

**B4M36DS2, BE4M36DS2: Database Systems 2**

<http://www.ksi.mff.cuni.cz/~svoboda/courses/171-B4M36DS2/>

Lecture 10

# **Graph Databases: Neo4j**

**Martin Svoboda**

[martin.svoboda@fel.cvut.cz](mailto:martin.svoboda@fel.cvut.cz)

4. 12. 2017

**Charles University in Prague**, Faculty of Mathematics and Physics

**Czech Technical University in Prague**, Faculty of Electrical Engineering

# Lecture Outline

## Graph databases

- Introduction

## Neo4j

- Data model: **property graphs**
- **Traversal framework**
- **Cypher** query language
  - Read, write, and general clauses

# Graph Databases

## Data model

- **Property graphs**
  - **Directed / undirected graphs**, i.e. collections of ...
    - **nodes** (vertices) for real-world entities, and
    - **relationships** (edges) among these nodes
  - Both the nodes and relationships can be associated with additional **properties**

## Types of databases

- **Non-transactional** = small number of large graphs
- **Transactional** = large number of small graphs

# Graph Databases

## Query patterns

- Create, update or remove a node / relationship in a graph
- **Graph algorithms** (shortest paths, spanning trees, ...)
- General **graph traversals**
- **Sub-graph** queries or **super-graph** queries
- Similarity based queries (approximate matching)

# Neo4j Graph Database



# Neo4j

## Graph database

- <https://neo4j.com/>
- Features
  - Open source, massive scalability (billions of nodes), high availability, fault-tolerant, master-slave replication, **ACID transactions**, embeddable, ...
  - Expressive graph query language (**Cypher**), **traversal framework**
- Developed by **Neo Technology**
- Implemented in Java
- Operating systems: cross-platform
- Initial release in 2007

# Data Model

Database system structure

Instance → single **graph**

**Property graph** = directed labeled multigraph

- Collection of vertices (**nodes**) and edges (**relationships**)

Graph **node**

- Has a unique (internal) **identifier**
- Can be associated with a **set of labels**
  - Allow us to categorize nodes
- Can also be associated with a **set of properties**
  - Allow us to store additional data together with nodes

# Data Model

## Graph **relationship**

- Has a unique (internal) **identifier**
- Has a **direction**
  - Relationships are equally well traversed in either direction!
  - Directions can even be ignored when querying at all
- Always has a **start** and **end node**
  - Can be recursive (i.e. loops are allowed as well)
- Is associated with **exactly one type**
- Can also be associated with a **set of properties**



# Data Model

Node and relationship **property**

- **Key-value pair**
  - Key is a string
  - Value is an **atomic value** of any primitive data type, or an **array of atomic values** of one primitive data type

Primitive **data types**

- boolean – **boolean** values true and false
- byte, short, int, long – **integers** (1B, 2B, 4B, 8B)
- float, double – **floating-point numbers** (4B, 8B)
- char – one Unicode character
- String – sequence of **Unicode characters**

# Sample Data

## Sample graph with **movies and actors**

```
(m1:MOVIE { id: "vratnelahve", title: "Vratné lahve", year: 2006 })
(m2:MOVIE { id: "samotari", title: "Samotáři", year: 2000 })
(m3:MOVIE { id: "medvidek", title: "Medvídek", year: 2007 })
(m4:MOVIE { id: "stesti", title: "Šťěstí", year: 2005 })

(a1:ACTOR { id: "trojan", name: "Ivan Trojan", year: 1964 })
(a2:ACTOR { id: "machacek", name: "Jiří Macháček", year: 1966 })
(a3:ACTOR { id: "schneiderova", name: "Jitka Schneiderová", year: 1973 })
(a4:ACTOR { id: "sverak", name: "Zdeněk Svěrák", year: 1936 })

(m1)-[c1:PLAY { role: "Robert Landa" }]->(a2)
(m1)-[c2:PLAY { role: "Josef Tkaloun" }]->(a4)
(m2)-[c3:PLAY { role: "Ondřej" }]->(a1)
(m2)-[c4:PLAY { role: "Jakub" }]->(a2)
(m2)-[c5:PLAY { role: "Hanka" }]->(a3)
(m3)-[c6:PLAY { role: "Ivan" }]->(a1)
(m3)-[c7:PLAY { role: "Jirka", award: "Czech Lion" }]->(a2)
```

# Neo4j Interfaces

## Database architecture

- Client-server
- **Embedded database**
  - Directly integrated within your application

## Neo4j drivers

- Official: Java, .NET, JavaScript, Python
- Community: C, C++, PHP, Ruby, Perl, R, ...

