Graph Databases: Neo4j

Big Data Management

Faculty of Computer Science & Information Technology
University of Malaya

December 9, 2019

Borrowed from http://www.ksi.mff.cuni.cz/~svoboda/courses/171-NDBI040/

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

1. **Non-transactional.** small number of large graphs
2. **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)

Outline

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**
  1. Key is a **string**
  2. Value is an **atomic value** of any primitive data type, or an **array of atomic values** of one primitive data type

Primitive data types

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

Sample Data

Sample graph with movies and actors

```
(m1:MOVIE { id: "thedouble", title: "The double", year: 2006 })
(m2:MOVIE { id: "zombieland", title: "Zombieland", year: 2000 })
(m3:MOVIE { id: "thesocialnetwork", title: "The social network", year: 2007 })
(m4:MOVIE { id: "inception", title: "Inception", year: 2005 })

(a1:ACTOR { id: "harrelson", name: "Woody Harrelson", year: 1964 })
(a2:ACTOR { id: "eisenberg", name: "Jesse Eisenberg", year: 1966 })
(a3:ACTOR { id: "stone", name: "Emma Stone", year: 1973 })
(a4:ACTOR { id: "wasikowska", name: "Mia Wasikowska", year: 1936 })

(m1)-[c1:PLAY { role: "Simon James" }]->(a2)
(m1)-[c2:PLAY { role: "Hannah" }]->(a4)
(m2)-[c3:PLAY { role: "Tallahassee" }]->(a1)
(m2)-[c4:PLAY { role: "Columbus" }]->(a2)
(m2)-[c5:PLAY { role: "Wichita" }]->(a3)
(m3)-[c6:PLAY { role: "Eduardo Saverin" }]->(a1)
(m3)-[c7:PLAY { role: "Mark Zuckerberg", award: "Golden Globe" }]->(a2)
```

Neo4j Interfaces

Database architecture

1. Client-server
2. 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

1. **Cypher** – declarative graph query language
2. **Traversal framework**

Outline

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 The social network movie

```
1  TraversalDescription td = db.traversalDescription()
2    .breadthFirst()
3    .relationships(Types.PLAY, Direction.OUTGOING)
4    .evaluator(Evaluators.atDepth(1));
5
6  Node s = db.findNode(Label.label("MOVIE"), "id", "thesocialnetwork");
7  Traverser t = td.traverse(s);
8
9  for (Path p : t) {
10   Node n = p.endNode();
11   System.out.println(
12     n.getProperty("name")
13   );
14 }
```

```
1 Woody Harrelson
2 Jesse Eisenberg
```

Traversal Description

Components of a traversal description

1. **Order**
   - Which graph traversal **algorithm** should be **used**
2. **Expanders**
   - What **relationships** should be **considered**
3. **Uniqueness**
   - Whether **nodes/relationships** can be **visited repeatedly**
4. **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?
  1. **Standard** depth-first or breadth-first methods can be selected, or
  2. **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...
  1. should this path be **included** in the result?
  2. 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

1. **Evaluators.all()**
   - Never prunes, includes everything
2. **Evaluators.excludeStartPosition()**
   - Never prunes, includes everything except the starting nodes
3. **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!

1. When **multiple evaluators** are provided ...
   - then they must **all agree on** both the questions
2. When **no evaluator** is provided...
   - then the traversal **never prunes** and **includes everything**

Traverser

Traverser

- Allows to **perform** a particular graph traversal
  - with respect to a **given traversal description**
  - **starting** at a **given node/nodes**
- **Usage:** t = td.traverse(node, ...)
  1. for (Path p : t) ...
     - Iterates over **all the paths**
  2. for (Node n : t.nodes()) ...
     - Iterates over all the paths, returns their **end nodes**
  3. 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 Mia Wasikowska

```
1  TraversalDescription td = db.traversalDescription()
2    .depthFirst()
3    .uniqueness(Uniqueness.NODE_GLOBAL)
4    .relationships(Types.PLAY)
5    .evaluator(Evaluators.atDepth(2))
6    .evaluator(Evaluators.excludeStartPosition());
7
8  Node s = db.findNode(Label.label("ACTOR"), "id", "wasikowska");
9  Traverser t = td.traverse(s);
10
11 for (Node n : t.nodes()) {
12   System.out.println(
13     n.getProperty("name")
14   );
15 }
```

```
1  Jesse Eisenberg
```

Outline

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 The social network movie

```
1 MATCH (m:MOVIE)-[r:PLAY]->(a:ACTOR)
2   WHERE m.title = "The social network"
3 RETURN a.name, a.year
4   ORDER BY a.year
```

a.name	a.year
Woody Harrelson	1964
Jesse Eisenberg	1966

Clauses

Read clauses and their sub-clauses

1. **MATCH** – specifies graph patterns to be searched for
   - **WHERE** – adds additional filtering constraints
2. ...

