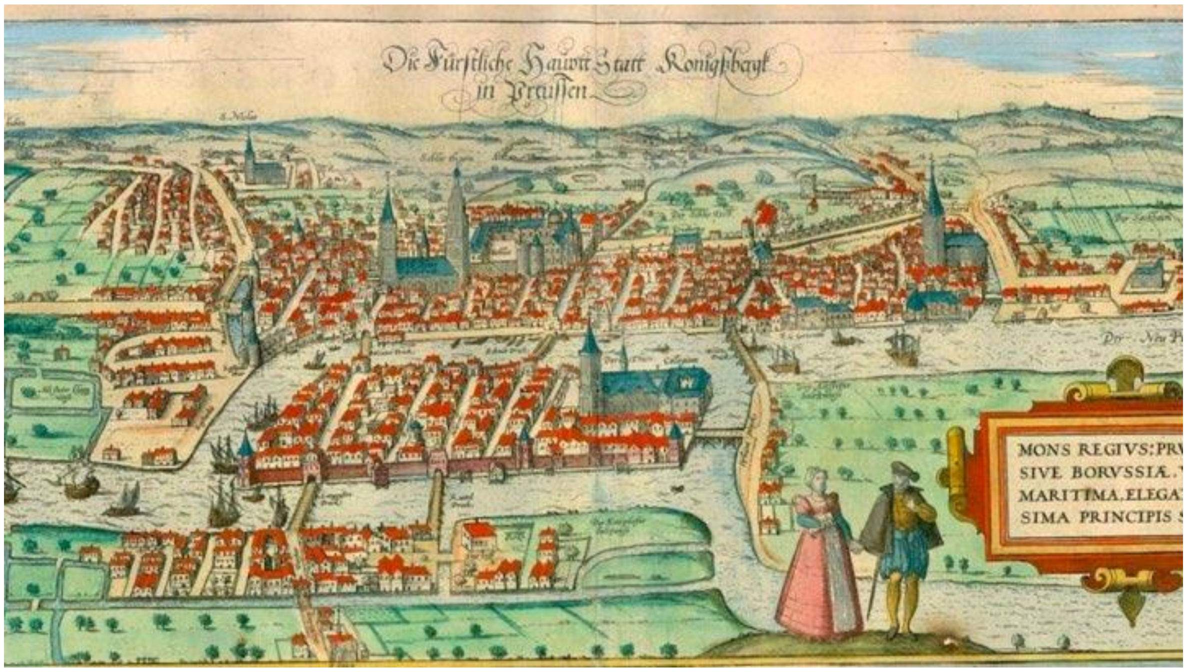
Once upon a time…

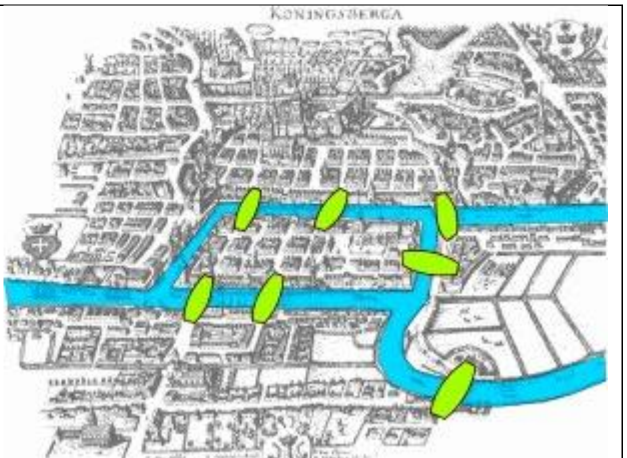
To find out how we got here, we first need to take a trip back in time.

It’s 1736, in Königsberg, Prussia.



Leonhard Euler is presumably sitting at his desk, with a dilemma. Kongsberg (modern day Kaliningrad, Russia) is divided by the Pregel River into four sections which are connected by seven bridges.

The question that Euler is pondering is: Can we take a walk through the city that would cross each of the seven bridges only once?



Foundation for graph theory

He eventually solved the problem by reformulating it, and in doing so laid the foundations for graph theory.

He realized that the land masses themselves weren't an important factor. In fact, it was the bridges that connected the land masses that were the most important thing.

His approach was to define the problem in abstract terms, taking each land mass and representing it as an abstract vertex or node, then connecting these land masses together with a set of seven edges that represent the bridges. These elements formed a "graph".

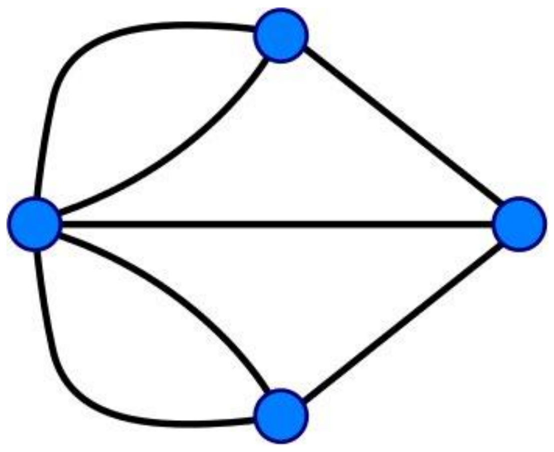
Leonhard Euler is presumably sitting at his desk, with a dilemma. Kongsberg (modern day Kaliningrad, Russia) is divided by the Pregel River into four sections which are connected by seven bridges.

The question that Euler is pondering is: Can we take a walk through the city that would cross each of the seven bridges only once?

Applying the theory

A Graph representation of the SevenBridges of Königsberg problem

Using this abstraction, Euler was able to definitively demonstrate that there was no solution to this problem. Regardless of where you enter this graph, and in which order you take the bridges, you can't travel to every land mass without taking one bridge at least twice.



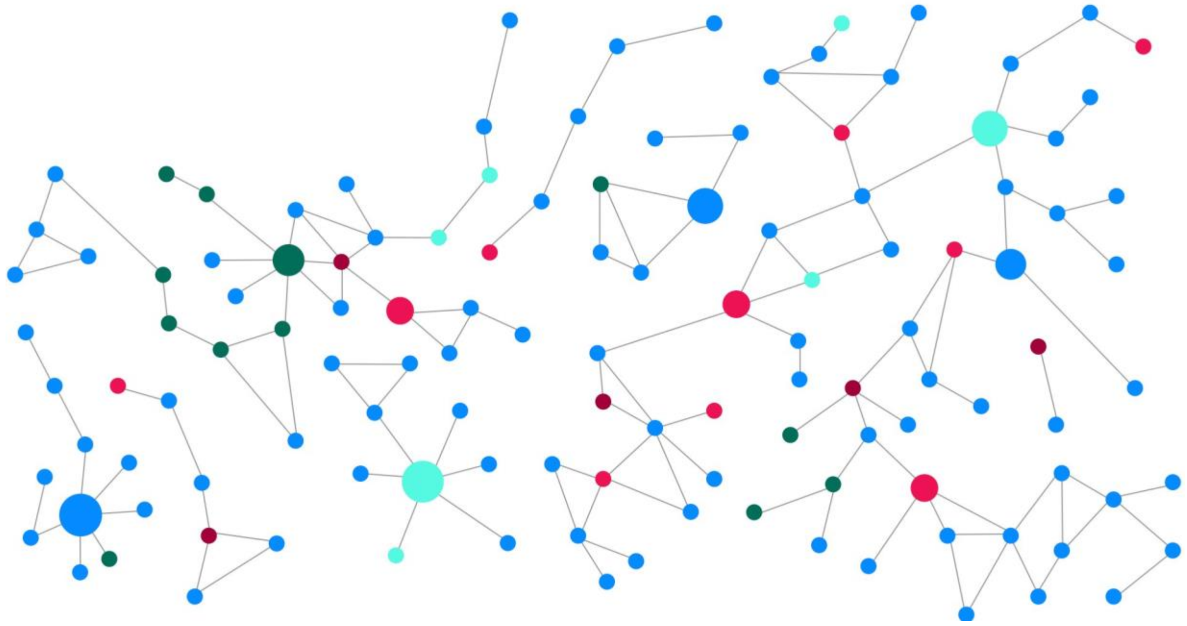
But it wasn't a completely wasted effort. Although graphs originated in mathematics, they are also a very convenient way of modeling and analyzing data. While there is certainly value in the data that we hold, it is the connections between data that can really add value. Creating or inferring relationships between your records can yield real insights into a dataset.

Fast forward 300 years and these founding principles are used to solve complex problems including route finding, supply chain analytics, and real-time recommendations.

Graph elements

Let's take a closer look at the two elements that make up a graph:

- Nodes (also known as vertices)
- Relationships (also known as edges)



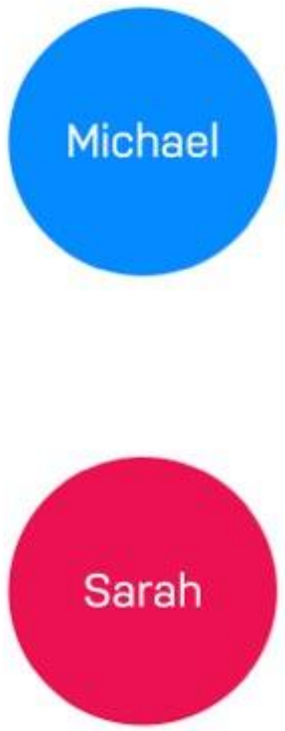
Nodes

Nodes (or vertices) are the circles in a graph. Nodes commonly represent *objects*, *entities*, or merely *things*.

In the **Seven Bridges of Königsberg** example in the previous lesson, nodes were used to represent the land masses.

Another example that everyone can relate to is the concept of a social graph. People interact with each other and form relationships of varying strengths.

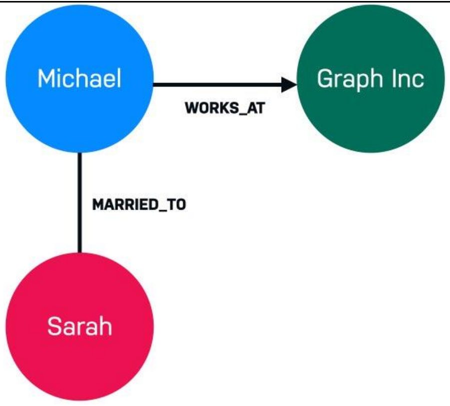
The diagram to the right has two nodes which represent two people, **Michael** and **Sarah**. On their own, these elements are uninspiring. But when we start to connect these circles together, things start to get interesting.

