Getting Started Guide

Table of Contents

Neo4j v4.4

Welcome to Neo4j

```
<div class="responsive-embed">
<iframe width="680" height="425" src="https://www.youtube.com/embed/urO5FyP9PoI" title="YouTube video
player" frameborder="0" allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope;
picture-in-picture" allowfullscreen></iframe>
</div>
```

With a native graph database at the core, Neo4j stores and manages data in its more natural, connected state. The graph database takes a property graph approach, which is beneficial for both traversal performance and operations runtime.

Neo4j is dedicated to kind and open communication and aims to create developer-friendly environment for everybody.
Neo4j is open-source and has the largest and most vibrant community of graph database enthusiasts. You can reach out to our Community Forum or Discord Chat for any help or advice you may need.

To help you to get more information about Neo4j products and projects, we would like to provide you with the Getting started resources, which may inspire you to explore different possibilities of graph technologies.

If you want learn how to build, optimize, and launch your Neo4j project, visit our GraphAcademy page.
GraphAcademy courses are free, interactive, and hands-on.
All of them have been developed by Neo4j professionals with years of experience. Our set of courses aims a wide range of job role: data scientists, developers, and database administrators.

Neo4j Graph Data Platform

Starting as a graph database Neo4j has evolved into a rich ecosystem with many tools, applications, and libraries, which give you opportunity to integrate graph technologies with your working environment.

Let’s take a look at the three pillars of the Neo4j ecosystem:

- First, **Neo4j** is the world’s leading **graph database**. The architecture is designed for optimal management, storage, and traversal of *nodes* and *relationships*.
- Second, **Neo4j Aura** is the fully managed graph data platform service in the cloud. Aura empowers developers and data scientists to quickly build scalable, AI-driven applications and analyze big data with algorithms without the hassle of managing infrastructure. Neo4j Aura includes AuraDB for applications and AuraDS for data science.
- Third, **Neo4j Graph Data Science** is a connected data analytics and machine learning platform that helps you understand the connections in big data.

Cypher

Cypher is Neo4j’s graph query language that allows users to store and retrieve data from the graph database. It is a declarative, SQL-inspired language for describing patterns in graphs. The syntax provides a visual and logical way to match patterns of nodes and relationships in the graph. Cypher has been designed to be easy to learn, understand, and use for everyone, but also incorporate the power and functionality of other standard data access languages.

Contents of this guide

The Neo4j Getting Started Guide provides you with links to the relevant information resources and covers the following areas:

- Get started with Neo4j — Introduction to the Neo4j Graph Data Platform components.
- Introduction to Cypher — Overview of the graph query language Cypher.
- Data modeling — Basic principles of graph data modeling, guidelines on how to convert an RDBMS schema to graph, and introduction to data modeling tools.
- Data importing — Articles on how to import data into a Neo4j instance, representing different methods and tools.
- Neo4j Drivers — Overview of the officially supported Neo4j drivers and Neo4j Community drivers.
- Neo4j Connectors — Overview of the Neo4j connectors which help you to integrate with familiar technologies and minimize pain associated with a rip-and-replace approach to solutions.
- Visualization tools — Overview of different visualization tools provided by Neo4j and its Community.

This guide is written for anyone who is exploring Neo4j ecosystem.

Get started with Neo4j

There are a number of options on how to get started with Neo4j, and your choice depends upon your needs and aims. Each option allows you to explore different possibilities of the Neo4j ecosystem.

First steps with Neo4j

1. If you want to learn Neo4j graph database features, how to write Cypher queries, use developer tools like Neo4j Browser and Bloom, and get an overview on how to connect to Neo4j via drivers and connectors, go to the AuraDB page, where you can create a new Neo4j instance free forever.
   AuraDB is the best choice for cloud developers who are interested in graph technologies.
   Besides the free option, you can select the subscription plan which suits you best.
   At the moment, you can choose one of the example datasets on Aura to learn more about Neo4j and Cypher queries:
   - Movies
   - Graph based Recommendations
   - Graphs for Cybersecurity
   - StackOverflow Data

     If you are a data scientist, you might be interested in AuraDS — graph data science platform as a fully managed cloud service. You get access to more than 65 pretuned graph algorithms.

2. If you want to learn more about Neo4j graph database and Cypher queries applied to a specific use case, visit Neo4j Sandbox, where you find a number of example datasets based on different use cases.

   All databases are grouped into two categories:

   - for developers
   - for data scientists

     You can also select a blank sandbox and populate it with your own data.

3. One of the ways to set up an environment for developing an application with Neo4j and Cypher is to use Neo4j Desktop. Download Neo4j Desktop from https://neo4j.com/download/ and follow the installation instructions for your operating system.
   You get the opportunity to try all the graph applications, which are available in our **Graph Apps Gallery**.
4. If you need an Enterprise Edition, visit the Neo4j Pricing page where you can get all necessary instructions and contacts.

For all options on how to get started with Neo4j see https://neo4j.com/try-neo4j/.

Neo4j documentation page

All the official documentation is available at https://neo4j.com/docs/.

Documentation homepage is designed to make your journey across all Neo4j products comfortable and pleasant.

That is where you find reference documentation to all components of the Neo4j Graph Data Platform. Guides are grouped by categories. In the leftside navigation bar you see:

- Neo4j DBMS — Documentation for the Neo4j graph database, a core product:
  - Getting Started Guide — you are here. The starting point to enter the Neo4j world.
  - The Neo4j Operations Manual — the docset tells you how to deploy, configure, support, and maintain Neo4j. You find comprehensive descriptions of Neo4j Causal Clustering here.
  - Neo4j Upgrade and Migration Guide — the guide describes how to keep your Neo4j deployment up-to-date.
  - Status Code — overview of Neo4j status codes classifications, version changes, and also a complete list of all status codes Neo4j may return, and what they mean.
  - Java Reference tells you about advanced Java-centric usage of Neo4j, how to embed Neo4j in your own software and write extensions.
  - The Neo4j Kerberos Add-on — Neo4j supports the use of Kerberos — a network authentication protocol, here you find description of using the *Neo4j Kerberos Add-on*.
- Neo4j Aura — Docsets for a fully automated graph platform offered as a cloud service:
  - Neo4j AuraDB — graph database as a fully managed service.
  - Neo4j AuraDS — Neo4j Graph Data Science platform as a fully managed service.
- Neo4j Tools — Docsets for developer tools designed to make it easier to develop graph applications and to interact with Neo4j database:
  - Neo4j Bloom — the guide for a graph visualization tool designed for interactive exploration of graph data.
  - Neo4j Browser — the guide for a developer-focused tool that allows to run Cypher queries and visualize the results.
  - Neo4j Desktop — the guide introduces a client application for working with Neo4j, whether using local database instances or databases located on remote servers.
  - Neo4j Ops Manager — the docset for a UI-based tool that enables any administrator to monitor, administer, and operate all of the Neo4j database management systems in an Enterprise Edition.
- Neo4j Graph Data Science Library provides extensive analytical capabilities centered around graph algorithms:
  - The Neo4j Graph Data Science Library Manual tells you how graph algorithms are implemented into Neo4j and how to use them as they are exposed as Cypher procedures.
- Cypher Query Language — Docsets for Neo4j graph query language:
  - The Neo4j Cypher Manual — the comprehensive manual for Cypher for the developers of a Neo4j client application.
  - Cypher Cheat Sheet — a printable set of notes for a quick overview of Cypher syntax basics.
  - Neo4j Cypher Refcard — a quick reference guide for Cypher language.

- Neo4j Drivers and APIs — Docsets for officially supported drivers, connectors, and APIs.
  - The Neo4j Go Driver Manual
  - The Neo4j Java Driver Manual
  - The Neo4j JavaScript Driver Manual
  - The Neo4j .NET Driver Manual
  - The Neo4j Python Driver Manual
  - Neo4j GraphQL Library
  - HTTP API
  - Neo4j-OGM — An Object Graph Mapping Library for Neo4j
  - Spring Data Neo4j
  - Neo4j Connector for Apache Spark
  - Neo4j Connector for BI (JDBC)
- Resources

If the officical documentation is not enough, you can refer to the aforementioned **Resources**:

- Documentation Archive — docsets for earlier releases are available here.
- PDF library — if you need, you can download PDF-versions of our documentation. Follow the link for a full list of all available PDFs.

Labs menu at the top contains the list of all Neo4j Labs projects, which are designed and developed as a way to extend and test functionality of the Neo4j Graph Data Platform. The Labs projects are supported via the Neo4j Community, and we cannot provide any commercial support for them or guarantee backwards compatibility.

At the top on the right side, you find the drop-down menu *Get Help* with the following choices:

- Community Forum — for learning and guidance.
- Discord Chat — a live chat environment for communicating with other Neo4j users (requires signup).
- Knowledge Base — troubleshooting articles written by developers for developers on how to solve issues both for Community ans Enterprise Editions.
- Neo4j Developer Blog — Neo4j channel on Medium platform for deep dives into technical topics and announcements of new products, releases.
- Neo4j Videos — a link to the Neo4j channel on YouTube.

The blue button *Get Started* in the upper right hand corner allows you to access:

- Neo4j AuraDB page, where you can choose AuraDB plan that best suits your needs.
- Neo4j Sandbox.
- Neo4j Desktop download page.

If you click the button Get Started, you are redirected to the AuraDB page.

What is a graph database?

A graph database stores nodes and relationships instead of tables, or documents. Data is stored just like you might sketch ideas on a whiteboard. Your data is stored without restricting it to a pre-defined model, allowing a very flexible way of thinking about and using it.

See Graph database concepts if you want to learn more about graph theory basics.

Why graph databases?

We live in a connected world, and understanding most domains requires processing rich sets of connections to understand what's really happening. Often, we find that the connections between items are as important as the items themselves.

How else do people do this today? While existing relational databases can store these relationships, they navigate them with expensive `JOIN` operations or cross-lookups, often tied to a rigid schema. It turns out that "relational" databases handle relationships poorly. In a graph database, there are no JOINs or lookups. Relationships are stored natively alongside the data elements (the nodes) in a much more flexible format. Everything about the system is optimized for traversing through data quickly; millions of connections per second, per core.

Graph databases address big challenges many of us tackle daily. Modern data problems often involve many-to-many relationships with heterogeneous data that sets up needs to:

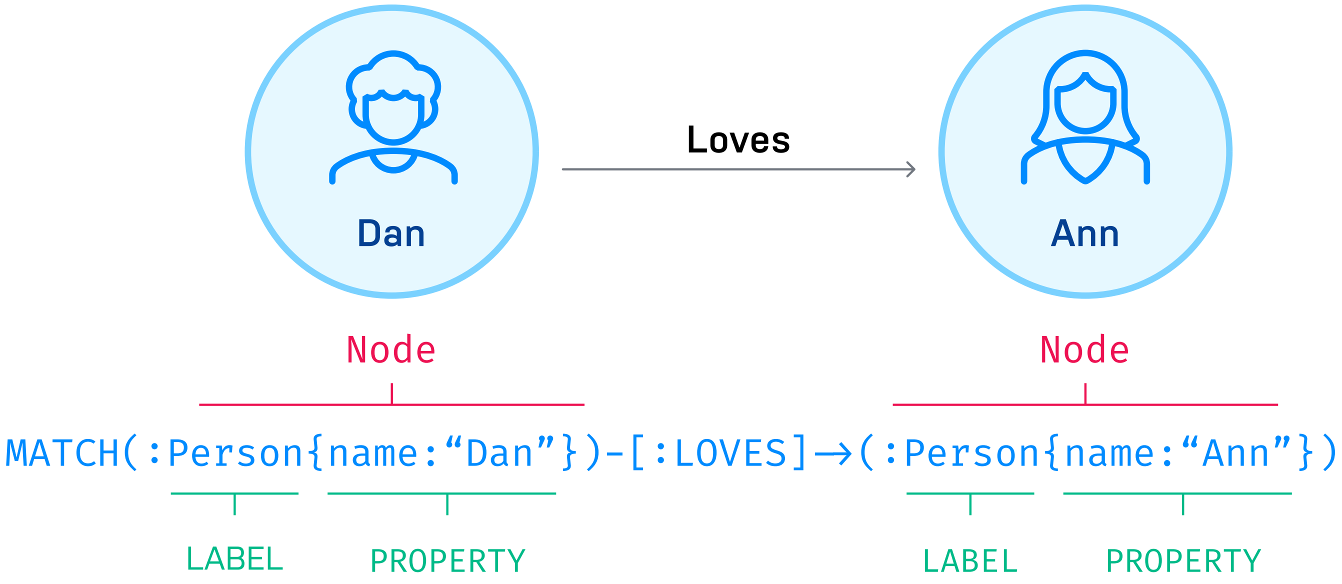
- Navigate deep hierarchies,
- Find hidden connections between distant items, and
- Discover inter-relationships between items.

Whether it's a social network, payment networks, or road network you'll find that everything is an interconnected graph of relationships. And when we want to ask questions about the real world, many questions are *about the relationships* rather than about the individual data elements.

The property graph model

In Neo4j, information is organized as nodes, relationships, and properties.

Building blocks of the property graph model



```
MATCH(:Person{name:“Dan”})-[:LOVES]→(:Person{name:“Ann”})
```

Nodes are the entities in the graph.

- Nodes can be tagged with *labels*, representing their different roles in your domain. (For example, `Person`).
- Nodes can hold any number of key-value pairs, or *properties*. (For example, `name`)
- Node labels may also attach metadata (such as index or constraint information) to certain nodes.

Relationships provide directed, named, connections between two node entities (for example, Person `LOVES` Person).

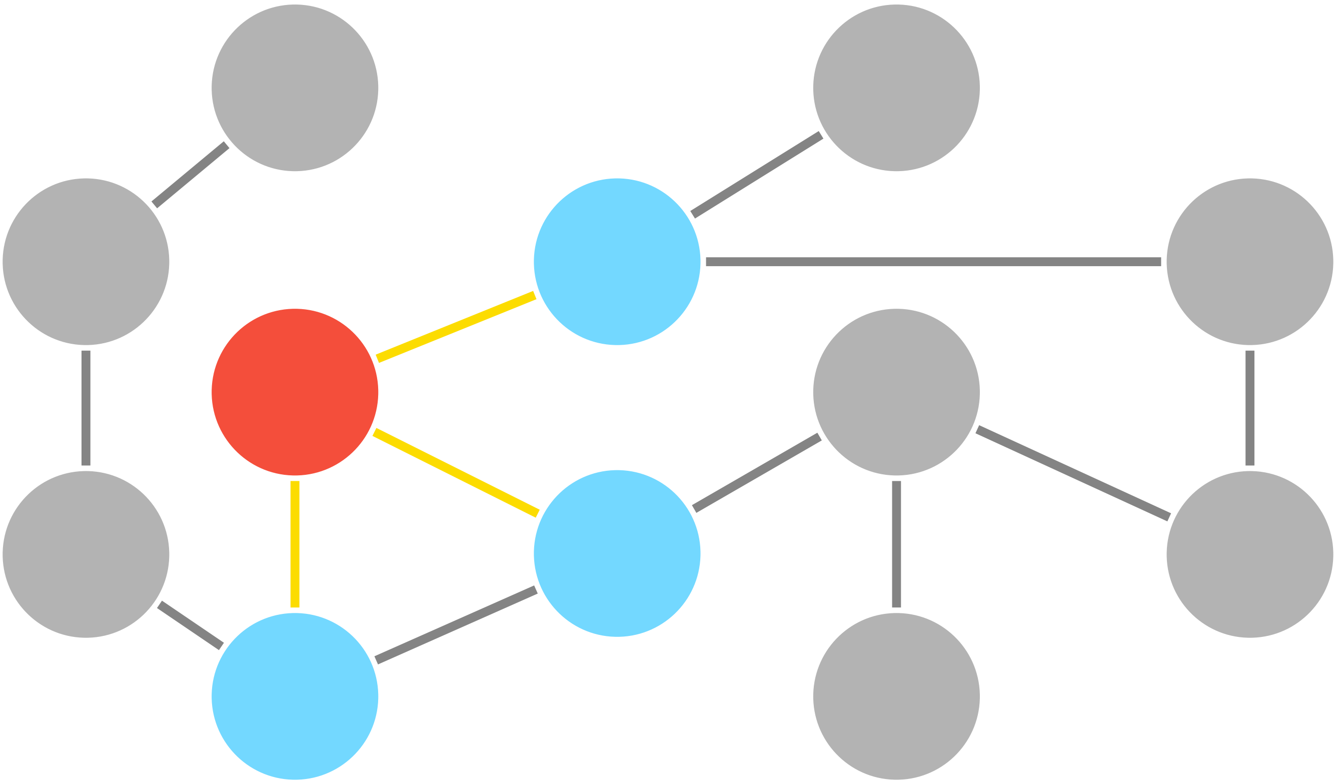
- Relationships always have a direction, a type, a start node, and an end node, and they can have properties, just like nodes.
- Nodes can have any number or type of relationships without sacrificing performance.
- Although relationships are always *directed*, they can be navigated efficiently in any direction.

If you'd like to learn more about any of these, you can read more about Graph Data Modeling.

What is Neo4j?

Neo4j is an open-source, NoSQL, native graph database that provides an ACID-compliant transactional backend for your applications that has been publicly available since 2007.

Neo4j is offered as a managed service via AuraDB. But you can also run Neo4j yourself with either Community Edition or Enterprise Edition. The Enterprise Edition includes all that Community Edition has to offer, plus extra enterprise requirements such as backups, clustering, and failover abilities. Neo4j is written in Java and Scala, and the source code is available on GitHub.



Neo4j is a *native graph database*, which means that it implements a true graph model all the way down to the storage level. The data isn't stored as a "graph abstraction" on top of another technology, it's stored just as you whiteboard it. This is important because it's the reason why Neo4j outperforms other graphs and stays so flexible. Beyond the core graph, Neo4j provides what you'd expect out of a database; ACID transactions, cluster support, and runtime failover. This stability and maturity is why it's been used in production scenarios for large enterprise workloads for years.

What makes Neo4j the easiest graph to work with?

- **Cypher**, a declarative query language similar to SQL, but optimized for graphs. Now used by other databases like SAP HANA Graph and Redis graph via the openCypher project.
- **Constant time traversals** in big graphs for both depth and breadth due to efficient representation of nodes and relationships. Enables scale-up to billions of nodes on moderate hardware.
- **Flexible** property graph schema that can adapt over time, making it possible to materialize and add new relationships later to shortcut and speed up the domain data when the business needs change.
- Drivers for popular programming languages, including Java, JavaScript, .NET, Python, and many more.

Where and how is Neo4j used?

Neo4j is used today by thousands of startups, educational institutions, and large enterprises in all sectors including financial services, government, energy, technology, retail, and manufacturing. From innovative new technology to driving businesses, users are generating insights with graph, generating new revenue, and improving their overall efficiency.

You can find more information about numerous use cases when connected data matters most on our website.

Real-Time Transaction Applications

Generate and protect revenue

Recommendations

Fraud detection

Dynamic pricing

IoT applications

Metadata and Advanced Analytics

Generate actionable insights

Customer engagement

Data lake integration

Risk mitigation

Knowledge graphs for AI

Internal Business Processes

Improve efficiency and cut costs

Network management

Supply chain efficiency

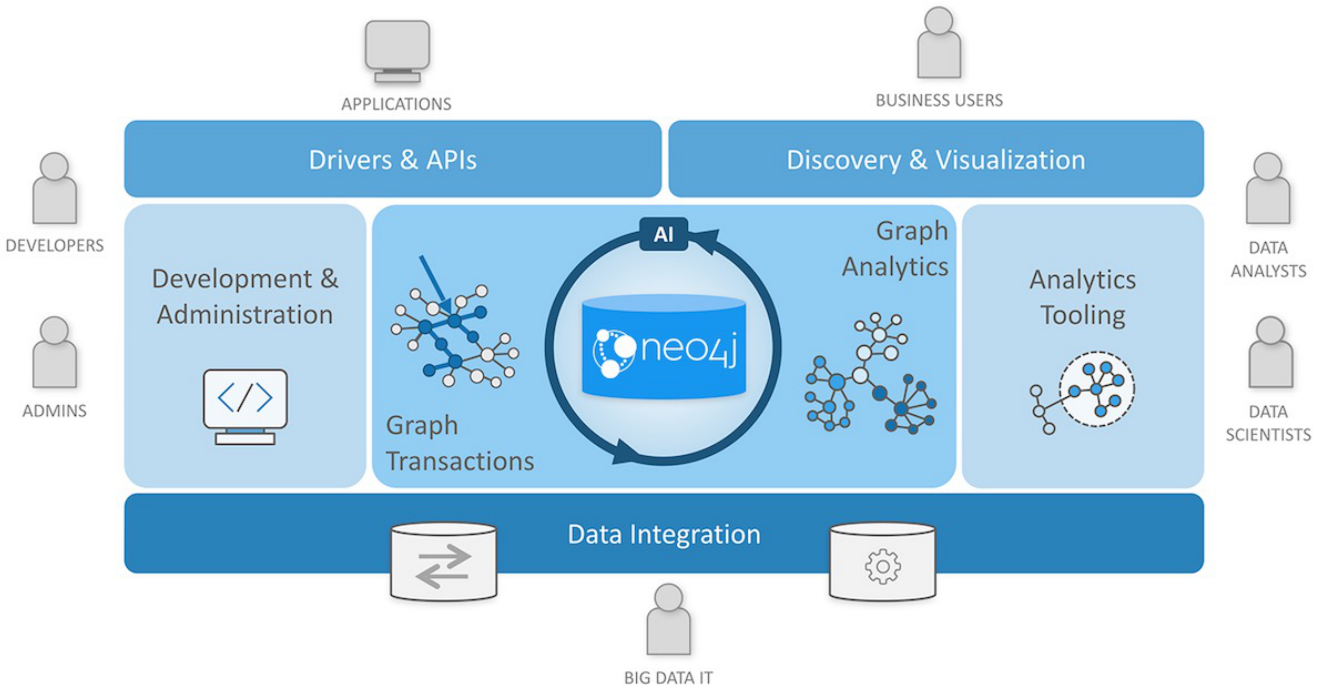
Identity and access management

Neo4j Graph Data Platform

There are a variety of ways to interact with and use graph data in Neo4j. This section introduces the different products we provide at Neo4j to suit your diverse business and technology needs and shows you how to get started using them.

From our core product (the graph database) to visualization for business users, we provide a variety of opportunities to maximize your business and simplify your data.

Components of the Neo4j Graph Data Platform



Each of the elements listed below is designed to fill a business or technical need. We continue to improve the different perspectives from which to work with data, as well as capabilities of the products themselves.

More information about each of the components can be found on the Product page.
Reference documentation is at https://neo4j.com/docs/.
If you have any questions or issues, don’t hesitate to reach out to the Online Community!

Below you also find links to the homepages of the Neo4j products.

Neo4j Graph Database

Neo4j Graph Database — our core product, native graph database that is built to store and retrieve connected data.
Two editions — a Community Edition and an Enterprise Edition — are available.
There are many ways to deploy Neo4j today: on-premises server installation, self-hosted in the cloud with pre-built images, or by simply using **Neo4j Aura** — the zero-admin, always-on graph database for cloud developers.
All necessary documentation on how to install and maintain Neo4j can be found on the docs page.

Neo4j Aura

Neo4j is a cloud-friendly database, with a variety of cloud deployment options readily available. You are able to run Neo4j on public clouds like AWS, Azure and Google Cloud Platform (GCP).
Neo4j Aura is a fully managed, cloud-native graph service, giving developers and data scientists the most advanced graph tech tools. Neo4j Aura includes **AuraDB** and **AuraDS**.

Neo4j AuraDB is the fully automated, always-on, and scalable graph database as a service for developers building modern applications.
Find out more information in the official documentation.

Neo4j AuraDS makes it easy to run graph algorithms on Neo4j by integrating two main components:

- **Neo4j Database**, where graph data are loaded and stored, and Cypher queries and all database operations (for example user management, query termination, etc.) are executed;
- **Graph Data Science**, a software component installed in the Neo4j Database, whose main purpose is to run graph algorithms on in-memory projections of Neo4j Database data.

Neo4j AuraDS is available on GCP. See pricing page for details. You can find more details in the official documentation for AuraDS.

Neo4j Graph Data Science

Neo4j Graph Data Science (GDS) gives you access to more than 65 graph algorithms, which can be executed with Neo4j and optimized for enterprise workloads and pipelines. GDS helps you to get insights from big data in order to answer critical questions and improve predictions. GDS is the only connected data analysis platform that unifies the ML surface and graph database into a single workspace. This way, data scientists run algorithms and ML models without jumping between tools for ETL.

See documentation for the library for more details.

Neo4j Tools

Neo4j provides a variety of tools that aim to make learning and development of graph applications easier.

The Neo4j Developer Tools page introduces the most important of them.

- Neo4j Desktop — a local development environment for working with Neo4j, whether using local database instances or databases located on remote servers. Free download includes Neo4j Enterprise Edition license.
- Neo4j Browser — online browser interface to query and view the data in the database. Basic visualization capabilities using Cypher query language.
- Neo4j Operations Manager (NOM) — a UI-based tool that enables a database administrator to monitor, administer, and operate all of the Neo4j database management systems in an Enterprise Edition. Video series: NOM Bytes introduces the product and provides some practical tips. Documentation for Neo4j Ops Manager gives all information you may need for using it.
- Data Importer — a no-code tool that allows you to load data from flat files (`.csv` and `.tsv`) into your Neo4j databases, define a graph model, and map your data to it.
- Neo4j Bloom — visualization tool for business users that does not require any code or programming skills to view and analyze data. Documentation is also available on our Docs page.

Cypher Query Language

Cypher is an open data query language, based on the openCypher initiative. It is the most established and intuitive query language to learn with property graphs. Cypher can be characterized by following:

- easy to learn
- visual and logical
- secure, reliable, and data-rich
- open and flexible

Connecting to Neo4j

It is important for us to provide a wide range of opportunities for integrating Neo4j with any working environment.

- Neo4j Drivers — Officially supported drivers and Community drivers.
- Neo4j Connectors — A set of connectors to integrate your regular working environment with Neo4j.
- GraphQL Library is a flexible, low-code, open source JavaScript library that enables rapid API development for cross-platform and mobile applications by tapping into the power of connected data.
- OGM — An Object Graph Mapping Library for Neo4j.

Neo4j Labs projects

Neo4j also has a variety of extension libraries and developer tools that can be added to existing products to enhance functionality. Some of these projects have been adopted by our Neo4j Labs team to help developers integrate with other innovative tools and libraries. You can find tutorials and other specific information for some of these on this page.

- Arrows — a web-based data modeling tool.
- APOC — the standard utility library of procedures and functions for Neo4j.

- ETL Tool — migrate data from a relational database to Neo4j using an application and simple user interface.
- NeoDash — an open-source, low-code dashboard builder for Neo4j. It lets you build an interactive dashboard with tables, graphs, bar charts, line charts, maps, and more.

Introduction to Cypher

This section introduces you to the graph query language — Cypher. It aims to help you start thinking about graphs and patterns, apply this knowledge to simple problems, and learn how to write Cypher statements.

For the full reference of Cypher, see the Cypher documentation.
You can visit Cypher page to get an overview of Neo4j graph query language.

Cypher query language

Cypher is Neo4j's graph query language that lets you retrieve data from the graph. It is like SQL for graphs, and was inspired by SQL so it lets you focus on what data you want out of the graph (not how to go get it). It is the easiest graph language to learn by far because of its similarity to other languages and intuitiveness.

You
:ARE_LEARNING
Cypher

Cypher is unique because it provides a visual way of matching patterns and relationships. Cypher was inspired by an ASCII-art type of syntax where `(nodes)-[:ARE_CONNECTED_TO]->(otherNodes)` using rounded brackets for circular `(nodes)`, and `-[:ARROWS]->` for relationships. When you write a query, you draw a graph pattern through your data.

Neo4j users use Cypher to construct expressive and efficient queries to do any kind of create, read, update, or delete (CRUD) on their graph, and Cypher is the primary interface for Neo4j.

Like Neo4j itself, Cypher is open source! The openCypher project provides all of the specs needed. Cypher is backed by a number of companies all of which benefit from cypher.

Once you have an AuraDB database, you can use the `:guide cypher` command inside of Neo4j Browser to get started.

Getting Started Guide covers the basics of the language, which you can explore topic by topic, starting with basic material and building up towards more complex material.

Getting started with Cypher

Why Cypher?

We already know that Neo4j's property graph model is composed of nodes and relationships, which may also have properties associated with them. However, nodes and relationships are the simple components that build the most valuable and powerful piece of the property graph model — the pattern. Patterns are comprised of node and relationship elements and can express simple or complex traversals and paths.

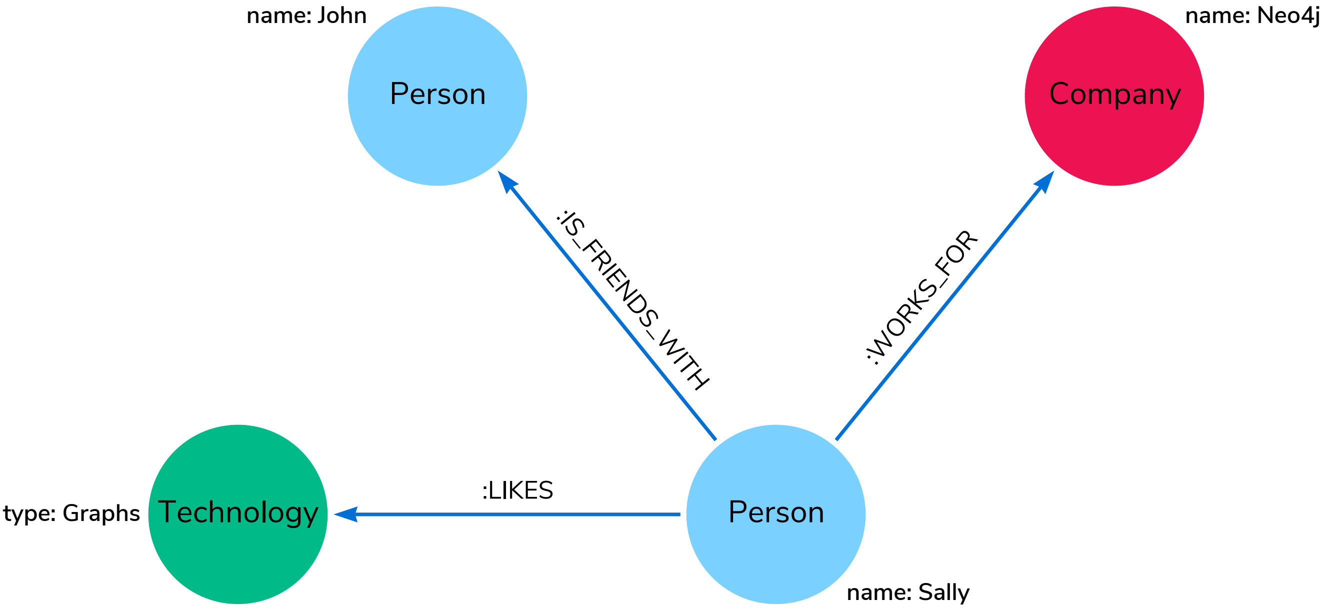
Pattern recognition is fundamental to the way that the brain works. Because of this, humans are very good at working with patterns (think of visual diagrams or even memory-matching games). Cypher is also heavily based on patterns and is designed to recognize various versions of these patterns in data, making it a simple and logical language for users to learn.

Cypher syntax

The video below walks through some background on Cypher, basic syntax, and some intermediate examples. The concepts in the video are discussed in the following paragraphs, as well as in upcoming chapters.

```
<div class="responsive-embed">
<iframe width="560" height="315" src="https://www.youtube.com/embed/_dup3YOZSm8" title="What is Cypher?"
frameborder="0" allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope; picture-in-
picture" allowfullscreen></iframe>
</div>
```

Cypher's constructs, based on English prose and neat iconography, make queries easy both to write and to read. For example, take a look at the simple graph data in the image below. How would you represent this data in English?
NOTE: the answer is in the paragraph below.



Sally likes Graphs. Sally is friends with John. Sally works for Neo4j.

Cypher syntax builds upon this English-language structure we just created. Further you can find information on how to write this example in Cypher.

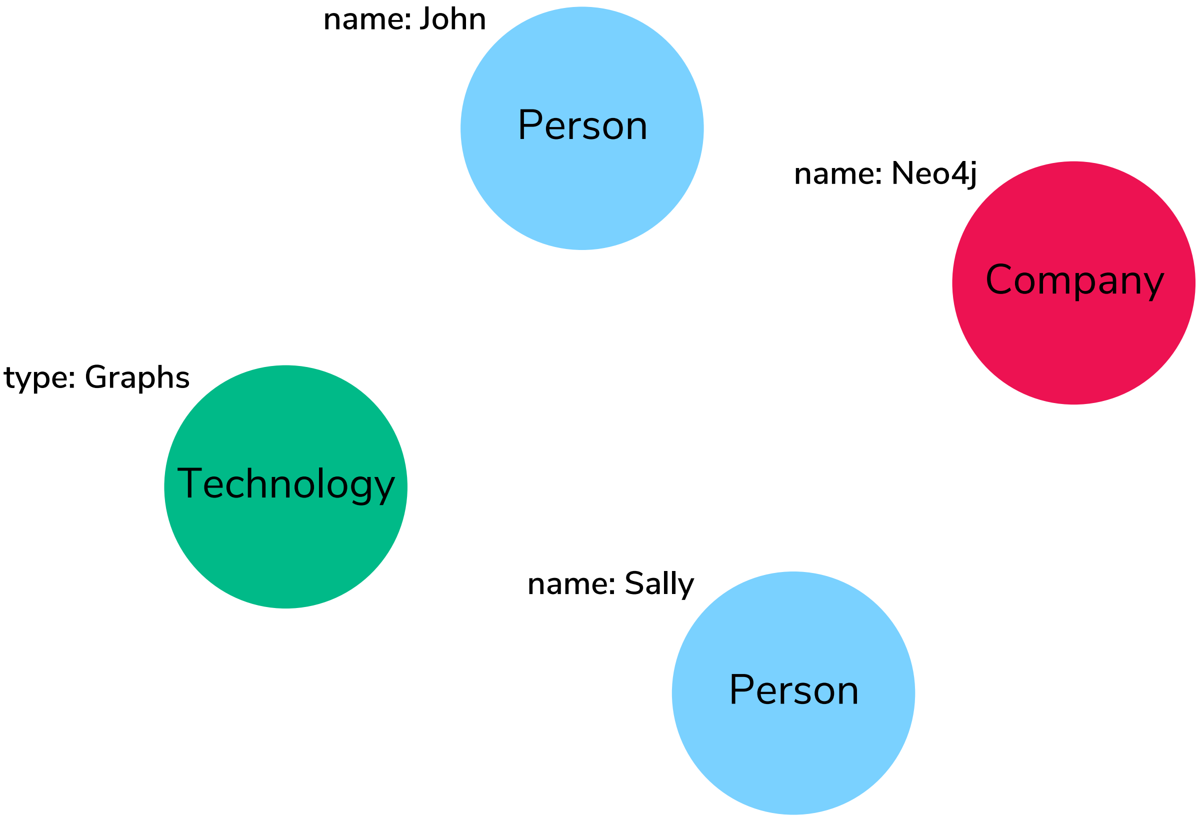
Cypher comments

As you work through this section, you see comments in the Cypher code to help explain the syntax or what a query is doing. Comments in Cypher are the same as in many programming languages. You can add comments by starting a line with `//` and putting text after the slashes. Just like in other languages, starting the line with two forward slashes means that anything on that line will become a comment.

> This is especially helpful to use in Neo4j Browser when saving queries. If you add a comment before the query, the comment automatically becomes the title of the saved query!

Representing nodes in Cypher

Since Cypher is inspired by ASCII-Art for patterns, we need a visual way to represent each component of our pattern above. We know that the main components of the property graph model are nodes and relationships. Remember that nodes are the data entities in your graph and that you can often identify nodes by finding the nouns or objects in your data model. In our example, `Sally`, `John`, `Graphs`, and `Neo4j` are our nodes.



To depict nodes in Cypher, we surround the node with parentheses, e.g. `(node)`. Notice how the parentheses look similar to the circles that the visual representation uses for nodes in our data model.

Node variables

If we later want to refer to the node, we can give it a variable like `(p)` for person or `(t)` for thing. In real-world queries, we might use longer, more expressive variable names like `(person)` or `(thing)`. Just like in programming language variables, you can name your variables what you want and reference them by that same name later in a query.

If the node is not relevant to your return results, you can specify an anonymous node using empty parentheses `()`. This means that you are not be able to return this node later in the query.

Node labels

If you remember from the property graph data model, you can also group similar nodes together by assigning a node label. Labels are kind of like tags and allow you to specify certain types of entities to look for or create. In our example, `Person`, `Technology`, and `Company` are the labels.

You can kind of think of this like telling SQL which table to look for the particular row. Just like to tell SQL to query a person's information from a `Person` or `Employee` or `Customer` table, you can also tell Cypher to only check those labels for that information. This helps Cypher distinguish between entities and optimize execution for your queries. It is always better to use node labels in your queries, where possible.

> If you do not specify a label for Cypher to filter out non-matching node categories, the query checks all of the nodes in the database! As you can imagine, this would be cumbersome if you had a very large graph.

Example: nodes in Cypher

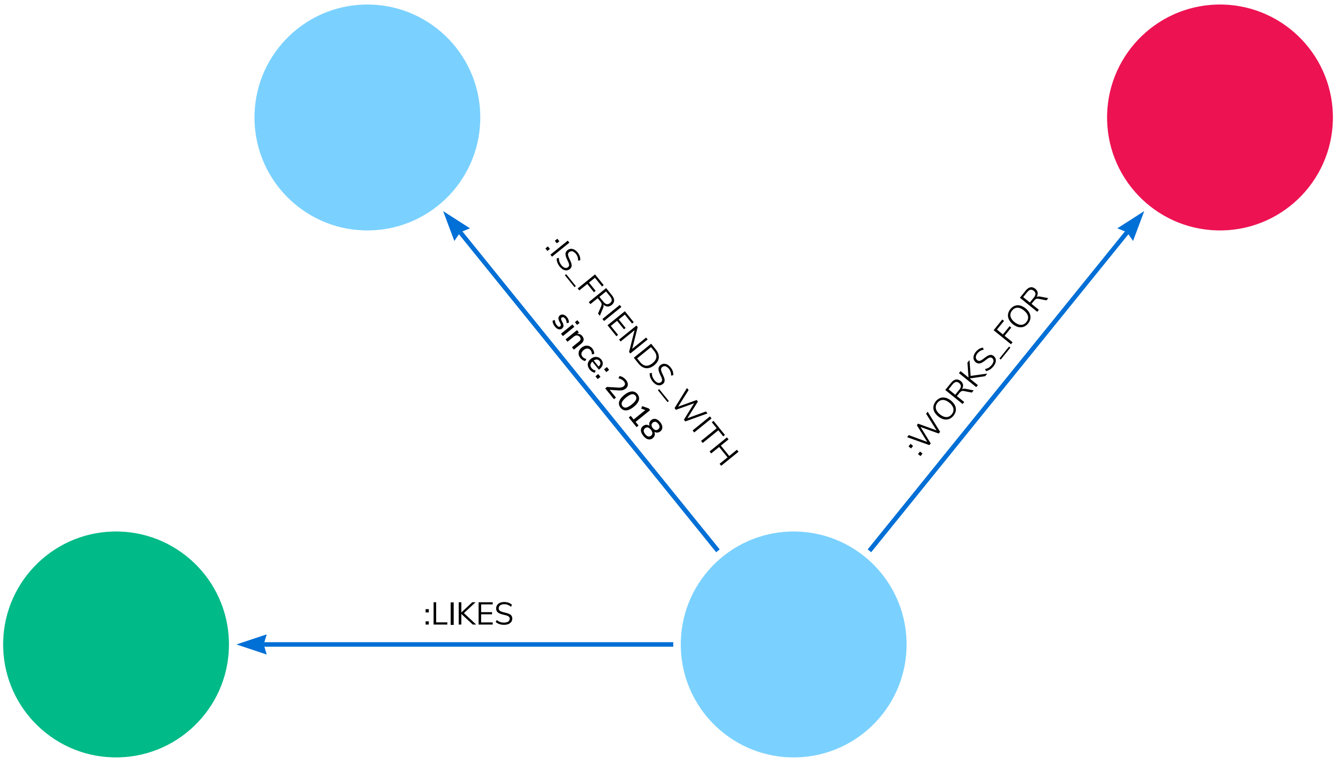
Using our graph example above, let's see how we could specify our nodes.

```
()                  //anonymous node (no label or variable) can refer to any node in the database
(p:Person)          //using variable p and label Person
(:Technology)       //no variable, label Technology
(work:Company)      //using variable work and label Company
```

Representing relationships in Cypher

To fully utilize the power of a graph database, we also need to express the relationships between our nodes. Relationships are represented in Cypher using an arrow `-->` or `<--` between two nodes. Notice how the syntax looks like the arrows and lines connecting our nodes in the visual representation. Additional information, such as how nodes are connected (relationship type) and any properties pertaining to the relationship, can be placed in square brackets inside of the arrow.

In our example, the lines with `LIKES`, `IS_FRIENDS_WITH`, and `WORKS_FOR` between nodes are our relationships.



Undirected relationships are represented with no arrow and just two dashes `--`. This means that the relationship can be traversed in either direction. While a direction **must** be inserted to the database, it can be matched with an undirected relationship where Cypher ignores any particular direction and retrieves the relationship and connected nodes, no matter what the physical direction is. This allows the queries to be flexible and not force the user to know the physical direction of the relationship stored in the database.

If data is stored with one relationship direction, and a query specifies the wrong direction, Cypher will not return any results. In these cases where you may not be sure of direction, it is better to use an undirected relationship and retrieve some results.

```
//data stored with this direction
CREATE (p:Person)-[:LIKES]->(t:Technology)

//query relationship backwards will not return results
MATCH (p:Person)<-[:LIKES]-(t:Technology)

//better to query with undirected relationship unless sure of direction
MATCH (p:Person)-[:LIKES]-(t:Technology)
```

Relationship types

Relationship types categorize and add meaning to a relationship, similar to how labels group nodes. In our property graph data model, relationships show how nodes are connected and related to each other. You can usually identify relationships in your data model by looking for actions or verbs.

You can specify any type of relationship you want between nodes, but we recommend good naming conventions using verbs and actions. Poor relationship type names make it more difficult to both read and write Cypher (remember, it should sound like English!).

For example, let us look at the relationship types from our example graph.

- `[:LIKES]` - makes sense when we put nodes on either side of the relationship (Sally LIKES Graphs)

- `[:IS_FRIENDS_WITH]` - makes sense when we put nodes with it (Sally IS_FRIENDS_WITH John)
- `[:WORKS_FOR]` - makes sense with nodes (Sally WORKS_FOR Neo4j)

Relationship variables

Just as we did with nodes, if we want to refer to a relationship later in a query, we can give it a variable like `[r]` or `[rel]`. We can also use longer, more expressive variable names like `[likes]` or `[knows]`. If you do not need to reference the relationship later, you can specify an anonymous relationship using two dashes `--`, `-->`, `<--`.

As an example, you could use either `-[rel]->` or `-[rel:LIKES]->` and call the `rel` variable later in your query to reference the relationship and its details.