Write clauses and their sub-clauses

1. **CREATE** – creates new nodes or relationships
2. **DELETE** – deletes nodes or relationships
3. **SET** – updates labels or properties
4. **REMOVE** – removes labels or properties
5. ...

Clauses

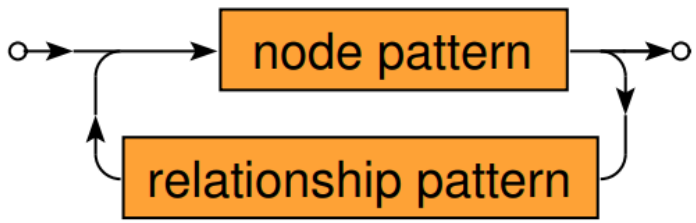
General clauses and their sub-clauses

1. **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
2. **WITH** – allows query parts to be chained together
3. ...

Path Patterns

Path pattern expression

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

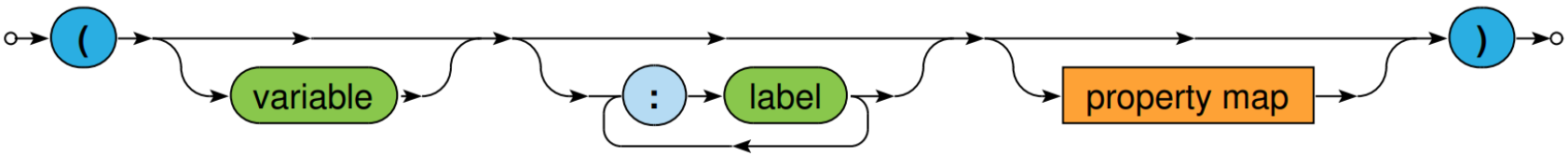


- ASCII-Art inspired syntax
  1. **Circles** () for nodes
  2. **Arrows** <--, --, --> for relationships