## Neo4j shell

- Interactive command-line tool

## Query patterns

- **Cypher** – declarative graph query language
- **Traversal framework**

# Traversal Framework

# Traversal Framework

## Traversal framework

- Allows us to express and execute graph traversal queries
- Based on callbacks, executed lazily

## Traversal description

- **Defines rules and other characteristics of a traversal**

## Traverser

- Initiates and **manages a particular graph traversal** according to...
  - the provided traversal description, and
  - graph node / set of nodes where the traversal starts
- Allows for the **iteration over the matching paths**, one by one

# Traversal Framework: Example

Find actors who played in *Medvídek* movie

```
TraversalDescription td = db.traversalDescription()
    .breadthFirst()
    .relationships(Types.PLAY, Direction.OUTGOING)
    .evaluator(Evaluators.atDepth(1));

Node s = db.findNode(Label.label("MOVIE"), "id", "medvidek");
Traverser t = td.traverse(s);

for (Path p : t) {
    Node n = p.endNode();
    System.out.println(
        n.getProperty("name")
    );
}
```

Ivan Trojan  
Jiří Macháček

# Traversal Description

## Components of a **traversal description**

- **Order**
  - Which graph traversal algorithm should be used
- **Expanders**
  - What relationships should be considered
- **Uniqueness**
  - Whether nodes / relationships can be visited repeatedly
- **Evaluators**
  - When the traversal should be terminated
  - What should be included in the query result

# Traversal Description: Order

## Order

*Which graph traversal algorithm should be used?*

- Standard **depth-first** or **breadth-first** methods can be selected or specific branch ordering policies can also be implemented
- Usage:  
td.**breadthFirst**()  
td.**depthFirst**()



# Traversal Description: Expanders

## Path expanders

*Being at a given node...*

*what relationships should next be followed?*

- **Expander specifies one allowed...**
  - relationship **type** and **direction**
    - Direction.**INCOMING**
    - Direction.**OUTGOING**
    - Direction.**BOTH**
- Multiple expanders can be specified at once
  - When none is provided,  
then all the relationships are permitted
- Usage:  
td.**relationships**(type, direction)

# Traversal Description: Uniqueness

## Uniqueness

*Can particular nodes / relationships be revisited?*

- Various **uniqueness levels** are provided
  - Uniqueness.**NONE** – no filter is applied
  - Uniqueness.**RELATIONSHIP\_PATH**  
Uniqueness.**NODE\_PATH**
    - Nodes / relationships within a current path must be distinct
  - Uniqueness.**RELATIONSHIP\_GLOBAL**  
Uniqueness.**NODE\_GLOBAL (default)**
    - No node / relationship may be visited more than once
- Usage:  
td.**uniqueness**(level)

# Traversal Description: Evaluators

## Evaluators

*Considering a particular path...*

*should this path be included in the result?*

*should the traversal further continue?*

- Available **evaluation actions**
  - Evaluation.**INCLUDE\_AND\_CONTINUE**
  - Evaluation.**INCLUDE\_AND\_PRUNE**
  - Evaluation.**EXCLUDE\_AND\_CONTINUE**
  - Evaluation.**EXCLUDE\_AND\_PRUNE**
- Meaning of these actions
  - INCLUDE / EXCLUDE = whether to include the path in the result
  - CONTINUE / PRUNE = whether to continue the traversal

# Traversal Description: Evaluators

## Predefined evaluators

- Evaluators.`all()`
  - Never prunes, includes everything
- Evaluators.`excludeStartPosition()`
  - Never prunes, includes everything except the starting nodes
- Evaluators.`atDepth(depth)`  
Evaluators.`toDepth(maxDepth)`  
Evaluators.`fromDepth(minDepth)`  
Evaluators.`includingDepths(minDepth, maxDepth)`
  - Includes only positions within the specified interval of depths
- ...

