# Property Graphs: Neo4j and Cypher

Neo4j - the most commonly used graph DBMS, by far

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

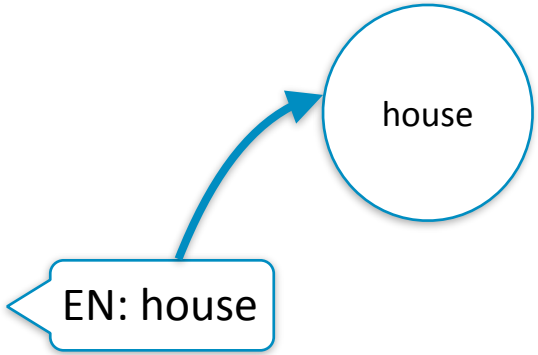
Model: property graphs (graphs with nodes and edges carrying multiple data values arranged as key-value pairs)

Query language: Cypher.
ASCII-art pattern matching + usual database features

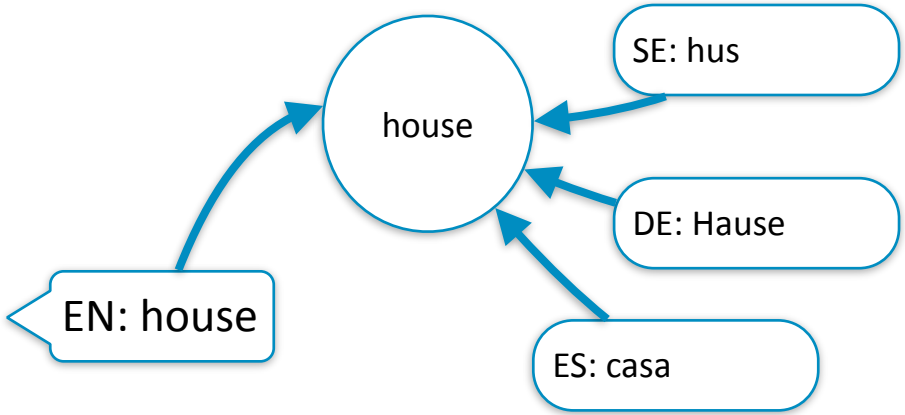
# Modelling translations and complex access rules

EN: house

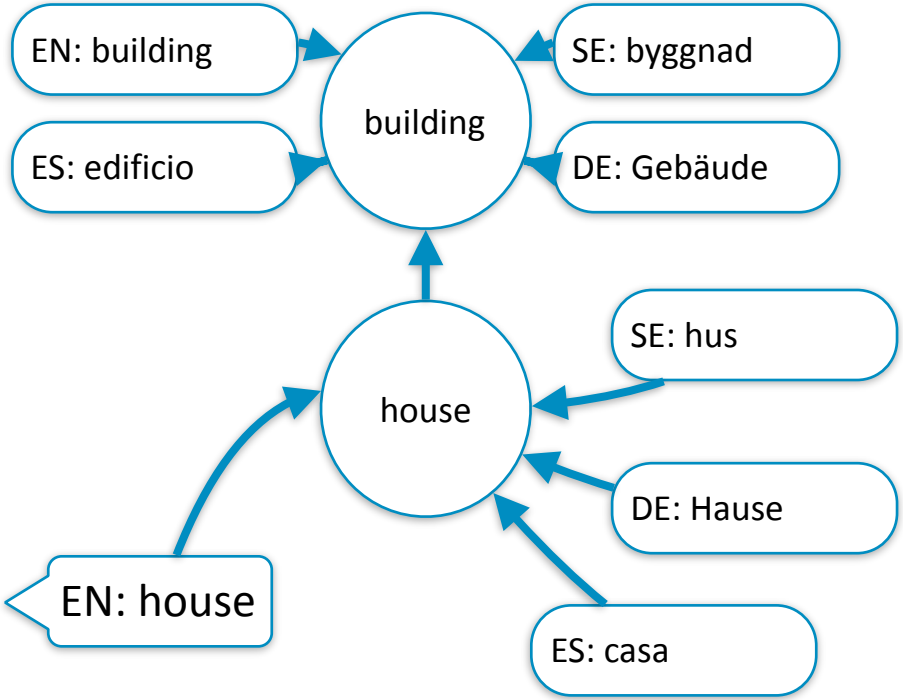
# Modelling translations and complex access rules

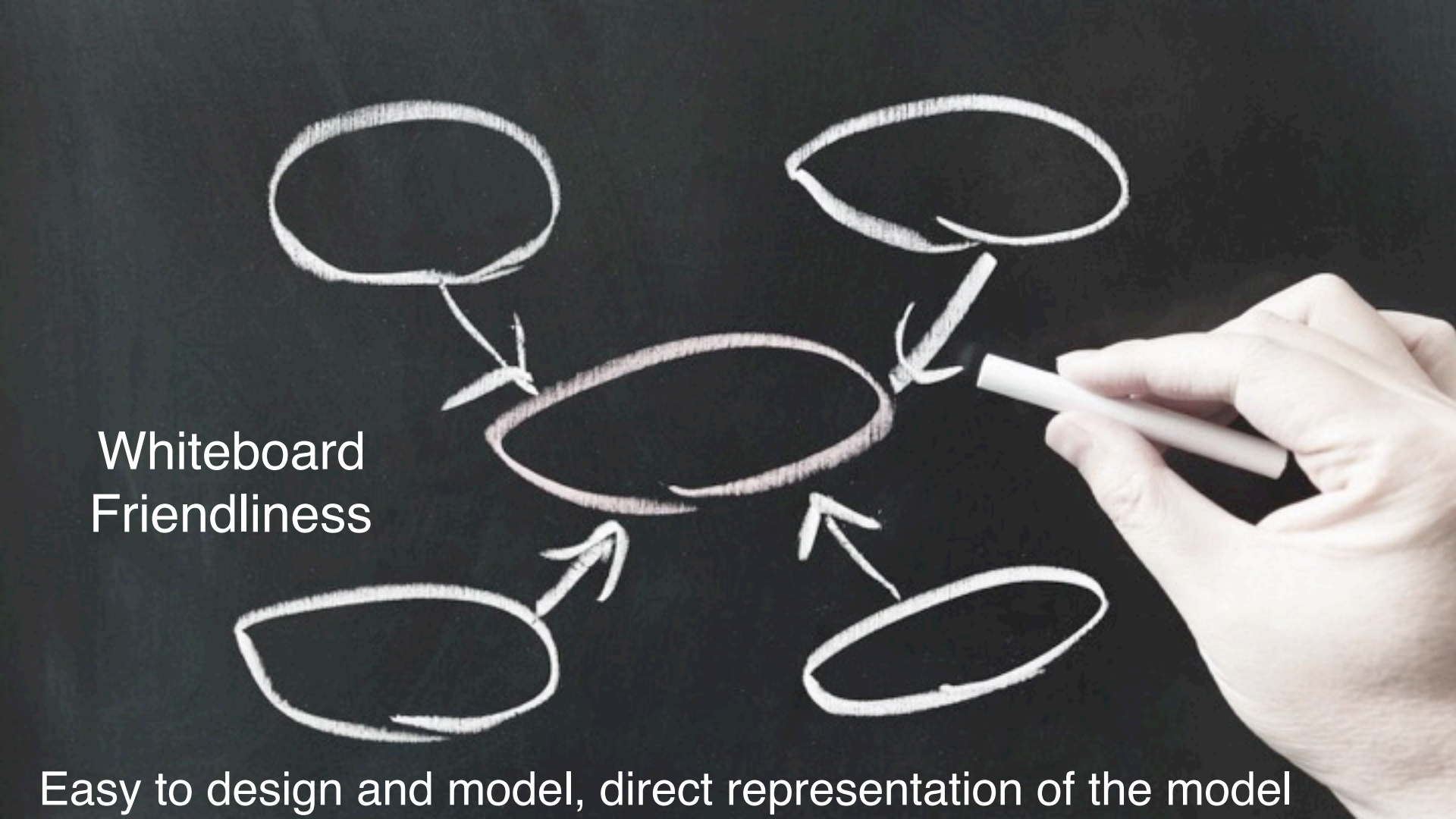


# Modelling translations and complex access rules



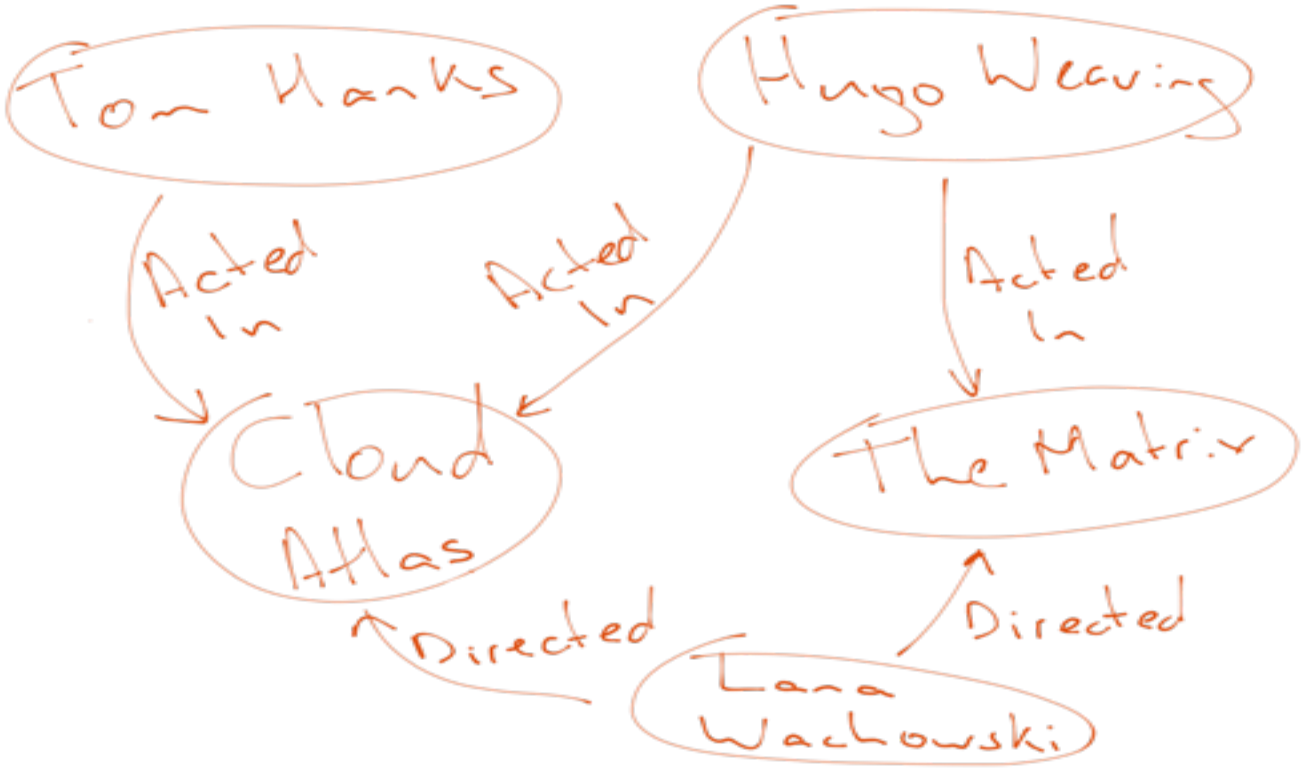
# Modelling translations and complex access rules



