the movie graph (a Neo4j dataset)