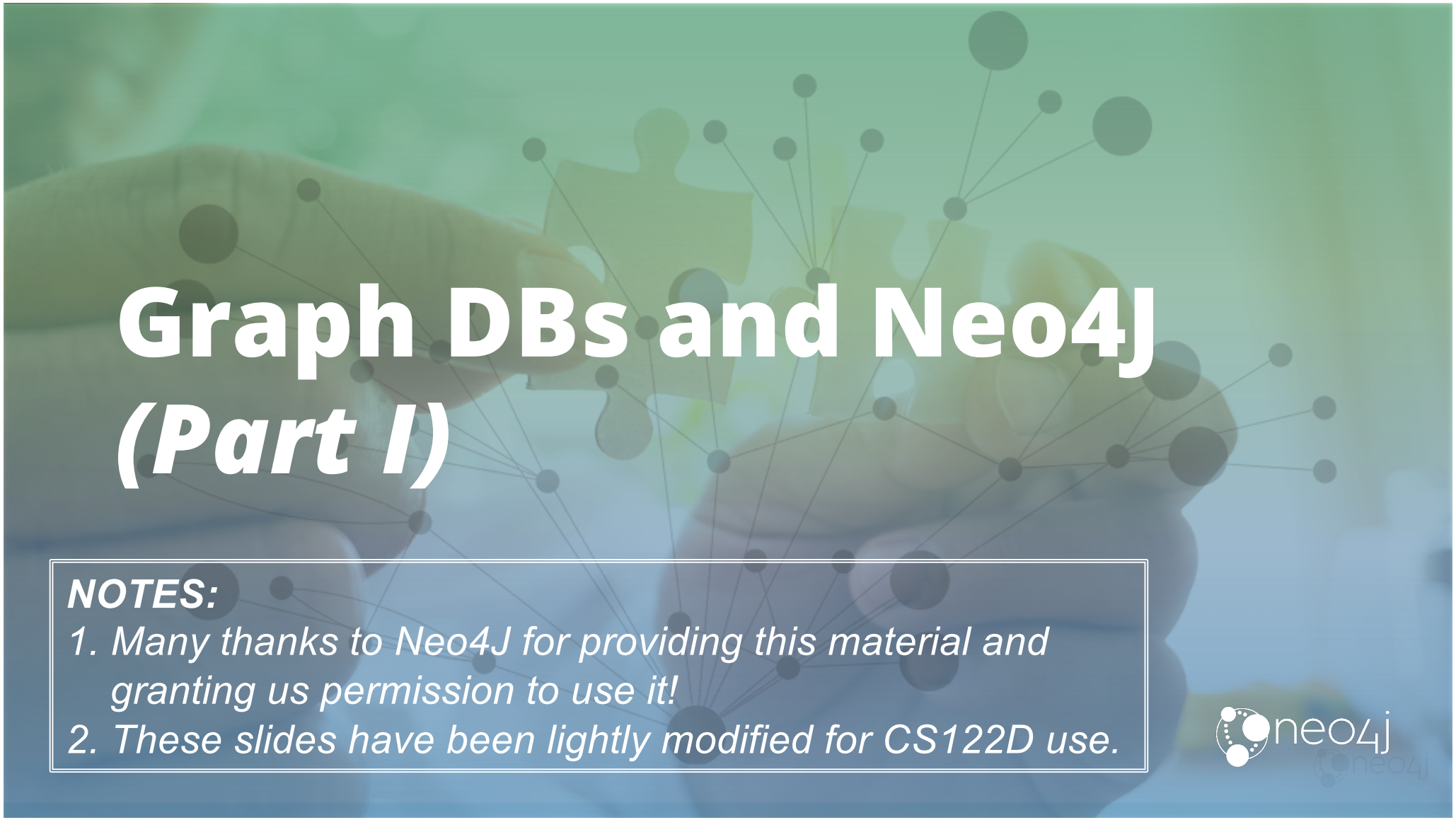
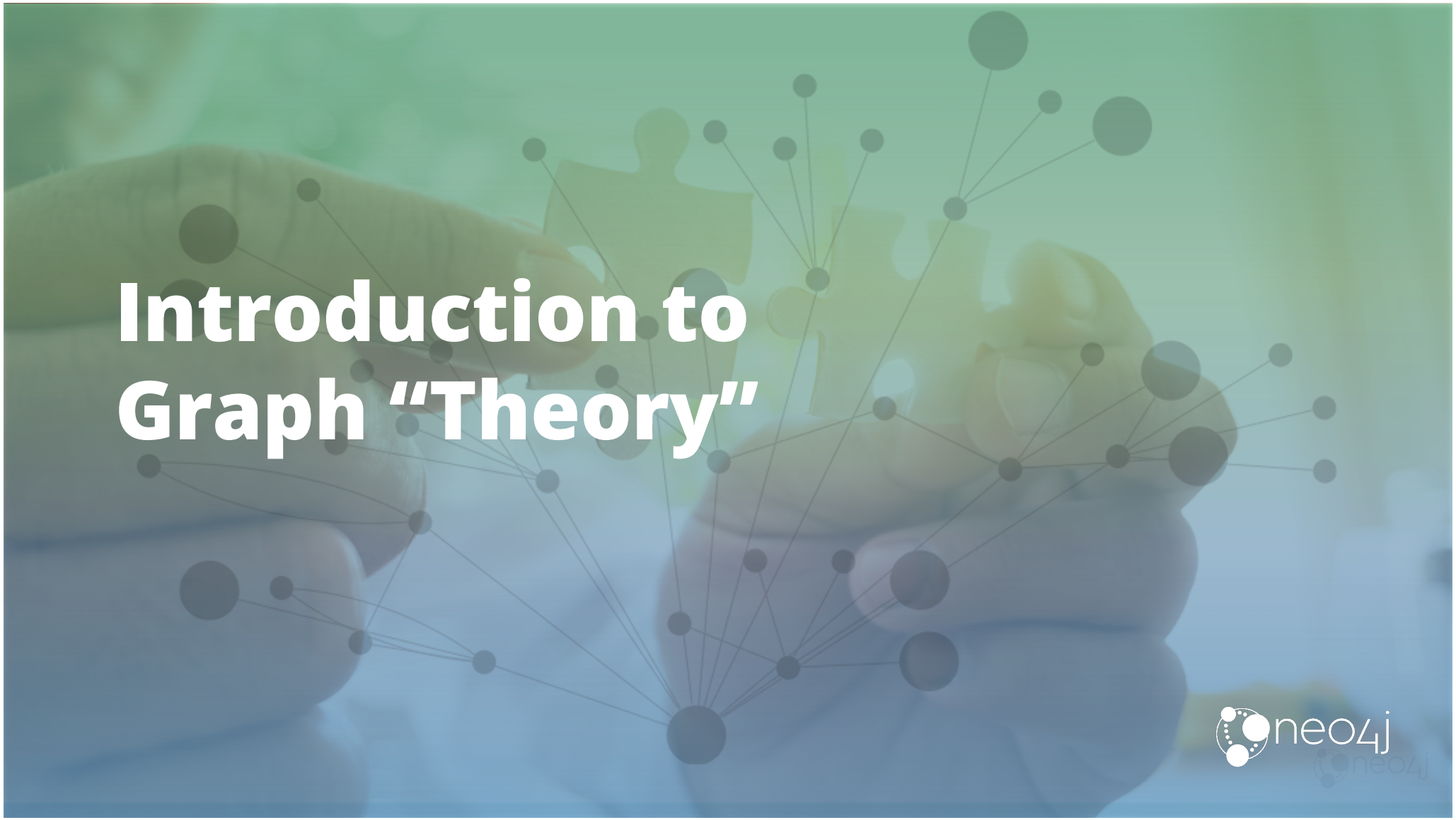


Graph DBs and Neo4J *(Part I)*

NOTES:

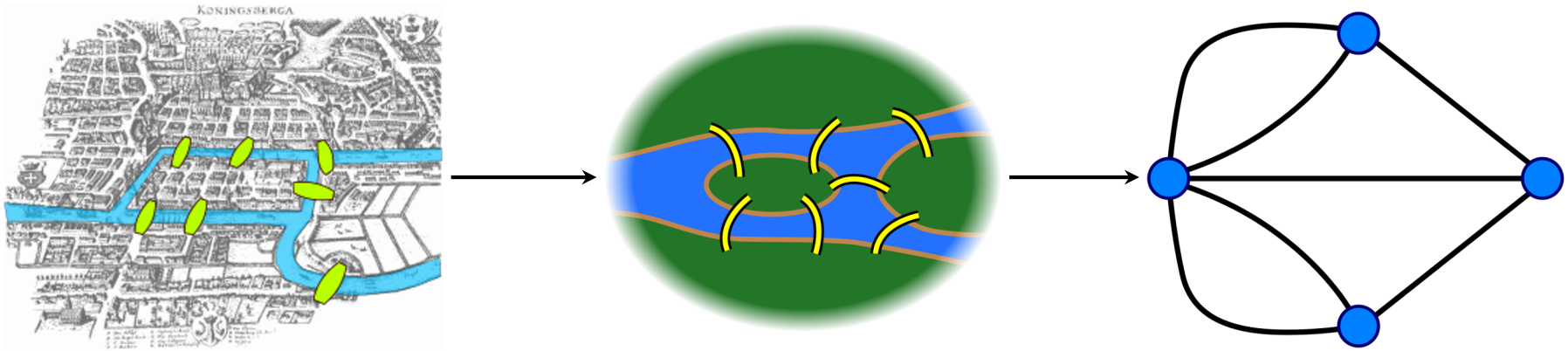
1. *Many thanks to Neo4J for providing this material and granting us permission to use it!*
2. *These slides have been lightly modified for CS122D use.*

neo4j


Introduction to
Graph “Theory”
neo4j

What is a Graph?

A **graph** is set of discrete objects, each of which has some set of relationships with the other objects

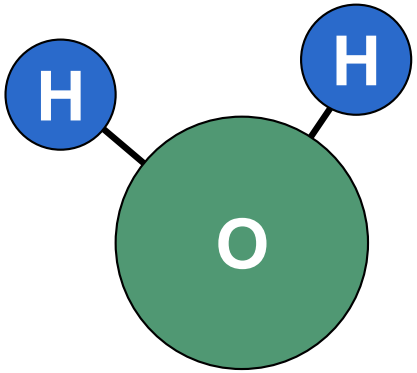


Seven Bridges of Konigsberg problem. Leonhard Euler, 1735

Anything can be a graph

the Internet

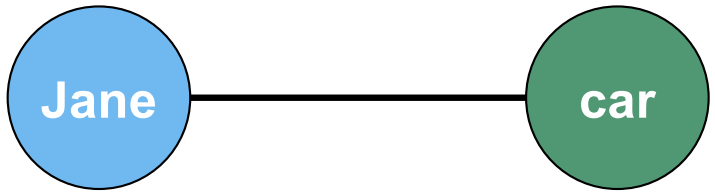
a water molecule



Graph Components

Node (Vertex)

- The main data element from which graphs are constructed
  A node without relationships is permitted. A relationship without nodes is not.
- A waypoint along a traversal route

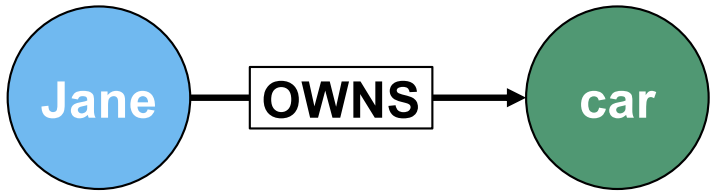


Relationship (Edge)

Graph Components

Node (Vertex)

- The main data element from which graphs are constructed
  A node without relationships is permitted. A relationship without nodes is not.
- A waypoint along a traversal route

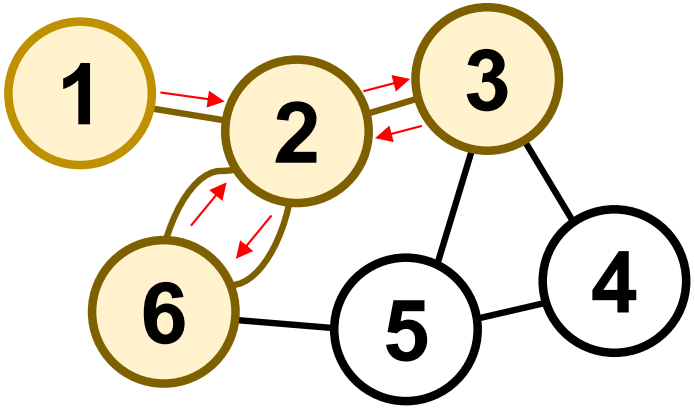


Relationship (Edge)

- A link between two nodes. May contain:
  - Direction
  - Metadata; e.g. *weight* or *relationship type*

Traversal

- **Walk**: an ordered, alternating sequence of nodes and relationships
  Nodes and relationships can be repeated

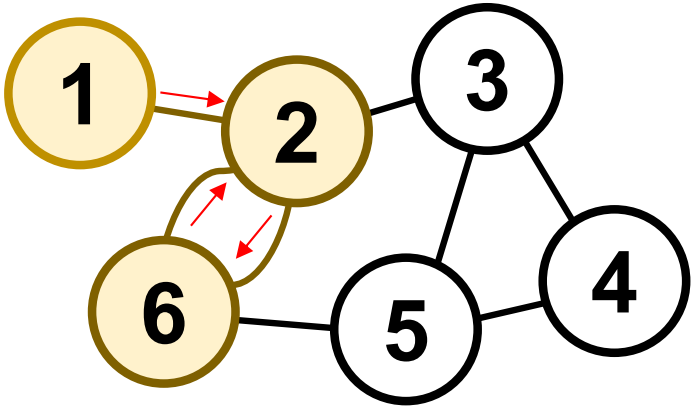


1-2-3-2-6-2

Traversal

- **Walk**: an ordered, alternating sequence of nodes and relationships
  Nodes and relationships can be repeated

- **Trail**: A walk in which no relationships are repeated
  Nodes can be repeated



1-2-3-2-6-2

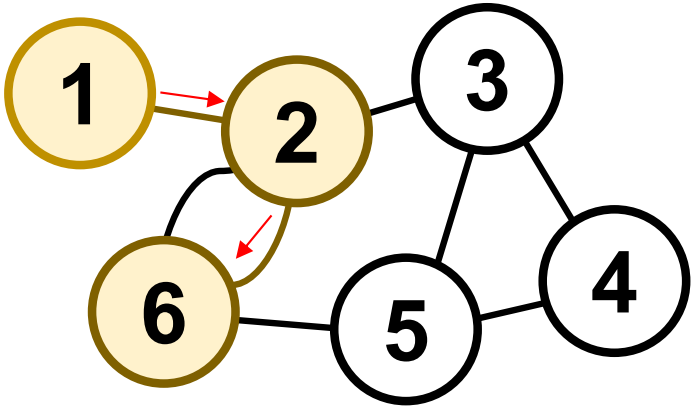
1-2-6-2

Traversal

- **Walk**: an ordered, alternating sequence of nodes and relationships
  Nodes and relationships can be repeated

- **Trail**: A walk in which no relationships are repeated
  Nodes can be repeated

- **Path**: A trail in which no nodes are repeated
  I.e., a walk where all items are unique



1-2-3-2-6-2

1-2-6-2

1-2-6

Summary

- A **graph** is a data model consisting of a set of **nodes** linked by **relationships**
- The purpose of modeling data as a graph is to enable **traversal**
- There are multiple traversal types, the most important of which is a **path**
  an ordered, alternating set of nodes and relationships
  in which all items are unique

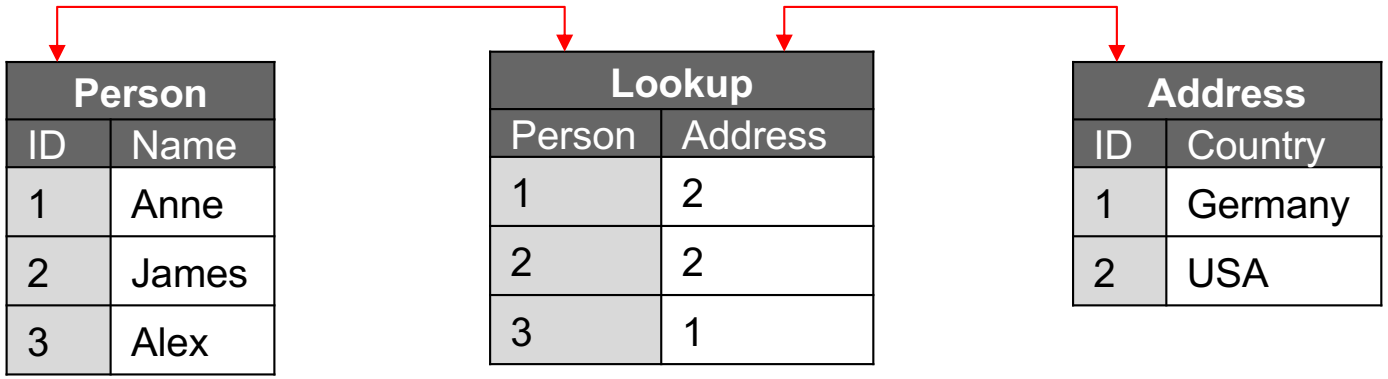

"Index-Free Adjacency”
neo4j

Relationships in RDBMS

122D warning: *A very simplistic claim! It ignores **numerous** technical questions and OODB era lessons and systems...* 😐

- Require foreign keys, and possibly a lookup table
- Traversing a foreign key requires an index lookup

The purpose of graphs is to do rapid traversal. The RDBMS model is too expensive for that.