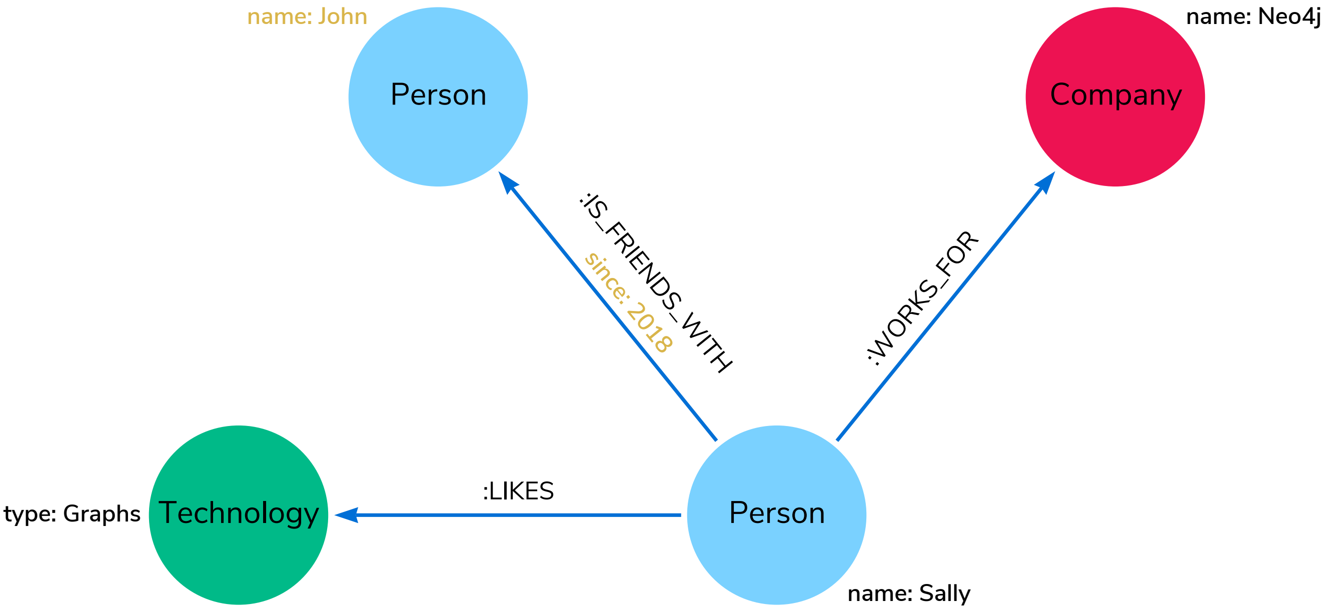
> ℹ If you forget the colon in front of a relationship type like this `-[LIKES]->`, it represents a variable (not a relationship type). Since no relationship type declared, Cypher searches all types of relationships.

Node or relationship properties

We have talked about how to write Cypher for nodes, relationships, and labels. The last piece of our property graph data model is for properties. Remember that properties are name-value pairs that provide additional details to our nodes and relationships.

To represent these in Cypher, we can use curly braces within the parentheses of a node or the brackets of a relationship. The name and value of the property then go inside the curly braces. Our example graph has both a node property (`name`) and a relationship property (`since`).

- Node property: `(p:Person {name: 'Sally'})`
- Relationship property: `-[rel:IS_FRIENDS_WITH {since: 2018}]->`



Properties can have values with a variety of data types. To see the full list that Cypher offers, see the manual section on values and types.

Patterns in Cypher

Nodes and relationships make up the building blocks for graph patterns. These building blocks can come together to express simple or complex patterns. Patterns are the most powerful capability of graphs. In Cypher, they can be written as a continuous path or separated into smaller patterns and tied together with commas.

To show a pattern in Cypher, you need to combine the node and relationship syntaxes you have learned so far. Let's use the example of `Sally likes Graphs`.

In Cypher, this pattern would look like the code below.

```
(p:Person {name: "Sally"})-[rel:LIKES]->(g:Technology {type: "Graphs"})
```

This bit of Cypher tells the pattern we want, but it does not tell whether we want to find that existing pattern or insert it as a new pattern. To tell Cypher what we want it to do with the pattern, we need to add some keywords.

Are you struggling?

If you need help with any of the information contained on this page, you can reach out to other members of our community. You can ask questions on the Neo4j Community Site.

Tutorial: Build a Recommendation Engine

With Cypher structure and syntax covered in the sections above, you can dive into building your own recommendation engine to use graph data and Cypher to recommend movies, colleagues, cuisines, and more.

Tutorial: Build a Recommendation Engine,role=more information walks through using queries and filtering that takes advantage of the relationships in a graph in order to lend insight into habits and hidden connections and provide valuable recommendations.

Cypher resources

Find out where else you can learn Cypher or increase your depth of knowledge from experts and solutions. There are a variety of training opportunities, blogs, videos, and more for taking the next steps in your Cypher (and Neo4j) journey!

Learn with GraphAcademy

Cypher Fundamentals

This course teaches you the essentials of using Cypher, Neo4j's powerful query language, in as little time as possible, with videos, quizzes and hands-on exercises.

Learn Cypher with GraphAcademy

Patterns

Neo4j’s Property Graphs are composed of nodes and relationships, either of which may have properties. Nodes represent entities, for example concepts, events, places and things. Relationships connect pairs of nodes.

However, nodes and relationships can be considered as low-level building blocks. The real strength of the property graph lies in its ability to encode *patterns* of connected nodes and relationships. A single node or relationship typically encodes very little information, but a pattern of nodes and relationships can encode arbitrarily complex ideas.

Cypher, Neo4j’s query language, is strongly based on patterns. Specifically, patterns are used to match desired graph structures. Once a matching structure has been found or created, Neo4j can use it for further processing.

A simple pattern, which has only a single relationship, connects a pair of nodes (or, occasionally, a node to itself). For example, *a Person* `LIVES_IN` *a City* or *a City is* `PART_OF` *a Country*.

Complex patterns, using multiple relationships, can express arbitrarily complex concepts and support a variety of interesting use cases. For example, we might want to match instances where *a Person* `LIVES_IN` *a Country*. The following Cypher code combines two simple patterns into a slightly more complex pattern which performs this match:

```
(:Person) -[:LIVES_IN]-> (:City) -[:PART_OF]-> (:Country)
```

Diagrams made up of icons and arrows are commonly used to visualize graphs. Textual annotations provide labels, define properties etc.

Node syntax

Cypher uses a pair of parentheses to represent a node: `()`. This is reminiscent of a circle or a rectangle with rounded end caps. Below are some examples of nodes, providing varying types and amounts of detail:

```
()
(matrix)
(:Movie)
(matrix:Movie)
(matrix:Movie {title: 'The Matrix'})
(matrix:Movie {title: 'The Matrix', released: 1997})
```

The simplest form, `()`, represents an anonymous, uncharacterized node. If we want to refer to the node elsewhere, we can add a variable, for example: `(matrix)`. A variable is restricted to a single statement. It may have different or no meaning in another statement.

The `:Movie` pattern declares a label of the node. This allows us to restricts the pattern, keeping it from matching (say) a structure with an `Actor` node in this position.

The node’s properties, for example `title`, are represented as a list of key-value pairs, enclosed within a pair of braces, for example: `{name: 'Keanu Reeves'}`. Properties can be used to store information and/or restrict patterns.

Relationship syntax

Cypher uses a pair of dashes (`--`) to represent an undirected relationship. Directed relationships have an arrowhead at one end (`<--`, `-->`). Bracketed expressions (`[...]`) can be used to add details. This may include variables, properties, and type information:

```
-->
-[role]->
-[:ACTED_IN]->
-[role:ACTED_IN]->
-[role:ACTED_IN {roles: ['Neo']}]->
```

The syntax and semantics found within a relationship's bracket pair are very similar to those used between a node's parentheses. A variable (e.g., `role`) can be defined, to be used elsewhere in the statement. The relationship's type (e.g., `:ACTED_IN`) is analogous to the node's label. The properties (e.g., `roles`) are entirely equivalent to node properties.

Pattern syntax

Combining the syntax for nodes and relationships, we can express patterns. The following could be a simple pattern (or fact) in this domain:

```
(keanu:Person:Actor {name: 'Keanu Reeves'})-[role:ACTED_IN {roles: ['Neo']}]->(matrix:Movie {title: 'The
Matrix'})
```

Equivalent to node labels, the `:ACTED_IN` pattern declares the relationship type of the relationship. Variables (e.g., `role`) can be used elsewhere in the statement to refer to the relationship.

As with node properties, relationship properties are represented as a list of key/value pairs enclosed within a pair of braces, for example: `{roles: ['Neo']}`. In this case, we used an array property for the `roles`, allowing multiple roles to be specified. Properties can be used to store information and/or restrict patterns.

Pattern variables

To increase modularity and reduce repetition, Cypher allows patterns to be assigned to variables. This allows the matching paths to be inspected, used in other expressions, etc.

```
acted_in = (:Person)-[:ACTED_IN]->(:Movie)
```

The `acted_in` variable would contain two nodes and the connecting relationship for each path that was found or created. There are a number of functions to access details of a path, for example: `nodes(path)`, `relationships(path)`, and `length(path)`.

Clauses

Cypher statements typically have multiple *clauses*, each of which performs a specific task, for example:

- create and match patterns in the graph

- filter, project, sort, or paginate results
- compose partial statements

By combining Cypher clauses, we can compose more complex statements that express what we want to know or create.

Patterns in practice

Creating and returning data

Let’s start by looking into the clauses that allow you to create data.

To add data, you just use the patterns you already know. By providing patterns, you can specify what graph structures, labels, and properties you would like to make part of your graph.

Obviously the simplest clause is called `CREATE`. It creates the patterns that you specify.

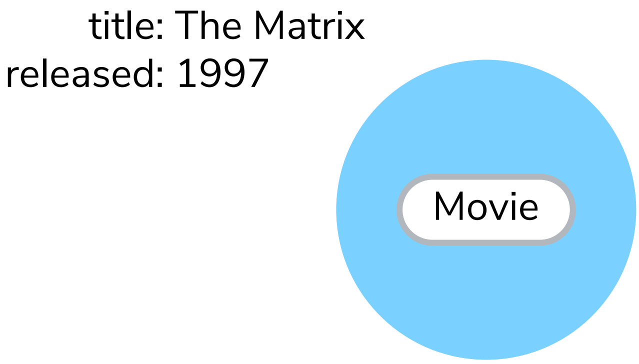
For the patterns you have looked at so far this could look like the following:

```
CREATE (:Movie {title: 'The Matrix', released: 1997})
```

If you run this statement, Cypher returns the number of changes: in this case adding one node, one label, and two properties.

```
Created Nodes: 1
Added Labels: 1
Set Properties: 2
Rows: 0
```

As you started out with an empty database, you now have a database with a single node in it:



If you also want to return the created data, you can add a `RETURN` clause, which refers to the variable you have assigned to your pattern elements.
The `RETURN` keyword in Cypher specifies what values or results you might want to return from a Cypher query.
You can tell Cypher to return nodes, relationships, node and relationship properties, or patterns in your query results.
`RETURN` is not required when doing write procedures, but is needed for reads.
The node and relationship variables, which are discussed earlier, become important when using `RETURN`.

```
CREATE (p:Person {name: 'Keanu Reeves', born: 1964})
RETURN p
```

This is what gets returned:

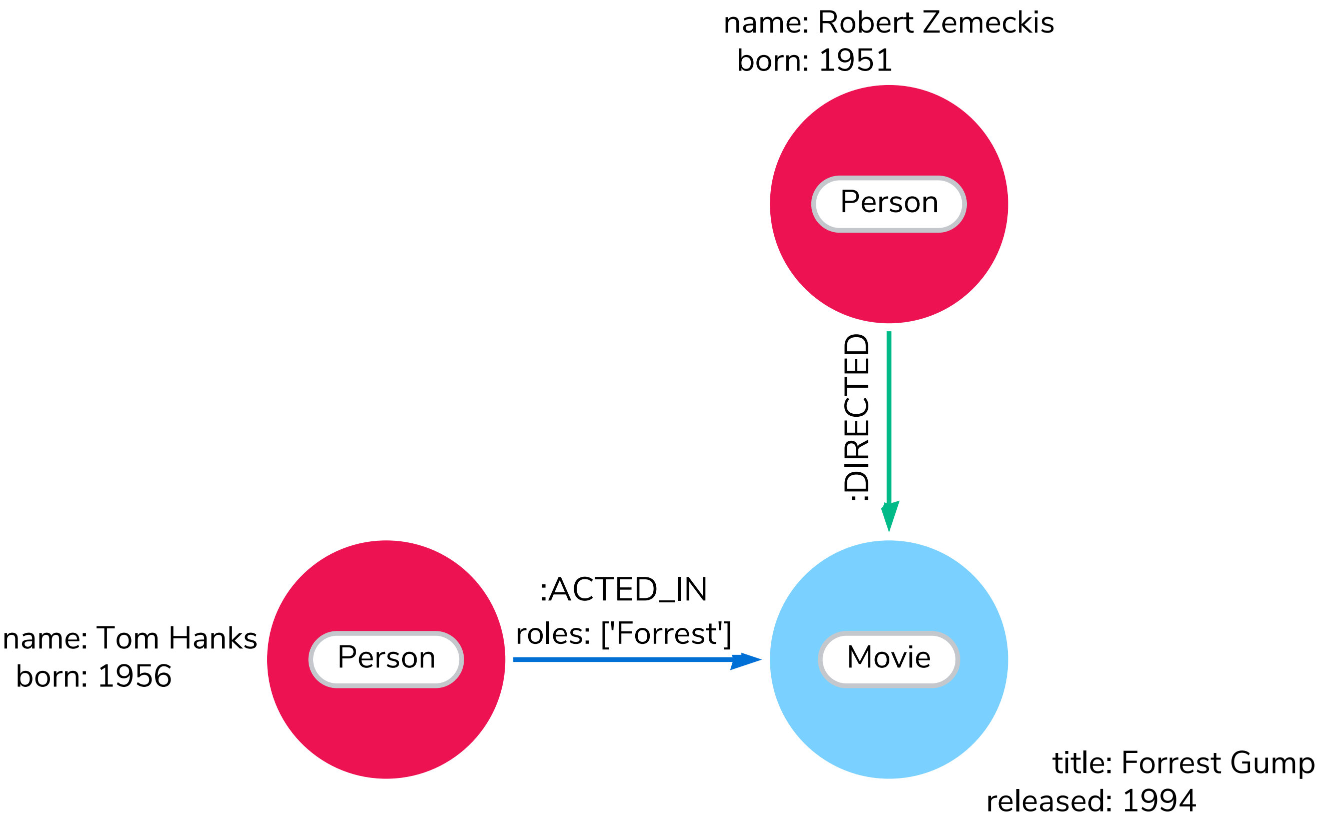
```
Created Nodes: 1
Added Labels: 1
Set Properties: 2
Rows: 1

+-----------------------------------------------+
| p                                             |
+-----------------------------------------------+
| (:Person {name: 'Keanu Reeves', born: 1964}) |
+-----------------------------------------------+
```

If you want to create more than one element, you can separate the elements with commas or use multiple `CREATE` statements.

You can, of course, also create more complex structures, like an `ACTED_IN` relationship with information about the character, or `DIRECTED` ones for the director.

```
CREATE (a:Person {name: 'Tom Hanks', born: 1956})-[r:ACTED_IN {roles: ['Forrest']}]->(m:Movie {title:
'Forrest Gump', released: 1994})
CREATE (d:Person {name: 'Robert Zemeckis', born: 1951})-[:DIRECTED]->(m)
RETURN a, d, r, m
```

This is the part of the updated graph:



In most cases, you want to add new data to existing structures. This requires knowing how to find existing patterns in your graph data, which is covered in the next section.

Matching patterns

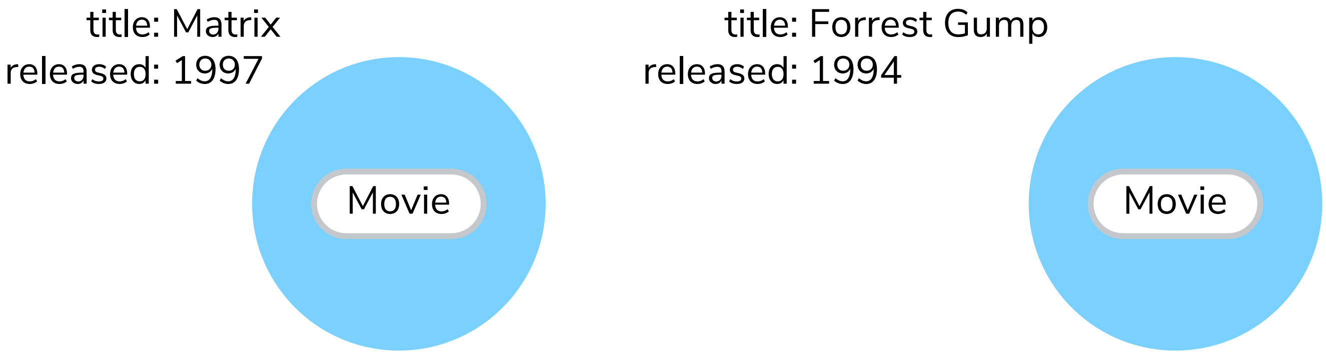
Matching patterns is a task for the MATCH statement. You pass the same kind of patterns you have used so far to MATCH to describe what you are looking for. It is similar to *query by example*, only that your examples also include the structures. To bring back nodes, relationships, properties, or patterns, you need to have variables specified in your MATCH clause for the data you want to return.

> A MATCH statement searches for the patterns you specify and return *one row per successful pattern match.*

To find the data you have created so far, you can start looking for all nodes labeled with the Movie label.

```
MATCH (m:Movie)
RETURN m
```

Here's the result:

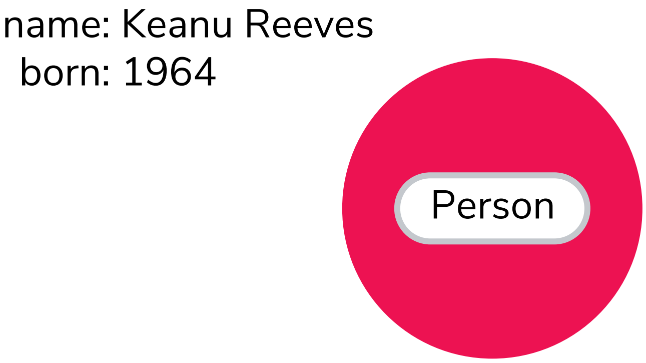


This should show both *The Matrix* and *Forrest Gump*.

You can also look for a specific person, like *Keanu Reeves*.

```
MATCH (p:Person {name: 'Keanu Reeves'})
RETURN p
```

This query returns the matching node:



Note that you only provide enough information to find the nodes, not all properties are required. In most cases, you have key-properties like SSN, ISBN, emails, logins, geolocation, or product codes to look for.

You can also find more interesting connections, like, for instance, the movies' titles that *Tom Hanks* acted in and roles he played.

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

```
Rows: 1

+------------------------------+
| m.title        | r.roles     |
+------------------------------+
| 'Forrest Gump' | ['Forrest'] |
+------------------------------+
```

In this case, you only returned the properties of the nodes and relationships that you are interested in. You can access them everywhere via a dot notation `identifer.property`.

Of course, this only lists T. Hank's role as *Forrest* in *Forrest Gump* because that's all data that you have added.

Now you know enough to add new nodes to existing ones and can combine `MATCH` and `CREATE` to attach structures to the graph.

Cypher examples

Let us look at some examples of using `MATCH` and `RETURN` keywords. Each subsequent example is more complicated than the previous one. The two last examples start with explanations of what we are trying to achieve. If you click the **Run Query** button below each Cypher code snippet, you can see the results in the format of a graph or table.

- **Example 1:** Find the labeled `Person` nodes in the graph. Note that you must use a variable like `p` for the `Person` node if you want to retrieve the node in the `RETURN` clause.

```
MATCH (p:Person)
RETURN p
LIMIT 1
```

- **Example 2:** Find `Person` nodes in the graph that have a name of 'Tom Hanks'. Remember that you can name your variable anything you want, as long as you reference that same name later.

```
MATCH (tom:Person {name: 'Tom Hanks'})
RETURN tom
```

- **Example 3:** Find which `Movie` Tom Hanks has directed.

Explanation: at first you should find Tom Hanks' `Person` node and after that the `Movie` nodes he is connected to. To do that, you have to follow the `DIRECTED` relationship from Tom Hanks' `Person` node to the `Movie` node. You have also specified a label of `Movie` so that the query only looks at nodes with that label. Since you only care about returning the movie in this query, you need to give that node a variable (`movie`) but do not need to give variables for the `Person` node or `DIRECTED` relationship.

```
MATCH (:Person {name: 'Tom Hanks'})-[:DIRECTED]->(movie:Movie)
RETURN movie
```

- **Example 4:** Find which `Movie` Tom Hanks has directed, but this time, return only the title of the movie.

Explanation: this query is similar to the previous one. Example 3 returned the entire `Movie` node with all its properties. For this example, you still need to find Tom's movies, but now you only care about their titles. You should access the node's `title` property using the syntax `variable.property` to return the name value.

```
MATCH (:Person {name: 'Tom Hanks'})-[:DIRECTED]->(movie:Movie)
RETURN movie.title
```

Aliasing return values

Not all properties are simple like `movie.title` in the example above. Some properties have poor names due to property length, multi-word descriptions, developer jargon, and other shortcuts. These naming conventions can be difficult to read, especially if they end up on reports and other user-facing interfaces.

Poorly-named properties

```
//poorly-named property
MATCH (tom:Person {name:'Tom Hanks'})-[rel:DIRECTED]-(movie:Movie)
RETURN tom.name, tom.born, movie.title, movie.released
```

Just like with SQL, you can rename return results by using the `AS` keyword and aliasing the property with a cleaner name.

Cleaner Results with aliasing

```
//cleaner printed results with aliasing
MATCH (tom:Person {name:'Tom Hanks'})-[rel:DIRECTED]-(movie:Movie)
RETURN tom.name AS name, tom.born AS `Year Born`, movie.title AS title, movie.released AS `Year Released`
```

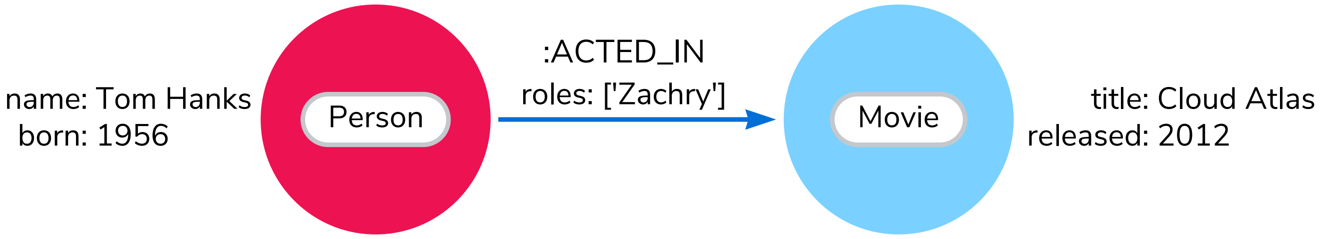
You can specify return aliases that have spaces by using the backtick character before and after the alias (movie.released AS `Year Released`). If you do not have an alias that contains spaces, then you do not need to use backticks.

Attaching structures

To extend the graph with new information, you first match the existing connection points and then attach the newly created nodes to them with relationships. Adding *Cloud Atlas* as a new movie for *Tom Hanks* could be achieved like this:

```
MATCH (p:Person {name: 'Tom Hanks'})
CREATE (m:Movie {title: 'Cloud Atlas', released: 2012})
CREATE (p)-[r:ACTED_IN {roles: ['Zachry']}]->(m)
RETURN p, r, m
```

Here's what the structure looks like in the database:



It is important to remember that you can assign variables to both nodes and relationships and use them later on, no matter if they were created or matched.

It is possible to attach both node and relationship in a single CREATE clause. For readability, it helps to split them up though.

A tricky aspect of the combination of MATCH and CREATE is that you get *one row per matched pattern*. This causes subsequent CREATE statements to be executed once for each row. In many cases, this is what you want. If that's not intended, move the CREATE statement before the MATCH, or change the cardinality of the query with means discussed later or use the *get or create* semantics of the next clause: **MERGE**.

Completing patterns

Whenever you get data from external systems or are not sure if certain information already exists in the graph, you want to be able to express a repeatable (idempotent) update operation. In Cypher **MERGE** clause has this function. It acts like a combination of MATCH or CREATE, which checks for the existence of data before creating it. With MERGE, you define a pattern to be found or created. Usually, as with MATCH, you only want to include the key property to look for in your core pattern. MERGE allows you to provide additional properties you want to set ON CREATE.

If you do not know whether your graph already contained *Cloud Atlas*, you could merge it again.

```
MERGE (m:Movie {title: 'Cloud Atlas'})
ON CREATE SET m.released = 2012
RETURN m
```

```
Created Nodes: 1
Added Labels: 1
Set Properties: 2
Rows: 1

+-------------------------------------------------+
| m                                               |
+-------------------------------------------------+
| (:Movie {title: 'Cloud Atlas', released: 2012}) |
+-------------------------------------------------+
```

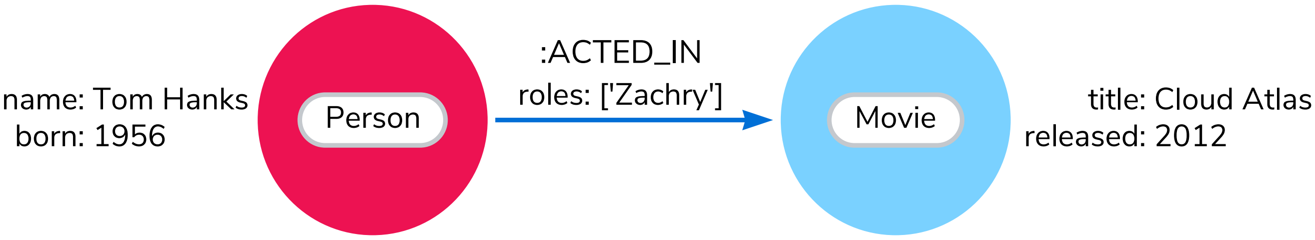
You get a result in both cases: either the data (potentially more than one row) that was already in the graph or a single, newly created Movie node.

A MERGE clause without any previously assigned variables in it either matches the full pattern or creates the full pattern. It never produces a partial mix of matching and creating within a pattern. To achieve a partial match/create, make sure to use already defined variables for the parts that shouldn't be affected.

So foremost MERGE makes sure that you can't create duplicate information or structures, but it comes with the cost of needing to check for existing matches first. Especially on large graphs, it can be costly to scan a large set of labeled nodes for a specific property. You can alleviate some of that by creating supporting indexes or constraints, which are discussed in the upcoming sections. But it's still not for free, so whenever you're sure to not create duplicate data use CREATE over MERGE.

MERGE can also assert that a relationship is only created once. For that to work you *have to pass in* both nodes from a previous pattern match.

```
MATCH (m:Movie {title: 'Cloud Atlas'})
MATCH (p:Person {name: 'Tom Hanks'})
MERGE (p)-[r:ACTED_IN]->(m)
ON CREATE SET r.roles =['Zachry']
RETURN p, r, m
```

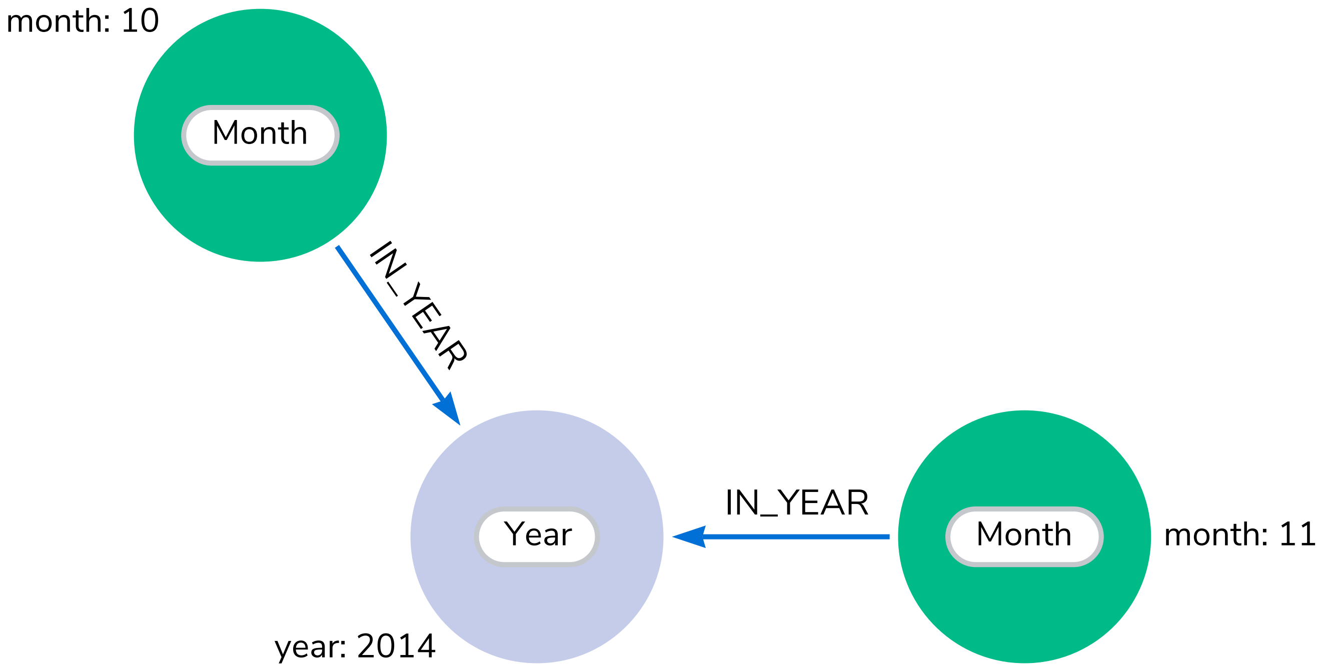


If the direction of a relationship is arbitrary, you can leave off the arrowhead. MERGE checks for the relationship in either direction and creates a new directed relationship if there is no matching relationship.

If you choose to pass in only one node from a preceding clause, MERGE offers an interesting functionality. It only matches within the direct neighborhood of the provided node for the given pattern, and if the pattern is not found creates it. This can come in very handy for creating, for example, tree structures.

```
CREATE (y:Year {year: 2014})
MERGE (y)<-[:IN_YEAR]-(m10:Month {month: 10})
MERGE (y)<-[:IN_YEAR]-(m11:Month {month: 11})
RETURN y, m10, m11
```

This is the graph structure that gets created:



Here is no global search for the two `Month` nodes; they are only searched for in the context of the *2014* `Year` node.

Code challenge Arrange

Now knowing the basics, use the parts below to build a Cypher statement to find the `title` and year of `release` for every `:Movie` that Tom Hanks has `:DIRECTED`. Click the parts to add them in order and once you are done, click **Run Query** to see whether you have got it right. You can click any part of the query inside the code block to remove it.

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

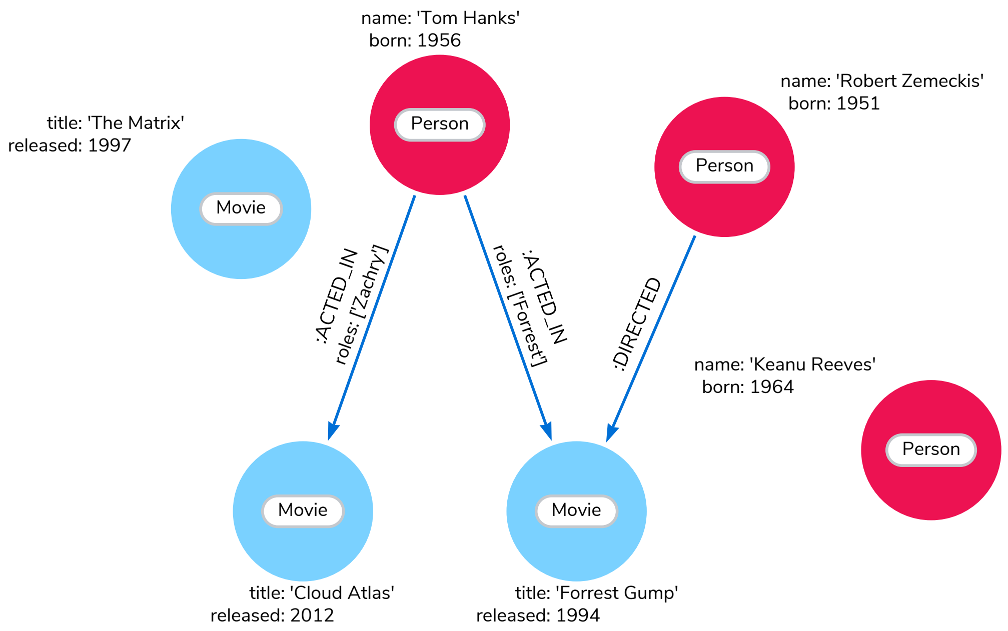
Getting the correct results

Example graphs

In this section, two example datasets are used. The first graph is based on the Movie database. The following code block helps you to create the data for exploring Cypher queries:

```
CREATE (matrix:Movie {title: 'The Matrix', released: 1997})
CREATE (cloudAtlas:Movie {title: 'Cloud Atlas', released: 2012})
CREATE (forrestGump:Movie {title: 'Forrest Gump', released: 1994})
CREATE (keanu:Person {name: 'Keanu Reeves', born: 1964})
CREATE (robert:Person {name: 'Robert Zemeckis', born: 1951})
CREATE (tom:Person {name: 'Tom Hanks', born: 1956})
CREATE (tom)-[:ACTED_IN {roles: ['Forrest']}]->(forrestGump)
CREATE (tom)-[:ACTED_IN {roles: ['Zachry']}]->(cloudAtlas)
CREATE (robert)-[:DIRECTED]->(forrestGump)
```

This is the resulting graph:



The second dataset is a small network of people, companies they work for, and technologies they like. You can find its image in the following chapter.

Filtering results

So far you have matched patterns in the graph and always returned all results you found. Now let's look into options for filtering the results and only return the subset of data that you are interested in. Those filter conditions are expressed using the `WHERE` clause. This clause allows to use any number of boolean expressions, *predicates*, combined with `AND`, `OR`, `XOR` and `NOT`. The simplest predicates are comparisons; especially equality.

```
MATCH (m:Movie)
WHERE m.title = 'The Matrix'
RETURN m
```

```
Rows: 1

+------------------------------------------------+
| m                                              |
+------------------------------------------------+
| (:Movie {title: 'The Matrix', released: 1997}) |
+------------------------------------------------+
```

The query above, using the `WHERE` clause, is equivalent to this query which includes the condition in the pattern matching:

```
MATCH (m:Movie {title: 'The Matrix'})
RETURN m
```

Cypher is designed to be flexible, so there is often more than one way to write a query.

Other options are numeric comparisons, matching regular expressions, and checking the existence of values within a list.

The WHERE clause in the following example includes a regular expression match, a greater-than comparison, and a test to see if a value exists in a list:

```
MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
WHERE p.name =~ 'K.+' OR m.released > 2000 OR 'Neo' IN r.roles
RETURN p, r, m
```

```
Rows: 1

+---------------------------------------------------------------------------------------------------------
----------------------+
| p                                         | r                               | m
|
+---------------------------------------------------------------------------------------------------------
----------------------+
| (:Person {name: 'Tom Hanks', born: 1956}) | [:ACTED_IN {roles: ['Zachry']}] | (:Movie {title: 'Cloud
Atlas', released: 2012}) |
+---------------------------------------------------------------------------------------------------------
----------------------+
```

An advanced aspect is that patterns can be used as predicates. Where MATCH expands the number and shape of patterns matched, a pattern predicate restricts the current result set. It only allows the paths to pass that satisfy the specified pattern. As you can expect, the use of NOT only allows the paths to pass that do *not* satisfy the specified pattern.

```
MATCH (p:Person)-[:ACTED_IN]->(m)
WHERE NOT (p)-[:DIRECTED]->()
RETURN p, m
```

```
Rows: 2

+----------------------------------------------------------------------------------------------+
| p                                         | m                                                |
+----------------------------------------------------------------------------------------------+
| (:Person {name: 'Tom Hanks', born: 1956}) | (:Movie {title: 'Cloud Atlas', released: 2012})  |
| (:Person {name: 'Tom Hanks', born: 1956}) | (:Movie {title: 'Forrest Gump', released: 1994}) |
+----------------------------------------------------------------------------------------------+
```

Here you are able to find actors because they sport an ACTED_IN relationship but then skip those that ever DIRECTED any movie.

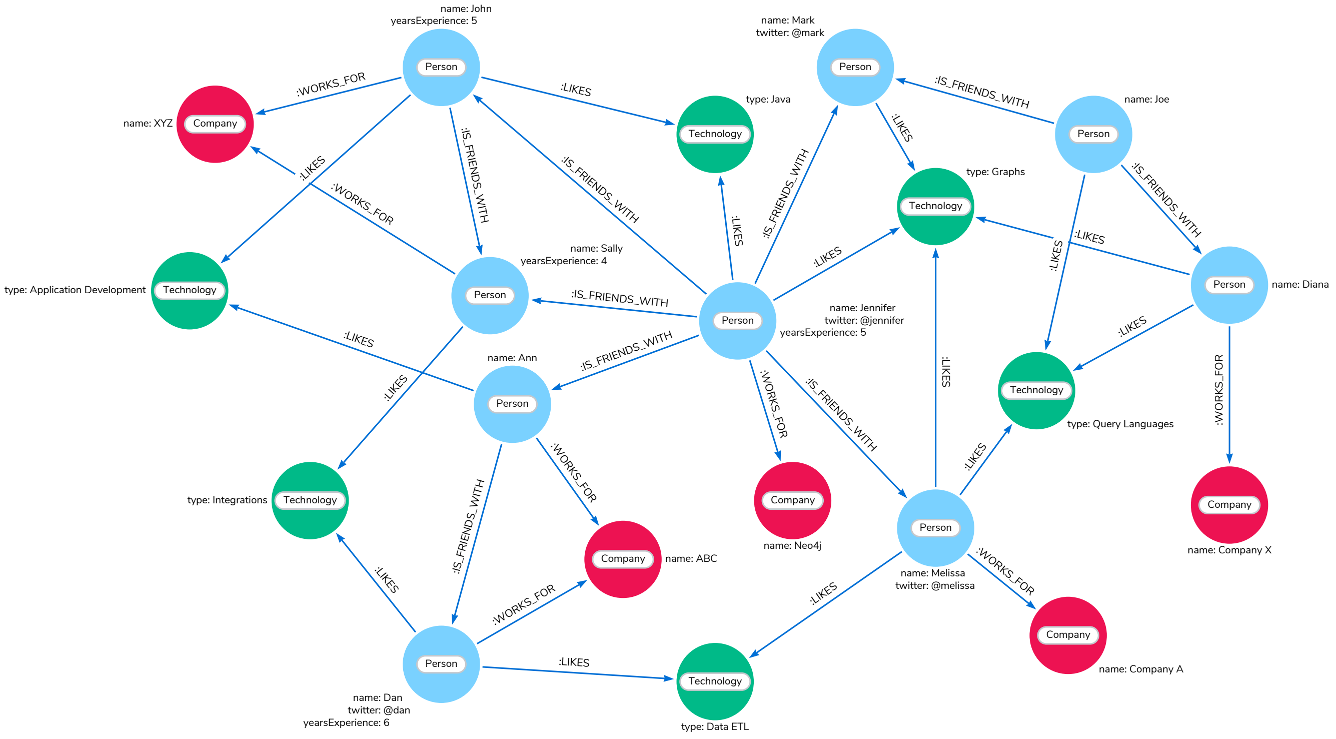
There are more advanced ways of filtering, for example *list predicates*, which will be discussed later in this section.

Querying ranges of values

There are frequent queries where you want to look for data within a certain range. Date or number ranges can be used to check for events within a certain timeline, age values, or other uses.

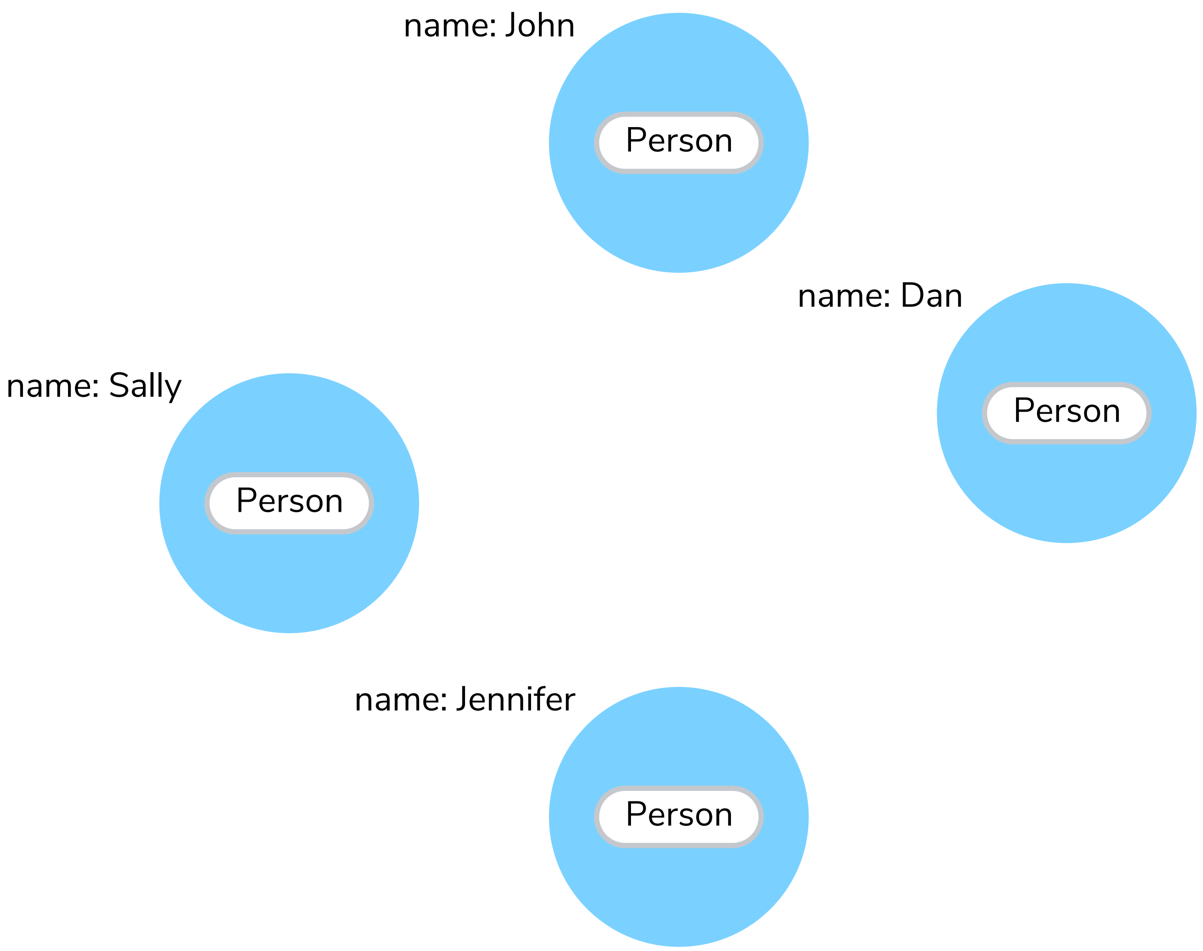
The syntax for this criteria is very similar to SQL and other programming language logic structures for checking ranges of values.

The following dataset is used to demonstrate the Cypher queries for these cases.



Imagine you would like to know who possesses experience ranging from three to seven years. The code block below shows the Cypher query for this case.

```
MATCH (p:Person)
WHERE 3 <= p.yearsExp <= 7
RETURN p
```



Testing if a property exists

You may only be interested if a property exists on a node or relationship. For instance, you might want to check which customers in your system have Twitter handles, so you can show relevant content. Or, you could check if all of your employees have a start date property to verify which entities might need to be updated.

> ❗ Remember: in Neo4j, a property only exists (is stored) if it has a value. A `null` property is not stored. This ensures that only valuable, necessary information is retained for your nodes and relationships.

