Property Graphs: Neo4j and Cypher
Neo4j - the most commonly used graph DBMS, by far

https://db-engines.com/en/ranking/graph+dbms

Model: property graphs (graphs with nodes and edges carrying multiple data values arranged as key-value pairs)

Query language: Cypher.
ASCII-art pattern matching + usual database features
Modelling translations and complex access rules

EN: house
Modelling translations and complex access rules
Modelling translations and complex access rules

EN: house
DE: Hause
ES: casa
SE: hus
Modelling translations and complex access rules

EN: building
ES: edificio
DE: Gebäude
SE: byggnad

EN: house
ES: casa
SE: hus
DE: Hause
Whiteboard
Friendliness

Easy to design and model, direct representation of the model
Whiteboard friendliness

Tom Hanks

Acted in

Cloud Atlas

Hugo Weaving

Acted in

The Matrix

Directed

Lana Wachowski
Tom Hanks ACTED_IN Cloud Atlas

Cloud Atlas ACTED_IN Hugo Weaving

Hugo Weaving ACTED_IN The Matrix

The Matrix DIRECTED Lana Wachowski

Lana Wachowski DIRECTED Tom Hanks

Whiteboard friendliness
Whiteboard friendliness

Person | Actor
---|---
name: Tom Hanks
born: 1956

Person | Actor
---|---
name: Hugo Weaving
born: 1960

Movie
ACTED IN
- roles: Zachry
- roles: Bill Smoke

Movie
ACTED IN
- roles: Agent Smith

Person | Director
---|---
name: Lana Wachowski
born: 1965

Movie
title: The Matrix
released: 1999

DIRECTED

DIRECTED
Whiteboard friendliness
Intro to the property graph model
Neo4j Fundamentals

• Nodes
• Relationships
• Properties
• Labels
Nodes
- Represent the objects in the graph
- Can be labeled
Property Graph Model Components

**Nodes**
- Represent the objects in the graph
- Can be *labeled*

**Relationships**
- Relate nodes by *type* and *direction*
Property Graph Model Components

Nodes
- Represent the objects in the graph
- Can be labeled

Relationships
- Relate nodes by type and direction

Properties
- Name-value pairs that can go on nodes and relationships.
Summary of the graph building blocks

• **Nodes** - Entities and complex value types

• **Relationships** - Connect entities and structure domain

• **Properties** - Entity attributes, relationship qualities, metadata

• **Labels** - Group nodes by role
Graph Querying
Why not SQL?

- SQL is inefficient in expressing graph pattern queries
  In particular for queries that
  - are recursive
  - can accept paths of multiple different lengths
- Graph patterns are more intuitive and declarative than joins
- SQL cannot handle path values
Cypher

A pattern matching query language made for graphs

- Declarative
- Expressive
- Pattern Matching
Pattern in our Graph Model

NODE

Dan

LOVES

Ann

Relationship

NODE
Cypher: Express Graph Patterns

Diagram:
- Dan
- Ann
- Relationship: LOVES
Cypher: Express Graph Patterns

\[
(:Person \{ \text{name:} "Dan" \}) -[:LOVES]-> (:Person \{ \text{name:} "Ann" \})
\]
Cypher: Express Graph Patterns

\[
(:Person \{ \text{name:"Dan"} \}) -[:LOVES]-> (:Person \{ \text{name:"Ann"} \})
\]
Cypher: Express Graph Patterns

```
(Person { name: "Dan"}) -[:LOVES]-> (Person { name: "Ann"})
```

- **NODE**: Dan
  - **LABEL**: PERSON
  - **PROPERTY**: name: "Dan"

- **NODE**: Ann
  - **LABEL**: PERSON
  - **PROPERTY**: name: "Ann"

- **RELATIONSHIP**: LOVES
Cypher: CREATE Graph Patterns

Relationship

Dan

LOVES

Ann
Cypher: CREATE Graph Patterns

CREATE (:Person { name:"Dan"}) -[LOVES]-> (:Person { name:"Ann"})

Relationship
Cypher: CREATE Graph Patterns

CREATE (:Person { name:"Dan"} ) -[:LOVES]-> (:Person { name:"Ann"} )
Cypher: CREATE Graph Patterns

```
CREATE (:Person { name:"Dan"}) -[:LOVES]-> (:Person { name:"Ann"})
```

NODE  
Dan  

Relationship  
LOVES  

NODE  
Ann  

LABEL  
PROPERTY  

LABEL  
PROPERTY
Cypher: MATCH Graph Patterns

MATCH Relationship
WHERE Relationship.name = 'LOVES'
RETURN Relationship

- Dan
- ?