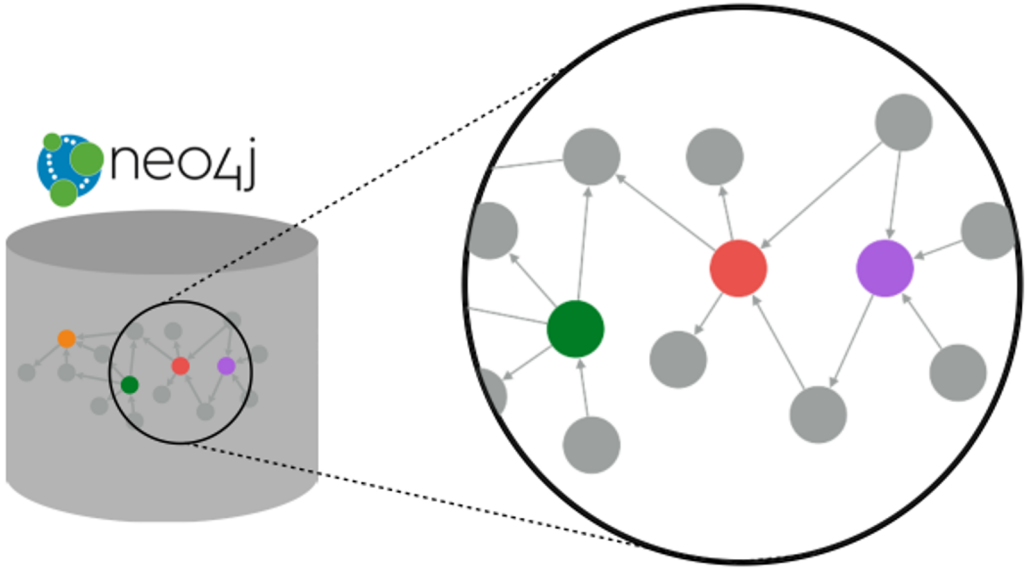
Person	
ID	Name
1	Anne
2	James
3	Alex

Lookup	
Person	Address
1	2
2	2
3	1

Address	
ID	Country
1	Germany
2	USA

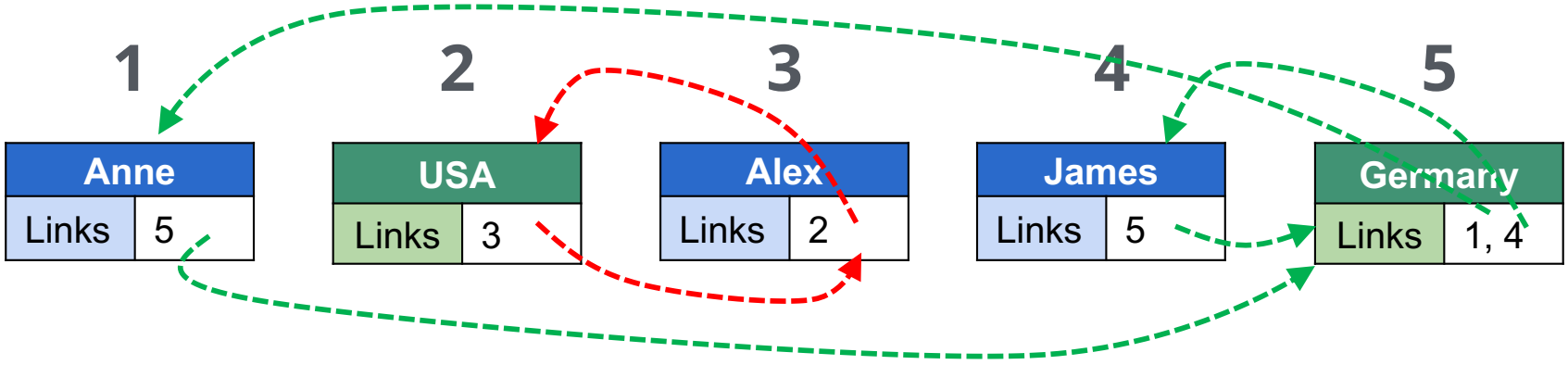
Neo4j Database: Index-free adjacency

Nodes and relationships are stored on disk as a graph for fast navigational access using **pointers**.



IFA Benefits

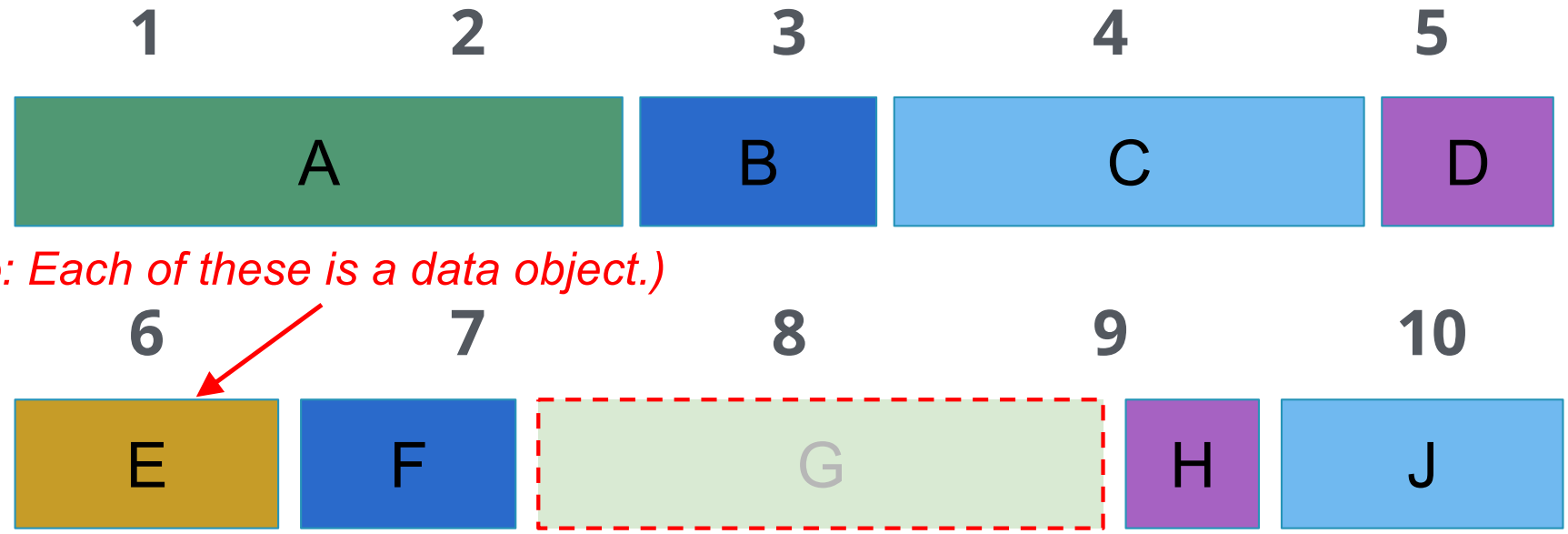
IFA uses **pointers** to the target address



- *No index lookups or table scans*
- *Reduced duplication of foreign keys*

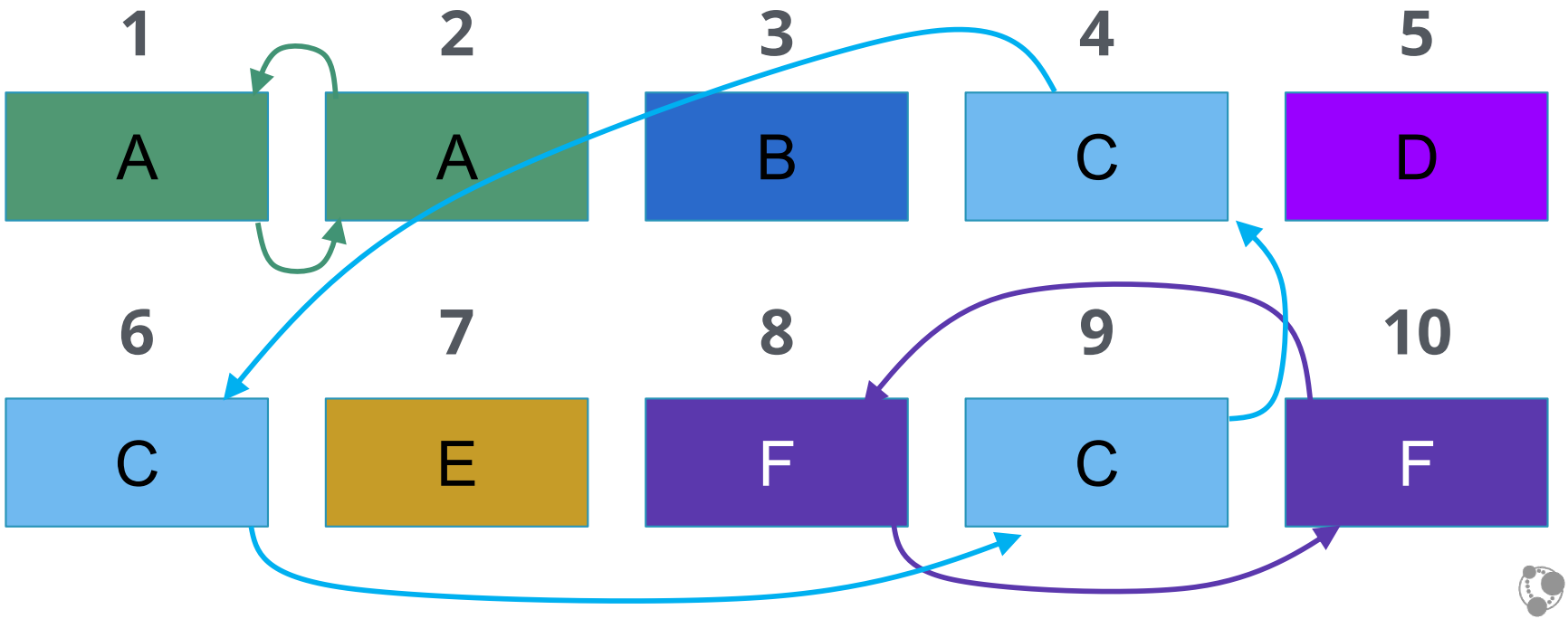
IFA Question: Free Space Management

How to avoid **wasted space** on deletion and insertion?



IFA Solution: Chunking

All data objects are chunked into **same-size** pieces



IFA Solution: Chunking

- Related chunks need not be adjacent
  - *e.g., a four-chunk object can use the first four open locations, no matter where they are*
- Related chunks reference one another via IFA

Effect: scanning for a value within a large data item can involve many "hidden" hops

Corollary: it is frequently cheaper to perform multiple relationship hops rather than scan for a value within a single large object

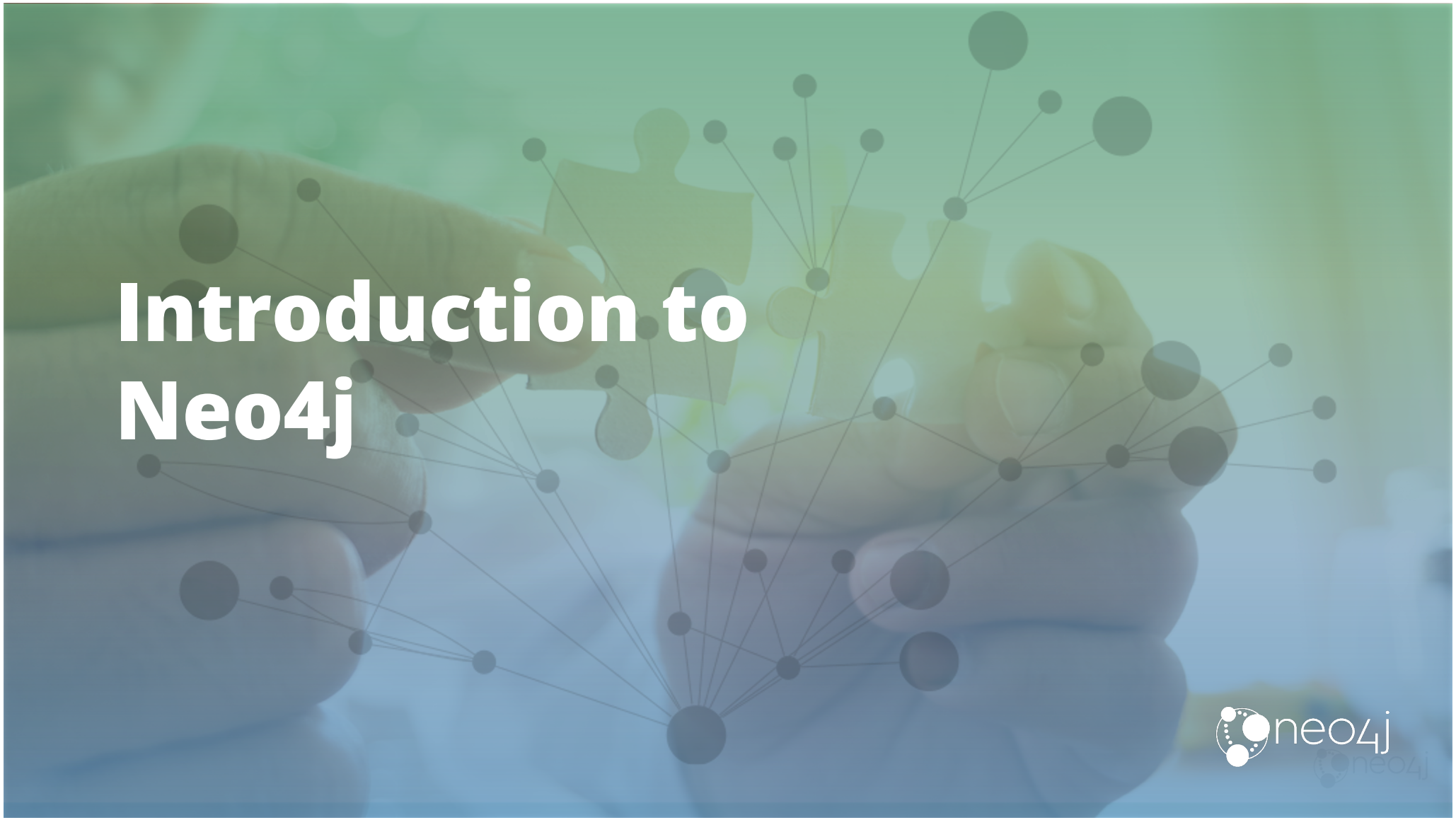
Corollary*:* in Neo4j, every object should store as little data as is feasible

Summary

122D warning: *This cries out for a "DGM" (data > memory) performance benchmark...* ☺

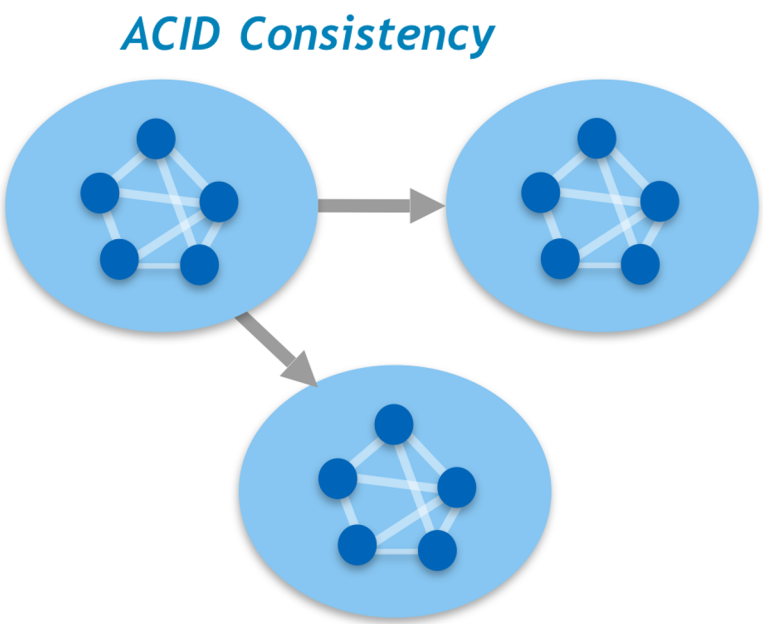
- RDBMS is very inefficient at traversal
  - *Foreign keys, lookup tables, and a four-step process are required*
- Index-Free Adjacency speeds traversal by using **pointers** instead of keys and lookup tables

