NDBI050: Query Languages II

http://www.ksi.mff.cuni.cz/~svoboda/courses/232-NDBI050/

Lecture

Neo4j

Martin Svoboda martin.svoboda@matfyz.cuni.cz

21. 5. 2024

Charles University, Faculty of Mathematics and Physics

Lecture Outline

Graph databases

Introduction

Neo4j

- Data model: property graphs
- 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, high availability, fault-tolerant, master-slave replication, ACID transactions, ...
- Developed by Neo Technology
- Implemented in Java
- Operating systems: cross-platform
- Initial release in 2007
 - Version we cover is 4.4.10 (August 2022)

Data Model

Dataspace structure

```
\boxed{ \mathsf{Instance} \left( \to \mathsf{database} \right) \to \mathsf{single} \ \mathsf{graph} }
```

Property graph = directed labeled multigraph

Collection of nodes (vertices) and relationships (edges)

Node

- Unique identity
 - Internal, should not be used directly
- Set of labels (zero or more)
 - Allow for node categorization via user-defined types
 - E.g.: ACTOR, MOVIE, ...
- Property map = set of individual properties
 - Allow to associate a given node with additional data

Data Model

Relationship

- Unique identity
 - Once again, internal only
- Direction (immutable and compulsory)
 - Relationships are traversable in both directions
 - There is no impact on efficiency
 - Directions can also be entirely ignored when querying
- Start node and end node
 - Can be the same, i.e., loops are allowed as well
- Exactly <u>one</u> type (<u>immutable</u>)
 - E.g.: PLAY, ...
- Property map

Data Types

Structural types

- Node, Relationship
- Path = sequence of interleaved nodes and relationships

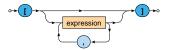
Property types

- String (e.g., "Samotáři")
 - Sequence of Unicode characters
 - Enclosed preferably by double quotes
 - Standard backslash escaping (\", \n, \\, ...)
- Integer (e.g., 165, 0xA5, 0o245)
- Float
- Boolean (literals true and false)
- ...

Data Types

Composite types

- List = ordered collection of values
 - Values can be anything
 - I.e., all types are permitted (property, structural, composite)
 - And so lists can contain other embedded lists, maps, ...
 - Lists can be heterogeneous

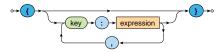


• E.g.: [2015, "Samotáři", [2020], 2015]

Data Types

Composite types (cont'd)

- Map = unordered collection of key-value pairs
 - Key is a string, unique in a given map
 - Value can be anything, again
 - Maps can be heterogeneous, too



• E.g.: { title: "Samotáři", genres: ["comedy"] }

Property map = restricted version of a general map

- Used for node and relationship property maps
 - Only atomic values or <u>homogeneous</u> arrays of atomic values of any property type are permitted for top-level properties

Sample Data

Sample graph with movies and actors

```
CREATE
 // Movies
  (m1:MOVIE { id: "vratnelahve", title: "Vratné lahve", year: 2006,
              rating: 76, language: "cs", genres: [ "comedv" ] }).
  (m2:MOVIE { id: "samotari", title: "Samotáři", year: 2000,
              rating: 84, language: "cs", genres: [ "comedy", "drama" ] }).
  (m3:MOVIE { id: "medvidek", title: "Medvidek", vear: 2007.
              rating: 53, language: "cs", genres: [ "comedy", "drama" ] }),
  (m4:MOVIE { id: "stesti", title: "Štěstí", year: 2005,
              rating: 72, language: "cs", genres: [ "drama" ] }),
 // Actors
  (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", vear: 1936 }).
  (a5:ACTOR { id: "vilhelmova", name: "Tatiana Vilhelmová", year: 1978 }),
```