# Traversal Description: Evaluators

## Evaluators

- Usage:  
td.**evaluator**(evaluator)
- Note that evaluators are **applied even for the starting nodes!**
- When **multiple evaluators** are provided...
  - then they must all agree on both the questions
- When **no evaluator** is provided...
  - then the traversal never prunes and includes everything

# Traverser

## Traverser

- Allows us to perform a particular graph traversal
  - with respect to a given traversal description
  - starting at a given node / nodes
- Usage: `t = td.traverse(node, ...)`
  - for (`Path p : t`) { ... }
    - Iterates over all the paths
  - for (`Node n : t.nodes()`) { ... }
    - Iterates over all the paths, returns their end nodes
  - for (`Relationship r : t.relationships()`) { ... }
    - Iterates over all the paths, returns their last relationships

## Path

- Well-formed **sequence of interleaved nodes and relationships**

# Traversal Framework: Example

Find actors who played with *Zdeněk Svěrák*

```
TraversalDescription td = db.traversalDescription()
    .depthFirst()
    .uniqueness(Uniqueness.NODE_GLOBAL)
    .relationships(Types.PLAY)
    .evaluator(Evaluators.atDepth(2))
    .evaluator(Evaluators.excludeStartPosition());

Node s = db.findNode(Label.label("ACTOR"), "id", "sverak");
Traverser t = td.traverse(s);

for (Node n : t.nodes()) {
    System.out.println(
        n.getProperty("name")
    );
}
```

Jiří Macháček

**Cypher**



# Cypher

## Cypher

- Declarative **graph query language**
  - Allows for expressive and efficient querying and updates
  - Inspired by SQL (query clauses) and SPARQL (pattern matching)
- **OpenCypher**
  - Ongoing project aiming at Cypher standardization
  - <http://www.opencypher.org/>

## Clauses

- E.g. MATCH, RETURN, CREATE, ...
- Clauses can be (almost arbitrarily) **chained together**
  - Intermediate result of one clause is passed to a subsequent one

# Sample Query

Find names of actors who played in *Medvídek* movie

```
MATCH (m:MOVIE)-[r:PLAY]->(a:ACTOR)
  WHERE m.title = "Medvídek"
RETURN a.name, a.year
ORDER BY a.year
```

a.name	a.year
Ivan Trojan	1964
Jiří Macháček	1966

# Clauses

## **Read clauses** and their sub-clauses

- MATCH – specifies graph patterns to be searched for
  - WHERE – adds additional filtering constraints
- ...

## **Write clauses** and their sub-clauses

- CREATE – creates new nodes or relationships
- DELETE – deletes nodes or relationships
- SET – updates labels or properties
- REMOVE – removes labels or properties
- ...

# Clauses

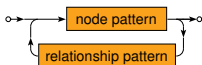
## General clauses and their sub-clauses

- RETURN – defines what the query result should contain
  - ORDER BY – describes how the query result should be ordered
  - SKIP – excludes certain number of solutions from the result
  - LIMIT – limits the number of solutions to be included
- WITH – allows query parts to be chained together
- ...

# Path Patterns

## Path pattern expression

- **Sequence of interleaved node and relationship patterns**
- Describes a single path (not a general subgraph)

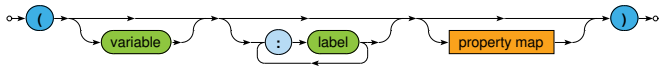


- ASCII-Art inspired syntax
  - Circles ( ) for nodes
  - Arrows <-- , -- , --> for relationships

# Path Patterns

## Node pattern

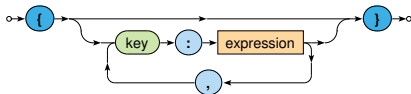
- Matches one data node



- **Variable**
  - Allows us to access a given node later on
- Set of **labels**
  - Data node must have **all the specified labels** to be matched
- **Property map**
  - Data node must have **all the requested properties** (including their values) to be matched (the order is unimportant)