Path Patterns

Node pattern

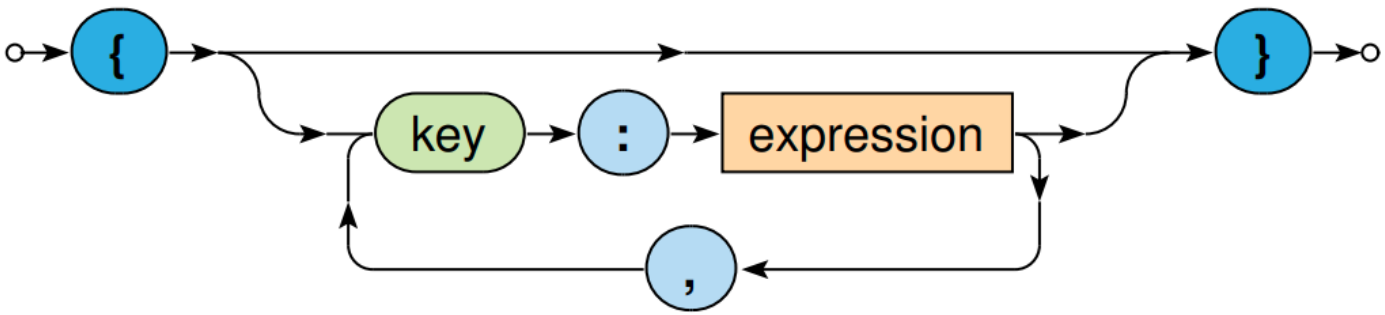
- Matches one data node



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

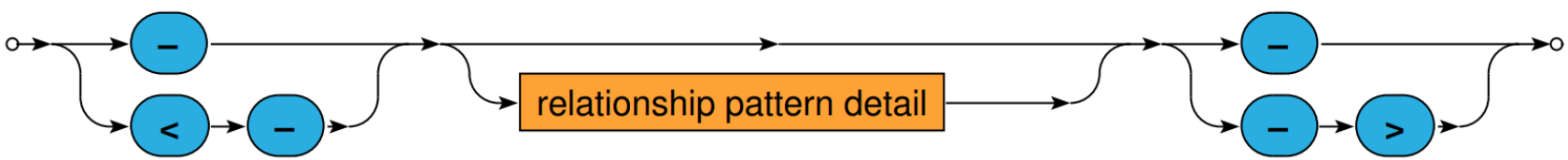
Path Patterns

Property map



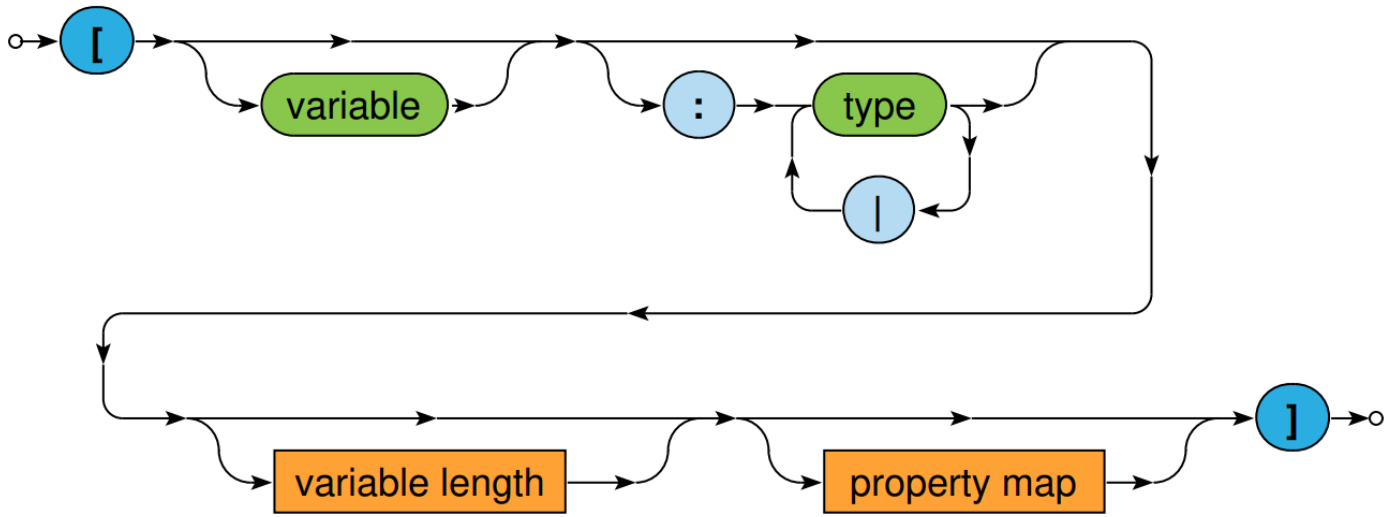
Relationship pattern

- Matches one data relationship



Path Patterns

Relationship pattern

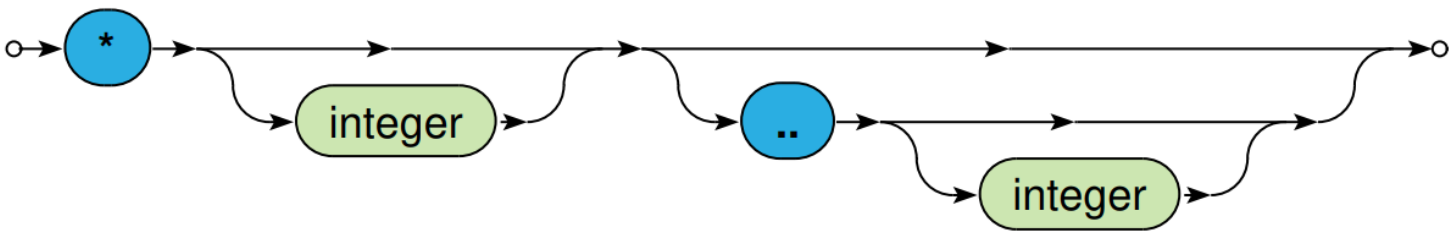


1. **Variable**
   - Allows us to **access** a given node later on
2. **Set of types**
   - Data relationship must be of one of the **enumerated types** to be **matched**

Path Patterns

Relationship pattern (cont.)

1. **Property map**
   - Data relationship must have all the **requested properties**
2. **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

```
1 ()
```

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

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

```
1 (:MOVIE)-->(a { name: "Woody Harrelson" })
```