Sample Data

Sample graph with movies and actors (cont'd)

```
// Vratné lahve --> Jiří Macháček
(m1)-[p1:PLAY { role: "Robert Landa" }]->(a2),
// Vratné lahve --> Zdeněk Svěrák
(m1)-[p2:PLAY { role: "Josef Tkaloun" }]->(a4),
// Samotáři --> Ivan Trojan
(m2)-[p3:PLAY { role: "Ondřej" }]->(a1),
// Samotáři --> Jiří Macháček
(m2)-[p4:PLAY { role: "Jakub" }]->(a2),
// Samotáři --> Jitka Schneiderová
(m2)-[p5:PLAY { role: "Hanka" }]->(a3),
// Medvídek --> Ivan Trojan
(m3)-[p6:PLAY { role: "Ivan" }]->(a1),
// Medvídek --> Jiří Macháček
(m3)-[p7:PLAY { role: "Jirka" }]->(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, ...

Cypher shell

Interactive command-line tool

Query patterns

Cypher query language, Traversal framework

Cypher

Cypher

Cypher

- Declarative graph query language
 - Allows for expressive and efficient querying and updates
- Based on sub-graph pattern matching, similarly as SPARQL
 - Patterns are expressed using ASCII-Art inspired syntax
 - Circles () for nodes
 - Arrows <---, ---> for relationships
- Each query is evaluated to a solution sequence (table)
 - Ordered collection of individual solutions (matching subgraphs)

Chaining of clauses

- Not only individual clauses can be used repeatedly...
- ... they can also be (almost arbitrarily) chained together
 - Intermediate result of one clause is passed to the following one

Sample Query

Names of actors who played in Medvidek movie

```
MATCH (m:MOVIE)-[:PLAY]->(a:ACTOR)

WHERE m.title = "Medvidek"

RETURN a.name, a.year

ORDER BY a.year
```

m	a		a.name	a.year
(medvidek)	(trojan)	\rightarrow	Ivan Trojan	1964
(medvidek)	(machacek)		Jiří Macháček	1966

Clauses and Subclauses

Read clauses

- MATCH describes graph pattern to be searched for
 - WHERE adds additional filtering constraints

Write clauses

- CREATE, DELETE, SET, REMOVE
 - Creates / deletes nodes / relationships / labels / properties

General clauses

- RETURN defines what the query result should contain
 - ORDER BY, SKIP, and LIMIT subclauses
- WITH constructs auxiliary intermediate query result
 - ORDER BY, SKIP, LIMIT, and also WHERE

Path Patterns

Path Patterns

Node pattern

- Describes one data node and conditions it must satisfy
 - E.g.: (), (:ACTOR { name: "Ivan Trojan" }),...

Relationship pattern

- Describes one data relationship and conditions it must satisfy
 - E.g.: ()--(), (:MOVIE)-[:PLAY]->(:ACTOR), ...

Path pattern

- Describes one data path to be found
 - Via a sequence of interleaved node and relationship patterns



Node Patterns

Node pattern

- Describes one data node to be matched
 - When inner conditions are provided, they must <u>all</u> be satisfied



- Variable
 - Makes a given node accessible in subsequent query fragments
 - I.e., for selection in WHERE conditions, projection in RETURN clauses, alignment with other node patterns, ...
 - Such a thing would otherwise be impossible
 - E.g.: (m)

Node Patterns

Node pattern (cont'd)

- Labels condition
 - Set of zero or more labels can be provided
 - Data node to be matched then...
 - Must have at least <u>all</u> the specified labels
 - I.e., there may also be other, but these are compulsory
 - E.g.: (m:MOVIE)
- Property map condition
 - Data node to be matched...
 - Must have at least all the specified properties
 - I.e., they are present and have identical values
 - Note that mutual order of such properties is unimportant
 - E.g.: (m:MOVIE { title: "Medvidek", year: 2007 })

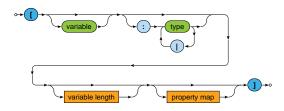
Relationship pattern

- Describes one data relationship to be matched
 - When the direction condition is provided, it must be satisfied
 - When inner conditions are given, they must also be satisfied



- Direction condition
 - Data relationship to be matched...
 - Must have the same direction
 - I.e., --> for outgoing direction or <-- for incoming
 - When -- is used, direction is ignored

Relationship pattern (cont'd)



- Variable
 - Allows us to access a given relationship later on
 - E.g.: ()-[r]->()

Relationship pattern (cont'd)

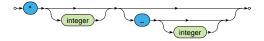
- Type condition
 - Set of zero or more types can be provided
 - Data relationship to be matched then...
 - Must have one of the enumerated types

