



# Grafowe bazy danych

Jarosław Jankowski

# Plan wykładu

## Graflowe bazy danych

- Wprowadzenie

## Neo4j

- Model danych: **struktura grafowa**
- **Framework do przeszukiwania grafów**
- Język zapytań **Cypher**
  - Odczyt, zapis i operacje przeszukiwania

# Grafowe bazy danych

## Model danych

- **struktury grafowe**
  - **Skierowane / nieskierowane** zbiory:
    - **węzłów** (wierzchołków) reprezentujących obiekty świata rzeczywistego lub wirtualnego
  - – **oraz relacji między węzłami (krawędzi)**

Zarówno węzły jak i relacje mogą być powiązane z dodatkowymi **właściwościami np. etykiety, wagi**.

## Typy baz danych

- **Nietransakcyjne** = mała liczba bardzo dużych grafów
- **Transakcyjnych** = duża liczba z małych grafów

# Graflowe bazy danych

## Zapytania

- Tworzenie, aktualizowanie, usuwanie węzłów/krawędzi
- **Algorytmy grafowe** (najkrótsze ścieżki, drzewa rozpinające)
- **Marszruty i ścieżki**
- **Zapytania do sub-grafów**
- Zapytania oparte na podobieństwie (dopasowanie przybliżone)

## Bazy danych:

- **Neo4j, Titan**, Apache Giraph, InfiniteGraph, FlockDB
- *Multi-model*: **OrientDB**, OpenLink **Virtuoso**, **ArangoDB**

# Graflowe bazy danych

## Zastosowania

- Sieci społecznościowe, wyznaczanie tras, usługi oparte na lokalizacji, systemy rekomendacyjne, związki chemiczne, układy biologiczne, struktury lingwistyczne
  - I inne struktury grafowe

Trudności z wykorzystaniem:

- **Złożone operacje wsadowe**
  - Dużo węzłów / krawędzi podlega aktualizacji
- **Operacje na bardzo dużych grafach**
  - Miary sieciowe trudne do wyznaczenia

# Grafowe bazy danych

Przykładowe systemy:



# Neo4j

## Baza danych grafów

- <https://neo4j.com/>
- Funkcje
  - Open source, rozwiązania skalowalne (miliardy węzłów), wysoka dostępność, odporne na błędy, replikacja, obsługa transakcji, technologia osadzania
  - Ekspresyjny język zapytań graficznych (**Cypher**)
- System opracowany przez **Neo Technology**
- Realizowane w Java
- Cross platformowy
- Pierwsze wydanie w 2007

# Model danych

Struktura bazy danych

Instancja → pojedynczy **graf**

**Graf właściwości** = skierowany etykietowany multigraph

- Kolekcji wierzchołków (**węzłów**) i krawędzi (**relacje**)

Węzeł **grafu**

- Ma unikalny (wewnętrzny) **identyfikator**
- Może być powiązany ze zbiorem etykiet
  - Pozwala kategoryzować węzły
- Może być powiązany ze zbiorem **właściwości**
  - Pozwala łączyć dodatkowe dane z węzłami



# Model danych

## Reprezentacja **Relacji**

- Ma unikalny (wewnętrzny) **identyfikator**
- Posiada **Kierunek**
  - Relacje mogą być eksplorowane w obu kierunkach
  - Kierunkowość może być ignorowana podczas zapytań
- Relacja zawsze posiada węzeł początkowy i końcowy
  - Relacje mogą być cykliczne z pętlami
- Odzwierciedla tylko jeden typ **połączenia**
- Może być powiązana ze zbiorem cech/atributów

# Analizy ścieżek

## Dedykowany Framework

- Formułowanie zapytań i ich wykonywanie na grafie

## Definiowanie ścieżek

- Określanie reguł i charakterystyk poszukiwanych ścieżek
- **Weryfikacja ścieżek**
  - Inicjowanie marszruty po grafie według:
    - Formalnego opisu marszruty
      - Zbioru węzłów od których marszruta się rozpoczyna
  - Porównywanie różnych ścieżek i ich własności

# Traversal Framework

Elementy składowe opisu ścieżek

- **Ekspandery**
  - Jakie relacje powinny być rozpatrywane?
- **Wykonanie obliczeń**
  - Jaki algorytm powinien być wykorzystany?
- **Unikatowość**
  - Czy węzły / relacje można odwiedzać wielokrotnie?
- **Ocena efektywności**
  - Kiedy marszruta po grafie powinna zostać zakończona
  - Jakie ścieżki powinny być włączone do wynikowego zapytania?