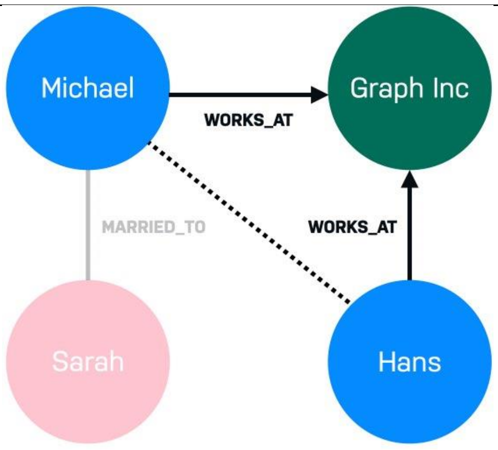


Nodes typically represent things

Examples of entities that could typically be represented as a node are: person, product, event, book or subway station.

Relationships

	**Relationships** (or *edges*) are used to connect nodes. We can use relationships to describe how nodes are connected to each other. For example **Michael** has the **WORKS_AT** relationship to **Graph Inc** because he works there. **Michael** has the **MARRIED_TO** relationship to **Sarah** because he is married to her. All of a sudden, we know that we are looking at the beginnings of some sort of *social graph*. Now, let's introduce a third person, **Hans**, to our Graph.

Hans also *works for* **Graph Inc** along with Michael. Depending on the size of the company and the properties of the relationship, we may be able to infer that Michael and Hans know each other. If that is the case, how likely is it that Sarah and Hans know each other? These are all questions that can be answered using a graph.	 Michael Graph Inc WORKS_AT MARRIED_TO WORKS_AT Sarah Hans

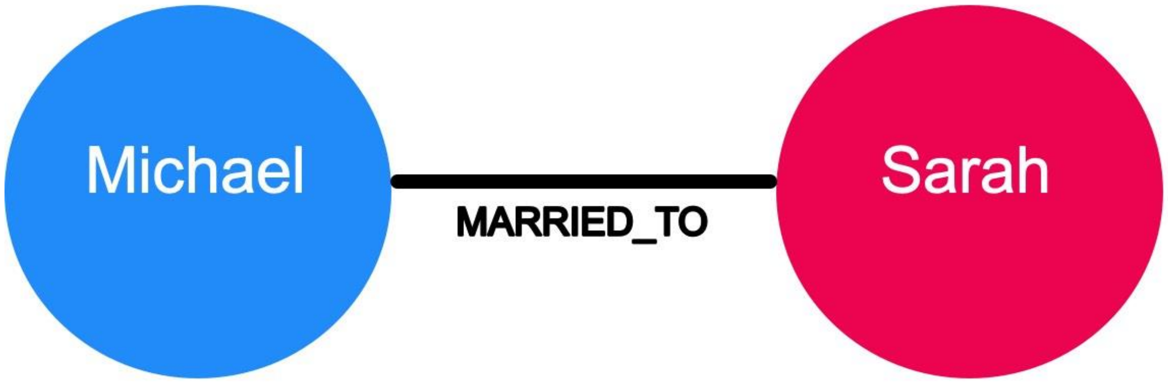
Relationships are typically verbs.

We could use a relationship to represent a personal or professional connection (*Person* ***knows*** *Person*, *Person* ***married to*** *Person*), to state a fact (*Person* ***lives in*** *Location*, *Person* ***owns*** *Car*, *Person* ***rated*** *Movie*), or even to represent a hierarchy (*Parent* ***parent of*** *Child, Software* ***depends on*** *Library*).

Graph characteristics and traversal

There are a few types of graph characteristics to consider. In addition, there are many ways that a graph may be traversed to answer a question.

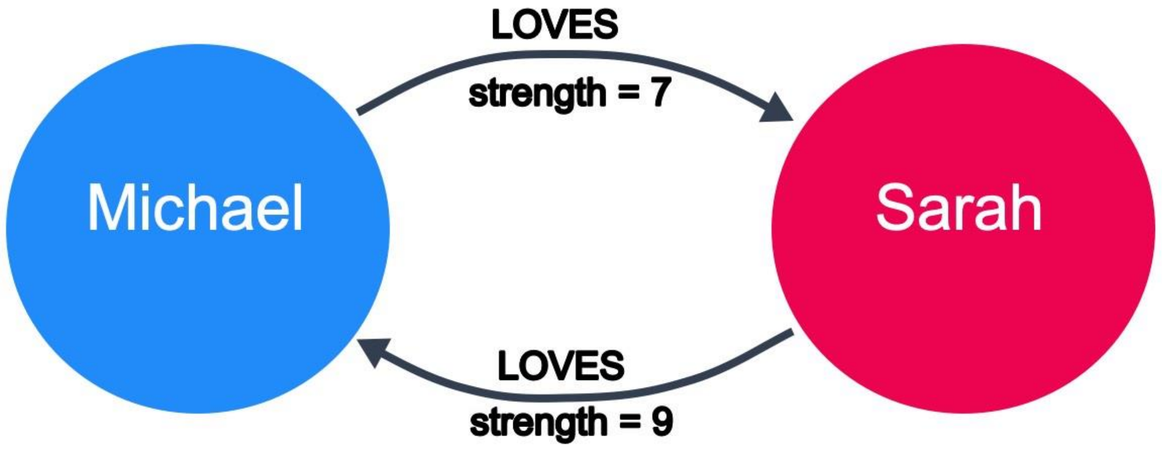
Directed vs. undirected graphs



In an undirected graph, relationships are considered to be bi-directional or symmetric.

An example of an undirected graph would include the concept of marriage. If *Michael* is married to *Sarah*, then it stands to reason that *Sarah* is also married to *Michael*.

A directed graph adds an additional dimension of information to the graph. Relationships with the same type but in opposing directions carry a different semantic meaning.



For example, if marriage is a symmetrical relationship, then the concept of love is asymmetrical. Although two people may like or love each other, the amount that they do so may vary drastically. Directional relationships can often be qualified with some sort of weighting. Here we see that the strength of the LOVES relationship describes how much one person loves another.

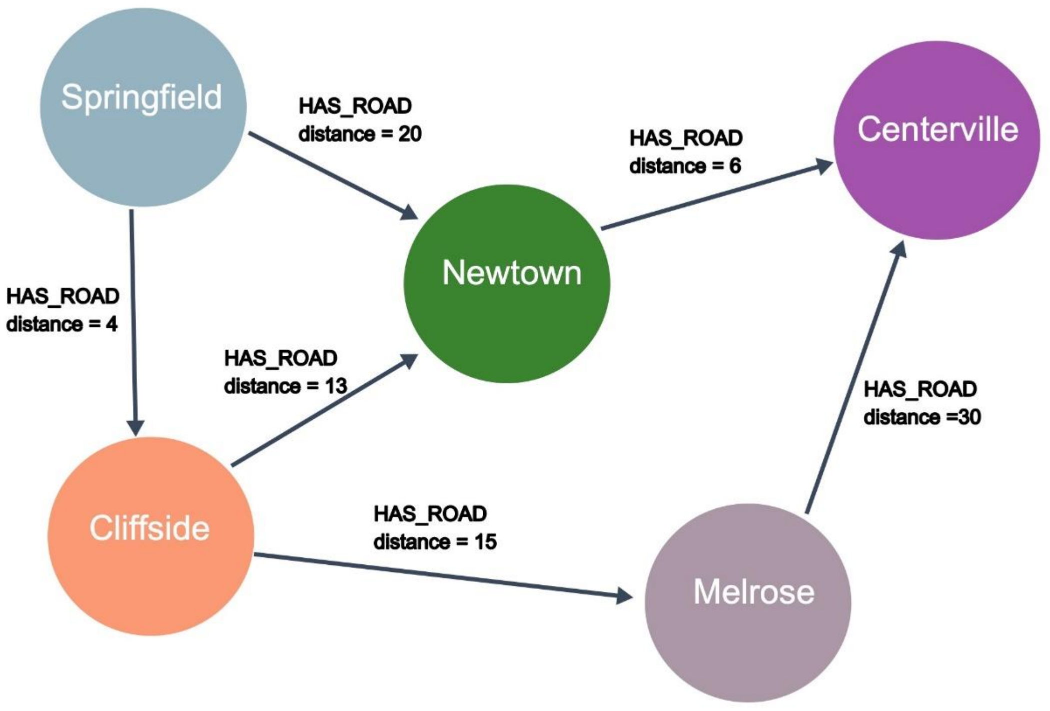
At a larger scale, a large network of social connections may also be used to understand network effects and predict the transfer of information or disease. Given the strength of connections between people, we can predict how information would spread through a network.

Weighted vs. unweighted graphs

The concept of love is also an example of a weighted graph.

In a weighted graph, the relationships between nodes carry a value that represents a variety of measures, for example cost, time, distance or priority.

A basic shortest path algorithm would calculate the shortest distance between two nodes in the graph. This could be useful for finding the fastest walking route to the local store or working out the most efficient route to travel from city to city.



In this example, the question that we might have for this graph is: What is the shortest drive from Springfield to Centerville? Using the *HAS_ROAD* relationships and the distance for these relationships, we can see that the shortest drive will be to start in Springfield, then go to Cliffside, then to Newtown, and finally arrive in Centerville.

More complex shortest path algorithms (for example, Dijkstra's algorithm or A* search algorithm) take a weighting property on the relationship into account when calculating the shortest path. Say we have to send a package using an international courier, we may prefer to send the package by air so it arrives quickly, in which case the weighting we would take into account is the time it takes to get from one point to the next.

Inversely, if cost is an issue we may prefer to send the package by sea and therefore use a property that represents cost to send the package.

Graph traversal

How one answers questions about the data in a graph is typically implemented by traversing the graph. To find the shortest path between Springfield to Centerville, the application would need to traverse all paths between the two cities to find the shortest one.

- Springfield-Newtown-Centerville = 26
- Springfield-Cliffside-Newtown-Centerville = 23
- Springfield-Cliffside-Melrose-Certerville = 49

Traversal implies that the relationships are followed in the graph. There are different types of traversals in graph theory that can impact application performance. For example, can a relationship be traversed multiple times or can a node be visited multiple times?

Neo4j's Cypher statement language is optimized for node traversal so that relationships are not traversed multiple times, which is a huge performance win for an application.