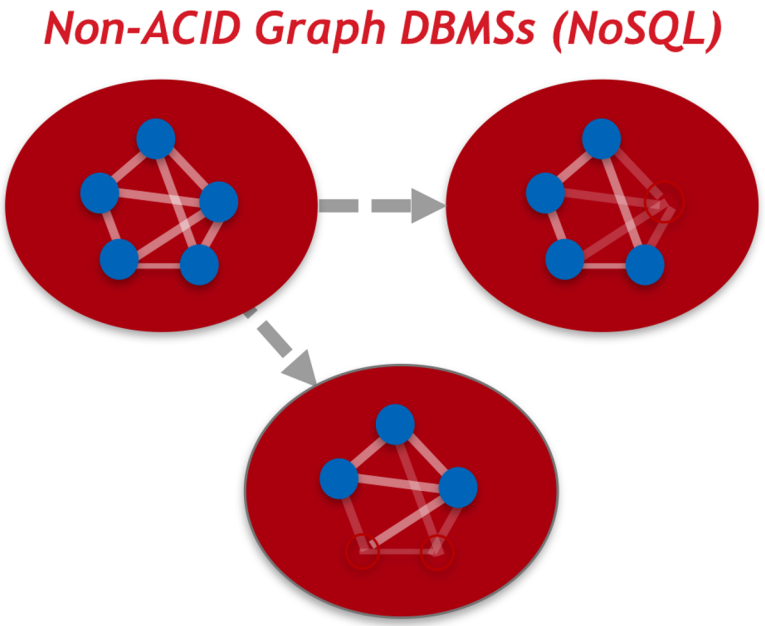
- Neo4j uses **chunking** to store data maximally efficiently
  - *Because of this, all objects should store **as little data as feasible***


Introduction to
Neo4j
neo4j

Neo4j Database: ACID

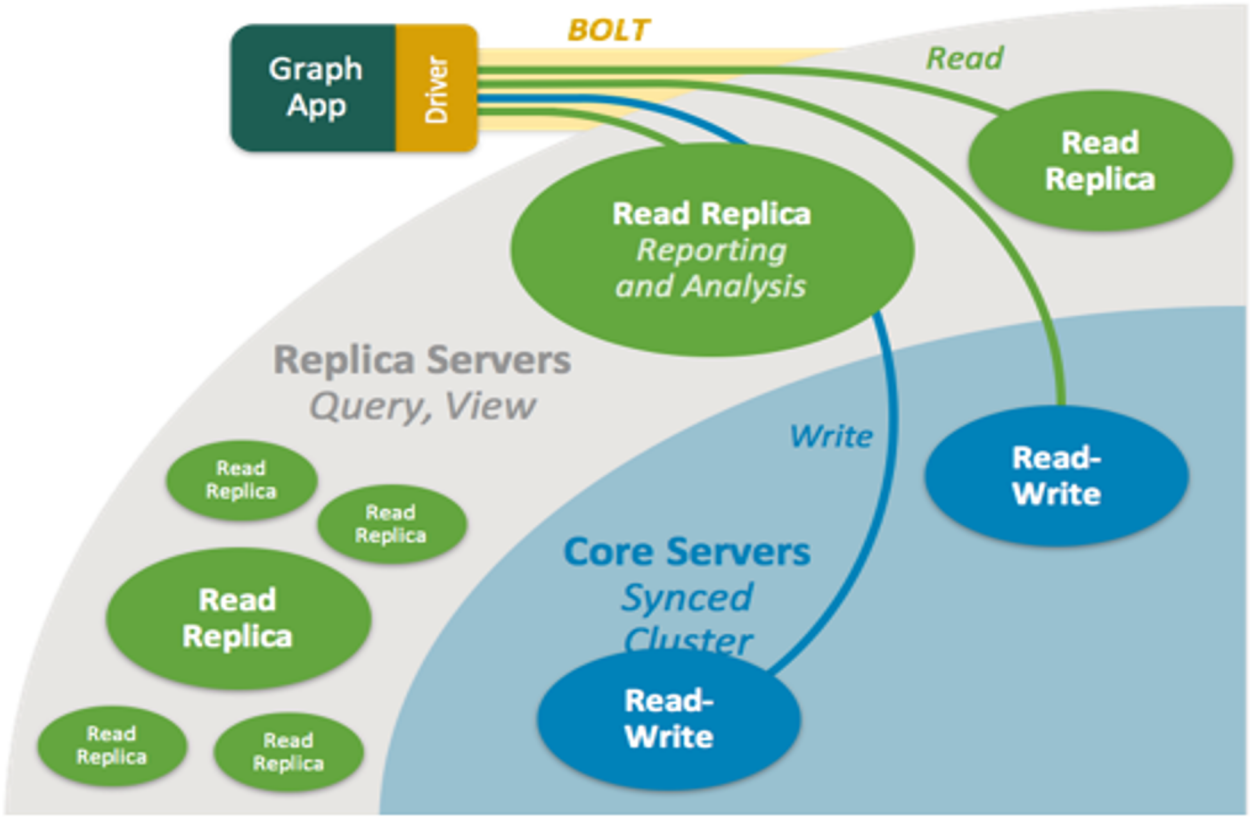
Transactional consistency - all updates either succeed or fail.





Clusters

ACID across locations.



neo4j

Graph engine

- Interpret Cypher statements
- Store and retrieve data
- Kernel-level access to filesystem
- Scalable (via replicas)
- Performant

Language and driver support

- Cypher to access the database
- Open source Cypher
- You can write server-side extensions to access the database
- Out-of-the-box drivers to access the database via **bolt** protocol:
  - Java
  - JavaScript
  - Python
  - C#
  - Go
- Neo4j community contributions for other languages

Libraries

Out-of-the-box:

- "Awesome Procedures" on Cypher (APOC)
- Graph Algorithms
- GraphQL

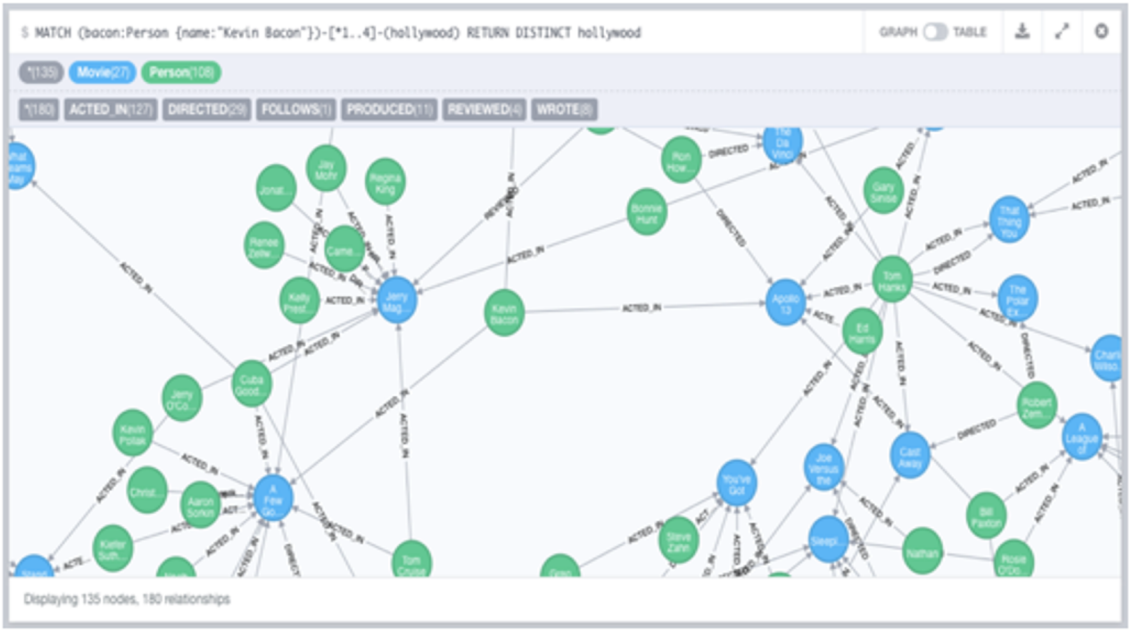
Neo4j community has contributed many specialized libraries also.

neo4j

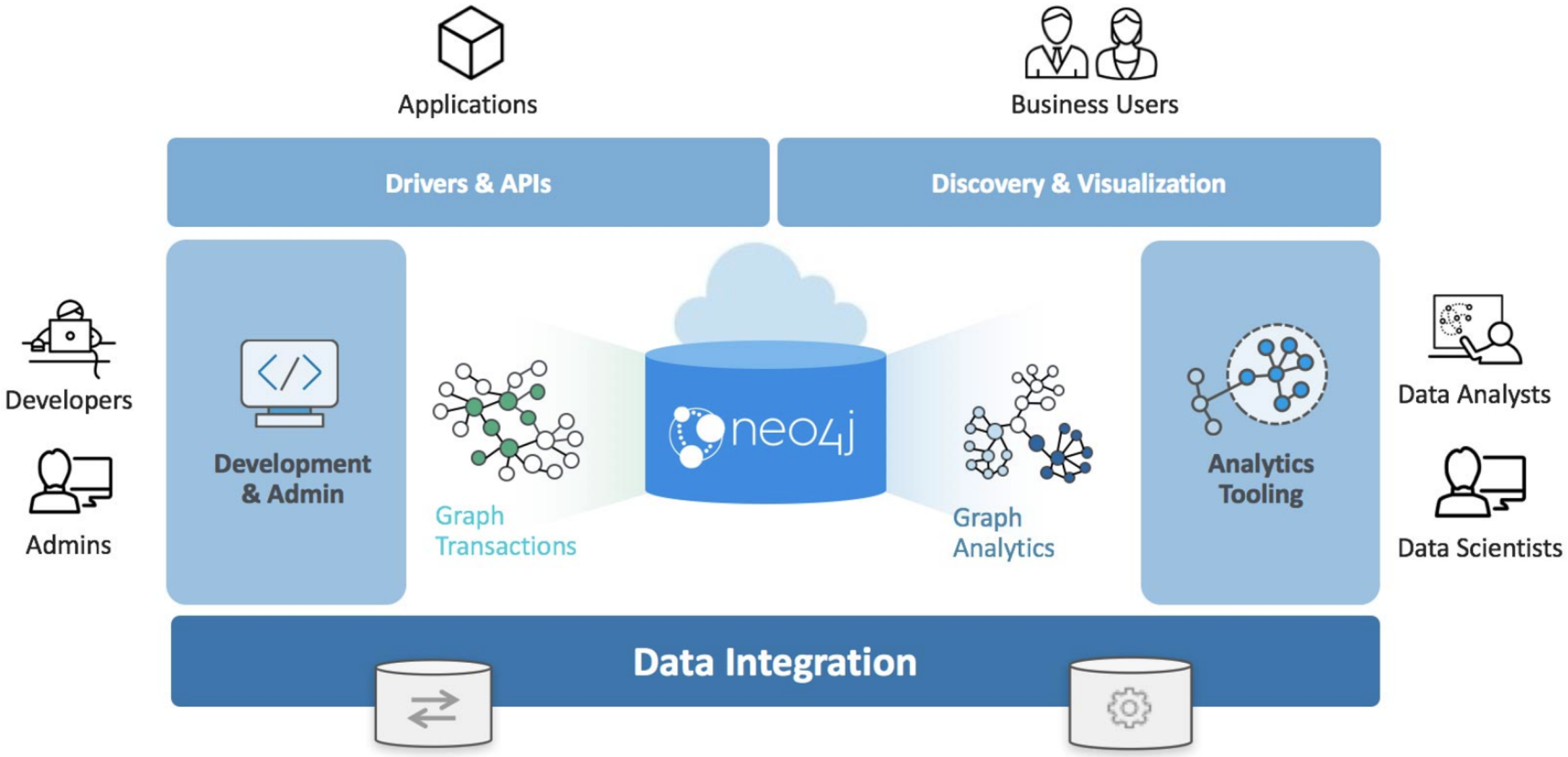
Tools

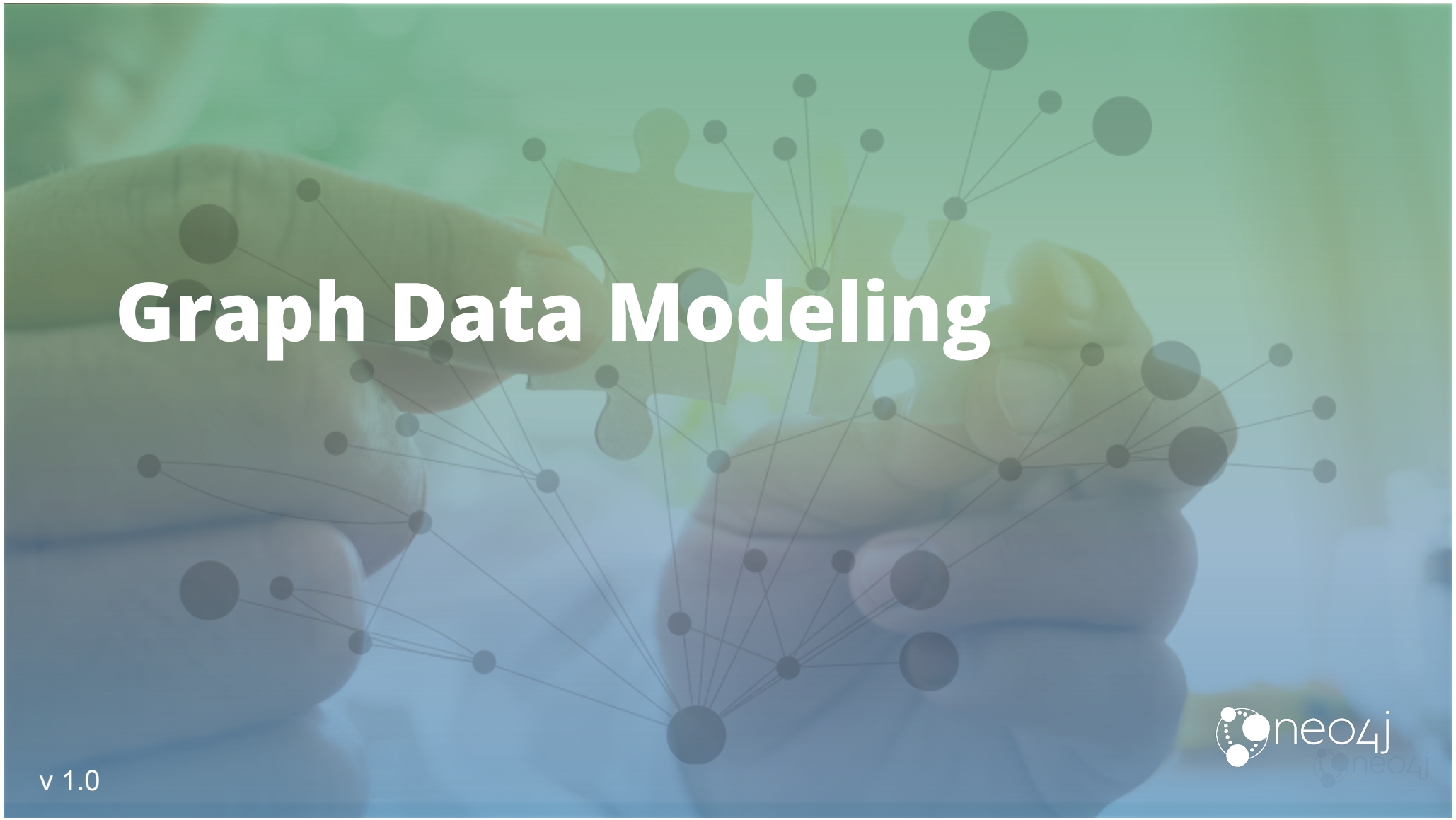
- Neo4j Desktop
- Neo4j Browser
- Neo4j Bloom
- Neo4j ETL Tool

Neo4j community has contributed many specialized tools also.



Neo4j Graph Platform architecture




Graph Data Modeling
v 1.0
neo4j

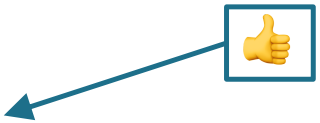
Neo4j is a *property graph* data model

The components of a Neo4j property graph include:

- Nodes (Entities)
- Relationships (Connections between Entities)
- Properties
- Labels

You create graphs to answer important questions for your domain.

Paths are created and navigated based upon the relationships between nodes.