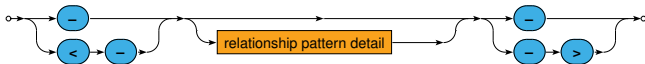
# Path Patterns

## Property map



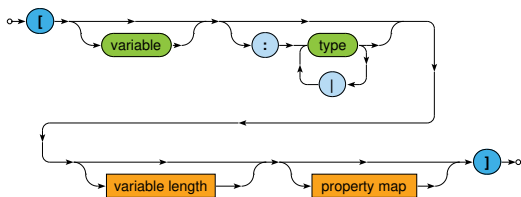
## Relationship pattern

- Matches one data relationship



# Path Patterns

## Relationship pattern



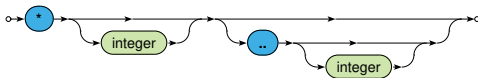
- **Variable**
  - Allows us to access a given node later on
- Set of **types**
  - Data relationship must be of **one of the enumerated types** to be matched



# Path Patterns

## Relationship pattern (*cont.*)

- **Property map**
  - Data relationship must have **all the requested properties**
- Variable path **length**
  - Allows us to match **paths of arbitrary lengths** (not just exactly one relationship)



- Examples: `*`, `*4`, `*2..6`, `*..6`, `*2..`

# Path Patterns

## Examples

```
()
```

```
(x)--(y)
```

```
(m:MOVIE)-->(a:ACTOR)
```

```
(:MOVIE)-->(a { name: "Ivan Trojan" })
```

```
()<-[r:PLAY]-()
```

```
(m)-[:PLAY { role: "Ivan" }]->()
```

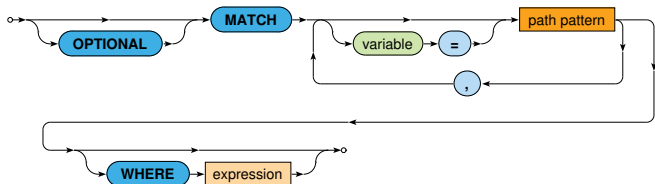
```
(:ACTOR { name: "Ivan Trojan" })-[:KNOW *2]->(:ACTOR)
```

```
()-[:KNOW *5..]->(f)
```

# Match Clause

## MATCH clause

- Allows to search for **sub-graphs of the data graph** that match the provided path pattern / patterns (all of them)
  - **Query result** (table) = unordered **set of solutions**
  - One solution (row) = set of **variable bindings**
- Each variable has to be bound



# Match Clause

**WHERE sub-clause** may provide additional constraints

- These constraints are **evaluated directly during the matching phase** (i.e. not after it)
- Typical usage
  - Boolean expressions
  - Comparisons
  - Path patterns – true if at least one solution is found
  - ...

# Match Clause: Example

Find names of actors who played with *Ivan Trojan* in any movie

```
MATCH (i:ACTOR)-[:PLAY]-(m:MOVIE)-[:PLAY]->(a:ACTOR)
  WHERE (i.name = "Ivan Trojan")
RETURN a.name
```

```
MATCH (i:ACTOR { name: "Ivan Trojan" })
  <-[:PLAY]-(m:MOVIE)-[:PLAY]->
  (a:ACTOR)
RETURN a.name
```

i	m	a		a.name
(a1)	(m2)	(a2)	⇒	Jiří Macháček
(a1)	(m2)	(a3)		Jitka Schneiderová
(a1)	(m3)	(a2)		Jiří Macháček

# Match Clause

## Uniqueness requirement

- One data node may match several query nodes, but one data relationship may not match several query relationships

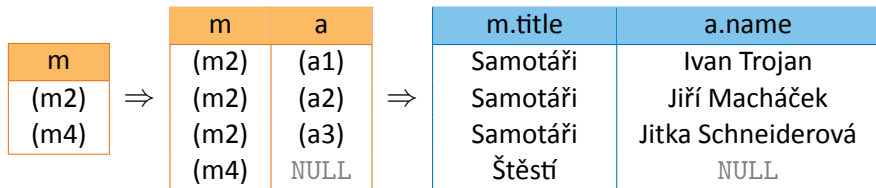
## OPTIONAL MATCH

- Attempts to find matching data sub-graphs as usual...
- but **when no solution is found**, one specific solution with **all the variables bound to NULL** is generated
- Note that either the whole pattern is matched, or nothing is matched