Graphs Are Everywhere

VideoTranscript

Use cases for graphs

As we discovered previously, the fundamental structure of a graph has applications far beyond mathematics. In fact, you may have seen the phrase **Graphs are Everywhere** across the **neo4j.com** website. It is our hope that you will start to see the connections between things everywhere.

Neo4j hosts a site that contains example graphs (data models) that have been designed by Neo4j engineers and Neo4j Community members. You can browse the graphgists by use case or industry. You can also use a graphgist as a starting point for your application’s graph.

Explore the Neo4j Graphgists.

Here are a some commonly-used use cases for Neo4j.

E-commerce and real-time recommendations

Many online stores are traditionally built and run on relational databases. But by adding a graph database, either as a primary data store or as an additional data store, we can start to serve real time recommendations.

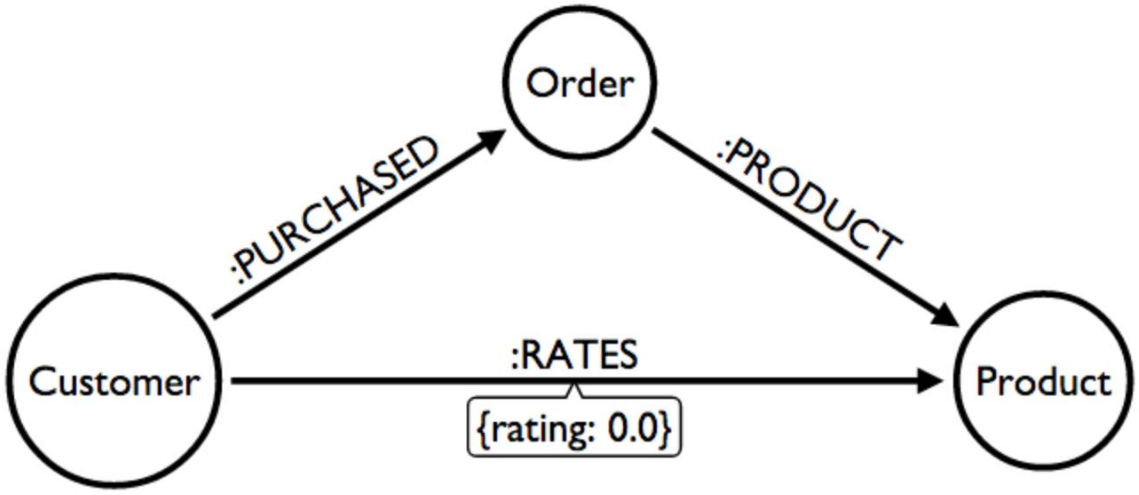
The first area that can be improved in e-commerce is the category hierarchy. To find products in a parent and subsequent child categories can be difficult in a traditional SQL query, or require the duplication of data. Conversely, this can be represented in a couple of lines of Cypher:

```cypher
MATCH (c:Category)-[:HAS_CHILD|HAS_PRODUCT*1..3]->(p:Product)
RETURN p.id, p.title, collect(c.name) AS categories
```

You may also be familiar with the **People who bought *{Product A}* also bought…** sections on your favorite online store. These types of recommendations can be computationally expensive to generate due to the large amount of data that needs to be held in memory. This creates the need for batch processes to be deployed in order to generate the recommendations.

Where graph databases have the advantage in this use case, is that a much smaller proportion of the graph needs to be traversed in order to generate the recommendation. You can simply traverse from one Product node, through the users who have purchased that product and onwards to the subsequent products that they have bought.

Given the existing data in the graph about Customers, Orders, and Products, we can infer the rating for a product based upon the number of times the customer ordered a product.

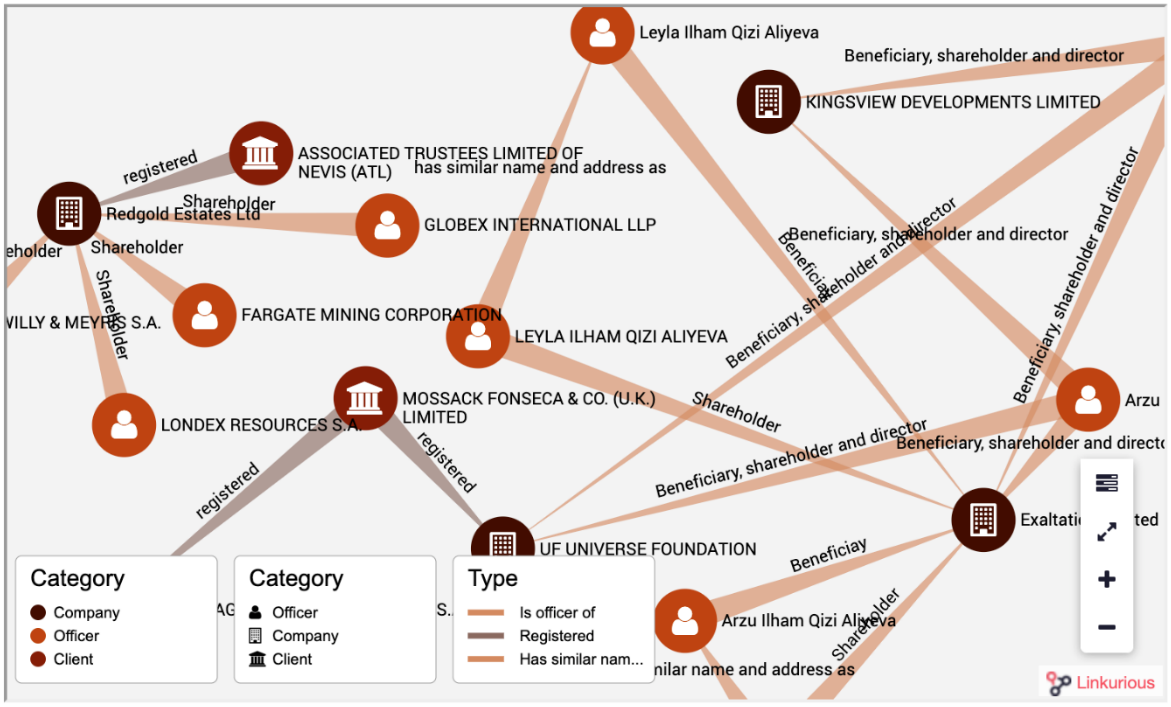


This uses case is described in the Neo4j GraphGist site. **[View the Northwind Recommendation Engine example GraphGist]**

Investigative journalism

The most prominent user of Neo4j for investigative journalism is the International Consortium of Investigative Journalists (**ICIJ**). One such graph that was created by the ICIJ was the Panama Papers. The purpose of this graph was to identify possible corruption based upon the relationships between people, companies, and most importantly financial institutions.

We have a subset of the Panama Papers investigation in a **Neo4j Graphgist** representing the family of the Azerbaijan's President Ilham Aliyev.



The purpose of this graph to enable one to answer these questions:

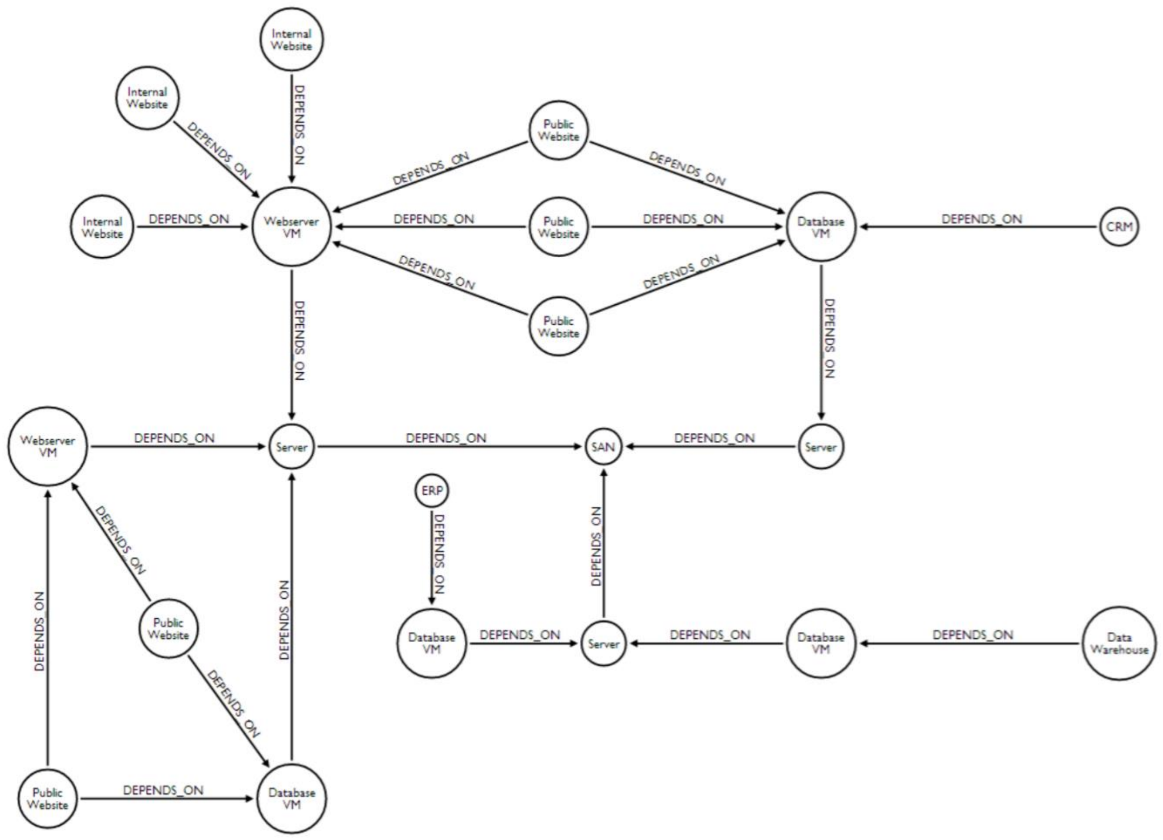
- What families with the name that contains the string 'aliye' are Officers of Companies?
- How is the family with the name that contains the string 'aliye' related to Companies?
- How are Officers related to each other?
- What are the connections between multiple companies and a family?

Another graph that has been created by the ICIJ contains information on almost 350,000 offshore entities that are part of the Paradise and Panama Papers and the Offshore Leaks investigations. The Offshore Leaks data exposes a set of connections between people and offshore entities. You can play with this graph by creating a **Paradise Papers Sandbox** and querying the data.

Network and IT operations

Many enterprises use Neo4j to help them understand how information flows through a system and how components of a network are related. This is useful for planning, analysis of costs, and also to troubleshoot problems when a problem arises.