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

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

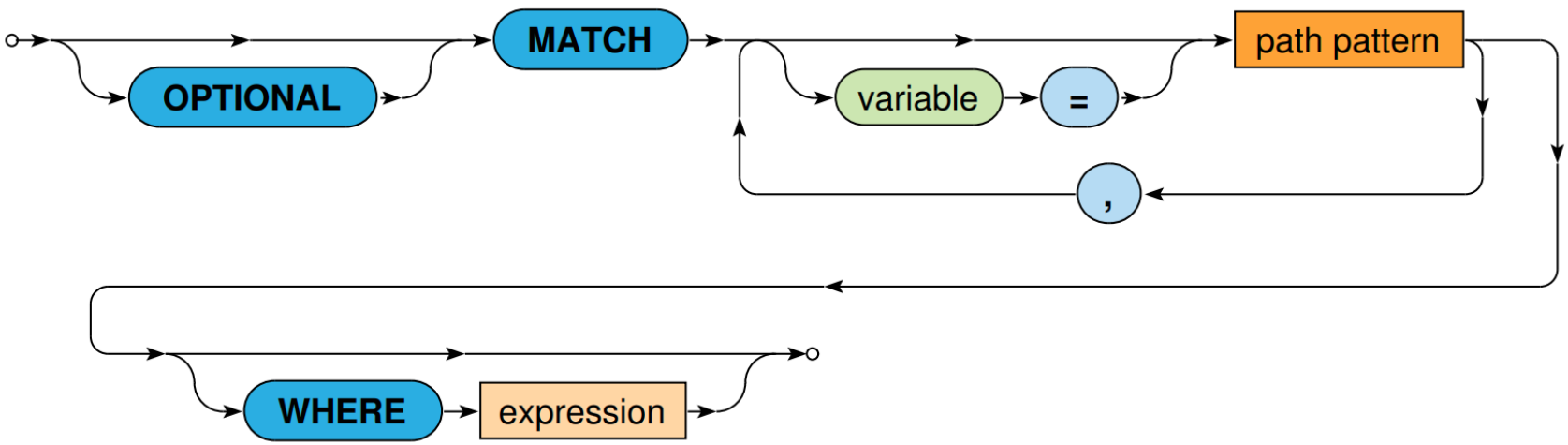
```
1 (:ACTOR { name: "Woody Harrelson" })-[:KNOW *2]->(:ACTOR)
```

```
1 ()-[: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)
  1. **Query result (table)** unordered set of solutions
  2. **One solution (row)** set of variable bindings
- Each variable has to be **bound OPTIONAL**



Match Clause

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

- **Evaluated** directly during the **matching phase** (i.e. not after it)
- **Typical usage**
  1. Boolean expressions
  2. Comparisons
  3. Path patterns – true if at least one solution is found
  4. ...

Match Clause: Example

Find names of actors who played with Woody Harrelson in any movie

```
1 MATCH (i:ACTOR)<-[:PLAY]-(m:MOVIE)-[:PLAY]->(a:ACTOR)
2   WHERE (i.name = "Woody Harrelson")
3 RETURN a.name
```