# Match Clause: Example

Find movies filmed in 2005 or earlier and names of their actors (if any)

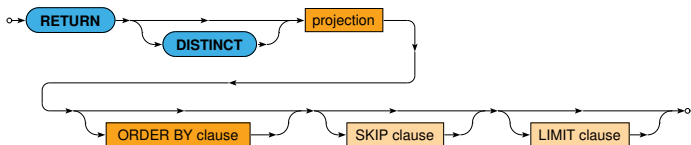
```
MATCH (m:MOVIE)
  WHERE (m.year <= 2005)
OPTIONAL MATCH (m)-[:PLAY]->(a:ACTOR)
RETURN m.title, a.name
```



# Return Clause

## RETURN clause

- Defines what to include in the query result
  - Projection of variables, properties of nodes or relationships (via dot notation), aggregation functions, ...
- Optional ORDER BY, SKIP and LIMIT sub-clauses



## RETURN DISTINCT

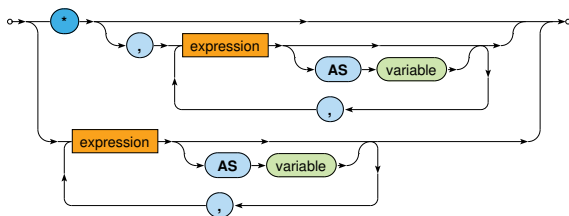
- Duplicate solutions (rows) are removed



# Return Clause

## Projection

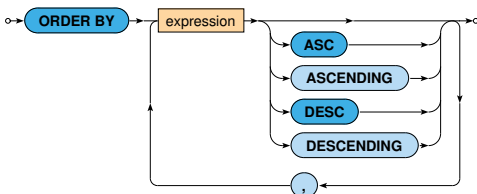
- \* = **all the variables**
  - Can only be specified as the very first item
- AS allows to **explicitly (re)name** output records



# Return Clause

## ORDER BY sub-clause

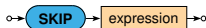
- Defines the **order of solutions** within the query result
  - Multiple criteria can be specified
  - Default direction is ASC
- The order is undefined unless explicitly defined
- Nodes and relationships as such cannot be used as criteria



# Return Clause

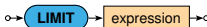
## SKIP sub-clause

- Determines the **number of solutions to be skipped** in the query result



## LIMIT sub-clause

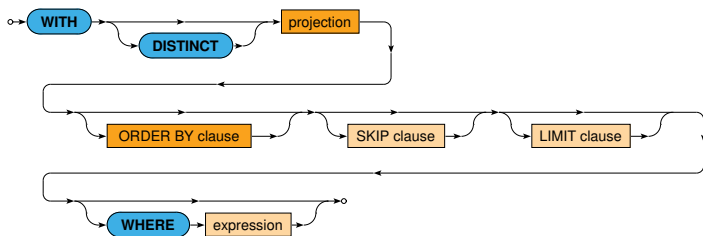
- Determines the **number of solutions to be included** in the query result



# With Clause

## WITH clause

- **Constructs intermediate result**
  - Analogous behavior to the RETURN clause
  - Does not output anything to the user, just **forwards the current result to the subsequent clause**
- Optional WHERE sub-clause can also be provided