One of our Neo4j Community members contributed this sample data model to demonstrate how one might use a graph to identify network dependencies. Here is the data model:

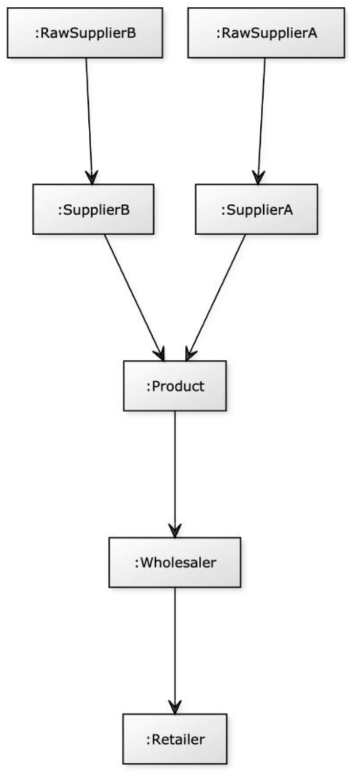


You can use this type of data model to answer:

- What are the direct dependencies of public websites?
- What are the direct dependencies of internal websites?
- What is the most depended-upon component?
- Find the dependency chain for a business critical component.
- What is the impact of removing a server?

View the Network Dependency Graphgist.

Transportation and logistics



Here is an example data model contributed by a Neo4j Community member related to supply chain management. Entities that are modeled include raw suppliers, suppliers, products, wholesalers, and retailers. All of these entities are located somewhere and there is a distance between them that will impact how quickly products can be transported.

With this graph, one can answer these questions:

- Who is the best wholesaler for each retailer based upon distance?
- Which raw supplier will give a particular retailer the freshest products?
- Which retailer provides locally grown products?
- How can we rate each supply chain?

View the Supply Chain Management Graphgist.

Property Graphs

Module Overview

In this module, you will learn about:

- The key elements a property graph and why they are useful.
- How Neo4j implements a native graph using index-free adjacency.
- Some of the ways that Relational and other NoSQL databases can be implemented in Neo4j.

Property graph

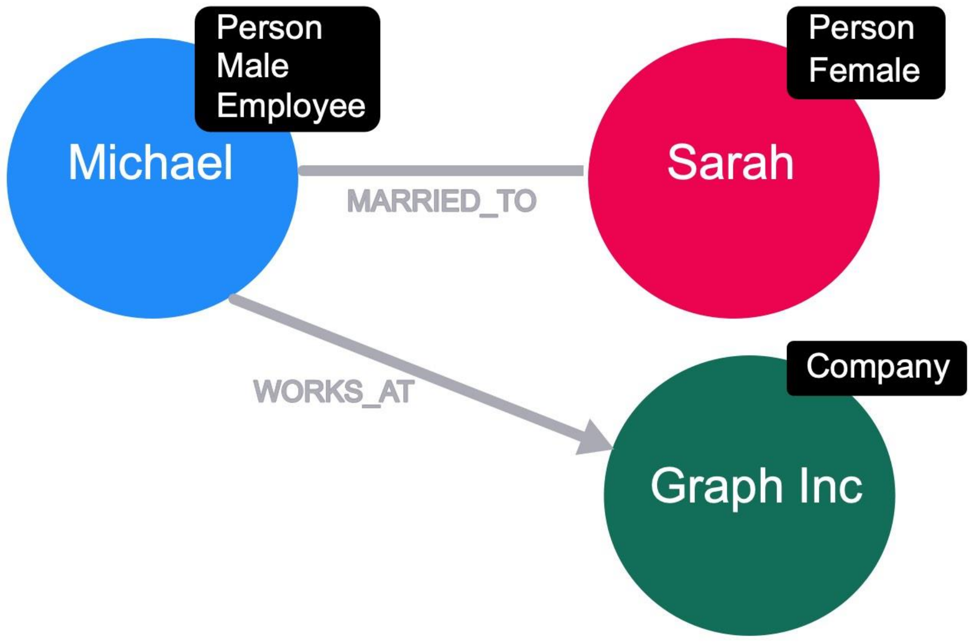
In the previous module we referred to nodes and relationships as the fundamental building blocks for a graph. In this lesson you will learn about the additional elements that Neo4j supports to make a **property graph**.

Nodes, Labels and Properties

Recall that nodes are the graph elements that represent the *things* in our data. We can use two additional elements to provide some extra context to the data.

Let’s take a look at how we can use these additional elements to improve our social graph.

Labels



By adding a label to a node, we are signifying that the node belongs to a subset of nodes within the graph. Labels are important in Neo4j because they provide a starting point for a Cypher statement.

Let's take **Michael** and **Sarah** - in this context both of these nodes are **persons**.

We can embellish the graph by adding more labels to these nodes; Michael identifies as **male** and Sarah is **female**. In this context, Michael is an **employee** of a company, but we don't have any information about Sarah's employment status.

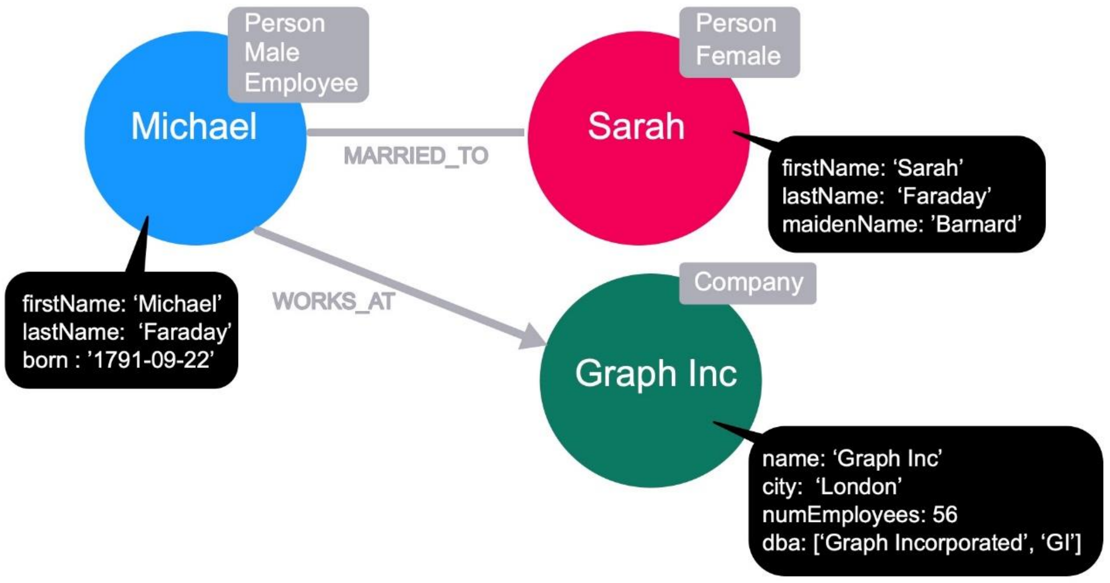
Michael works for a **company** called Graph Inc, so we can add that label to the node that represents a company.

In Neo4j, a node can have zero, one, or many labels.

Node properties

So far we're assuming that the nodes represent Michael, Sarah, and Graph Inc. We can make this concrete by adding properties to the node.

Properties are key, value pairs and can be added or removed from a node as necessary. Property values can be a single value or list of values **that conform to the Cypher type system**.



By adding *firstName* and *lastName* properties, we can see that the Michael node refers to **Michael Faraday**, known for Faraday's law of induction, the Faraday cage and lesser known as the inventor of the Party Balloon. Michael was *born* on 22 September 1791.

Sarah's full name is **Sarah Faraday**, and her *maidenName* is **Barnard**.

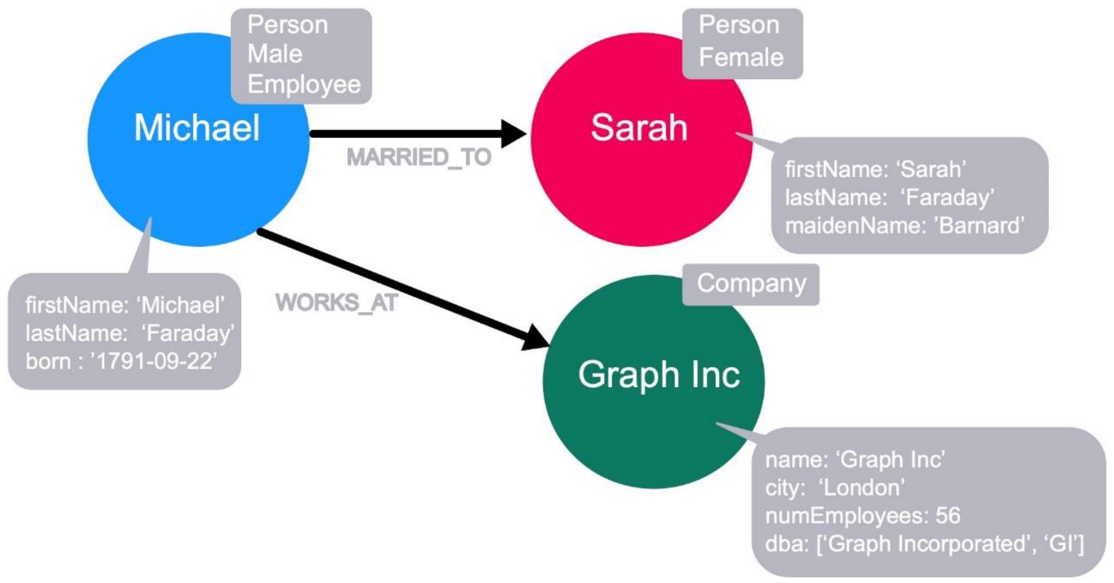
By looking at the *name* property on the Graph Inc node, we can see that it refers to the company **Graph Inc**, with a *city* of **London**, has 56 employees (*numEmployees*), and does business as Graph Incorporated and GI (*dba*).

Properties do not need to exist for each node with a particular label. If a property does not exist for a node, it is treated as a `null` value.

Relationships

A relationship in Neo4j is a connection between two nodes.