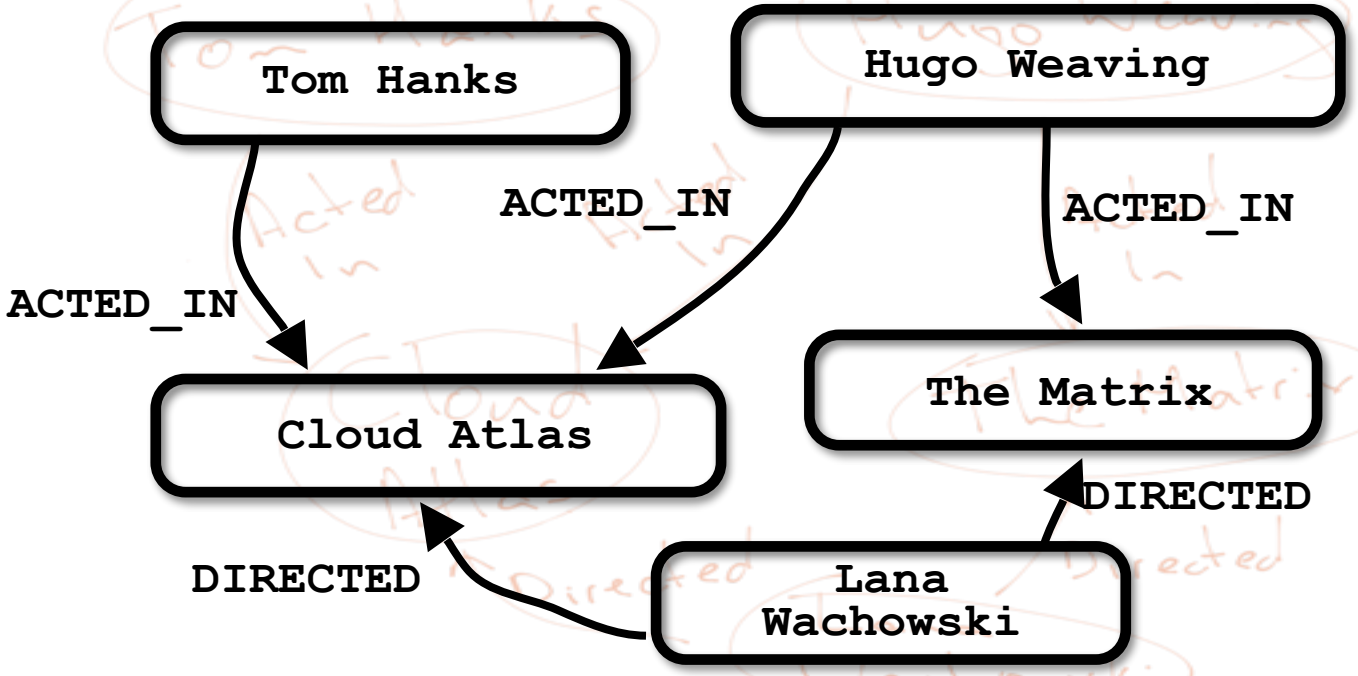



Easy to design and model, direct representation of the model

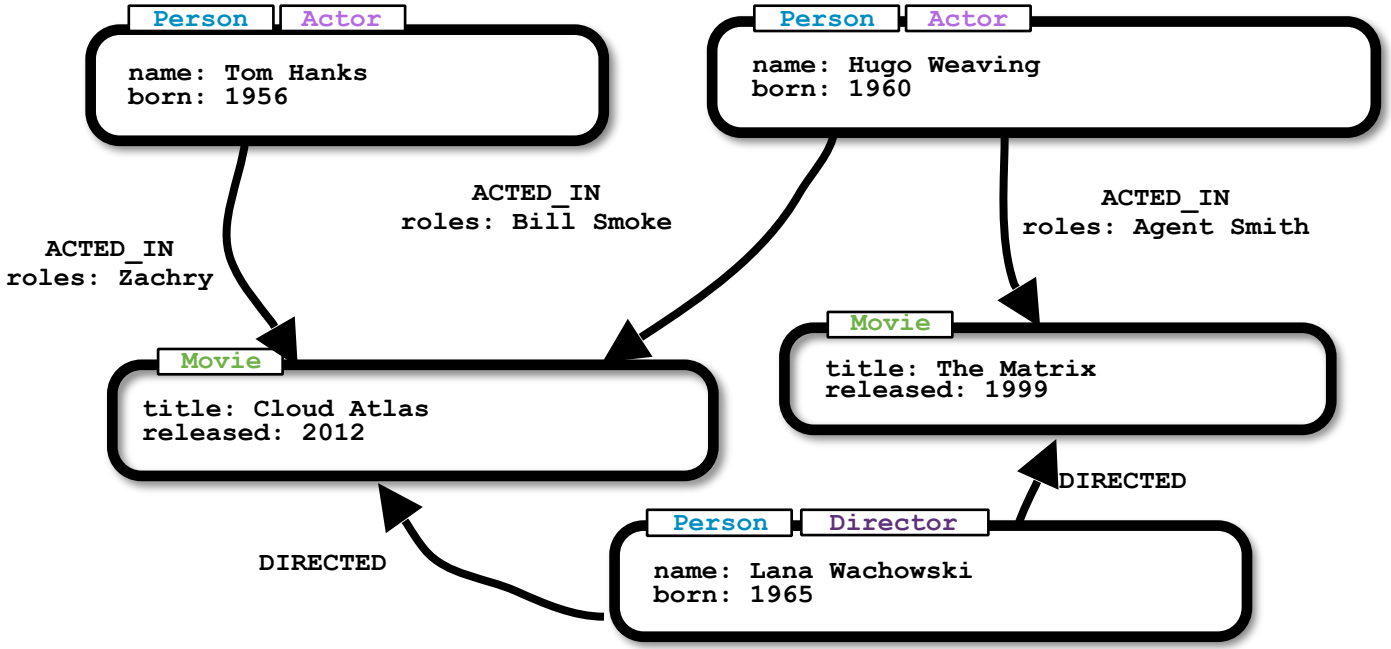
# Whiteboard friendliness



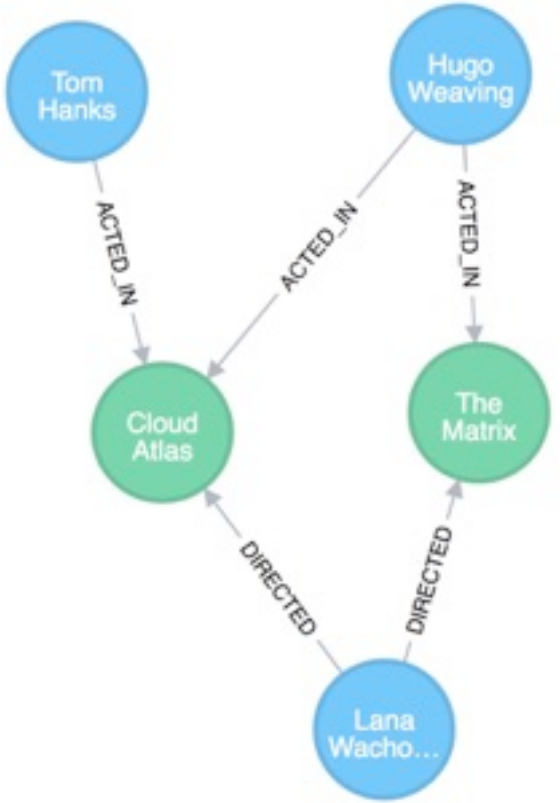
# Whiteboard friendliness



# Whiteboard friendliness



# Whiteboard friendliness



# Intro to the property graph model

# Neo4j Fundamentals


- Nodes
- Relationships
- Properties
- Labels

# Property Graph Model Components

## Nodes

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



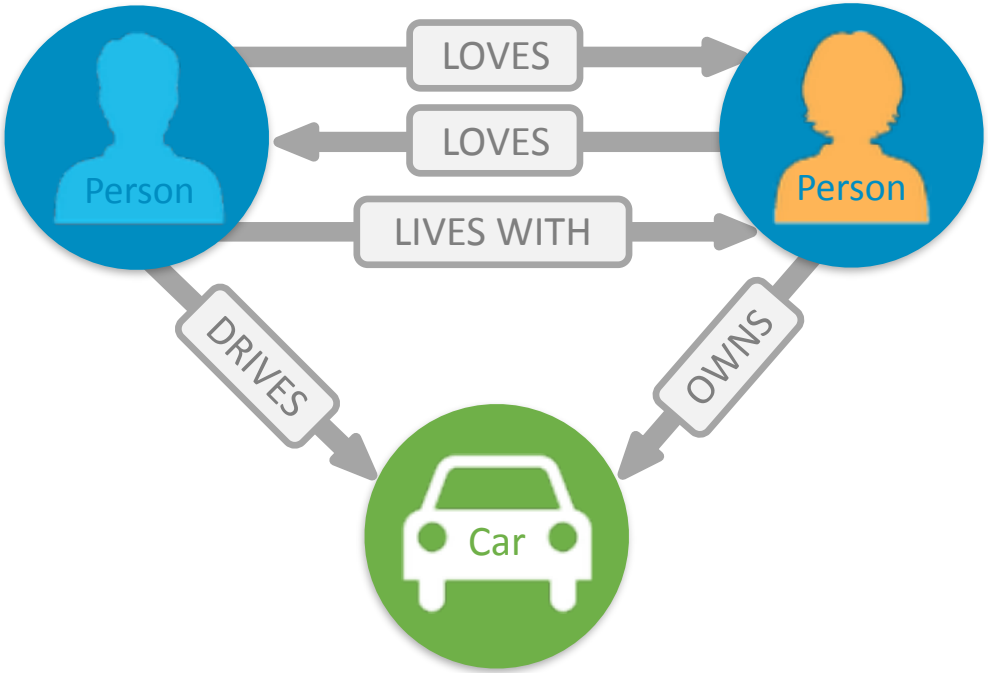
# Property Graph Model Components

## Nodes

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

## Relationships

- Relate nodes by *type* and *direction*



# Property Graph Model Components

## Nodes

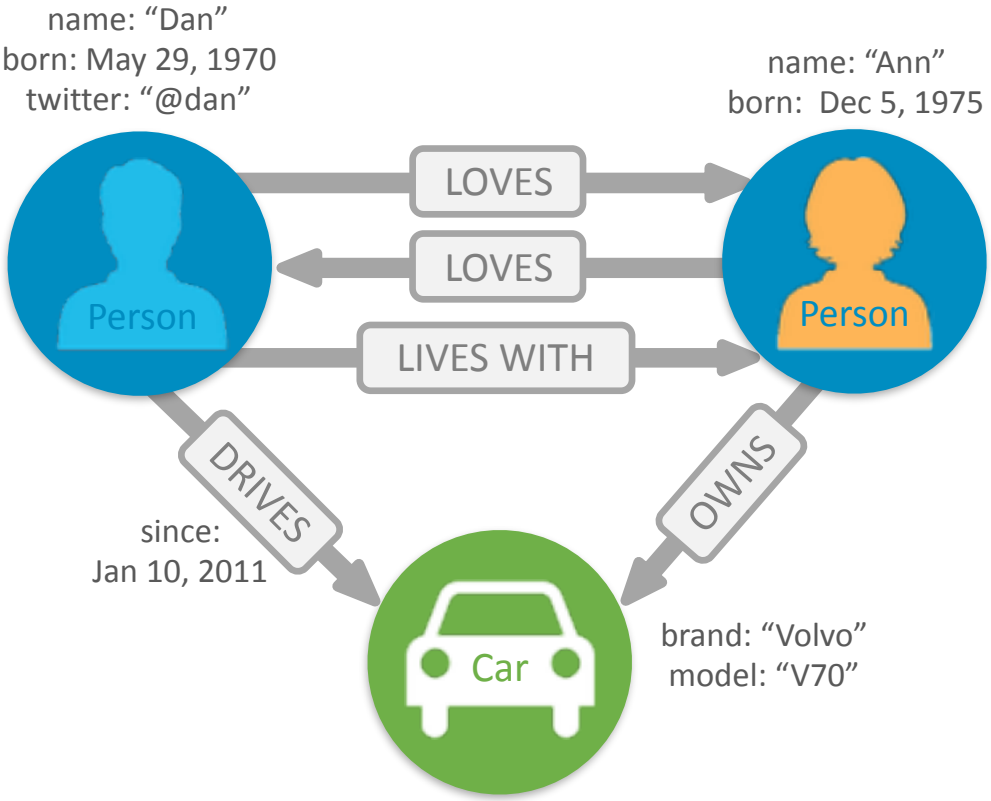
- Represent the objects in the graph
- Can be *labeled*