Relationship
Cypher: MATCH Graph Patterns

MATCH (:Person { name:"Dan"}) -[:LOVES]-> (whom) RETURN whom
Cypher: MATCH Graph Patterns

MATCH (:Person { name:"Dan"}) -[:LOVES]-> (whom) RETURN whom

MATCH (:Person { name:"Dan"}) -[:LOVES]-> (whom) RETURN whom
Cypher: MATCH Graph Patterns

MATCH (:Person { name: "Dan" }) -[:LOVES]-> (whom)
RETURN whom
A graph query example
A social recommendation
MATCH (person:Person)-[:IS_FRIEND_OF]->(friend),
(friend)-[:LIKES]->(restaurant),
(restaurant)-[:LOCATED_IN]->(loc:Location),
(restaurant)-[:SERVES]->(type:Cuisine)
WHERE person.name = 'Philip'
AND loc.location='New York'
AND type.cuisine='Sushi'
RETURN restaurant.name
The Syntax
Nodes

Nodes are drawn with parentheses.

()
Relationships

Relationships are drawn as arrows, with additional detail in brackets.

\[ \rightarrow \]

\[-[ : DIRECTED ] -\rightarrow \]
Patterns

Patterns are drawn by connecting nodes and relationships with hyphens, optionally specifying a direction with > and < signs.

( ) - [ ] - ( )
( ) - [ ] - > ( )
( ) <- [ ] - ( )
The components of a Cypher query

MATCH (m:Movie)
RETURN m

MATCH and RETURN are Cypher keywords
m is a variable
:Movie is a node label
The components of a Cypher query

MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
RETURN p, r, m

MATCH and RETURN are Cypher keywords
p, r, and m are variables
:Movie is a node label
:ACTED_IN is a relationship type
The components of a Cypher query

```
MATCH path = (:Person)-[:ACTED_IN]->(:Movie)
RETURN path
```

MATCH and RETURN are Cypher keywords
path is a variable
:Movie is a node label
:ACTED_IN is a relationship type
Graph versus Tabular results

MATCH (m:Movie)
RETURN m
Graph versus Tabular results

MATCH (m:Movie)
RETURN m.title, m.released

Properties are accessed with `{variable}.{property_key}`
Case sensitivity

Case sensitive

Node labels
Relationship types
Property keys

Case insensitive

Cypher keywords
## Case sensitivity

<table>
<thead>
<tr>
<th>Case sensitive</th>
<th>Case insensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>:Person</td>
<td>MaTcH</td>
</tr>
<tr>
<td>:ACTED_IN</td>
<td>return</td>
</tr>
<tr>
<td>name</td>
<td></td>
</tr>
</tbody>
</table>
Write queries
The CREATE Clause

```sql
CREATE (m:Movie {title:'Mystic River', released:2003})
RETURN m
```
The SET Clause

MATCH (m:Movie {title: 'Mystic River'})
SET m.tagline = 'We bury our sins here, Dave. We wash them clean.'
RETURN m
The CREATE Clause

MATCH (m:Movie {title: 'Mystic River'})
MATCH (p:Person {name: 'Kevin Bacon'})
CREATE (p)-[r:ACTED_IN {roles: ['Sean']}]-(m)
RETURN p, r, m
The MERGE Clause

```mergo
MERGE (p:Person {name: 'Tom Hanks'})
RETURN p
```
The MERGE Clause

```
MERGE (p:Person {name: 'Tom Hanks', oscar: true})
RETURN p
```
The MERGE Clause

```
MERGE (p:Person {name: 'Tom Hanks', oscar: true})
RETURN p
```

There is not a :Person node with name:'Tom Hanks' and oscar: true in the graph, but there is a :Person node with name:'Tom Hanks'.

