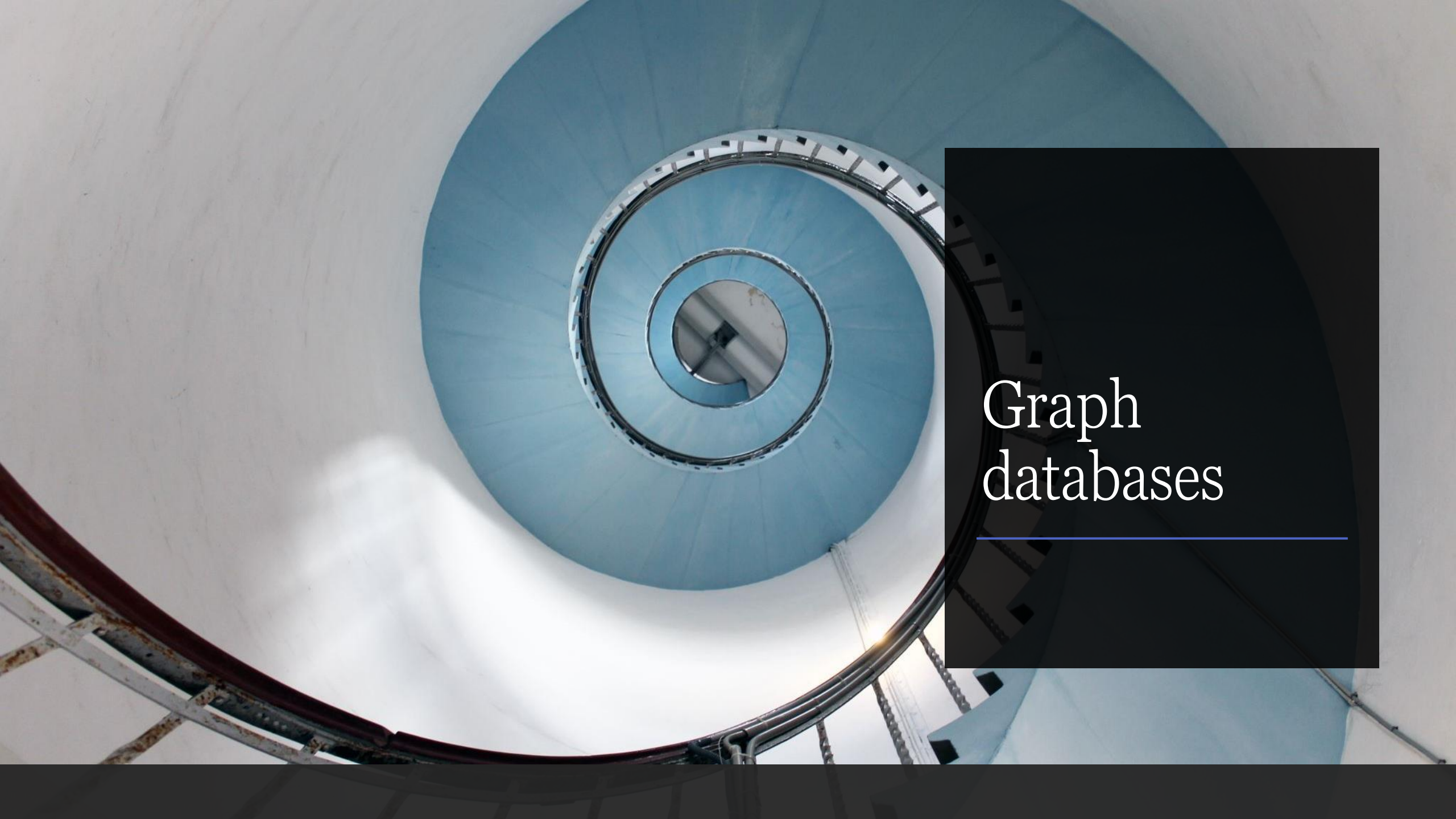


Graph databases

Graph databases

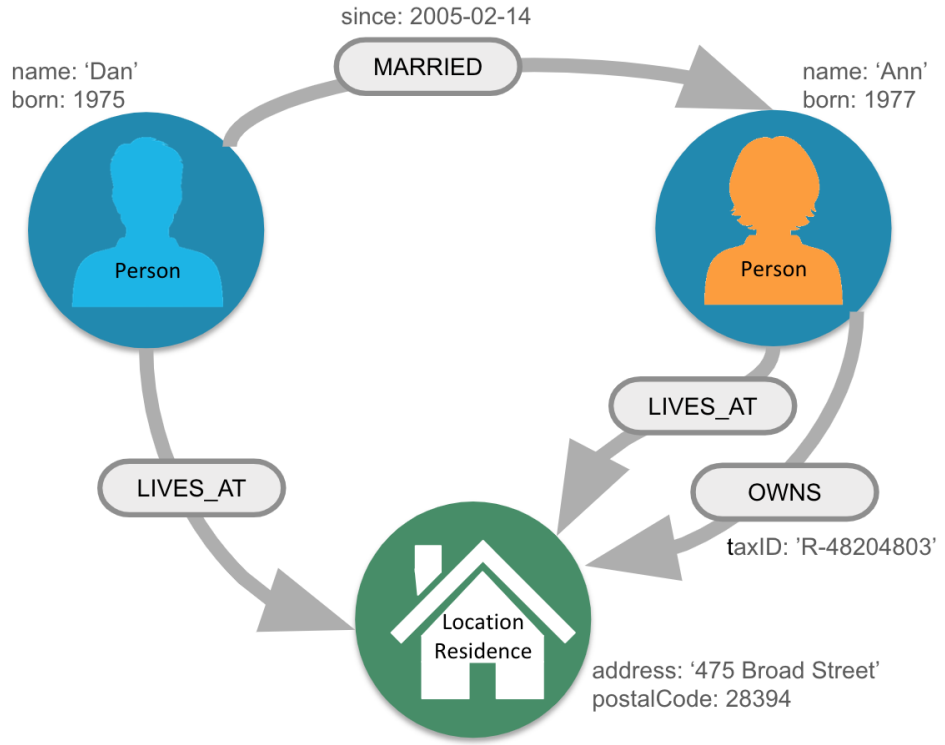
Definition

- A graph database is an online database management system with CRUD operations working on a graph data model.
- Relationships take first priority in graph databases.
  - Your app don't have to infer data connections using foreign keys or out-of-band processing

Concepts

- Graph: Collection of two elements: nodes and relationships
- Node: It represents an entity (a person, place, thing, category or other piece of data) consisting of labels, used to define types for nodes, and attributes.
