

NDBI006: **Query Languages II**

<http://www.ksi.mff.cuni.cz/~svoboda/courses/202-NDBI006/>

Lecture 5

Neo4j

Martin Svoboda

svoboda@ksi.mff.cuni.cz

30. 3. 2021

Charles University, Faculty of Mathematics and Physics

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

Lecture Conclusion

Neo4j = graph database

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