```
E.g.: ()-[r:PLAY]->()
```

- Property map condition
 - Data relationship to be matched...
 - Must have at least all the specified properties
 - E.g.: ()-[r:PLAY { role: "Jakub" }]->()

Relationship pattern (cont'd)

- Variable length mode
 - When activated, paths of arbitrary lengths can be found
 - Otherwise (i.e., by default), one relationship pattern will be matched by exactly one data relationship
 - Length condition ranges: *, *4, *2..6, *..6, *2..



- <u>Each</u> data relationship on the path must...
 - Satisfy all the involved conditions (direction, type, properties)
- E.g.: ()-[r:FRIEND *..2]-()
 - If variable is introduced, it then references the whole <u>path</u>

Graph Patterns

Relationship uniqueness requirement

- One data node may match multiple node patterns at once
 - E.g.: (a)-[:FRIEND]-()-[:FRIEND]-(b)
 - It may happen that both a and b will actually be the same node
 - However, only when distinct data relationships were used...
- I.e., one data relationship <u>cannot</u> be matched repeatedly

Node pattern alignment

- Intentional alignment of nodes (not relationships) is possible
 - Simply by using the same shared variables
- E.g.: (a)-[:FRIEND]-()-[:FRIEND]-(a)

General graph pattern

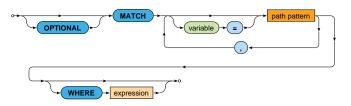
- Graphs can be decomposed into individual path patterns
 - Uniqueness requirement / shared variables work the same way

Read Clauses

Match Clause

MATCH clause

- Allows to search for sub-graphs of the data graph that match the provided graph pattern
 - Solution sequence is produced, each variable has to be bound
- Natural join is used on the previous result when chaining



- WHERE condition
 - Only solutions satisfying a given condition are preserved

Match Clause: Example

Names of actors who played with Ivan Trojan in any movie

- Notice that Ivan Trojan himself is not included in the result
 - Because of the uniqueness requirement

```
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
(trojan)	(samotari)	(machacek)	"Jiří Macháček"
(trojan)	(samotari)	(schneiderova)	"Jitka Schneiderová"
(trojan)	(medvidek)	(machacek)	"Jiří Macháček"

Match Clause: Example

Names of actors who played with *Ivan Trojan* in any movie (cont'd)

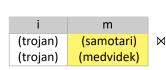
- Uniqueness requirement is <u>not</u> applied across clauses
 - And so internal identities must be used to exclude Ivan Trojan

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

MATCH (m:MOVIE)-[:PLAY]->(a:ACTOR)

WHERE (i <> a)