```
1 MATCH (i:ACTOR { name: "Woody Harrelson" })
2   <-[:PLAY]-(m:MOVIE)-[:PLAY]->
3   (a:ACTOR)
4 RETURN a.name
```

i	m	a
(a1)	(m2)	(a2)
(a1)	(m2)	(a3)
(a1)	(m3)	(a2)

$\Rightarrow$

a.name
Jesse Eisenberg
Emma Stone
Jesse Eisenberg

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)

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

m
(m2)
(m4)

$\Rightarrow$

m	a
(m2)	(a1)
(m2)	(a2)
(m2)	(a3)
(m4)	NULL

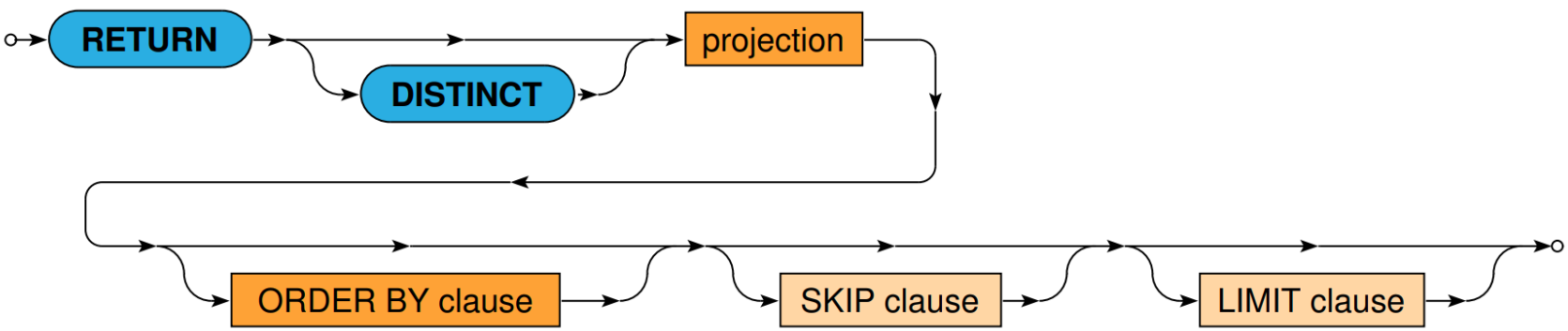
$\Rightarrow$

m.title	a.name
Zombieland	Woody Harrelson
Zombieland	Jesse Eisenberg
Zombieland	Emma Stone
Inception	NULL

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



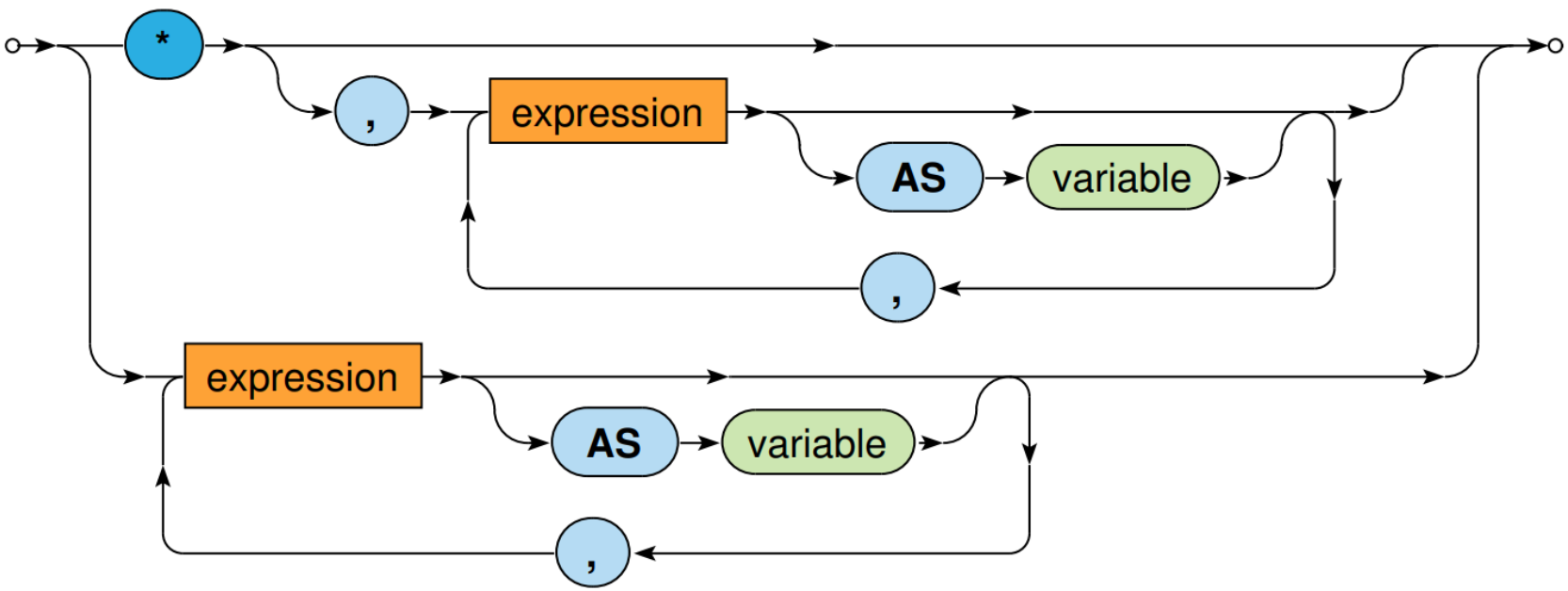
RETURN DISTINCT