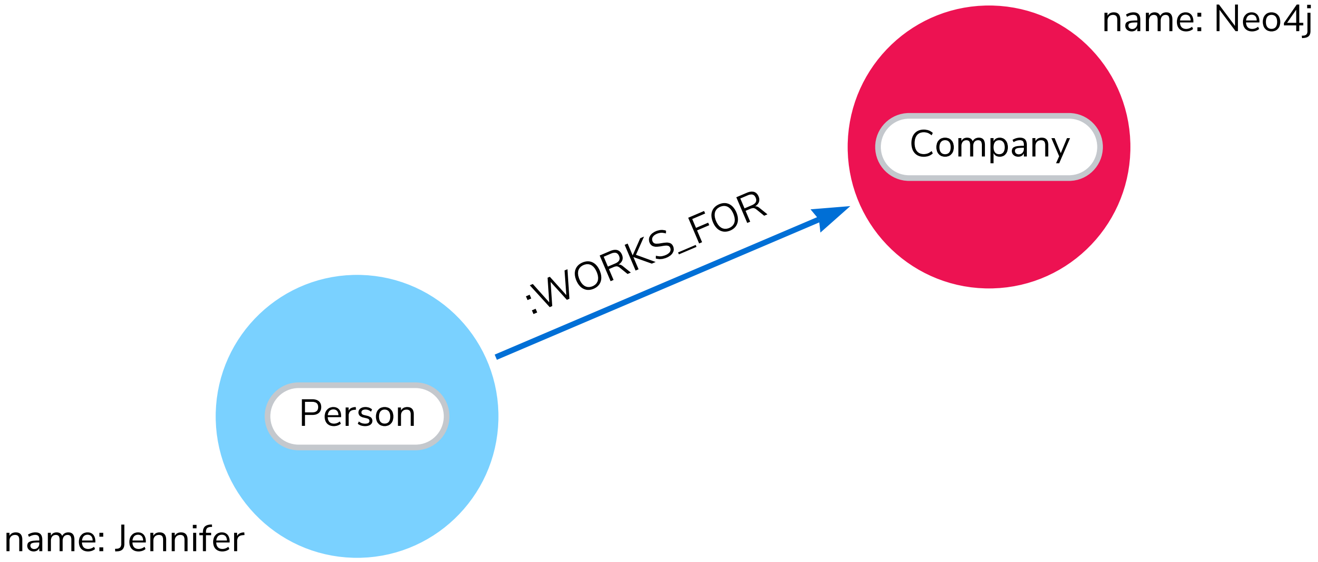
To write this type of existence check, you need to use the `WHERE` clause and the `exists()` method for that property. The Cypher code is written in the block below.

```
//Query1: find all users who have a birthdate property
MATCH (p:Person)
WHERE exists(p.birthdate)
RETURN p.name;

//Query2: find all WORKS_FOR relationships that have a startYear property
MATCH (p:Person)-[rel:WORKS_FOR]->(c:Company)
WHERE exists(rel.startYear)
RETURN p, rel, c;
```

Query1 results:

```
Rows: 9

+---------------------------------+
| p.name                          |
+---------------------------------+
| 'Jennifer'                      |
| 'Melissa'                       |
| 'Diana'                         |
| 'Mark'                          |
| 'Dan'                           |
| 'John'                          |
| 'Sally'                         |
| 'Joe'                           |
| 'Ann'                           |
+---------------------------------+
```

Query2 results:



Checking strings — partial values, fuzzy searches

Some scenarios require query syntax that matches on partial values or broad categories within a string. To do this kind of query, you need some flexibility and options for string matching and searches. Whether you are looking for a string that starts with, ends with, or includes a certain value, Cypher offers the ability to handle it performantly and easily.

There are a few keywords in Cypher used with the `WHERE` clause to test string property values. The `STARTS WITH` keyword allows you check the value of a property that begins with the string you specify. With the `CONTAINS` keyword, you can check if a specified string is part of a property value. The `ENDS_WITH` keyword checks the end of the property string for the value you specify.

An example of each is in the following Cypher block.

```
//check if a property starts with 'M'
MATCH (p:Person)
WHERE p.name STARTS WITH 'M'
RETURN p.name;

//check if a property contains 'a'
MATCH (p:Person)
WHERE p.name CONTAINS 'a'
RETURN p.name;

//check if a property ends with 'n'
MATCH (p:Person)
WHERE p.name ENDS WITH 'n'
RETURN p.name;
```

You can also use regular expressions to test the value of strings. For example, you could look for all the `Person` nodes that share a first name or you could find all the classes with a certain department code.

Let’s look at an example.

```
MATCH (p:Person)
WHERE p.name =~ 'Jo.*'
RETURN p.name
```

```
Rows: 2

+--------------------------------+
| p.name                         |
+--------------------------------+
| 'John'                         |
| 'Joe'                          |
+--------------------------------+
```

Just like in SQL and other languages, you can check if a property value is a value in a list. The `IN` keyword allows you to specify an array of values and validate a property's contents against each one in the list.

Here is an example:

```
MATCH (p:Person)
WHERE p.yearsExp IN [1, 5, 6]
RETURN p.name, p.yearsExp
```

```
Rows: 3

+--------------------------------+
| p.name      | p.yearsExp       |
+--------------------------------+
| 'Jennifer'  | 5                |
| 'Dan'       | 6                |
| 'John'      | 5                |
+--------------------------------+
```

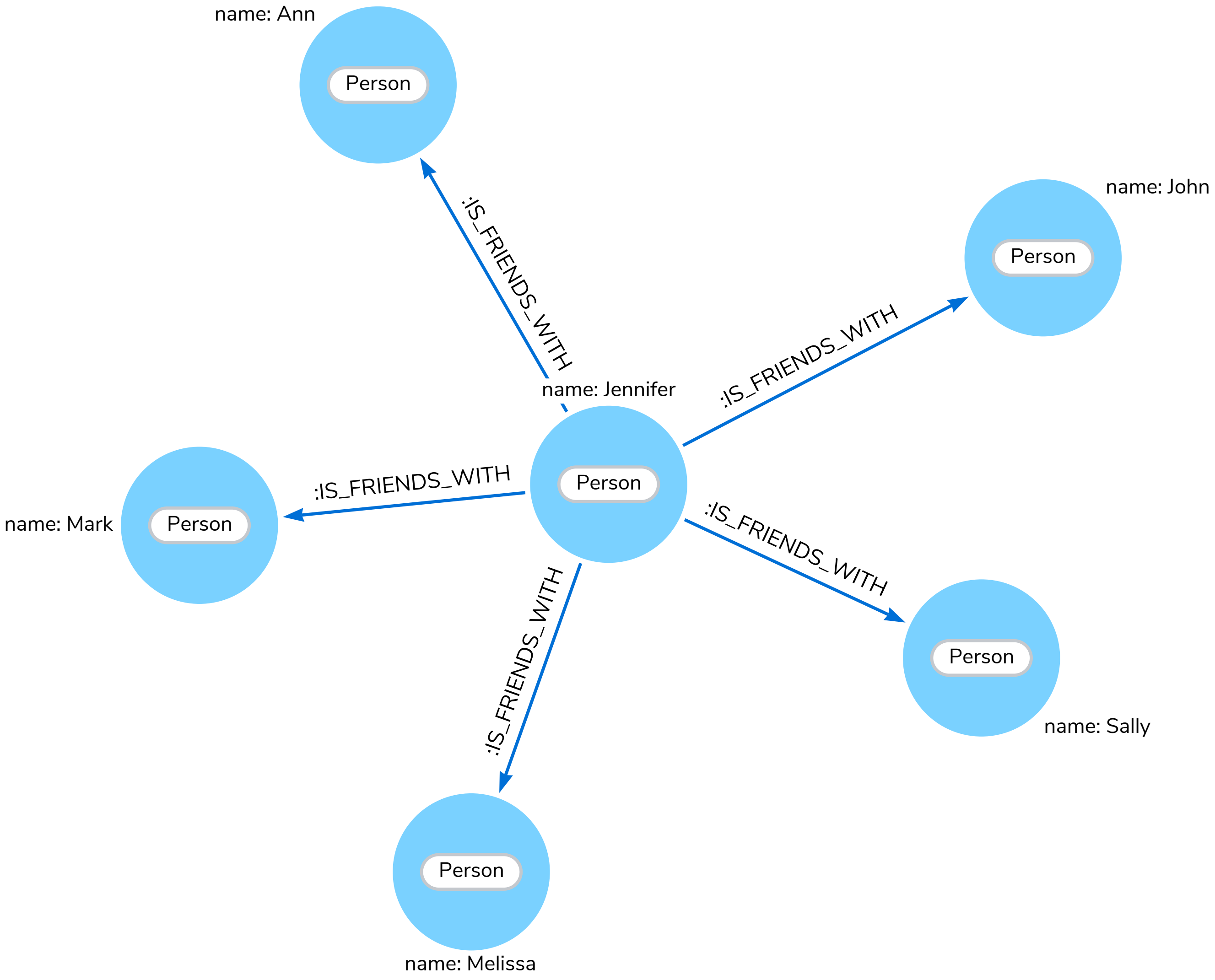
Filtering on patterns

One thing that makes graph unique is its focus on relationships. Just as you can filter queries based on node labels or properties, you can also filter results based on relationships or patterns. This allows you to test if a pattern also has a certain relationship or does not, or if another pattern exists.

The following Cypher code shows how this is done.

```
//Query1: find which people are friends of someone who works for Neo4j
MATCH (p:Person)-[r:IS_FRIENDS_WITH]->(friend:Person)
WHERE exists((p)-[:WORKS_FOR]->(:Company {name: 'Neo4j'}))
RETURN p, r, friend;

//Query2: find Jennifer's friends who do not work for a company
MATCH (p:Person)-[r:IS_FRIENDS_WITH]->(friend:Person)
WHERE p.name = 'Jennifer'
AND NOT exists((friend)-[:WORKS_FOR]->(:Company))
RETURN friend.name;
```

Query1 results:



Query2 results:

```
Rows: 1

+----------------------------------+
| friend.name                      |
+----------------------------------+
| 'Mark'                           |
+----------------------------------+
```

Optional patterns

There are cases where you might want to retrieve results from patterns, even if they do not match the entire pattern or all of the criteria. This is how an outer join in SQL functions. In Cypher, you can use an `OPTIONAL MATCH` pattern to try to match it, but if it doesn't find results, those rows will return `null` for those values.

You can see how this would look in Cypher by querying for people whose name starts with a specific letter and who may work for a company.

```
//Find all people whose name starts with J and who may work for a company.
MATCH (p:Person)
WHERE p.name STARTS WITH 'J'
OPTIONAL MATCH (p)-[:WORKS_FOR]-(other:Company)
RETURN p.name, other.name;
```

```
Rows: 3

+----------------------------------+
| p.name      | other.name         |
+----------------------------------+
| 'Jennifer'  | 'Neo4j'            |
| 'John'      | 'XYZ'              |
| 'Joe'       | null               |
+----------------------------------+
```

Notice that Joe is returned because his name starts with the letter 'J', but his company's name is `null`. That is because he does not have a `WORKS_FOR` relationship to a `COMPANY` node. Since you used `OPTIONAL MATCH`, his `Person` node is still returned from the first match, but the second match is not found, so returns `null`.

> To see the difference, try running the query without the `OPTIONAL` in front of the second match. You can see that Joe's row is no longer returned. That is because Cypher reads the statement with an `AND` match, so the person must match the first criteria (name starts with 'J') and the second criteria (person works for a company).

More complex patterns

You are able to handle many simple graph queries even at this point. But what happens when you want to extend your patterns past a single relationship? What if you want to know who else likes graphs besides Jennifer?

We handle this functionality and many others by simply adding on to our first pattern or matching additional patterns. Let us look at a couple of examples.

```
//Query1: find who likes graphs besides Jennifer
MATCH (j:Person {name: 'Jennifer'})-[r:LIKES]-(graph:Technology {type: 'Graphs'})-[r2:LIKES]-(p:Person)
RETURN p.name;

//Query2: find who likes graphs besides Jennifer that she is also friends with
MATCH (j:Person {name: 'Jennifer'})-[:LIKES]->(:Technology {type: 'Graphs'})<-[:LIKES]-(p:Person),
      (j)-[:IS_FRIENDS_WITH]-(p)
RETURN p.name;
```

Query1 results:

```
Rows: 3

+------------------------+
| p.name                 |
+------------------------+
| 'Diana'                |
| 'Mark'                 |
| 'Melissa'              |
+------------------------+
```

Query2 results:

```
Rows: 2

+------------------------+
| p.name                 |
+------------------------+
| 'Mark'                 |
| 'Melissa'              |
+------------------------+
```

Notice that on the second query a comma is used after the first `MATCH` line and another pattern is added to match on the next line. This allows you to chain patterns together, similar to when you used the `WHERE exists(<pattern>)` syntax above. With this structure, you can add multiple different patterns and link them together, allowing you to traverse various pieces of the graph with certain patterns.

Returning results

So far, you have returned nodes, relationships, and paths directly via their variables. However, the `RETURN` clause can return any number of expressions. But what are expressions in Cypher?

The simplest expressions are literal values. Examples of literal values are: numbers, strings, arrays (for example: `[1,2,3]`), and maps (for example: `{name: 'Tom Hanks', born:1964, movies: ['Forrest Gump', ...], count: 13}`). Individual properties of any node, relationship or map can be accessed using the dot syntax, for example: `n.name`. Individual elements or slices of arrays can be retrieved with subscripts, for example: `names[0]` and `movies[1..-1]`. Each function evaluation, for example: `length(array)`, `toInteger('12')`, `substring('2014-07-01', 0, 4)` and `coalesce(p.nickname, 'n/a')`, is also an expression.

Predicates used in `WHERE` clauses count as *boolean expressions*.

Simple expressions can be composed and concatenated to form more complex expressions.

By default the expression itself is used as a label for the column, in many cases you want to alias that with a more understandable name using `expression AS alias`. The alias can be used subsequently to refer to that column.

```
MATCH (p:Person)
RETURN
  p,
  p.name AS name,
  toUpper(p.name),
  coalesce(p.nickname, 'n/a') AS nickname,
  {name: p.name, label: head(labels(p))} AS person
```

```
Rows: 3

+---------------------------------------------------------------------------------------------------------
----------------------------------------+
| p                                                | name               | toUpper(p.name)    | nickname |
person                                     |
+---------------------------------------------------------------------------------------------------------
----------------------------------------+
| (:Person {name: 'Keanu Reeves', born: 1964})    | 'Keanu Reeves'     | 'KEANU REEVES'     | 'n/a'    |
{name: 'Keanu Reeves', label: 'Person'}    |
| (:Person {name: 'Robert Zemeckis', born: 1951}) | 'Robert Zemeckis' | 'ROBERT ZEMECKIS' | 'n/a'    |
{name: 'Robert Zemeckis', label: 'Person'} |
| (:Person {name: 'Tom Hanks', born: 1956})       | 'Tom Hanks'        | 'TOM HANKS'        | 'n/a'    |
{name: 'Tom Hanks', label: 'Person'}       |
+---------------------------------------------------------------------------------------------------------
----------------------------------------+
```

If you wish to display only unique results you can use the `DISTINCT` keyword after `RETURN`:

```
MATCH (n)
RETURN DISTINCT labels(n) AS Labels
```

```
Rows: 2

+------------+
| Labels     |
+------------+
| ['Movie']  |
| ['Person'] |
+------------+
```

Returning unique results

You can return unique results using `DISTINCT` keyword in Cypher. Some of your queries may return duplicate results due to multiple paths to the node or a node that meets multiple criteria. This redundancy can clutter results and make sifting through a long list difficult to find what you need.

To trim out duplicate entities, you can use the `DISTINCT` keyword.

```
//Query: find people who have a twitter or like graphs or query languages
MATCH (user:Person)
WHERE user.twitter IS NOT null
WITH user
MATCH (user)-[:LIKES]-(t:Technology)
WHERE t.type IN ['Graphs','Query Languages']
RETURN DISTINCT user.name
```

Query results:

```
Rows: 3

+-----------------------+
| user.name             |
+-----------------------+
| 'Jennifer'            |
| 'Melissa'             |
| 'Mark'                |
+-----------------------+
```

For the preceding query, the use case is that you are launching a new Twitter account for tips and tricks on

Cypher, and you want to notify users who have a Twitter account and who like graphs or query languages. The first two lines of the query look for `Person` nodes that have a Twitter handle. Then, you use `WITH` to pass those users over to the next `MATCH`, where you find out if the person likes graphs or query languages. Notice that running this statement without the `DISTINCT` keyword results in 'Melissa' shown twice. This is because she likes graphs and she also likes query languages. When `DISTINCT` is used, you only retrieve unique users.

Limiting number of results

There are times when you want a sampling set, or you only want to pull so many results to update or process at a time. The `LIMIT` keyword takes the output of the query and limits the volume returned based on the number you specify.

For instance, you can find each person’s number of friends in our graph. If the graph were thousands or millions of nodes and relationships, the number of results returned would be massive. What if you only cared about the top three people who had the most friends? Let’s write a query for that!

```
//Query: find the top 3 people who have the most friends
MATCH (p:Person)-[r:IS_FRIENDS_WITH]-(other:Person)
RETURN p.name, count(other.name) AS numberOfFriends
ORDER BY numberOfFriends DESC
LIMIT 3
```

```
Rows: 3

+-------------------------------+
| p.name      | numberOfFriends  |
+-------------------------------+
| 'Jennifer'  | 5                |
| 'Mark'      | 2                |
| 'Ann'       | 2                |
+-------------------------------+
```

The query pulls persons and the friends they are connected to and returns the person name and count of their friends. You could run just that much of the query and return a messy list of names and friend counts, but you probably want to order the list based on the number of friends each person has starting with the biggest number at the top (`DESC`). You could also run that much of the query to see the friends and counts all in order, but in the example above the top three people with the most friends have been pulled from the graph. The `LIMIT` pulls the top results from the ordered list.

Try mixing up the query by removing the `ORDER BY` and `LIMIT` lines and then add each one separately. Notice that only removing the `ORDER BY` line pulls the starting three values from the list, getting a random sampling of the return results.

Aggregating information

In many cases, we wish to aggregate or group the data encountered while traversing patterns in our graph. In Cypher, aggregation happens in the `RETURN` clause while computing the final results. Many common aggregation functions are supported, for example `count`, `sum`, `avg`, `min`, and `max`, but there are several more.

Counting the number of people in your Movie database could be achieved by this:

```
MATCH (:Person)
RETURN count(*) AS people
```

```
Rows: 1

+---------+
| people |
+---------+
| 3      |
+---------+
```

Note that `NULL` values are skipped during aggregation. For aggregating only unique values use `DISTINCT`, for example: `count(DISTINCT role)`.

Aggregation works implicitly in Cypher. You specify which result columns you wish to aggregate. Cypher uses all non-aggregated columns as grouping keys.

Aggregation affects which data is still visible in ordering or later query parts.

The following statement finds out how often an actor and director have worked together:

```
MATCH (actor:Person)-[:ACTED_IN]->(movie:Movie)<-[:DIRECTED]-(director:Person)
RETURN actor, director, count(*) AS collaborations
```

```
Rows: 1

+---------------------------------------------------------------------------------------------------------
-----+
| actor                                     | director                                        |
collaborations |
+---------------------------------------------------------------------------------------------------------
-----+
| (:Person {name: 'Tom Hanks', born: 1956}) | (:Person {name: 'Robert Zemeckis', born: 1951}) | 1
|
+---------------------------------------------------------------------------------------------------------
-----+
```

The `count()` function in Cypher allows you to count the number of occurences of entities, relationships, or results returned.

There are two different ways you can count return results from your query.

- The first is by using `count(n)` to count the number of occurences of `n` and does not include `null` values. You can specify nodes, relationships, or properties within the parentheses for Cypher to count.
- The second way to count results is with `count(*)`, which counts the number of result rows returned (including those with `null` values).

In the dataset, some of the `Person` nodes have a Twitter handle, but others do not. If you run the first example query below, you will see that the `twitter` property has a value for four people and is `null` for the other five people. The second and third queries show how to use the different `count` options.

```
//Query1: see the list of Twitter handle values for Person nodes
MATCH (p:Person)
RETURN p.twitter;
```

Query1 results:

```
Rows: 9

+--------------+
| p.twitter    |
+--------------+
| '@jennifer'  |
| '@melissa'   |
| null         |
| '@mark'      |
| '@dan'       |
| null         |
| null         |
| null         |
| null         |
+--------------+
```

```
//Query2: count of the non-null `twitter` property of the Person nodes
MATCH (p:Person)
RETURN count(p.twitter);
```

Query2 results:

```
Rows: 1

+--------------------+
| count(p.twitter)   |
+--------------------+
| 4                  |
+--------------------+
```

```
//Query3: count on the Person nodes
MATCH (p:Person)
RETURN count(*);
```

Query3 results:

```
Rows: 1

+--------------------+
| count(*)           |
+--------------------+
| 9                  |
+--------------------+
```

Collecting aggregation

A very helpful aggregation function is `collect(expression)`, which returns a single aggregated list of the values returned by an expression. This is very useful in many situations, since no information of details is lost while aggregating.

`collect()` is well-suited for retrieving typical parent-child structures, where one core entity (parent, root, or *head*) is returned per row with all its dependent information in associated lists created with `collect()`. This means that there is no need to repeat the parent information for each child row, or running `n+1` statements to retrieve the parent and its children individually.

The following statement could be used to retrieve the cast of each movie in our database:

```
MATCH (m:Movie)<-[:ACTED_IN]-(a:Person)
RETURN m.title AS movie, collect(a.name) AS cast, count(*) AS actors
```

```
Rows: 2

+-------------------------------------------+
| movie          | cast           | actors |
+-------------------------------------------+
| 'Forrest Gump' | ['Tom Hanks'] | 1      |
| 'Cloud Atlas'  | ['Tom Hanks'] | 1      |
+-------------------------------------------+
```

The lists created by `collect()` can either be used from the client consuming the Cypher results or directly within a statement with any of the list functions or predicates.

Looping through list values

If you have a list that you want to inspect or separate the values, Cypher offers the `UNWIND` clause. This does the opposite of `collect()` and separates a list into individual values on separate rows.

`UNWIND` is frequently used for looping through JSON and XML objects when importing data, as well as everyday arrays and other types of lists. Let us look at a couple of examples where we assume that the technologies someone likes also mean they have some experience with each one. If you are interested in hiring people who are familiar with `Graphs` or `Query Languages`, you can write the following query to find people to interview.

```
//Query1: for a list of techRequirements, look for people who have each skill
WITH ['Graphs','Query Languages'] AS techRequirements
UNWIND techRequirements AS technology
MATCH (p:Person)-[r:LIKES]-(t:Technology {type: technology})
RETURN t.type, collect(p.name) AS potentialCandidates;
```

Query1 results:

```
Rows: 2

+-------------------+-------------------------------------------+
| t.type            | potentialCandidates                       |
+-------------------+-------------------------------------------+
| 'Graphs'          | ['Diana', 'Mark', 'Melissa', 'Jennifer'] |
| 'Query Languages' | ['Diana', 'Melissa', 'Joe']              |
+-------------------+-------------------------------------------+
```

```
//Query2: for numbers in a list, find candidates who have that many years of experience
WITH [4, 5, 6, 7] AS experienceRange
UNWIND experienceRange AS number
MATCH (p:Person)
WHERE p.yearsExp = number
RETURN p.name, p.yearsExp;
```

Query2 results:

```
Rows: 4

+--------------+-----------------+
| p.name       | p.yearsExp      |
+--------------+-----------------+
| 'Sally'      | 4               |
| 'Jennifer'   | 5               |
| 'John'       | 5               |
| 'Dan'        | 6               |
+--------------+-----------------+
```

Ordering and pagination

It is common to sort and paginate after aggregating using `count(x)`.

Ordering is done using the `ORDER BY expression [ASC|DESC]` clause. The expression can be any expression, as long as it is computable from the returned information.

For instance, if you return `person.name` you can still `ORDER BY person.age` since both are accessible from the `person` reference. You cannot order by things that are not returned. This is especially important with aggregation and `DISTINCT` return values, since both remove the visibility of data that is aggregated.

Pagination is done using the `SKIP {offset}` and `LIMIT {count}` clauses.

A common pattern is to aggregate for a count (score or *frequency*), order by it, and only return the top-n entries.

For instance, to find the most prolific actors you could do:

```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)
RETURN a, count(*) AS appearances
ORDER BY appearances DESC LIMIT 10
```

```
Rows: 1

+-----------------------------------------------------------+
| a                                         | appearances |
+-----------------------------------------------------------+
| (:Person {name: 'Tom Hanks', born: 1956}) | 2           |
+-----------------------------------------------------------+
```

Ordering results

Our list of potential hiring candidates from the preceding examples might be more useful if you could order the candidates by most or least experience. Or perhaps you want to rank all of our people by age.

The `ORDER BY` keyword sorts the results based on the value you specify in ascending or descending order (ascending is default). Let’s use the same queries from our examples with `UNWIND` and see how you can order the candidates.

```
//Query1: for a list of techRequirements, look for people who have each skill
WITH ['Graphs','Query Languages'] AS techRequirements
UNWIND techRequirements AS technology
MATCH (p:Person)-[r:LIKES]-(t:Technology {type: technology})
WITH t.type AS technology, p.name AS personName
ORDER BY technology, personName
RETURN technology, collect(personName) AS potentialCandidates;
```

Query1 results:

```
Rows: 2

+-------------------+------------------------------------------+
| technology        | potentialCandidates                      |
+-------------------+------------------------------------------+
| 'Graphs'          | ['Diana', 'Jennifer', 'Mark', 'Melissa'] |
| 'Query Languages' | ['Diana', Joe]                           |
+-------------------+------------------------------------------+
```

```
//Query2: for numbers in a list, find candidates who have that many years of experience
WITH [4, 5, 6, 7] AS experienceRange
UNWIND experienceRange AS number
MATCH (p:Person)
WHERE p.yearsExp = number
RETURN p.name, p.yearsExp ORDER BY p.yearsExp DESC;
```

Query2 results:

```
Rows: 4

+--------------+-----------------+
| p.name       | p.yearsExp      |
+--------------+-----------------+
| 'Dan'        | 6               |
| 'Jennifer'   | 5               |
| 'John'       | 5               |
| 'Sally'      | 4               |
+--------------+-----------------+
```

Notice that the first query has to order by `Person` name before collecting the values into a list. If you do not sort first (put the `ORDER BY` after the `RETURN` clause), you will sort based on the size of the list and not by the first letter of the values in the list. The results are also sorted into two values: technology, then a person. This allows you to sort the technology so that all the persons that like a technology are listed together.

You can try out the difference in sorting by both values or one value by running the following queries:

```
//only sorted by person's name in alphabetical order
WITH ['Graphs','Query Languages'] AS techRequirements
UNWIND techRequirements AS technology
MATCH (p:Person)-[r:LIKES]-(t:Technology {type: technology})
WITH t.type AS technology, p.name AS personName
ORDER BY personName
RETURN technology, personName;
```

```
//only sorted by technology (person names are out of order)
WITH ['Graphs','Query Languages'] AS techRequirements
UNWIND techRequirements AS technology
MATCH (p:Person)-[r:LIKES]-(t:Technology {type: technology})
WITH t.type AS technology, p.name AS personName
ORDER BY technology
RETURN technology, personName;
```

```
//sorted by technology, then by person's name
WITH ['Graphs','Query Languages'] AS techRequirements
UNWIND techRequirements AS technology
MATCH (p:Person)-[r:LIKES]-(t:Technology {type: technology})
WITH t.type AS technology, p.name AS personName
ORDER BY technology, personName
RETURN technology, personName;
```

Counting values in a list

If you have a list of values, you can also find the number of items in that list or calculate the size of an expression using the `size()` function. The examples below return the number of items or patterns found.

```
//Query1: find number of items in collected list
MATCH (p:Person)-[:IS_FRIENDS_WITH]->(friend:Person)
RETURN p.name, size(collect(friend.name)) AS numberOfFriends;
```

Query1 results:

```
Rows: 4

+--------------+-----------------+
| p.name       | numberOfFriends |
+--------------+-----------------+
| 'John'       | 1               |
| 'Jennifer'   | 5               |
| 'Ann'        | 1               |
| 'Joe'        | 2               |
+--------------+-----------------+
```

```
//Query2: find number of friends who have other friends
MATCH (p:Person)-[:IS_FRIENDS_WITH]->(friend:Person)
WHERE size((friend)-[:IS_FRIENDS_WITH]-(:Person)) > 1
RETURN p.name, collect(friend.name) AS friends, size((friend)-[:IS_FRIENDS_WITH]-(:Person)) AS
numberOfFoFs;
```

Query2 results:

```
Rows: 3

+--------------+-----------------------------------+---------------+
| p.name       | friends                           | numberOfFofs  |
+--------------+-----------------------------------+---------------+
| 'Joe'        | ['Mark']                          | 2             |
| 'Jennifer'   | ['Mark', 'John', 'Sally', 'Ann'] | 2             |
| 'John'       | ['Sally']                         | 2             |
+--------------+-----------------------------------+---------------+
```

Resources

- Neo4j Cypher Manual: WITH, UNWIND, & More
- Neo4j Cypher Manual: Aggregation
- Neo4j Cypher Manual: size()

Updating the data

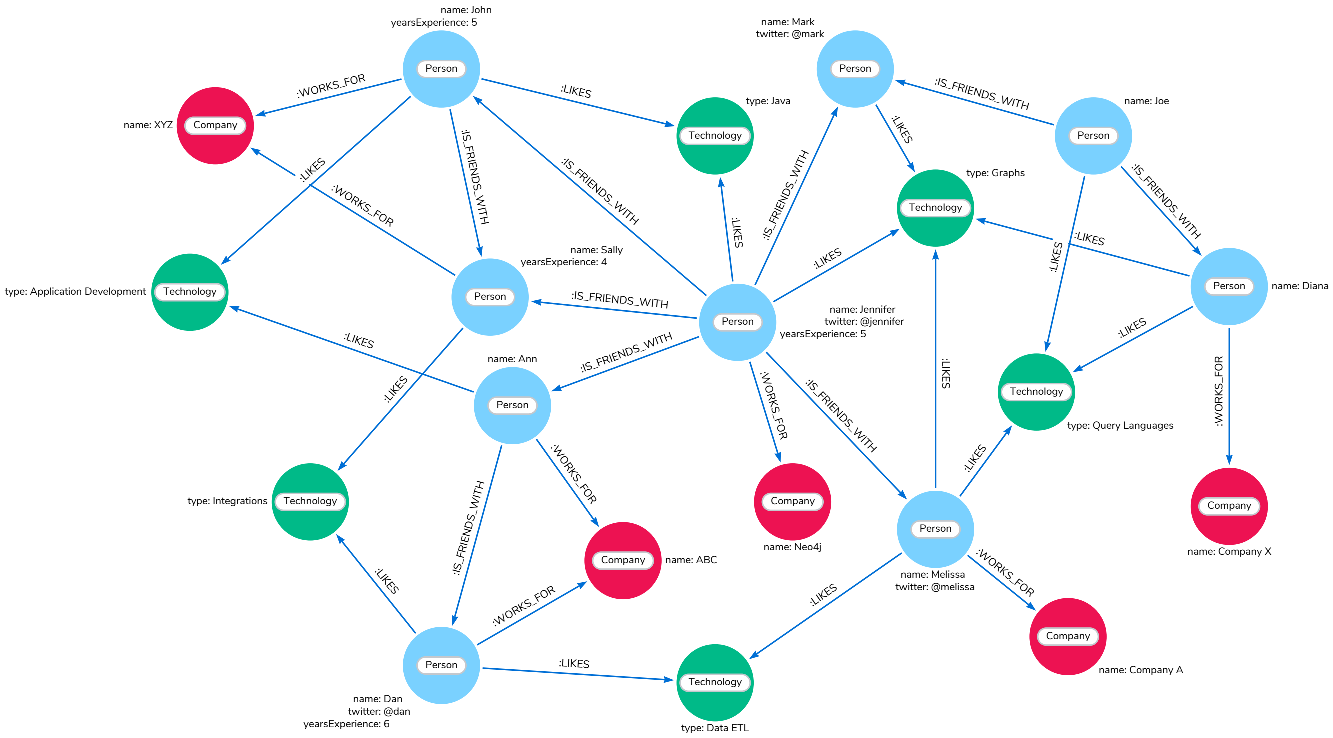
Earlier you learned how to represent nodes, relationships, labels, properties, and patterns in Cypher. This section adds another level to your knowledge by introducing how to update and delete data with Cypher.

While these are the standard CRUD (create, update, and delete) operations, some things function a bit differently in a graph than in other types of databases. You will probably recognize some of the similarities and differences as we go along.

Updating data with Cypher

You may already have a node or relationship in the data, but you want to modify its properties. You can do this by matching the pattern you want to find and using the `SET` keyword to add, remove, or update properties.

We continue to use the following dataset:



Using the above example dataset thus far, you could update Jennifer's node to add her birthdate. The next Cypher statement shows how to do this.

1. First, you need to find the existing node for Jennifer.
2. Next, use `SET` to create the new property (with syntax `variable.property`) and set its value.

3. Finally, you can return Jennifer's node to ensure that the information was updated correctly.

```
MATCH (p:Person {name: 'Jennifer'})
SET p.birthdate = date('1980-01-01')
RETURN p
```

Query result:

```
Set Properties: 1
Rows: 1

+------------------------------------------------------+
| p                                                    |
+------------------------------------------------------+
|(Person: {birthdate: '1980-01-01', name: 'Jennifer'}) |
+------------------------------------------------------+
```

For more information on using `date()` and other temporal functions, visit the reference documentation on temporal values in Cypher.

If you want to change Jennifer's birthdate, you can use the same query above to find Jennifer's node again and put a different date in the `SET` clause.

You are able also update Jennifer's `WORKS_FOR` relationship with the `Company` node to include the year that she started working there. To do this, you can use similar syntax as above for updating nodes.

```
MATCH (:Person {name: 'Jennifer'})-[rel:WORKS_FOR]-(:Company {name: 'Neo4j'})
SET rel.startYear = date({year: 2018})
RETURN rel
```

Query result:

```
Set Properties: 1
Rows: 1

+------------------------------------------+
| rel                                      |
+------------------------------------------+
| [:WORKS_FOR {startYear: '2018-01-01'}]   |
+------------------------------------------+
```

If you want to return a graph view on the above query, you can add variables to the nodes for `p:Person` and `c:Company` and write the return line as `RETURN p, rel, c`.

Deleting data with Cypher

Another operation to cover is how to delete data in Cypher. For this operation, Cypher uses the `DELETE` keyword for deleting nodes and relationships. It is very similar to deleting data in other languages like SQL, with one exception.

Because Neo4j is ACID-compliant, you cannot delete a node if it still has relationships. If you could do that, then you might end up with a relationship pointing to nothing and an incomplete graph.

Delete a relationship

To delete a relationship, you need to find the start and end nodes for the relationship you want to delete and then use the `DELETE` keyword, as shown in the code block below. Let us go ahead and delete the `IS_FRIENDS_WITH` relationship between Jennifer and Mark for now. We will add this relationship back in a later exercise.

```
MATCH (j:Person {name: 'Jennifer'})-[r:IS_FRIENDS_WITH]->(m:Person {name: 'Mark'})
DELETE r
```

Query result:

```
+-----------------------------------------+
| Deleted Relationships: 1                |
| Rows: 0                                 |
+-----------------------------------------+
```

Delete a node

To delete a node that does not have any relationships, you need to find the node you want to delete and then use the `DELETE` keyword, just as you did for the relationship above. You can delete Mark's node for now and bring him back later.

```
MATCH (m:Person {name: 'Mark'})
DELETE m
```

Query result:

```
+-----------------------------------------+
| Deleted Nodes: 1                        |
| Rows: 0                                 |
+-----------------------------------------+
```

If you have created an empty node by mistake and you need to delete it, you can use the following Cypher statement to do it:

```
MATCH (n)
WHERE id(n) = 5
DETACH DELETE n
```

This statement deletes not only a node but also all relationships it has. To run the statement, you should know a node's internal ID.

Delete a node and its relationship

Instead of running the last two queries to delete the `IS_FRIENDS_WITH` relationship and the `Person` node for Mark, you can actually run a single statement to delete the node and its relationship at the same time. As it was mentioned above, Neo4j is ACID-compliant so it doesn't allow to delete a node if it still has relationships. Using the `DETACH DELETE` syntax tells Cypher to delete any relationships the node has, as well as remove the node itself.

The statement would look like the code below. First, you find Mark’s node in the database. Then, the `DETACH DELETE` line removes any existing relationships `Mark` node has before also deleting the node.

```
MATCH (m:Person {name: 'Mark'})
DETACH DELETE m
```

Delete properties

You can also remove properties, but instead of using the `DELETE` keyword, you can use a couple of other approaches.

The first option is to use `REMOVE` on the property. This tells Neo4j that you want to remove the property from the node entirely and no longer store it.

The second option is to use the `SET` keyword from earlier to set the property value to `null`. Unlike other database models, Neo4j does not store null values. Instead, it only stores properties and values that are meaningful to your data. This means that you can have different types and amounts of properties on various nodes and relationships in your graph.

To show you both options, let us look at the code for each.

```
//delete property using REMOVE keyword
MATCH (n:Person {name: 'Jennifer'})
REMOVE n.birthdate

//delete property with SET to null value
MATCH (n:Person {name: 'Jennifer'})
SET n.birthdate = null
```

Query result:

```
+-------------------------------------------+
| Set Properties: 1                         |
| Rows: 0                                   |
+-------------------------------------------+
```

Avoiding duplicate data using *MERGE*

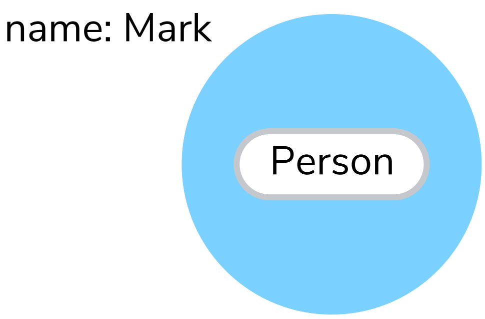
It was briefly mentioned earlier that there are some ways in Cypher to avoid creating duplicate data. One of those ways is using the `MERGE` keyword. `MERGE` does a "select-or-insert" operation that first checks if the data exists in the database. If it exists, then Cypher returns it as is or makes any updates you specify on the existing node or relationship. If the data does not exist, then Cypher will create it with the information you specify.

Using *MERGE* on a node

To start, let us look at an example of this by adding Mark back to our database using the query below. You can use `MERGE` to ensure that Cypher checks the database for an existing node for Mark. Since you removed Mark’s node in the previous examples, Cypher will not find an existing match and will create the node new with the `name` property set to 'Mark'.

```
MERGE (mark:Person {name: 'Mark'})
RETURN mark
```

Query result:



If you run the same statement again, Cypher will find an existing node this time that has the `name` property set to `Mark`, so it will return the matched node without any changes.

Using *MERGE* on a relationship

Just like you use `MERGE` to find or create a node in Cypher, you can do the same thing to find or create a relationship. Let’s re-create the `IS_FRIENDS_WITH` relationship between Mark and Jennifer that we had in a previous example.

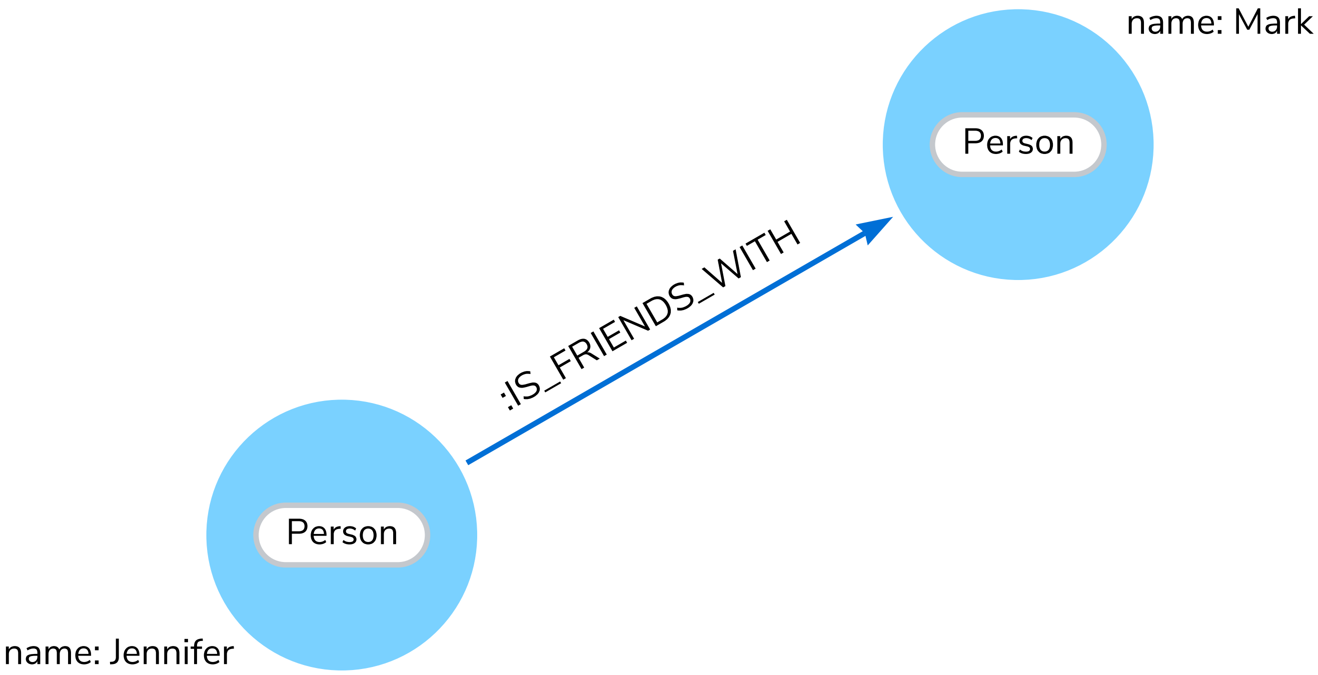
```
MATCH (j:Person {name: 'Jennifer'})
MATCH (m:Person {name: 'Mark'})
MERGE (j)-[r:IS_FRIENDS_WITH]->(m)
RETURN j, r, m
```

Notice that here `MATCH` is used to find both Mark’s node and Jennifer’s node before we use `MERGE` to find or create the relationship between them.

Why do we not use a single statement?

`MERGE` looks for an entire pattern that you specify to see whether to return an existing one or create it new. If the entire pattern (nodes, relationships, and any specified properties) does not exist, Cypher creates it.

Cypher never produces a partial mix of matching and creating within a pattern. To avoid a mix of match and create, you need to match any existing elements of your pattern first before doing a merge on any elements you might want to create, just as we did in the statement above.



Just for reference, the Cypher statement that causes duplicates is below. Since this pattern (Jennifer IS_FRIENDS_WITH Mark) does not exist in the database, Cypher creates the entire pattern new — both nodes, as well as the relationship between them.

```
//this statement will create duplicate nodes for Mark and Jennifer
MERGE (j:Person {name: 'Jennifer'})-[r:IS_FRIENDS_WITH]->(m:Person {name: 'Mark'})
RETURN j, r, m
```

Handling *MERGE* criteria

Perhaps you want to use MERGE to ensure you do not create duplicates, but you want to initialize certain properties if the pattern is created and update other properties if it is only matched. In this case, you can use ON CREATE or ON MATCH with the SET keyword to handle these situations.