RETURN a.name
```



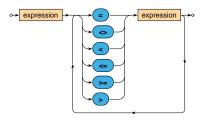
m	i		
(vratnelahve)	(machacek)		
(vratnelahve)	(sverak)		
(samotari)	(trojan)		
(samotari)	(machacek)		
(samotari)	(schneiderova)		
(medvidek)	(trojan)		
(medvidek)	(machacek)		

WHERE subclause conditions

- Only solutions satisfying a given condition are preserved
 - Evaluated directly during the matching phase (<u>not</u> after it)
- Possible conditions
 - Comparisons
 - NULL testing predicate
 - IN predicate
 - Path patterns
 - Existential subqueries
 - Quantifiers
 - Boolean expressions
 - ٠..

Comparison conditions

- Traditional comparison operators are available
 - Chained comparisons can be created, too



- E.g.: 2015 <= m.year < 2020
 - Equivalent to 2015 <= m.year AND m.year < 2020</pre>

NULL testing conditions

- Three-valued logic is assumed
 - Traditional true and false values
 - But also null representing the third unknown value
- Indirect testing is thus necessary



IN predicate conditions

- Allow for both fixed enumerations as well as arbitrary lists
- E.g.: m.language IN ["cs", "sk"]
- E.g.: "comedy" IN m.genres

String matching conditions

STARTS WITH / CONTAINS / ENDS WITH operators



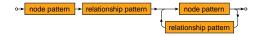
- E.g.: m.title ENDS WITH "Bobule"
 - Matches Bobule, 2Bobule, ...

Regular expression conditions

- Special operator =~ is used for this purpose
- E.g.: m.title =~ ".*Bobule"

Path pattern predicate conditions

- Path pattern can directly be used as a condition
 - At least one relationship pattern is required, though

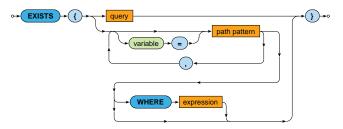


- Satisfied if and only if a non-empty result is yielded
- E.g.: (m)-[:PLAY]->(:ACTOR)
 - Ensures the existence of at least one actor for an already resolved movie node m

Search Conditions

Existential subquery conditions

- Subquery with top-level query expressive power
- Or standard graph pattern with optional filtering



- Satisfied if and only if a non-empty result is yielded
- E.g.: EXISTS { (m)-[:PLAY]->(a:ACTOR) WHERE a.name = "Ivan Trojan" }

Search Conditions

Quantifier conditions

Allow to simulate quantifiers and their derivatives



- Satisfied if and only if...
 - ALL: all items satisfy a given condition
 - NONE: no item satisfies a given condition
 - ANY: at least one item satisfies a given condition
 - SINGLE: exactly one item satisfies a given condition
- E.g.: ANY (g IN m.genres WHERE g = "comedy")

Search Conditions

Logical conditions

- Standard logical connectives are available
 - AND (conjunction)
 - OR (disjunction)
 - NOT (negation)

Match Clause

OPTIONAL mode of MATCH clauses

- Optionally attempts to find matching data sub-graphs...
 - When not possible, one solution with all variables bound to null is generated
- Left outer natural join is used when chaining

Match Clause: Example

Movies from 2005 or earlier, optionally their actors born after 1965

```
MATCH (m:MOVIE)
WHERE (m.year <= 2005)
OPTIONAL MATCH (m)-[:PLAY]->(a:ACTOR)
WHERE (a.year > 1965)
RETURN m.title, a.name
```

		m	i	
m		(vratnelahve)	(machacek)	
(samotari)	\bowtie	(samotari)	(machacek)	=
(stesti)		(samotari)	(schneiderova)	
	-	(medvidek)	(machacek)	

m	a
(samotari)	(machacek)
(samotari)	(schneiderova)
(stesti)	null

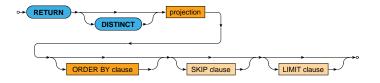
m.title	a.name
"Samotáři"	"Jiří Macháček"
"Samotáři"	"Jitka Schneiderová"
"Štěstí"	null

General Clauses

Return Clause

RETURN clause

- Defines the final query result to be returned to the user
 - Can only be provided as the very last clause in the chain

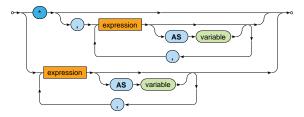


- DISTINCT modifier: removes duplicate solutions
- ORDER BY subclause
- SKIP and LIMIT subclauses: pagination of solutions

Return Clause

Projection

- Enumeration of columns to appear in the result
 - Variables for nodes, relationships, or even paths
 - Properties via the dot notation
 - Arithmetic expressions, aggregate functions, ...
- Wildcard * = all the existing columns
 - Can only be specified as the very first item



Return Clause: Example

Actors born in 1965 or later and numbers of movies they played in

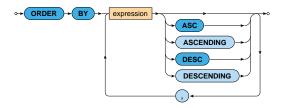
```
MATCH (a:ACTOR)
WHERE (a.year >= 1965)
RETURN a.name, SIZE([ (a)<-[:PLAY]-(m:MOVIE) | m ]) AS count
ORDER BY count DESC
```

a		a.name	count
(schneiderova)	\rightarrow	"Jiří Macháček"	3
(machacek)		"Jitka Schneiderová"	1
(vilhelmova)		"Tatiana Vilhelmová"	0

Solution Modifiers

ORDER BY subclause

- Defines the order of solutions within the query result
 - Multiple criteria can be specified
 - Nodes, relationships, nor paths cannot be used for this purpose
 - The order is undefined unless explicitly defined
- Default direction is ASC



Solution Modifiers

Pagination

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