Relationship direction



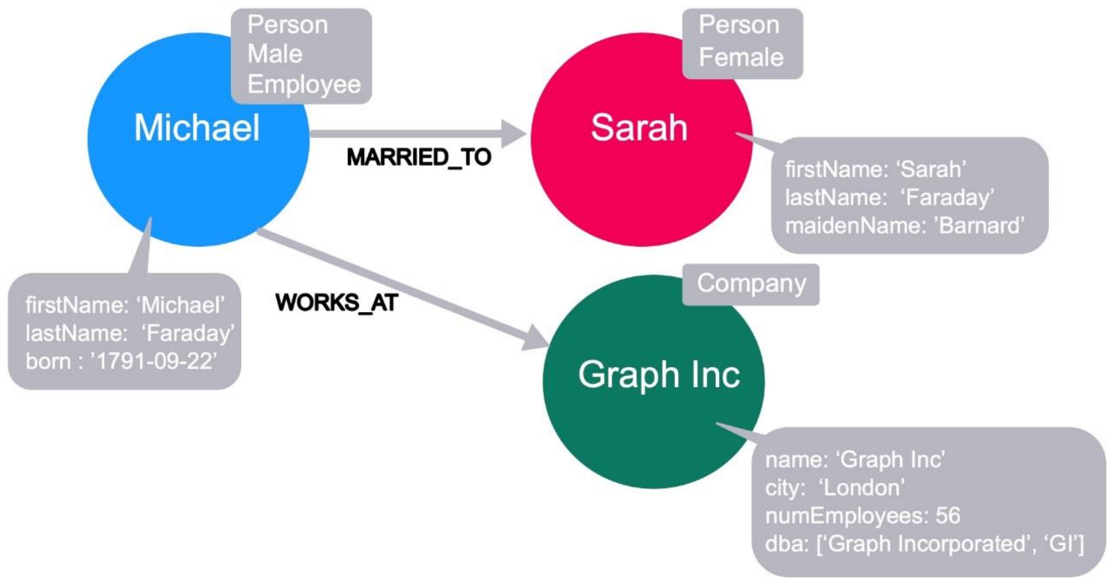
In Neo4j, each relationship **must** have a direction in the graph. Although this direction is required, the relationship can be queried in either direction, or ignored completely at query time.

A relationship is created between a **source node** and a **destination node**, so these nodes must exist before you create the relationship.

If we consider the concept of directed & undirected graphs that we discussed in the previous module, the direction of the *MARRIED_TO* relationship must exist and may provide some additional context but can be ignored for the purpose of the query. In Neo4j, the *MARRIED_TO* relationship must have a direction.

The direction of a relationship can be important when it comes to hierarchy, although whether the relationships point up or down towards the tree is an arbitrary decision.

Relationship type



Each relationship in a neo4j graph **must** have a type. This allows us to choose at query time which part of the graph we will traverse.

For example, we can traverse through *every* relationship from Michael, or we can specify the *MARRIED_TO* relationship to end up only at Sarah's node.

Here are sample Cypher statement statements to support this:

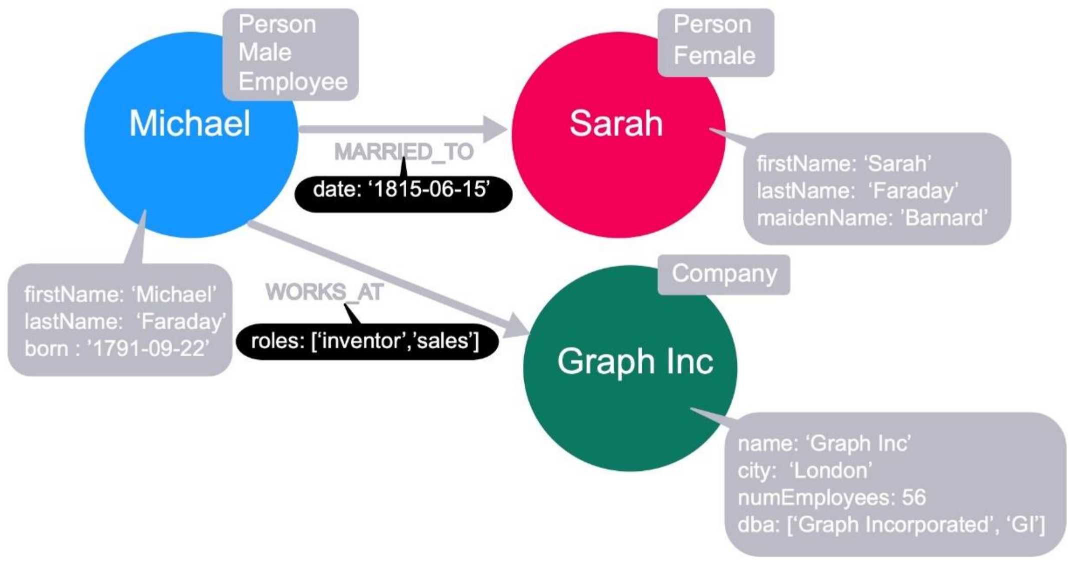
```cypher
// traverse the Michael node to return the Sarah node
MATCH (p:Person {firstName: 'Michael'})-[:MARRIED_TO]-(n) RETURN n;

// traverse the Michael node to return the Graph Inc node
MATCH (p:Person {firstName: 'Michael'})-[:WORKS_AT]-(n) RETURN n;

// traverse all relationships from the Michael node
// to return the Sarah node and the Graph Inc node
MATCH (p:Person {firstName: 'Michael'})--(n) RETURN n
```

Relationship properties

As with nodes, relationships can also have properties. These can refer to a cost or distance in a weighted graph or just provide additional context to a relationship.



In our graph, we can place a property on the *MARRIED_TO* relationship to hold the date in which Michael and Sarah were married. This *WORKS_AT* relationship has a *roles* property to signify any roles that the employee has filled at the company. If Michael also worked at another company,
his *WORKS_AT* relationship to the other company would have a different value for the *roles* property.

Native Graph Advantage

Neo4j is a native graph database

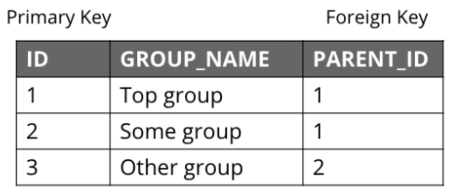
Neo4j is a native graph database, meaning that everything from the storage of the data to the query language have been designed specifically with traversal in mind. Just like any other enterprise DBMS, Neo4j is **ACID** compliant. A group of modifications in a transaction will all either commit or fail.

Where native graph databases stand apart from other databases is the concept of **index-free adjacency**. When a database transaction is committed, a reference to the relationship is stored with the nodes at both the start and end of the relationship. As each node is aware of every incoming and outgoing relationship connected to it, the underlying graph engine will simply chase pointers in memory - something that computers are exceptionally good at.

Index-free adjacency (IFA)

One of the key features that makes Neo4j graph databases different from an RDBMS is that Neo4j implements **index-free adjacency**.

RDBMS query



Primary Key   Foreign Key

ID	GROUP_NAME	PARENT_ID
1	Top group	1
2	Some group	1
3	Other group	2

To better understand the benefit of index-free adjacency, let’s look at how a query executes in an RDBMS.

Suppose you have this table in the RDBMS.

You execute this SQL query to find the third-degree parents of the group with the ID of 3:

```SQL
SELECT PARENT_ID
FROM GROUPS
WHERE ID = (SELECT PARENT_ID
    FROM GROUPS
    WHERE ID = (SELECT PARENT_ID
        FROM GROUPS
        WHERE ID = 3))
```

The result of this query is 1, but in order to determine this result, the SQL Server needed to:

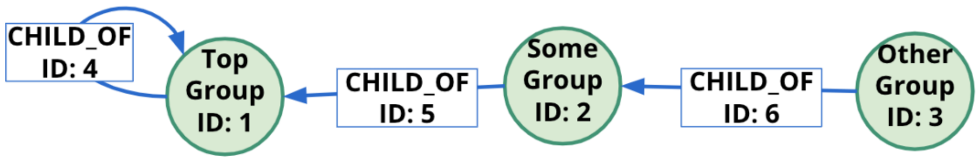
1. Locate the innermost clause.
2. Build the query plan for the subclause.
3. Execute the query plan for the subclause.
4. Locate the next innermost clause.
5. Repeat Steps 2-4.

Resulting in:

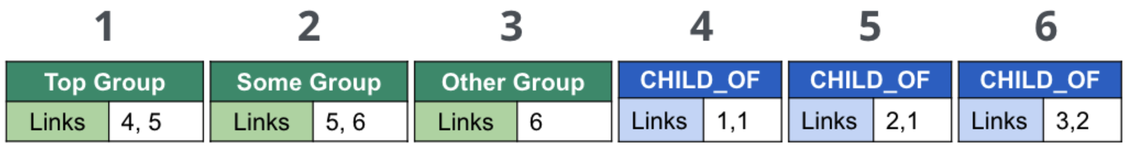
- 3 planning cycles
- 3 index lookups
- 3 DB reads

Neo4j storage

With index-free adjacency, Neo4j stores nodes and relationships as objects that are linked to each other via pointers. Conceptually, the graph looks like:



These nodes and relationships are stored as:



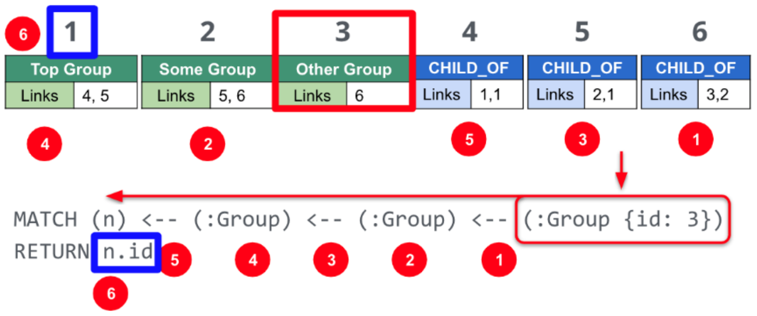
Neo4j Cypher statement

Suppose we had this query in Cypher:

```
Cypher

MATCH (n) ⟵ (:Group) ⟵ (:Group) ⟵ (:Group {id: 3})
RETURN n.id
```

Using IFA, the Neo4j graph engine starts with the anchor of the query which is the Group node with the id of 3. Then it uses the links stored in the relationship and node objects to traverse the graph pattern.


```
MATCH (n) <-- (:Group) <-- (:Group) <-- (:Group {id: 3})
RETURN n.id
```

To perform this query, the Neo4j graph engine needed to:

1. Plan the query based upon the anchor specified.
2. Use an index to retrieve the anchor node.
3. Follow pointers to retrieve the desired result node.