- Relationship: It represents how two nodes are connected. It has labels and attributes as well.

Sample graph



Relational DBs vs Graph DBs

Relational	Graph
Rows	Nodes
Joins	Relationships
Table names	Labels
Columns	Properties

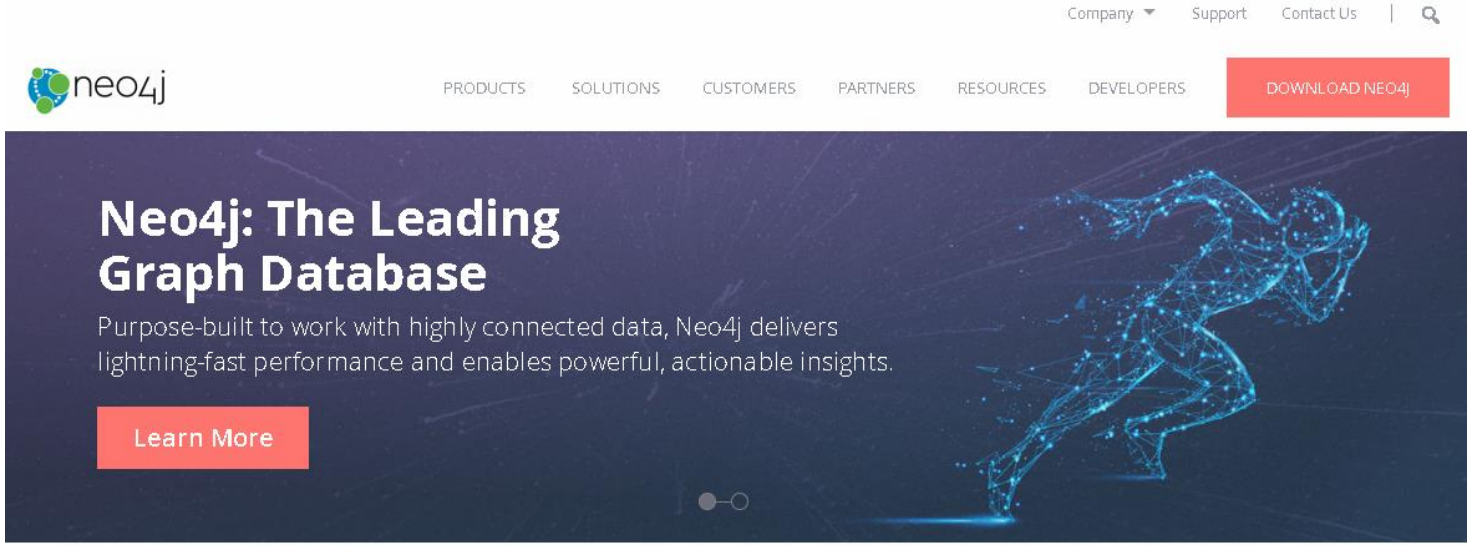
Relational	Graph
Each column must have a field value.	Nodes with the same label aren't required to have the same set of properties.
Joins are calculated at query time.	Relationships are stored on disk when they are created.
A row can belong to one table.	A node can have many labels.