Let us look at an example.

```
MERGE (m:Person {name: 'Mark'})-[r:IS_FRIENDS_WITH]-(j:Person {name:'Jennifer'})
  ON CREATE SET r.since = date('2018-03-01')
  ON MATCH SET r.updated = date()
RETURN m, r, j
```

Resources

- Neo4j Cypher Manual: CREATE
- Neo4j Cypher Manual: SET
- Neo4j Cypher Manual: REMOVE
- Neo4j Cypher Manual: DELETE
- Neo4j Cypher Manual: MERGE
- Neo4j Cypher Manual: ON CREATE/ON MATCH

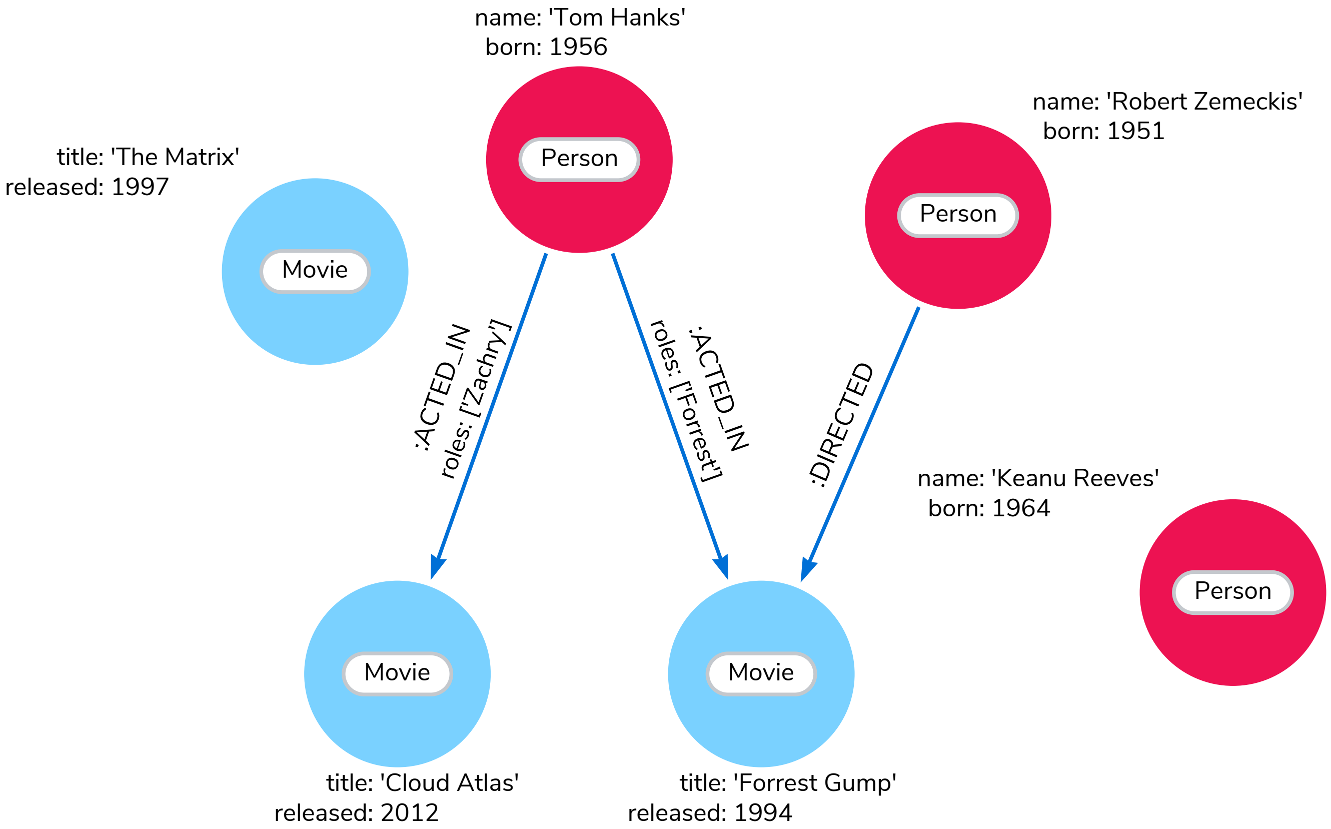
Composing large statements

Example graph

We continue using the same example data as before:

```
CREATE (matrix:Movie {title: 'The Matrix', released: 1997})
CREATE (cloudAtlas:Movie {title: 'Cloud Atlas', released: 2012})
CREATE (forrestGump:Movie {title: 'Forrest Gump', released: 1994})
CREATE (keanu:Person {name: 'Keanu Reeves', born: 1964})
CREATE (robert:Person {name: 'Robert Zemeckis', born: 1951})
CREATE (tom:Person {name: 'Tom Hanks', born: 1956})
CREATE (tom)-[:ACTED_IN {roles: ['Forrest']}]->(forrestGump)
CREATE (tom)-[:ACTED_IN {roles: ['Zachry']}]->(cloudAtlas)
CREATE (robert)-[:DIRECTED]->(forrestGump)
```

This is the resulting graph:



UNION

If you want to combine the results of two statements that have the same result structure, you can use `UNION [ALL]`.

For example, the following statement lists both actors and directors:

```
MATCH (actor:Person)-[r:ACTED_IN]->(movie:Movie)
RETURN actor.name AS name, type(r) AS type, movie.title AS title
UNION
MATCH (director:Person)-[r:DIRECTED]->(movie:Movie)
RETURN director.name AS name, type(r) AS type, movie.title AS title
```

```
Rows: 3

+---------------------------------------------------+
| name              | type       | title          |
+---------------------------------------------------+
| 'Tom Hanks'       | 'ACTED_IN' | 'Cloud Atlas'  |
| 'Tom Hanks'       | 'ACTED_IN' | 'Forrest Gump' |
| 'Robert Zemeckis' | 'DIRECTED' | 'Forrest Gump' |
+---------------------------------------------------+
```

Note that the returned columns must be aliased in the same way in all the sub-clauses.

The query above is equivalent to this more compact query:

```
MATCH (actor:Person)-[r:ACTED_IN|DIRECTED]->(movie:Movie)
RETURN actor.name AS name, type(r) AS type, movie.title AS title
```

WITH

In Cypher, you can chain fragments of statements together, similar to how it is done within a data-flow pipeline. Each fragment works on the output from the previous one, and its results can feed into the next one. *Only* columns declared in the `WITH` clause are available in subsequent query parts.

The `WITH` clause is used to combine the individual parts and declare which data flows from one to the other. `WITH` is similar to the `RETURN` clause. The difference is that the `WITH` clause does not finish the query, but prepares the input for the next part. Expressions, aggregations, ordering, and pagination can be used in the same way as in the `RETURN` clause. The only difference is all columns must be aliased.

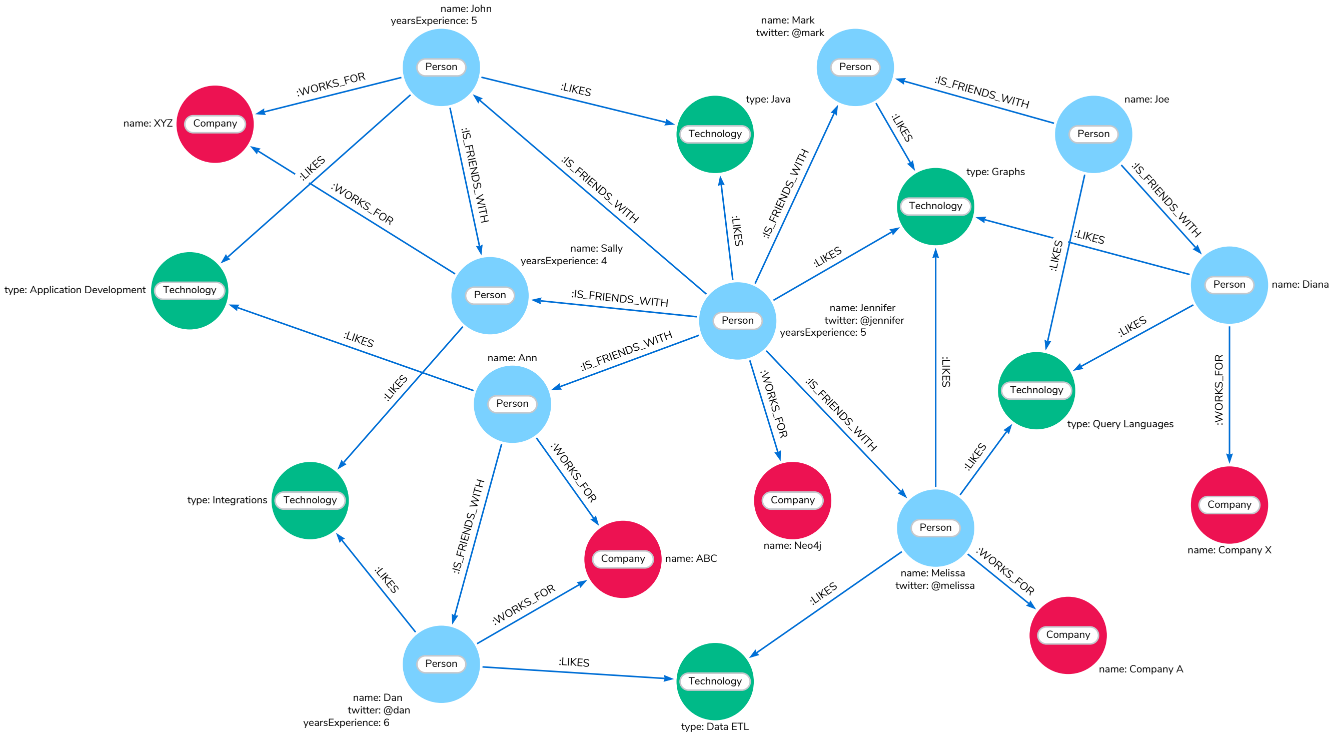
In the example below, collect the movies someone appeared in and then filter out those which appear in only one movie.

```
MATCH (person:Person)-[:ACTED_IN]->(m:Movie)
WITH person, count(*) AS appearances, collect(m.title) AS movies
WHERE appearances > 1
RETURN person.name, appearances, movies
```

```
Rows: 1

+---------------------------------------------------------------+
| person.name | appearances | movies                            |
+---------------------------------------------------------------+
| 'Tom Hanks' | 2           | ['Cloud Atlas', 'Forrest Gump'] |
+---------------------------------------------------------------+
```

Using the `WITH` clause, you can pass values from one section of a query to another. This allows you to perform some intermediate calculations or operations within your query to use later.

The following examples are based on this dataset:



In the WITH clause, specify the variables you want to use later. This functionality can be applied in various ways (for example, count, collect, filter, or limit results). A few examples of this follows, including a cleaner version of the size() query from above (see Counting values in a list).

For more information and ways to use WITH, check out the Cypher Manual section.

```
//Query1: find and list the technologies people like
MATCH (a:Person)-[r:LIKES]-(t:Technology)
WITH a.name AS name, collect(t.type) AS technologies
RETURN name, technologies;
```

Query1 results:

```
Rows: 9

+-------------------------------------------------------------+
| name        | technologies                                  |
+-------------------------------------------------------------+
| 'Sally'     | ['Integrations']                              |
| 'Dan'       | ['Data ETL', 'Integrations']                  |
| 'John'      | ['Java', 'Application Development']           |
| 'Diana'     | ['Query Languages', 'Graphs']                 |
| 'Jennifer'  | ['Java', 'Graphs']                            |
| 'Ann'       | ['Application Development']                   |
| 'Mark'      | ['Graphs']                                    |
| 'Joe'       | ['Query Languages']                            |
| 'Melissa'   | ['Query Languages', 'Data ETL', 'Graphs']     |
+-------------------------------------------------------------+
```

```
//Query2: find number of friends who have other friends
MATCH (p:Person)-[:IS_FRIENDS_WITH]->(friend:Person)
WITH p, collect(friend.name) AS friendsList, size((friend)-[:IS_FRIENDS_WITH]-(:Person)) AS numberOfFoFs
WHERE numberOfFoFs > 1
RETURN p.name, friendsList, numberOfFoFs;
```

Query2 results:

```
Rows: 3

+-------------------------------------------------------------------+
| p.name     | friendList                           | numberOfFoFs   |
+-------------------------------------------------------------------+
| 'Joe'      | ['Mark']                             | 2              |
| 'John'     | ['Sally']                            | 2              |
| 'Jennifer' | ['Sally', 'John', 'Ann', 'Mark']     | 2              |
+-------------------------------------------------------------------+
```

In the first query, the Person name and a collected list of the Technology types are declared. Only those items can be referenced in the RETURN clause. The relationship (r) or the Person twitter cannot be used because those values were not passed along.

In the second query, only p and any of its properties (name, yearsExperience, twitter), the collection of friends (as a whole, not each value), and the number of friend-of-friends can be referenced. Since those values were passed along in the WITH clause, they can be used in the WHERE or RETURN clauses.

WITH requires all values passed to have a variable (if they do not already have one). The Person nodes were given a variable (p) in the MATCH clause, so no variable needs to be assigned there.

WITH is also very helpful for setting up parameters before the query. Often useful for parameter keys, url strings, and other query variables when importing data.

```
//Find people with 2-6 years of experience
WITH 2 AS experienceMin, 6 AS experienceMax
MATCH (p:Person)
WHERE experienceMin <= p.yearsExperience <= experienceMax
RETURN p
```

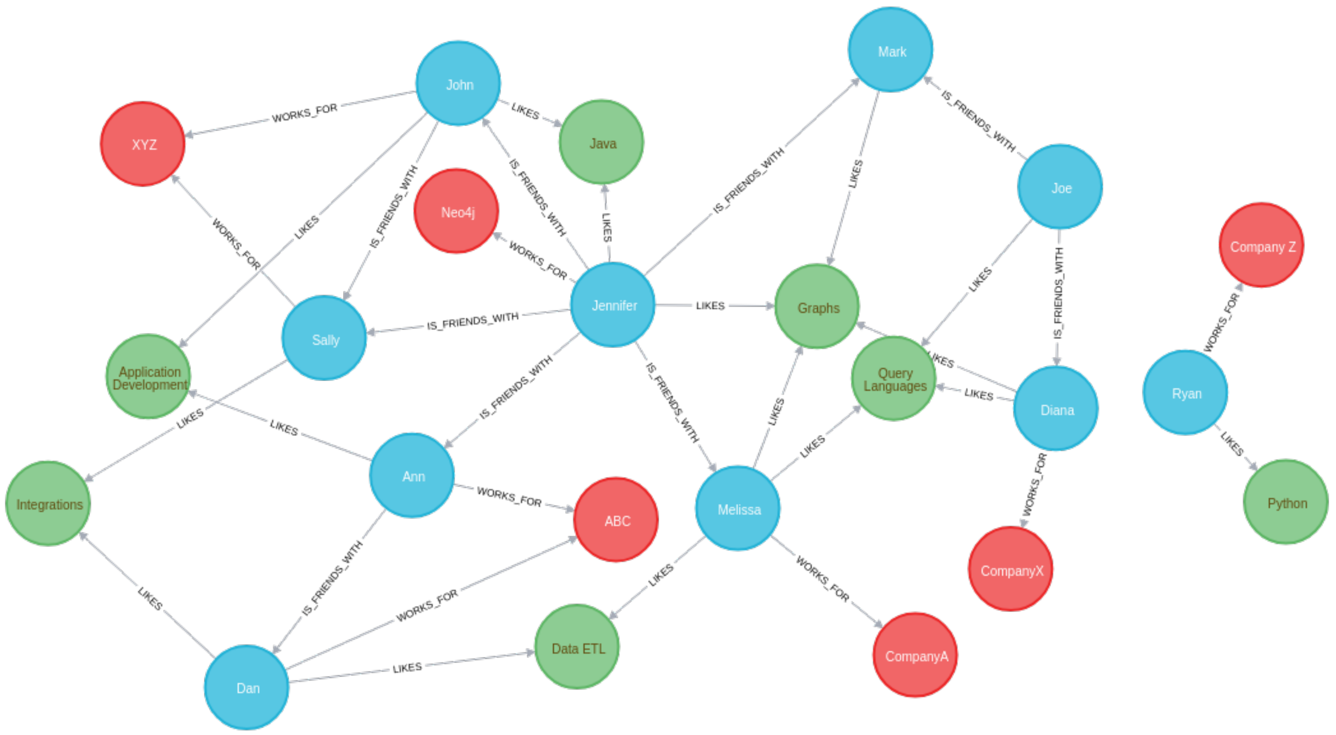
Subqueries

Recap our example graph

All of our examples will continue with the graph example we have been using before, but include some more data for some of our later queries. Below is an image of the new graph.

We have added more Person nodes (blue) who WORK_FOR different Company nodes (red) and LIKE different Technology (green) nodes.

To recap, each person could also have multiple IS_FRIENDS_WITH relationships to other people. This gives us a network of people, the companies they work for, and the technologies they like.



An introduction to subqueries

Neo4j 4.0 introduced support for two different types of subqueries:

- Existential subqueries in a `WHERE` clause
- Result returning subqueries using the `CALL {}` syntax

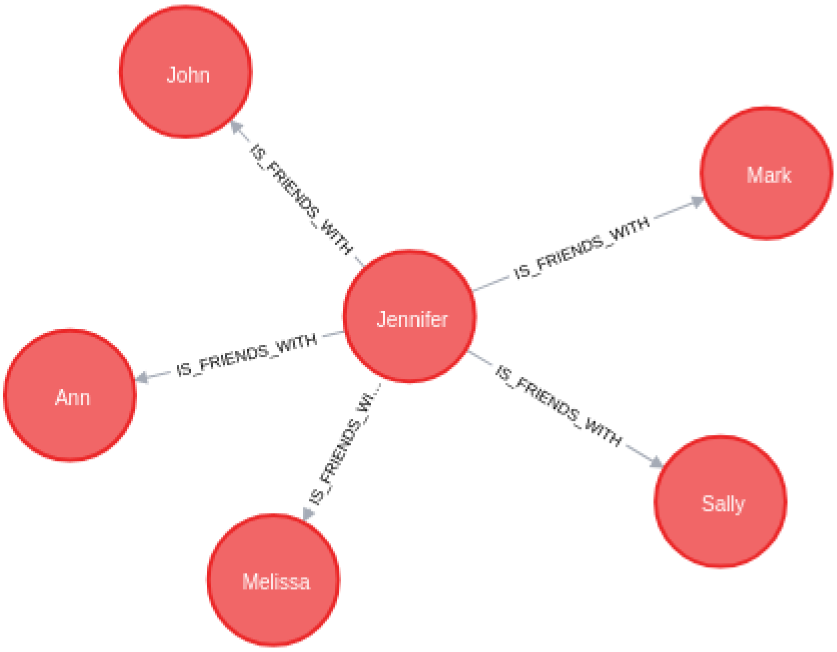
In this section we're going to learn how to write queries that use both these approaches.

Existential subqueries

In the filtering on patterns section of the getting the correct results chapter, we learnt how to filter based on patterns. For example, we wrote the following query to find the friends of someone who works for Neo4j:

```
MATCH (p:Person)-[r:IS_FRIENDS_WITH]->(friend:Person)
WHERE exists((p)-[:WORKS_FOR]->(:Company {name: 'Neo4j'}))
RETURN p, r, friend
```

If we run this query in the Neo4j Browser, the following graph is returned:



Existential subqueries enable more powerful pattern filtering. Instead of using the `exists` function in our `WHERE` clause, we use the `EXISTS {}` clause. We can reproduce the previous example with the following query:

```
MATCH (p:Person)-[r:IS_FRIENDS_WITH]->(friend:Person)
WHERE EXISTS {
  MATCH (p)-[:WORKS_FOR]->(:Company {name: 'Neo4j'})
}
RETURN p, r, friend
```

We'll get the same results, which is nice, but so far all we've achieved is the same thing with more code!

Let's next write a subquery that does more powerful filtering than what we can achieve with the `WHERE` clause or `exists` function alone.

Imagine that we want to find the people who:

- work for a company whose name starts with 'Company' and
- like at least one technology that's liked by three or more people

We aren't interested in knowing what those technologies are. We might try to answer this question with the following query:

```
MATCH (person:Person)-[:WORKS_FOR]->(company)
WHERE company.name STARTS WITH "Company"
AND (person)-[:LIKES]->(t:Technology)
AND size((t)<-[:LIKES]-()) >= 3
RETURN person.name as person, company.name AS company;
```

If we run this query, we'll see the following output:

```
Variable `t` not defined (line 4, column 25 (offset: 112))
"AND (person)-[:LIKES]->(t:Technology)"
                         ^
```

We can find people that like a technology, but we can't check that at least 3 people like that technology as well, because the variable `t` isn't in the scope of the `WHERE` clause. Let's instead move the two `AND` statements into an `EXISTS {}` block, resulting in the following query:

```
MATCH (person:Person)-[:WORKS_FOR]->(company)
WHERE company.name STARTS WITH "Company"
AND EXISTS {
  MATCH (person)-[:LIKES]->(t:Technology)
  WHERE size((t)<-[:LIKES]-()) >= 3
}
RETURN person.name as person, company.name AS company;
```

Now we're able to successfully execute the query, which returns the following results:

person	company
"Melissa"	"CompanyA"
"Diana"	"CompanyX"

If we recall the graph visualisation from the start of this guide, Ryan is the only other person who works for a company whose name starts with "Company". He's been filtered out in this query because the only `Technology` that he likes is Python, and there aren't 3 people who like Python.

Result returning subqueries

So far we've learnt how to use subqueries to filter out results, but this doesn't fully show case their power. We can also use subqueries to return results as well.

Let's say we want to write a query that finds people who like Java or have more than one friend. And we want to return the results ordered by date of birth in descending order. We can get some of the way there using the `UNION` clause:

```
MATCH (p:Person)-[:LIKES]->(:Technology {type: "Java"})
RETURN p.name AS person, p.birthdate AS dob
ORDER BY dob DESC

UNION

MATCH (p:Person)
WHERE size((p)-[:IS_FRIENDS_WITH]->()) > 1
RETURN p.name AS person, p.birthdate AS dob
ORDER BY dob DESC;
```

If we run that query, we'll see the following output:

person	dob
"Jennifer"	1988-01-01
"John"	1985-04-04
"Joe"	1988-08-08

We've got the correct people, but the `UNION` approach only lets us sort results per `UNION` clause, not for all rows.

We can try another approach, where we execute each of our subqueries separately and collect the people from each part using the COLLECT function. There are some people who like Java and have more than one friend, so we'll also need to use a function from the APOC Library to remove those duplicates:

```
// Find people who like Java
MATCH (p:Person)-[:LIKES]->(:Technology {type: "Java"})
WITH collect(p) AS peopleWhoLikeJava

// Find people with more than one friend
MATCH (p:Person)
WHERE size((p)-[:IS_FRIENDS_WITH]->()) > 1
WITH collect(p) AS popularPeople, peopleWhoLikeJava

// Filter duplicate people
WITH apoc.coll.toSet(popularPeople + peopleWhoLikeJava) AS people

// Unpack the collection of people and order by birthdate
UNWIND people AS p
RETURN p.name AS person, p.birthdate AS dob
ORDER BY dob DESC
```

If we run that query, we'll get the following output:

person	dob
"Joe"	1988-08-08
"Jennifer"	1988-01-01
"John"	1985-04-04

This approach works, but it's more difficult to write, and we have to keep passing through parts of state to the next part of the query.

The CALL {} clause gives us the best of both worlds:

- We can use the UNION approach to run the individual queries and remove duplicates
- We can sort the results afterwards

Our query using the CALL {} clause looks like this:

```
CALL {
    MATCH (p:Person)-[:LIKES]->(:Technology {type: "Java"})
    RETURN p

    UNION

    MATCH (p:Person)
    WHERE size((p)-[:IS_FRIENDS_WITH]->()) > 1
    RETURN p
}
RETURN p.name AS person, p.birthdate AS dob
ORDER BY dob DESC;
```

If we run that query, we'll get the following output:

person	dob
"Joe"	1988-08-08

person	dob
"Jennifer"	1988-01-01
"John"	1985-04-04

We could extend our query further to return the technologies that these people like, and the friends that they have. The following query shows how to do this:

```
CALL {
    MATCH (p:Person)-[:LIKES]->(:Technology {type: "Java"})
    RETURN p

    UNION

    MATCH (p:Person)
    WHERE size((p)-[:IS_FRIENDS_WITH]->()) > 1
    RETURN p
}
WITH p,
     [(p)-[:LIKES]->(t) | t.type] AS technologies,
     [(p)-[:IS_FRIENDS_WITH]->(f) | f.name] AS friends

RETURN p.name AS person, p.birthdate AS dob, technologies, friends
ORDER BY dob DESC;
```

person	dob	technologies	friends
"Joe"	1988-08-08	["Query Languages"]	["Mark", "Diana"]
"Jennifer"	1988-01-01	["Graphs", "Java"]	["Sally", "Mark", "John", "Ann", "Melissa"]
"John"	1985-04-04	["Java", "Application Development"]	["Sally"]

We can also apply aggregation functions to the results of our subquery. The following query returns the youngest and oldest of the people who like Java or have more than one friend

```
CALL {
    MATCH (p:Person)-[:LIKES]->(:Technology {type: "Java"})
    RETURN p

    UNION

    MATCH (p:Person)
    WHERE size((p)-[:IS_FRIENDS_WITH]->()) > 1
    RETURN p
}
RETURN min(p.birthdate) AS oldest, max(p.birthdate) AS youngest
```

oldest	youngest
1985-04-04	1988-08-08

Next steps

We have seen how to use the `EXISTS {}` clause to write complex filtering patterns, and the `CALL {}` clause to execute result returning subqueries. In the next section, we will learn how to use aggregation in Cypher and how to do more with the return results.

Resources

- Neo4j Cypher Manual: Using existential subqueries in WHERE
- Neo4j Cypher Manual: CALL {} (subquery)

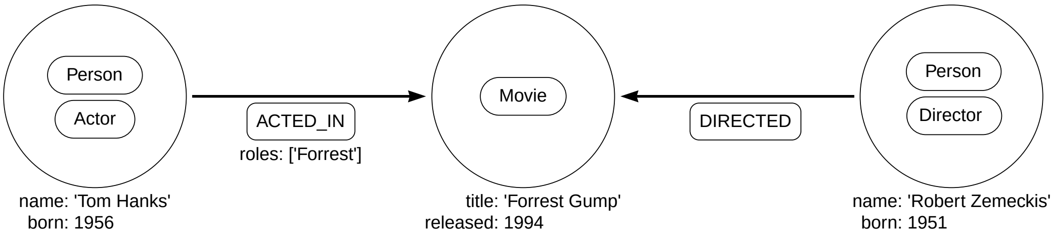
Defining a schema

Example graph

First create some data to use for our examples:

```
CREATE (forrestGump:Movie {title: 'Forrest Gump', released: 1994})
CREATE (robert:Person:Director {name: 'Robert Zemeckis', born: 1951})
CREATE (tom:Person:Actor {name: 'Tom Hanks', born: 1956})
CREATE (tom)-[:ACTED_IN {roles: ['Forrest']}]->(forrestGump)
CREATE (robert)-[:DIRECTED]->(forrestGump)
```

This is the resulting graph:



Using indexes

The main reason for using indexes in a graph database is to find the starting point of a graph traversal. Once that starting point is found, the traversal relies on in-graph structures to achieve high performance.

Indexes can be added at any time.

> If there is existing data in the database, it will take some time for an index to come online.

The following query creates an index to speed up finding actors by name in the database:

```
CREATE INDEX example_index_1 FOR (a:Actor) ON (a.name)
```

In most cases it is not necessary to specify indexes when querying for data, as the appropriate indexes will be used automatically.

> It is possible to specify which index to use in a particular query, using *index hints*. This is one of several options for query tuning, described in detail in Cypher manual → Query tuning.

For example, the following query will automatically use the `example_index_1`:

```
MATCH (actor:Actor {name: 'Tom Hanks'})
RETURN actor
```

A *composite index* is an index on multiple properties for all nodes that have a particular label. For example, the following statement will create a composite index on all nodes labeled with `Actor` and which have both a `name` and a `born` property. Note that since the node with the `Actor` label that has a `name` of "Keanu Reeves" does not have the `born` property. Therefore that node will not be added to the index.

```
CREATE INDEX example_index_2 FOR (a:Actor) ON (a.name, a.born)
```

You can query a database with `SHOW INDEXES` to find out what indexes are defined.

```
SHOW INDEXES YIELD name, labelsOrTypes, properties, type
```

```
Rows: 2

+-------------------------------------------------------------------+
| name              | labelsOrTypes | properties       | type     |
+-------------------------------------------------------------------+
| 'example_index_1' | ['Actor']     | ['name']         | 'BTREE'  |
| 'example_index_2' | ['Actor']     | ['name', 'born'] | 'BTREE'  |
+-------------------------------------------------------------------+
```

Learn more about indexes in Cypher Manual → Indexes.

Using constraints

Constraints are used to make sure that the data adheres to the rules of the domain. For example:

> "If a node has a label of `Actor` and a property of `name`, then the value of `name` must be unique among all nodes that have the `Actor` label".

Example 1. Uniqueness constraint

This example shows how to create a constraint for nodes that have the label `Movie` and the property `title`. The constraint specifies that the `title` property must be unique.

Adding the unique constraint will implicitly add an index on that property. If the constraint is dropped, but the index is still needed, the index will have to be created explicitly.

```
CREATE CONSTRAINT constraint_example_1 FOR (movie:Movie) REQUIRE movie.title IS UNIQUE
```

The syntax was changed in Neo4j 4.4, the old syntax is:

```
CREATE CONSTRAINT constraint_example_1 ON (movie:Movie) ASSERT movie.title IS
UNIQUE <button>Deprecated</button>
```

Constraints can be added to database that already has data in it. This requires that the existing data complies with the constraint that is being added.

You can query a database to find out what constraints are defined with the `SHOW CONSTRAINTS` Cypher syntax.

Example 2. Constraints query

This example shows a Cypher query that returns the constraints that has been defined for the database.

```
SHOW CONSTRAINTS YIELD id, name, type, entityType, labelsOrTypes, properties, ownedIndexId
```

```
Rows: 1

+----------------------------------------------------------------------------------------------------
-+
| id | name                   | type         | entityType | labelsOrTypes | properties | ownedIndexId
|
+----------------------------------------------------------------------------------------------------
-+
| 4  | 'constraint_example_1' | 'UNIQUENESS' | 'NODE'     | ['Movie']     | ['title']  | 3
|
+----------------------------------------------------------------------------------------------------
-+
```

The constraint described above is available for all editions of Neo4j. Additional constraints are available for Neo4j Enterprise Edition.

Learn more about constraints in Cypher manual → Constraints.

Dates, datetimes, and durations

Creating and updating values

Let's start by creating some nodes that have a Datetime property. We can do this by executing the following Cypher query:

```
UNWIND [
    { title: "Cypher Basics I",
      created: datetime("2019-06-01T18:40:32.142+0100"),
      datePublished: date("2019-06-01"),
      readingTime: {minutes: 2, seconds: 15} },
    { title: "Cypher Basics II",
      created: datetime("2019-06-02T10:23:32.122+0100"),
      datePublished: date("2019-06-02"),
      readingTime: {minutes: 2, seconds: 30} },
    { title: "Dates, Datetimes, and Durations in Neo4j",
      created: datetime(),
      datePublished: date(),
      readingTime: {minutes: 3, seconds: 30} }
] AS articleProperties

CREATE (article:Article {title: articleProperties.title})
SET article.created = articleProperties.created,
    article.datePublished = articleProperties.datePublished,
    article.readingTime = duration(articleProperties.readingTime)
```

In this query:

- the `created` property is a `DateTime` type equal to the datetime at the time the query is executed.
- the `date` property is a `Date` type equal to the date at the time the query is executed.
- the `readingTime` is a `Duration` type of 3 minutes 30 seconds.

Maybe we want to make some changes to this article node to update the `datePublished` and `readingTime` properties.

We've decided to publish the article next week rather than today, so we want to make that change. If we want to create a new `Date` type using a supported format, we could do so using the following query:

```
MATCH (article:Article {title: "Dates, Datetimes, and Durations in Neo4j"})
SET article.datePublished = date("2019-09-30")
```

But what if we want to create a `Date` type based on an unsupported format? To do this we'll use a function from the APOC library to parse the string.

The following query parses an unsupported data format into a millisecond based timestamp, creates a `Datetime` from that timestamp, and then creates a `Date` from that `Datetime`:

```
WITH apoc.date.parse("Sun, 29 September 2019", "ms", "EEE, dd MMMM yyyy") AS ms
MATCH (article:Article {title: "Dates, Datetimes, and Durations in Neo4j"})
SET article.datePublished = date(datetime({epochmillis: ms}))
```

We could use this same approach to update the `created` property. The only thing we need to change is that we don't need to convert the `Datetime` type to a `Date`:

```
WITH apoc.date.parse("25 September 2019 06:29:39", "ms", "dd MMMM yyyy HH:mm:ss") AS ms
MATCH (article:Article {title: "Dates, Datetimes, and Durations in Neo4j"})
SET article.created = datetime({epochmillis: ms})
```

Perhaps we also decide that the reading time is actually going to be one minute more than what we originally thought. We can update the `readingTime` property with the following query:

```
MATCH (article:Article {title: "Dates, Datetimes, and Durations in Neo4j"})
SET article.readingTime =  article.readingTime + duration({minutes: 1})
```

Formatting values

Now we want to write a query to return our article. We can do this by executing the following query:

```
MATCH (article:Article)
RETURN article.title AS title,
       article.created AS created,
       article.datePublished AS datePublished,
       article.readingTime AS readingTime
```

Table 1. Results

title	created	datePublished	readingTime
"Dates, Datetimes, and Durations in Neo4j"	2019-09-25T06:29:39Z	2019-09-29	P0M0DT270S

If we want to format these values we can use temporal functions in the APOC library. The following query formats each of the temporal types into more friendly formats:

```
MATCH (article:Article)
RETURN article.title AS title,
       apoc.temporal.format(article.created, "dd MMMM yyyy HH:mm") AS created,
       apoc.temporal.format(article.datePublished,"dd MMMM yyyy") AS datePublished,
       apoc.temporal.format(article.readingTime, "mm:ss") AS readingTime
```

Table 2. Results

title	created	datePublished	readingTime
"Dates, Datetimes, and Durations in Neo4j"	"25 September 2019 06:29"	"29 September 2019"	"04:30"

Comparing and filtering values

What if we want to filter our articles based on these temporal values.

Let’s start by finding the articles that were published on 1st June 2019. The following query does this:

```
MATCH (article:Article)
WHERE article.datePublished = date({year: 2019, month: 6, day: 1})
RETURN article.title AS title,
       article.created AS created,
       article.datePublished AS datePublished,
       article.readingTime AS readingTime
```

Table 3. Results

title	created	datePublished	readingTime
"Cypher Basics I"	2019-06-01T18:40:32.142+01:00	2019-06-01	P0M0DT135S

What about if we want to find all the articles published in June 2019? We might write the following query to do this:

```
MATCH (article:Article)
WHERE article.datePublished = date({year: 2019, month: 6})
RETURN article.title AS title,
       article.created AS created,
       article.datePublished AS datePublished,
       article.readingTime AS readingTime
```

If we run this query we'll get the following results:

Table 4. Results

title	created	datePublished	readingTime
"Cypher Basics I"	2019-06-01T18:40:32.142+01:00	2019-06-01	P0M0DT135S

This doesn't seem right - what about the `Cypher Basics II` article that was published on 2nd June 2019? The problem we have here is that `date({year: 2019, month:6})` returns `2019-06-01`, so we're only finding articles published on 1st June 2019.

We need to tweak our query to find articles published between June 1st 2019 and July 1st 2019. The following query does this:

```
MATCH (article:Article)
WHERE date({year: 2019, month: 7}) > article.datePublished >= date({year: 2019, month: 6})
RETURN article.title AS title,
       article.created AS created,
       article.datePublished AS datePublished,
       article.readingTime AS readingTime
```

Table 5. Results

title	created	datePublished	readingTime
"Cypher Basics I"	2019-06-01T18:40:32.142+01:00	2019-06-01	P0M0DT135S
"Cypher Basics II"	2019-06-02T10:23:32.122+01:00	2019-06-02	P0M0DT150S

What about if we want to filter based on the `created` property, which stores `Datetime` values? We need to take the same approach when filtering `Datetime` values as we did with `Date` values. The following query finds the articles created after July 2019:

```
MATCH (article:Article)
WHERE article.created > datetime({year: 2019, month: 7})
RETURN article.title AS title,
       article.created AS created,
       article.datePublished AS datePublished,
       article.readingTime AS readingTime
```

Table 6. Results

title	created	datePublished	readingTime
"Dates, Datetimes, and Durations in Neo4j"	2019-09-25T06:04:39.072Z	2019-09-25	P0M0DT210S

And finally filtering durations. We might be interested in finding articles that can be read in 3 minutes or less.

We'll start with the following query:

```
MATCH (article:Article)
WHERE article.readingTime <= duration("PT3M")
RETURN article.title AS title,
       article.created AS created,
       article.datePublished AS datePublished,
       article.readingTime AS readingTime
```

However, that query results in the following output: *no changes, no records*.

If we want to compare durations we need to do that comparison by adding those durations to dates. We don't really care about dates for our query so we'll just use the current time to work around this issue. We can get the current time by calling the datetime() function.

Our updated query reads like this:

```
MATCH (article:Article)
WHERE datetime() + article.readingTime <= datetime() + duration("PT3M")
RETURN article.title AS title,
       article.created AS created,
       article.datePublished AS datePublished,
       article.readingTime AS readingTime
```

Table 7. Results

title	created	datePublished	readingTime
"Cypher Basics I"	"01 June 2019 18:40"	"01 June 2019"	"02:15"
"Cypher Basics II"	"02 June 2019 10:23"	"02 June 2019"	"02:30"

Resources

This section has shown how to work more effectively with temporal types using the APOC libary. Below are some resources for learning more about using Temporal types in Neo4j:

- Temporal data types
- Knowledge Base: Converting strings to dates

- APOC Library
    - Date parsing
    - Temporal Functions
- Developer Blog: Cypher Sleuthing: Dealing with Dates, Part 1

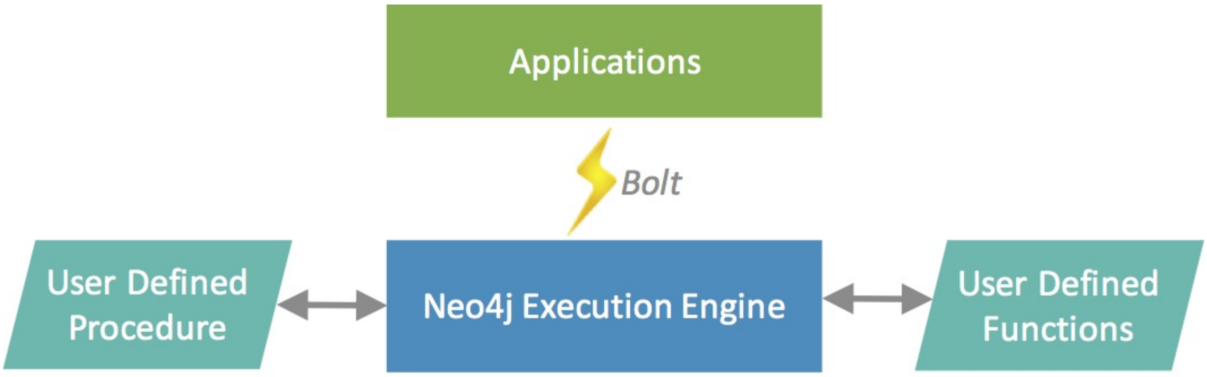
User defined procedures and functions

> *This guide explains how to use, create and deploy user defined procedures and functions, the extension mechanism of Cypher, Neo4j's query language. It also covers existing, widely used procedure libraries*

Extending Cypher

Cypher is a quite powerful and expressive language, with first class graph pattern and collection support. But sometimes you need to do more than it currently offers, like additional graph algorithms, parallelization or custom conversions.

That's why Neo4j and Cypher can be extended with *User defined procedures and functions*. Neo4j itself provides and utilizes custom procedures. Many of the monitoring, introspection and security features exposed by Neo4j Browser are implemented using procedures.



What are procedures and functions?

- Functions are simple computations / conversions and return a single value.
- Functions can be used in any expression or predicate.
- Procedures are more complex operations and generate streams of results.
- Procedures must be used within the `CALL` clause and `YIELD` their result columns.
- They can generate, fetch or compute data to make it available to later processing steps in your Cypher query.

Listing & using functions and procedures

There are a number of built in procedures, two of which are used to list available functions and procedures.

Run the following statements along to get a hang of the usage and see their results.

```
CALL dbms.procedures()
```

Call `dbms.functions()` to list functions.

Each procedure returns one or more columns of data. With `yield` these columns can be selected and also aliased and are then available in your Cypher statement.

```
CALL dbms.procedures()
YIELD name, signature, description as text
WHERE name STARTS WITH 'db.'
RETURN * ORDER BY name ASC
```

Of course you can also process the result columns with other Cypher clauses. Here we group them by package.

```
CALL dbms.procedures()
YIELD name, signature, description
WITH split(name,".") AS parts
RETURN parts[0..-1] AS package,  count(*) AS count,
       collect(parts[-1]) AS names
ORDER BY count DESC
```

package	count	names
["dbms","security"]	16	["activateUser","addRoleToUser","changePassword",....]
["apoc","refactor"]	11	["categorize","cloneNodes","from"....]
["apoc","load"]	9	["csv","driver","jdbc","jdbcParams","json","jsoe"]
["db"]	9	["awaitIndex","constraints","indexes","labels",...,"schema"]
["dbms"]	9	["components","functions","queries","procedures",...]

As of Neo4j 3.1, all functions available are directly part of the Cypher implementation, so User Defined Functions would only come from installed libraries.

You can take any procedure library and deploy it to your server to make additional procedures and functions available.

Also take a look at the procedure section in the Neo4j Manual.

Deploying procedures & functions

If you built your own procedures or downloaded them from a community project, they are packaged in a jar-file. You can copy that file into the `$NEO4J_HOME/plugins` directory of your Neo4j server and restart.

> ⚠ **A word of caution.** As procedures and functions use the low level Java API they can access all Neo4j internals as well as the file system and machine. That’s why you should know which procedures you deploy and why. Only install procedures from trusted sources. If they are open source, check their source-code and best build them yourself.

> ❗ Certain procedures and functions are available for self-managed Neo4j Enterprise Edition and Community Edition. Custom code described in this section is not compatible with AuraDB

Procedure and function gallery

In our Neo4j Labs projects, we provide an impressive set of libraries built by our community and staff. Check it out to see what’s already there. Many of your needs will already be covered by those, for example:

- index operations
- database/api integration
- graph refactorings
- import and export
- spatial index lookup
- rdf import and export
- and many more

Here are two cool examples of what you can do:

A procedure to load data from another database:

```
WITH "jdbc:mysql://localhost:3306/northwind?user=root" as url
CALL apoc.load.jdbc(url,"products") YIELD row
RETURN row
ORDER BY row.UnitPrice DESC
LIMIT 20
```

[apoc load jdbc] | //raw.githubusercontent.com/neo4j-contrib/neo4j-apoc-procedures/3.2/docs/img/apoc-

load-jdbc.jpg

Functions to format and parse timestamps of different resolutions:

```
RETURN apoc.date.format(timestamp()) as time,
       apoc.date.format(timestamp(),'ms','yyyy-MM-dd') as date,
       apoc.date.parse('13.01.1975','s','dd.MM.yyyy') as unixtime,
       apoc.date.parse('2017-01-05 13:03:07') as millis
```

time	date	unixtime	millis
"2017-01-05 13:06:39"	"2017-01-05"	158803200	1483621387000

Developing your own procedures and functions

You can find details on writing and testing procedures in the Neo4j Manual. The example GitHub repository contains detailed documentation and comments that you can clone directly and use as a starting point.

Here are just some initial tips:

User-defined functions are simpler, so let's start with them:

- `@UserFunction` are annotated, public Java methods in a class
- their default name is package-name.method-name
- they return a single value
- are read only

User defined procedures are similar:

- `@Procedure` annotated, Java methods
- with an additional `mode` attribute (`READ, WRITE, DBMS`)
- return a Java 8 `Stream` of simple objects with `public` fields
- these fields names are turned into result columns available for `YIELD`

These things are valid for both:

- take `@Name` annotated parameters (with optional default values)
- can use an injected `@Context public GraphDatabaseService`
- run within transaction of the Cypher statement
- supported types for parameters and results are: `Long, Double, Boolean, String, Node, Relationship, Path, Object`

Tutorial: Import data

> *This tutorial demonstrates how to import data from CSV files using `LOAD CSV`.*

With the combination of the Cypher clauses `LOAD CSV`, `MERGE`, and `CREATE` you can conveniently import data into Neo4j. `LOAD CSV` allows you to access the data values and perform actions on them.

- For a full description of `LOAD CSV` , see Cypher Manual → `LOAD CSV`.
- For a full list of Cypher clauses, see Cypher Manual → Clauses.

The data files

In this tutorial, you import data from the following CSV files:

- *persons.csv*
- *movies.csv*
- *roles.csv*

The content of the *persons.csv* file:

persons.csv

```
id,name
1,Charlie Sheen
2,Michael Douglas
3,Martin Sheen
4,Morgan Freeman
```

The persons.csv file contains two columns `id` and `name`. Each row represents one person that has a unique `id` and a `name`.

The content of the *movies.csv* file:

movies.csv

```
id,title,country,year
1,Wall Street,USA,1987
2,The American President,USA,1995
3,The Shawshank Redemption,USA,1994
```

The *movies.csv* file contains the columns `id`, `title`, `country`, and `year`. Each row represents one movie that has a unique `id`, a `title`, a `country` of origin, and a release `year`.