The benefits of IFA compared to relational DBMS access are:

- Fewer index lookups.
- No table scans.
- Reduced duplication of data.

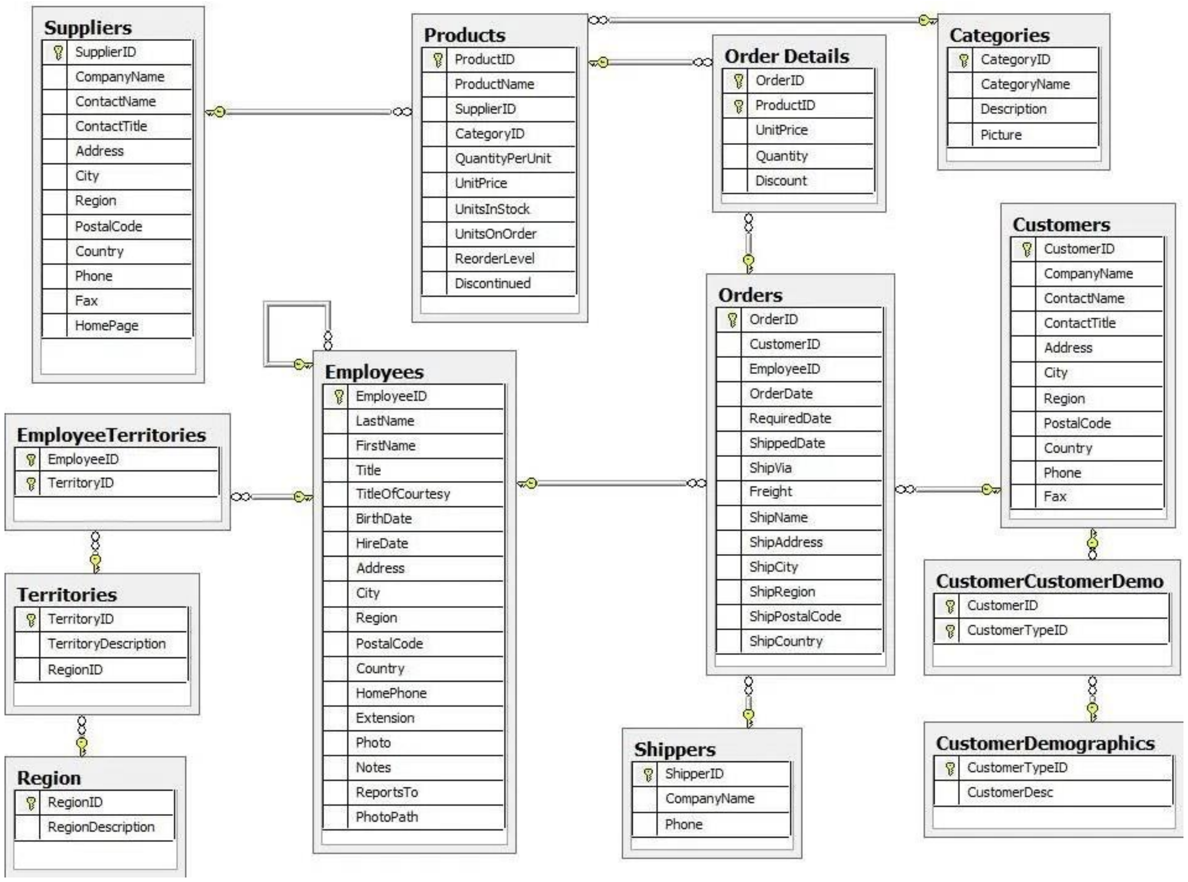
Benefit of Neo4j over Relational

As mentioned in the last lesson, index-free adjacency is a huge differentiator between relational and graph databases. While relationships are stored at write-time in a graph database, the joins made in a relational database are computed at read-time. This means that, as the number of records in a relational database increases, the slower the query becomes. The query time in a graph database will remain consistent to the size of the data that is actually touched during a query.

Having relationships treated as first class citizens also provides an advantage when starting out. Modelling relationships in a graph is more natural than creating pivot tables to represent many-to-many relationships.

Northwind RDBMS to graph

Let’s look at the Northwind RDBMS data model.



In this example, an order can contain one or more products and a product can appear in one or more orders. In a relational database, the *Order Details* table is required to handle the many-to-many relationships. The more orders added, and subsequently the larger the *Order Details* table grows, the slower order queries will become.

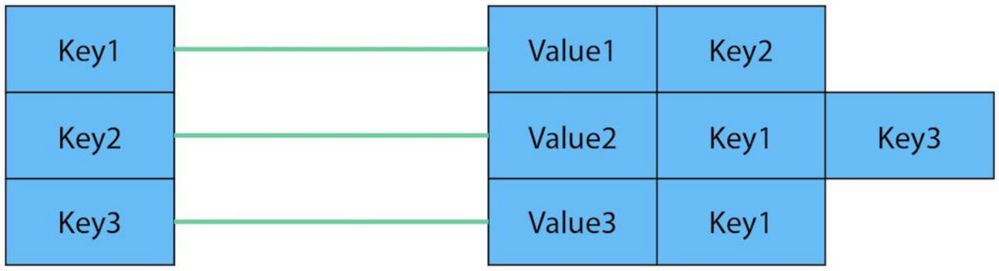
In a graph, we can simply model a *CONTAINS* relationship from the *Order* node to each *Product* node. The *Product* node has a unit price property and the *CONTAINS* relationship which has properties to represent the quantity and discount.

NoSQL datastores to graph

NoSQL databases solve many of the problems, and they are great for write throughput.

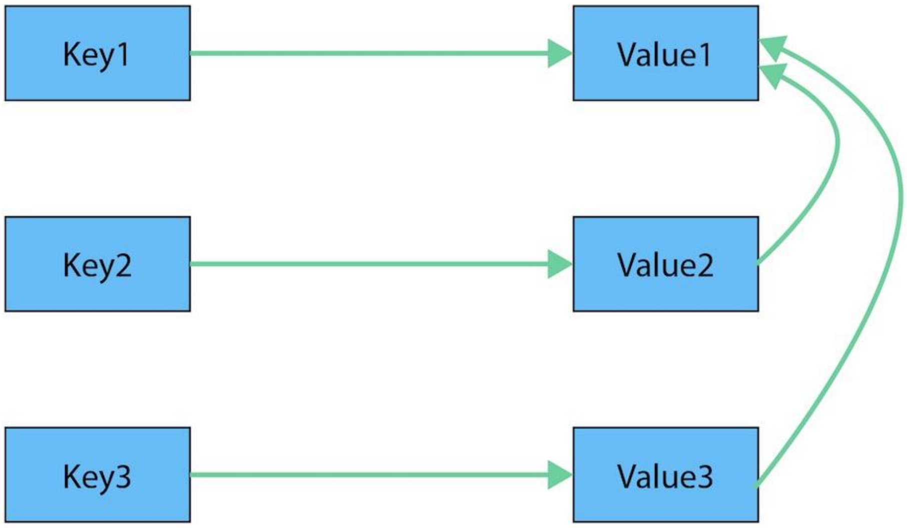
But there are problems with how data is queried. The two most common NoSQL databases represent key/value stores and documents.

Key-value stores



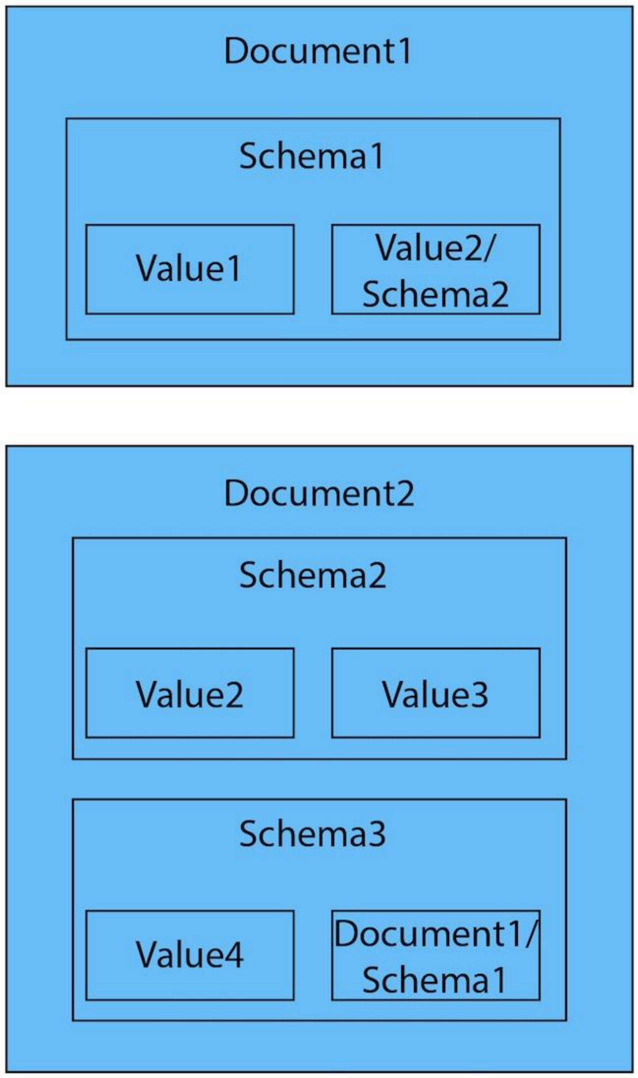
The **key-value** model is great and highly performant for lookups of huge amounts of simple or even complex values. Here is how a typical key-value store is structured.