```
→ SKIP → expression → ○
```

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



Grouping and Aggregation

Traditional grouping is supported, too

- Works exactly as in the relational databases
 - However, there are no specific GROUP BY or HAVING clauses
- Happens <u>automatically</u>...
 - When at least one aggregate function is called in projection
- All columns are then divided into two disjoint types...
 - Aggregating colums
 - All the columns calling an aggregate function
 - E.g.: COUNT, SUM, MIN, MAX, AVG, COLLECT, ...
 - Grouping columns
 - All the remaining ones
 - They all become the classification columns
 - And so they determine the individual groups to be created

Grouping and Aggregation: Example

Actors born in 1965 or later and movies they played in

```
MATCH (a:ACTOR)<-[:PLAY]-(m:MOVIE)

WHERE (a.year >= 1965)

RETURN a.name, COUNT(m) AS count, COLLECT(m.title) AS movies
```

а	m
(machacek)	(vratnelahve)
(machacek)	(samotari)
(machacek)	(medvidek)
(schneiderova)	(samotari)

	a.name	
\rightarrow	"Jiří Macháček"	 \rightarrow
	"Jitka Schneiderová"	

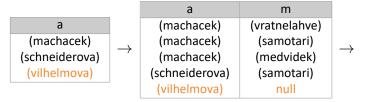
a.name	count	movies
"Jiří Macháček"	3	["Vratné lahve", "Samotáři", "Medvídek"]
"Jitka Schneiderová"	1	["Samotáři"]

- Note that Tatiana Vilhelmová will not be included
 - Since she did not play in any movie

Grouping and Aggregation: Example

Actors born in 1965 or later and movies they played in (cont'd)

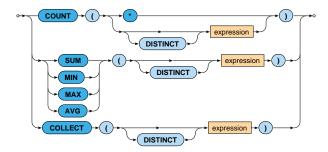
```
MATCH (a:ACTOR)
WHERE (a.year >= 1965)
OPTIONAL MATCH (a)<-[:PLAY]-(m:MOVIE)
RETURN a.name, COUNT(m) AS count, COLLECT(m.title) AS movies
```



a.name	count	movies
"Jiří Macháček"	3	["Vratné lahve", "Samotáři", "Medvídek"]
"Jitka Schneiderová"	1	["Samotáři"]
"Tatiana Vilhelmová"	0	[]

Grouping and Aggregation

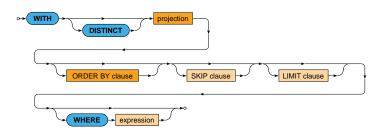
Aggregate functions



With Clause

WITH clause

- Constructs another intermediate result in the chain
 - Analogous behavior to the RETURN clause
 - Except that no output is sent to the user
 - Optional WHERE subclause can also be provided



With Clause: Example

Movies with above average number of actors and rating at least 75