Neo4j: A graph database

Neo4J

Definition

- The Neo4j Graph Platform enables developers to create applications that are best architected as graph-databases



Cypher: Query Language

Cypher

Definition

- Cypher is a declarative query language that allows for expressive and efficient querying and updating of graph data
  - Declarative languages: They focus on the clarity of expressing **what** to retrieve from a graph, not on **how** to retrieve it

Query elements

- Nodes: Represented by pairs of parenthesis ()
- Labels: They are used to group nodes and filter queries against the graph.

Creating elements

A node is an element of a graph representing a domain entity that has zero or more labels, properties, and relationships to or from other nodes in the graph

Creating nodes

- When you create a node, you can add it to the graph without connecting it to another node.
- CREATE:

```
CREATE (optionalVariable optionalLabels {optionalProperties})
```

- SET: Using with MATCH to find a node first and then add the given label

```
SET x:Label // adding one label to node referenced by the variable x
```

- MERGE: A combination of MATCH and CREATE. If the node exists, returns it, otherwise it creates it.

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

Create

Definition

- The CREATE clause is used to create nodes and relationships.

Syntax

- Single node
  - Create (n)
- Node with label
  - Create (n:LABEL)
- Node with multiple labels
  - Create(n:LABEL_1:LABEL_2)
- Node with properties
  - Create(n:LABEL {key:value, key2:value})

Merge

Definition

- The MERGE clause ensures that a pattern exists in the graph. Either the pattern already exists, or it needs to be created.

Syntax

- Similar to MATCH, but will crate node if not present
  - MERGE (some query involving n) RETURN n
  - MATCH (query involving n and m)
    MERGE (n)-[r]->(m)
    RETURN n, m, r

Additional keywords

OnCreate and OnMatch

- You can set properties for indicating whether the node is created or matched.

Syntax

MERGE (node:label {properties . . . . . .})

ON CREATE SET node.property=value

ON MATCH SET node.property=value

Examples

Create a node representing “John Frost”

```
create ({first_name:"John", last_name:"Frost"});
```

Create a node representing the student “John Frost”

```
create (:Student {first_name:"John", last_name:"Frost"});
```

Set the property bachelor with the value “ITC” to the student.

```
match (st:Student {first_name:"John"}) set st.bachelor = "ITC";
```

Add a new label to the student to identify it as a person.

```
match (st:Student {first_name:"John"}) set st:Person;
```

Creating relationships

A relationship is a connection between nodes.

Creating relationships

```
CREATE (x)-[:REL_TYPE]->(y)
```

Creating relationships

Creating relationships:

- CREATE:

```
CREATE (x)-[:REL_TYPE]->(y)
```

- SET:

```
SET r.propertyName = value
```

Examples

Create a student a teacher, and a relationship between them:

- create(:Student {first_name:"John", last_name:"Frost", bachelor:"ITC"});
- create(:Teacher {first_name:"Ivan", last_name:"Guerrero"});
- match (n),(m) where id(n)=39 and id(m)=40 create (m)-[:Teaches]->(n);

Delete

Definition

- The DELETE clause is used to delete nodes, relationships or paths.

Syntax

- Single node
  - MATCH(n: QUERY)
  - DELETE n
- All nodes and relationships
  - MATCH (n: QUERY)
  - DETACH DELETE n
- Relationships
  - MATCH (n)-[r: QUERY]
  - DELETE r

Element retrieval

MATCH:

- The MATCH clause performs a pattern match against the data in the graph.
- During the query processing, the graph engine traverses the graph to find all nodes that match the graph pattern.

Retrieving values

- As part of query, you can return nodes or data from the nodes using the RETURN clause.
- The RETURN clause must be the last clause of a query to the graph.