What do you think will happen here?
The MERGE Clause

```
MERGE (p:Person {name: 'Tom Hanks'})
SET p.oscar = true
RETURN p
```
The MERGE Clause

```
MERGE (p:Person {name: 'Tom Hanks'})-[[:ACTED_IN] ->(m:Movie {title: 'The Terminal'})
RETURN p, m
```
The MERGE Clause

MERGE (p:Person {name: 'Tom Hanks'})-[[:ACTED_IN] ->(m:Movie {title: 'The Terminal'})

RETURN p, m

There is not a :Movie node with title:"The Terminal" in the graph, but there is a :Person node with name:"Tom Hanks".

What do you think will happen here?
The MERGE Clause

MERGE (p:Person {name: 'Tom Hanks'})
MERGE (m:Movie {title: 'The Terminal'})
MERGE (p)-[r:ACTED_IN]->(m)
RETURN p, r, m
ON CREATE and ON MATCH

MERGE (p:Person {name: 'Your Name'})
  ON CREATE SET p.created = timestamp(), p.updated = 0
  ON MATCH SET p.updated = p.updated + 1
RETURN p.created, p.updated;
Graph Modeling
Models
The modeling workflow

1. Derive the question
2. Obtain the data
3. Develop a model
4. Ingest the data
5. Query/Prove our model
Developing the model and the query

1. Identify application/end-user goals
2. Figure out what questions to ask of the domain
3. Identify entities in each question
4. Identify relationships between entities in each question
5. Convert entities and relationships to paths
   - These become the basis of the data model
6. Express questions as graph patterns
   - These become the basis for queries
1. Application/End-User Goals

As an employee
I want to know who in the company has similar skills to me
So that we can exchange knowledge
2. Questions to ask of the Domain

As an employee
I want to know who in the company has similar skills to me
So that we can exchange knowledge

Which people, who work for the same company as me, have similar skills to me?
3. Identify Entities

Which *people*, who work for the same *company* as me, have similar *skills* to me?

- *Person*
- *Company*
- *Skill*
Which people, who work for the same company as me, have similar skills to me?

- **Person WORKS FOR Company**
- **Person HAS SKILL Skill**
5. Convert to Cypher Paths
5. Convert to Cypher Paths

- Person WORKS FOR Company
- Person HAS SKILL Skill
5. Convert to Cypher Paths

- **Person** WORKS FOR **Company**
- **Person** HAS SKILL **Skill**
5. Convert to Cypher Paths

- **Person WORKS FOR Company**

- **Person HAS SKILL Skill**
5. Convert to Cypher Paths

- **Person WORKS FOR** Company
- **Person HAS SKILL** Skill

- (:Person)-[:WORKS_FOR]->(:Company),
- (:Person)-[:HAS_SKILL]->(:Skill)
5. Convert to Cypher Paths

- **Person** WORKS FOR **Company**

- **Person** HAS SKILL **Skill**

- (:Person)-[:WORKS_FOR]->(:Company),

- (:Person)-[:HAS_SKILL]->(:Skill)
5. Convert to Cypher Paths

- **Person** WORKS FOR **Company**

- **Person** HAS SKILL **Skill**

- (:Person)-[:WORKS_FOR]->(:Company),
- (:Person)-[:HAS_SKILL]->(:Skill)
Consolidate Pattern

(:Person)-[:WORKS_FOR]->(:Company),
(:Person)-[:HAS_SKILL]->(:Skill)

(:Company)<-[:WORKS_FOR]-(:Person)-[:HAS_SKILL]->(:Skill)
Candidate Data Model

(:Company)<-[[:WORKS_FOR]]-(:Person)-[[:HAS_SKILL]]->(:Skill)
Which people, who work for the same company as me, have similar skills to me?
Cypher Query

Which people, who work for the same company as me, have similar skills to me?

MATCH (company)<-[WORKS_FOR]-(me:Person)-[HAS_SKILL]->(skill)
  (company)<-[WORKS_FOR]-(colleague)-[HAS_SKILL]->(skill)
WHERE me.name = $name
RETURN colleague.name AS name,
    count(skill) AS score,
    collect(skill.name) AS skills
ORDER BY score DESC
Which people, who work for the same company as me, have similar skills to me?