## 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

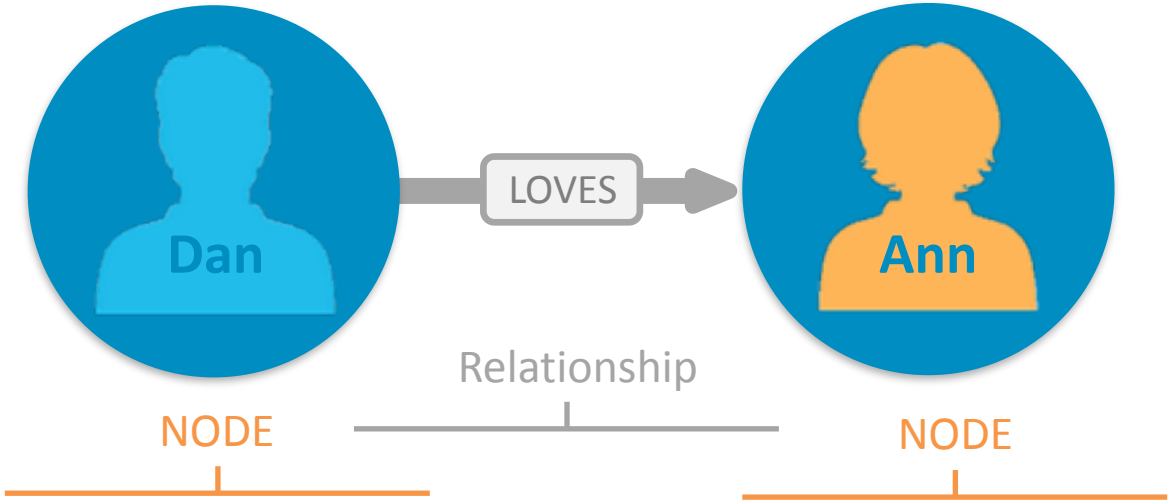
- SQL cannot handle path values

# Cypher

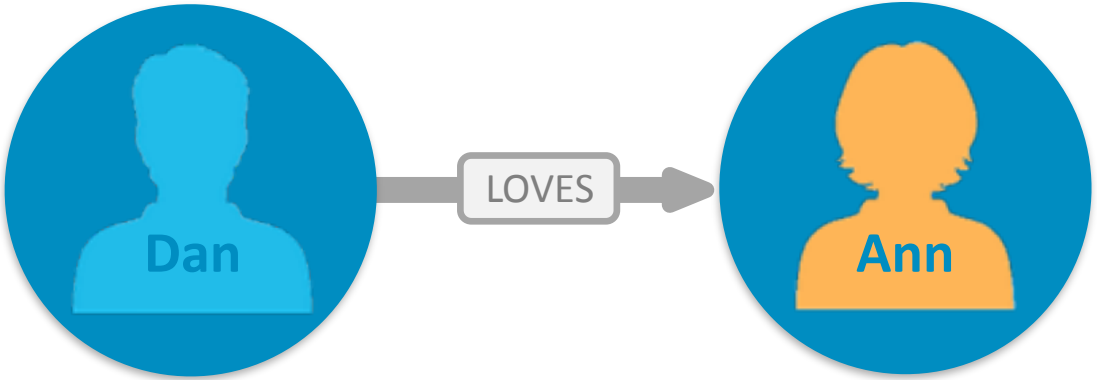
A pattern matching query language made for graphs