Syntax

```
MATCH (variable:Label {prop1: value, prop2: value})
RETURN variable.prop3, variable.prop4
```

Element retrieval using relationships

In a graph where you want to retrieve nodes, you can use relationships between nodes to filter a query

```
() // a node
()--() // 2 nodes have some type of relationship
()-->() // the first node has a relationship to the second node
()<--() // the second node has a relationship to the first node
```

Syntax:

```
MATCH (node1)-[:REL_TYPE]->(node2)
RETURN node1, node2
```

Where clauses

Definition

- In a WHERE clause, you can place conditions that are evaluated at runtime to filter the query.

Syntax

MATCH EXPRESSION

WHERE CONDITIONS
RETURN VALUES

Examples

Sample queries

Display the database schema:

CALL db.schema.visualization

CALL call db.schema.relTypeProperties

CALL db.schema.nodeTypeProperties

Retrieve all the nodes in the database:

MATCH (n) RETURN n

Retrieve all the nodes of a given type:

MATCH (n:TYPE) RETURN n

OR

MATCH (n) WHERE n:TYPE RETURN n

Retrieve all the nodes of a given type with a given property:

MATCH (n:TYPE {PROPERTY_NAME: VALUE}) RETURN n

OR

MATCH (n) WHERE n:TYPE AND n.PROPERTY = VALUE RETURN n

Retrieve a given property of all the nodes of a given type

MATCH (n:TYPE) RETURN n.PROPERTY_NAME

Queries using relationships

Retrieve all the nodes related to another node of a given type

```
MATCH (n:TYPE)->(m) RETURN n
```

Retrieve all the nodes of a given type with a certain relation with nodes of another type

```
MATCH (n:TYPE_1)-[:RELATION]->(m:TYPE_2) RETURN n
```

Retrieve all the nodes and its relation with nodes of another type

```
MATCH (n:TYPE)-[rel]->(m) return n, type(rel)
```

Additional processing

WITH clause

During the execution of a MATCH clause, you can specify that you want some intermediate calculations or values that will be used for further processing of the query, or for limiting the number of results before further processing is done.

With the WITH clause, you specify the variables from the previous part of the query you want to pass on to the next part of the query.

Available operations within a WITH clause:

- Count: Determines the number of occurrences of the given variable
- Collect: Builds a list with the elements proposed

Example

Determine the name of the actors that acted in 2 or 3 movies. Obtain the actor's name, the number of movies and the titles of such movies.

```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)
WITH a, count(a) AS numMovies, collect(m.title) as movies
WHERE numMovies > 1 AND numMovies < 4
RETURN a.name, numMovies, movies
```

a.name	numMovies	movies
"Bill Paxton"	3	["Apollo 13", "Twister", "A League of Their Own"]
"Rosie O'Donnell"	2	["Sleepless in Seattle", "A League of Their Own"]
"Oliver Platt"	2	["Frost/Nixon", "Bicentennial Man"]
"Helen Hunt"	3	["As Good as It Gets", "Twister", "Cast Away"]
"Gary Sinise"	2	["The Green Mile", "Apollo 13"]
"Nathan Lane"	2	["Joe Versus the Volcano", "The Birdcage"]

Information resources

Create

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

Merge

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

Delete

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

https://neo4j.com/graphacademy/online-training/introduction-to-neo4j/

Cypher: Query Language

ADVANCED QUERIES

Sample database

We'll be using this database to rehearse: https://github.com/neo4j-graph-examples/wwc2019

Download the wwc2019-50.dump file

Put it in the Files section

Create a new DBMS from the dump file

Patterns, Clauses and Functions

Patterns

Patterns

Definition

- They describe the shape of the data you're looking for

Types of patterns

- For nodes
- For related nodes
- For labels
- Specifying properties
- Variable length
- Assigning of patterns to variables

Types of patterns

Patterns for nodes

- A single node
  - (a)
- Two nodes and a relationship between them
  - (a)-->(b)
- Defining paths
  - (a)-->(c)<--(b)

Patterns for labels

- A node with a label
  - (a:Label)
- A node with two labels
  - (a:Label1:Label2)

Patterns for properties

- A node with a property
  - (a {property: value})
- A relationship with a property
  - (a)-[{property:value}]->(b)

Examples

Determine the number of tournaments available

```
match (t:Tournament)
return count(t)
```

Determine the number of teams per tournament

```
match (t:Tournament)<--(te:Team)
return t.name, count(te)
```

Determine the number of matches in the France 2019 world cup

```
match (t:Tournament {year:2019})<--(m:Match)
return count(m)
```