MATCH (company)<-[::WORKS_FOR]-(me:Person)-[::HAS_SKILL]->(skill)
(company)<-[::WORKS_FOR]-(colleague)-[::HAS_SKILL]->(skill)
WHERE me.name = $name
RETURN colleague.name AS name,
    count(skill) AS score,
    collect(skill.name) AS skills
ORDER BY score DESC

I. Graph pattern
Cypher Query

Which people, who work for the same company as me, have similar skills to me?

MATCH (company)<-[::WORKS_FOR]-(me:Person)-[:HAS_SKILL]->(skill)
  (company)<-[::WORKS_FOR]-(colleague)-[:HAS_SKILL]->(skill)
WHERE me.name = $name
RETURN colleague.name AS name,
  count(skill) AS score,
  collect(skill.name) AS skills
ORDER BY score DESC

1. Graph pattern
2. Filter, using index if available
Cypher Query

Which people, who work for the same company as me, have similar skills to me?

MATCH (company)<-[[:WORKS_FOR]]-(me:Person)-[:HAS_SKILL]->(skill)
  (company)<-[[:WORKS_FOR]]-(colleague)-[:HAS_SKILL]->(skill)
WHERE me.name = $name
RETURN colleague.name AS name,
  count(skill) AS score,
  collect(skill.name) AS skills
ORDER BY score DESC

1. Graph pattern
2. Filter, using index if available
3. Create projection of result
First Match

Person
- name: Tobias
  - HAS_SKILL: Scala

Person
- name: Ian
  - HAS_SKILL: C#
  - WORKS_FOR: ACME

Person
- name: Jacob
  - HAS_SKILL: Python
  - WORKS_FOR: ACME

Person
- name: Tobias
  - WORKS_FOR: me

Person
- name: colleague
  - WORKS_FOR: company

Company
- name: ACME
  - WORKS_FOR: me

Skill
- name: Scala
  - HAS_SKILL: Tobias

Skill
- name: Python
  - HAS_SKILL: Jacob

Skill
- name: Neo4j
  - HAS_SKILL: Tobias
  - HAS_SKILL: Ian

Skill
- name: C#
  - HAS_SKILL: Ian

Skill
- name: Python
  - HAS_SKILL: colleague
Second Match
Third Match

Person
- name: Tobias
  - HAS SKILL: Scala
  - WORKS FOR: ACME

Person
- name: Ian
  - HAS SKILL: C#, Python, Neo4j
  - WORKS FOR: ACME

Person
- name: Jacob
  - HAS SKILL: Scala, Python
  - WORKS FOR: ACME

Skill
- name: Scala
- name: C#
- name: Neo4j
- name: Python

Company
- me
- colleague

HAS SKILL: Scala, Python, C#, Neo4j
WORKS FOR: ACME
Result of the Query

<table>
<thead>
<tr>
<th>name</th>
<th>score</th>
<th>skills</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Ian&quot;</td>
<td>2</td>
<td>[&quot;Scala&quot;,&quot;Neo4j&quot;]</td>
</tr>
<tr>
<td>&quot;Jacob&quot;</td>
<td>1</td>
<td>[&quot;Neo4j&quot;]</td>
</tr>
</tbody>
</table>

2 rows
Modeling exercise: Movie genres
Adding movie genres

The question: should we model them as properties or as nodes?
Genres as properties

MATCH (m:Movie {title: 'The Matrix'})
SET m.genre = ['Action', 'Sci-Fi']
RETURN m
Genres as properties

MATCH (m:Movie {title: 'Mystic River'})
SET m.genre = ['Action', 'Mystery']
RETURN m
The good side of properties

Accessing a movie’s genres is quick and easy.

MATCH (m:Movie {title: "The Matrix"})
RETURN m.genre;
Finding movies that share genres is painful and we have a disconnected pattern in the MATCH clause - a sure sign you have a modeling issue.