- Declarative
- Expressive
- Pattern Matching

# Pattern in our Graph Model

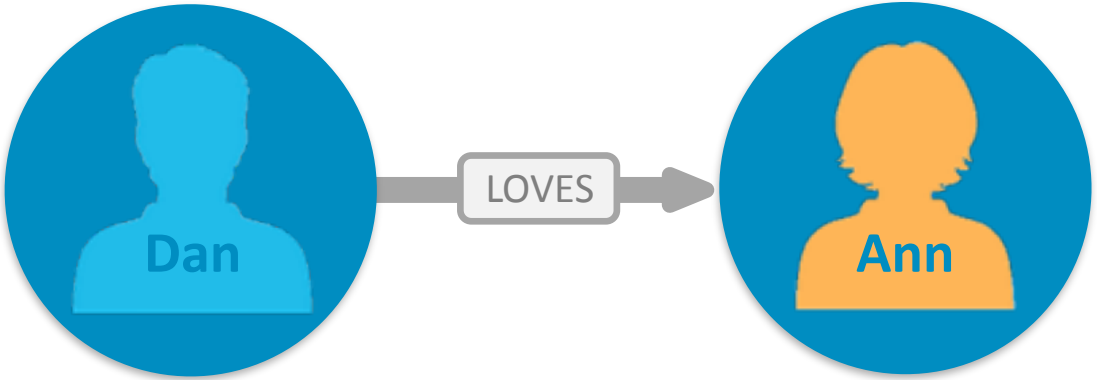


# Cypher: Express Graph Patterns




Relationship

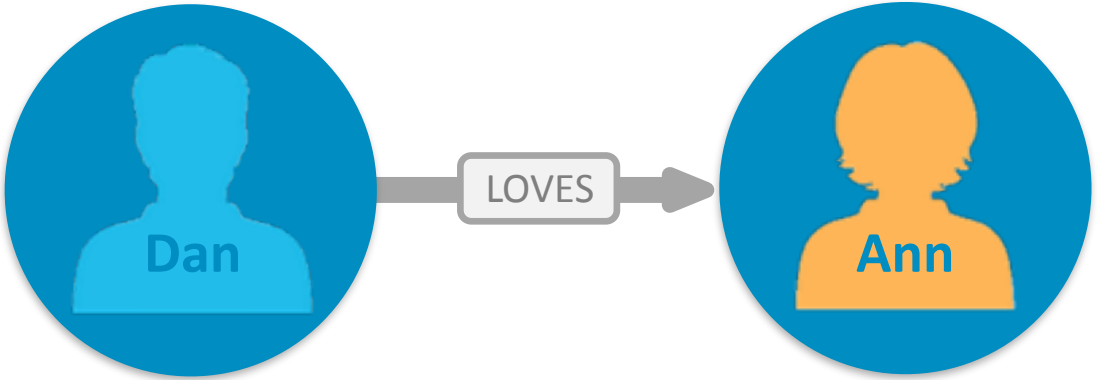
# Cypher: Express Graph Patterns




Relationship

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

# Cypher: Express Graph Patterns

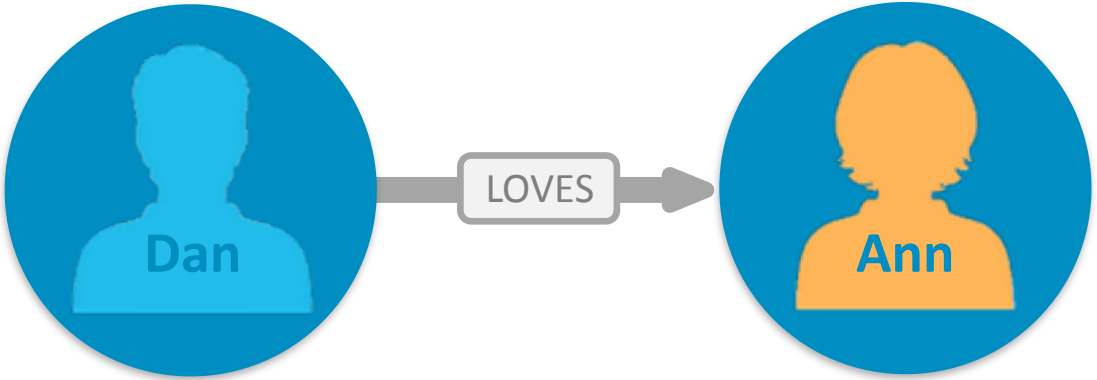



Relationship

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

LABEL PROPERTY LABEL PROPERTY

# Cypher: Express Graph Patterns

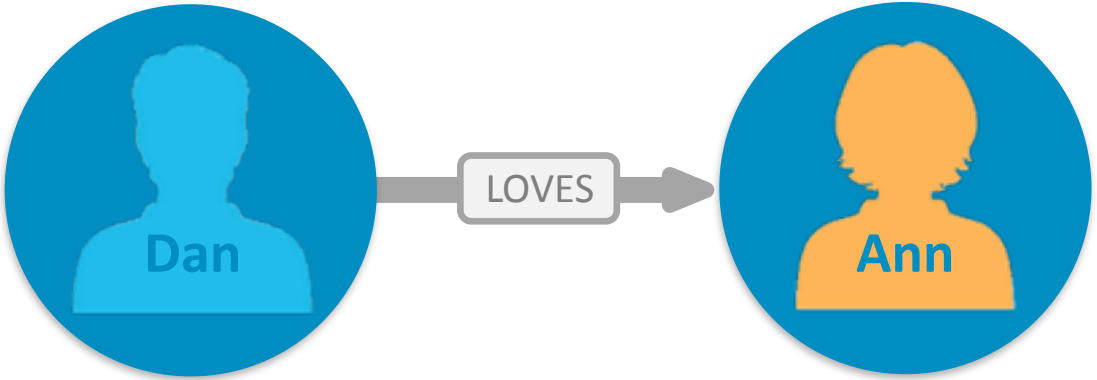



NODE — Relationship — NODE

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

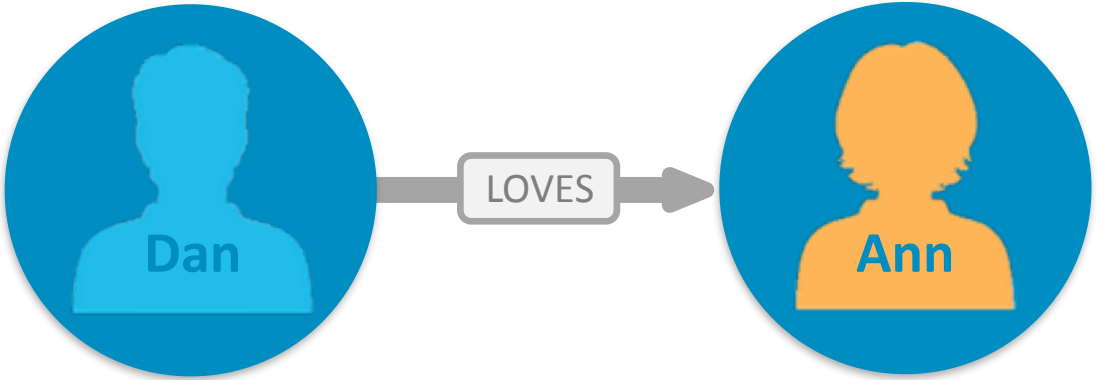
LABEL — PROPERTY — LABEL — PROPERTY

# Cypher: CREATE Graph Patterns




Relationship

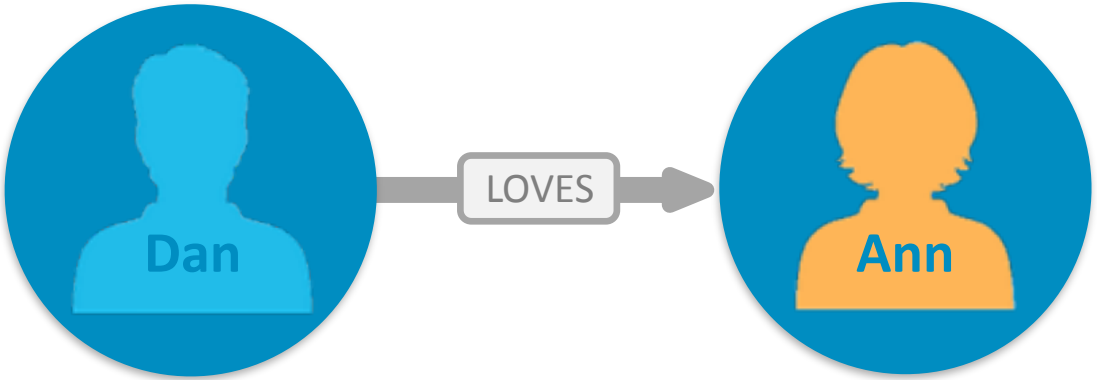
# Cypher: CREATE Graph Patterns




Relationship

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

# Cypher: CREATE Graph Patterns

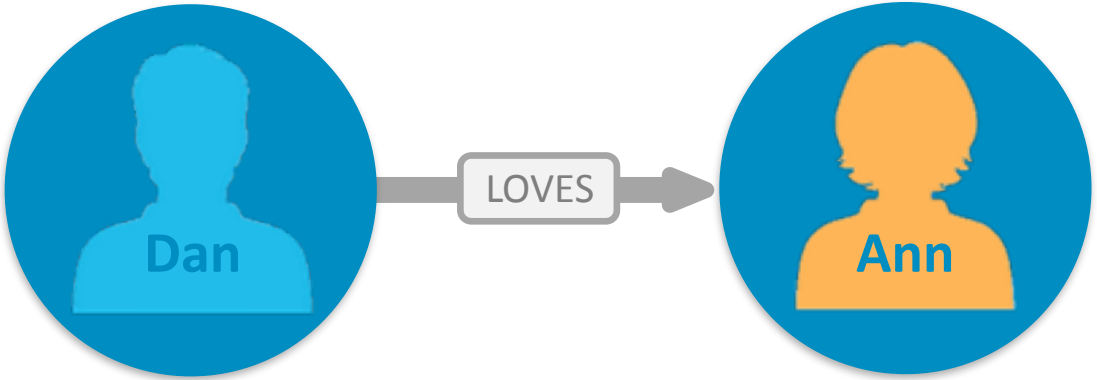



Relationship

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

LABEL PROPERTY LABEL PROPERTY

# Cypher: CREATE Graph Patterns

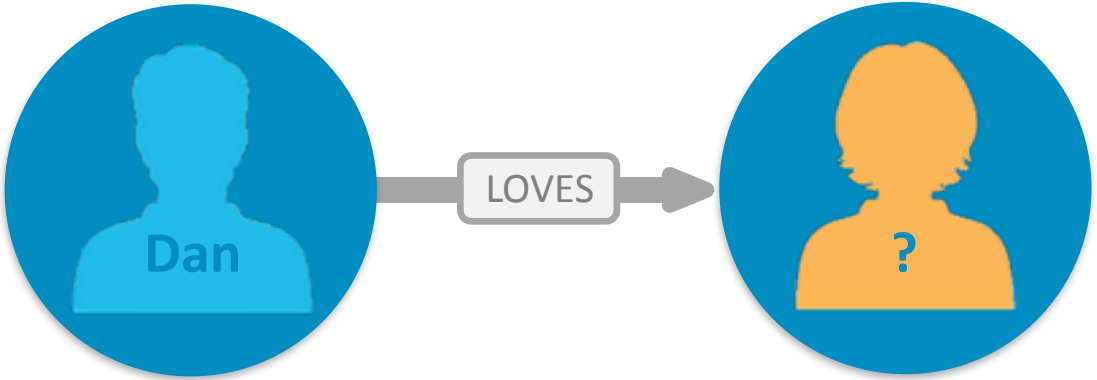



NODE — Relationship — NODE

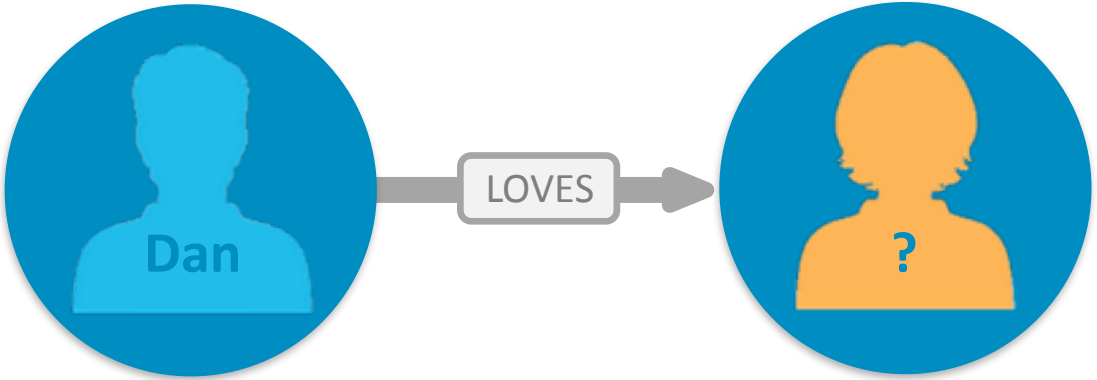
```
CREATE (:Person { name:"Dan"} ) -[:LOVES]-> (:Person { name:"Ann"} )
```

LABEL PROPERTY — LABEL PROPERTY

# Cypher: MATCH Graph Patterns

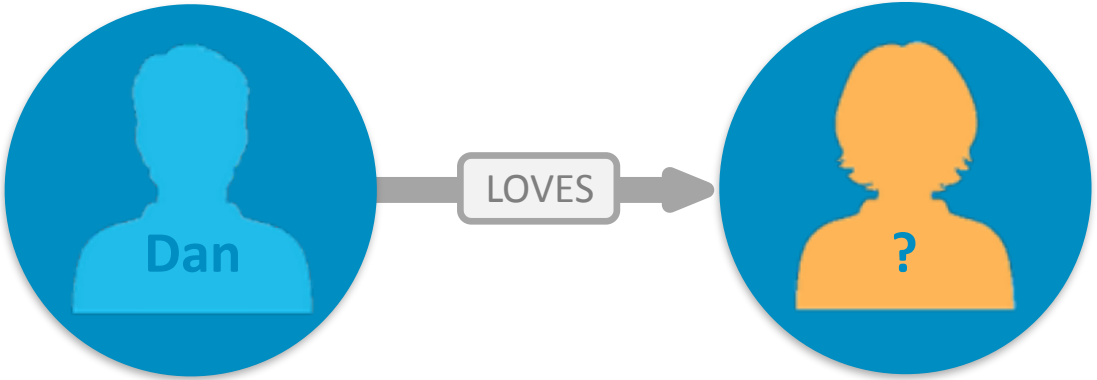


# Cypher: MATCH Graph Patterns




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

# Cypher: MATCH Graph Patterns

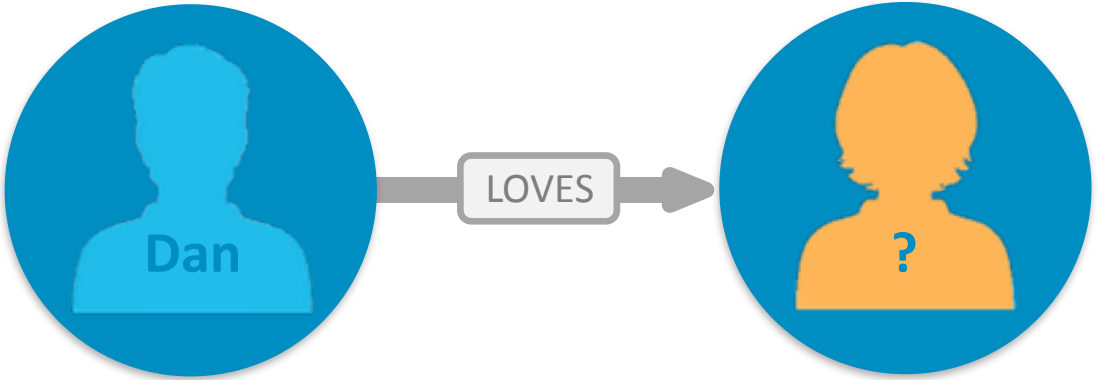



Relationship

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

LABEL PROPERTY VARIABLE

# Cypher: MATCH Graph Patterns



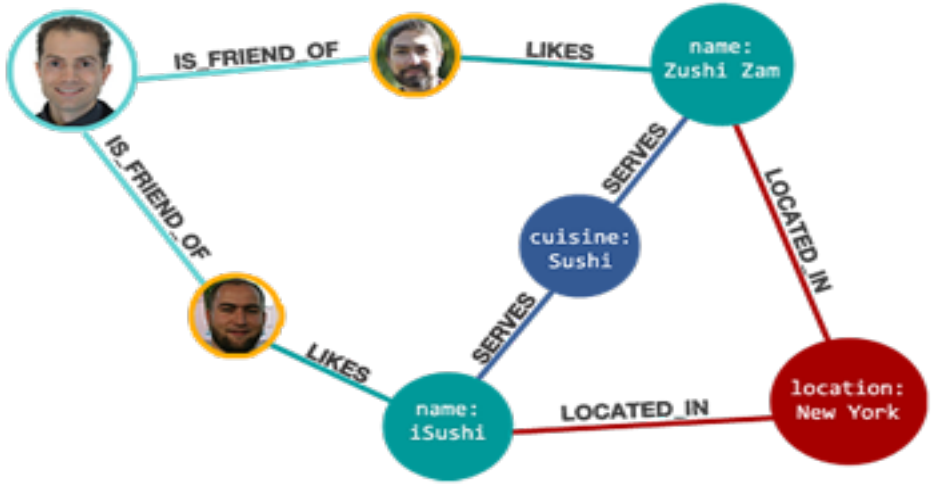

NODE — Relationship — NODE

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

LABEL — PROPERTY — VARIABLE

# A graph query example

# A social recommendation

# 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.

```
-->
-[:DIRECTED]->
```

# 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


| Case sensitive | Case insensitive |
| --- | --- |
| :Person | MaTcH |
| :ACTED_IN | return |
| name | |

# Write queries

# The CREATE Clause


```
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

```
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

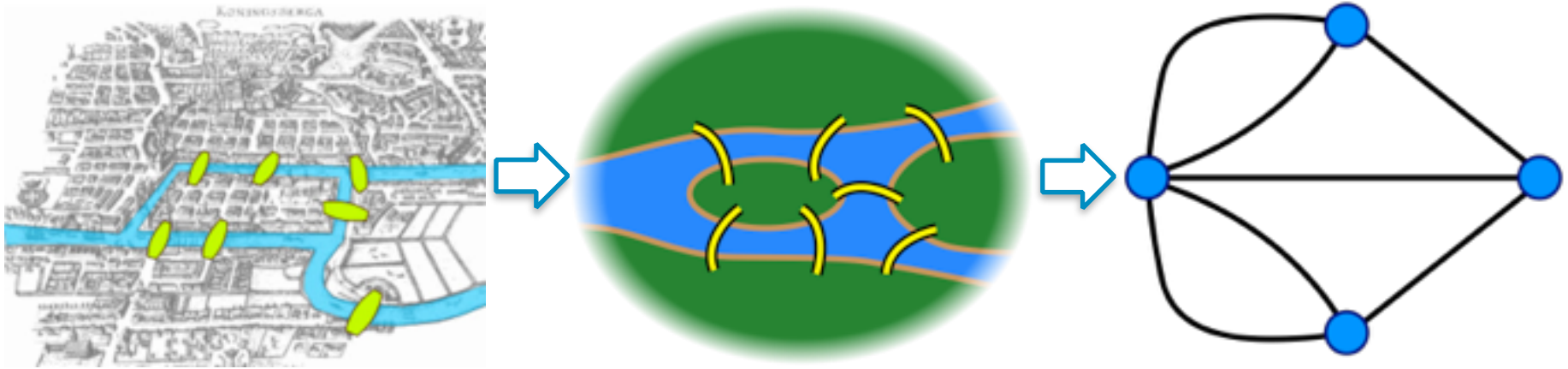

```
MERGE (p:Person {name: 'Tom Hanks'})
MERGE (m:Movie {title: 'The Terminal'})
MERGE (p)-[r:ACTED_IN]->(m)
RETURN p, r, m
```