The content of the *roles.csv* file:

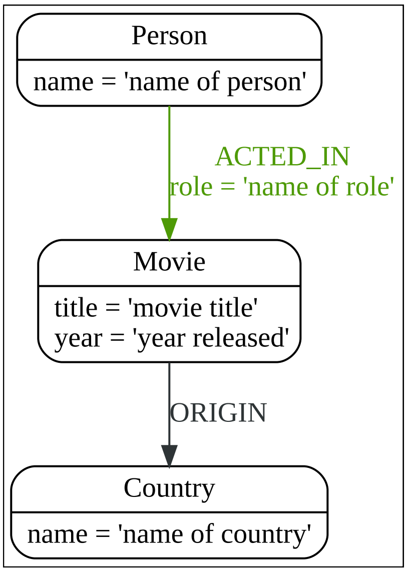
roles.csv

```
personId,movieId,role
1,1,Bud Fox
4,1,Carl Fox
3,1,Gordon Gekko
4,2,A.J. MacInerney
3,2,President Andrew Shepherd
5,3,Ellis Boyd 'Red' Redding
```

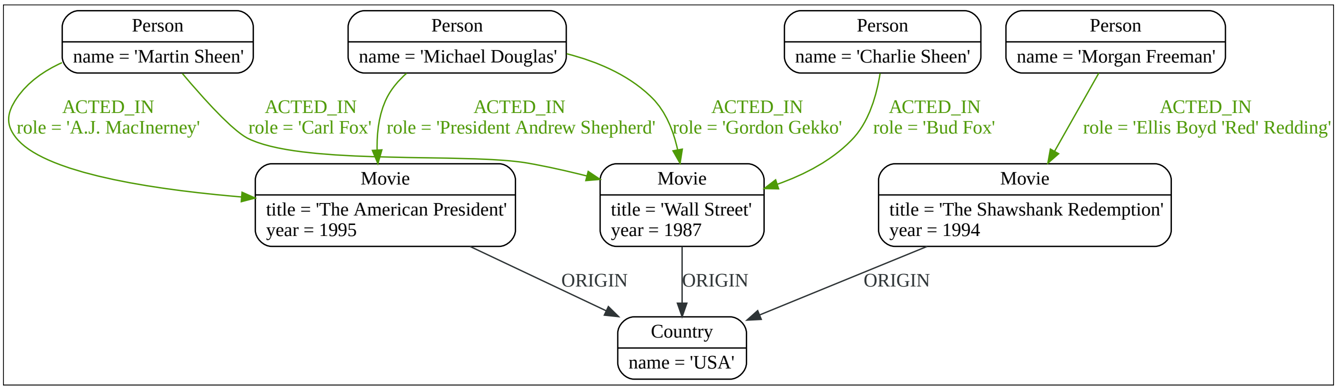
The *roles.csv* file contains the columns `personId`, `movieId`, and `role`. Each row represents one role with relationship data about the person `id` (from the *persons.csv* file) and the movie `id` (from the *movies.csv* file).

The graph model

The following simple data model shows what a graph model for this data set could look like:



This is the resulting graph, based on the data from the CSV files:



Prerequisites

This tutorial uses the Linux or macOS tarball installation.

It assumes that your current work directory is the *<neo4j-home>* directory of the tarball installation, and the CSV files are placed in the default *import* directory.

- For the default directory of other installations see, Operations Manual → File locations.
- The import location can be configured with Operations Manual → `dbms.directories.import`.

Prepare the database

Before importing the data, you should prepare the database you want to use by creating indexes and constraints.

You should ensure that the `Person` and `Movie` nodes have unique `id` properties by creating constraints on them.

Creating a unique constraint also implicitly creates an index. By indexing the `id` property, node lookup (e.g. by `MATCH`) will be much faster.

Additionally, it is a good idea to index the country `name` for a fast lookup.

1. Start neo4j.

Run the command:

```
bin/neo4j start
```

ℹ The default user name is `neo4j` and password `neo4j`.

2. Create a constraint so that each `Person` node has a unique `id` property.

You create a constraint on the `id` property of `Person` nodes to ensure that nodes with the `Person` label will have a unique `id` property.

Using *Neo4j Browser*, run the following Cypher:

```
CREATE CONSTRAINT personIdConstraint FOR (person:Person) REQUIRE person.id IS UNIQUE
```

Or using *Neo4j Cypher Shell*, run the command:

```
bin/cypher-shell --database=neo4j "CREATE CONSTRAINT personIdConstraint FOR (person:Person) REQUIRE
person.id IS UNIQUE"
```

3. Create a constraint so that each `Movie` node has a unique `id` propery.

You create a constraint on the `id` property of `Movie` nodes to ensure that nodes with the `Movie` label will have a unique `id` property.

Using *Neo4j Browser*, run the following Cypher:

```
CREATE CONSTRAINT movieIdConstraint FOR (movie:Movie) REQUIRE movie.id IS UNIQUE
```

Or using *Neo4j Cypher Shell*, run the command:

```
bin/cypher-shell --database=neo4j "CREATE CONSTRAINT movieIdConstraint FOR (movie:Movie) REQUIRE movie.id
IS UNIQUE"
```

4. Create an index for `Country` node for the `name` property.

Create an index on the `name` property of `Country` nodes to ensure fast lookups.

When using `MERGE` or `MATCH` with `LOAD CSV`, make sure you have an index or a unique constraint on the property that you are merging on. This will ensure that the query executes in a performant way.

Using Neo4j Browser, run the following Cypher:

```
CREATE INDEX FOR (c:Country) ON (c.name)
```

Or using Neo4j Cypher Shell, run the command:

```
bin/cypher-shell --database=neo4j "CREATE INDEX FOR (c:Country) ON (c.name)"
```

Import data using `LOAD CSV`

1. Load the data from the *persons.csv* file.

You create nodes with the `Person` label and the properties `id` and `name`.

Using Neo4j Browser, run the following Cypher:

```
LOAD CSV WITH HEADERS FROM "file:///persons.csv" AS csvLine
CREATE (p:Person {id: toInteger(csvLine.id), name: csvLine.name})
```

Or using Neo4j Cypher Shell, run the command:

```
bin/cypher-shell --database=neo4j 'LOAD CSV WITH HEADERS FROM "file:///persons.csv" AS csvLine CREATE
(p:Person {id:toInteger(csvLine.id), name:csvLine.name})'
```

Output:

```
Added 4 nodes, Set 8 properties, Added 4 labels
```

`LOAD CSV` also supports accessing CSV files via `HTTPS`, `HTTP`, and `FTP`, see Cypher Manual → `LOAD CSV`.

2. Load the data from the *movies.csv* file.

You create nodes with the `Movie` label and the properties `id`, `title`, and `year`.

Also you create nodes with the `Country` label. Using `MERGE` avoids creating duplicate `Country` nodes in the case where multiple movies have the same country of origin.

The relationship with the type `ORIGIN` will connect the `Country` node and the `Movie` node.

Using Neo4j Browser, run the following Cypher:

```
LOAD CSV WITH HEADERS FROM "file:///movies.csv" AS csvLine
MERGE (country:Country {name: csvLine.country})
CREATE (movie:Movie {id: toInteger(csvLine.id), title: csvLine.title, year:toInteger(csvLine.year)})
CREATE (movie)-[:ORIGIN]->(country)
```

Or using Neo4j Cypher Shell, run the command:

```
bin/cypher-shell --database=neo4j 'LOAD CSV WITH HEADERS FROM "file:///movies.csv" AS csvLine MERGE
(country:Country {name:csvLine.country}) CREATE (movie:Movie {id:toInteger(csvLine.id),
title:csvLine.title, year:toInteger(csvLine.year)}) CREATE (movie)-[:ORIGIN]->(country)'
```

Output:

```
Added 4 nodes, Created 3 relationships, Set 10 properties, Added 4 labels
```

3. Load the data from the *roles*.csv file

Importing the data from the *roles*.csv file is a matter of finding the `Person` node and `Movie` node and then creating relationships between them.

> ℹ For larger data files, it is useful to use the hint `USING PERIODIC COMMIT` clause of `LOAD CSV`. This hint tells Neo4j that the query might build up inordinate amounts of transaction state, and thus needs to be periodically committed. For more information, see https://neo4j.com/docs/pdf/neo4j-cypher-manual-4.4.pdf#query-using-periodic-commit-hint.

Using *Neo4j Browser*, run the following Cypher:

```
USING PERIODIC COMMIT 500
LOAD CSV WITH HEADERS FROM "file:///roles.csv" AS csvLine
MATCH (person:Person {id: toInteger(csvLine.personId)}), (movie:Movie {id: toInteger(csvLine.movieId)})
CREATE (person)-[:ACTED_IN {role: csvLine.role}]->(movie)
```

Or using Neo4j Cypher Shell, run the command:

```
bin/cypher-shell --database=neo4j 'USING PERIODIC COMMIT 500 LOAD CSV WITH HEADERS FROM
"file:///roles.csv" AS csvLine MATCH (person:Person {id:toInteger(csvLine.personId)}), (movie:Movie
{id:toInteger(csvLine.movieId)}) CREATE (person)-[:ACTED_IND {role:csvLine.role}]->(movie)'
```

Output:

```
Created 5 relationships, Set 5 properties
```

Validate the imported data

Check the resulting data set by finding all the nodes that have a relationship.

Using *Neo4j Browser*, run the following Cypher:

```
MATCH (n)-[r]->(m) RETURN n, r, m
```

Or using Neo4j Cypher Shell, run the command:

```
bin/cypher-shell --database=neo4j 'MATCH (n)-[r]->(m) RETURN n, r, m'
```

Output:

```
+---------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------+
| n                                                               | r
| m                                                             |
+---------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------+
| (:Movie {id: 3, title: "The Shawshank Redemption", year: 1994}) | [:ORIGIN]
| (:Country {name: "USA"})                                      |
| (:Movie {id: 2, title: "The American President", year: 1995})   | [:ORIGIN]
| (:Country {name: "USA"})                                      |
| (:Movie {id: 1, title: "Wall Street", year: 1987})              | [:ORIGIN]
| (:Country {name: "USA"})                                      |
| (:Person {name: "Morgan Freeman", id: 4})                       | [:ACTED_IN {role: "Carl Fox"}]
| (:Movie {id: 1, title: "Wall Street", year: 1987})            |
| (:Person {name: "Charlie Sheen", id: 1})                        | [:ACTED_IN {role: "Bud Fox"}]
| (:Movie {id: 1, title: "Wall Street", year: 1987})            |
| (:Person {name: "Martin Sheen", id: 3})                         | [:ACTED_IN {role: "Gordon Gekko"}]
| (:Movie {id: 1, title: "Wall Street", year: 1987})            |
| (:Person {name: "Martin Sheen", id: 3})                         | [:ACTED_IN {role: "President Andrew
Shepherd"}] | (:Movie {id: 2, title: "The American President", year: 1995}) |
| (:Person {name: "Morgan Freeman", id: 4})                       | [:ACTED_IN {role: "A.J. MacInerney"}]
| (:Movie {id: 2, title: "The American President", year: 1995}) |
+---------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------+
```

Cypher resources

Cypher resources

To help you along your path of learning more about Cypher and how to use it, we want to provide you with the resources we used throughout this section, as well as a few additional links for further knowledge and development.

Cypher basics and documentation

- GraphAcademy - Cypher Fundamentals. Learn Cypher in 60 minutes.
- Documentation: Cypher Manual
- Cypher Refcard
- Neo4j Community Site: Ask Questions, Get Answers on Cypher

Cypher for SQL developers

- Video: SQL to Cypher
- Free eBook: Graphs for RDBMS Developers

Other resources

- Medium Blog: Cypher Optimization
- Blog series, Handling Dates and Temporals in Cypher: Part 1, Part 2, Part 3
- Blog: Mark Needham on Cypher
- Blog: Max De Marzi on Cypher
- Tutorial by Eve Freeman: Building an ACL with Cypher
- Neo4j Medium Blog Channel

Learn with GraphAcademy Discrete.ad

[badge] | *//graphacademy.neo4j.com/courses/cypher-fundamentals/badge/*

Cypher Fundamentals Discrete

This course teaches you the essentials of using Cypher, Neo4j's powerful query language, in as little time as possible, with videos, quizzes and hands-on exercises.

Learn Cypher with GraphAcademy

Data modeling

This section is designed to give you the tools you need to design and implement an efficient and flexible graph database technology through a good graph data model.
Best practices and tips gathered from Neo4j's tenure of building and recommending graph technologies provide you with the confidence to build graph-based solutions with rich data models. The focus of this section is to provide you with the necessary guidelines and tools to help you model your domain as a graph.

How to create a graph data model

To start, *Graph Modeling Guidelines* introduces the basic process of designing a graph data model and walks you through the first steps to create a graph data model, building upon the foundations of the property graph data model.

It helps you determine the questions you need to ask and share design considerations, best practices learned from experts through the years, and tips for building a more flexible and clean data model to structure your data model for the best results.

Graph modeling guidelines
Modeling designs

Translating an RDBMS schema to graph

If you want to relate your existing knowledge of relational data models to the graph data model or to convert an existing relational model to graph, next section helps you translate that existing ERD skill and design to a graph data model.

From typical process steps to conversion mappings, we will walk through how the process can differ and what tables and columns look like as a graph.

Modeling: relational to graph

Optimizing graph data models

Finally, your data model may be working, but you find that query performance or other aspects are not giving you the quality you desired. The data model can affect queries and performance of your use case.

Learn how to improve your graph solution and maximize the capabilities of what is existing with recommendations for optimization techniques and ideas.

Graph modeling tips

Live graph models - GraphGists

If you look for graph data model examples or ideas, go to our Neo4j GraphGists, where the Neo4j Community share examples of their solutions. Based on your use case or industry, you can find some projects that could aid your design process.

Visit our GraphGists page and explore a rich variety of examples the Neo4j user community created! They are valuable and help developers and others by showing real-life solutions.

Online training

Learn everything you need to know about data modeling in Neo4j with the free Graph Data Modeling Fundamentals course on Neo4j GraphAcademy.
This course is part of the Beginners learning path which features four courses designed to teach you everything you need to know to feel confident working with Neo4j.

Graph modeling guidelines

Introduction

If you have ever worked with an object model or an entity-relationship diagram, the labeled property graph model will seem familiar.

Graph data modeling is the process in which a user describes an arbitrary domain as a connected graph of nodes and relationships with properties and labels. A Neo4j graph data model is designed to answer questions in the form of Cypher queries and solve business and technical problems by organizing a data structure for the graph database.

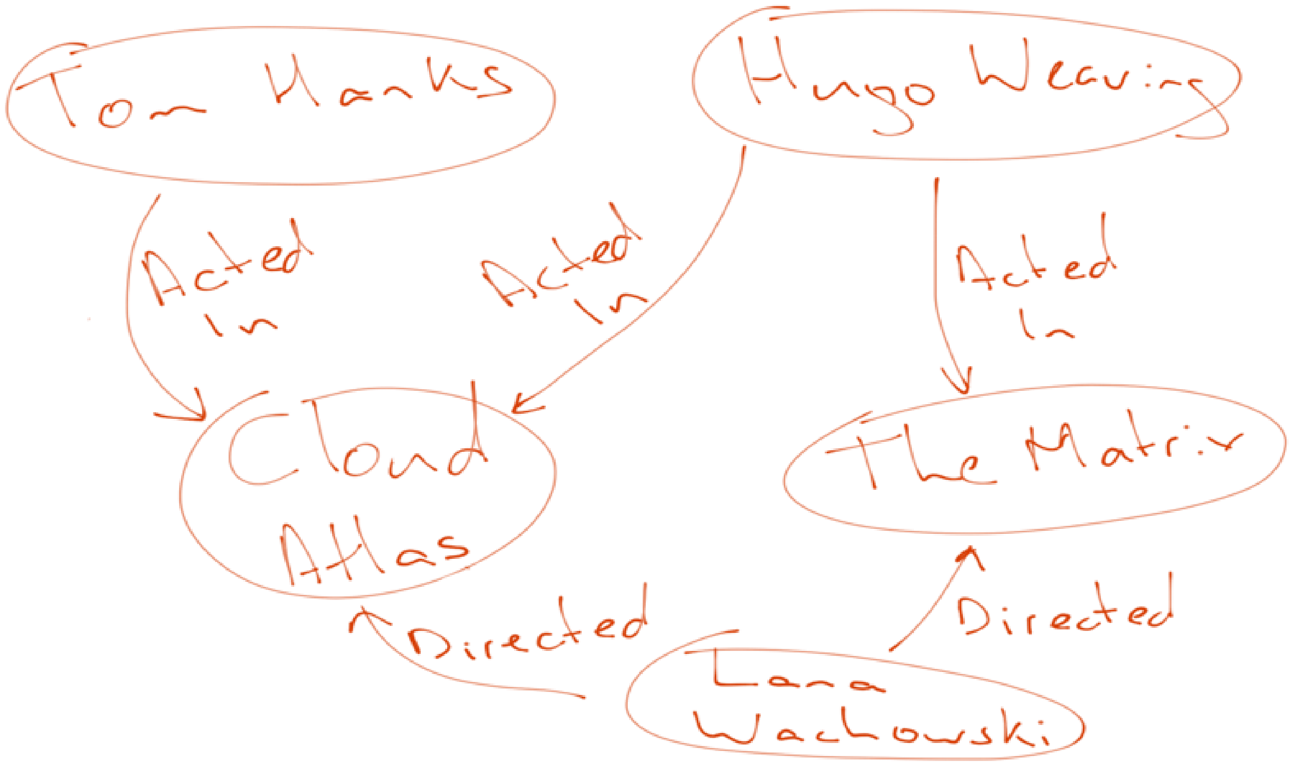
Graph data model = whiteboard-friendly

The graph data model is often referred to as being "whiteboard-friendly". Typically, when designing a data model, people draw example data on the whiteboard and connect it to other data drawn to show how different items connect. The whiteboard model is then re-formatted and structured to fit normalized tables for a relational model.

A similar process exists in graph data modeling, as well. However, instead of modifying the data model to fit a normalized table structure, the graph data model stays exactly as it was drawn on the whiteboard. This is where the graph data model gets its name for being "whiteboard-friendly".

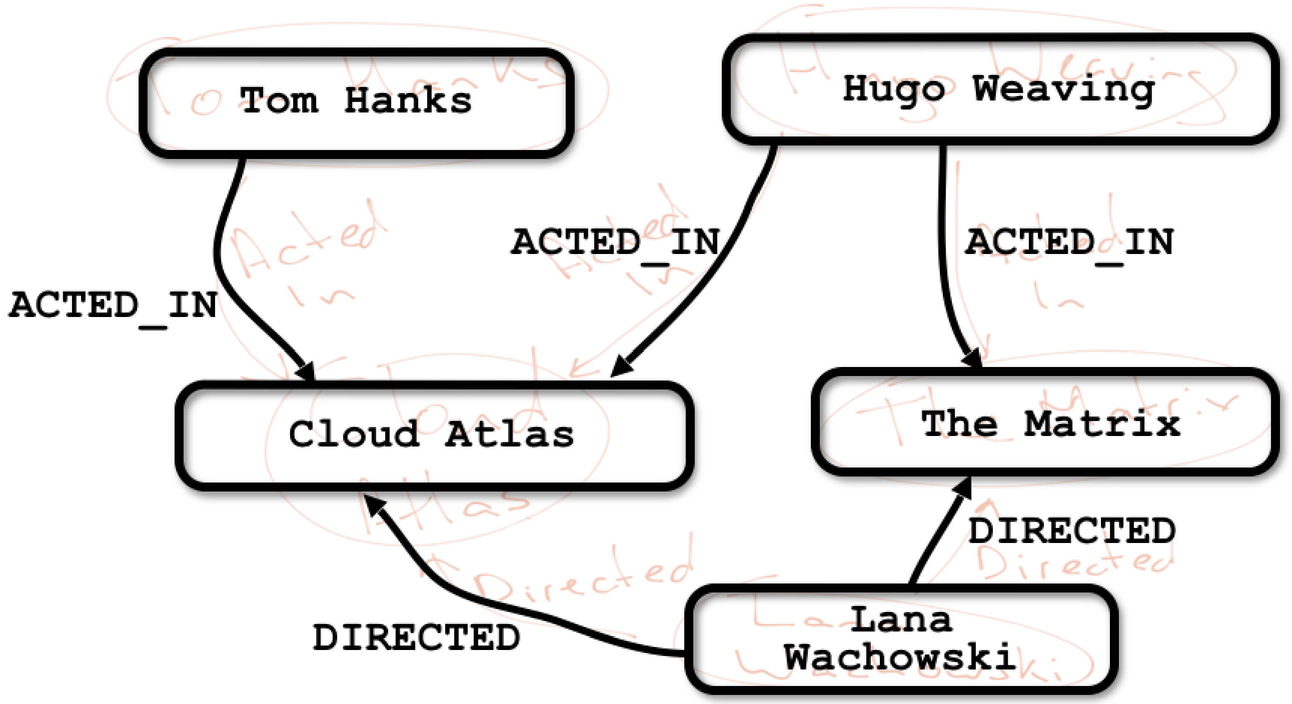
Let us look at an example to demonstrate this. In the whiteboard drawing below, we have a data set about the movie "The Matrix".

Matrix - whiteboard model



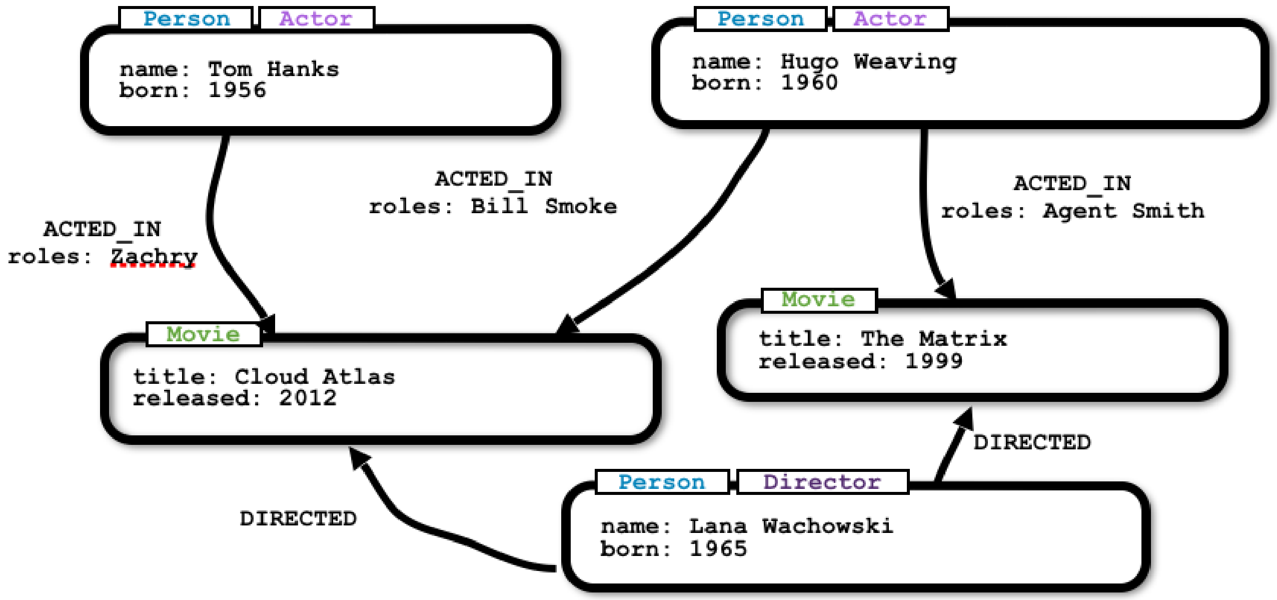
Next, we formalize our entities a bit and match expected syntax for relationship types to create the node/relationship view for the property graph model.

Matrix - match node and relationship format of property graph model



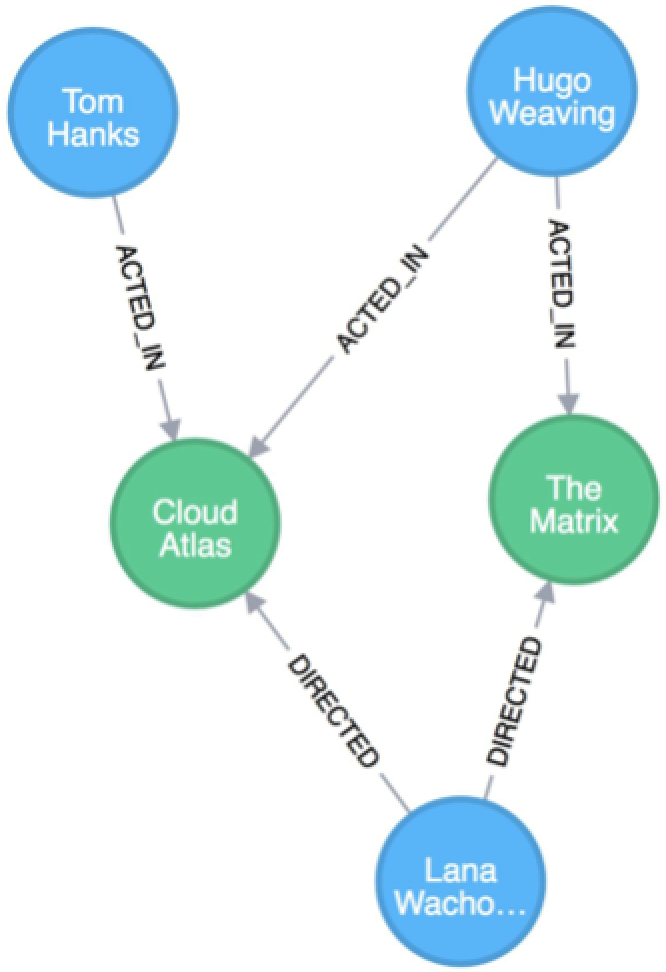
For our next step, we add labels and determine properties of our nodes and relationships for the property graph model.

Matrix - add labels and properties



Finally, we can view this data model in Neo4j and ensure it matches what we drew on the whiteboard. Also, notice how it is nearly identical to the whiteboard model we initially designed.

Matrix - final model in Neo4j



The ability to easily whiteboard your data model makes the graph data model incredibly simple and visual. There is no need to draw up business model versions or explain ERD terms to business users. Instead, the graph data model is easily understood by anyone.

Describing a domain

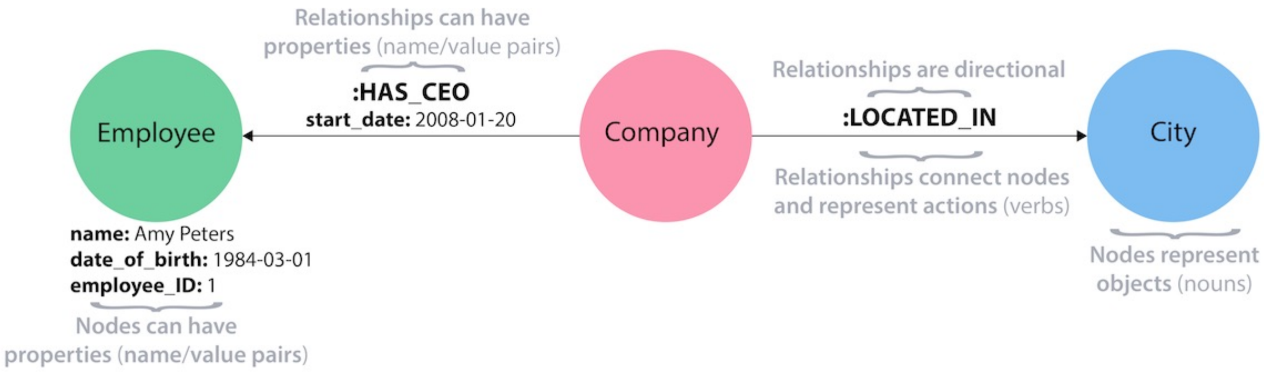
To better understand the process of designing a graph data model, let us take an example domain for a small set of data and walk through each step of how to create a graph data model from it. Consider the following scenario describing our example data entities and connections.

Scenario

> Two <em>people</em>, <strong>Sally</strong> and <strong>John</strong>, <u>are friends</u>. Both <strong>John</strong> and <strong>Sally</strong> <u>have read</u> the <em>book</em>, <strong>Graph Databases</strong>.

We can use the information in this statement to build our model by identifying the components as labels, nodes, and relationships. Let us take the scenario into pieces and define them as parts of our property graph model.

Review - Property Graph Elements (click to zoom)


Relationships can have properties (name/value pairs)
:HAS_CEO
start_date: 2008-01-20
Employee
Company
Relationships are directional
:LOCATED_IN
City
Relationships connect nodes and represent actions (verbs)
name: Amy Peters
date_of_birth: 1984-03-01
employee_ID: 1
Nodes can have properties (name/value pairs)
Nodes represent objects (nouns)

Nodes

The first entities that we identify in our domain are the nodes. Nodes are one of two fundamental units that form a graph (the other fundamental unit is relationships).

Nodes are often used to represent entities, but can also represent other domain components, depending on the use case. Nodes can contain properties that hold name-value pairs of data. Nodes can be assigned roles or types using one or more labels.

> You can often find nodes for the graph model by identifying nouns in your domain. Entities such as a car, a person, a customer, a company, an asset, and others similar can be defined as nodes for a good starting point.

We can identify nodes as entities with a unique conceptual identity. In our scenario we began for Sally and John, these entities are outlined below in bold.

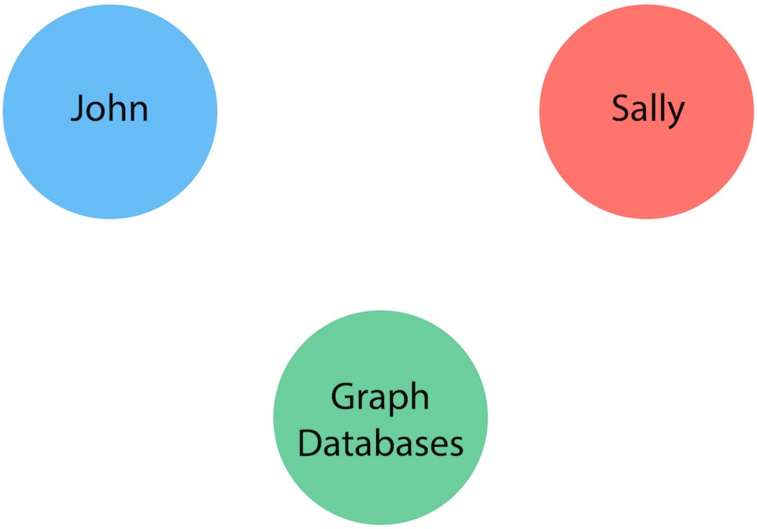
Scenario - Defining Nodes

> Two people, **John** and **Sally**, are friends. Both **John** and **Sally** have read the book, **Graph Databases.**

Extracting the nodes:
* John
* Sally
* Graph Databases

Remember that a graph database takes each instance of an entity as a separate node (John and Sally would be two separate nodes, even though they are both people), and Graph Databases would be a separate node from another book.

Graph Model - Nodes



Labels

Now when we have an idea of what our nodes will be, we can decide what labels (if any) to assign our nodes to group or categorize them. Let us remind the definition of what labels do and how they are used in the graph data model.

A label is a named graph construct that is used to group nodes into sets. All nodes labeled with the same label belongs to the same set.

Many database queries can work with these sets instead of the whole graph, making queries easier to write and more efficient. A node may be labeled with any number of labels, including none, making labels an optional addition to the graph.

Similar to how we found the nodes for our graph model by identifying the nouns in our scenario, you can identify labels by generic nouns or groups of persons, places, or things. General nouns that fit groups of items such as Vehicle, Person, Customer, Company, Asset, and similar terms can be used as labels in your graph.

To find out if we can group objects in our Sally and John scenario, we start by identifying the roles of our nodes (John, Sally, Graph Databases) mentioned in the statement. We can find two different types of objects in the statement, which are emphasized below.

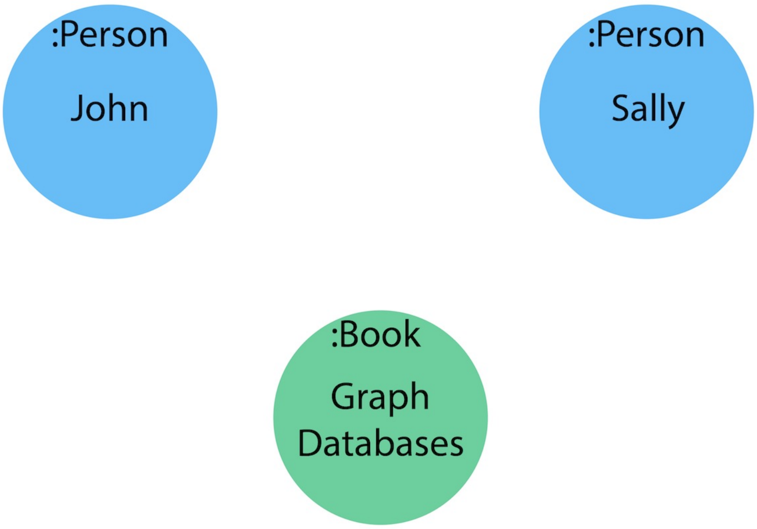
Scenario - Defining Labels

> Two *people*, John and Sally, are friends. Both John and Sally have read the *book*, Graph Databases.

Extracting the labels:
* *Person*
* *Book*

Now that we have identified both our nodes and labels, we can update our graph data model to assign the labels to the nodes they describe. For **John** and **Sally**, we apply the label *Person*. For **Graph Databases**, we apply the label *Book*.

Graph Model - Labels



Relationships

We now have our main entities and a way to group them, but we are still missing one vital piece of a graph database model - the relationships between the data!

A relationship connects two nodes and allows us to find related nodes of data. It has a source node and a target node that shows the direction of the arrow. Although you must store a relationship in a particular direction, Neo4j has equal traversal performance in either direction, so you can query the relationship without specifying direction.

The one core, consistent rule in a graph database is **"No broken links"**, ensuring that an existing relationship will never point to a non-existing endpoint. Since a relationship always has a start and end node, you cannot delete a node without also deleting its associated relationships.

> Just as we have found nodes and labels by looking for nouns, you can often find relationships for the graph model by identifying actions or verbs in your domain. Actions such as DRIVES, HAS_READ, MANAGES, ACTED_IN, and others similar can be defined as different types of relationships to exist between nodes.

Scenario - Defining Relationships

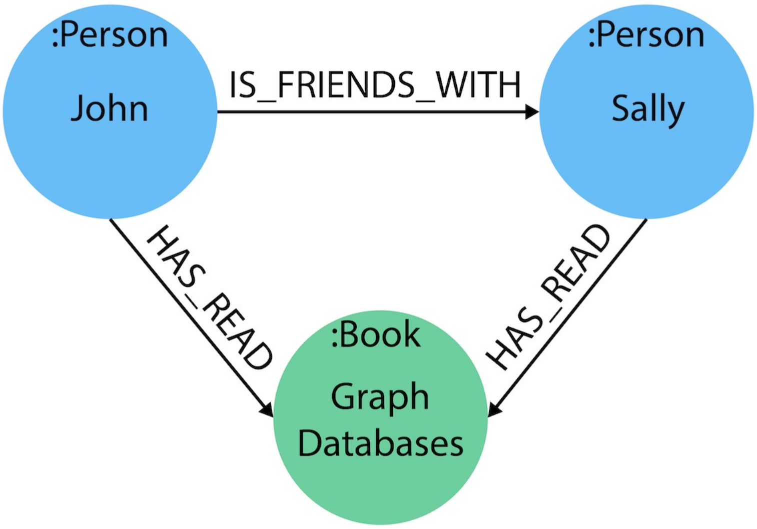
Let us identify the interactions (which are underlined in our scenario below) between the **John**, **Sally**, and **Graph Database** nodes.

> Two people, Sally and John, <u>are friends</u>. Both John and Sally <u>have read</u> the book, Graph Databases.

Relationships between nodes:<br> * John <u>is friends with</u> Sally<br> * Sally <u>is friends with</u> John<br> * John <u>has read</u> Graph Databases<br> * Sally <u>has read</u> Graph Databases

To sum up our findings, our John and Sally nodes (labeled <em>Person</em>) can be connected to each other by the <u>is friends with</u> relationship. John and Sally have both read the Graph Databases book, so we can connect each of their nodes (each labeled <em>Person</em>) to the Graph Databases node (labeled <em>Book</em>) with a <u>has read</u> relationship.

Graph Model - Relationships



Properties

We have gone through the process of creating a basic graph data model for the interactions between people and books. We can take this data model further by defining attributes of these entities as key-value properties.

Properties are name-value pairs of data that you can store on nodes or on relationships. Most standard data types are supported as properties, and you can find information on that in the section Graph database concepts^.

Properties allow you to store relevant data about the node or relationship with the entity it describes. They can often be found by knowing what kinds of questions your use case needs to ask of your data.

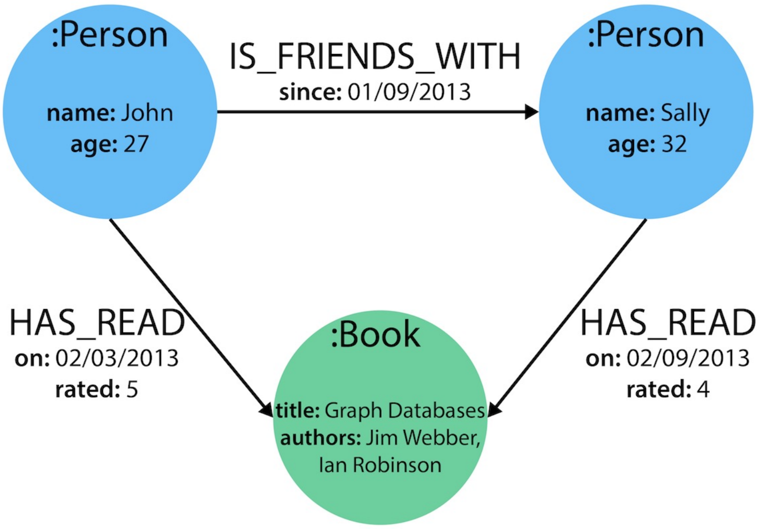
For our John and Sally scenario, we can list some questions that we might want to answer about the data.

Questions to ask of our John and Sally data model:

- When did John and Sally become friends? Or how long have they been friends?
- What is the average rating of the Graph Databases book?
- Who is the author of the Graph Databases book?
- How old is Sally?
- How old is John?
- Who is older, Sally or John?
- Who read the *Graph Databases* book first, Sally or John?

From this list of questions, you can identify the attributes that we need to store on the entities within our data model in order to answer these questions.

Graph Model - Properties



With the final model, we now can answer each of the questions we defined in our list. Of course, we can grow and change the model over time and add/remove relationships, nodes, properties, and labels. The flexibility and simplicity of the property graph data model allows users to easily review the data structure and update it according to the changing needs of the business.

Summary

That is the introduction to data modeling using a simple, straightforward scenario. There are plenty of opportunities throughout the upcoming sections to practice modeling domains and analyzing changes to the model that might need to be made.

Every data model is unique, depending on the use case and the types of questions that users need to answer with the data. Because of this, there is no "one-size-fits-all" approach to data modeling. Using best practices and careful modeling will provide the most valuable result in producing an accurate data model that benefits your processes and use case. A walkthrough of designs for different use cases is in the following section.

Resources

- Blog post: Graph Data Modeling Basics
- GraphGists: Graph Model Examples
- Blog post: Data Modeling Pitfalls to Avoid
- Free online training course: Graph Data Modeling Fundamentals

Modeling designs

> *In this section, you learn how to represent graph data using a variety of modeling decisions. The way you construct your data model can impact your queries and performance. Our goal is to show you how to evaluate your model and make appropriate changes, so you can define the best solution for your use case and maximize the performance of your queries.*

Why the data model makes a difference

As with any database, the data model that you design is important in determining the logic your queries and the structure of data in storage. This practice extends to graph databases, with one exception. Neo4j is schema-free, which means that your data model can adapt and change easily with your business.

Need to start collecting a new field and capture new analysis? Or need to change the way you interpret a customer or other entity and modify its definition? Or regulation requires systems to capture less information or restrict readability (change data format/types)?

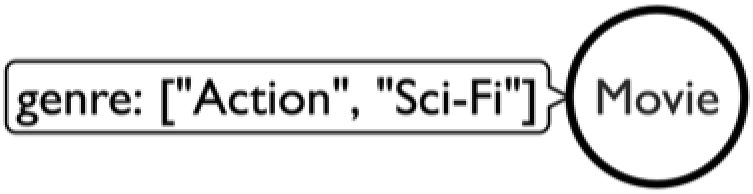
You may have worked for a company where each area or department defines a domain differently. Take, for instance, a generic customer domain. To different areas within the business, a customer can be defined as different types of individuals. These definitions may also change over time or the company may decide to unify the meaning of a customer across departments.

If you have worked with other types of databases, you will already be familiar with the development and administrative work that any of these scenarios entail. However, Neo4j allows you to effortlessly adjust detailed and broad changes across pieces or the entirety of the graph. Whether it is small changes over time or a broad definition that includes a variety of needed information about your entities, the database is able to handle it. It is simply up to the developers and architects to determine the structure of the data model and how to define entities for queries.

In the next few paragraphs, we will introduce a few different ways to look at different data sets and show how each impacts queries and performance for traversing graph data.

Property vs relationship

One of the earliest decisions you may encounter is whether to model something as a property on a node or as a relationship to a separate node. Take, for example, the data below modeling a movie genre as a property on the `Movie` node.

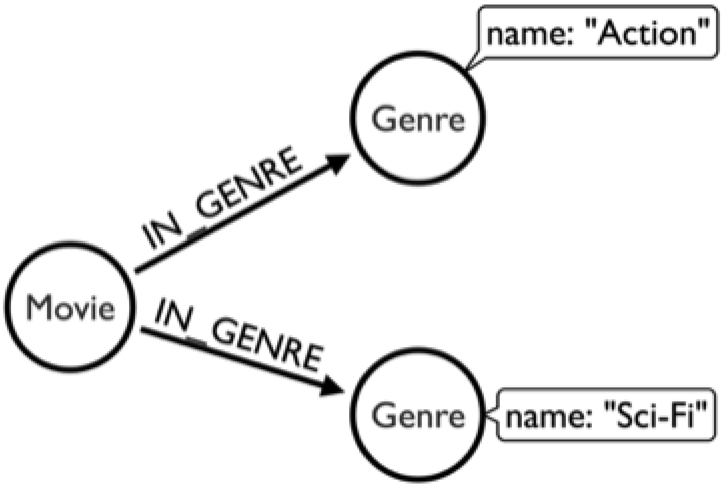


To write a query finding the genre(s) of a particular movie is very simple. It would find the `Movie` node it wants to know about, then return the values listed in the genre property. However, to find out which movies share genres, you would need a much more complex query to find each `Movie` node, loop through each of the genres in the property array, and compare with each value in the second movie’s property array of genres. This would take a toll on performance (nested looping and comparison of node properties), and the query would be much more complicated, as well.

The code block below is what the syntax would look like for each query. You can see the shift in logic and complexity of the loop in the second query.

```
//find the genres for a particular movie
MATCH (m:Movie {title:"The Matrix"})
RETURN m.genre;

//find which movies share genres
MATCH (m1:Movie), (m2:Movie)
WHERE any(x IN m1.genre WHERE x IN m2.genre)
AND m1 <> m2
RETURN m1, m2;
```

Now, instead, if we were to model our movies and genres as separate nodes and create a relationship between the two, we would come up with a model something like the image below.



This creates a completely separate entity (node) for the genre, allowing you to connect all the movies with a shared genre to that `Genre` node. Let us see how this changes our queries. To find the genres of a particular movie, it first needs to find the `Movie` node it is looking for (in this case, 'The Matrix'), then find the node that is connected to that movie through the `IN_GENRE` relationship.

The biggest difference is in the syntax for the second query to find which movies share genres. It is much simpler than our earlier version because it uses a natural, graph pattern (entity-relationship-entity) to find the information needed. First, Cypher finds a movie and the genre it is related to, then looks for a second movie that is in that same genre.

```
//find the genres for a particular movie
MATCH (m:Movie {title:"The Matrix"}),
      (m)-[:IN_GENRE]->(g:Genre)
RETURN g.name;

//find which movies share genres
MATCH (m1:Movie)-[:IN_GENRE]->(g:Genre),
      (m2:Movie)-[:IN_GENRE]->(g)
RETURN m1, m2, g
```

Neither version of the data model is worse or better, but the 'best' option highly depends on the types of queries you intend to run against your data.

If you plan to do analysis on individual items and return only details about that entity (like genres on a particular movie), then the first data model would serve perfectly well for your needs. However, if you need to run analysis to find common ground between entities or look at a group of nodes, then the second data model would definitely improve performance of those types of queries.