```
MATCH (m:MOVIE)
WITH m, SIZE([(m)-[:PLAY]->(a:ACTOR)|a]) AS actors
WITH AVG(actors) AS average
MATCH (m:MOVIE)
WHERE (SIZE([(m)-[:PLAY]->(a:ACTOR)|a]) > average) AND (m.rating >= 75)
RETURN m.title, m.rating
```

m		m	actors			
(vratnelahve)		(vratnelahve)	2		avorago	1
(samotari)	\rightarrow	(samotari)	3	\rightarrow	average	\rightarrow
(medvidek)		(medvidek)	2		1.75	
(stesti)		(stesti)	0			

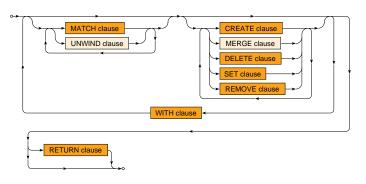
average	m		m.title	m.rating
1.75	(vratnelahve)	\rightarrow	"Vratné lahve"	76
1.75	(samotari)		"Samotáři"	84

Query Structure

Query Structure

Chaining of clauses

Certain rules must be followed when chaining the clauses...



Query Structure

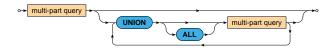
Query parts

- WITH clauses split the whole query into query parts
- · Within each query part...
 - Read clauses (if any) must precede write clauses (if any)
 - Read clauses: MATCH, ...
 - Write clauses: CREATE, DELETE, SET, REMOVE, ...
- As for the very <u>last</u> query part...
 - It must be terminated by RETURN clause
 - Unless this part contains at least one write clause
 - I.e., read-only queries must return data

Union Operation

UNION operation

- Combines results yielded by two or more multi-part queries
 - Standard union set operation is assumed



- Schemas of all involved results must be identical
 - I.e., the same number and the same names of columns
- Duplicates are automatically removed
 - Unless the ALL keyword is provided

Map and List Operations

Property lookup operator

- Allows to access a particular property of a given map
 - Static lookup: m.genres
 - Dot notation is used, for fixed property keys only
 - Dynamic lookup: m ["genres"]

List subscript operator

- Allows to access a particular list item based on its index
 - Position of the first item is 0
 - Negative values are also permitted
 - For positions starting at the end and in the reverse direction
 - \circ | expression | \bullet | \bullet
- E.g.: m.genres[0], m.genres[-1] (the last genre)

List Operations

List slice operator

- Allows to retrieve an arbitrary range of a given list
 - Lower bound is inclusive, upper bound is exclusive
 - At least one bound needs to be specified
 - Negative numbers are allowed as well



Examples

```
■ range(1, 5) \rightarrow [1, 2, 3, 4, 5]

■ range(1, 5) [1..3] \rightarrow [2, 3]

■ range(1, 5) [..3] \rightarrow [1, 2, 3]

■ range(1, 5) [1..] \rightarrow [2, 3, 4, 5]

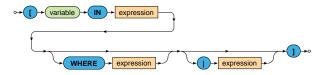
■ range(1, 5) [-3..-1] \rightarrow [3, 4]

■ range(1, 5) [3..-1] \rightarrow [4]
```

List Operations

List comprehension

- Creates a new list based on items of an existing list
 - Only items satisfying a given condition are considered
 - New output items can be constructed
 - Otherwise the original ones are returned intact

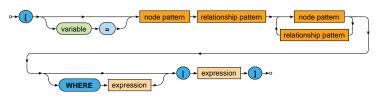


- Examples
 - [i IN range(1, 5) WHERE i % 2 = 0] \rightarrow [2, 4]
 - [i IN range(1, 5) WHERE i % 2 = 0 | i * 10] \rightarrow [20, 40]

List Operations

Pattern comprehension

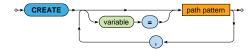
- Creates a new list based on solutions of a given path pattern
 - Only solutions satisfying a given condition are considered



- Example
 - [(m:MOVIE)-[:PLAY]->(a:ACTOR) WHERE (m.year >=
 2005) AND (a.name = "Jiří Macháček") | m.title]
 → ["Vratné lahve", "Medvídek"]

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)
```

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

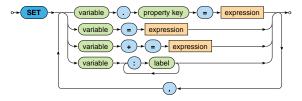


Example

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

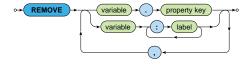
SET clause

- 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



REMOVE clause

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



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, ...