# 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

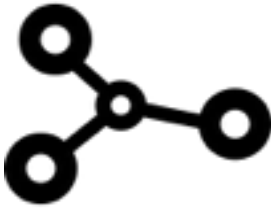
# Models



# The modeling workflow

1. Derive the question

2. Obtain the data



3. Develop a model

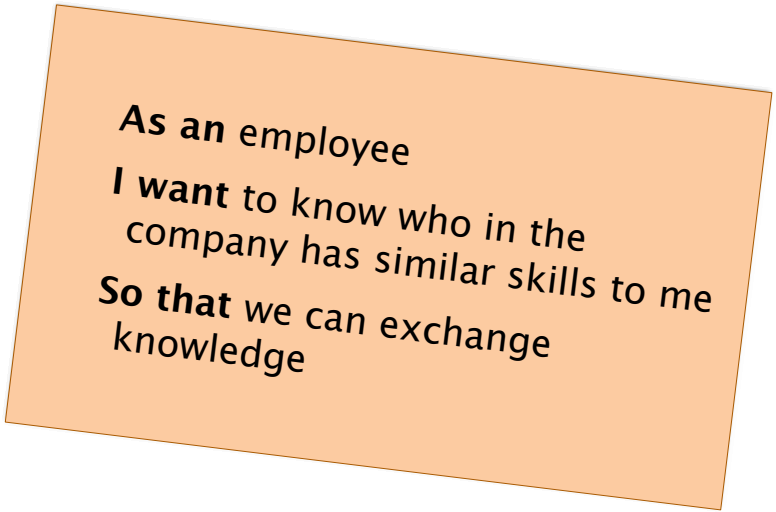
4. Ingest the data

5. Query/ Prove our model

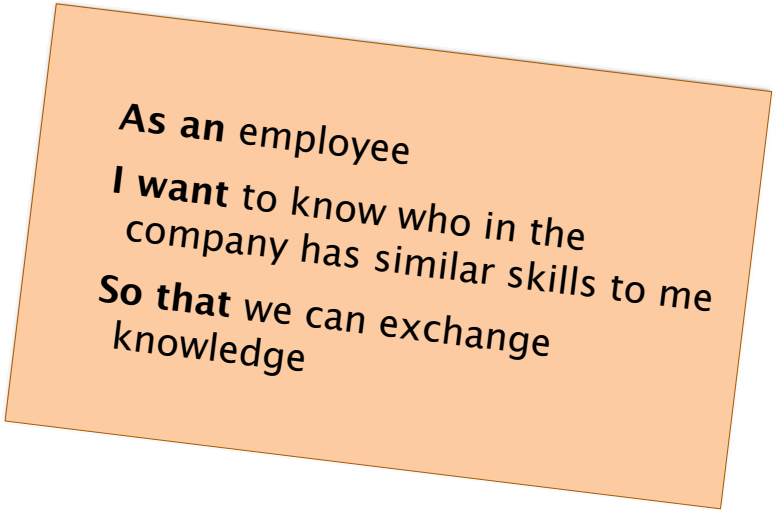
# Developing the model and the query

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



# 2. Questions to ask of the Domain




**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***

# 4. Identify Relationships Between Entities

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

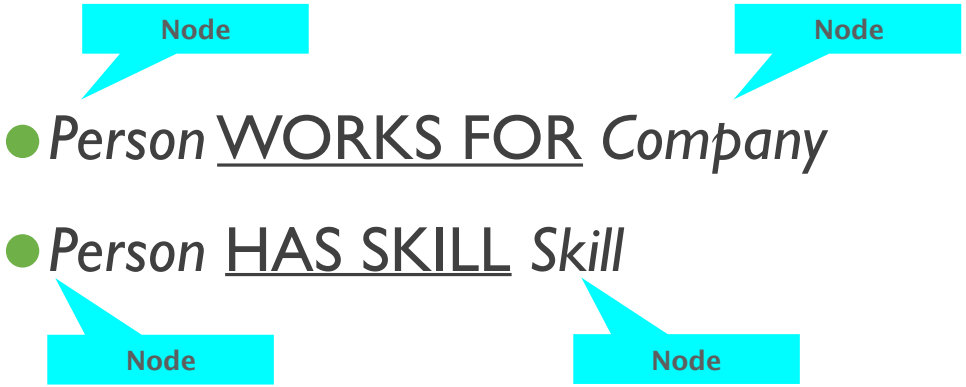
- ***Person*** WORKS FOR ***Company***
- ***Person*** HAS SKILL ***Skill***

# 5. Convert to Cypher Paths

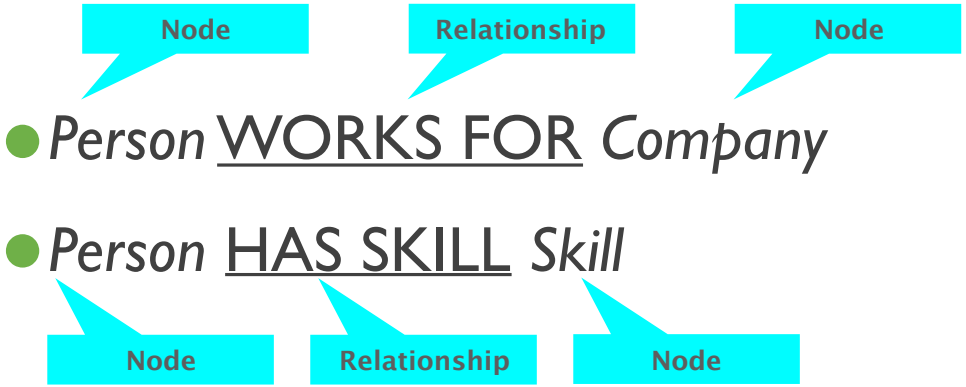
# 5. Convert to Cypher Paths

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

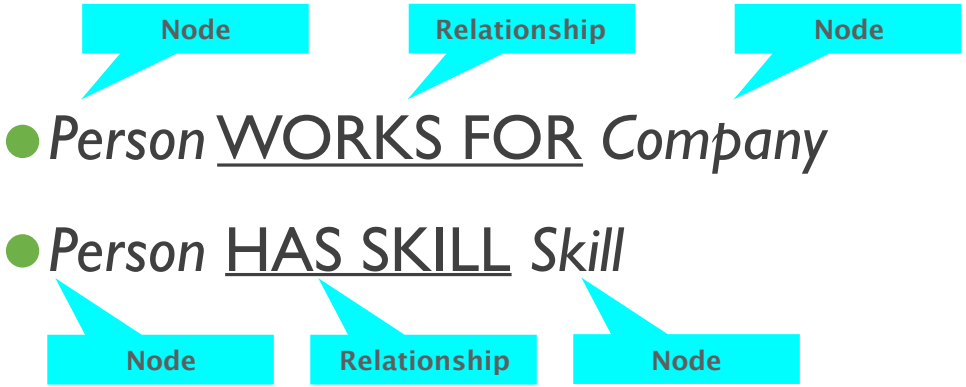
# 5. Convert to Cypher Paths



# 5. Convert to Cypher Paths

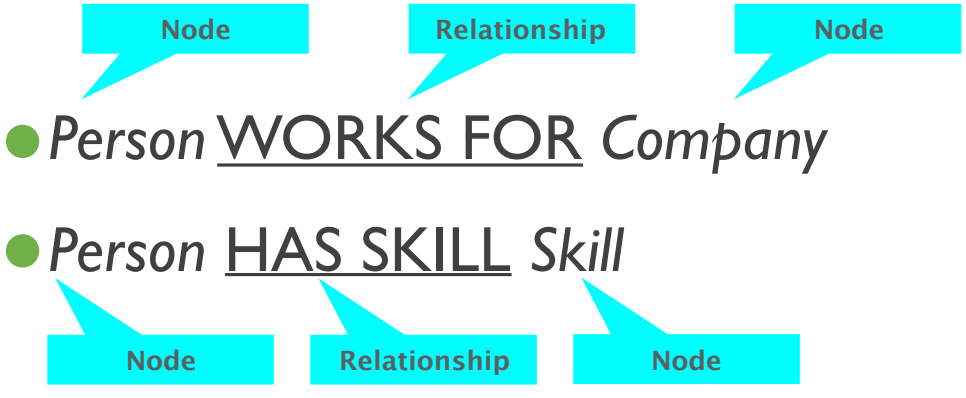


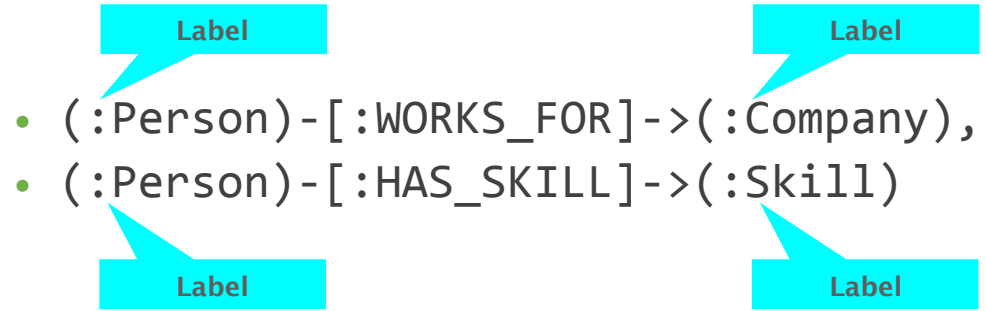
# 5. Convert to Cypher Paths



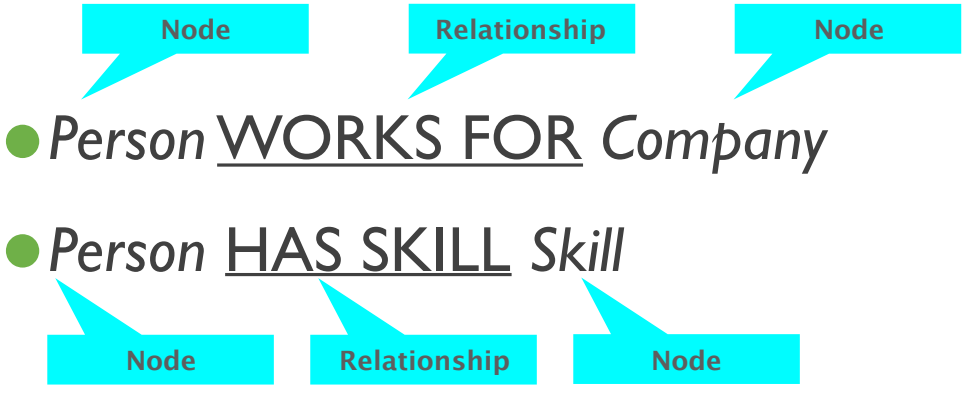

- `(:Person)-[:WORKS_FOR]->(:Company),`
- `(:Person)-[:HAS_SKILL]->(:Skill)`

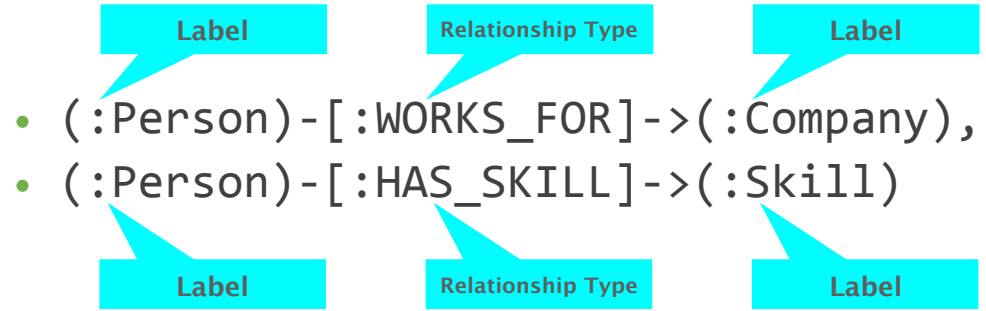
# 5. Convert to Cypher Paths



# 5. Convert to Cypher Paths

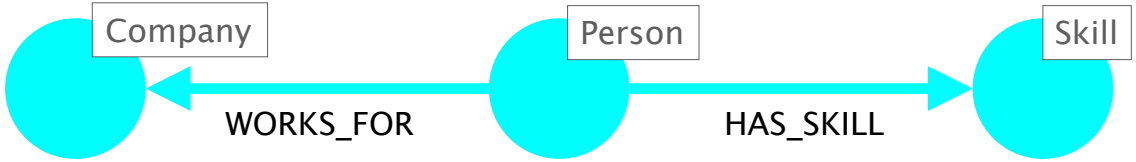


# Consolidate Pattern

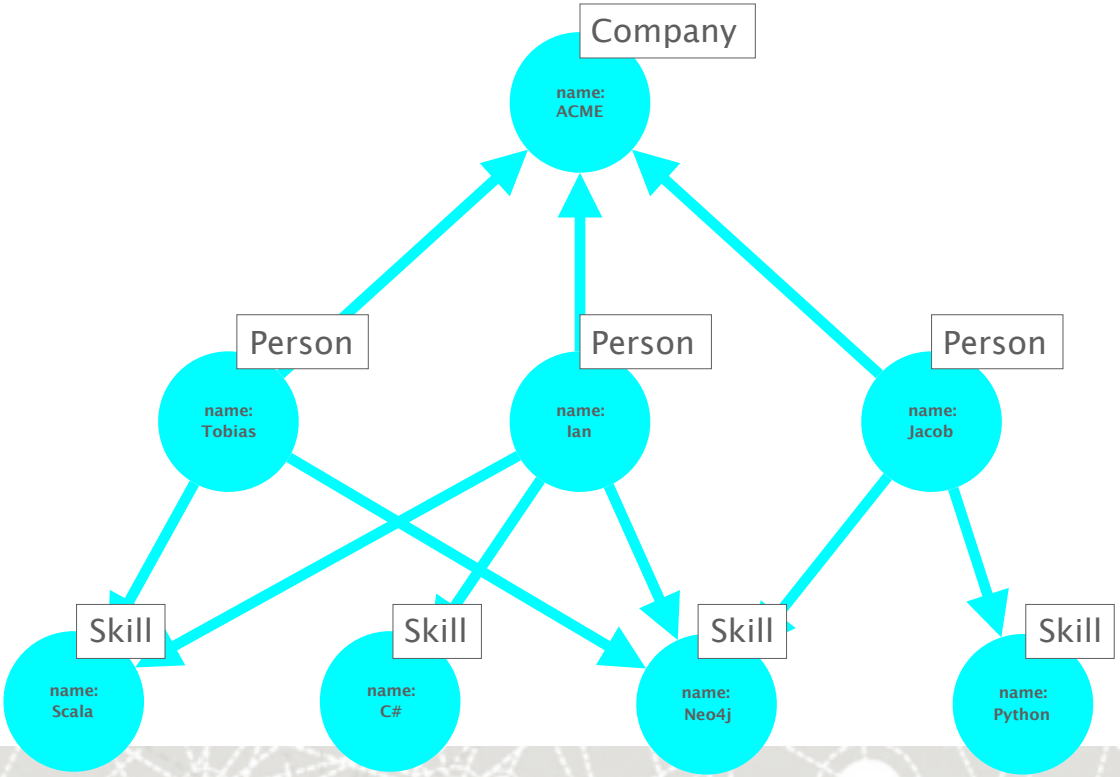
```
(:Person)-[:WORKS_FOR]->(:Company),
(:Person)-[:HAS_SKILL]->(:Skill)
```