Complex data structures

As many of us can probably agree, not all data models are simple and straightforward. Data is messy, and the model must attempt to better-organize it to help us see patterns and make decisions.

One excellent example of a complex data structure that is difficult to model is Marvel comic data. In the Marvel universe, there are comics that have characters who make appearances or play lead roles. Comics can be organized into a series of particular storylines or narratives for a certain time, and major events can take place in a comic that define a character path or series. Creators (including writers, illustrators, etc) are the authors of comics, defining storyline, character adaptations, and events that happen. Multiple creators can also participate interchangably to create a comic or series.

This dataset already seems complicated, with several entities and relationships at work. It adds a new layer of complexity when trying to model the hierarchies and intermediate entities that exist here.

If you have some time, you can view the full video link to Peter's presentation on Vimeo, but we want to highlight two key challenges that Peter discusses in the data set.

First, he found that comic characters tend to be extremely dynamic. Many characters cannot be identified by name or costume or any particular property, as all of those change often.

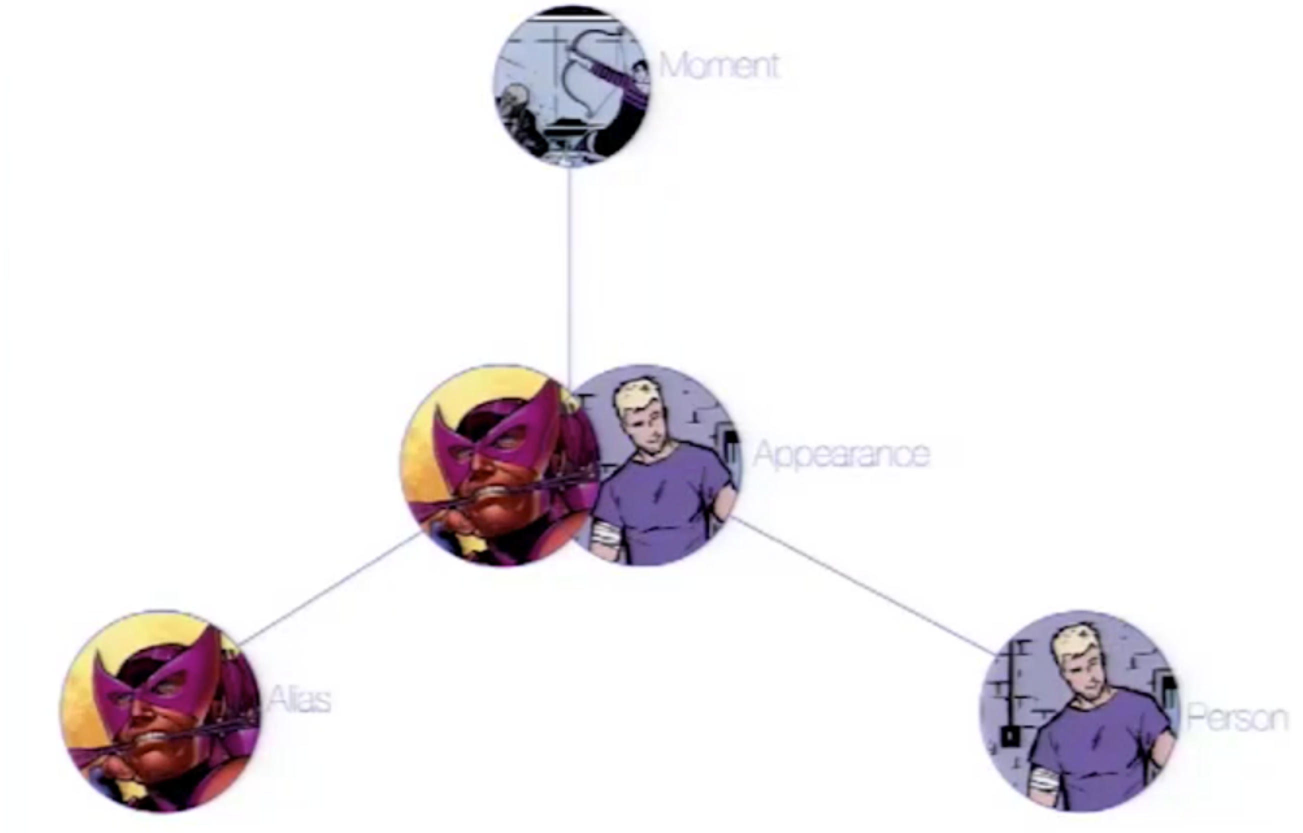
Second, Peter identified the issue of chronology. For those new to the comic universe, some might want to determine where to start or what comic(s) come next. However, comic issues are not always sequentially numbered, and there are even some storylines that appear across multiple series and back again. This makes it incredibly difficult to separate certain blocks of stories or events, along with renditions of characters.

Example: hyperedges or intermediate nodes

One modeling technique that is useful in this model is the concept of a hyperedge. Hyperedges (or intermediate nodes) are often created to model relationships that exist between more than two entities. They are often created to represent the connection of multiple entities at a point in time.

A common example of this is a university course. There may be multiple offerings of the same course with the same instructor in the same building, etc. Each section of the class (or offering) would then become an instance of the course.

The way Peter at Marvel handled hyperedges in their data is by creating an `Appearance` node that represents the intersection of a `Person` and an `Alias` at a particular time. This `Appearance` can be related to multiple `Moment` nodes where the person and alias appear as a unit. This is represented in the models shown below (also in the video).

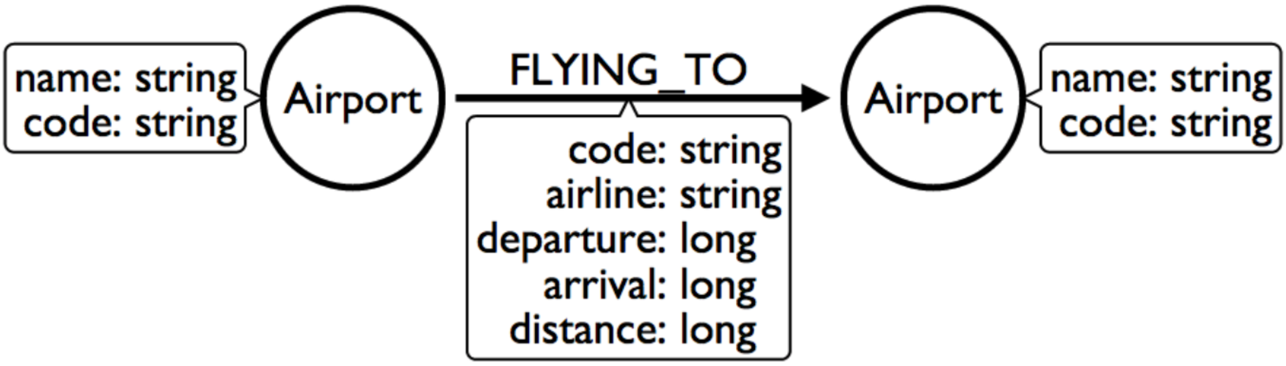


In a relational store, attempting to categorize and relate all of these complicated aspects would be extremely difficult and further complicate analysis and review of the data as a whole. The graph model allowed them to model this heavily dynamic universe and track all of the changing connections throughout their data. For this use case, graph was the perfect fit.

Time-bound data and versioning

One way to model time-specific data and relationships is by including data in the relationship type. Because Neo4j is optimized specifically for traversing relationships between entities, you can often improve query performance by specifying a date as the relationship type and only traversing particular dated relationships.

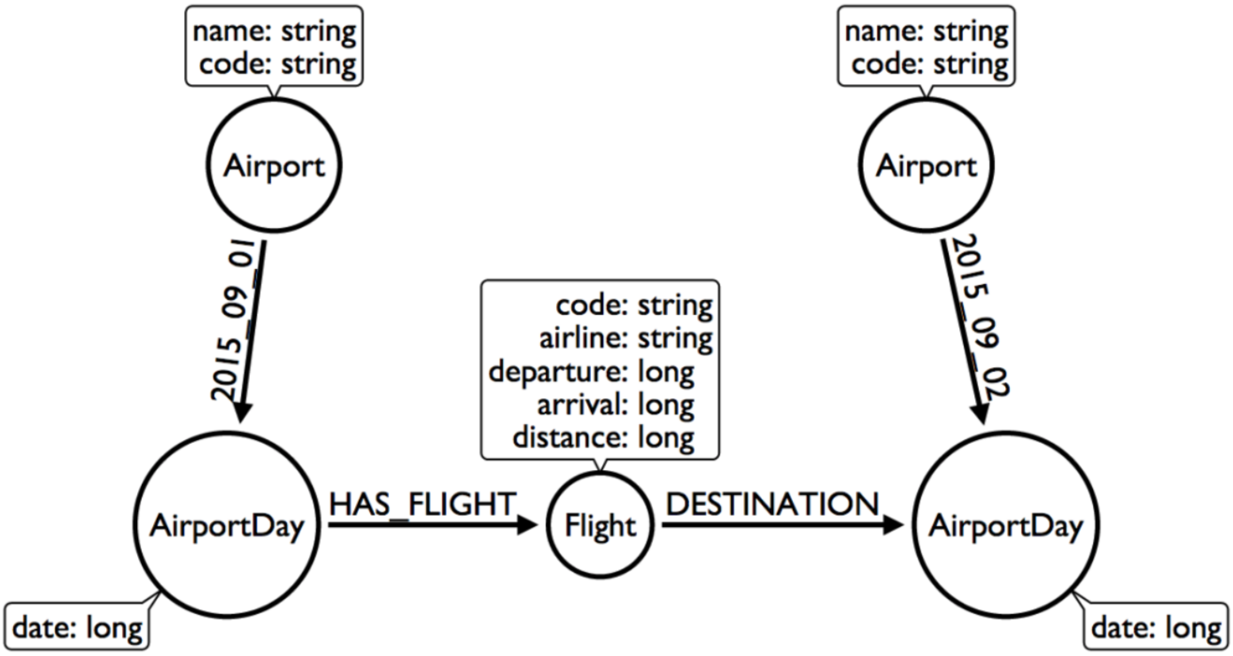
A common example is for modeling airline flights. An airline has a particular flight on a certain day to and from a specific location. We might start with a model like the first image below to show how flights travel from airport to airport.



We would soon realize that we need to model a `Flight` entity that exists between two destinations because multiple planes can travel between two destinations several times in one day.

However, your queries probably still show the model's weakness in filtering through all of the flights at a specific airport - especially for London and other major cities that have hundreds of flights connected to an `Airport` node over any span of time. Inspecting the several properties of each `Flight` node could be expensive on resources.

If we were to create a node for a particular airport day and a relationship with a date in the type, then we could write queries to find flights from an airport on any specified date (or date range). This way, you wouldn't need to check each flight relationship to an airport. Instead, you would only look at the relationships for the dates you cared about. This model turns out like the one below.



For the full walkthrough of the modeling process for airline flights, see Max's blog post.

Versioning

Similar to the model above where we create a dated relationship type, we can also use this to track versions of our data. Tracking changes in the data structure or showing a current and past value can be incredibly important for auditing purposes, trend analysis, etc.

For instance, if you wanted to create a new effective-dated relationship between a person and their current address, but also retain past addresses, you could use the same principle of including a date in the relationship type. To find the current address of the person, the query would look for the most recently dated relationship.

Taking the best of both worlds

Sometimes, you might find that one model works really well for one scenario you need, but another model is better for something else. For instance, some models will perform better with write queries and other models handle read queries better. Both capabilities are important to your use case, so what do you do?

In these cases, you can combine both models and use the benefits of each. Yes, you can use more than one data model in your graph!

The tradeoff is that now you will need to maintain two models. Each time you create a new node or relationship or update pieces of the graph, you will need to make changes to accommodate both models. This can also impact query performance, as you might have double the syntax needed to update each model.

While this is definitely a possible option, you should know the maintenance costs and evaluate whether those costs are overcome by the performance improvements you will see for each needed query. If so, being able to use more than one data model is a great solution!

Resources

- Blog post: Modeling relationships
- Max's blog post: Modeling airline flights
- Follow-up blog post: Flight search
- Blog post: Modeling data categories
- Blog post: Modeling mutual funds
- Blog post series: Building a Dating Site
- Blog series: Building a Twitter Clone
- Ask Questions on the Neo4j Community Site!

Modeling: relational to graph

> *For those with a background in relational data modeling, this guide helps transfer your existing knowledge of the processes and components used for relational data modeling into graph data modeling. It will help to compare and contrast the steps of each process and help you identify where the data modeling is similar or different for each type of database.*

Introduction

If you are familiar with the relational data model that has tables, columns, relationship cardinalities, and other components, graph data modeling will not seem entirely foreign. The design of the data model still needs to be based upon requirements for access, queries, performance expectation, and business logic. However, the structure of a graph data model is laid out slightly differently. These differences were discussed in the Concepts: RDBMS to Graph section earlier.

You may have an entirely new project that you want to create a graph data model, but are only familiar with how to create the relational model. Or you could already have an existing project with a relational model that you want to convert to graph. Either way, this guide will take existing knowledge of the relational data model and show you how to use that to create a graph model.

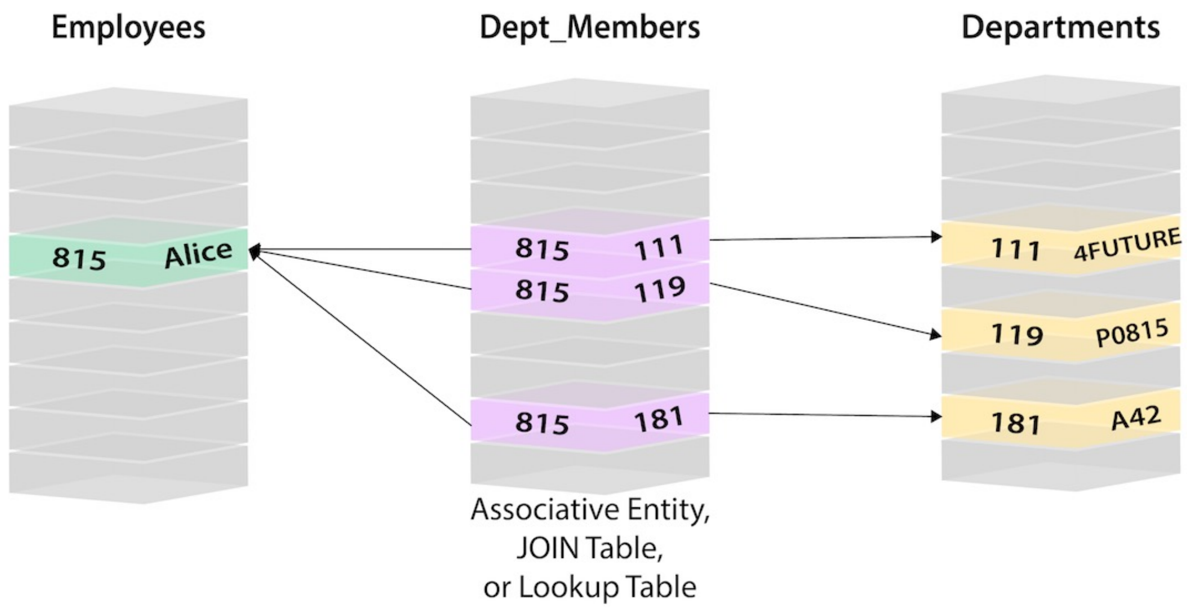
Relational and graph architecture

As a quick overview, remember that relational databases rely upon index lookups and table joins to connect different entities. This quickly becomes a problem for performance, especially when there are several tables joined, millions of rows on tables, or complex queries that traverse various levels through subqueries.

In our example from the concepts page, to find which departments Alice works for, you would need to query the `Person` table to find the row representing Alice, which is tied to a unique ID as the primary key. Then, your query would go to the associative entity table (`Person_Dept`) to find where her ID is tied to one or more department IDs. Finally, the query would check the `Department` table to find the actual values for those department IDs you found in the associative entity table.

The image below reviews this example we just described.

Relational - Person and Department tables (click to zoom)



In a graph, you do not need to worry about table joins and index lookups because graph data is structured by each, individual entity and its relationships with other individual entities.

Ok, so how do we go from creating relational data models to a graph data model?

Data model transformation tips

Let us look at some of the key components in a relational data model and translate those into components of a graph data model. The steps to help you with the transformation of a relational diagram are listed below.

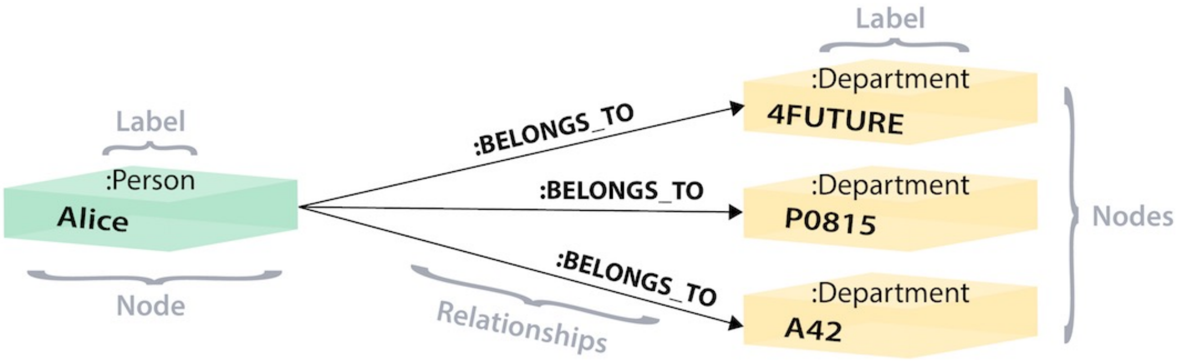
- *Table to Node Label* - each entity table in the relational model becomes a label on nodes in the graph model.
- *Row to Node* - each row in a relational entity table becomes a node in the graph.
- *Column to Node Property* - columns (fields) on the relational tables become node properties in the

graph.

- *Business primary keys only* - remove technical primary keys, keep business primary keys.
- *Add Constraints/Indexes* - add unique constraints for business primary keys, add indexes for frequent lookup attributes.
- *Foreign keys to Relationships* - replace foreign keys to the other table with relationships, remove them afterwards.
- *No defaults* - remove data with default values, no need to store those.
- *Clean up data* - duplicate data in denormalized tables might have to be pulled out into separate nodes to get a cleaner model.
- *Index Columns to Array* - indexed column names (like email1, email2, email3) might indicate an array property.
- *Join tables to Relationships* - join tables are transformed into relationships, columns on those tables become relationship properties

If you apply the items in the list above to our example finding Alice's departments, we can come to a graph like the one shown below.

Graph - Alice and 3 Departments as nodes (click to zoom)

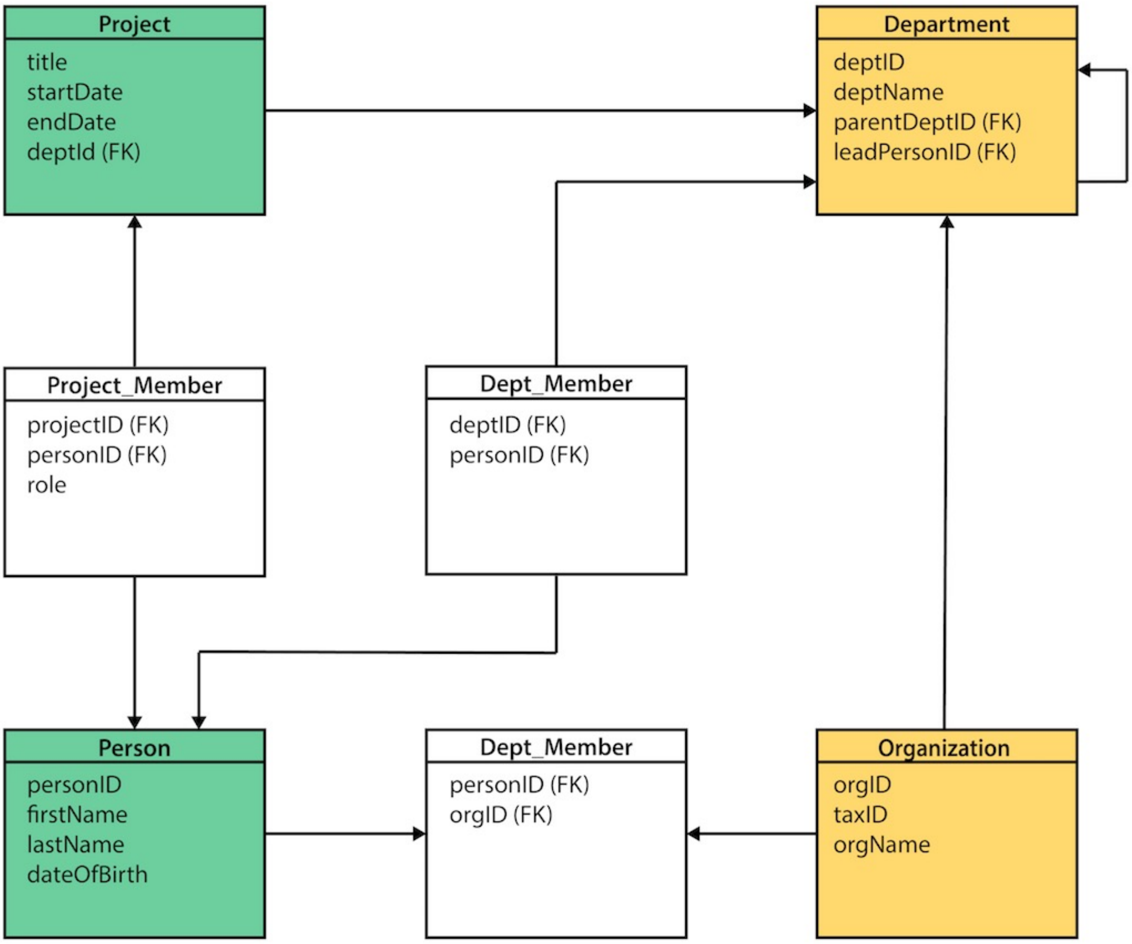


Though the two models have similarities such as categorizing data by using either a table structure or a label, the graph model does not confine data to a pre-defined and strict table/column layout. We will look at another example in the next section.

Organizational domain data model

To give us another chance to practice, we will use a standard organizational domain and show how it would be modeled in a relational database versus a graph database. To give yourself an extra challenge, try to create the graph data model on your own and then see how closely it lines up!

Organizational Domain - Relational Model



Conversion steps

First, we can categorize our tables by main domain tables and associative entity tables by colors. Then, we can turn our table names into node labels. In this case, `Project`, `Person`, `Department`, and `Organization` become labels in our graph model.

The rows on our tables become their own nodes and the columns in those rows become the properties on those nodes. For example, your row on the Person table will become a node with your name and date of birth as the properties on your node. Any indexed columns that allow multiple similar values will become an array (such as skill1, skill2, skill3 columns translate to three values stored in an array property on a node).

If there are any technical primary keys (in other words, primary keys that were created simply to make the row unique - like a project_id in case there are multiple projects with the same title), then remove those and only keep the properties that are needed for the business requirements. You will also need to add unique constraints for the business primary keys in order to ensure the database will not allow duplicates.

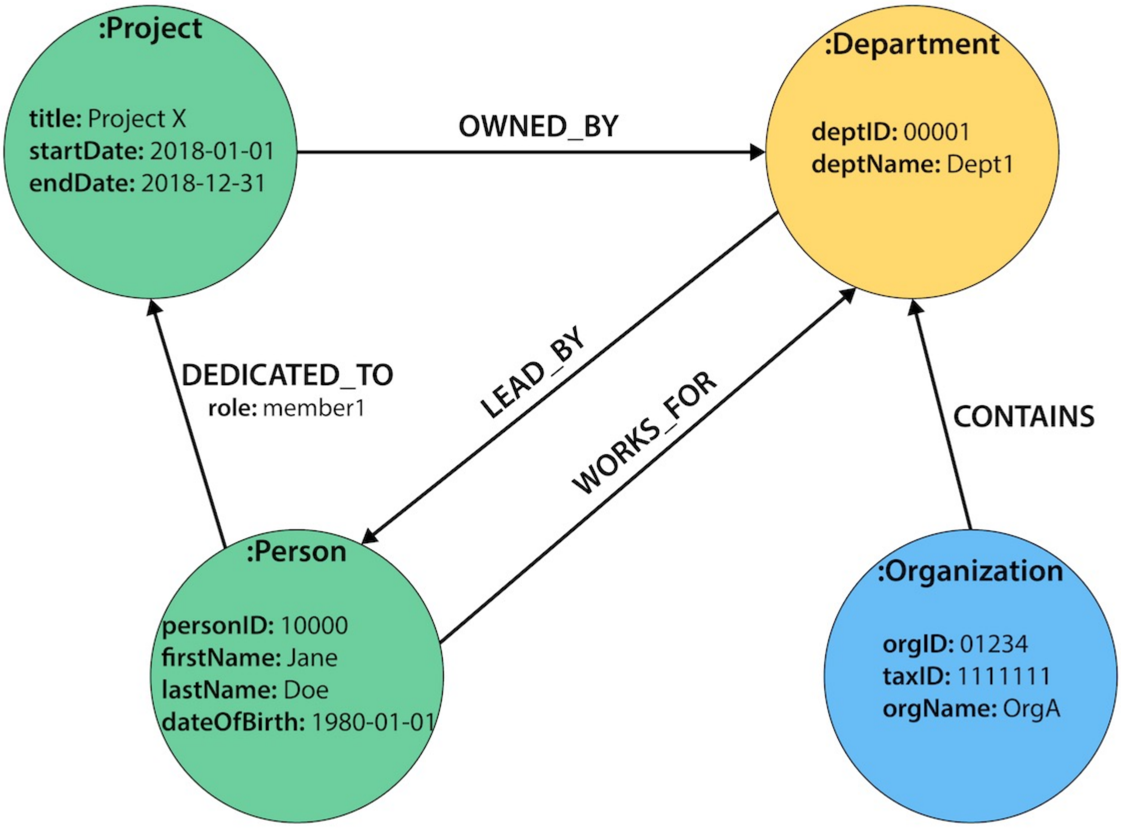
Foreign keys that would aid in relational join lookups are transformed into relationships, as they show the links between the nodes. Join tables (or associative entity tables) become relationships, as well, with any join table columns moved to relationship properties.

Since you only store the needed properties in Neo4j, you do not need to store nulls and empty values, so you can remove any default values that may have been created in a relational model.

Finally, any duplicate data created to normalize tables or de-normalize for simplicity's sake needs removed, as it is unneeded in a graph.

After this process, your graph data model should look something like the image below.

Answer: Organizational Domain - Graph Model



It is important to have an basic understanding of the graph model before you start to import data, as it becomes easier to hydrate that model or adjust it later, as needs change. In an upcoming guide, how you model your graph data can impact queries, performance, and model changes.

Resources

- DZone Refcard: From Relational to Graph
- Concepts: Relational to Graph
- Review: Property Graph Model

Graph modeling tips

Goals

> *In this guide, you find some helpful information to designing a data model for your domain. Optimizing the model helps developers to maximize performance of the system and queries.*

Tips and tricks of modeling

As you may have found in reading the modeling guides or in your own experience with graph data modeling, there is no right or wrong way to model your data. Some ways may be better-suited to your needs and more performant on the aspects you prioritize, but you have options.

To find the best data model for your needs, it often helps to approach with a few techniques and make data model decisions from that analysis. We will talk about a few tips and tricks in the next paragraphs to help you decide upon your data model.

Write your queries first

Knowing the kinds of questions and queries you want to ask of your data is a great way to determining the structure of your data model. If you know your queries need to return results within a certain date range, then you probably should ensure that date is not a property on a node, but rather stored as a separate node or relationship. In contrast, for a university program domain, finding similar class offerings to a current course might work well with a high-level category hierarchy that makes searching all classes within a subject topic more efficient.

Even if you do not know the exact query syntax just yet, understanding the intention of the system or application you are building and then constructing the model around the business need will help you organize it in a more accurate way.

Prioritize queries

It is very difficult (if not impossible) to find the perfect model for every query or functionality. As we talked about in the modeling designs guide, there are tradeoffs with choosing one particular model over another (or using multiple). While you may improve certain things, there is no way to get a one-size-fits-all solution.

Instead, you should determine which model best suits your needs. You may not be able to max out performance on every individual query, but you may be able to get the most out of your system with certain resources, time, and code.

To do this, you will need to decide which queries must absolutely have maximum performance and which capabilities are critical to provide value. This may be a tough decision, but no matter the technology you are working with, these decisions will exist in some facet or other. What makes Neo4j more valuable is that the model is flexible and able to change if your priorities adjust over time.

Test it out

You may come across scenarios that you did not realize in the design stages. One of the best ways to find these is to actually test the model out.

Loading portions of your data and executing tests and queries on the system will determine if the results you receive fit your needs or your expected performance. Again, Neo4j is flexible so that you can adjust the model or optimize your queries to fine-tune the outputs.

Having trouble deciding between one or more models? Try creating a proof-of-concept test for each model and both together and see how they operate. What is complicated or what is not worth the hassle? Is there one that actually performs better in real life or does a multiple-data-model approach truly give you the best results? Sometimes, the best way to find out is to test it out with live data.

Refactoring your graph

As mentioned in above and in other guides, changes are always possible with Neo4j. The data model is purposely flexible and easy to adjust for this very reason. Business needs and priorities tend to fluctuate. Users may also change their behaviors and cause shifts for the business.

Cypher allows you to write queries to run mass updates across labels, add or remove properties, and insert additional nodes and relationships into the structure. There are also procedures to aid in batching queries and executing updates to cluster instances, as available.

For more information in this topic, check out the APOC standard library!

Other concerns

The size of your data set also can impact queries and performance. If you have a smaller data set, then you may not see much performance impact in more complex queries. It is only when the amount of your data grows that you may see increased impacts. This is where the data model and query optimizations become vital to maximizing the value from your system.

Resources

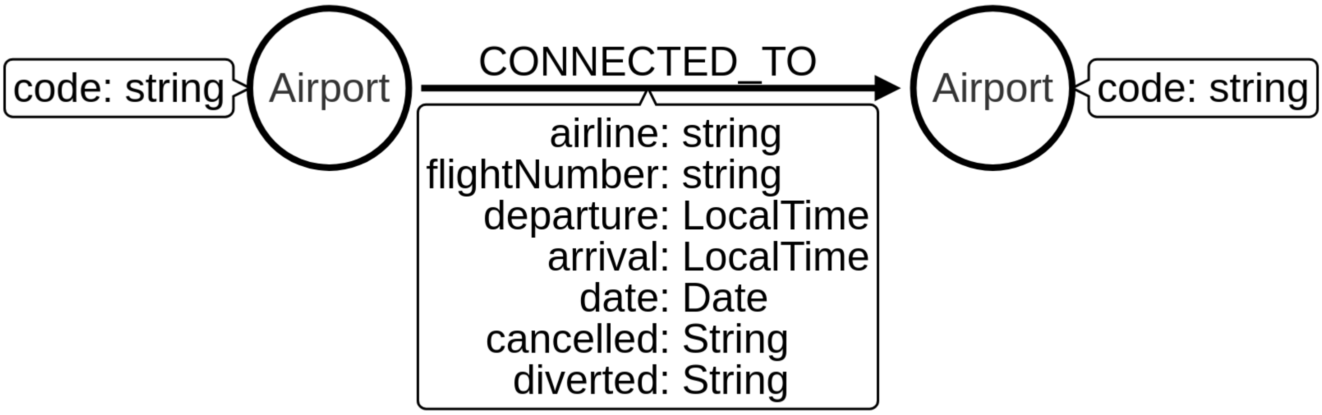
- Ask Questions on the Neo4j Community Site!

Graph model refactoring

> *Building on the Cypher Basics guides, this guide provides a worked example of changing a graph model. Upon finishing this guide, you should be able to evolve your graph model based on changing requirements.*

Airports dataset

In this guide we're going to use an airports dataset that contains connections between US airports in January 2008. We have the data in a CSV file, and this is the graph model that we're going to import it into:



Before we import any data, we're going to create a unique constraint on the `Airport` label and `code` property to ensure that we don't accidentally import duplicate airports. The following query creates this constraint:

```
CREATE CONSTRAINT airport_id
FOR (airport:Airport) REQUIRE airport.code IS UNIQUE
```

Table 8. Results

0 rows available after 86 ms, consumed after another 0 ms. Added 1 constraints

And the following query loads the data from a CSV file using the `LOAD CSV` tool:

```
LOAD CSV WITH HEADERS FROM "https://raw.githubusercontent.com/neo4j-
contrib/training/master/modeling/data/flights_1k.csv" AS row
MERGE (origin:Airport {code: row.Origin})
MERGE (destination:Airport {code: row.Dest})
MERGE (origin)-[connection:CONNECTED_TO {
  airline: row.UniqueCarrier,
  flightNumber: row.FlightNum,
  date: date({year: toInteger(row.Year), month: toInteger(row.Month), day: toInteger(row.DayofMonth)}),
  cancelled: row.Cancelled,
  diverted: row.Diverted}]->(destination)
ON CREATE SET connection.departure = localtime(apoc.text.lpad(row.CRSDepTime, 4, "0")),
              connection.arrival = localtime(apoc.text.lpad(row.CRSArrTime, 4, "0"))
```

This query:

- Creates a node with an `Airport` label with a `code` property that has a value from the `Origin` column in the CSV file
- Creates a node with an `Airport` label with a `code` property that has a value from the `Dest` column in the CSV file
- Creates a relationship of type `CONNECTED_TO` with several properties based on columns in the CSV file.

If we run this query we'll see the following output:

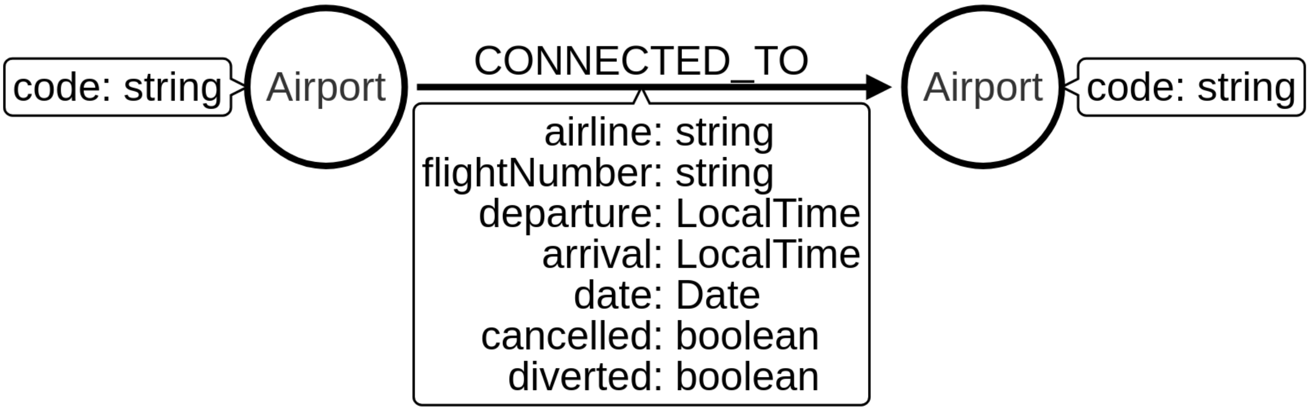
Table 9. Results

Added 62 labels, created 62 nodes, set 7062 properties, created 1000 relationships, completed after 376 ms.

This model is a good starter one, but there are some improvements that we can make.

Convert property to boolean

The diverted and cancelled properties on the CONNECTED_TO relationships contain string values of 1 and 0. Since these values are representing booleans, we can use the apoc.refactor.normalizeAsBoolean procedure to convert the values from strings to booleans.



The following query does the conversion for the diverted property:

```
MATCH (:Airport)-[connectedTo:CONNECTED_TO]->(:Airport)
CALL apoc.refactor.normalizeAsBoolean(connectedTo, "diverted", ["1"], ["0"])
RETURN count(*)
```

Table 10. Results

count(*)
1000

And the following query does the conversion for the cancelled property:

```
MATCH (origin:Airport)-[connectedTo:CONNECTED_TO]->(departure)
CALL apoc.refactor.normalizeAsBoolean(connectedTo, "cancelled", ["1"], ["0"])
RETURN count(*)
```

Table 11. Results

count(*)
1000

If we have a lot of relationships to update, we may get an OutOfMemory exception if we try to refactor them all in one transaction. We can therefore process them in batches using the apoc.periodic.iterate procedure. The following query does this for the cancelled and reverted properties in the same query:

```
UNWIND ["cancelled", "reverted"] AS propertyToDelete
CALL apoc.periodic.iterate(
  "MATCH (:Airport)-[connectedTo:CONNECTED_TO]->(:Airport) RETURN connectedTo",
  "CALL apoc.refactor.normalizeAsBoolean(connectedTo, $propertyToDelete, ['1'], ['0'])
   RETURN count(*)",
  {params: {propertyToDelete: propertyToDelete}, batchSize: 100})
YIELD batches
RETURN propertyToDelete, batches
```

The apoc.periodic.iterate procedure in this query takes in three parameters:

- An outer Cypher query that finds and returns a stream of CONNECTED_TO relationships to be processed.
- An inner Cypher query that processes those CONNECTED_TO relationships, converting to boolean any values for the specified property on those relationships. It does this using the apoc.refactor.normalizeAsBoolean procedure, which itself takes in several parameters:
    - the entity on which the property exists
    - the name of the property to normalize
    - a list of values that should be considered true
    - a list of values that should be considered false
- Configuration values for the procedure, including:
    - params - parameters passed into those Cypher queries
    - batchSize- controls the number of inner statements that are run within a single transaction

When we run this query we see the following output:

Table 12. Results

propertyToDelete	batches
"cancelled"	10
"reverted"	10

Once we’ve done this, we can write the following query to return all cancelled connections:

```
MATCH (origin:Airport)-[connectedTo:CONNECTED_TO]->(destination)
WHERE connectedTo.cancelled
RETURN origin.code AS origin,
       destination.code AS destination,
       connectedTo.date AS date,
       connectedTo.departure AS departure,
       connectedTo.arrival AS arrival
```

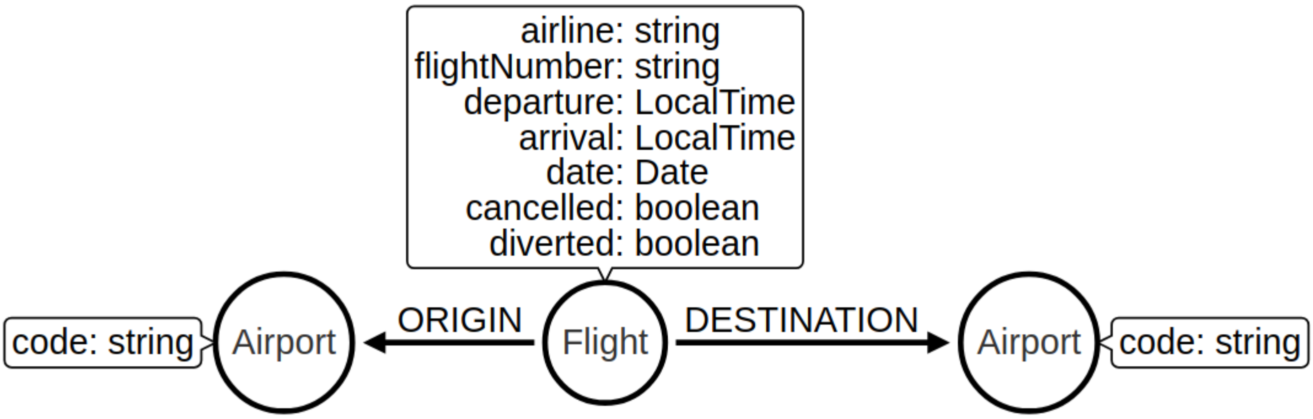
Table 13. Results

origin	destination	date	departure	arrival
"LAS"	"OAK"	2008-01-03	07:00	08:30
"LAX"	"SFO"	2008-01-03	09:05	10:25
"LAX"	"OAK"	2008-01-03	11:00	12:15
"LAX"	"SJC"	2008-01-03	19:30	20:35
"LAX"	"SFO"	2008-01-03	16:20	17:40
"MDW"	"STL"	2008-01-03	11:10	12:15
"MDW"	"BDL"	2008-01-03	08:45	11:40
"MDW"	"DTW"	2008-01-03	06:00	08:05

origin	destination	date	departure	arrival
"MDW"	"STL"	2008-01-03	14:45	15:50
"MDW"	"BNA"	2008-01-03	19:25	20:45
"OAK"	"BUR"	2008-01-03	13:10	14:15
"OAK"	"BUR"	2008-01-03	17:05	18:10

Create node from relationship

Next, imagine that we want to write a query that finds a specific flight. This is quite difficult with our existing model because flights are represented as relationships. We can evolve our model to create a `Flight` node from the properties stored on the `CONNECTED_TO` relationship.



The following query does this refactoring:

```
CALL apoc.periodic.iterate(
  "MATCH (origin:Airport)-[connected:CONNECTED_TO]->(destination:Airport) RETURN origin, connected,
destination",
  "CREATE (flight:Flight {
     date: connected.date,
     airline: connected.airline,
     number: connected.flightNumber,
     departure: connected.departure,
     arrival: connected.arrival,
     cancelled: connected.cancelled,
     diverted: connected.diverted
   })
   MERGE (origin)<-[:ORIGIN]-(flight)
   MERGE (flight)-[:DESTINATION]->(destination)
   DELETE connected",
  {batchSize: 100})
```

As with our previous query, this query uses the `apoc.periodic.iterate` procedure so that we can do the refactoring in batches rather than within a single transaction. The procedure takes in three parameters:

- An outer Cypher query that finds and returns a stream of `CONNECTED_TO` relationships, and origin and destination airports that need to be processed.
- An inner Cypher query that processes those entities, creating a node with the label `Flight` and creating relationships from that node to the origin and destination airports.
- `batchSize` configuration, which sets to `100` the number of inner statements that are run within a single transaction.