Key-value as a graph



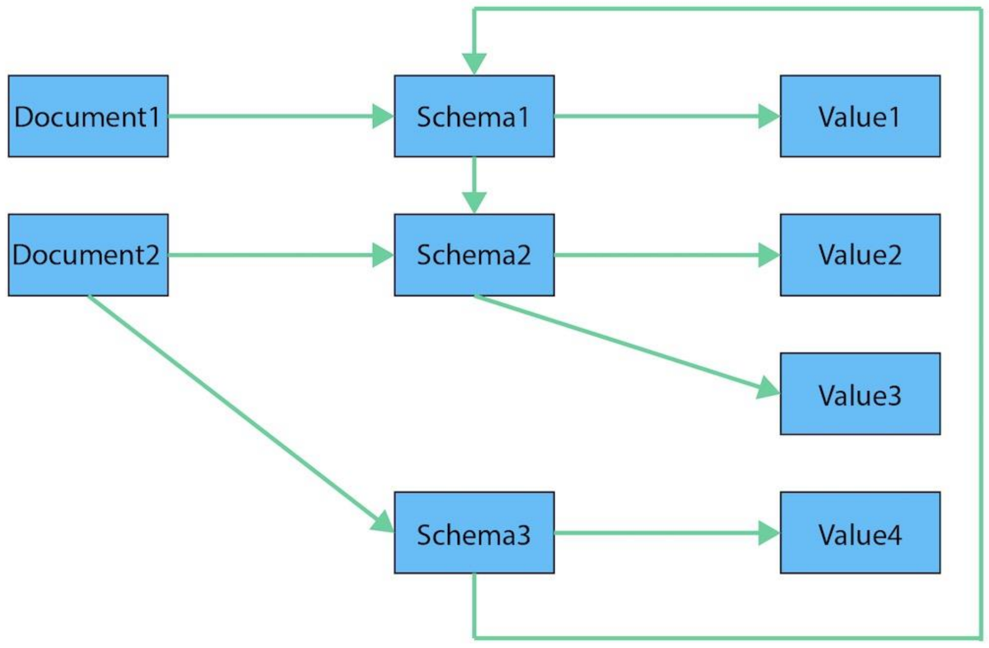
However, when the values are themselves interconnected, you have a graph. Neo4j lets you traverse quickly among all the connected values and find insights in the relationships. The graph version shows how each key is related to a single value and how different values can be related to one another (like nodes connected to one another through relationships).

Document stores



The structured hierarchy of a **Document** model accommodates a lot of schema-free data that can easily be represented as a tree. Although trees are a type of graph, a tree represents only one projection or perspective of your data. This is how a document store hierarchy is structured as pieces within larger components.

Document model as graph



If you refer to other documents (or contained elements) within that tree, you have a more expressive representation of the same data that you can easily navigate using a graph. A graph data model lets more than one natural representation emerge dynamically as needed. This graph version demonstrates how moving this data to a graph structure allows you to view different levels and details of the tree in different combinations.

Your First Graph

The data model that is used in many of our GraphAcademy courses is the Movie graph.

In this module you will be introduced to the data that is in the "starter" Movie graph.

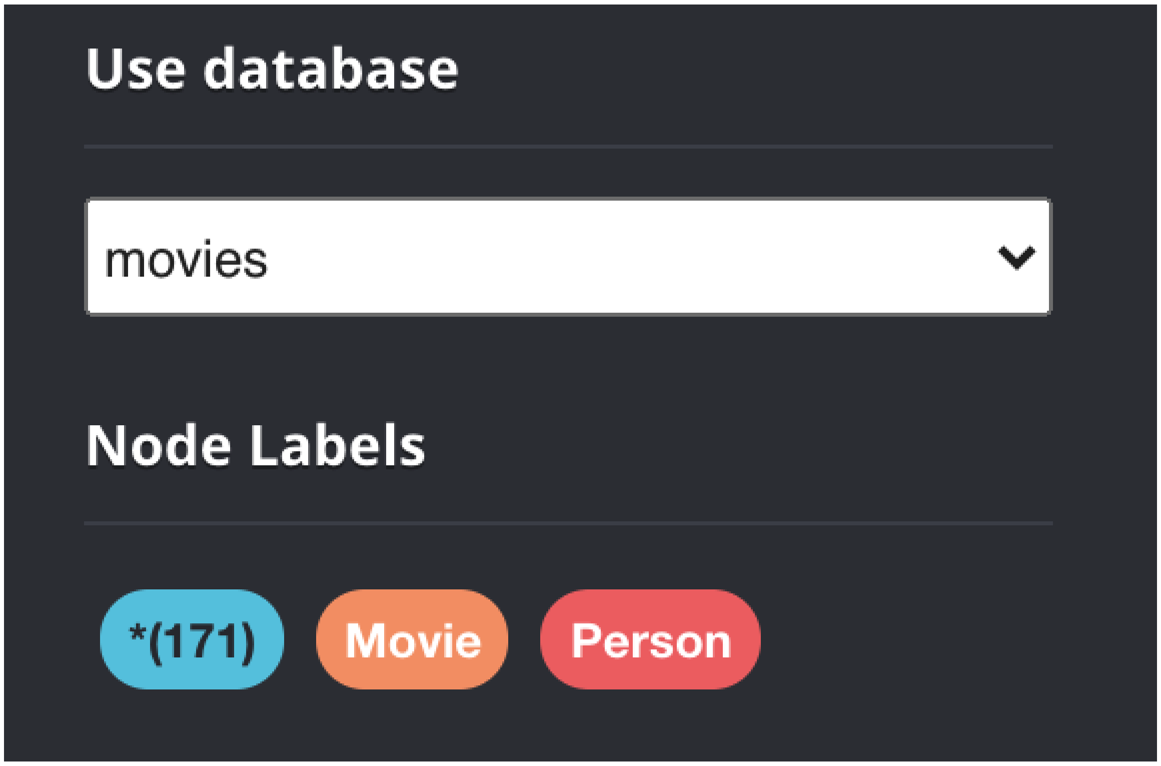
The Movie Graph

The Movie graph

Throughout the courses of GraphAcademy, you will use some version of the Movie database to gain experience with Neo4j. In this lesson you will learn about the data in the "starter" Movie database that is used when you are learning Cypher for the first time.

Nodes

The nodes in the Movie database represent people, movies, and in some versions of the Movie database, genres for the movies.



The "starter" version of the Movie database contains 171 nodes:

- 38 Movie nodes (nodes with the label Movie)
- 133 Person nodes (nodes with the label Person)

This is the database you use to first learn Cypher.

Node properties

```
{
  "identity": 10,
  "labels": [
    "Movie"
  ],
  "properties": {
"tagline": "Everything that has a beginning has an end",
"title": "The Matrix Revolutions",
"released": 2003
  }
}
```

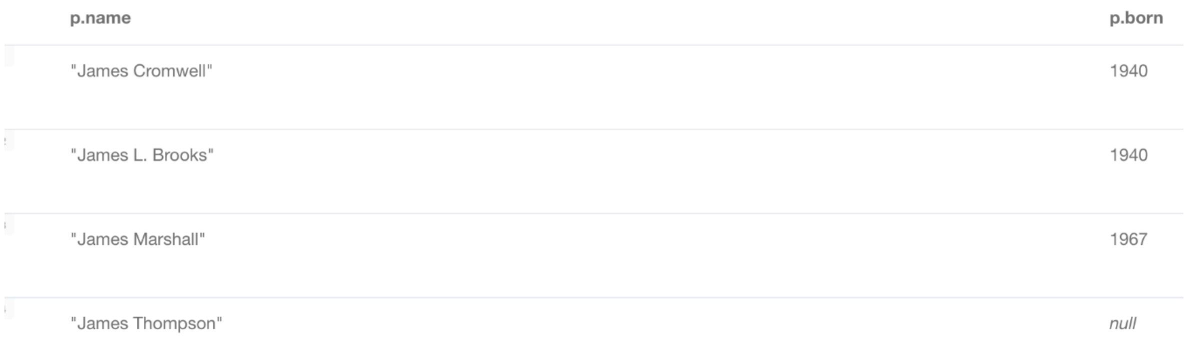
```
{
  "identity": 154,
  "labels": [
    "Movie"
  ],
  "properties": {
"title": "Something's Gotta Give",
"released": 2003
  }
}
```

All *Movie* nodes have a property, *title* that is used to uniquely identify a movie. This property exists for all *Movie* nodes.

Other properties that a *Movie* node may have are:

- *released*, the year that the movie was released.
- *tagline*, a phrase to describe the movie.

So for example, we see in these two Movie nodes, they both have a *title* and *released* property, but only one of them has a *tagline* property.



p.name	p.born
"James Cromwell"	1940
"James L. Brooks"	1940
"James Marshall"	1967
"James Thompson"	*null*

All *Person* nodes have a property, *name* that is used to uniquely identify a person. Some *Person* nodes have a property, *born*, but not all of them.

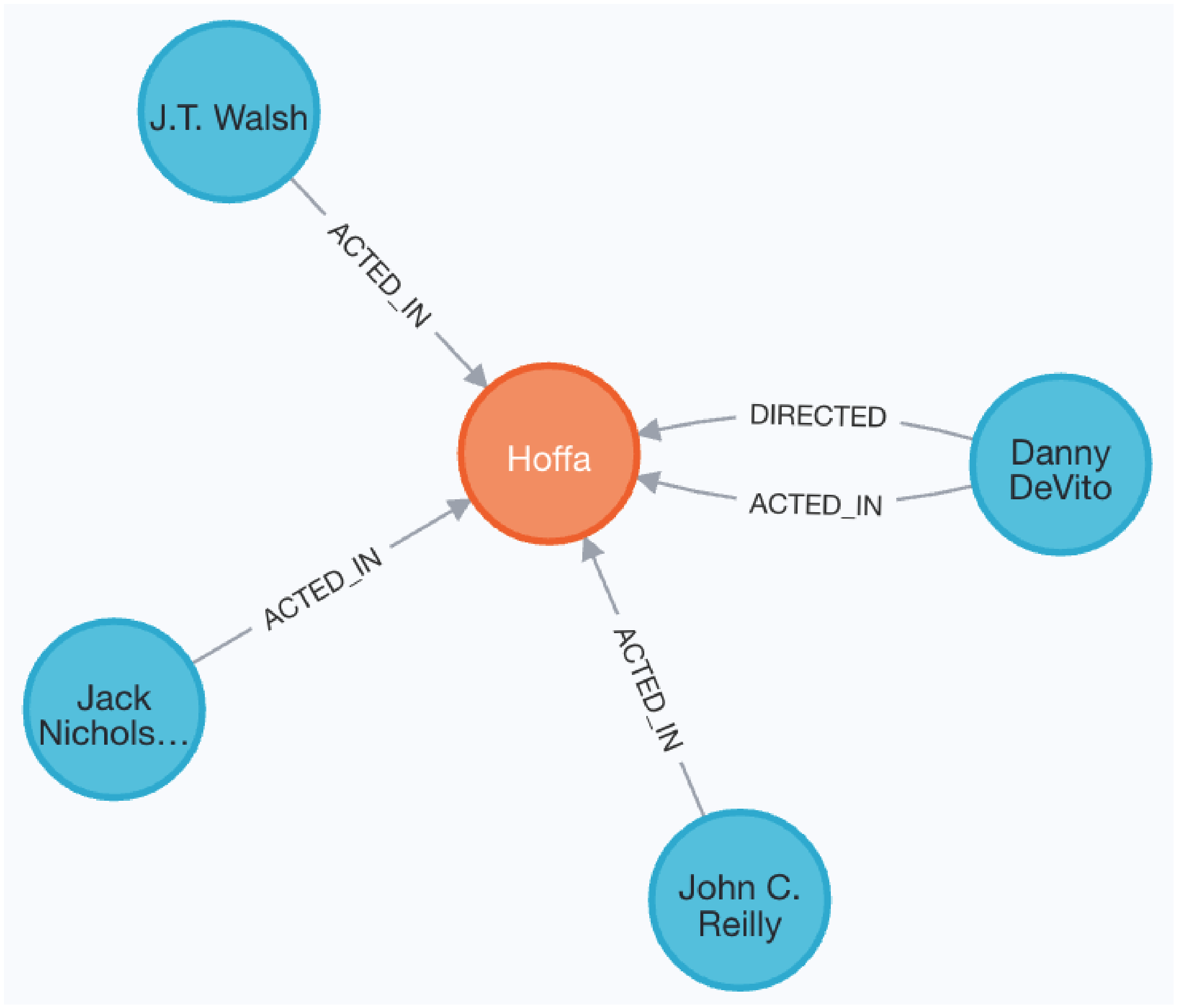
Relationships

As you have learned, the most important element of a graph database is its relationships. A relationship has a type and a direction and represents the relationship between two specific nodes.