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



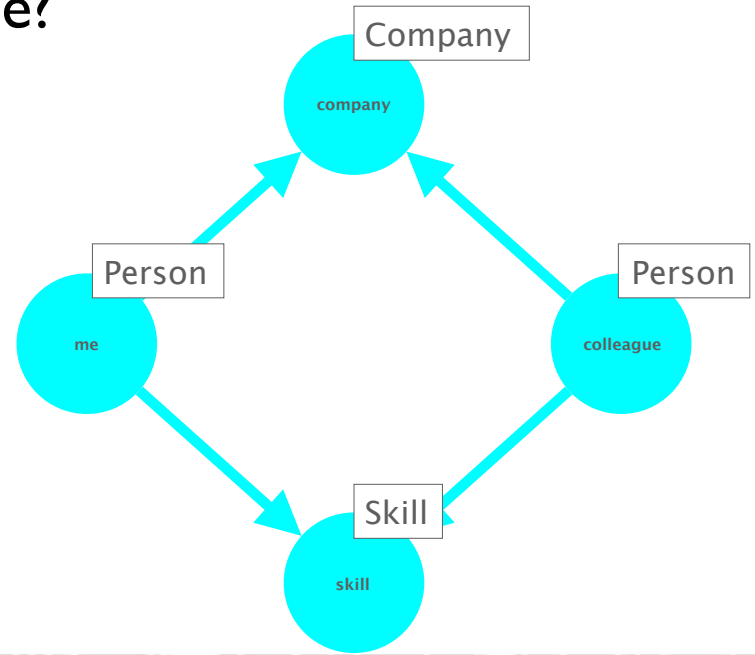
# Candidate Data Model

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



# 6. Express Question as Graph Pattern

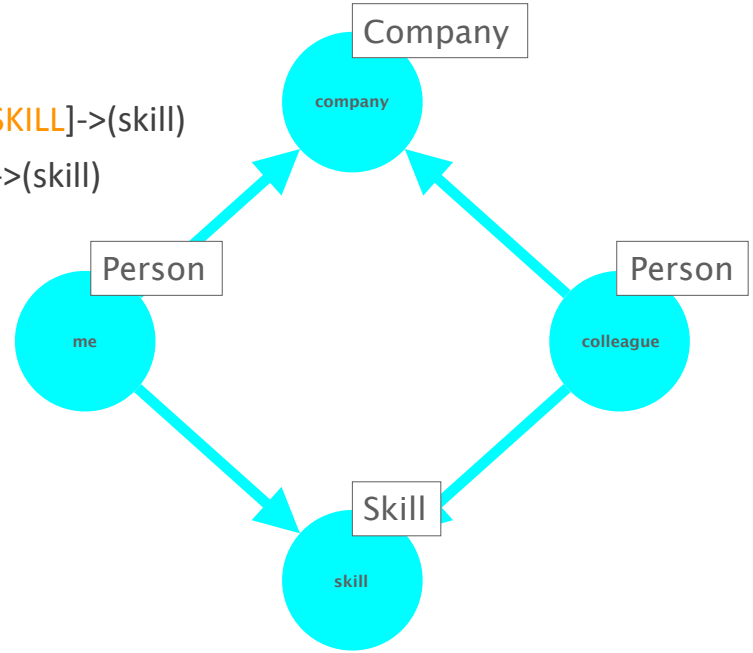
Which people, who work for the same company as me, have similar skills to me?



# 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
```

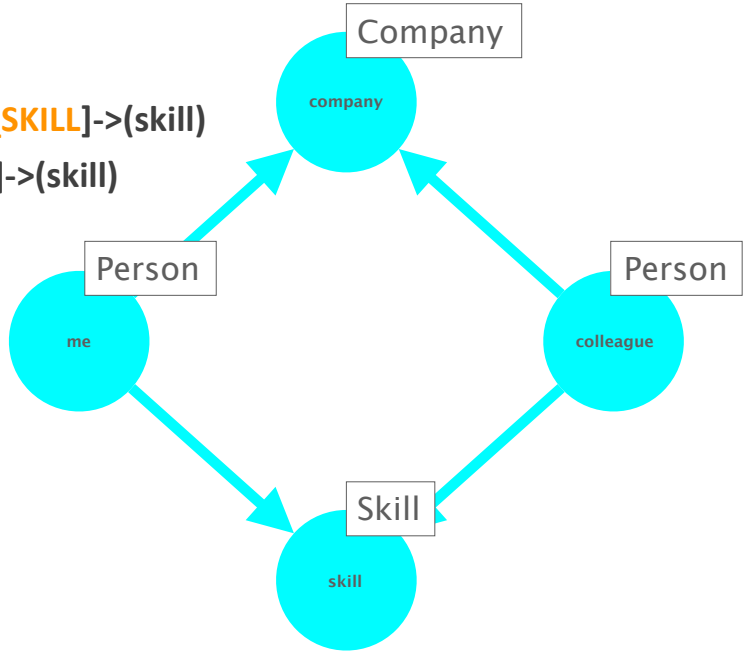


# Cypher Query

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**

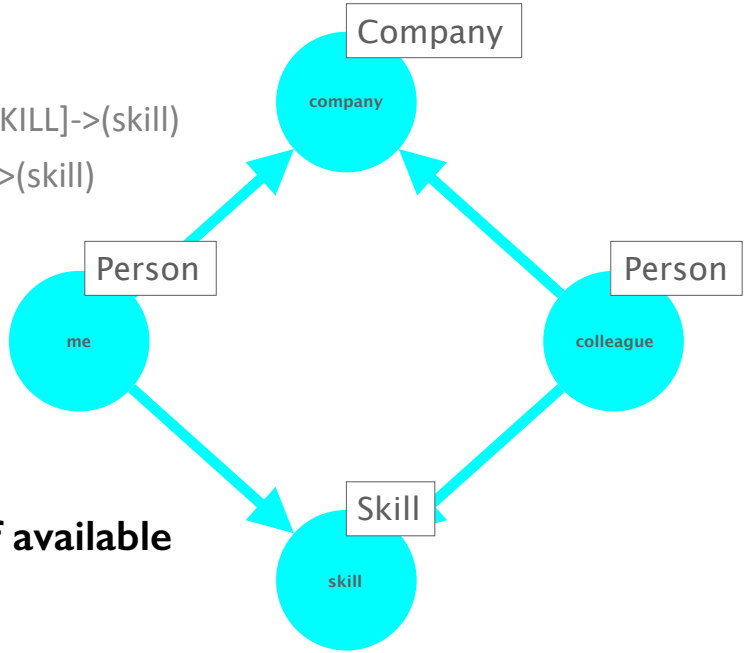


# Cypher Query

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**

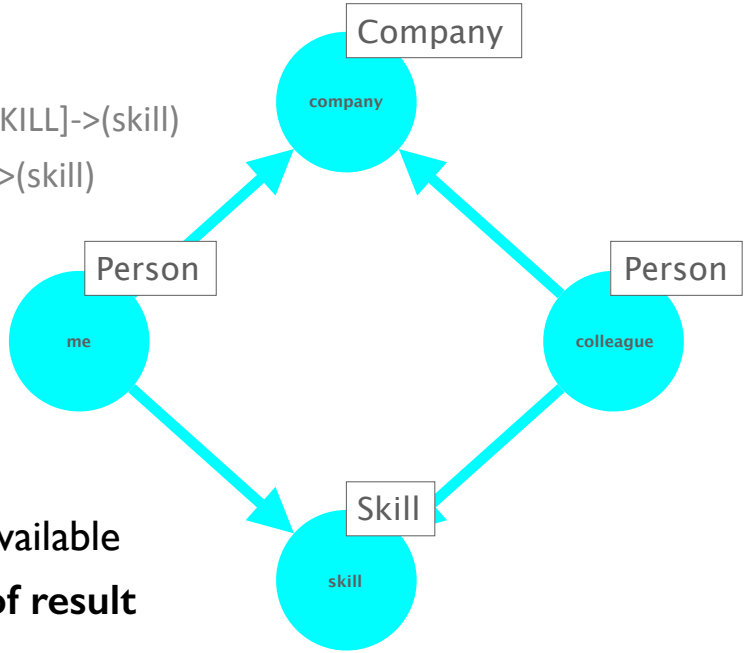


# Cypher Query

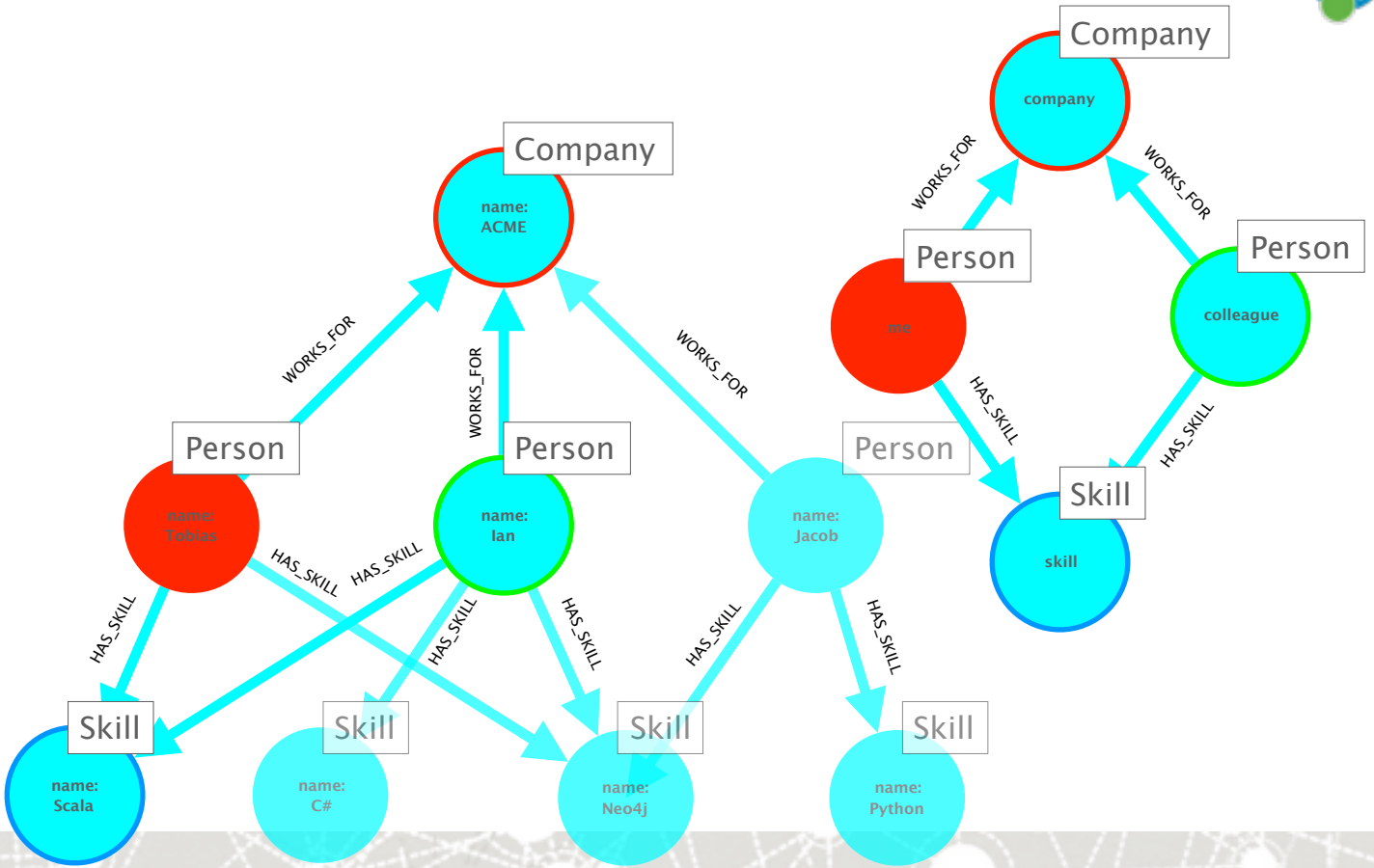
Which people, who work for the same company as me, have similar skills to me?