Types of patterns

Patterns for relationships

- Two nodes with a relationship where we don’t care about the direction
  - (a)--(b)
- Two nodes with a relationship from left to right
  - (a)-->(b)
- Two nodes with a relationship from right to left
  - (a)<--(b)
- Two nodes with a named relationship
  - (a)-[r]->(b)

Examples

Find the final matches of every tournament

```
match (t2:Team)-->(m:Match)<--(t:Team)
where m.stage="Final"
return t.name, t2.name
```

Find the final matches of every tournament and incorporate the name of the tournament. Finally, sort the results by the tournament's year

```
match (t2:Team)-->(m:Match {stage:"Final"})<--(t:Team),
(m)-->(tour:Tournament)
where t.name < t2.name
return tour.name, t.name, t2.name
order by tour.year
```

Now, incorporate the results of the matches

```
match (t2:Team)-[r2:PLAYED_IN]->(m:Match
{stage:"Final"})<-[r1:PLAYED_IN]-(t:Team), (m)--
>(tour:Tournament)
where t.name < t2.name
return tour.name, t.name, t2.name,
r1.score + "-" + r2.score as score
```

Types of patterns

Variable-length patterns

- Two related nodes with a node in the middle
  - (a)-->()-->(b) is equivalente to (a)-[*2]->(b)
- Two related nodes with 2, 3 or 4 intermediate nodes
  - (a)-[*3..5]->(b)
    - IMPORTANT: Both bounds can be omitted

Assigning paths to variables

- p = (a)-[*3..5]->(b)

Pattern comprehension

Definition

- Creates a list based on matchings of a pattern

Syntax

- [PATTERN | ELEMENT_TO_ADD] AS VARIABLE_NAME

Example

```
MATCH (a:Person { name: 'Keanu Reeves' })
RETURN [(a)-->(b) WHERE b:Movie | b.released] AS years
```

Output

- [1997,2000,2003,1999,2003,2003,1995]

Clauses

Clauses

Definition

- Operations available for handling information

Types

- Reading clauses: read data from the database
- Sub-reading clauses: operate as part of reading clauses
- Projecting clauses: define which expressions to return in the result set
- Writing clauses: write the data to the database

Types of clauses

Reading clauses

- MATCH: Specify the patterns to search for in the database.
  - Use all the available patterns
- OPTIONAL MATCH: Specify the patterns to search for in the database while using nulls for missing parts of the pattern.
  - Use all the available patterns

Sub-reading clauses

- **WHERE:** Adds constraints to the patterns
- **WHERE EXISTS:** used to filter the results
- ORDER BY [ASCENDING | DESCENDING]
- **SKIP:** Defines from which row to start including the rows in the output
- **LIMIT:** Constrains the number of rows in the output

Types of clauses

Projecting clauses

- RETURN: Defines what to include in the query result set
- WITH: Allows query parts to be chained together, piping the results from one to be used as starting points or criteria in the next.
- UNWIND: Expands a list into a sequence of rows

Examples

Determine the top scorer in France 2019 world cup

```
match (p:Person)-[r:SCORED_GOAL]->(m:Match)-
[:IN_TOURNAMENT]->(t:Tournament {year:2019})
with p, count(p) as goals
return p.name, goals
order by goals desc limit 1
```

Determine the names of the all the players who scored a goal in a final game

```
match (m:Match {stage:"Final"})-->(t:Tournament)
match (p:Person)-[:IN_SQUAD]->(s:Squad)<--(team:Team)-->(m)
match (p)-[:SCORED_GOAL]->(m) return t.name, team.name,
collect(distinct(p.name))
```

Functions

Functions

Definition

- Scripts to enrich the behavior of the clauses

Types

- Predicate: Return either true or false for the given arguments
- Scalar: Return a single value
- Aggregating: Take multiple values as arguments, and calculate and return an aggregated value from them
- List: Return lists of other values
- String: Used to manipulate strings or to create a string representation of another value

Types of functions

Predicate functions
- all(), any(), exists(), none(), single()

Scalar functions
- head(), id(), last(), length(), properties(), size(), type()

Aggregating functions
- avg(), count(), max(), min(), sum(), collect()

List functions
- keys(), labels(), nodes(), relationships(), reverse(), tail()

String functions
- left(), right(), trim(), split(), toUpper(), toLower(), substring(), replace(), reverse()

collect(): Returns a list containing the values returned by an expression

tail(): Returns all but the first element in a list.

Information resources

CYPHER Manual:

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

Patterns

- https://neo4j.com/docs/cypher-manual/current/syntax/patterns/

Clauses

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

Functions

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