- **Duplicate** solutions (rows) are removed

Return Clause

Projection

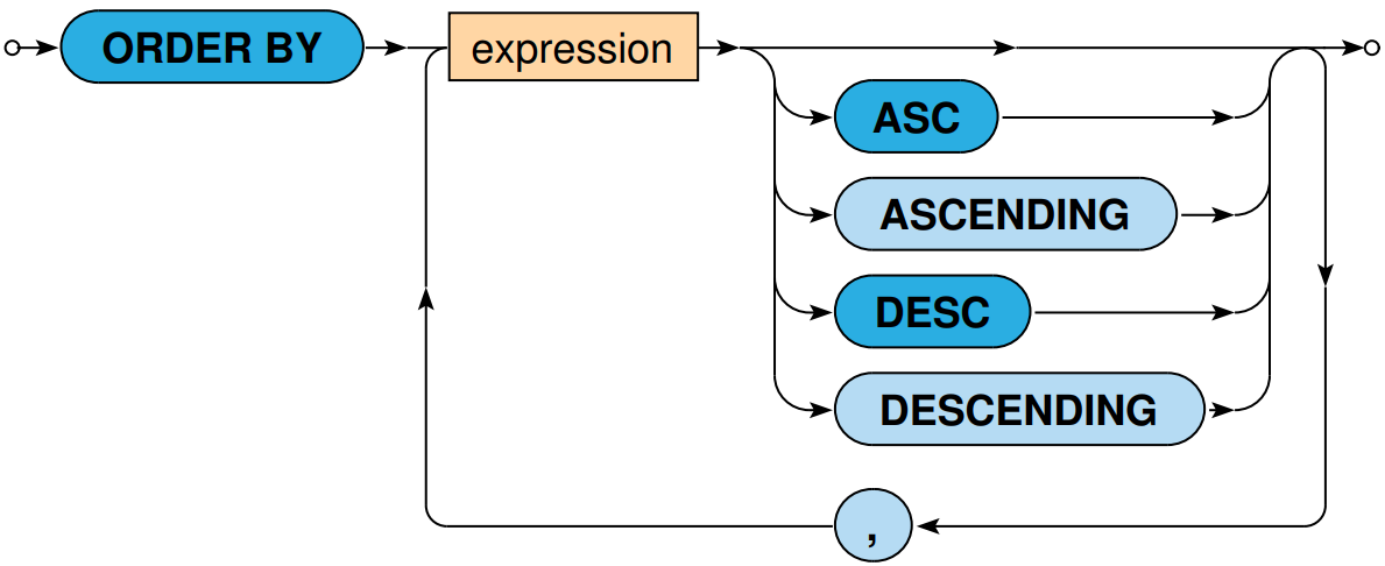
1. * = all the variables
   - Can only be specified as the very first item
2. AS allows to **explicitly (re)name** output records



Return Clause

ORDER BY sub-clause

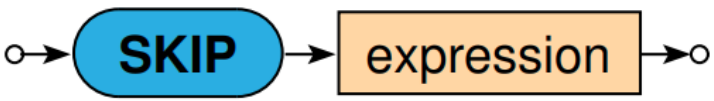
- Defines the **order** of solutions within the query result
  - **Multiple criteria** can be specified
  - **Default direction** is **ASC**
- The order is **undefined** unless explicitly defined
- Nodes and relationships as such **cannot** be used as criteria



Return Clause

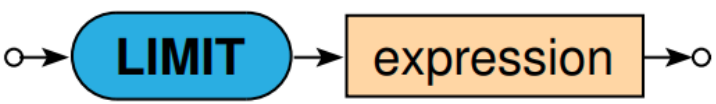
SKIP sub-clause

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