Some of the relationships in the "starter" Movie graph include:

Relationship type	**Description**	**Number in graph**
ACTED_IN	A Person acted in a Movie	172
DIRECTED	A Person directed a Movie	44
WROTE	A Person wrote a Movie	10
PRODUCED	A Person produced a Movie	15

A person can have multiple relationships to a movie. For example, a person can be both an actor and a director for a particular movie. In the Movie graph, people are either actors, directors, writers and/or producers given these relationships.



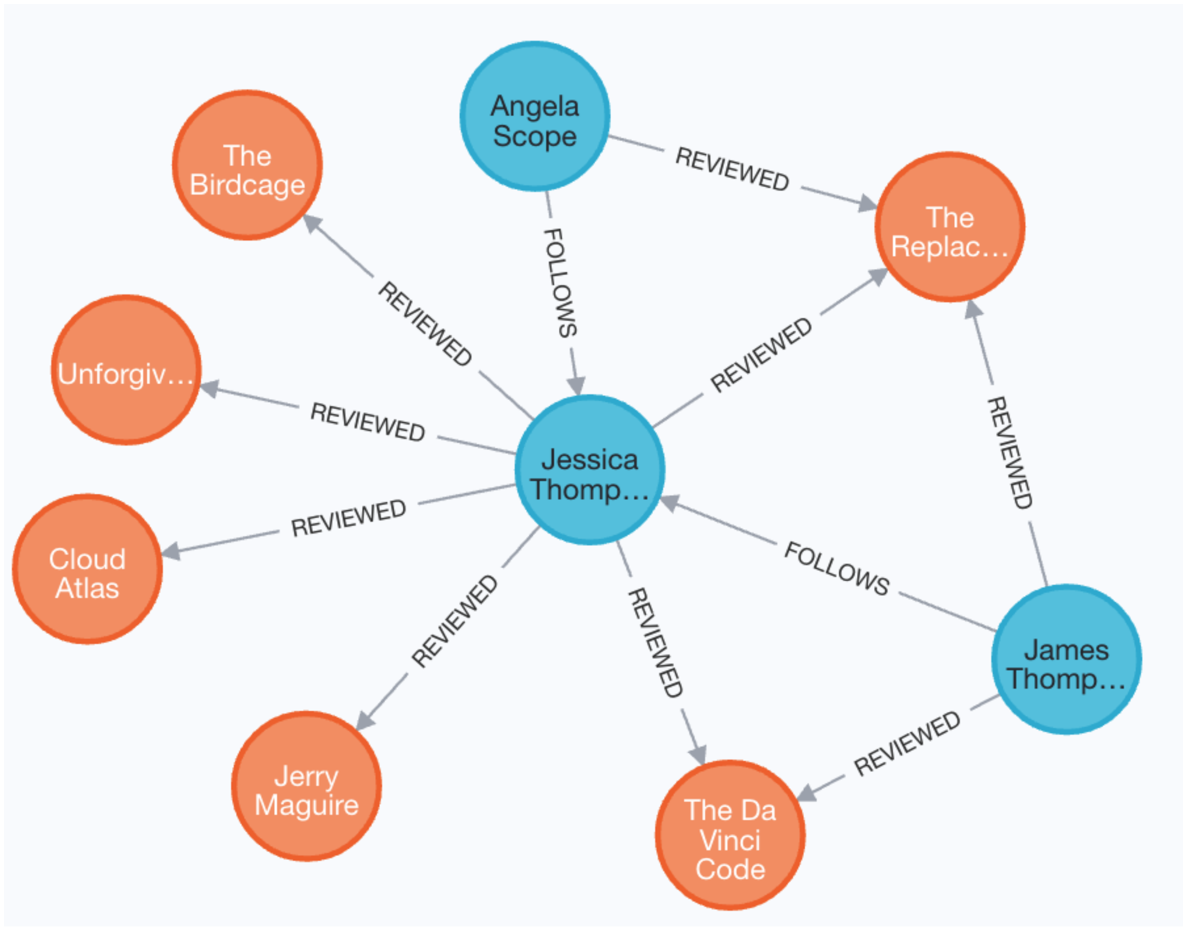
So, for example, the movie "Hoffa" in the Movie graph has these relationships. It has four actors and one director. Danny DeVito directed and acted in this movie. In our "starter" Movie graph, this movie has no writers or producers defined.

Other relationships in the graph include:

Relationship type	Description	Number in graph
REVIEWED	A Person reviewed a Movie	9
FOLLOWS	A Person follows another Person	3

Using these relationships, people can be reviewers, followers, or followees. In the Movie graph, people who review movies or follow other people are **not** actors, directors, writers, or producers.

Here are the reviewers in our "starter" Movie graph:

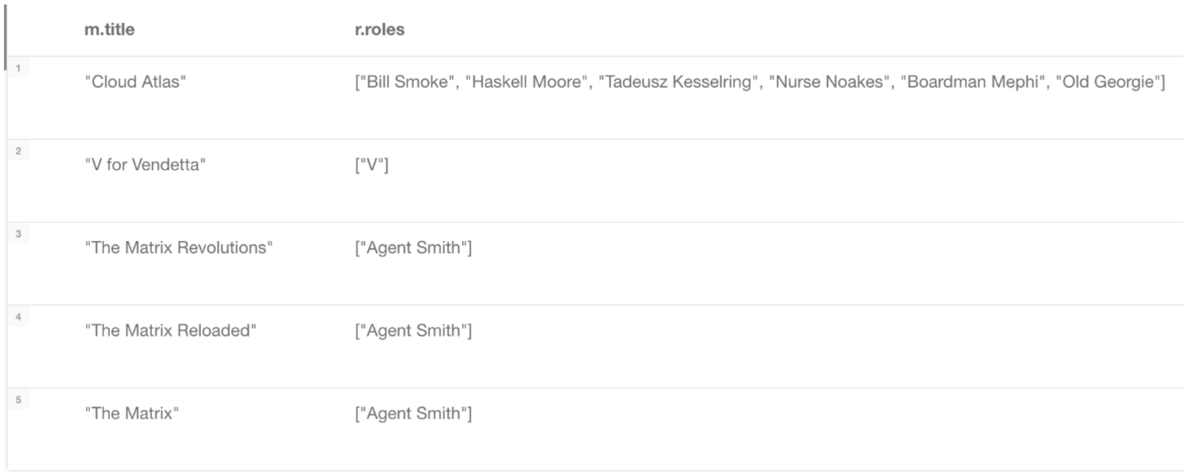


We have three *Person* nodes here for people who reviewed movies. All three of these reviewers reviewed the movie, The Replacements. Two people here are following Jessica Thompson.

Relationship properties

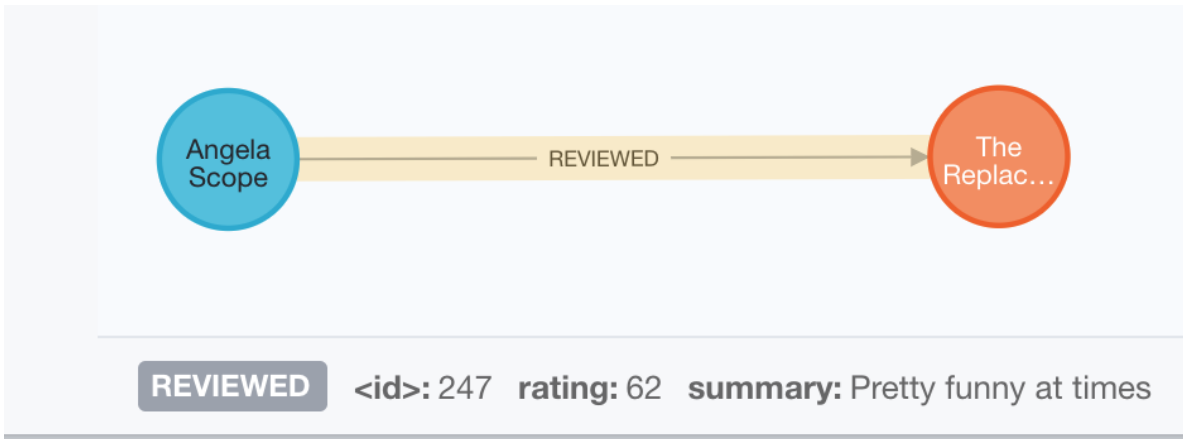
The *ACTED_IN* relationship may have the *roles* property that represents the roles that an actor had when s/he acted in a specific movie.

For example, in the "starter" Movie database, the actor, Hugo Weaving, has these properties defined for each of his *ACTED_IN* relationships to these movies:



	m.title	r.roles
1	"Cloud Atlas"	["Bill Smoke", "Haskell Moore", "Tadeusz Kesselring", "Nurse Noakes", "Boardman Mephi", "Old Georgie"]
2	"V for Vendetta"	["V"]
3	"The Matrix Revolutions"	["Agent Smith"]
4	"The Matrix Reloaded"	["Agent Smith"]
5	"The Matrix"	["Agent Smith"]

For movie reviewers, the *REVIEWED* relationship has the *rating* and *summary* properties:



REVIEWED <id>: 247 rating: 62 summary: Pretty funny at times