MATCH (m1:Movie), (m2:Movie)
WHERE any(x IN m1.genre WHERE x IN m2.genre)
AND m1 <> m2
RETURN m1, m2;
Genres as nodes

MATCH (m:Movie {title:"The Matrix"})
MERGE (action:Genre {name:"Action"})
MERGE (scifi:Genre {name:"Sci-Fi"})
MERGE (m)-[:IN_GENRE]->(action)
MERGE (m)-[:IN_GENRE]->(scifi)
MATCH (m:Movie {title: "Mystic River"})
MERGE (action:Genre {name: "Action"})
MERGE (mystery:Genre {name: "Mystery"})
MERGE (m)-[:IN_GENRE]->(action)
MERGE (m)-[:IN_GENRE]->(mystery)
The good side of nodes

Finding movies that share genres is a natural graph pattern.

MATCH (m1:Movie)-[:IN_GENRE]->(g:Genre),
     (m2:Movie)-[:IN_GENRE]->(g)
RETURN m1, m2, g
The (not too) bad side of nodes

Accessing the genres of movies requires a bit more typing.

MATCH (m:Movie {title:"The Matrix"}),
    (m)-[:IN_GENRE]->(g:Genre)
RETURN g.name;
Symmetric Relationships
Symmetric relationships

OR

OR
Bidirectional Relationships
Use single relationship and ignore direction in queries

MATCH (:Person {name:'Eric'})-[[:MARRIED_TO]]-(p2)
RETURN p2
Formal semantics

- Nadime Francis, Paolo Guagliardo, Leonid Libkin

- Formally defines a (large) core of Cypher
Future improvements
Regular Path Queries

- Conjunctive bi-directional Regular Path Queries with Data
  - Plus specification of a cost function for paths, which allows for more interesting notions of shortest path

```
PATH PATTERN coauth=( )-[ :WROTE ]->(b)<-[ :WROTE ]-( ),
(b)<-[sale:SELLS]-()
COST min(sale.price)
```

```
MATCH (a)-/~coauth* COST x/->(b)
ORDER BY x LIMIT 10
```
Further improvements

• Support for querying from multiple graphs
• Support for returning graphs
• Support for defining views

• Integration with SQL
RDBMSs and Graphs
RDBMS can’t handle relationships well

- Cannot model or store data and relationships without complexity
- Performance degrades with number and levels of relationships, and database size
- Query complexity grows with need for JOINs
- Adding new types of data and relationships requires schema redesign, increasing time to market
Express Complex Queries Easily with Cypher

Find all managers and how many people they manage, up to 3 levels down.

**Cypher**

MATCH (boss)-[:MANAGES*0..3]->(mgr)

WHERE boss.name = "John Doe"
AND (mgr)-[:MANAGES]->()

RETURN mgr.name AS Manager,
    size((mgr)-[:MANAGES*1..3]->()) AS Total

**SQL**

```
SELECT T.directReporters AS directReporters, SUM(T.count) AS count
FROM person AS T
GROUP BY T.reportees
UNION
SELECT T.directReporters AS directReporters, SUM(T.count) AS count
FROM person AS T
GROUP BY T.reportees
ORDER BY count DESC
```
Unlocking Value from Your Data Relationships

- Model your data **naturally** as a graph of data and relationships
- Drive graph model **from domain and use-cases**
- Use relationship information in **real-time** to transform your business
- Add new relationships **on the fly** to adapt to your changing requirements
Relationships are first class citizen
• No need for joins, just follow pre-materialized relationships of nodes
• Query & Data-locality – navigate out from your starting points
• Only load what’s needed
• Aggregate and project results as you go
• Optimized disk and memory model for graphs
The performance advantage of materialised relationships

- a sample social graph with ~1,000 persons
- average 50 friends per person
- pathExists(a,b) limited to depth 4
- caches warmed up to eliminate disk I/O

<table>
<thead>
<tr>
<th></th>
<th># persons</th>
<th>query time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relational database</td>
<td>1,000</td>
<td>2000ms</td>
</tr>
<tr>
<td>Neo4j</td>
<td>1,000</td>
<td>2ms</td>
</tr>
<tr>
<td>Neo4j</td>
<td>1,000,000</td>
<td>2ms</td>
</tr>
</tbody>
</table>
1. Download Neo4j: [http://neo4j.com/download/](http://neo4j.com/download/)
2. Start the server.
3. It should be running on: [http://localhost:7474](http://localhost:7474)
4. Log-in with default credentials
   
   user: *neo4j*
   
   password: *neo4j*

5. Choose a new password

We’re good to go!