# With Clause: Example

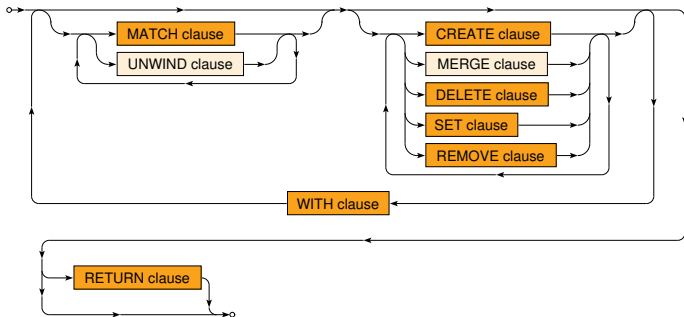
Numbers of movies in which actors born in 1965 or later played

```
MATCH (a:ACTOR)
  WHERE (a.year >= 1965)
WITH a, SIZE( (a)-[:PLAY]-(m:MOVIE) ) AS movies
RETURN a.name, movies
ORDER BY movies ASC
```

a		a	movies		a.name	movies
(a2)	⇒	(a2)	3	⇒	Jitka Schneiderová	1
(a3)		(a3)	1		Jiří Macháček	3

# Query Structure

## Chaining of Cypher clauses (*simplified*)



- **Read** clauses: MATCH, ...
- **Write** clauses: CREATE, DELETE, SET, REMOVE, ...

# Query Structure

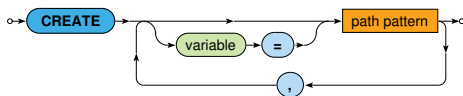
## Query parts

- **WITH clauses split the whole query into query parts**
- Certain restrictions apply...
  - **Read clauses (if any) must precede write clauses (if any)** in every query part
  - **The last query part must be terminated by a RETURN clause**
    - Unless this part contains at least one write clause
    - I.e. **read-only queries must return data**
  - ...

# Write Clauses

## CREATE clause

- Inserts new nodes or relationships into the data graph



## Example

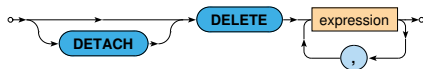
```
MATCH (m:MOVIE { id: "stesti"})
CREATE
  (a:ACTOR { id: "vilhelmova", name: "Tatiana Vilhelmová", year: 1978}),
  (m)-[:PLAY]->(a)
```



# Write Clauses

## DELETE clause

- **Removes nodes, relationships or paths** from the data graph
- Relationships must always be removed before the nodes they are associated with
  - Unless the DETACH modifier is specified



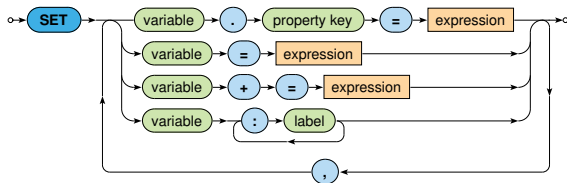
## Example

```
MATCH (:MOVIE { id: "stesti"})-[r:PLAY]->(a:ACTOR)
DELETE r
```

# Write Clauses

## SET clause

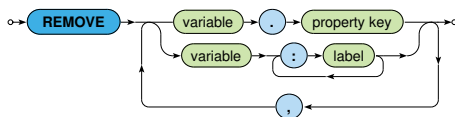
- Allows to...
  - **set a value of a particular property**
    - or remove a property when NULL is assigned
  - **replace properties** (all of them) with new ones
  - **add new properties** to the existing ones
  - **add labels** to nodes
- Cannot be used to set relationship types



# Write Clauses

## REMOVE clause

- Allows to...
  - **remove a particular property**
  - **remove labels** from nodes
- Cannot be used to remove relationship types



# Expressions

## Literal expressions

- Integers: decimal, octal, hexadecimal
- Floating-point numbers
- Strings
  - Enclosed in double or single quotes
  - Standard escape sequences
- Boolean values: `true`, `false`
- NULL value (cannot be stored in data graphs)

## Other **expressions**

- Collections, variables, property accessors, function calls, path patterns, boolean expressions, arithmetic expressions, comparisons, regular expressions, predicates, ...



# Lecture Conclusion

**Neo4j** = graph database

- **Property graphs**
- **Traversal framework**
  - Path expanders, uniqueness, evaluators, traverser

**Cypher** = graph query language

- Read (sub-)clauses: MATCH, WHERE, ...
- Write (sub-)clauses: CREATE, DELETE, SET, REMOVE, ...
- General (sub-)clauses: RETURN, WITH, ORDER BY, LIMIT, ...