# Traversal Framework

## Ekspandery ścieżek

*Jaka relacja powinna być rozpatrywana z poziomu danego wężła?*

- **Expander określa jaki ruch jest dozwolony**

- Direction.**INCOMING**
- Direction.**OUTGOING**
- Direction.**BOTH**

Jednocześnie można wprowadzić kilka warunków dla ekspanderów:

Gdy żaden **■** nie jest określony dozwolone są wszystkie relacje

Użycie: **td.relationships**(type, direction)

# Traversal Framework

## Unikalność

*Czy węzły / ścieżki mogą być wielokrotnie odwiedzane?*

- Różne poziomy unikalności:

Uniqueness.**NONE** – bez filtracji

Uniqueness.**NODE\_PATH**

Uniqueness.**RELATIONSHIP\_PATH**

- Unikalne węzły / ścieżki dla bieżącej trasy

Uniqueness.**NODE\_GLOBAL** (domyślne)

Uniqueness.**RELATIONSHIP\_GLOBAL**

- Unikalne węzły / ścieżki dla całej sesji

- Użycie: `td.uniqueness (level)`

# Traversal Framework

## Evaluacja

*Czy aktualna ścieżka powinna być włączona w rezultaty? Czy marszruta powinna być kontynuowana?*

- Dostępne akcje

Evaluation.**INCLUDE\_AND\_CONTINUE**

Evaluation.**INCLUDE\_AND\_PRUNE**

Evaluation.**EXCLUDE\_AND\_CONTINUE**

Evaluation.**EXCLUDE\_AND\_PRUNE**

- Komentarz

- INCLUDE / EXCLUDE = czy włączyć ścieżkę do wyników

- CONTINUE / PRUNE = czy kontynuować przeszukiwanie

-

# Traversal Framework

## Evaluatory

- **Predefiniowane**

- `Evaluators.all()`
  - Nigdy się nie kończy, wszystko włącza
- `Evaluators.excludeStartPosition()`
  - Wyłączone są punkty startowe
- `Evaluators.atDepth(depth)`  
`Evaluators.toDepth(maxDepth)`  
`Evaluators.fromDepth(minDepth)`  
`Evaluators.includingDepths(minDepth, maxDepth)`
  - Włączone określone zakresy głębokości
-

# Traversal Framework

## Evaluatory

- Użycie: `td.evaluator (evaluator)`
- Mogą być włączone też do węzłów startowych!
- Włączenie złożonych ewaluatorów ...
  - Wszystkie muszą akceptować warunki i ograniczenia



# Traversal Framework

## Path

- Zadana sekwencja węzłów i powiązań
- Pozwala przeprowadzić przeszukiwanie grafu rozpoczynając od wskazanego węzła i krawędzi
- Użycie: `t = td.traverse (node,...)`
  - `for (Path p : t) { ... }`
    - Indeksowanie ścieżek
  - `for (Node n : t.nodes()) { ... }`
    - Przeszukiwanie ścieżek i zwrot węzłów
  - `for (Relationship r : t.relationships()) { ... }`
    - Zwrot powiązań
  -

# Przykłady

## Znajdź aktorów

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

Node s = db.findNode(Label.label("movie"), "id", "The
Shining"); Traverser t = td.traverse(s);

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

Nicholson  
Duvall

# Przykłady

Znajdź aktorów, którzy grali z wskazanym aktorem

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

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

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

Duvall

# Cypher

## Cypher

- Deklaratywny grafowy język zapytań
  - Zapytania i aktualizacje
  - Inspirowany przez SQL i SPARQL (wzorce)
- **OpenCypher**
  - Standaryzacja
  - <http://www.opencypher.org/>