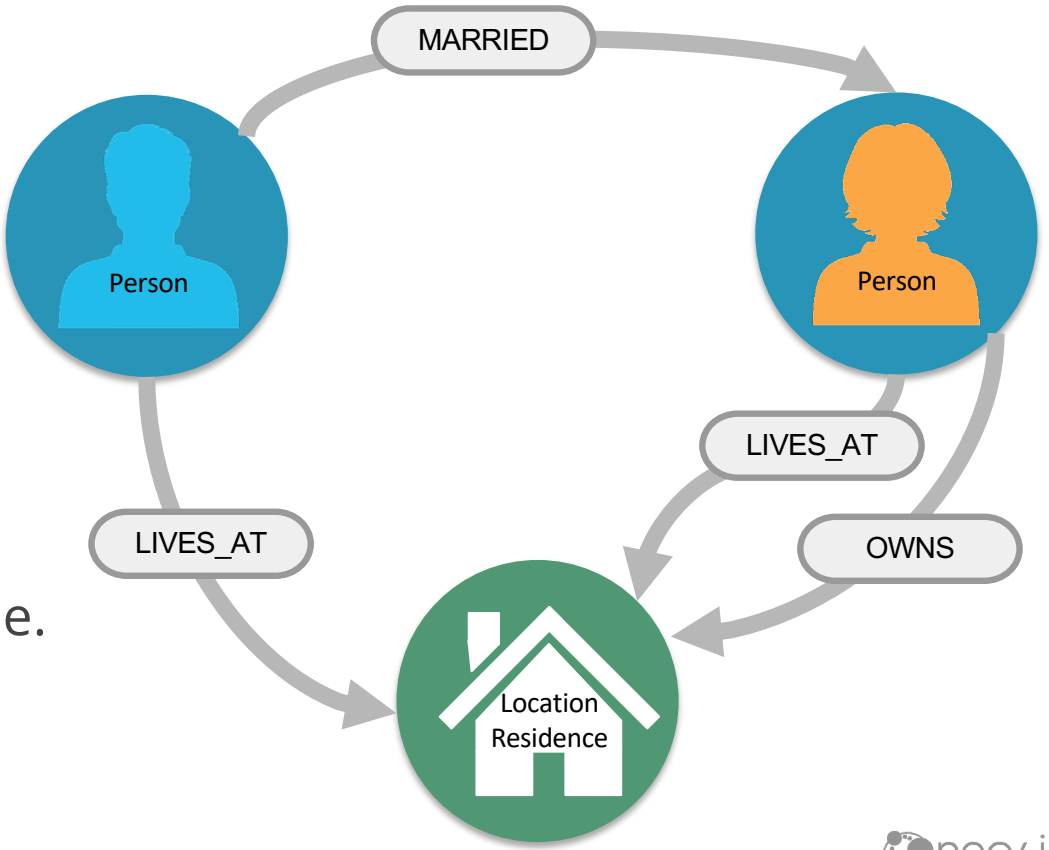
https://neo4j.com/docs/getting-started/current/graphdb-concepts/

In Neo4j, entities are *nodes* with *labels*

- Represent the nouns in your domain questions .
- Represent the objects or entities in the graph.
- Can be ***labeled*** to represent the type of node or role of node:
  - Person
  - Location
  - Residence
  - Business
- Nodes are grouped by their **labels** for optimizing queries.

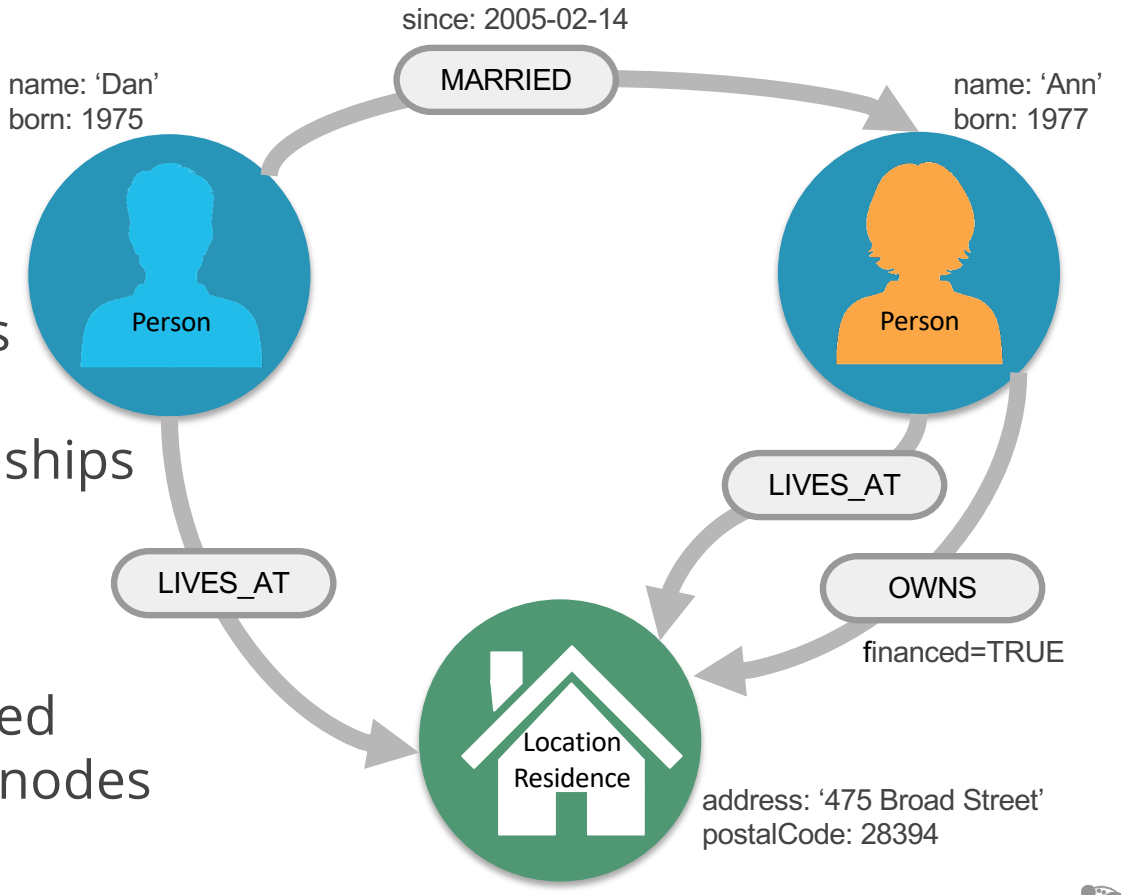
In Neo4j, connections are *relationships*

- Represent verbs in your domain questions.
- Represent the **connection** between exactly two nodes in the graph.
- Connect nodes of same type or of different types.
- Has a type:
  - MARRIED
  - LIVES_AT
  - OWNS
- Directed relationship in the database.
- Used to define paths navigated at runtime.



In Neo4j, *properties* provide the data

- Adjectives to describe nodes
  - Proper nouns
- Adverbs to describe relationships
  - Strength, weight, quality
- Property:
  - Key/value pair
  - Can be optional or required
  - Values can be unique for nodes

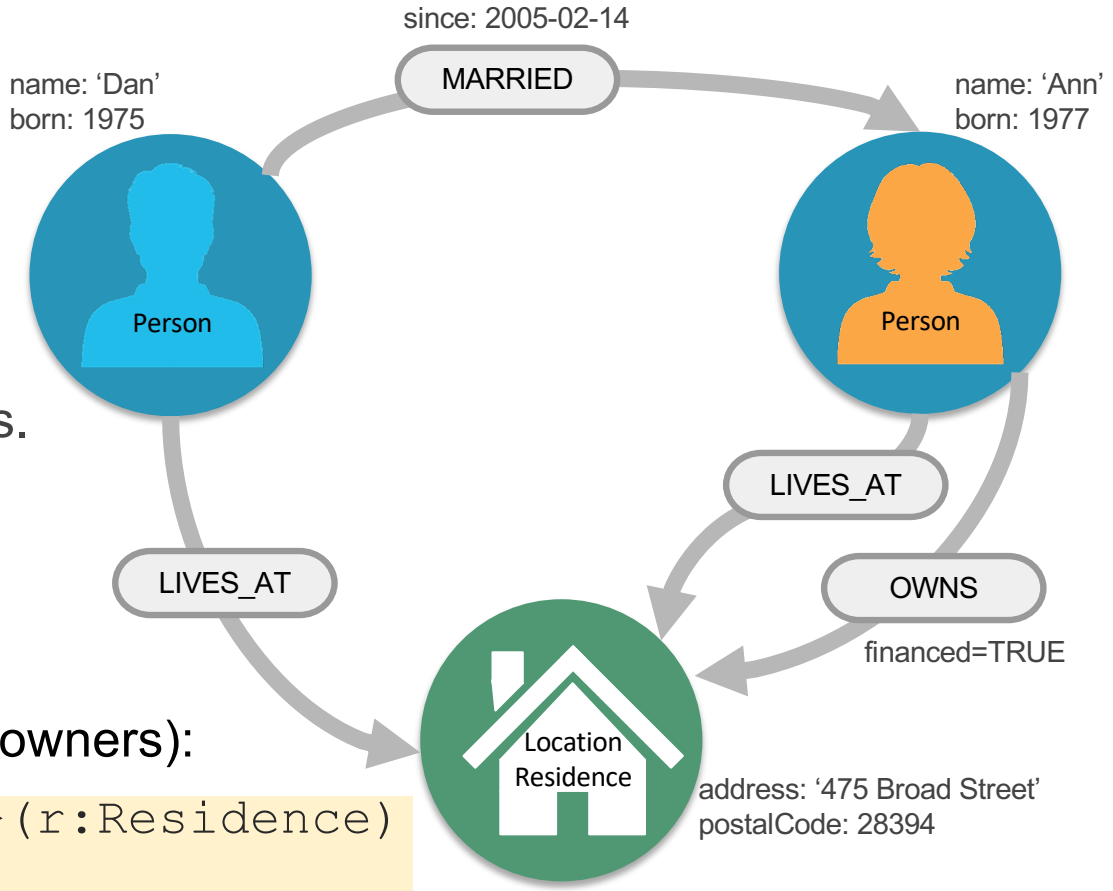


neo4j

In Neo4j, paths are traversed at runtime

Paths are used to:

- Discover what is absolutely necessary to answer questions.
- Eliminate parts of graph not needed to answer a question.

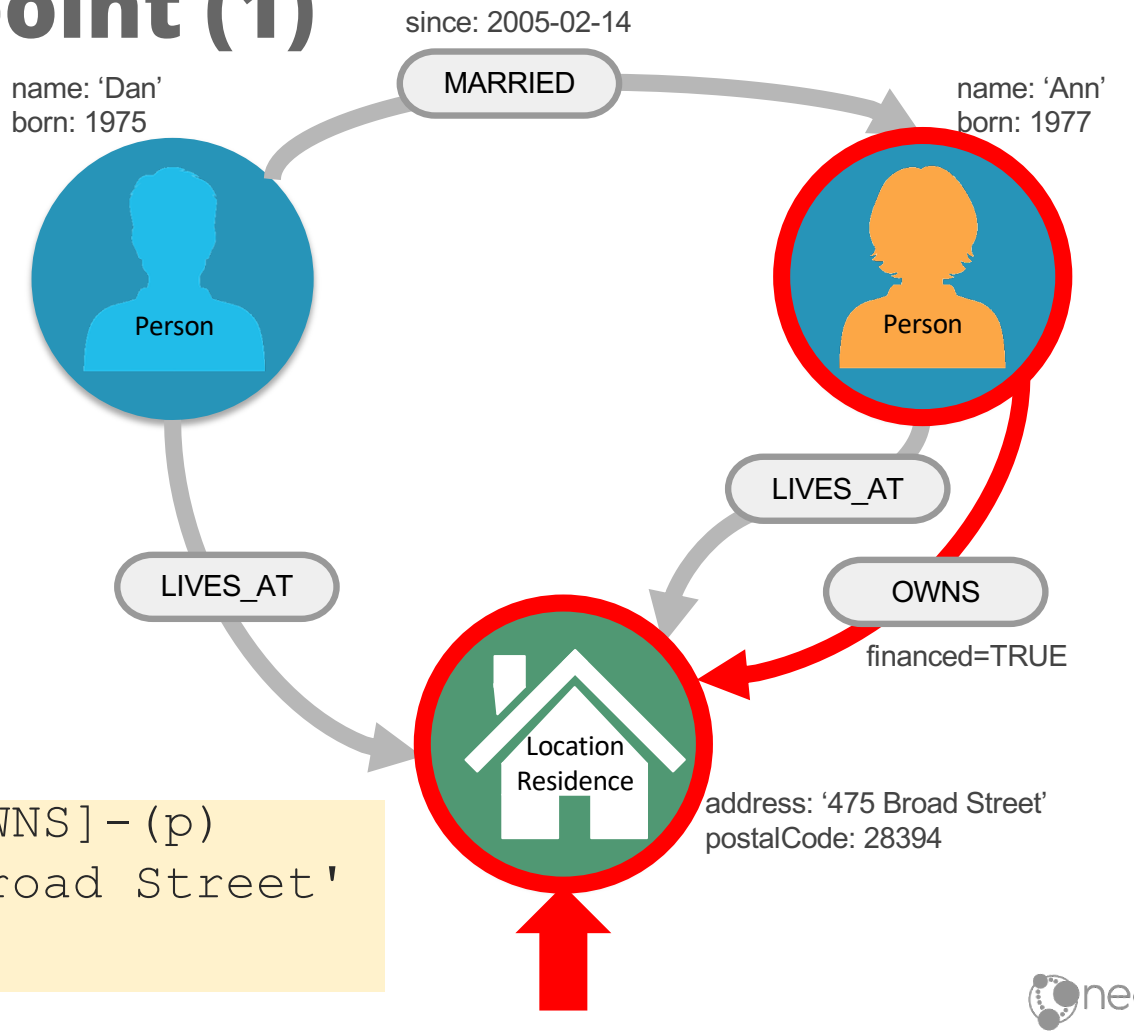


Example Cypher query (residence owners):

```
MATCH (p:Person)-[:OWNS]->(r:Residence)
RETURN p,r
```

Path is starting point (1)

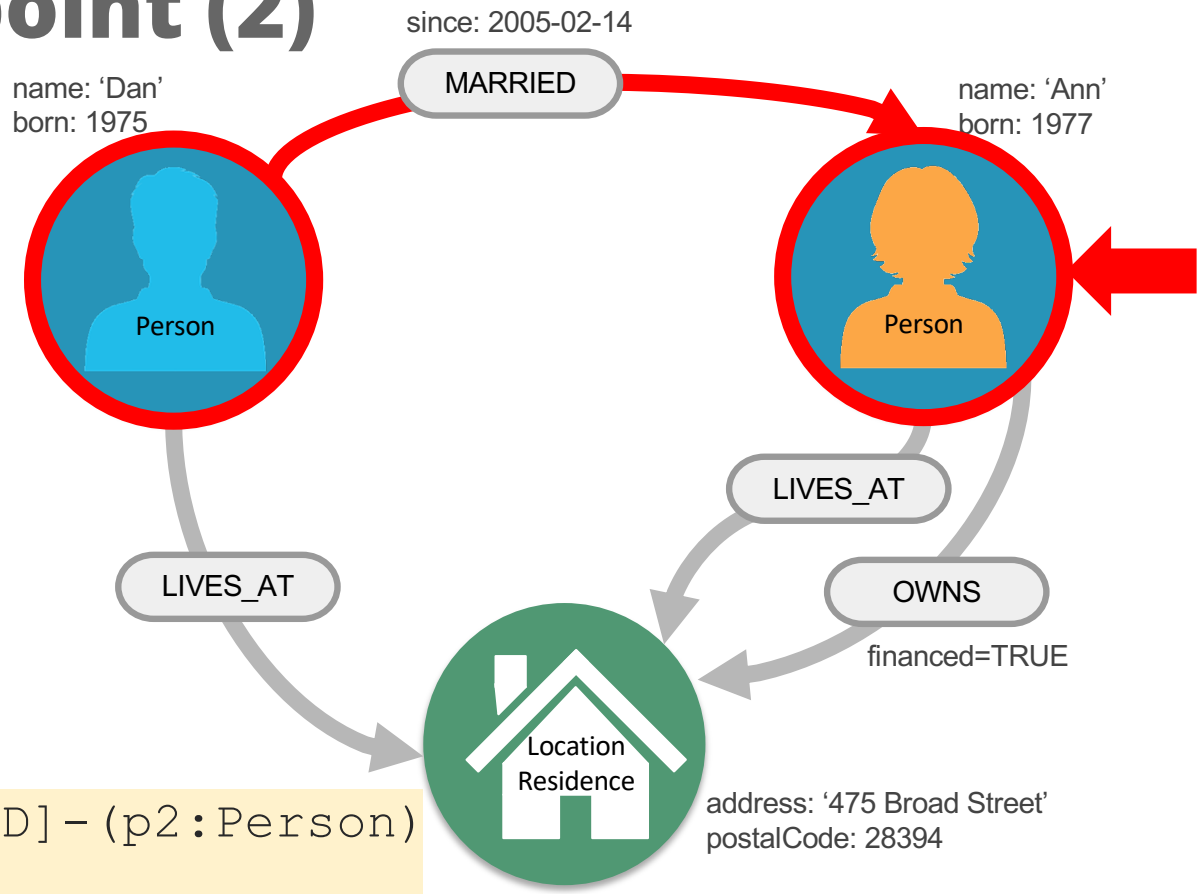
What nodes are visited to find *all Person nodes that own the house at 475 Broad Street*?