If we execute the query we'll see the following output:

Table 14. Results

batches	total	timeTaken	committedOperations	failedOperations	failedBatches	retries	errorMessages	batch	operations	wasTerminated
10	1000	0	1000	0	0	0	{}	{total: 10, committed: 10, failed: 0, errors: {}}	{total: 1000, committed: 1000, failed: 0, errors: {}}	FALSE

We can also do this refactoring using the `apoc.refactor.extractNode` procedure.

```
CALL apoc.periodic.iterate(
  "MATCH (origin:Airport)-[connected:CONNECTED_TO]->(destination:Airport)
   RETURN origin, connected, destination",
  "CALL apoc.refactor.extractNode([connected], ['Flight'], 'DESTINATION', 'ORIGIN')
   YIELD input, output, error
   RETURN input, output, error",
  {batchSize: 100});
```

This does the same as the previous query, but the outer Cypher query uses the `apoc.refactor.extractNode` procedure to create the `Flight` node and create relationships to origin and destination airports. If we run this query we'll see the following output:

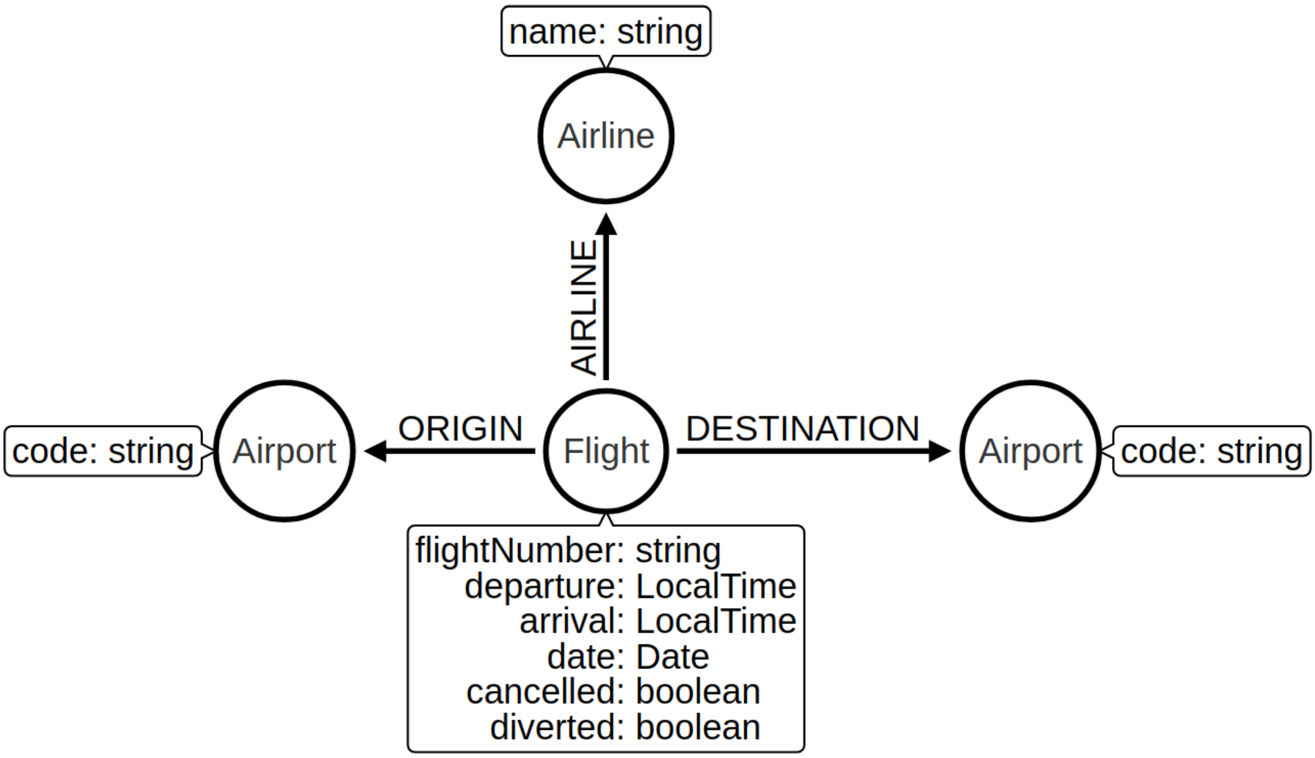
Table 15. Results

batches	total	timeTaken	committedOperations	failedOperations	failedBatches	retries	errorMessages	batch	operations	wasTerminated
10	1000	0	1000	0	0	0	{}	{total: 10, committed: 10, failed: 0, errors: {}}	{total: 1000, committed: 1000, failed: 0, errors: {}}	FALSE

Create node from property

At the moment the airline for our flights is stored in the `airline` property on `Flight` nodes. This means that if we wanted to return a stream of all airlines we'd need to scan through every flight and check the `airline` property on each of those flights.

We can make it easier, and more efficient, to write this query by creating a node with an `Airline` label for each airline:



Let’s first create a constraint on the `Airline` label and `name` property so that we don’t create duplicate airline nodes:

```
CREATE CONSTRAINT airline_id
FOR (airline:Airline) REQUIRE airline.name IS UNIQUE
```

Table 16. Results

0 rows available after 107 ms, consumed after another 0 ms. Added 1 constraints

And now we can execute the following query to do the refactoring:

```
CALL apoc.periodic.iterate(
   'MATCH (flight:Flight) RETURN flight',
   'MERGE (airline:Airline {name:flight.airline})
    MERGE (flight)-[:AIRLINE]->(airline)
    REMOVE flight.airline',
   {batchSize:10000, iterateList:true, parallel:false}
)
```

Again we’re using the `apoc.periodic.iterate` procedure, with the following parameters:

- An outer Cypher statement that returns a stream of `Flight` nodes to be processed
- An inner Cypher statementthat processes these flight nodes, creating `Airline` nodes based on flights' `airline` property and created an `AIRLINE` relationship from the `Flight` to the `Airline` node. We then remove the `airline` property from the `Flight` node.

If we run this query we’ll see the following output:

Table 17. Results

batches	total	timeTaken	committedOperations	failedOperations	failedBatches	retries	errorMessages	batch	operations	wasTerminated
1	1000	0	1000	0	0	0	{}	{total: 1, committed: 1, failed: 0, errors: {}}	{total: 1000, committed: 1000, failed: 0, errors: {}}	FALSE

We can then write the following query to find the airlines and number of flights involving each:

```
MATCH (airline:Airline)<-[:AIRLINE]-(:Flight)
RETURN airline.name AS airline, count(*) AS numberOfFlights
```

This does the same as the previous query, but the outer Cypher query uses the `apoc.refactor.extractNode` procedure to create the `Flight` node and create relationships to origin and destination airports. If we run this query we'll see the following output:

Table 18. Results

airline	numberOfFlights
"WN"	1000

Resources

This guide has shown how to refactor a graph model, with help from procedures in the APOC Library. Below are some resources for learning more about refactoring in Neo4j:

- APOC Library
  - Graph Refactoring Procedures

Data modeling tools

To help you with designing a new graph database structure, the following tools were created:

- Neo4j Data Importer
- Arrows.app

If you have data stored in a flat file format, use **Neo4j Data Importer** — no-code tool for importing flat file data (`.csv`, `.tsv`) into Neo4j databases. Today Data Importer is available on **AuraDB** instances.

Arrows.app is ideal for making a sketch of a graph, therefore if your aim is to design a domain model that your data will follow, go to the Arrows.app page.

Projects overview

Neo4j Data Importer

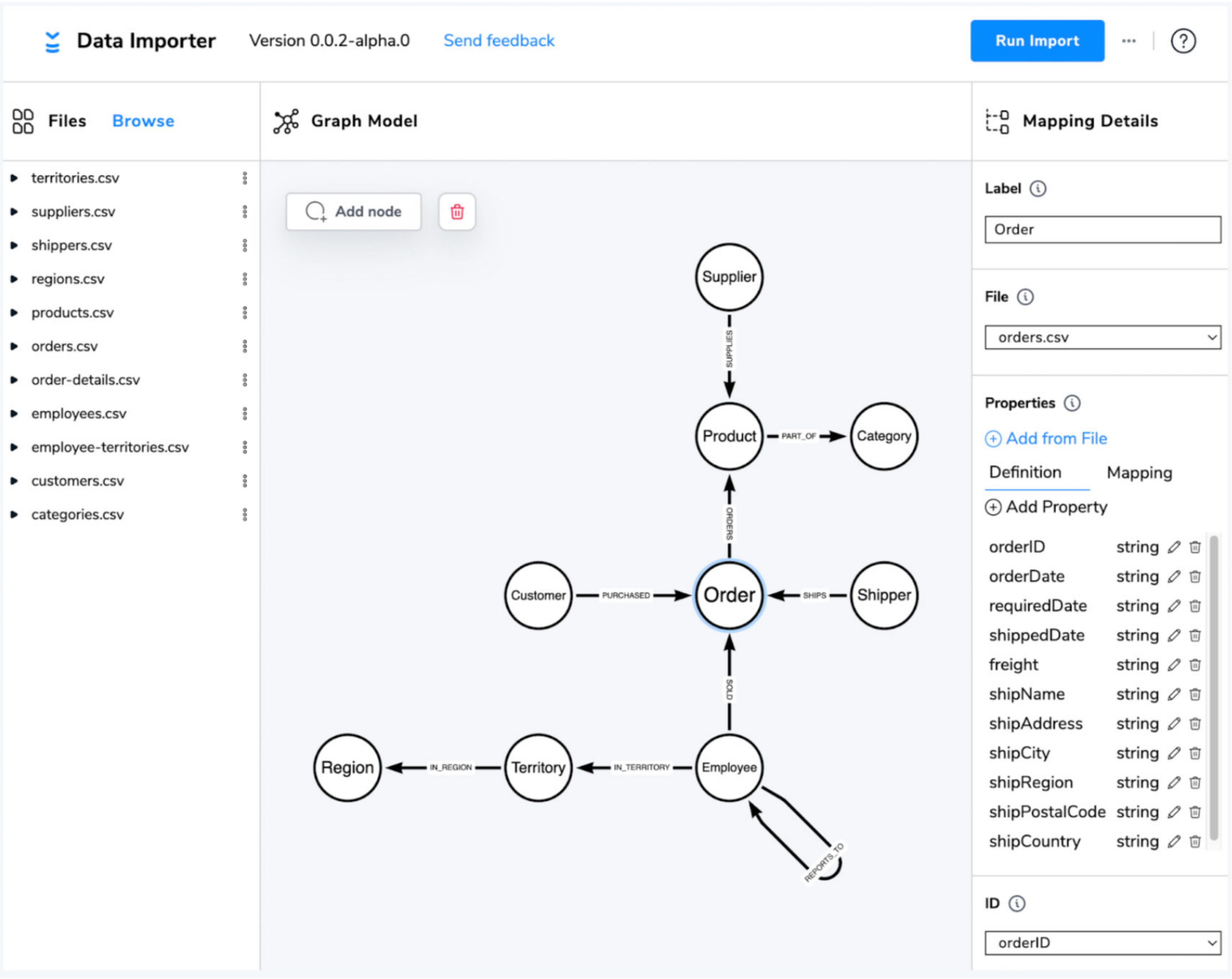
If you are new to Neo4j or completely new to graph, it could be hard to know where to start when it comes to loading data.

Neo4j has many data import options:

- `LOAD CSV` Cypher command for online import
- APOC procedures for other formats
- `neo4j-admin import` command to load a full database

These tools are capable, but you have to spend some time on learning them that can get in the way when you are just starting out.

To address this need, **Neo4j Data Importer** was created, a small but capable no-code user-friendly interface for loading flat file data into Neo4j databases.



Data Importer Version 0.0.2-alpha.0

Files

- territories.csv
- suppliers.csv
- shippers.csv
- regions.csv
- products.csv
- orders.csv
- order-details.csv
- employees.csv
- employee-territories.csv
- customers.csv
- categories.csv

Mapping Details

Label: Order

File: orders.csv

Properties

orderID	string
orderDate	string
requiredDate	string
shippedDate	string
freight	string
shipName	string
shipAddress	string
shipCity	string
shipRegion	string
shipPostalCode	string
shipCountry	string

ID: orderID

Today Neo4j Data Importer provides:

- Support for flat-file inputs (`.csv`, `.tsv`) in the region of 1 million total rows equivalent (no upper limit is enforced, but larger loads take longer and rely on a reliable network connection to your database).
- The ability to sketch out a graph model and map your input data to its structure and properties.

- Loads into any Neo4j database which are reachable from your machine.

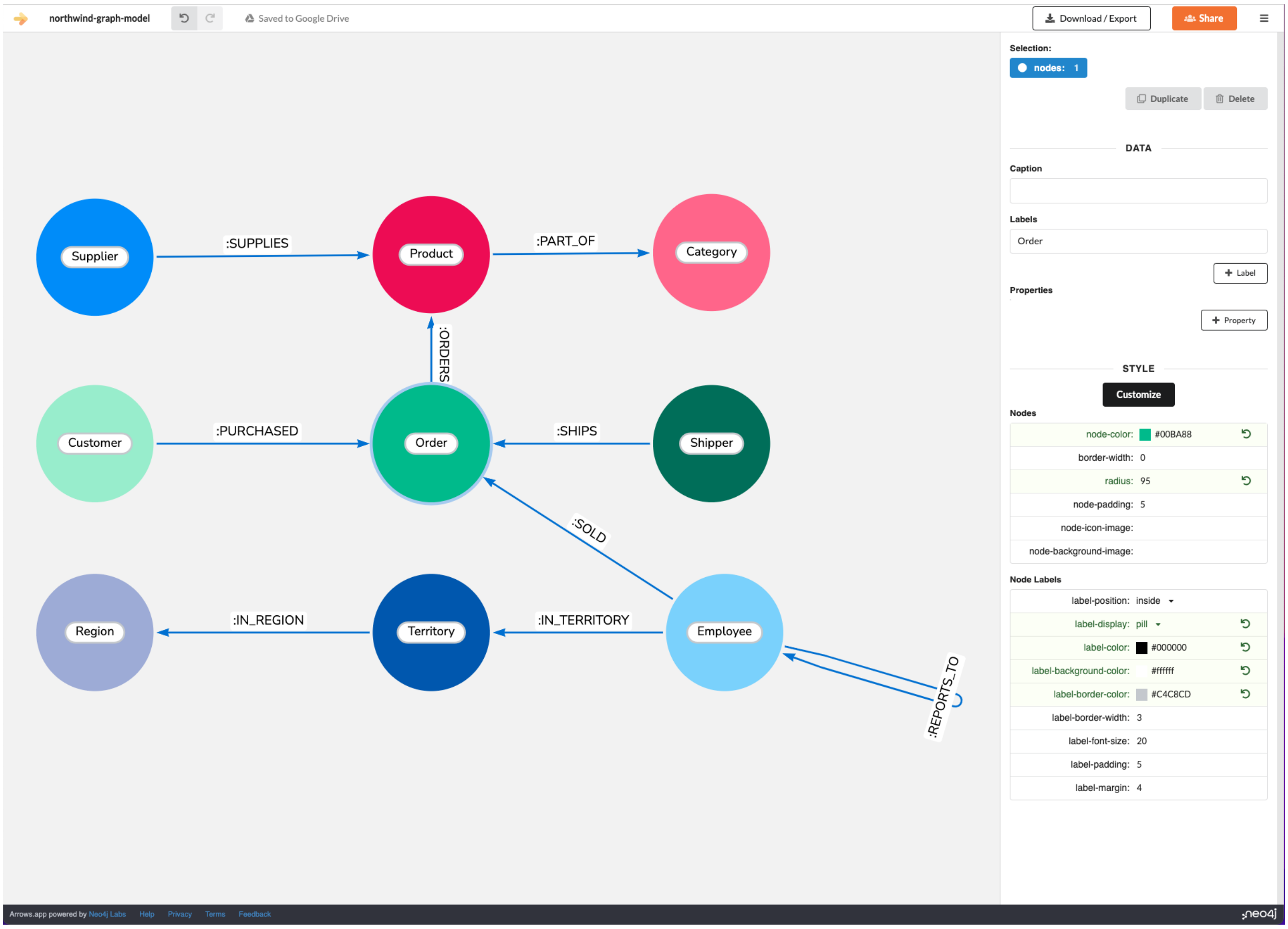
Data is only kept in your web browser; the tool does not use a server side component.

❗	Today Neo4j Data Importer is available on Neo4j AuraDB instances only.

Arrows.app

Arrows.app is a web-based tool from Neo4j Labs for drawing graphs.

While designing a graph data model, you may need to visualize different versions. Remember, there is no right or wrong way to structure your data for a graph database! Your domain model depends on your business needs and aims to solve your specific problems. Its flexibility allows you to refactor data model to improve performance and maximize capabilities of your graph database.



Selection:
nodes: 1

DATA

Caption:

Labels: Order

Properties

STYLE

Nodes

node-color: #00BA88
border-width: 0
radius: 95
node-padding: 5
node-icon-image:
node-background-image:

Node Labels

label-position: inside
label-display: pill
label-color: #000000
label-background-color: #ffffff
label-border-color: #C4C8CD
label-border-width: 3
label-font-size: 20
label-padding: 5
label-margin: 4

Graphs are easy and intuitive to sketch on a whiteboard (we say whiteboard-friendly), and Arrows.app tries to be just as easy and intuitive as using a whiteboard. You are able to create a visual layout of the organization’s data entities, relationships, and properties. Graph data model stays exactly as it was drawn on the whiteboard/Arrows.app canvas.

With the help of Arrows.app:

1. You can create, modify, and delete essential parts of the Neo4j graph database: nodes and relationships with their labels and properties.
2. You can convert your table into a graph by using Arrows.app import functions.

3. You are able to export images from Arrows.app as Cypher statements and load data into Neo4j database.

Arrows.app features:

- quick intuitive drawing with a mouse
- fine-grained styling control: sizes, layouts, and colors
- several options for exporting images
- functionality for importing JSON files or tables

Resources

- Blog Post. Aura Series: Discover AuraDB Free and Learn More about New Data Importer
- Blog Post. Neo4j DevTools' Happy New Year: New Data Importer and Neo4j Browser Updates
- Video. Drawing and Creating Graphs with Arrows.app
- Blog Post. Drawing graphs with Arrows.app

Data importing

The goal of the following articles and tutorials is to help you understand how to import various types of data into Neo4j. From JSON to APIs to another database, you can retrieve data from nearly any source and use it to populate your graph.

Importing CSV files

One of the most common formats of data is in rows and columns on flat files. This spreadsheet format is used for a variety of imports and exports to/from relational databases, so it is easy to retrieve existing data this way.

You can also use this format of data for Neo4j. The `LOAD CSV` command in Cypher allows us to specify a filepath, headers or not, different value delimiters, and the Cypher statements for how we want to model that tabular data in a graph.

We will walk through the details of how to take any CSV file and import the data into Neo4j.

Importing CSV data into Neo4j

Importing API data

There are now many data sources that use an API to expose data via a URL - many of these in JSON format. You can also import this type of data into Neo4j using the APOC standard extension library and executing the commands in the Neo4j Browser command line or in a script.

The `apoc.load.json` command allows us to specify a URL path and any necessary parameters, followed by Cypher statements to model that tree-like data in a graph.

This guide shows how to retrieve data from a JSON-based REST API and import it into Neo4j.

Importing API data

Importing data from a relational database to Neo4j

Many existing systems store data in relational or tabular types of formats. Knowing how to translate and migrate this data into graph data for analyzing the relationships can seem complex.

There are a variety of tools for migrating data from relational formats into graphs. In this article, we want to discuss all of the options and why you can or should choose some over others for your use case.

Import: RDBMS to graph

Tutorials

In the Appendix, you can find two tutorials on how to import data from the relational database and how to import CSV data with Neo4j Desktop.

The first guide uses a common relational data set (Northwind) and walks you through how to transform and import data from a relational database to Neo4j graph database. You will learn what steps are needed to retrieve the data from the relational data store and import the same data as a graph in Neo4j, as well as how to take the relational data model and convert it to graph in the process.

- Tutorial: Import data from a relational database into Neo4j
- How-To: Import CSV data with Neo4j Desktop

Importing CSV data into Neo4j

This article demonstrates different approaches to importing CSV data into Neo4j, and provides solutions to potential issues that might arise during the process.

CSV is a file of comma-separated values, often viewed in Excel or some other spreadsheet tool. There can be other types of values as the delimiter, but the most standard is the comma. Many systems and processes today already convert their data into CSV format for file outputs to other systems, human-friendly reports, and other needs. It is a standard file format that humans and systems are already familiar with using and handling.

Ways to import CSV files

There are a few different approaches to get CSV data into Neo4j, each with varying criteria and functionality. The option you choose depends on the dataset size, as well as your degree of comfort with various tools.

Let us see some of the ways Neo4j can read and import CSV files.

1. `LOAD CSV` Cypher command: this command is a great starting point and handles small- to medium-sized data sets (up to 10 million records). *Works with any setup, including AuraDB.*
2. The `neo4j-admin import` tool: command-line tool useful for straightforward loading of large datasets. *Works with Neo4j Desktop, Neo4j EE Docker image and local installations.*
3. Neo4j ETL tool: Neo4j Labs project. For more details and documentation, visit Neo4j ETL Tool page.
4. Kettle import tool: maps and executes steps for the data process flow and works well for very large datasets, especially if you are already familiar with using this tool. *Works with any setup, including AuraDB.*

In the below section, you will find a brief overview of the `LOAD CSV` Cypher command and `neo4j-admin import` command, how they operate, and how to get started with a general use case. Data quality can be an issue for any type of data import to any system. A few of these potential difficulties and ways to overcome them will, therefore, be covered in this section.

LOAD CSV command with Cypher

The `LOAD CSV` clause is a part of the Cypher query language. For more information about the `LOAD CSV` clause, see the Cypher Manual → LOAD CSV. It is widely applicable. `LOAD CSV` is not simply a basic data ingestion mechanism. It performs multiple actions in a single operation:

- Supports loading/ingesting CSV data from a URI.
- Directly maps input data into complex graph/domain structures.
- Handles data conversion.
- Supports complex computations.
- Creates or merges entities, relationships, and structures.

For better control, you can run the `LOAD CSV` command with Cypher Shell instead of in Neo4j Browser. For more information about Cypher Shell, see Operations Manual → Cypher shell.

Reading CSV files

`LOAD CSV` can handle local and remote files, and there is some syntax associated with each. This can be an easy thing to miss and result in an access error, so the rules are clarified here.

Local files may be loaded using a `file:///` prefix before the file name.
Since AuraDB is cloud based, this local file approach does not work with AuraDB.

Due to security reasons, local files, by default, can only be read from the Neo4j import directory, which differs depending on your operating system. File locations for each OS are listed in the Operations Manual → File locations. It is recommended to put files in Neo4j's *import* directory, as it keeps the environment secure. However, if you need to access files in other locations, you can find out which setting to alter in the Cypher Manual → LOAD CSV introduction. Default import folder paths are shown in this Knowledge Base article.

Examples

```
//Example 1 - file directly placed in import directory (import/data.csv)
LOAD CSV FROM "file:///data.csv"

//Example 2 - file placed in subdirectory within import directory (import/northwind/customers.csv)
LOAD CSV FROM "file:///northwind/customers.csv"
```

Web-hosted files can be referenced directly with their URL, like `https://host/path/data.csv`. However, permissions must be set so that an external source can read the file. For more information about access related to online file imports, see this Knowledge Base article.

Examples

```
//Example 1 - website
LOAD CSV FROM 'https://data.neo4j.com/northwind/customers.csv'

//Example 2 - Google
LOAD CSV WITH HEADERS FROM 'https://docs.google.com/spreadsheets/d/<yourFilePath>/export?format=csv'
```

Important tips for **LOAD CSV**

There are a few things to keep in mind with `LOAD CSV` and a few helpful tips for handling the variety of data scenarios you are likely to encounter.

- All data from the CSV file is read as a string, so you need to use `toInteger()`, `toFloat()`, `split()`, or similar functions to convert values.
- Check your Cypher import statement for typos. Labels, property names, relationship types, and variables are **case-sensitive**.
- The cleaner the data, the easier the load. Try to handle complex cleanup/manipulation before load.

Converting data values with **LOAD CSV**

Cypher has some scrubbing and conversion capabilities to help with data cleanup. These are extremely useful for handling missing data or splitting a field into multiple values for the graph.

First, remember that Neo4j does not store null values. Null or empty fields in a CSV files can be skipped or replaced with default values in `LOAD CSV`.

Suppose you have this CSV file:

companies.csv

```
Id,Name,Location,Email,BusinessType
1,Neo4j,San Mateo,contact@neo4j.com,P
2,AAA,,info@aaa.com,
3,BBB,Chicago,,G
```

The default location for CSV files for import is the **import** directory for your Neo4j instance.

Here are some examples of importing this data.

Examples

```
//skip null values
LOAD CSV WITH HEADERS FROM 'file:///companies.csv' AS row
WITH row WHERE row.Id IS NOT NULL
MERGE (c:Company {companyId: row.Id});

// clear data
MATCH (n:Company) DELETE n;

//set default for null values
LOAD CSV WITH HEADERS FROM 'file:///companies.csv' AS row
MERGE (c:Company {companyId: row.Id, hqLocation: coalesce(row.Location, "Unknown")})

// clear data
MATCH (n:Company) DELETE n;

//change empty strings to null values (not stored)
LOAD CSV WITH HEADERS FROM 'file:///companies.csv' AS row
MERGE (c:Company {companyId: row.Id})
SET c.emailAddress = CASE trim(row.Email) WHEN "" THEN null ELSE row.Email END
```

Next, if you have a field in the CSV that is a list of items that you want to split, you can use the Cypher `split()` function to separate arrays in a cell.

Suppose you have this CSV file:

employees.csv

```
Id,Name,Skills,Email
1,Joe Smith,Cypher:Java:JavaScript,joe@neo4j.com
2,Mary Jones,Java,mary@neo4j.com
3,Trevor Scott,Java:JavaScript,trevor@neo4j.com
```

Example

```
LOAD CSV WITH HEADERS FROM 'file:///employees.csv' AS row
MERGE (e:Employee {employeeId: row.Id, email: row.Email})
WITH e, row
UNWIND split(row.Skills, ':') AS skill
MERGE (s:Skill {name: skill})
MERGE (e)-[r:HAS_EXPERIENCE]->(s)
```

Conditional conversions can be achieved with `CASE`. You saw one example of this when we were checking for null values or empty strings, but let us look at another example.

Example

```
// clear data
MATCH (n:Company) DELETE n;

//set businessType property based on shortened value in CSV
LOAD CSV WITH HEADERS FROM 'file:///companies.csv' AS row
WITH row WHERE row.Id IS NOT NULL
WITH row,
(CASE row.BusinessType
 WHEN 'P' THEN 'Public'
 WHEN 'R' THEN 'Private'
 WHEN 'G' THEN 'Government'
 ELSE 'Other' END) AS type
MERGE (c:Company {companyId: row.Id, hqLocation: coalesce(row.Location, "Unknown")})
SET c.emailAddress = CASE trim(row.Email) WHEN "" THEN null ELSE row.Email END
SET c.businessType = type
RETURN *
```

Optimizing **LOAD CSV** for performance

Often, there are ways to improve performance during data load, which are especially helpful when dealing with large amounts of data or complex loading.

To improve inserting or updating unique entities into your graph (using `MERGE` or `MATCH` with updates), you can create indexes and constraints declared for each of the labels and properties you plan to merge or match on.

ℹ	For best performance, always `MATCH` and `MERGE` on a single label with the indexed primary-key property.

Suppose you use the preceding **companies.csv** file, and now you have a file that contains people and the companies they work for:

people.csv

```
employeeId,Name,Company
1,Bob Smith,1
2,Joe Jones,3
3,Susan Scott,2
4,Karen White,1
```

You should also separate node and relationship creation into separate processing. For instance, instead of the following:

```
MERGE (e:Employee {employeeId: row.employeeId})
MERGE (c:Company {companyId: row.companyId})
MERGE (e)-[r:WORKS_FOR]->(c)
```

You can write it like this:

```
// clear data
MATCH (n)
DETACH DELETE n;
// load Employee nodes
LOAD CSV WITH HEADERS FROM 'file:///people.csv' AS row
MERGE (e:Employee {employeeId: row.employeeId, name: row.Name})
RETURN count(e);
// load Company nodes
LOAD CSV WITH HEADERS FROM 'file:///companies.csv' AS row
WITH row WHERE row.Id IS NOT NULL
WITH row,
(CASE row.BusinessType
 WHEN 'P' THEN 'Public'
 WHEN 'R' THEN 'Private'
 WHEN 'G' THEN 'Government'
 ELSE 'Other' END) AS type
MERGE (c:Company {companyId: row.Id, hqLocation: coalesce(row.Location, "Unknown")})
SET c.emailAddress = CASE trim(row.Email) WHEN "" THEN null ELSE row.Email END
SET c.businessType = type
RETURN count(c);
// create relationships
LOAD CSV WITH HEADERS FROM 'file:///people.csv' AS row
MATCH (e:Employee {employeeId: row.employeeId})
MATCH (c:Company {companyId: row.Company})
MERGE (e)-[:WORKS_FOR]->(c)
RETURN *;
```

This way, the load is only doing one piece of the import at a time and can move through large amounts of data quickly and efficiently, reducing heavy processing.

When the amount of data being loaded is too much to fit into memory, there are a couple of different approaches you can use to combat running out of memory during the data load.

1. Batch the import into sections with `PERIODIC COMMIT`.

   This clause can be added before the `LOAD CSV` clause to tell Cypher to only process so many rows of the file before clearing memory and transaction state. For more information, see the Cypher Manual → PERIODIC COMMIT query hints.

   Example

   ```
   :auto USING PERIODIC COMMIT 500
   LOAD CSV WITH HEADERS FROM 'file:///data.csv' AS row
   ...
   ```

❗ PERIODIC COMMIT was deprecated in **Neo4j v4.4**. It is recommended to use CALL { … } IN TRANSACTIONS instead. For more information, see the Cypher Manual → Subqueries.

2. Avoid the Eager operator.

   Some statements pull in more rows than it is necessary, adding extra processing up front. To avoid this, you can run PROFILE on your queries to see if they use Eager loading and either modify queries or run multiple passes on the same file, so it does not do this. For more information about the Eager operator, see the Cypher Manual → Eager operator.

3. Adjust configuration for the database on heap and memory to avoid page-faults.

   To help handle larger volumes of transactions, you can increase some configuration settings for the database and restart the instance for them to take effect. Usually, you can create or update 1 million records in a single transaction per 2 GB of heap. In neo4j.conf:

   - dbms.memory.heap.initial_size and dbms.memory.heap.max_size: set to at least 4G.
   - dbms.memory.pagecache.size: ideally, value large enough to keep the whole database in memory.

LOAD CSV resources

- How-To: Import CSV data with Neo4j Desktop
- Cypher Manual: LOAD CSV
- Tutorial: Import relational data into Neo4j
- GraphAcademy: Importing CSV Data into Neo4j

The neo4j-admin import command for large datasets

LOAD CSV is great for importing small- or medium-sized data (up to 10 million records). For datasets larger than this, you can use the neo4j-admin import command. This allows you to import CSV data to an empty database by specifying node files and relationship files.

Suppose you want to import the below order data via neo4j-admin import into a Neo4j instance. Notice that some of the following CSV files include headers and some have separate header files. If you want to perform the import, you place them in the **import** folder for your Neo4j instance.

customers.csv

```
customerId:ID(Customer), name
23, Delicatessen Inc
42, Delicious Bakery
```

products.csv

```
productId:ID(Product), name, price, :LABEL
11,Chocolate,10,Product;Food
```

orders_header.csv

```
orderId:ID(Order),date,total,customerId:IGNORE
```

customer_orders_header.csv

```
:END_ID(Order),date:IGNORE,total:IGNORE,:START_ID(Customer)
```

orders1.csv

```
1041,2020-05-10,130,23
```

orders2.csv

```
1042,2020-05-12,20,42
```

order_details.csv

```
:START_ID(Order),amount,price,:END_ID(Product)
1041,13,130,11
1042,2,20,11
```

The tool is located in `<neo4j-instance-location>/bin/neo4j-admin` and you run the command in a terminal window where you have navigated to the **import** folder for your Neo4j instance.

Here is an example of importing the preceding CSV files in Neo4j 4.x. You must specify the name of the database. In this case we specify **orders**.

```
../bin/neo4j-admin import --database orders
     --nodes=Customer=customers.csv
     --nodes=products.csv
     --nodes=Order="orders_header.csv,orders1.csv,orders2.csv"
     --relationships=CONTAINS=order_details.csv
     --relationships=ORDERED="customer_orders_header.csv,orders1.csv,orders2.csv"
     --trim-strings=true
```

> You must specify the parameters to this script on a **single** line. Line feeds are shown here for readability.

When you run this command, it creates a new database named **orders**.

The repeated `--nodes` and `--relationships` parameters are groups of multiple (potentially split) CSV files of the same entity, i.e. with the same column structure.

All files per group are treated as if they could be concatenated as a single large file. A **header row** in the first file of the group or in a separate, single-line file is required. Placing the header in a separate file can make it easier to handle and edit than having it in a multi-gigabyte text file. Compressed files are also supported.

- The `--id-type=STRING` indicates that all `:ID` columns contain alphanumeric values (there is an optimization for numeric-only IDs).
- The `customers.csv` is imported directly as nodes with the `:Customer` label and the properties are taken

directly from the file.

- `Product` nodes follow the same pattern where the node labels are taken from the `:LABEL` column.
- The `Order` nodes are taken from three files - one header and two content files.
- Line item relationships typed `:CONTAINS` are created from `order_details.csv`, relating orders with the contained products via their IDs.
- Orders are connected to customers by using the order CSV files again, but this time with a different header, which :IGNORE's the non-relevant columns.

The column names are used for property-names of your nodes and relationships. There is specific markup on specific columns:

- `name:ID` - global id column used to look up the node later reconnecting.
  - if the property name is left off, it will be not stored (temporary), which is what the `--id-type` refers to.
  - if you have repeated IDs across entities, you have to provide the entity (id-group) in parentheses like `:ID(Order)`.
  - if your IDs are globally unique, you can leave that off.
- `:LABEL` - label column for nodes. Multiple labels can be separated by delimiter.
- `:START_ID`, `:END_ID` - relationship file columns referring to the node IDs. For id-groups, use `:END_ID(Order)`.
- `:TYPE` - column to specify relationship-type.
- All other columns are treated as properties but skipped if empty or annotated with `:IGNORE`.
- Type conversion is possible by suffixing the name with indicators like `:INT`, `:BOOLEAN`, etc.

For more details on this header format and the tool, see the Operations Manual → Neo4j admin import and the accompanying tutorial.

CSV data quality

Real-world data is messy. Any time you work with data, you will see some values that need cleaned up or transformed before you move it to another system. Small syntax errors, format descriptions, consistency, correct quoting, and even differing assumptions on data requirements or standards can easily cause hours of cleanup down the road.

We will highlight some of the data quality issues easily missed when loading data from other systems into Neo4j and try to help avoid problems with data import and cleanup.

Common pitfalls

- **Headers are inconsistent with data (missing, too many columns, different delimiter in header).**

  Verify headers match the data in the file. Adjusting formatting, delimiters, columns, etc. at this stage will save a great deal of time later.

- **Extra or missing quotes throughout file.**

  Standalone double or single quotes in the middle of non-quoted text or non-escaped quotes in quoted text can cause issues reading the file for loading. It is best to either escape or remove stray quotes. Documentation for proper escaping can be found in the Cypher Manual → style guide, and in this Knowledge Base article.

- **Special or Newline characters in file.**

  When dealing with any special characters in a file, ensure they are quoted or remove them. For newline characters in quoted or unquoted fields, either add quotes for these or remove them.

- **Inconsistent line breaks.**

  One thing that computers do not handle well is inconsistent data. Ensure line breaks are consistent throughout. We recommend choosing the Unix style for compatibility with Linux systems (common format for import tools).

- **Binary zeros, BOM byte order mark (2 UTF-8 bytes) at beginning of the file, or other non-text characters.**

  Any unusual characters or tool-specific formatting are sometimes hidden in application tools, but become easily apparent in basic editors. If you come across these types of characters in your file, it is best to remove them entirely.

Tools

As mentioned above, certain applications have special formatting to make documents look nice, but this hidden extra code is not handled by regular file readers and scripts. Other times, it is hard to find small syntax changes or make broad adjustments for files with a lot of data.

For handling these types of situations or general data cleanup, there are a number of tools that help you check and validate your CSV data files.

Basic tools, such as hexdump, vi, emacs, UltraEdit, and Notepad++ work well for handling shortcut-based commands for editing and manipulating files. However, there are also other more efficient or user-friendly options available that assist in data cleanup and formatting.

- Cypher - what Cypher sees is what will be imported, so you can use that to your advantage. Using `LOAD CSV` without creating a graph structure only outputs samples, counts, or distributions to make it possible to detect incorrect header column counts, delimiters, quotes, escapes, or header name spellings.
- CSVKit - a set of Python tools that provides statistics (csvstat), search (csvgrep), and more for your CSV files.
- CSVLint - an online service to validate CSV files. You can upload the file or provide an URL to load it.
- Papa Parse - a comprehensive Javascript library for CSV parsing that allows you to stream CSV data and provides good, human-readable error reporting on issues.

```
// assert correct line count
LOAD CSV FROM "file-url" AS line
RETURN count(*);

// check first 5 line-sample with header-mapping
LOAD CSV WITH HEADERS FROM "file-url" AS line
RETURN line
LIMIT 5;
```

Importing JSON data from a REST API into Neo4j

> *This article demonstrates some techniques for loading data from JSON-based REST APIs into Neo4j.*

Importing JSON data into Neo4j

There are a plethora of JSON-based Web APIs that we can import into Neo4j, and we can use one of the Load JSON procedures to retrieve data from these APIs and turn it into map values ready for Cypher to consume.

The APOC user guide provides a worked example showing how to import data from StackOverflow into Neo4j.

The Strava API

Strava is an application used by runners and cyclists to record their activities and share them with their friends. This data is available to users via a JSON-based REST API.

Before we start calling the API, we need to create an application. We will then be provided with an access token that we will need to use in all our requests to the API.

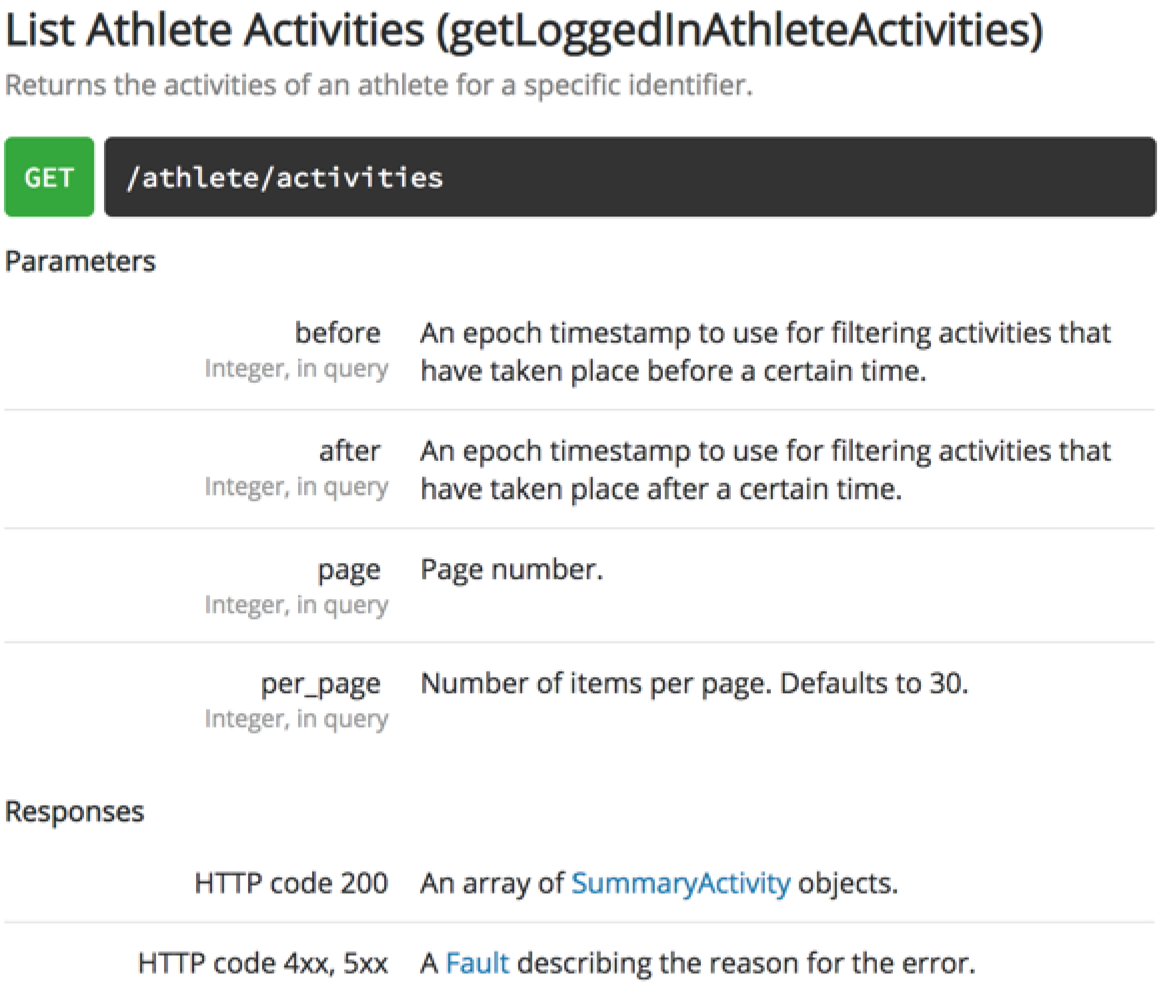
We can create a parameter in the Neo4j Browser or Cypher shell by executing the following command:

```
:params {stravaToken: "Bearer <insert-strava-token>"}
```

❗ Don't forget to replace `<insert-strava-token>` with the token for your Strava application.

Working with a paginated endpoint

We are interested in importing the activities for the athlete who is logged in. That endpoint takes the following parameters:



List Athlete Activities (getLoggedInAthleteActivities)

Returns the activities of an athlete for a specific identifier.

GET /athlete/activities

Parameters

before Integer, in query	An epoch timestamp to use for filtering activities that have taken place before a certain time.
after Integer, in query	An epoch timestamp to use for filtering activities that have taken place after a certain time.
page Integer, in query	Page number.
per_page Integer, in query	Number of items per page. Defaults to 30.

Responses

HTTP code 200	An array of SummaryActivity objects.
HTTP code 4xx, 5xx	A Fault describing the reason for the error.

We're interested in `per_page` (where we can define the number of activities returned per call to the endpoint) and `after` (where we can tell the API to only return results after a provided epoch timestamp).

Let's imagine that we have more activities that we can return in one request to the API. We'll need to paginate to retrieve all our activities and import them into Neo4j.

Before we paginate the API, let's first learn how to import one page worth of activities into Neo4j. The following query will return activities starting from the earliest timestamp:

```
WITH 0 AS after

WITH 'https://www.strava.com/api/v3/athlete/activities?after=' + after AS uri
CALL apoc.load.jsonParams(uri, {Authorization: $stravaToken}, null)
YIELD value

CREATE (run:Run {id: value.id})
SET run.distance = toFloat(value.distance),
    run.startDate = datetime(value.start_date_local),
    run.elapsedTime = duration({seconds: value.elapsed_time})
```

We create a node with the label `Run` for each activity and set a few properties, as well. The most interesting one for this example is `startDate` which we will pass to the `after` parameter later on.

This query will load the first 30 activities, but what if we want to get the next 30? We can change the first line of the query to find the most recent timestamp of any of our `Run` nodes and then pass that to the API. If there aren't any `Run` nodes, then we can use a value of 0 like in the query below.

```
OPTIONAL MATCH (run:Run)
WITH run ORDER BY run.startDate DESC LIMIT 1
WITH coalesce(run.startDate.epochSeconds, 0) AS after

WITH 'https://www.strava.com/api/v3/athlete/activities?after=' + after AS uri
CALL apoc.load.jsonParams(uri, {Authorization: $stravaToken}, null)
YIELD value

CREATE (run:Run {id: value.id})
SET run.distance = toFloat(value.distance),
    run.startDate = datetime(value.start_date_local),
    run.elapsedTime = duration({seconds: value.elapsed_time})
```

We could continue to run this query manually, but it's about time that we automated it.

Automated API pagination

One way to do this is by using a scripting language and creating a loop inside which we make calls to that endpoint until we run out of activities to retrieve. If we're a bit creative, we can achieve the same outcome with the `apoc.periodic.commit` procedure.

From the APOC documentation, this is the description of the periodic iterate procedure:

> It is useful to run a query repeatedly in separate transactions until it doesn't process and generates any results anymore. So you can iterate in batches over elements that don't fulfil a condition and update them so that they do afterwards.

In our case, the exit condition will be when we receive less than 30 activities from the API. Let's first update our query to return a value of `0` if less than 30 activities are returned and the actual count if it's 30.

```
OPTIONAL MATCH (run:Run)
WITH run ORDER BY run.startDate DESC LIMIT 1
WITH coalesce(run.startDate.epochSeconds, 0) AS after

WITH 'https://www.strava.com/api/v3/athlete/activities?after=' + after AS uri
CALL apoc.load.jsonParams(uri, {Authorization: $stravaToken}, null)
YIELD value

CREATE (run:Run {id: value.id})
SET run.distance = toFloat(value.distance),
    run.startDate = datetime(value.start_date_local),
    run.elapsedTime = duration({seconds: value.elapsed_time})

RETURN CASE WHEN count(*) < 30 THEN 0 ELSE count(*) END AS count
```

All that's left to do now is wrap the whole thing in periodic commit. We call `apoc.periodic.commit` method with two arguments:

- the first is the Cypher statement to run until the `RETURN` clause returns 0,
- the second are parameters that are passed to the Cypher statement.

```
call apoc.periodic.commit("
  OPTIONAL MATCH (run:Run)
  WITH run ORDER BY run.startDate DESC LIMIT 1
  WITH coalesce(run.startDate.epochSeconds, 0) AS after

  WITH 'https://www.strava.com/api/v3/athlete/activities?after=' + after AS uri
  CALL apoc.load.jsonParams(uri, {Authorization: $stravaToken}, null)
  YIELD value

  CREATE (run:Run {id: value.id})
  SET run.distance = toFloat(value.distance),
      run.startDate = datetime(value.start_date_local),
      run.elapsedTime = duration({seconds: value.elapsed_time})

  RETURN CASE WHEN count(*) < 30 THEN 0 ELSE count(*) END AS count
", {stravaToken: $stravaToken})
```

This query sends multiple commits to the API until we have loaded all our activities.