LIMIT sub-clause

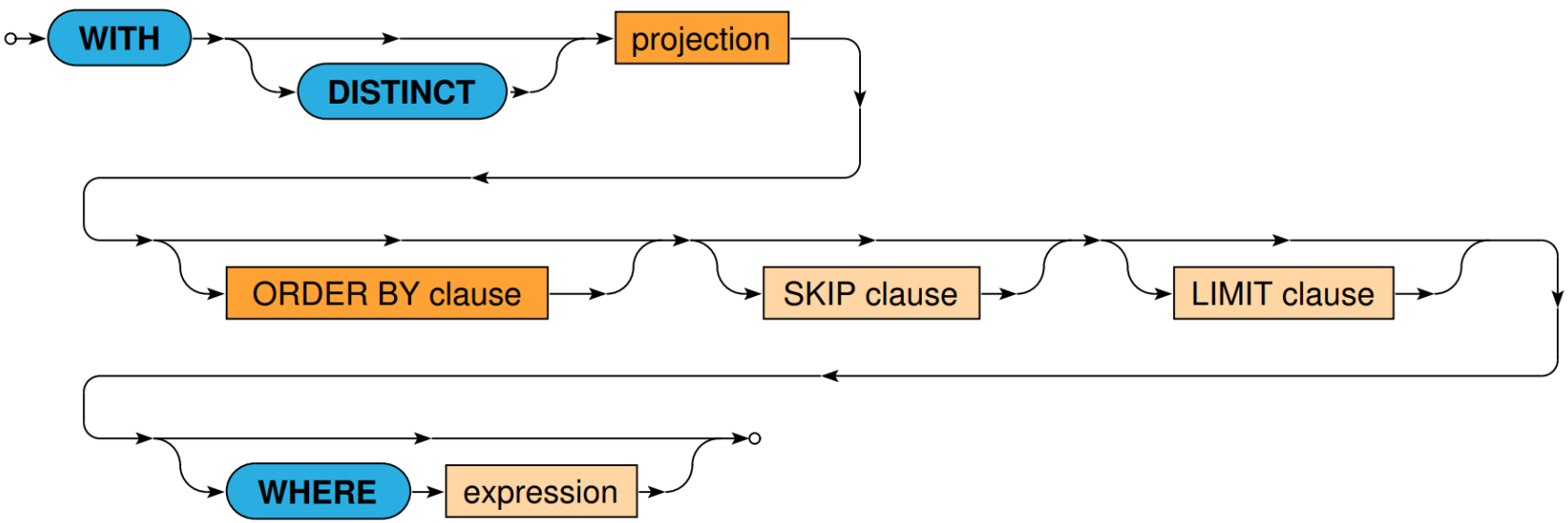
- Determines the **number of solutions** to be **included** in the query result



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

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

a
(a2)
(a3)

$\Rightarrow$

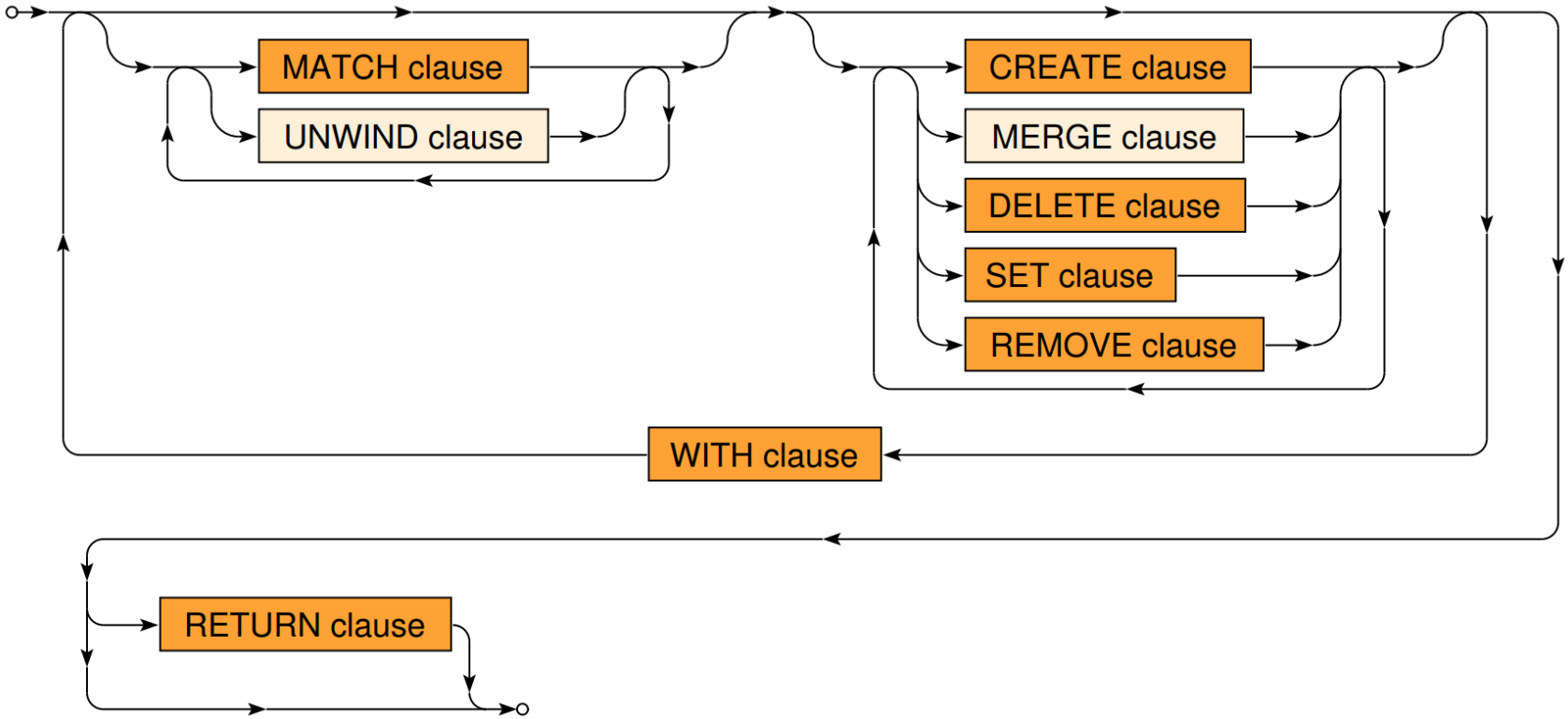
a	movies
(a2)	3
(a3)	1

$\Rightarrow$

a.name	movies
Emma Stone	1
Jesse Eisenberg	3

Query Structure

Chaining of Cypher clauses (simplified)



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

Query Structure

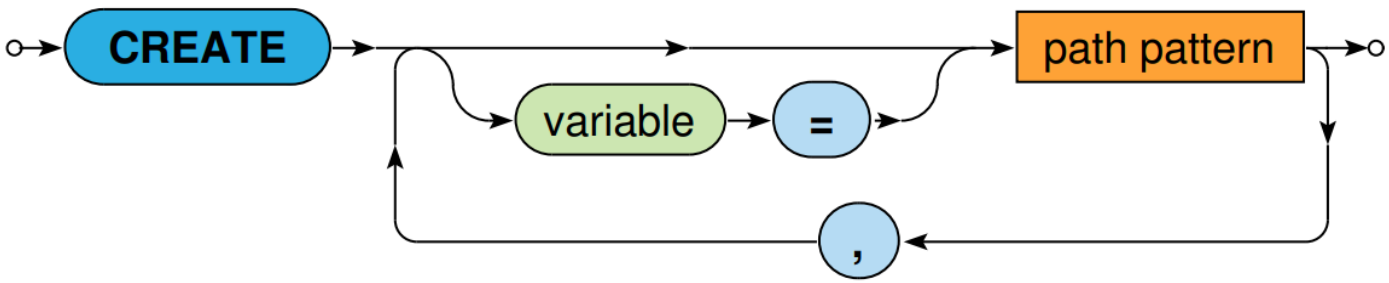
Query parts

- **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



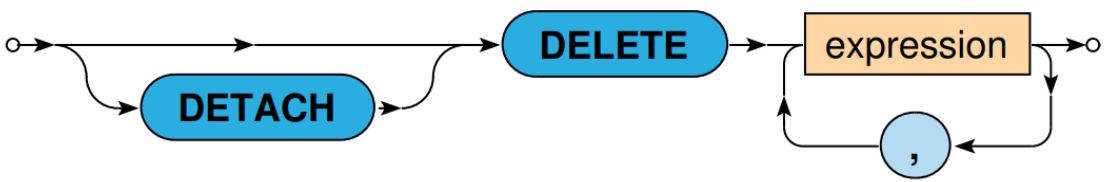
Example