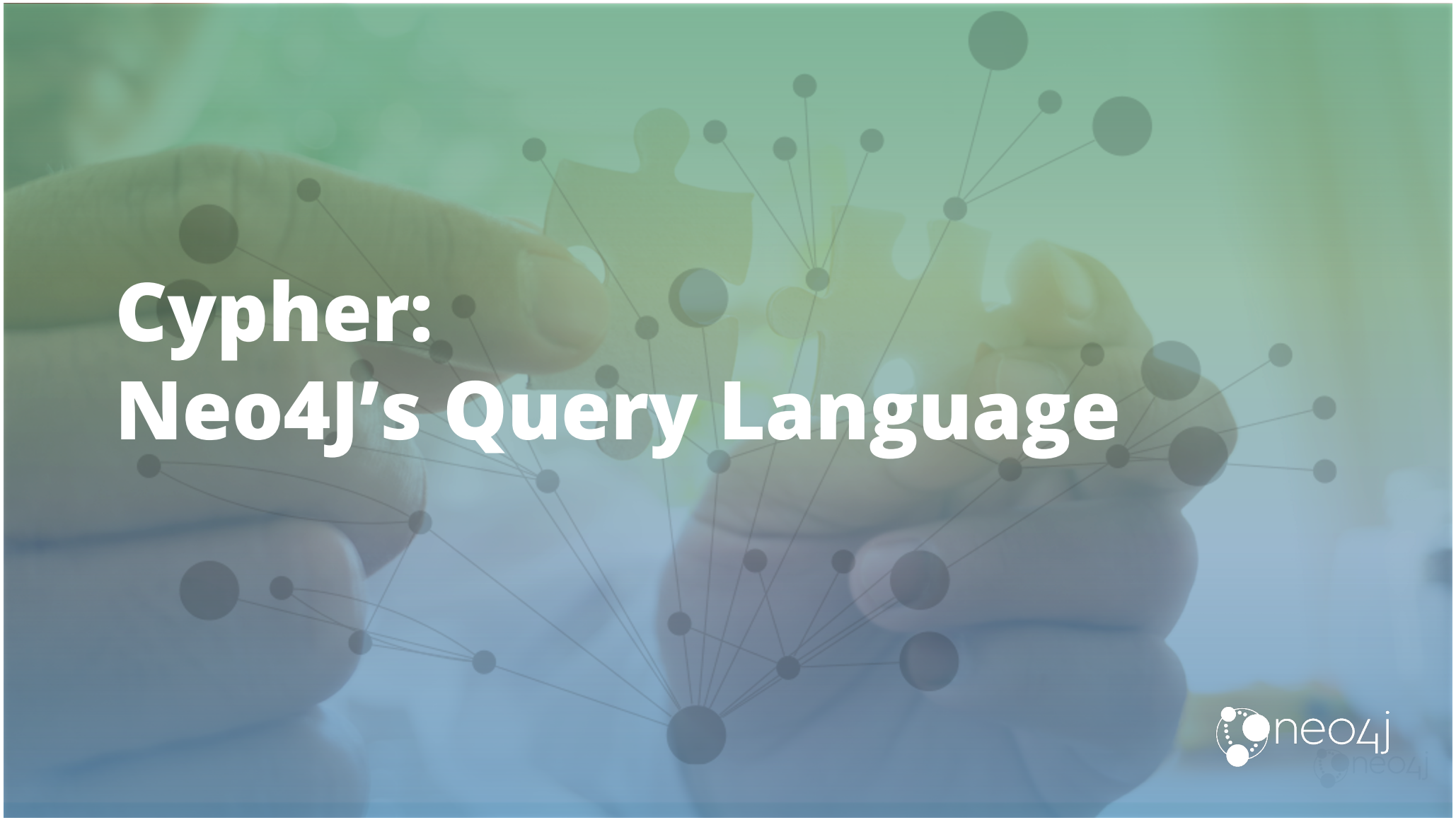

```
MATCH (r:Residence)<-[:OWNS]-(p)
WHERE r.address = '475 Broad Street'
RETURN p
```

Path is starting point (2)

What nodes are visited to find *who is married to Ann*?



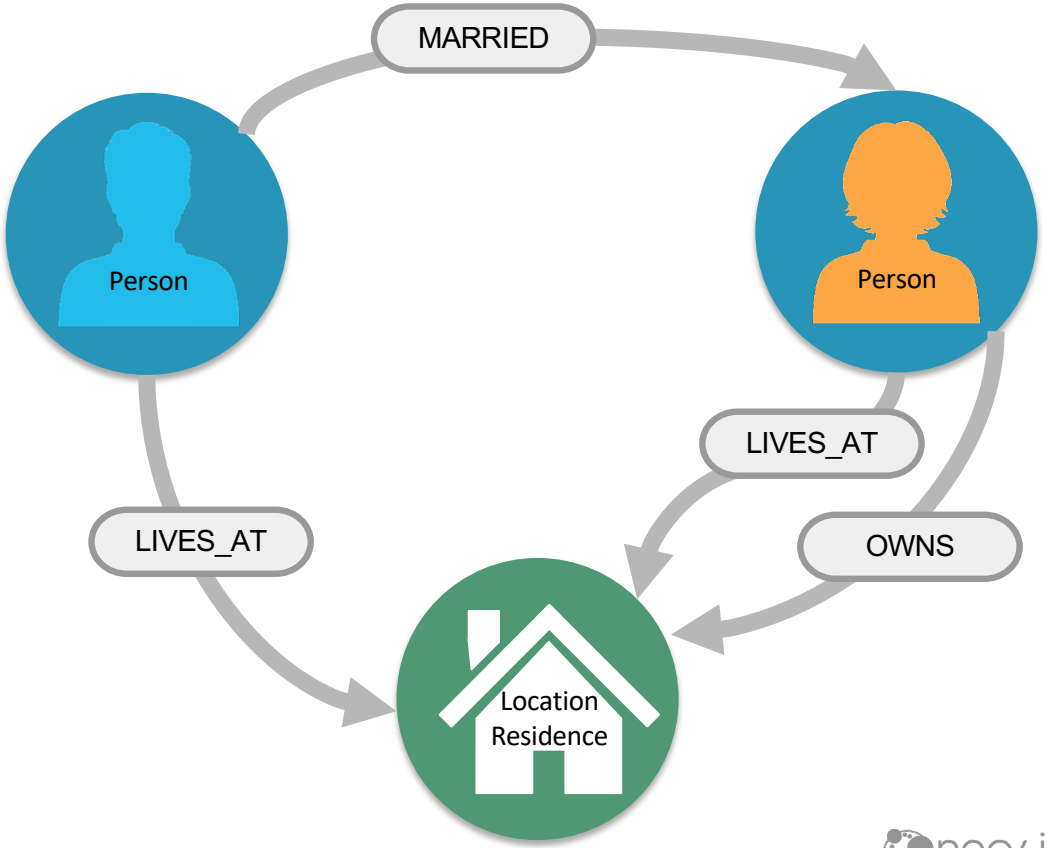

```
MATCH (p1:Person)-[:MARRIED]-(p2:Person)
WHERE p1.name = 'Ann'
RETURN p2
```


Cypher:
Neo4J's Query Language
neo4j

What is Cypher?

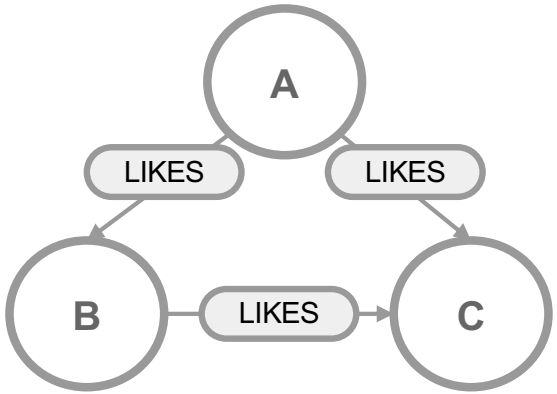
- Declarative query language
- Focuses on **what**, not how to retrieve
- Uses keywords such as **MATCH**, **WHERE**, **CREATE**
- Runs in the database server for the graph
- **ASCII art** to represent nodes and relationships

;-)



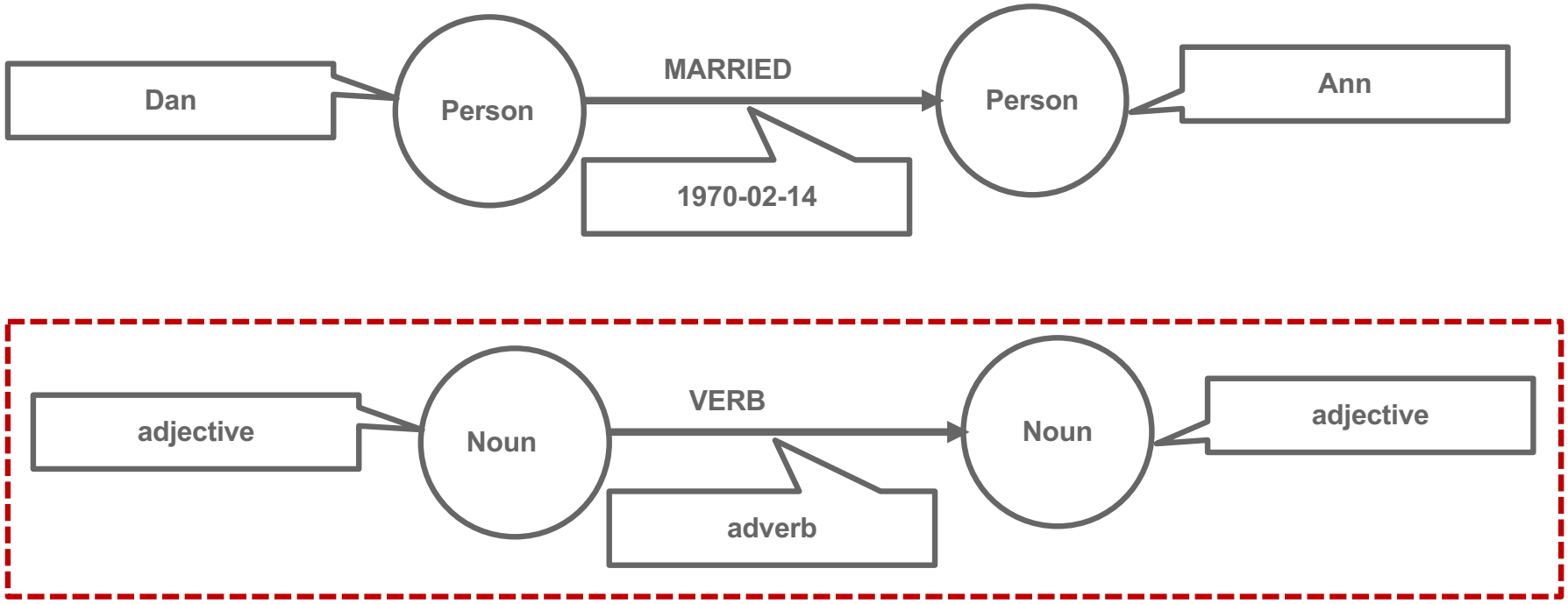
neo4j

Cypher is ASCII Art



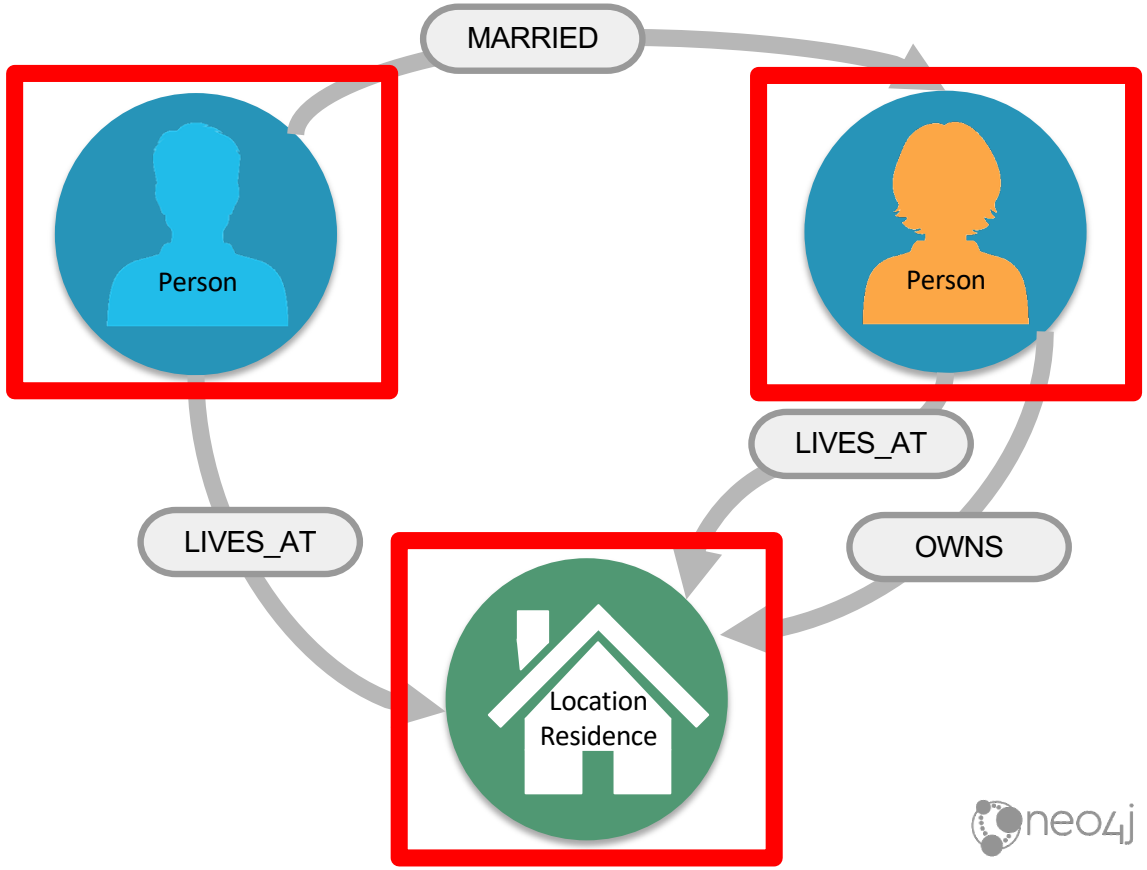

```
(A)-[:LIKES]->(B),(A)-[:LIKES]->(C),(B)-[:LIKES]->(C)

    (A)-[:LIKES]->(B)-[:LIKES]->(C)<-[:LIKES]-(A)
```

Cypher is readable



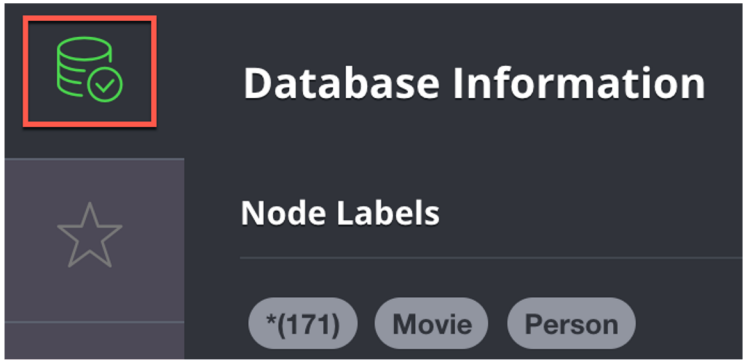
Nodes

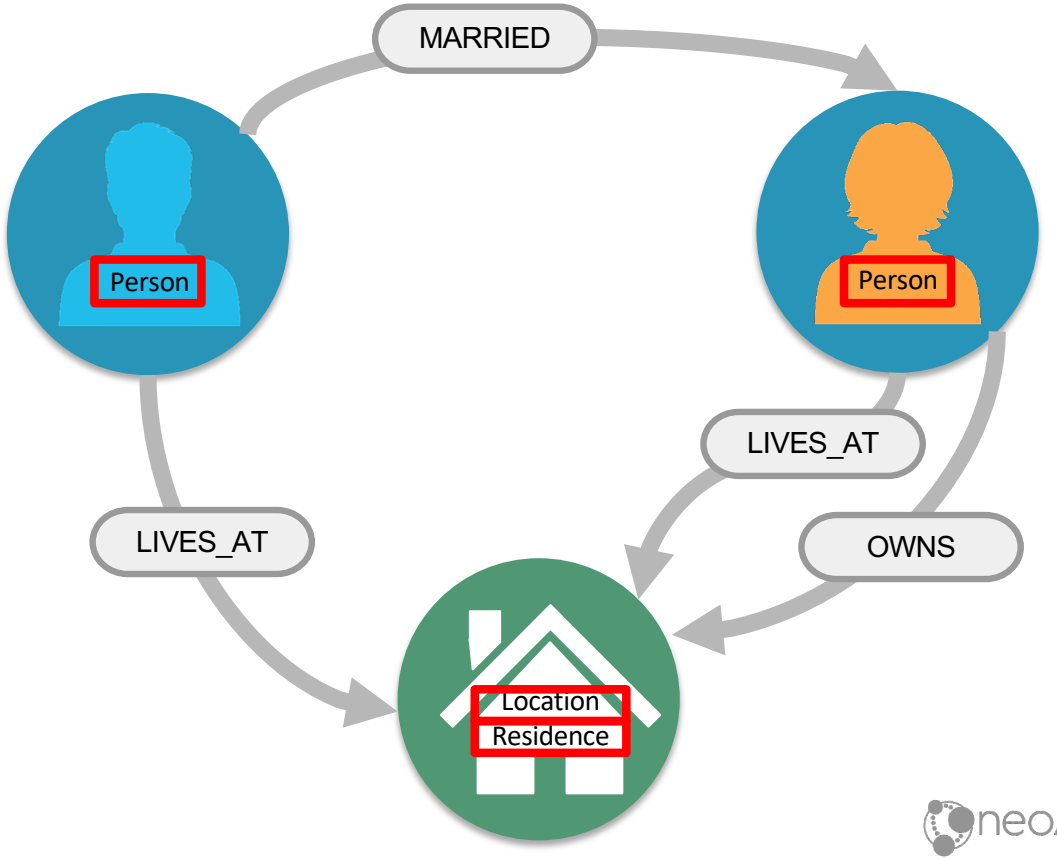
```
()
(p)
(l)
(n)
```



Labels

```
(:Person)
(p:Person)
(:Location)
(l:Location)
(n:Residence)
(x:Location:Residence)
```