```
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
```

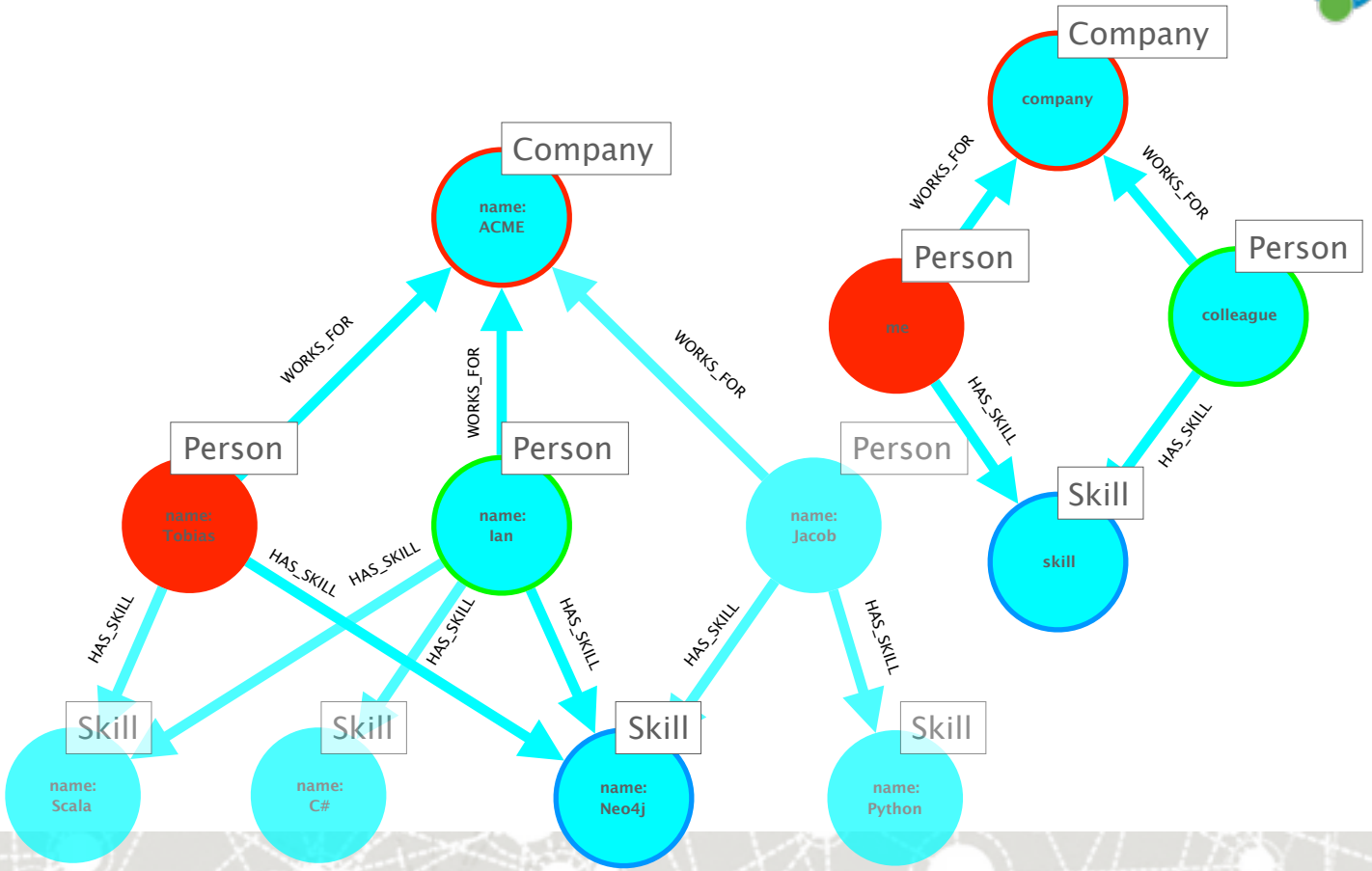
1. Graph pattern
2. Filter, using index if available
3. **Create projection of result**



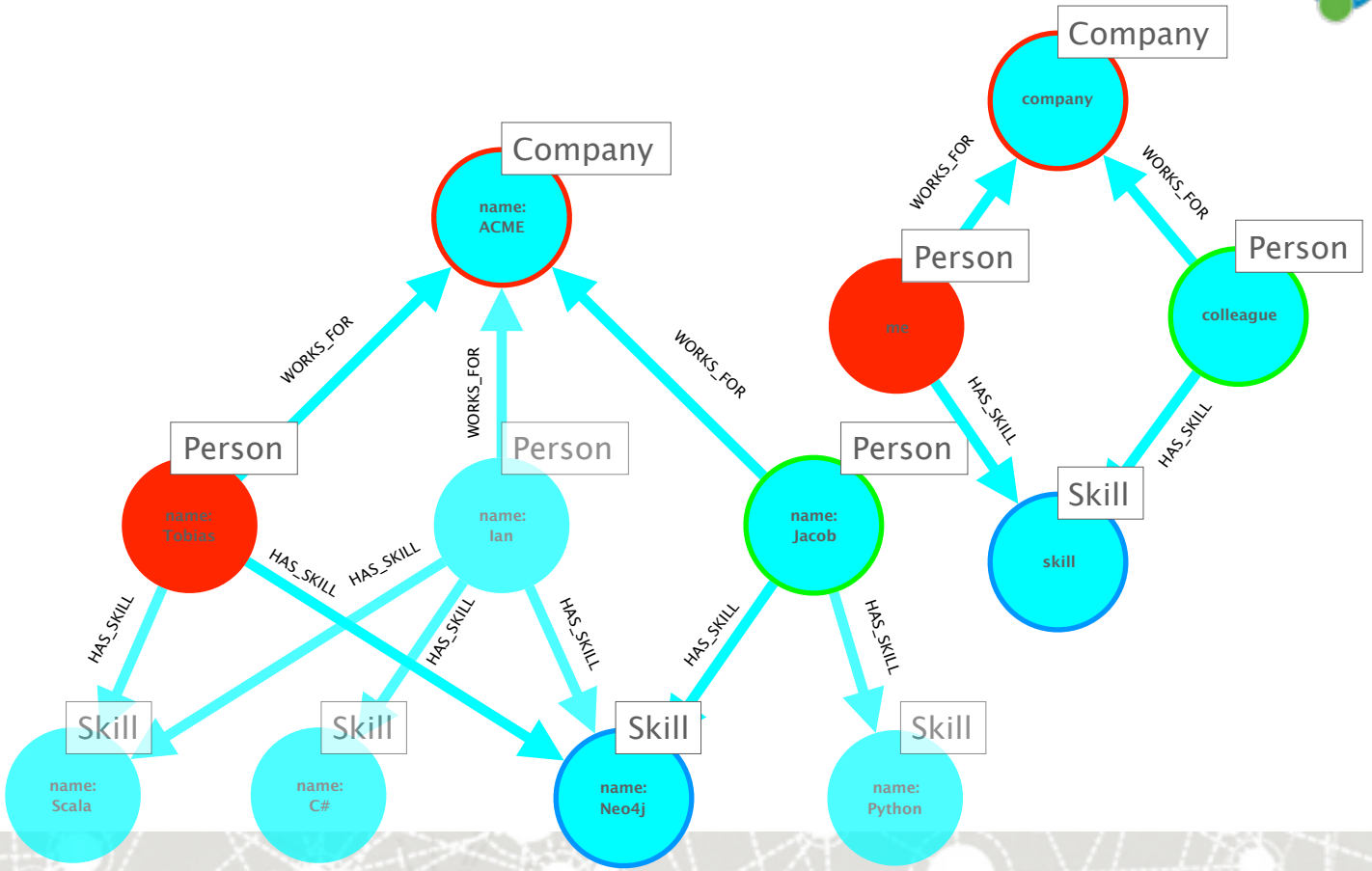
# First Match


Company
name:
ACME
WORKS_FOR
WORKS_FOR
WORKS_FOR
Person
name:
Tobias
Person
name:
Ian
Person
name:
Jacob
HAS_SKILL
HAS_SKILL
HAS_SKILL
HAS_SKILL
HAS_SKILL
HAS_SKILL
Skill
name:
Scala
Skill
name:
C#
Skill
name:
Neo4j
Skill
name:
Python
Company
company
WORKS_FOR
WORKS_FOR
Person
me
Person
colleague
HAS_SKILL
HAS_SKILL
Skill
skill

# Second Match



# Third Match


Company
name:
ACME
WORKS_FOR
WORKS_FOR
WORKS_FOR
Person
name:
Tobias
Person
name:
Ian
Person
name:
Jacob
HAS_SKILL
HAS_SKILL
HAS_SKILL
HAS_SKILL
HAS_SKILL
HAS_SKILL
Skill
name:
Scala
Skill
name:
C#
Skill
name:
Neo4j
Skill
name:
Python
Company
company
WORKS_FOR
WORKS_FOR
Person
me
Person
colleague
HAS_SKILL
HAS_SKILL
Skill
skill

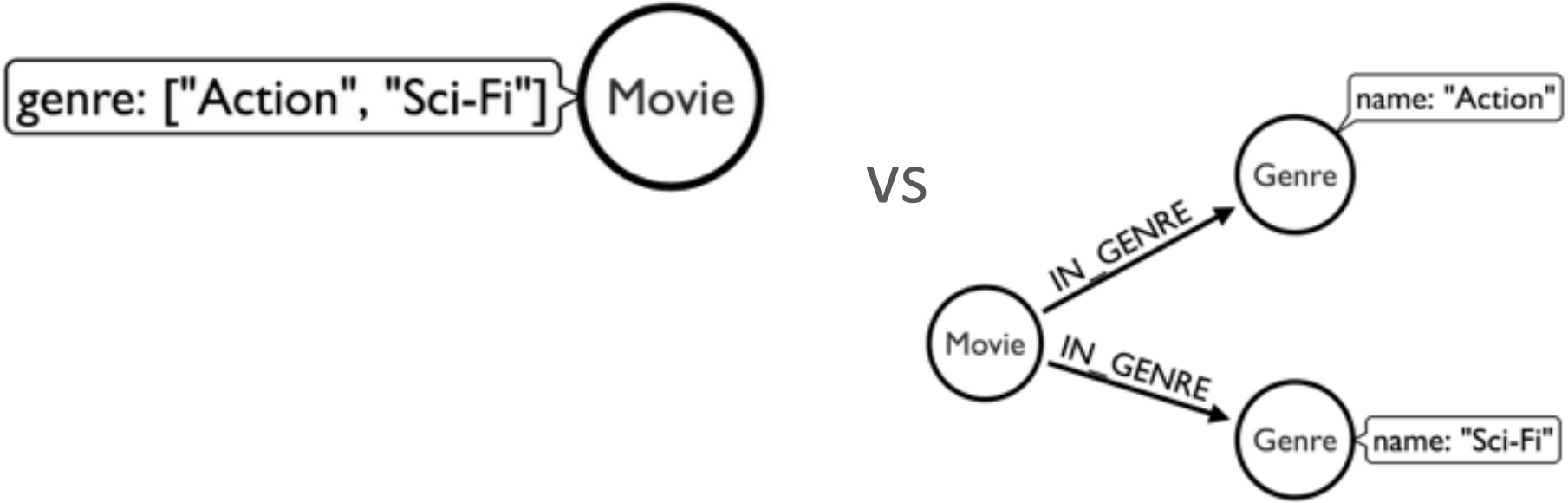
# Result of the Query

```
+-------------------------------------+
| name    | score | skills            |
+-------------------------------------+
| "Ian"   | 2     | ["Scala","Neo4j"] |
| "Jacob" | 1     | ["Neo4j"]         |
+-------------------------------------+
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;
```

# The bad side of properties

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)
```