```
1 MATCH (m:MOVIE { id: "inception"})
2 CREATE
3   (a:ACTOR { id: "dicaprio", name: "Leonardo Dicaprio", year: 1978}),
4   (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**



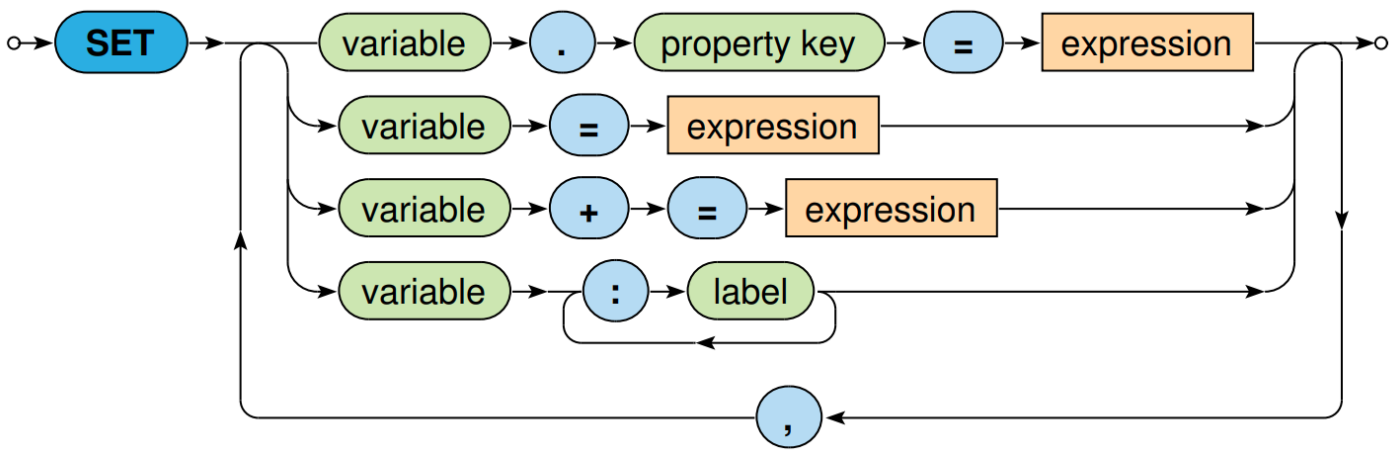
Example

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

Write Clauses

SET clause

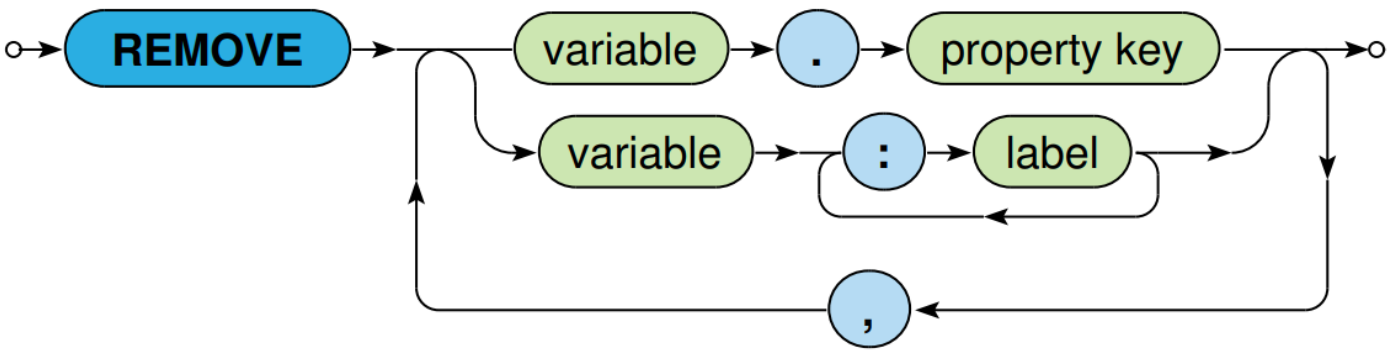
- Allows to...
  - **set** a value of a particular **property**
    - or **remove** a property when **NULL** is assigned
  - **replace** properties (all of them) with new ones
  - **add** **new properties** to the existing ones
  - **add** **labels** to nodes
- **Cannot** be used to **set relationship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

1. **Integers:** decimal, octal, hexadecimal
2. **Floating-point numbers**
3. **Strings**
   - Enclosed in **double** or **single quotes**
   - Standard **escape sequences**
4. **Boolean values:** true, false
5. **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, ...