## Polecenia

- MATCH, RETURN, CREATE, ...
- **Połączone w sekwencje zapytań**
  -

## Odczyt danych z warunkami

- MATCH – wskazuje wzorzec do poszukiwania
  - WHERE – warunki filtrowania

## Modyfikacje danych

- CREATE – nowy węzeł lub powiązanie
- DELETE – usuwanie obiektów
- SET – aktualizacja etykiet
- REMOVE – usuwanie etykiet

## Klauzule ogólne

- RETURN – zestaw informacji zwrotnych
  - ORDER BY – porządek sortowania
  - SKIP – wyłączenie obiektów
  - LIMIT – ograniczenie zbioru
- WITH – łączenie elementów

# Przykładowe zapytanie

Znajdź aktorów, którzy grali w *The Shining*

```
MATCH (m:movie)-[r:HAS_ACTOR]->(a:actor)
  WHERE m.title = "Nicholson"
RETURN a.name, a.year
ORDER BY a.year
```

a.Name	a.year
Nicholson	1980
Duvall	1980

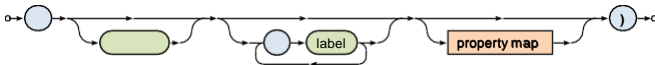
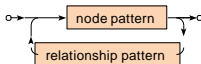
# Wzorce ścieżek

- Reprezentacja ASCII

Dla węzłów ( )

- Strzałki <--, --, --> dla zależności

- Opis pojedynczych ścieżek





# Klauzula Match

Znajdź aktorów, którzy grali z actor: *Nicholson* w dowolnym filmie

```
MATCH (:actor { name: "Nicholson" })  
      <-[:HAS_ACTOR]-(:movie)-[:HAS_ACTOR]->  
      (a:actor)  
RETURN a.name
```

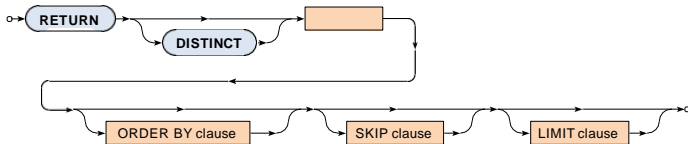
```
MATCH (i:actor)<-[:HAS_ACTOR]-(:movie)-[:HAS_ACTOR]->(a:actor)  
      WHERE (i.name = "Nicholson")  
RETURN a.name
```

a.name
--------

Duvall
--------

# Klauzula Return

- Definiuje zmienne, właściwości, funkcje agregujące
- Opcjonalne ORDER BY, SKIP i LIMIT



## RETURN DISTINCT

- Eliminuje duplikaty

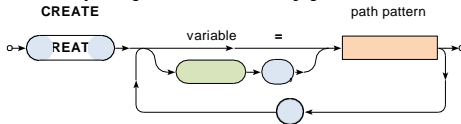
# Operacje write

- CREATE – tworzenie nowych węzłów i relacji
- DELETE – kasowanie obiektów
- SET – etykiety
- REMOVE – usuwanie etykiet

# Operacje write

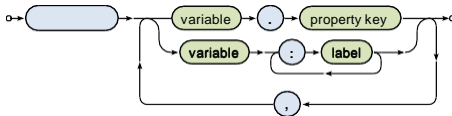
## CREATE

- Wprowadza nowy węzeł lub relację



## REMOVE

- Kasuje węzeł lub relację



# Preinstalowane przykłady

:play cypher

:play movie-graph

:play northwind-graph

:play [http://guides.neo4j.com/modeling\\_airports](http://guides.neo4j.com/modeling_airports)