# Genres as nodes


```
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
```

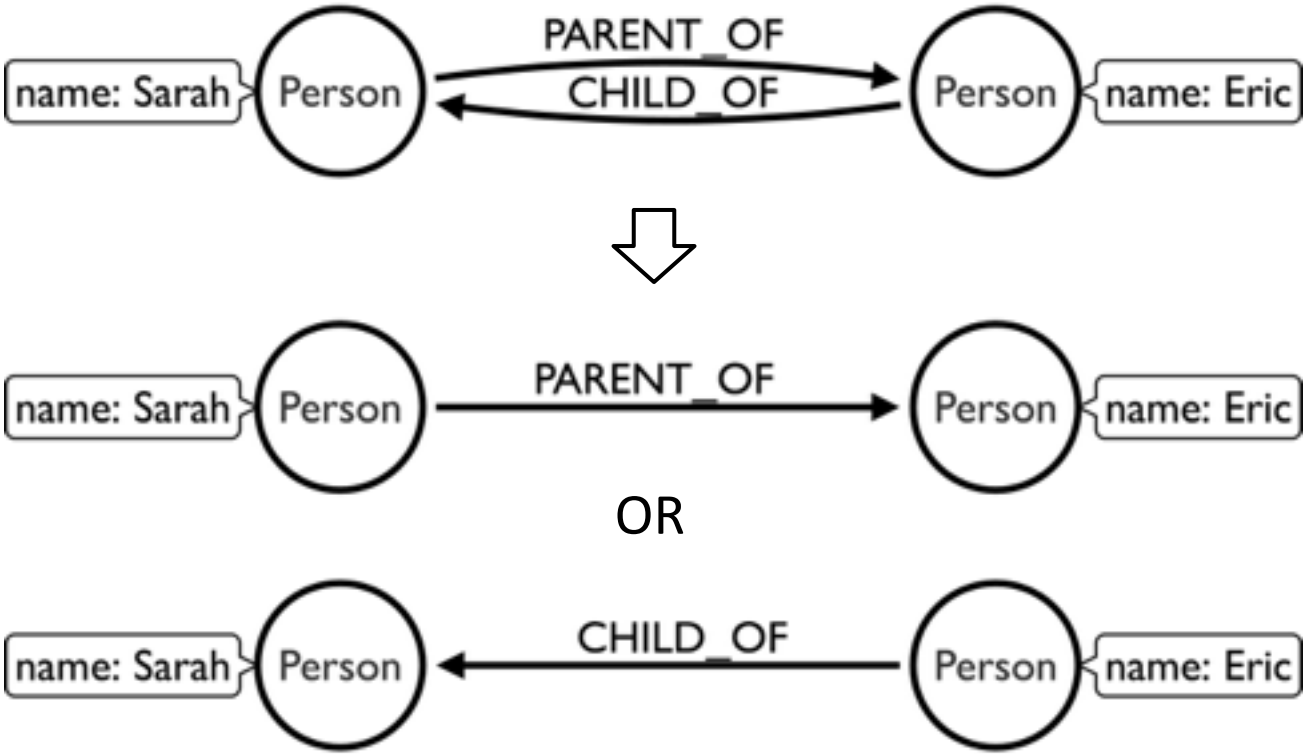
# The (not too) bad side of nodes

Accessing the genres of movies requires a bit more typing.

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

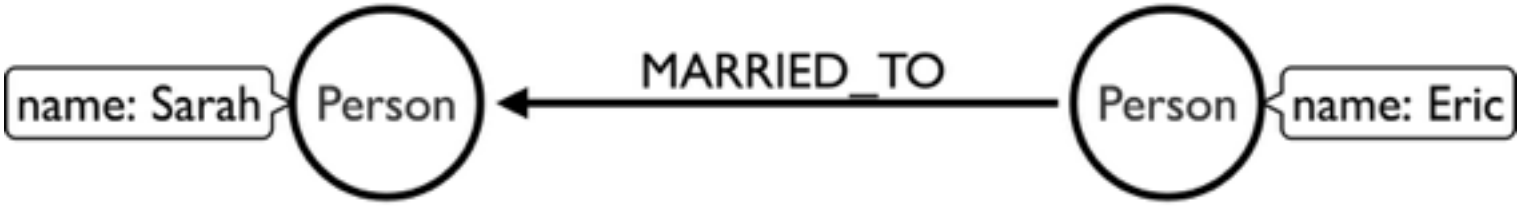
# Symmetric Relationships

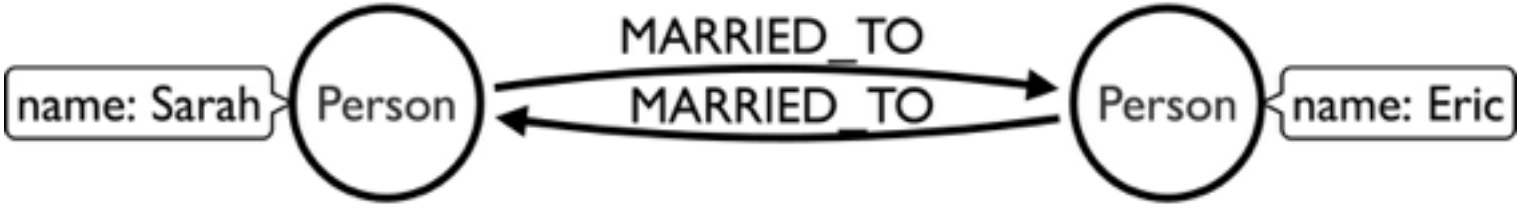
# Symmetric relationships



# Bidirectional Relationships

# Use single relationship and ignore direction in queries



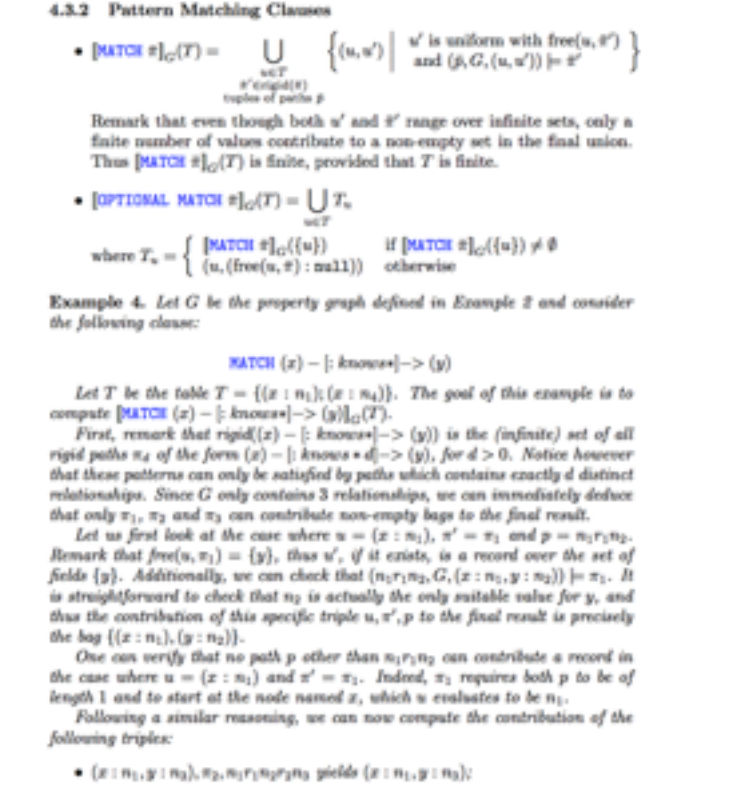




```
MATCH (:Person {name:'Eric'})-[:MARRIED_TO]-(p2)
RETURN p2
```

# Formal semantics

- Nadime Francis,
  Paolo Guagliardo,
  Leonid Libkin

- Formally defines a (large) core of Cypher



4.3.2 Pattern Matching Clauses

- $[\![\texttt{MATCH}\ \pi]\!]_G(T) = \bigcup_{u \in T} \bigcup_{\substack{\pi' \in \mathrm{rigid}(\pi) \\ \text{tuples of paths } p}} \left\{ (u, u') \;\middle|\; \begin{array}{l} u' \text{ is uniform with } \mathrm{free}(u, \pi') \\ \text{and } (p, G, (u, u')) \models \pi' \end{array} \right\}$

  Remark that even though both $u'$ and $\pi'$ range over infinite sets, only a finite number of values contribute to a non-empty set in the final union. Thus $[\![\texttt{MATCH}\ \pi]\!]_G(T)$ is finite, provided that $T$ is finite.

- $[\![\texttt{OPTIONAL MATCH}\ \pi]\!]_G(T) = \bigcup_{u \in T} T_u$

  where $T_u = \begin{cases} [\![\texttt{MATCH}\ \pi]\!]_G(\{u\}) & \text{if } [\![\texttt{MATCH}\ \pi]\!]_G(\{u\}) \neq \emptyset \\ (u, \{\mathrm{free}(u, \pi) : \mathtt{null}\}) & \text{otherwise} \end{cases}$

**Example 4.** *Let $G$ be the property graph defined in Example 2 and consider the following clause:*

$$\texttt{MATCH}\ (x) -[:\mathit{knows}*]-> (y)$$

*Let $T$ be the table $T = \{(x : n_1); (x : n_4)\}$. The goal of this example is to compute $[\![\texttt{MATCH}\ (x) -[:\mathit{knows}*]-> (y)]\!]_G(T)$.*

*First, remark that $\mathrm{rigid}((x) -[:\mathit{knows}*]-> (y))$ is the (infinite) set of all rigid paths $\pi_d$ of the form $(x) -[:\mathit{knows} * d]-> (y)$, for $d \geq 0$. Notice however that these patterns can only be satisfied by paths which contains exactly $d$ distinct relationships. Since $G$ only contains 3 relationships, we can immediately deduce that only $\pi_1$, $\pi_2$ and $\pi_3$ can contribute non-empty bags to the final result.*

*Let us first look at the case where $u = (x : n_1)$, $\pi' = \pi_1$ and $p = n_1 r_1 n_2$. Remark that $\mathrm{free}(u, \pi_1) = \{y\}$, thus $u'$, if it exists, is a record over the set of fields $\{y\}$. Additionally, we can check that $(n_1 r_1 n_2, G, (x : n_1, y : n_2)) \models \pi_1$. It is straightforward to check that $n_2$ is actually the only suitable value for $y$, and thus the contribution of this specific triple $u, \pi', p$ to the final result is precisely the bag $\{(x : n_1), (y : n_2)\}$.*

*One can verify that no path $p$ other than $n_1 r_1 n_2$ can contribute a record in the case where $u = (x : n_1)$ and $\pi' = \pi_1$. Indeed, $\pi_1$ requires both $p$ to be of length 1 and to start at the node named $x$, which $u$ evaluates to be $n_1$.*

*Following a similar reasoning, we can now compute the contribution of the following triples:*

- $(x : n_1, y : n_3), \pi_2, n_1 r_1 n_2 r_2 n_3$ *yields* $(x : n_1, y : n_3)$;

# 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

# Unlocking Value from Your Data Relationships

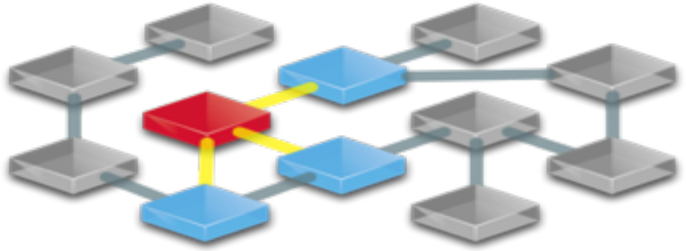
- Model your data **naturally** as a graph of data and relationships
- Drive graph model **from domain and use-cases**
- Use relationship information in **real-time** to transform your business
- Add new relationships **on the fly** to adapt to your changing requirements

# High Query Performance with a Native Graph DB

- 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



| | # persons | query time |
|---|---|---|
| Relational database | 1,000 | 2000ms |
| Neo4j | 1,000 | 2ms |
| Neo4j | 1,000,000 | 2ms |

# Getting started with Neo4j

1. Download Neo4j: http://neo4j.com/download/
2. Start the server.
3. It should be running on: `http://localhost:7474`
4. Log-in with default credentials
   user: **neo4j**
   password: **neo4j**
5. Choose a **new** password

**We're good to go!**