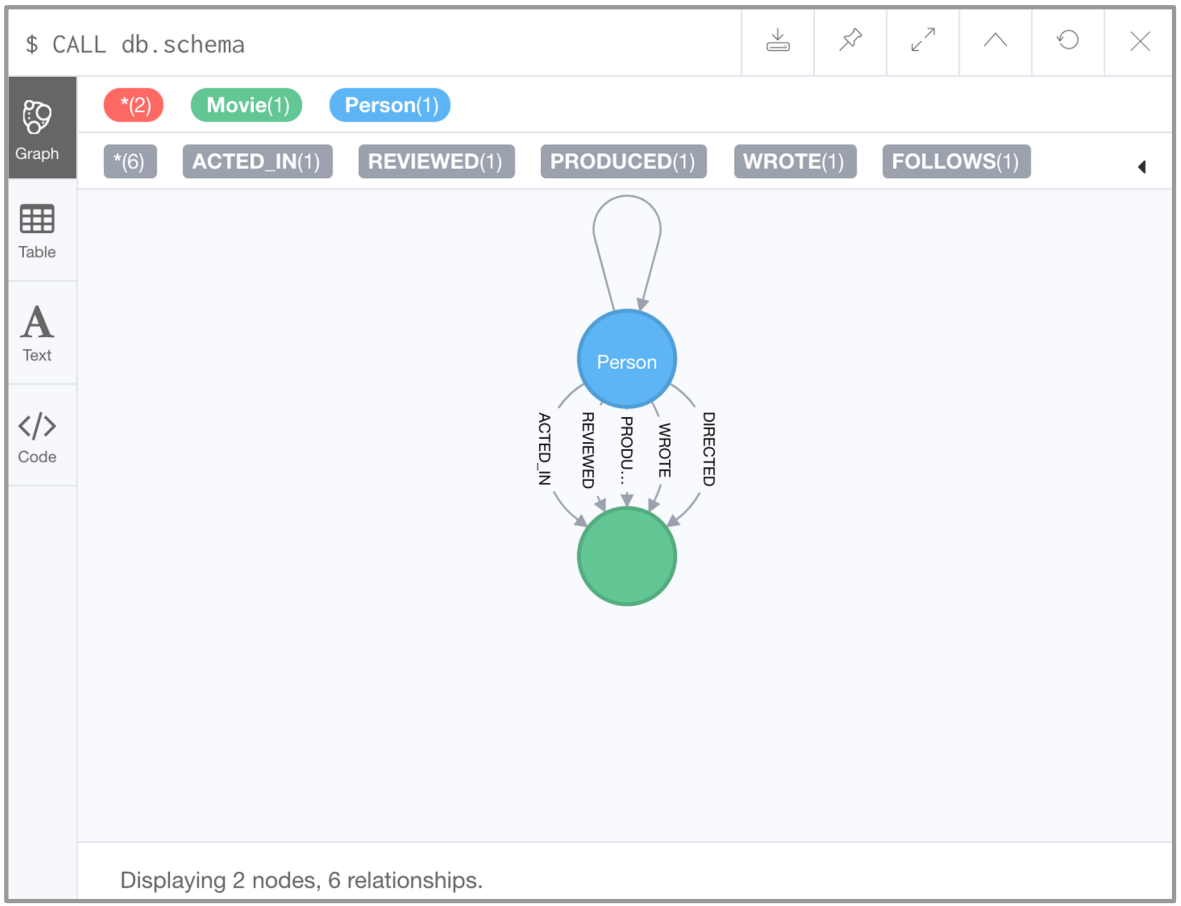




Comments in Cypher

```
()                  // anonymous node not be referenced later in the query
(p)                 // variable p, a reference to a node used later
(:Person)           // anonymous node of type Person
(p:Person)          // p, a reference to a node of type Person
(p:Actor:Director)  // p, a reference to a node of types Actor and Director
```

Examining the data model

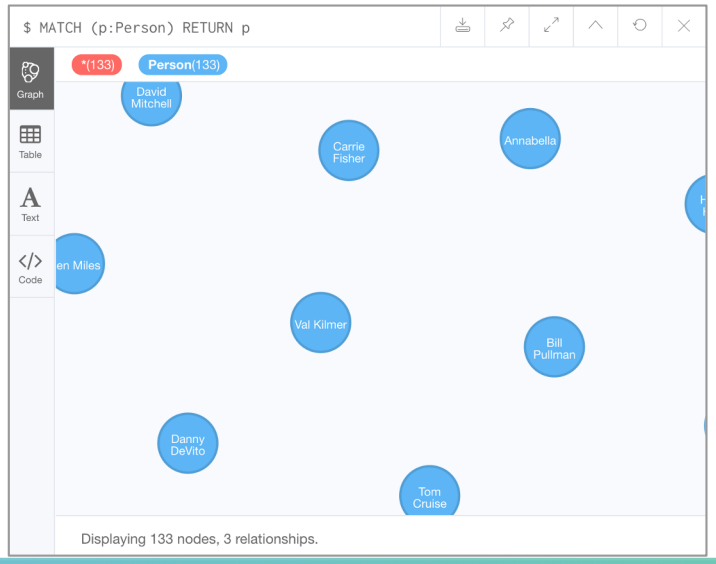


neo4j

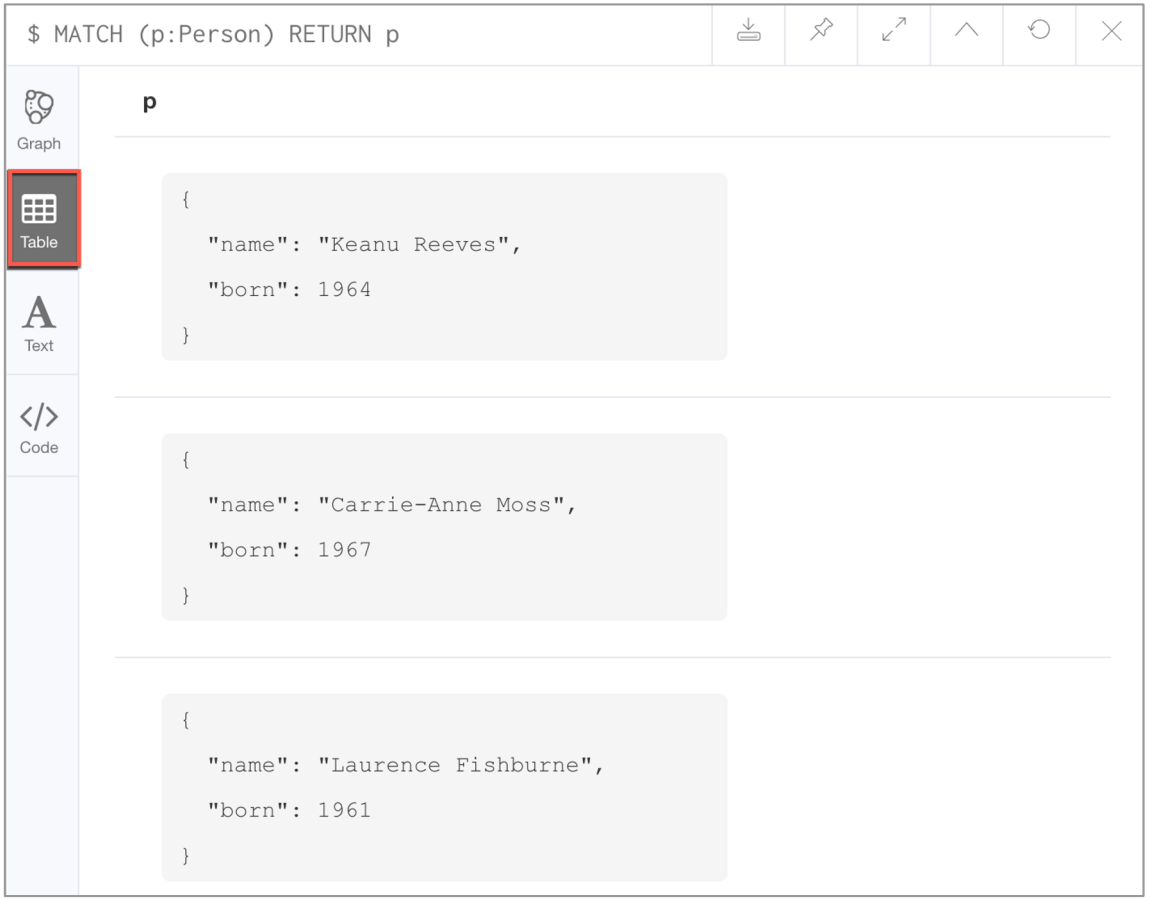
Using MATCH to retrieve nodes

```
MATCH (n) // returns all nodes in the graph
RETURN n
```

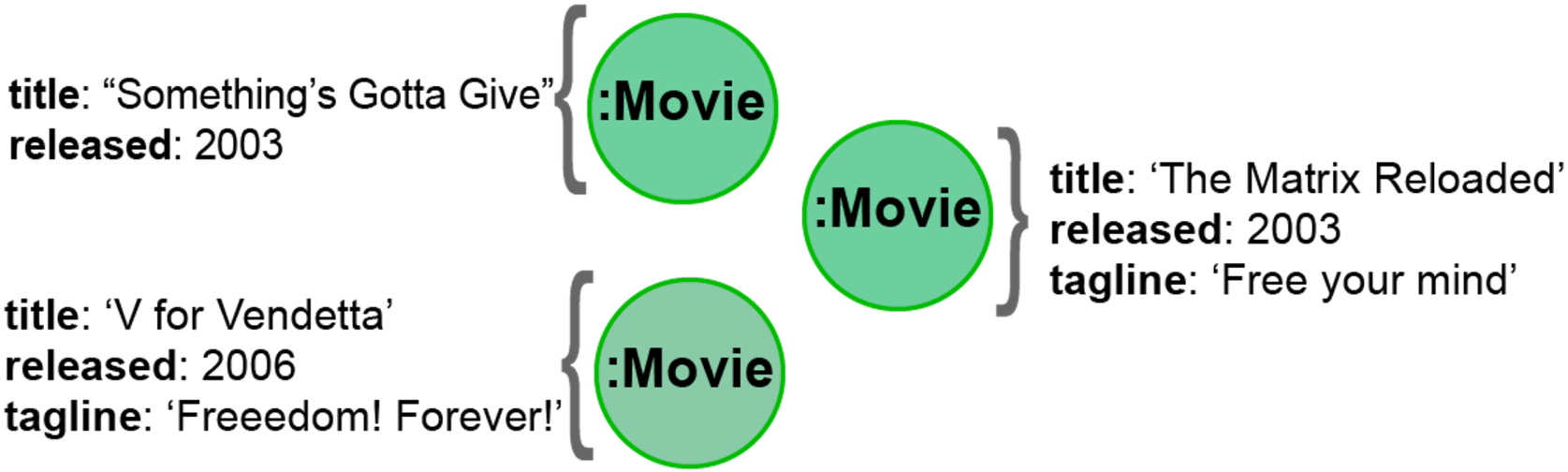
```
MATCH (p:Person) // returns all Person nodes in the graph
RETURN p
```



Viewing nodes as table data

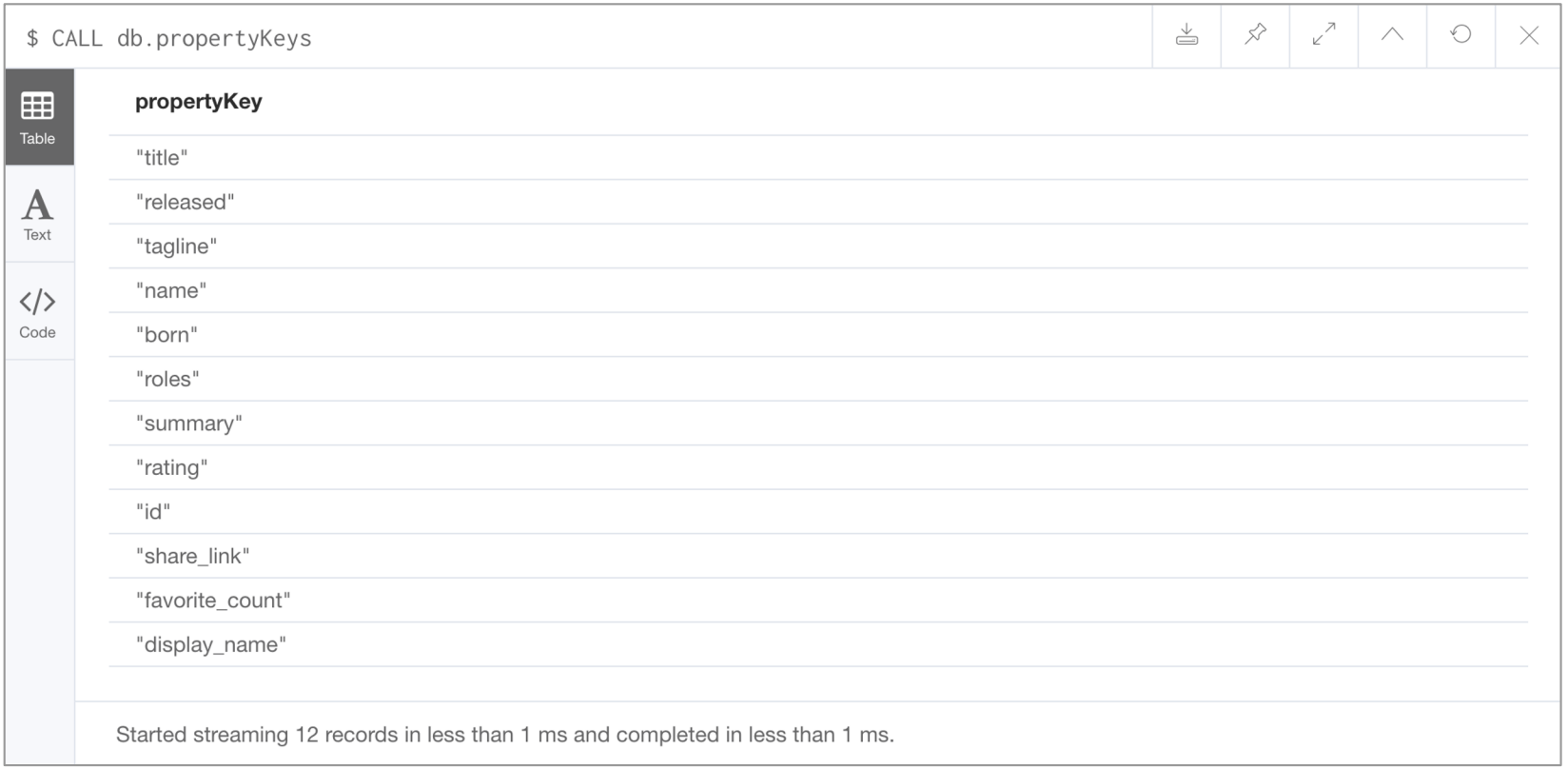


Properties


title: “Something’s Gotta Give”
released: 2003
:Movie
:Movie
title: ‘The Matrix Reloaded’
released: 2003
tagline: ‘Free your mind’
title: ‘V for Vendetta’
released: 2006
tagline: ‘Freeedom! Forever!’
:Movie

Examining property keys

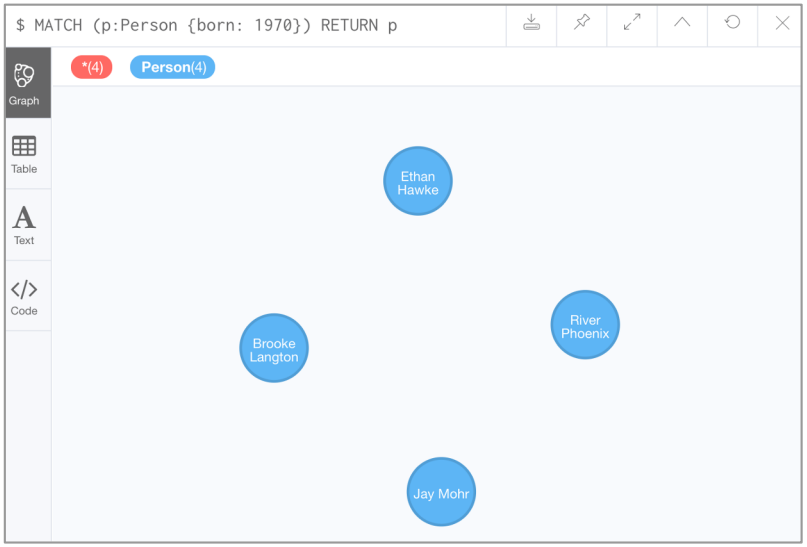
```
CALL db.propertyKeys
```



Retrieving nodes filtered by a property value - 1

Find all *people* born in *1970, returning the nodes*:

```
MATCH (p:Person {born: 1970})
RETURN p
```