<https://neo4j.com/graphgist/first-steps-with-cypher>

\$ :play movie-graph

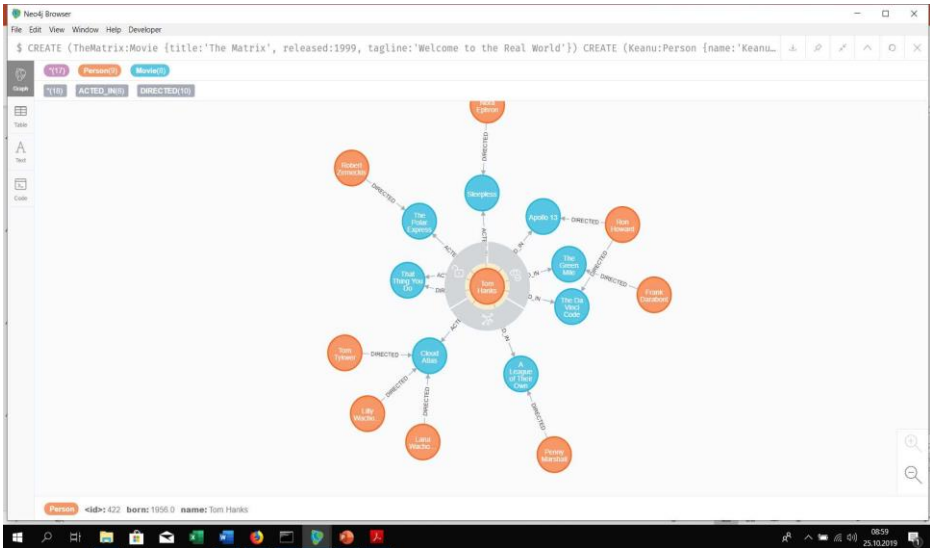
### Movie Graph

Pop-cultural connections between actors and movies

*The Movie Graph* is a mini graph application containing actors and directors that are related through the movies they've collaborated on. This guide will show you how to:

1. Create: insert movie data into the graph
2. Find: retrieve individual movies and actors
3. Query: discover related actors and directors
4. Solve: the Bacon Path

## Preinstalowane przykłady



# Preinstalowane przykłady

\$ :play movie-graph

## The Movie Graph

### Solve

You've heard of the classic "Six Degrees of Kevin Bacon"? That is simply a shortest path query called the "Bacon Path".

1. Variable length patterns
2. Built-in shortestPath() algorithm

Movies and actors up to 4 "hops" away from Kevin Bacon

```
◎ MATCH (bacon:Person {name:"Kevin Bacon"})-[*1..4]-(hollywood)
RETURN DISTINCT hollywood
```


Bacon path, the shortest path of any relationships to Meg Ryan

```
◎ MATCH p=shortestPath(
  (bacon:Person {name:"Kevin Bacon"})-[*]-(meg:Person {name:"Meg Ryan"})
)
RETURN p
```

Note you only need to compare property values like this when first creating relationships


# Inne grafy

<https://neo4j.com/graphgists/>

 [PRODUCTS](#) [SOLUTIONS](#) [CUSTOMERS](#) [PARTNERS](#) [RESOURCES](#) [DEVELOPERS](#) [DOWNLOAD NEO4J](#)



















## Neo4j GraphGists

### Introduction



Building a graph of your data is fairly simple as the graph structure represents the real world much better than columns and rows of data. GraphGists are teaching tools which allow you to explore how data in a particular domain would be modeled as a graph and see some example queries of that graph data. Any developer can create a GraphGist by visiting [portal.graphgist.org](https://portal.graphgist.org).

### Explore By Use Case

 <a href="#">GraphGist Challenge Entries</a>	 <a href="#">Sports and Recreation</a>	 <a href="#">Master Data Management</a>
 <a href="#">Real-Time Recommendations</a>	 <a href="#">Optimization</a>	 <a href="#">Fraud Detection</a>
 <a href="#">Pop Culture</a>	 <a href="#">Network and IT Operations</a>	 <a href="#">Holidays</a>
 <a href="#">Graph-Based Search</a>	 <a href="#">General Business</a>	 <a href="#">Graph Gist How-tos</a>
 <a href="#">Data Analysis</a>	 <a href="#">Public Web APIs</a>	 <a href="#">Internet of Things</a>
 <a href="#">Investigative Journalism</a>	 <a href="#">Open Government Data and Politics</a>	 <a href="#">Identity and Access Management</a>