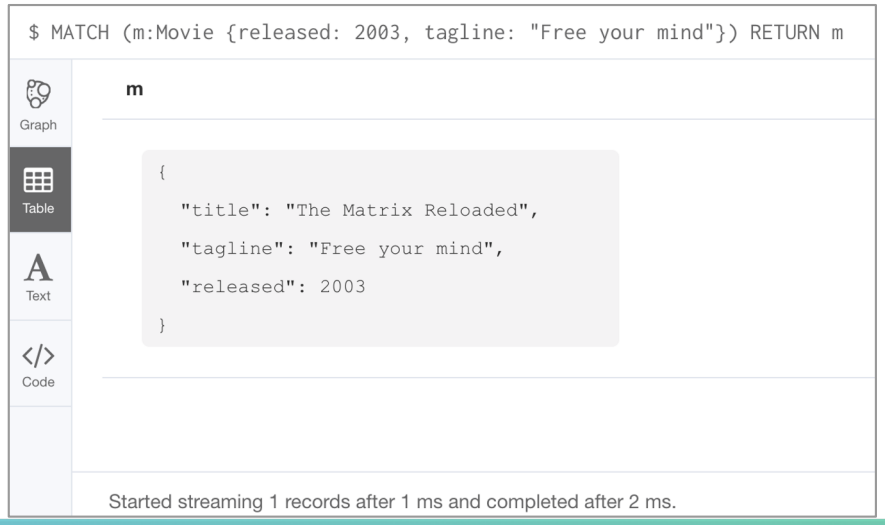
Retrieving nodes filtered by a property value - 2

Find all movies released in *2003* with the tagline,

Free your mind, returning the nodes:

```
MATCH (m:Movie {released: 2003, tagline: 'Free your mind'})
RETURN m
```

Q: *Does this syntax remind you of anything....?*



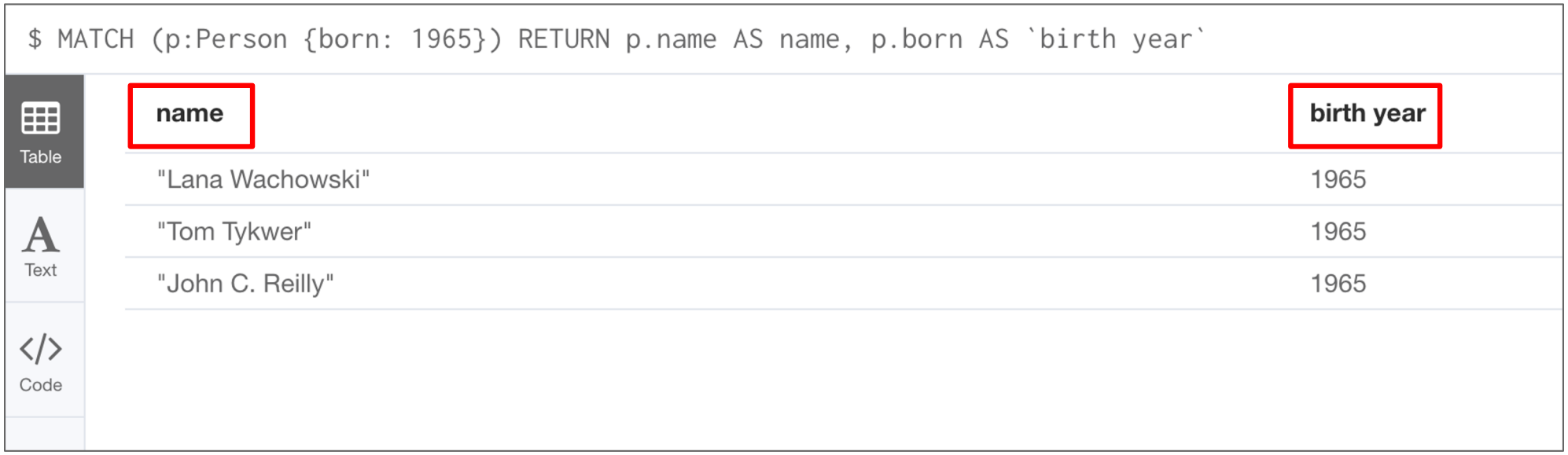
Returning property values

Find all people born in *1965* and return their names:

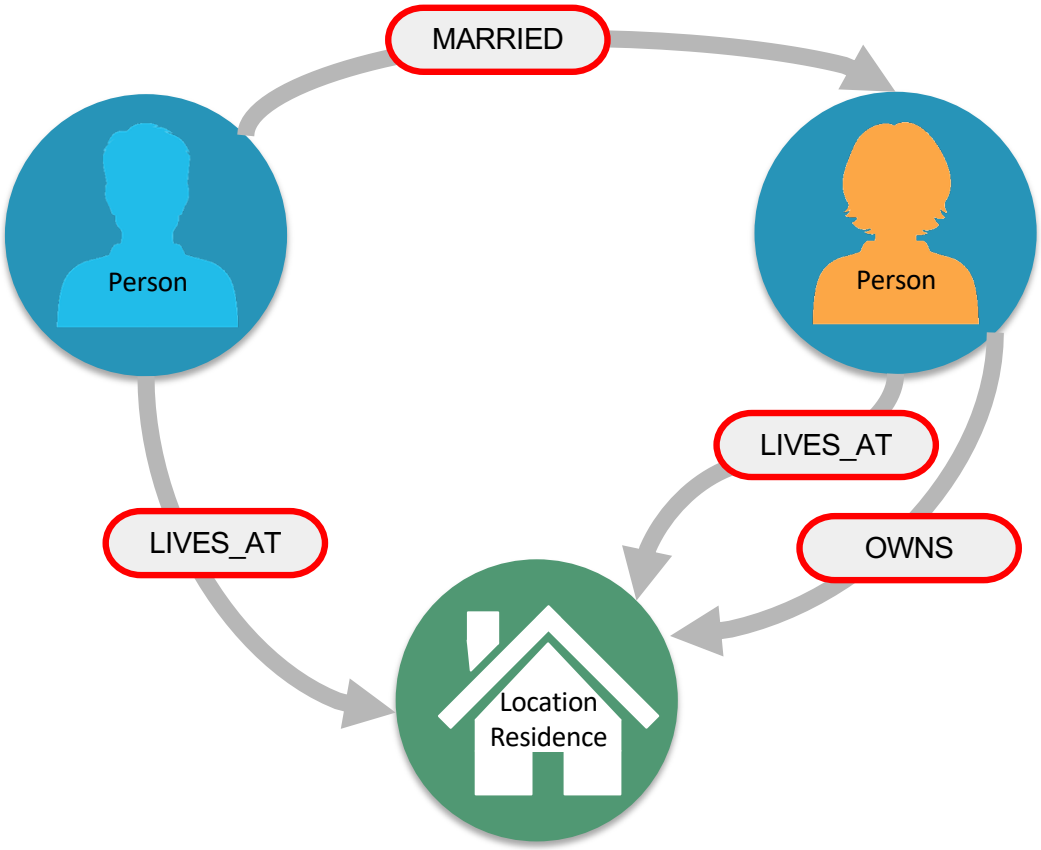
```
MATCH (p:Person {born: 1965})
RETURN p.name, p.born
```

Specifying aliases

```
MATCH (p:Person {born: 1965})
RETURN p.name AS name, p.born AS `birth year`
```



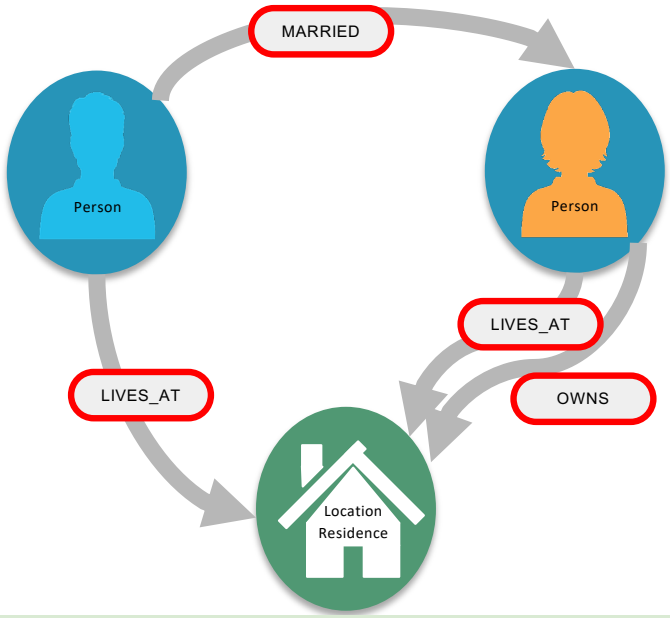
Relationships



ASCII art for nodes and relationships

```
()          // a node
()--()      // 2 nodes have some type of relationship
()-->()     // the first node has a relationship to the second node
()<--()     // the second node has a relationship to the first node
```

Querying using relationships

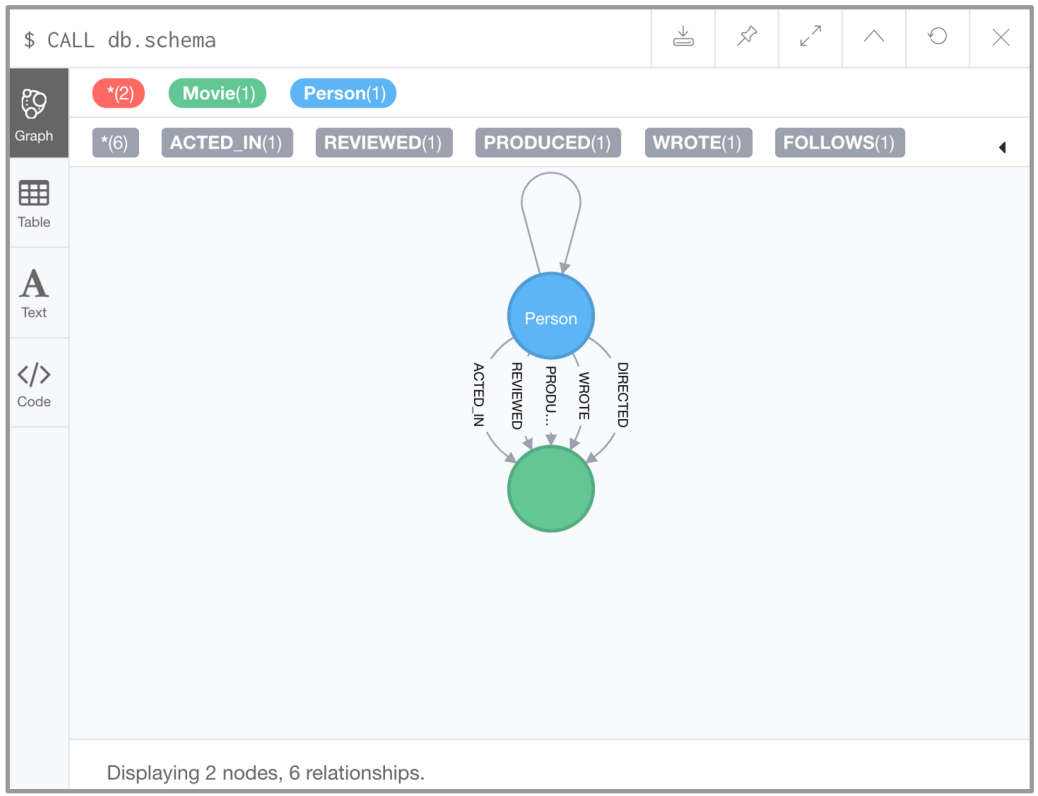



```
MATCH (p:Person)-[:LIVES_AT]->(h:Residence)
RETURN p.name, h.address
```

```
MATCH (p:Person)--(h:Residence) // any relationship
RETURN p.name, h.address
```

Examining relationships

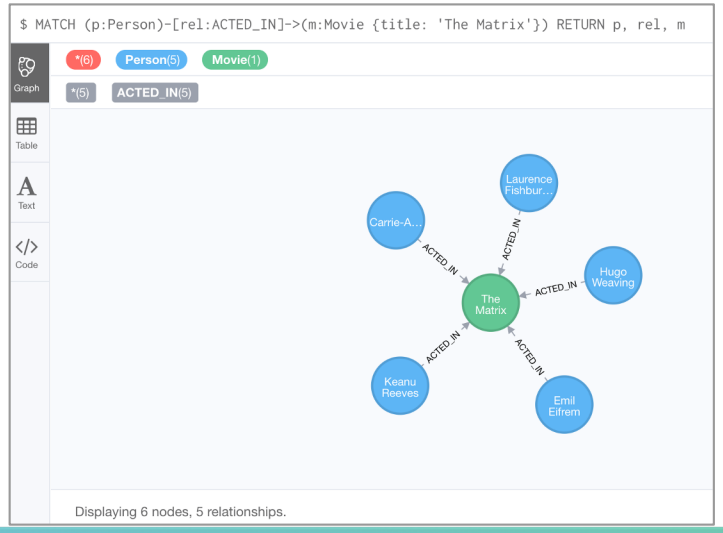
```
CALL db.schema
```



Using a relationship in a query

Find all people who acted in the movie, *The Matrix*, returning the nodes and relationships found:

```
MATCH (p:Person)-[rel:ACTED_IN]->(m:Movie {title: 'The Matrix'})
RETURN p, rel, m
```

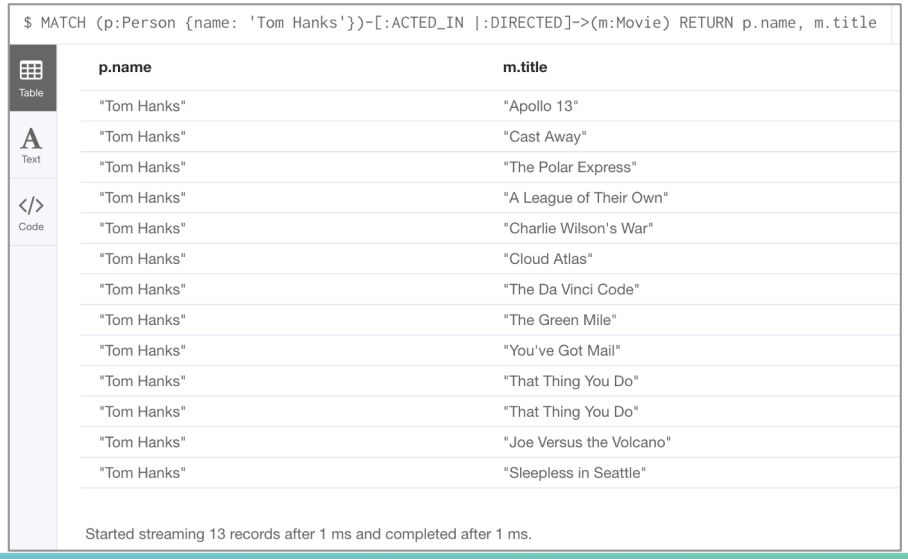


neo4j

Querying by multiple relationships

Find all movies that *Tom Hanks* acted in or directed and return the title of the move:

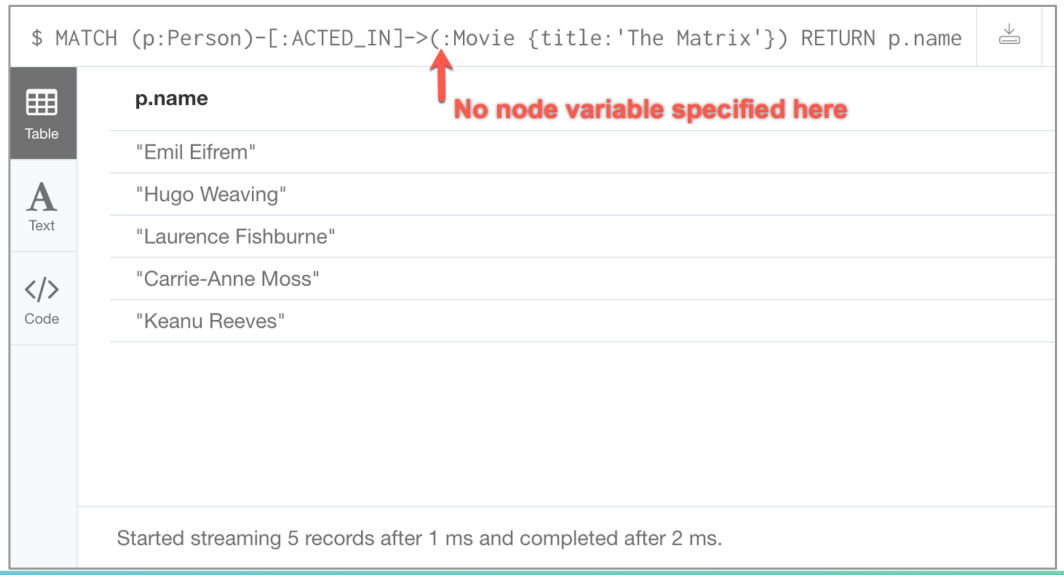
```
MATCH (p:Person {name: 'Tom Hanks'})-[:ACTED_IN |:DIRECTED]->(m:Movie)
RETURN p.name, m.title
```



Using anonymous nodes in a query

Find all people who acted in the movie, *The Matrix* and return their names:

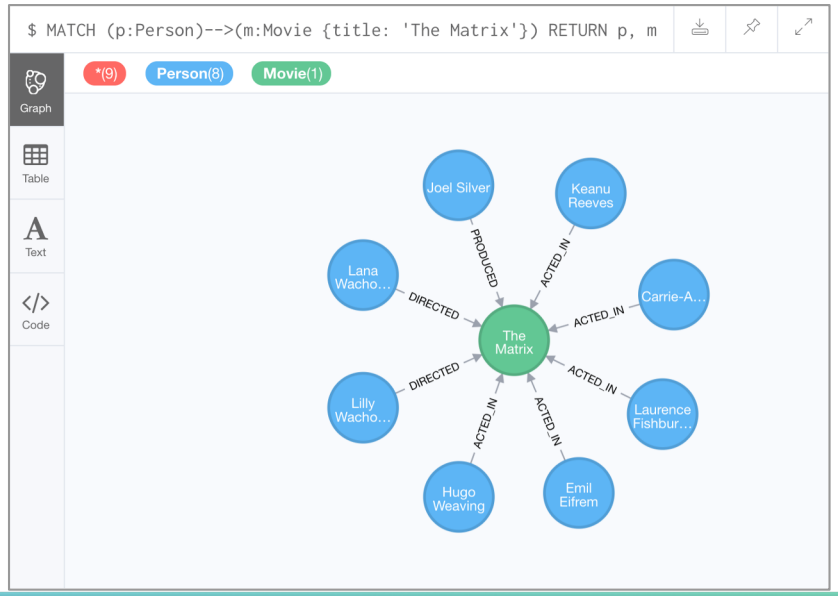
```
MATCH (p:Person)-[:ACTED_IN]->(:Movie {title: 'The Matrix'})
RETURN p.name
```



Using an anonymous relationship for a query

Find all people who have any type of relationship to the movie, *The Matrix* and return the nodes:

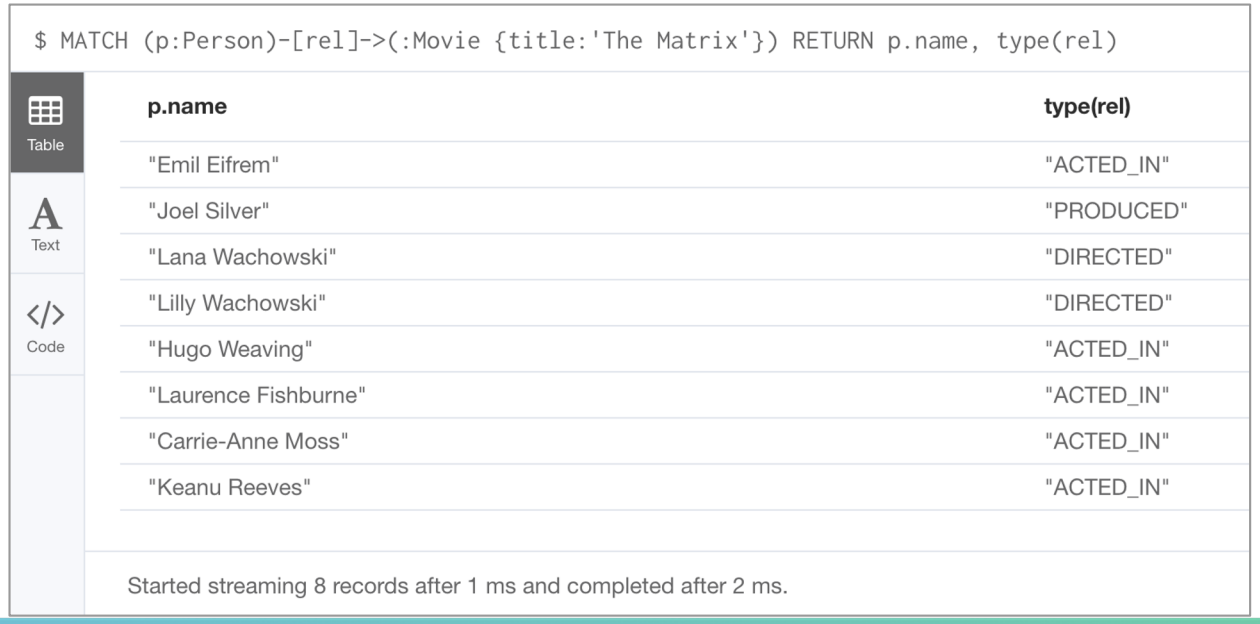
```
MATCH (p:Person)-->(m:Movie {title: 'The Matrix'})
RETURN p, m
```



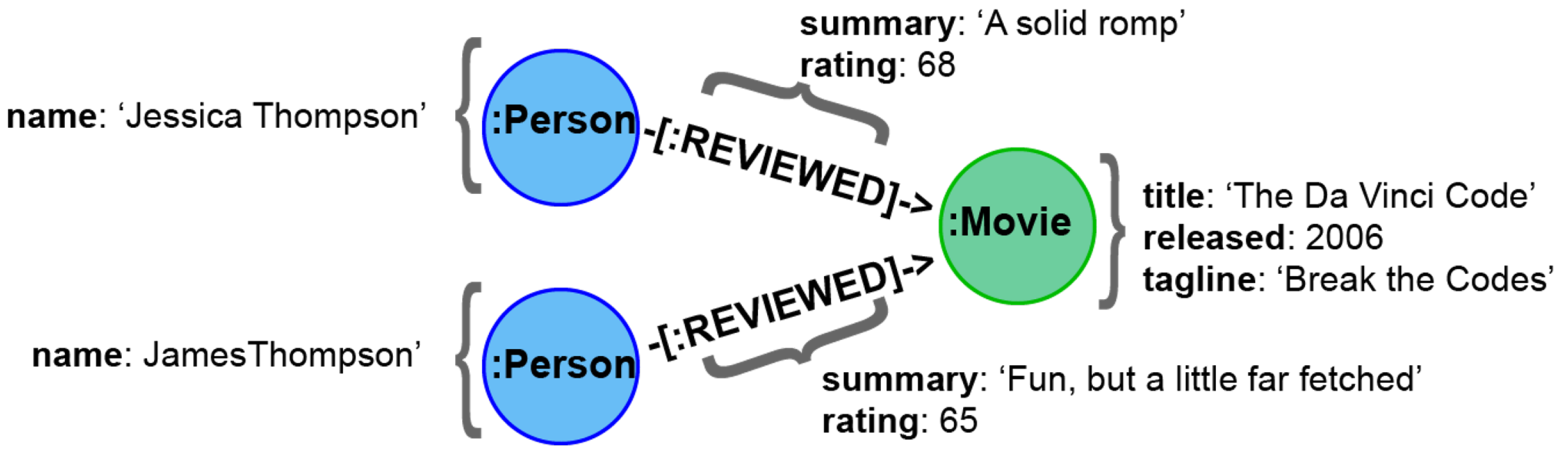
Retrieving relationship types

Find all people who have any type of relationship to the movie, *The Matrix* and return the name of the person and their relationship type:

```
MATCH (p:Person)-[rel]->(:Movie {title:'The Matrix'})
RETURN p.name, type(rel)
```



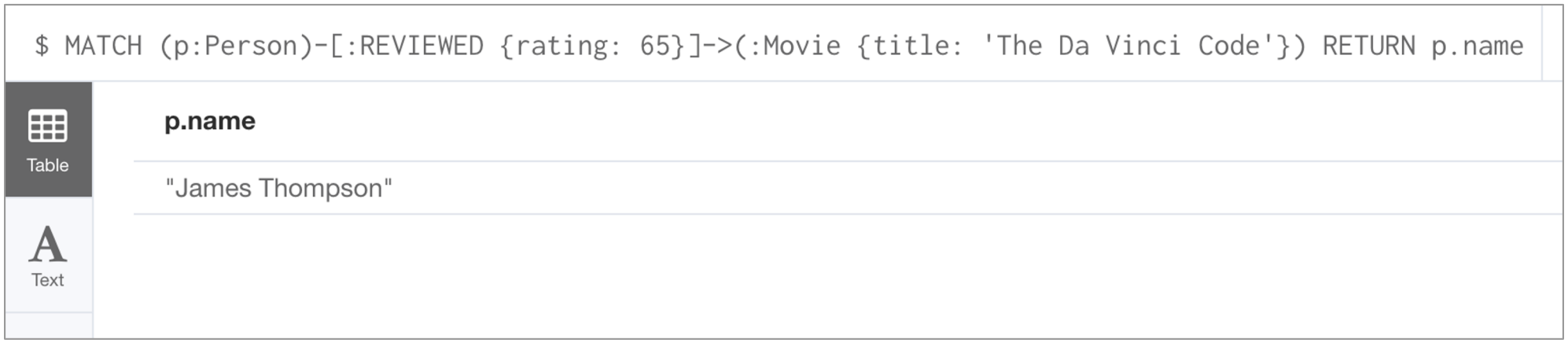
Retrieving properties for a relationship - 1



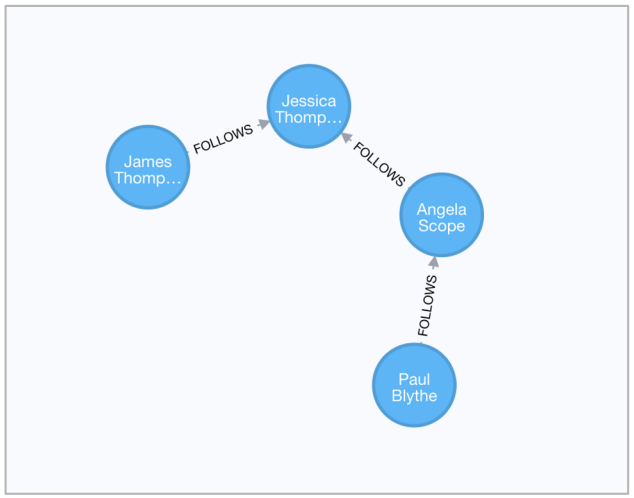
Retrieving properties for a relationship - 2

Find all people who gave the movie, *The Da Vinci Code*, a rating of *65*, returning their names:

```
MATCH (p:Person)-[:REVIEWED {rating: 65}]->(:Movie {title: 'The Da Vinci Code'})
RETURN p.name
```

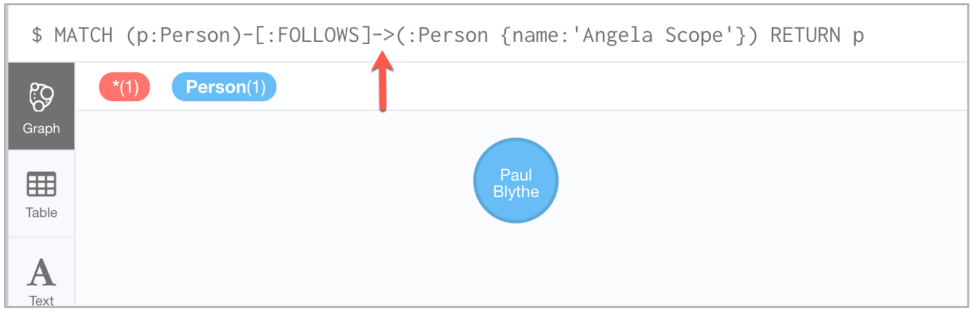


Using patterns for queries - 1



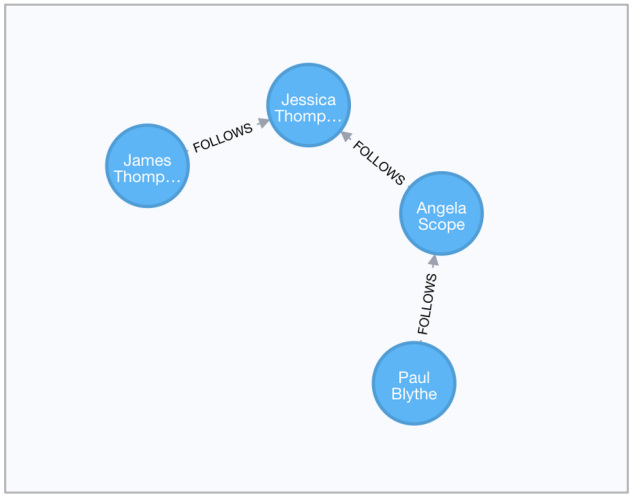
Find all people who follow *Angela Scope*, returning the nodes:

```
MATCH  (p:Person)-[:FOLLOWS]->(:Person {name:'Angela
Scope'})
RETURN p
```



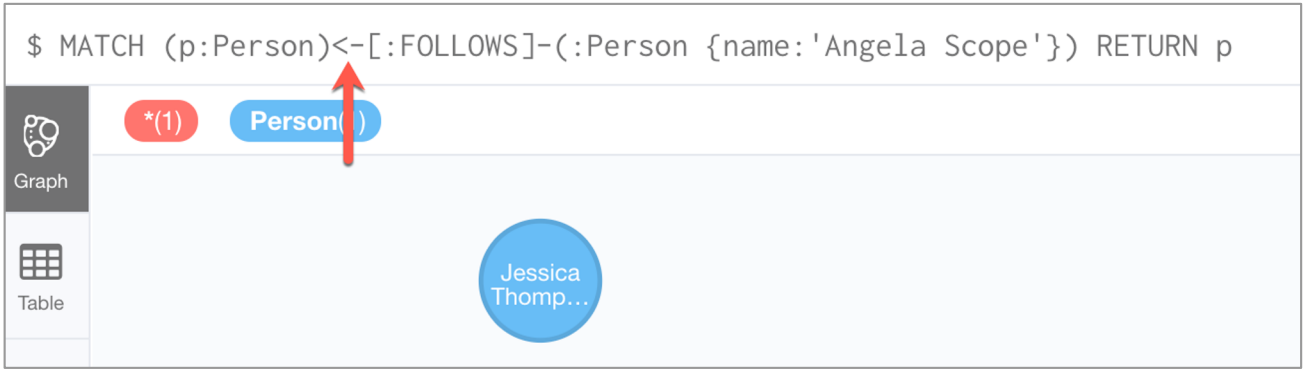
neo4j

Using patterns for queries - 2



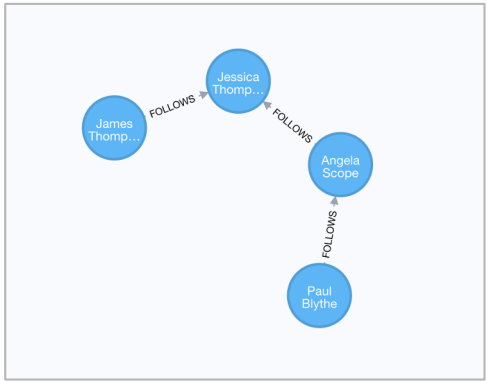
Find all people who *Angela Scope* follows, returning the nodes:

```
MATCH  (p:Person)<-[:FOLLOWS]-(:Person {name:'Angela
Scope'})
RETURN p
```



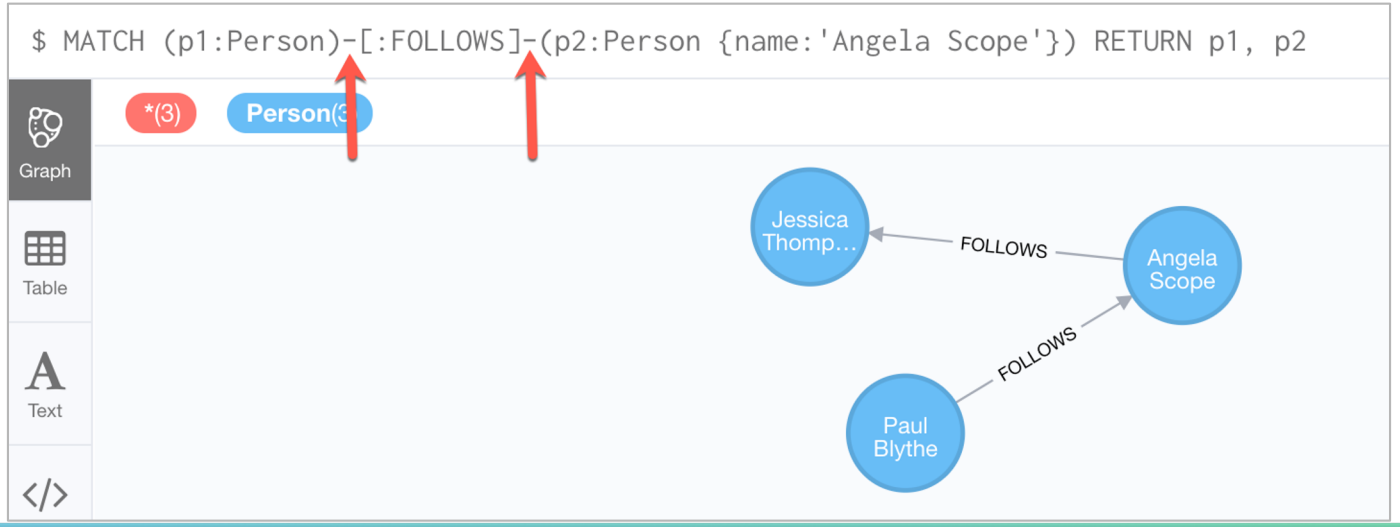
neo4j

Querying by any direction of the relationship



Find all people who follow *Angela Scope* or who *Angela Scope* follows, returning the nodes:

```
MATCH  (p1:Person)-[:FOLLOWS]-(p2:Person {name:'Angela Scope'})
RETURN p1, p2
```



neo4j


To be continued...
neo4j