Resources

- APOC Documentation: StackOverflow JSON Data Example
- APOC Documentation: Load JSON

Import: RDBMS to graph

Importing data from a relational database

Often, when in a company setting, you have existing data in a system that will need transferred or manipulated for a new project. It is rare to have cases where some or all of the data for a new project is not already captured somewhere. In order to get existing data where you need it for the new process, application, or system, you will need to perform an extract-transform-load (ETL) process. Very simply, you will need to export data from the existing system(s), handle any necessary manipulations on the data for the new structure, and then import the transformed data to the new data store.

Depending on the particular environment you are working in, different tools for importing relational to graph may provide better or faster solutions than others. In this guide, we want to discuss all of the options and why you can or should choose some over others for your use case.

Relational to graph import tools

There are three main approaches to moving relational data to a graph. We will briefly cover how each operates on this page, but more detailed walkthroughs are in the linked pages.

1) LOAD CSV: possibly the simplest way to import data from your relational database. Requires a dump of individual entity-tables and join-tables formatted as CSV files.

2) APOC: Awesome Procedures on Cypher. Created as an extension library to provide common procedures and functions to developers. This library is especially helpful for complex transformations and data manipulations. Useful procedures include apoc.load.jdbc, apoc.load.json, and others.

3) ETL Tool: Neo4j Labs UI tool that translates relational to graph from a JDBC connection. Allows bulk

data import for large data sets with a fast performance and simple user experience.

4) Kettle: open-source tool for enterprise-scale data export and import. Handles a variety of data sources and large data sets easily and organizes the data flow process.

5) Other ETL tools: there are also a few vendor and community tools available for similar etl processes and GUI interaction for getting data in various formats into and out of Neo4j. Some of these tools also can map out the flow and transformation of data through the system.

6) Programmatic via drivers: ability to retrieve data from a relational database (or other tabular structure) and use the bolt protocol to write it to Neo4j through one of the drivers with your programming language of choice.

> You should create and understand your graph data model before transferring the data from an existing relational structure to a graph. If you do not have a good data model, then jumping into the import can cause frustration on data cleanup later.

`LOAD CSV`

This built-in Cypher function allows users to take existing or exported CSV files and load them into Neo4j with Cypher statements to read, transform, and import the data to the graph database. It allows the user to run statements individually or run them batched in a Cypher script. Because this functionality is provided in Cypher out-of-the-box, you do not need any additional plugins or configuration, and those already familiar with Cypher may prefer this route.

However, certain difficult or complex transformations may not be easily achievable or provided in Cypher. For those cases, you might need to add an `APOC` procedure to the `LOAD CSV` statements or use another import tool.

LOAD CSV resources

- Cypher Manual: LOAD CSV
- Guide: Importing CSV data into Neo4j
- Docs Tutorial: LOAD CSV for import

APOC

APOC is Neo4j’s utility library for handling data import, as well as data transformations and manipulations. From converting values to altering the data model, this library can manage it all, allowing you to combine and chain procedures in order to get exactly the results you are looking for.

For data import, APOC offers several options depending on your data source and format. It can import files or data from a URL in CSV, JSON, or XML formats, as well as loading data straight from a database (using JDBC). When you call these procedures, you can pass in the data source and use other procedures to manipulate data or regular Cypher to insert or update to the database. There are also procedures for batching data, adding wait/sleep commands, and handling large data sets or temperamental data sources.

The transformation procedures in this library are nearly endless, allowing the developer to process dynamic labels or relationships, correct/skip null or empty values, format dates or other values, generate hashes, and handle other tricky data scenarios. If you are in need of a way for flexible and custom data handling, `APOC` could be the way to go. The downside to using this library for complicated scenarios is that it may result in many lines of code to handle multiple data transformations.

APOC resources

- Documentation: APOC
- Videos: APOC Video Series
- Source code: Github project

ETL Tool

Neo4j's ETL tool provides a simple GUI that allows you to load data from nearly any type of relational database to a Neo4j instance. The process has you set up a JDBC connection to nearly any type of relational database, then does some auto-mapping to a graph data model rendered as a visualization that you can edit to your use case. Finally, you can choose whether the load occurs on a running or shutdown Neo4j instance and import the data.

This tool provides a simple, straightforward process for an initial import from a relational database to Neo4j quickly and efficiently. However, it does not provide the ability at this point in time to handle incremental loads or updates to existing data. It is a community-driven tool, so updates are made as needed and not on a scheduled timeline.

ETL Tool Resources

- Developer guide: Neo4j ETL Tool
- Blog post: Translating Relational Data to Graph
- Source code: Github project

Kettle

This highly diverse and flexible data loading tool has several connection options to and from Neo4j, as well as capabilities to generate CSV files from other systems to load into your graph database. Its goal is to help you create and manage a simple, self-describing, and maintainable data integration process from beginning to end.

Kettle builds a data loading process that is self-documenting and transparent. It is especially helpful if the data import requires data retrieval from multiple sources or if there are multiple dependent steps to build or update your graph. If you need to transformation the data coming in or going out, Kettle can handle different kinds of manipulations, including aggregations. Processes that need to log information to Neo4j or flexibility for embedding in various environments also make excellent cases for using Kettle.

All of this functionality is bundled out-of-the-box through a simple, yet powerful GUI for your ETL developers. Cooperation with Neo4j simply requires the plugins for our graph data integration.

Kettle Resoures

- Kettle Download: Open-source project on SourceForge
- Neo4j Plugins: Integrate Kettle with Neo4j
- Blog post: Getting Started with Kettle and Neo4j

Import programmatically with drivers

For importing data using a programming language, you can use the Neo4j driver for your preferred language and execute Cypher statements to/from the database. This process is also helpful if you do not have access to the Cypher shell or if the data is not available as an accessible file.

You can set up the driver connection to Neo4j, and then execute Cypher statements that pass from the application-level through the driver and to the database for various operations - including large amounts of inserts and updates. Using the driver and programming language can be very useful for incremental updates to data passed from other systems into Neo4j.

Driver import resources

- Blog post: Tips and Tricks for Fast-Batched Import with Neo4j

Neo4j Drivers

Neo4j language guides

This section is designed to provide detailed examples of how to integrate Neo4j with your preferred programming language. Neo4j **officially supports the drivers for .Net, Java, JavaScript, Go, and Python** for the binary Bolt protocol. Our community contributors provide drivers for all major programming languages for all protocols and APIs. In this section, we provide an introduction and a consistent example application for several languages and Neo4j drivers.

You should be familiar with graph database concepts and the property graph model. You should have created an Neo4j AuraDB cloud instance, or installed Neo4j locally.

How to connect to Neo4j?

If you've created an AuraDB instance, or installed and started Neo4j as a server on your system, you can already work interactively with the database via the built-in Neo4j Browser, which you find in the AuraDB console, or if you're running locally, on localhost:7474.

To build an application, you want to connect to Neo4j from your technology stack. Fortunately it is very easy using a driver which connects to Neo4j via Bolt or HTTP.

The binary Bolt Protocol

Starting with Neo4j 3.0 we support a binary protocol called Bolt. It is based on the PackStream serialization and supports the Cypher type system, protocol versioning, authentication and TLS via certificates. For Neo4j Clusters, Bolt provides smart client routing with load balancing and failover.

The binary protocol is enabled in Neo4j by default, so you can use any language driver that supports it.

Neo4j officially provides drivers for

- .NET
- Java
- Spring
- JavaScript
- Go
- Python

For more details on the protocol implementation, see the implementers documentation.

All Neo4j Drivers

Thanks to the Neo4j contributor community, there are additionally drivers for almost every popular programming language, most of which mimic existing database driver idioms and approaches. Get started with your stack now, see the dedicated page for more detail.

[check circle o Java]	[check circle o Spring]	[check circle o Neo4j-OGM]	[check circle o .NET]	[check circle o JavaScript]
[check circle o Python]	[check circle o Go]	Ruby	PHP	Erlang/Elixir
Perl	C/C++	Clojure	Haskell	R

If you are new to development, we recommend the Jetbrains IDE for a good developer experience, which also comes with a Neo4j Database plugin.

To demonstrate connection to and usage of Neo4j in different programming languages we've created an example application. It is a simple, one-page webapp, that uses Neo4j's movie demo database (movie, actor, director) as data set. The same front-end web page in all applications consumes three REST endpoints provided by backend implemented in the different programming languages and drivers:

- movie search by title
- single movie listing
- graph visualization of the domain

GitHub

The source code for all the different language examples is available on GitHub as individual repositories that can be cloned and directly used as starting points.

Using the HTTP API

> The HTTP API is available in Community Edition and Enterprise Edition, but not in AuraDB.

If you want to access Neo4j programmatically, you can also use the HTTP-API which allow you to:

- POST one or more Cypher statements with parameters.
- Keep transactions open over multiple requests.
- Choose different result formats.

Let's look at one of the underlying remote API endpoint that Neo4j offers to run queries. These APIs can be then used directly via a HTTP library or a driver for your language.

A simple HTTP Cypher request, executable in the Neo4j Browser, would look like this:

```
:POST /db/<database-name>/tx/commit {"statements":[
      {"statement":"CREATE (p:Person {firstName: $name}) RETURN p",
       "parameters":{"name":"Daniel"}}
    ]}
```

Some of the language drivers use the HTTP API under the hood, but make them available in a more convenient way.

Using Neo4j from Java

> *If you are a Java developer, this guide provides an overview of options for connecting to Neo4j. While this guide is not comprehensive, it introduces the different APIs and links to the relevant resources.*

> *You should be familiar with graph database concepts and the property graph model. You should have created an Neo4j AuraDB cloud instance, or installed Neo4j locally. When developing with Neo4j, please use Java 8 or 11 and your favorite IDE.*

Neo4j for Java developers

Neo4j provides drivers which allow you to make a connection to the database and develop applications which create, read, update, and delete information from the graph.

For Community Edition and Enterprise Edition, you can also extend Neo4j by implementing **user defined procedures for Cypher** in Java or other JVM languages.

The Example Project

The Neo4j example project is a small, one page webapp for the movies database built into the Neo4j tutorial. The front-end page is the same for all drivers: movie search, movie details, and a graph visualization of actors and movies. Each backend implementation shows you how to connect to Neo4j from each of the different languages and drivers.

You can learn more about our small, consistent example project across many different language drivers here. You will find the implementations for all drivers as individual GitHub repositories, which you can clone and deploy directly.

Neo4j Java Driver

The Neo4j Java driver is **officially supported** by Neo4j and connects to the database using the binary protocol. It aims to be minimal, while being idiomatic to Java. We support Java 8 and 11 for the driver.

When using Maven, add this to your *pom.xml* file:

```
<groupId>org.neo4j.driver</groupId>
<artifactId>neo4j-java-driver</artifactId>
<version>4.4</version>
```

For Gradle or Grails, this is how to add the dependency:

```
compile 'org.neo4j.driver:neo4j-java-driver:4.4'
```

For other build systems, see information available at Maven Central.

```
link:https://github.com/neo4j/neo4j-java-
driver/raw/4.4/examples/src/main/java/org/neo4j/docs/driver/HelloWorldExample.java[tag=hello-world-import]

link:https://github.com/neo4j/neo4j-java-
driver/raw/4.4/examples/src/main/java/org/neo4j/docs/driver/HelloWorldExample.java[tag=hello-world]
```

Driver configuration

From Neo4j version **4.0** and onwards, the default encryption setting is **off** and Neo4j no longer generates self-signed certificates. This applies to default installations, installations through Neo4j Desktop and Docker images. You can verify the encryption level of your server by checking the `dbms.connector.bolt.enabled` key in `neo4j.conf`.

Table 19. Table Scheme Usage

Certificate Type	Neo4j Causal Cluster	Neo4j Standalone Server	Direct Connection to Cluster Member
Unencrypted	`neo4j`	`neo4j`	`bolt`
Encrypted with Full Certificate	`neo4j+s`	`neo4j+s`	`bolt+s`
Encrypted with Self-Signed Certificate	`neo4j+ssc`	`neo4j+ssc`	`bolt+ssc`
Neo4j AuraDB	`neo4j+s`	N/A	N/A

Review your SSL Framework settings when going into production. If necessary, you can also generate certificates for Neo4j with Letsencrypt

Name	Version	Authors
neo4j-java-driver	4.4	The Neo4j Team
Package	Example	Neo4j Online Community
Docs	API	[github Source]

Spring Data Neo4j

> *For Java developers who use the Spring Framework or Spring Boot and want to take advantage of reactive development principles, this guide introduces Spring integration*

through the Spring Data Neo4j project. The library provides convenient access to Neo4j including object mapping, Spring Data repositories, conversion, transaction handling, reactive support, and more.

- Familiarity with graph database concepts and the property graph model.
- Create an AuraDB Free instance and familiarity with the Cypher query language
- Some knowledge/experience with Spring. Knowing Spring Data and Spring Boot are both great additions to your toolbox, as well.
- For this library, please use JDK 11 or later and your favorite IDE.

Reactive Development

Neo4j (version 4.0+) incorporated the principles of the reactive manifesto for passing data between the database and client with the drivers. Developers can take advantage of the reactive approach to process queries and return results. This means that communication between the driver, and the database can be managed and adjusted dynamically according to data needs of the client.

Reactive programming principles allow the consuming side (applications and other systems) to specify the amount of data received within a certain window of time. Neo4j's database driver will also maintain rate limits for requesting data from the server, providing flow control throughout the entire Neo4j stack.

No matter the volume of transactions or data (even during times of high activity), the system can maintain limits on how much it can send and receive at once based on available resources. This prevents overloads and collapses or failures, as well as lost transmissions or later catch up loads during the downtime.

Project Reactor is the core foundation of many implementations of reactive development, including Spring's. Neo4j uses the Spring implementation of Project Reactor components to provide reactive support in related applications with the graph database.

Spring Data Neo4j

The Spring Data Neo4j 6 is the new major version of the Spring Data Neo4j project. One of its feature benefit is the capability and support for reactive transactions, though there are other improvements and additions such as fully immutable entity and Java record-based mapping support.

While SDN provides both imperative and reactive application development, this guide will focus on the reactive implementation. Imperative application code and documentation in SDN is available on the Github project.

We can see some of the most prominent features and changes in the SDN library listed below.

Features

- Support for both imperative and reactive application development
- Lightweight mapping with built-in OGM (object graph mapping) library

- Immutable entities (for both Java and Kotlin languages)
- New Neo4j client and reactive client feature for template-over-driver architecture

SDN has full support for the well-known and understood imperative programming model (much like Spring Data JDBC or JPA). It also provides full support for the newer reactive programming based on Reactive Streams, including reactive transactions. Both functionalities are included in the same binary.

> ℹ The reactive programming model requires a 4.0+ Neo4j instance (previous versions do not support reactive drivers) and reactive Spring on the application side.

One key difference of SDN 6 from the previous version of Spring Data Neo4j is that the OGM (object-graph mapping) layer is no longer a separate library. Instead, the Spring Data infrastructure now handles OGM's functionality.

Getting started

Over the next few sections, we will walk through all of the steps for creating a reactive application.

Prepare the database

For this example, we will use the Neo4j-standard movie graph data set because it comes for free with every Neo4j instance and is a small size.

If you haven't already, download Neo4j Desktop and create/start a database.

You can interact with the database and load the data in a web browser with the URL http://localhost:7474. Note the command ready to run in the prompt (`:play movies`). Execute that command, and an interactive slidedeck will appear just below the command line. On the second slide of that guide, execute the long Cypher statement to fill your database with our movie test data.

Create a new Spring Boot project

The easiest way to set up a Spring Boot project is with the Spring Initializr at start.spring.io. It is also integrated in the major IDEs, in case you prefer not to use the website.

Then, you can change the default group, artifact, name, and description for the project. Next, we can choose our project dependencies. We can search for and add the `Neo4j` and `Spring Reactive Web` starter to get what we need to create a reactive, Spring-based web application.

Once those steps are complete, we can click the `Generate` button at the bottom to create the skeleton for our project and download it. The Spring Initializr will take care of creating the project structure for you, with the basic files and settings in place for the selected build tool.

Other dependencies

If you are looking at the project in Github, you might notice that there are some other dependencies in the `pom.xml`. A couple are for adding tests to the project, then one dependency for developer tools, and a couple more for test containers.

More information on the testing functionality can be found in the documentation.

Testing and dev tools dependencies

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-test</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>1.15.3</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>neo4j</artifactId>
    <version>1.15.3</version>
    <scope>test</scope>
</dependency>
```

Adding configurations

Now, we need to add a few configurations to connect to the database. We can find the `application.properties` file and configure what we need.

```properties
spring.neo4j.uri=neo4j+s://abcd.databases.neo4j.io
spring.neo4j.authentication.username=neo4j
spring.neo4j.authentication.password=secret
```

> ℹ You need to adjust the password to whatever you set when you created your instance of Neo4j.

The first three lines are our Neo4j database URI and credentials. The username and password you enter here should match for your individual database. This is the bare minimum of what you need to connect to a Neo4j instance.

We do not need to add any other configuration for the driver, thanks to the Spring Boot Driver autoconfiguration provided out of the box with SDN 6.

Other configurations

Logging

There is also one additional property we could define. It is not a required property, but does allow us to see the Cypher statements and see better insight into what is running behind our application.

```properties
logging.level.org.springframework.data.neo4j=DEBUG
```

Database selection

Since version 4.0, Neo4j is multi-tenant. We can statically select the database by providing a property:

```
spring.data.neo4j.database = my-database
```

For more advanced use cases, it is possible to perform a dynamic selection, as documented here.

Create the domain

With our project dependencies defined and configurations set, we are ready to start defining our entities for our data domain! The domain layer should accomplish two things:

1. Map the graph to objects.
2. Provide access to those objects.

Our data contains movie and person entities that show how people were involved in various films, such as who acted in, directed, wrote, produced, etc. We will need to define a domain class for each of our entities - `Movie` and `Person`.

> SDN supports all data types that the Neo4j Java Driver supports. To find out how to map Neo4j types to native language types, see this section in the documentation.

Movie entity

```
@Node("Movie")
public class MovieEntity {
    @Id
    private final String title;
    @Property("tagline")
    private final String description;
    @Relationship(type = "ACTED_IN", direction = INCOMING)
    private Set<PersonEntity> actors = new HashSet<>();
    @Relationship(type = "DIRECTED", direction = INCOMING)
    private Set<PersonEntity> directors = new HashSet<>();
    public MovieEntity(String title, String description) {
        this.title = title;
        this.description = description;
    }
    //Getters omitted for brevity
}
```

In the first line, the `@Node` annotation is used to mark the class as a managed entity. It also configures the Neo4j label, which defaults to the name of the class, but you can define a custom one, as well.

The first couple of lines inside the class definition sets up the id field of the entity as the `title` attribute. The title is a unique business key in this domain, but if you don't have a unique key in another domain, you can use the combination of `@Id` and `@GeneratedValue` annotations on a field to generate a unique technical key. There are also generators provided for UUIDs.

The two lines below those set up the `tagline` (or `description`) property. The `@Property` annotation is used as a way for mapping a different name for the field than for the graph property. This way, you can map differences between application entities and database domains.

At the next annotation, the @Relationship defines a relationship between the movie and person entities with an ACTED_IN type for showing which persons acted in a particular movie. The two lines below that define another relationship between MovieEntity and PersonEntity for those who directed movies.

Then, the next code block defines a constructor for the entity with the properties of the node (title and description).

As mentioned above, you can use SDN with Kotlin and model your domain with Kotlin's data classes. Project Lombok is also available to shortcut definitions and boilerplate, if you want or need to stay purely within Java.

Person entity

```
@Node("Person")
public class PersonEntity {
    @Id
    private final String name;
    private final Integer born;
    public PersonEntity(Integer born, String name) {
        this.born = born;
        this.name = name;
    }
    //Getters omitted
}
```

This class for person entities looks very similar to our MovieEntity class above. The @Node annotation defines that it is a database domain entity. A unique key field is identified (in this case, the name property), and a born property is defined as another attribute on this class. The constructor for the class follows the properties.

Notice that we have not defined the relationships from a person back to a movie. In our use case, we only want to retrieve movies and the people involved in them. Our application does not need us to pull information for person entities separately, so we do not need to define the relationships back in the other direction.

If a domain needs to pull related entities on both sides, we would need to add the annotations and attributes from both sides.

Define a Spring Data repository

Our repositories in the application will extend a repository provided out-of-the-box called the ReactiveNeo4jRepository.

If building an imperative application, you can extend the Neo4jRepository. Also, while technically not prohibited, it is not recommended or supported to mix imperative and reactive database access in the same application.

Because our repositories are implementing reactive capabilities, we have access to the Mono and Flux reactive types from Project Reactor for method returns. The Mono type returns 0 or 1 results, while the Flux returns 0 or n results. We would use a return type of Mono if we were expecting a single object back from the query and use a Flux type if we were expecting potentially multiple objects back from the query.

Movie repository

```
public interface MovieRepository extends ReactiveNeo4jRepository<MovieEntity, String> {
    Mono<MovieEntity> findOneByTitle(String title);
}
```

For our application, we need to interact with a Neo4j graph database, so we will create an interface that extends the repository for Neo4j.

Since we want to use the reactive features for the application, we will extend the ReactiveNeo4jRepository, which provides reactive, Neo4j-specific implementation details on top of several extended Spring repositories. The ReactiveNeo4jRepository requires two types to be specified — our class type and its id type. Once we add our MovieEntity and String (our movie id field is the title) values here, we can start defining methods we want to use.

Inside the interface definition, there is one method we will define for findOneByTitle(). This method will let us search the database based on a movie title, and we expect to see a single movie return or none at all for the movie we are interested in.

To get that 0 or 1 return result, we can use the reactive return type of Mono<MovieEntity>. We will also pass a title (a String) to the method because we want to allow the user to enter any movie title as the search value.

Person repository

While there is a PersonRepository interface in the Github code, it serves testing purposes for that application, so we will not go into detail on it here. More information on testing in SDN with this application is in the documentation.

However, it does demonstrate using a custom query and the Flux return type, so it may be of interest as an example or for a template for other applications.

Setting up the controllers

With the repository, we have our methods for accessing movie data in our database. Let us now define endpoints allowing users to access those methods and query the database.

The controller acts as the messenger between the data layer and the user interface to accept requests from the user and return responses. This is where the code logic and data manipulation is typically placed, coordinating different responses based on the kind of input it receives.

Because our use case scope is interested in movies, we only need to create a controller to access movie data.

MovieController.java

```
@RestController
@RequestMapping("/movies")
public class MovieController {
    private final MovieRepository movieRepository;
    public MovieController(MovieRepository movieRepository) {
        this.movieRepository = movieRepository;
    }
    //method implementations with walkthroughs below
}
```

First, we need to have a couple of annotations to declare this as a controller for REST requests (`@RestController`) and map requests to controller methods for a certain path (`@RequestMapping` with an endpoint of `/movies`).

Within our class definition, we start by injecting our repository interface and creating a constructor for it. This gives us access to the data layer from our repository interface and domain class.

Now we need to add more code to define endpoints and implement our data methods.

```
@PutMapping
Mono<MovieEntity> createOrUpdateMovie(@RequestBody MovieEntity newMovie) {
    return movieRepository.save(newMovie);
}
```

Up first is the implementation for `createOrUpdateMovie()`. We start with a `@PutMapping` annotation to specify a put request (overwrite or replace an object). We want to specify a single movie to overwrite or create, so we use the return type of `Mono` and pass in the movie object with all of its expected fields. Within the method, we will save that new or updated movie by calling the movie repository's `save()` method.

Now, if you scroll back up to our defined `MovieRepository` interface above, you may notice that we did not define a save method there. This is because Spring Data repositories provide a few default methods for us out-of-the-box. Methods for `save()`, `findAll()`, etc are methods that nearly every application wants or needs, so Spring provides them, and we do not have to implement those basic methods each time we create data access.

Let us add another method to our controller for `getMovies()`.

```
@GetMapping(value = { "", "/" }, produces = MediaType.TEXT_EVENT_STREAM_VALUE)
Flux<MovieEntity> getMovies() {
    return movieRepository.findAll();
}
```

The `@GetMapping` annotation tells us we are only retrieving data from the database and not modifying or inserting. We have two parameters for the annotation, where we pass any additional depth on the url path (in this case, no additional depth - just `/movies`) and that we want to return a text event stream. This is our media type because we are expecting a `Flux` of results (0 to n amount), and we want to return those as they come in (reactive stream), rather than aggregating and returning all the results at once (imperative json object). Just like our previous method, we call the movie repository and access an out-of-the-box `findAll()` method to return all of the movies in our database.

The next method is the one we defined in our `MovieRepository` interface.

```
@GetMapping("/by-title")
Mono<MovieEntity> byTitle(@RequestParam String title) {
    return movieRepository.findOneByTitle(title);
}
```

The starting `@GetMapping` specifies a subpath of `/by-title`. Since we are searching for a single movie where the user will input a title as the search string, we expect 0 or 1 result back with the type `Mono` and pass the user-defined parameter of the movie's title into the method. In the return, we call the movie repository again and access our defined `findOneByTitle()` method, passing in the search title.

For the last method definition, we want to allow users to delete a movie from our database.

```
@DeleteMapping("/\{id\}")
Mono<Void> delete(@PathVariable String id) {
    return movieRepository.deleteById(id);
}
```

We use the `@DeleteMapping` annotation and specify the subpath endpoint as `/movies/{id}` (where id stands for the id of the movie we want to delete). We only want one movie to be deleted at a time, and we don't expect an object to return (since it will be deleted and no longer in the database), so we specify the `Mono<Void>` as the return type. The method is defined and passes in a path variable (where user input defines the url path) for the id of the movie to delete, then calls the movie repository with the out-of-the-box `deleteById()` method and the movie id.

Running the application

With all of our code in place, we should be ready to build and run our application and try out the endpoints we set up! We can run the application (from a menu option in our IDE or from the command line) and then either open a web browser or command line to interact with the endpoints. For this example, we will show how to interact from the command line perspective.

Either way you connect, we will use the `localhost:8080/movies` path to access the `findAll()` method and retrieve all movies in our database, and then add any defined subpaths to drill down into other methods. We can hit each of these endpoints shown below and verify everything is working as expected.

Interacting from a command line

Here is the syntax for each of the endpoints from a command line:

- `localhost:8080/movies` for getMovies() method

```
curl http://localhost:8080/movies
```

Results: retrieve all movies in our database

- `localhost:8080/movies <movieToUpdateOrCreate>` for createOrUpdateMovie() method

```
curl -X "PUT" "http://localhost:8080/movies" \
     -H 'Content-Type: application/json; charset=utf-8' \
     -d $'{
  "title": "Aeon Flux",
  "description": "Reactive is the new cool"
}'
```

Results: create new movie `Aeon Flux` in our database

- `localhost:8080/movies/by-title` for byTitle() method

```
curl http://localhost:8080/movies/by-title\?title\=Aeon%20Flux
```

Results: retrieve information about the specific movie (in this query, `Aeon Flux`)

- `localhost:8080/movies/{id}` for delete() method

```
curl -X DELETE http://localhost:8080/movies/847
```

Results: delete the movie using its id (in this case, the `Aeon Flux` movie)

Resources

[code fork] Projects	Spring Data Neo4j
[github] Source	https://github.com/spring-projects/spring-data-neo4j
Issues	GitHub Issues
Docs	Reference, JavaDoc, ChangeLog
Articles	Introducing SDN 6
Examples	SDN Example from Spring, Movies Application with SDN, Migration Example from SDN 5/OGM to SDN 6

Quarkus, Helidon, Micronaut

> *For Java developers who use Quarkus, Helidon or Micronaut and want to take advantage of a pre-configured Java driver instance. This page should give an overview of the existing support for the driver in other Java frameworks. Please consult the linked documentations for more information.*

Driver integration

The goal with all three integrations is to provide support for getting a managed instances of the Neo4j driver. Like in the Spring Framework, you can provide the driver properties to an *application.properties* file (or yaml) to configure your application. In the end you will have an injectable driver instance that can be used with

```
@Inject
Driver driver;
```

in the business operation code base.

Additional to the managed driver bean creation, the integrations also expose health metrics for the driver and connection to your Neo4j instance.

Quarkus

In an existing Quarkus application you need to add the `quarkus-neo4j` dependency to your project.

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-neo4j</artifactId>
</dependency>
```

Additionally configure the basic connection parameters as needed.

Quarkus application.properties

```
quarkus.neo4j.uri = bolt://localhost:7687
quarkus.neo4j.authentication.username = neo4j
quarkus.neo4j.authentication.password = secret
```

If you want to make use of the health check, the additional `quarkus-smallrye-health` dependency is needed.

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-smallrye-health</artifactId>
</dependency>
```

For metrics support, you would either need *MicroMeter* (recommended by Quarkus) or *SmallRye Metrics* (only if you really need MicroProfile specification) dependencies declared.

MicroMeter (Prometheus) dependency

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-micrometer-registry-prometheus</artifactId>
</dependency>
```

The metrics for Neo4j have to be manually enabled in the *application.properties*.

```
quarkus.neo4j.pool.metrics-enabled = true
```

Helidon

In a Helidon-based application you need to declare the Neo4j Java driver dependency in your Maven *pom.xml*.

```
<dependency>
    <groupId>io.helidon.integrations.neo4j</groupId>
    <artifactId>helidon-integrations-neo4j</artifactId>
    <version>${helidon.version}</version>
</dependency>
```

Providing the essential connection parameters will give you a managed instance of the Java driver.

Helidon application.properties

```
neo4j.uri = bolt://localhost:7687
neo4j.authentication.username = neo4j
neo4j.authentication.password = secret
# Enable metrics
neo4j.pool.metricsEnabled = true
```

If you want to use the health and metrics system, you have to also declare those dependencies provided by the Helidon framework.

```
<dependency>
        <groupId>io.helidon.integrations.neo4j</groupId>
        <artifactId>helidon-integrations-neo4j-health</artifactId>
        <version>${helidon.version}</version>
    </dependency>
    <dependency>
        <groupId>io.helidon.integrations.neo4j</groupId>
        <artifactId>helidon-integrations-neo4j-metrics</artifactId>
        <version>${helidon.version}</version>
</dependency>
```

Now you can put together the configuration

Configuration with metrics and health

```
Neo4JSupport neo4j = Neo4JSupport.builder()
        .config(config)
        .helper(Neo4JMetricsSupport.create())
        .helper(Neo4JHealthSupport.create())
        .build();

Routing.builder()
        .register(health)
        .register(metrics)
        .register(movieService)
        .build();
```

and get the managed driver bean.

Micronaut

To enable the Neo4j Driver support in Micronaut, the *micronaut-neo4j-bolt* dependency needs to get declared.

```
<dependency>
    <groupId>io.micronaut.neo4j</groupId>
    <artifactId>micronaut-neo4j-bolt</artifactId>
</dependency>
```

Adding the needed connection parameters to the *application.properties*.

Micronaut application.properties

```
neo4j.uri = bolt://localhost:7687
neo4j.username = neo4j
neo4j.password = secret
```

The module will automatically add its information to the built-in */health* endpoint.

Resources

Quarkus Documentation	Neo4j integration, Configuration properties, Guide
Helidon Documentation	Reference, Helidon Neo4j
Micronaut Documentation	Neo4j integration, Guide
Examples	Quarkus, Helidon, Micronaut examples

Neo4j - OGM Object Graph Mapper

> *For Java developers that require a simple mechanism to manage their domain objects with Neo4j, this guide introduces the Neo4j Object Graph Mapping (OGM) library. This is the same library that powers Spring Data Neo4j but can also be used independently of Spring as well.*

> *You should be familiar with graph database concepts and the property graph model. You should have created an Neo4j AuraDB cloud instance, or installed Neo4j locally When developing with Neo4j, please use JDK 8 and your favorite IDE.*

Neo4j-OGM

While using the Cypher query language alongside the Java driver works well for small projects and more advanced queries, the boilerplate of CRUD operations often gets tedious.

That’s where object graph mappers come in; they take your existing domain objects (POJOs) and map them to Neo4j. The great benefit with this approach is that with graph databases, there is no data-model impedence mismatch like there is with relational databases.

Besides the Neo4j-OGM, there is also Hibernate-OGM and others. If you want more abstraction Spring Data Neo4j can give you a Domain Driver Design (DDD) approach.

We created the Neo4j-OGM as a standalone, simple object graph mapper for Java which also acts as the base for Spring Data Neo4j.

The Neo4j-OGM uses Cypher in a consistent manner under the hood and can communicate via several transport protocols - bolt, http and embedded.

Features

The Neo4j-OGM supports the features you would expect:

- Object graph mapping of annotated node- and relationship-entities
- Neo4jSession for direct interaction with Neo4j
- Fast class metadata scanning
- Optimized management of data loading and change tracking for minimal data transfers
- Multiple transports: binary (bolt), HTTP and embedded
- Persistence lifecycle events
- Query result projection to DTOs

Minimal Code Snippet

This code example is taken from the Example Project (see below).

```
@NodeEntity(label="Film")
public class Movie {

   @GraphId Long id;

   @Property(name="title")
   private String name;
}

@NodeEntity
public class Actor extends DomainObject {

   @Id @GeneratedValue
   private Long id;

   @Property(name="name")
   private String fullName;

   @Relationship(type="ACTED_IN", direction=Relationship.OUTGOING)
   private List<Role> filmography;

}

@RelationshipEntity(type="ACTED_IN")
public class Role {
    @Id @GeneratedValue  private Long relationshipId;
    @Property            private String title;
    @StartNode           private Actor actor;
    @EndNode             private Movie movie;
}
```

The Example Project

The Neo4j example project is a small, one page webapp for the movies database built into the Neo4j tutorial. The front-end page is the same for all drivers: movie search, movie details, and a graph visualization of actors and movies. Each backend implementation shows you how to connect to Neo4j from each of the different languages and drivers.

You can learn more about our small, consistent example project across many different language drivers here. You will find the implementations for all drivers as individual GitHub repositories, which you can clone and deploy directly.

FAQ

https://raw.githubusercontent.com/neo4j/neo4j-ogm/master/neo4j-ogm-docs/src/main/asciidoc/appendix/faq.adoc

Resources

👤 Authors	The Neo4j and GraphAware Teams
🎁 Package	http://maven.org
[github] Source	https://github.com/neo4j/neo4j-ogm
📘 Docs	https://neo4j.com/docs/ogm-manual/current/
📘 Article	http://neo4j.com/blog/neo4j-java-object-graph-mapper-released/
▶ Example-University	https://github.com/neo4j-examples/neo4j-ogm-university

https://raw.githubusercontent.com/neo4j/neo4j-ogm/master/neo4j-ogm-docs/src/main/asciidoc/reference/getting-started.adoc

Procedures and Functions

User-Defined Procedures and Functions

User Defined **Procedures and Functions** are available within Cypher and encapsulate dedicated functionality.

Just by annotating methods of a Java class and deploying the resulting jar file into your Neo4j installation, you can make new functionality easily available within the query language.

To implement your procedures or functions you would use the Neo4j Embedded Java API. Besides an object-oriented API to the graph database, working with `Node`, `Relationship`, and `Path` objects, it also offers highly customizable, high-speed traversal- and graph-algorithm implementations.

We don’t provide code examples for the Java API on this page, because they are covered in detail in the Java developers manual.

Neo4j uses that functionality itself for built-in procedures for meta-data, cluster-, query- and user-management and more.

Several libraries already provide new capabilities using procedures and functions, here is an example from the APOC library.

```
MATCH (start:City {name: 'Berlin'}),(end:City {name: 'Malmö'})
CALL apoc.algo.dikjstra(start, end, "ROUTE","distance") yield path, weight
RETURN path
ORDER BY weight ASC LIMIT 10
```

To get you started we provided a template project and documentation in the Java developer manual.

Third-party libraries

Neo4j Community Drivers

Members of the each programming language community have invested a lot of time and love to develop each one of the community drivers for Neo4j, so if you use any one of them, please provide feedback to the authors.

> ℹ The community drivers have been graciously contributed by the Neo4j community. Many of them are fully featured and well-maintained, but some may not be. Neo4j does not take any responsibility for their usability.

Using Neo4j Server with JDBC

Being a Java developer you're probably familiar with JDBC as a way to work with relational databases, either directly or via abstractions like Spring's `JDBCTemplate` or MyBatis. Many tools for ETL, Report Generators, Business Intelligence, and data management also use JDBC drivers to interact with relational databases.

As Cypher, like SQL, is a textual parameterizable query language that can return tabular results, it was possible for us to support the JDBC APIs and make a Neo4j-JDBC driver available.

The driver supports the new binary "Bolt" protocol of Neo4j 3.x, the transactional http endpoint and Neo4j embedded connections.

```
// Connect
Connection con = DriverManager.getConnection("jdbc:neo4j:bolt://localhost");

// Querying
try (Statement stmt = con.createStatement()) {
    ResultSet rs = stmt.executeQuery("MATCH (n:User) RETURN n.name");
    while (rs.next()) {
        System.out.println(rs.getString("n.name"));
    }
}
con.close();
```

More details on how to use it can be found with our example project for Java-JDBC. There, we implement the backend as a minimal Java webapp that leverages Neo4j-JDBC to connect to Neo4j Server.

We tested the driver with many enterprise tools to ensure that it works in a variety of environments. Just grab the JAR of the latest release and add it to your tool of choice.

Authors	Developers from Larus BA Italy and Neo4j
Package	https://github.com/neo4j-contrib/neo4j-jdbc/releases/latest
[github] Source	https://github.com/neo4j-contrib/neo4j-jdbc
Docs	https://github.com/neo4j-contrib/neo4j-jdbc/blob/master/README.adoc
Example	https://github.com/neo4j-examples/movies-java-jdbc
Blog Post	http://neo4j.com/blog/couchbase-jdbc-integrations-neo4j-3-0/

JCypher

JCypher provides seamlessly integrated Java access to Neo4j at different levels of abstraction:

At the topmost level of abstraction, JCypher allows you to **map complex business domains** to graph databases. You can take an arbitrarily complex graph of domain objects or POJOs and store it in a straightforward way into Neo4j for later retrieval. You do not need to modify your domain object classes in any way, there is also no need for annotations. JCypher provides a default mapping out of the box.

At the same level of abstraction, **Domain Queries** provide the power and expressiveness of queries on a graph database, while being formulated on domain objects or on types of domain objects, respectively. The true power of Domain Queries comes from the fact, that the graph of domain objects is backed by a graph database.

At the next lower level of abstraction – access to graph databases – is provided based on a generic graph model. While simple, the model allows you to easily navigate and manipulate graphs.

At the bottom level of abstraction, a **native Java DSL** in the form of a fluent Java API allows you to intuitively and comfortably formulate queries against Neo4j.

Author	Wolfgang Schützelhofer
Package	http://maven.org
[github] Source	http://github.com/Wolfgang-Schuetzelhofer/jcypher
Docs	http://jcypher.iot-solutions.net/
Article	https://neo4j.com/blog/jcypher-focus-on-your-domain-model-not-how-to-map-it-to-the-database/

Groovy & Grails: Neo4j Grails Plugin

The goal of GORM for Neo4j is to provide a 'as-complete-as-possible' GORM implementation that maps domain classes and instances to the Neo4j nodespace. The following features are supported:

- Marshalling from Neo4j Nodes to Groovy types and back again
- Support for GORM dynamic finders, criteria and named queries
- Session-managed transactions

- Access to Neo4j’s traversal capabilities
- Access the Neo4j graph database via the Bolt driver

Authors	Stefan Armbruster, Graeme Rocher
Package	http://www.grails.org/plugin/neo4j
[github] Source	https://github.com/grails/grails-data-mapping/tree/master/grails-datastore-gorm-neo4j
Docs	http://grails.github.io/grails-data-mapping/latest/neo4j/

Scala: neotypes

Scala lightweight, type-safe, asynchronous driver (not opinionated on side-effect implementation) for neo4j. The project aims to provide seamless integration with most popular scala infrastructures such as lightbend (Akka, Akka-http, Lagom, etc), typelevel (cats, http4s, etc), twitter (finch, etc)…

Author	Dmitry Fedosov
[github] Source	https://github.com/neotypes/neotypes
Docs	https://neotypes.github.io/neotypes/docs.html
Article	http://dimafeng.com/2018/12/27/neotypes-1/
Example	https://github.com/neotypes/examples

JPA: Hibernate OGM

Hibernate Object/Grid Mapper (OGM) with Neo4j Support.

Authors	Davide D’Alto, Gunnar Moelling, Emmanuel Bernard
Package	http://maven.org
[github] Source	https://github.com/hibernate/hibernate-ogm/tree/master/neo4j
Docs	http://docs.jboss.org/hibernate/ogm/5.0/reference/en-US/html_single/#ogm-neo4j
Article	Blog, JPL Queries
Example	https://github.com/TimmyStorms/hibernate-ogm-neo4j-example

Using Neo4j from .NET

> *If you are a .NET developer, this guide provides an overview of options for connecting to Neo4j. While this guide is not comprehensive it will introduce the different drivers and link to the relevant resources.*

Prerequisites

> *You should be familiar with graph database concepts and the property graph model. You should have created an Neo4j AuraDB cloud instance, or installed Neo4j locally.*

Neo4j for .NET Developers

Neo4j provides drivers which allow you to make a connection to the database and develop applications which create, read, update, and delete information from the graph.

Neo4jDotNetDriver

The Neo4j .NET driver is **officially supported** by Neo4j and connects to the database using the binary protocol. It aims to be minimal, while being idiomatic to .NET.

Please note that the following example makes use of the `Neo4j.Driver.Simple` package that implements a blocking interface around the 'main' driver. This is to aid in clarity and simplicity for those just starting out with the Neo4j .Net driver.

```
PM> Install-Package Neo4j.Driver.Simple-4.4.0
```

For real projects, the `Neo4j.Driver` package should be used instead.

```
PM> Install-Package Neo4j.Driver-4.4.0
```

```
Unresolved directive in languages-guides/neo4j-dotnet.adoc - include::https
://raw.githubusercontent.com/neo4j/neo4j-dotnet-driver/{dotnet-driver-
version}/Neo4j.Driver/Neo4j.Driver.Tests.Integration/Examples.cs[tag=hello-world,indent=0]
```

Driver Configuration

From Neo4j version **4.0** and onwards, the default encryption setting is **off** by default and Neo4j will no longer generate self-signed certificates. This applies to default installations, installations through Neo4j Desktop and Docker images. You can verify the encryption level of your server by checking the `dbms.connector.bolt.enabled` key in `neo4j.conf`.

Table 20. Table Scheme Usage

Certificate Type	Neo4j Causal Cluster	Neo4j Standalone Server	Direct Connection to Cluster Member
Unencrypted	`neo4j`	`neo4j`	`bolt`
Encrypted with Full Certificate	`neo4j+s`	`neo4j+s`	`bolt+s`
Encrypted with Self-Signed Certificate	`neo4j+ssc`	`neo4j+ssc`	`bolt+ssc`
Neo4j AuraDB	`neo4j+s`	N/A	N/A

Review your SSL Framework settings when going into production. If necessary, you can also generate certificates for Neo4j with Letsencrypt

Name	Version	Authors
Neo4jDotNetDriver	4.4.0	The Neo4j Team, Charlotte Skardon, Martin Jensen
NuGet Package	.NET Example, .NET core example	Neo4j Online Community
Docs	API	[github Source]

The Example Project

The Neo4j example project is a small, one page webapp for the movies database built into the Neo4j tutorial. The front-end page is the same for all drivers: movie search, movie details, and a graph visualization of actors and movies. Each backend implementation shows you how to connect to Neo4j from each of the different languages and drivers.

You can learn more about our small, consistent example project across many different language drivers here. You will find the implementations for all drivers as individual GitHub repositories, which you can clone and deploy directly.

Neo4j Community Drivers

Members of the each programming language community have invested a lot of time and love to develop each one of the community drivers for Neo4j, so if you use any one of them, please provide feedback to the authors.

> The community drivers have been graciously contributed by the Neo4j community. Many of them are fully featured and well-maintained, but some may not be. Neo4j does not take any responsibility for their usability.

Neo4jClient

A .NET client for Neo4j, which makes it easy to write Cypher queries in C# with IntelliSense. It also supports basic CRUD and legacy indexing.

[github] Source	https://github.com/DotNet4Neo4j/neo4jclient
NuGet Package	https://nuget.org/packages/neo4jclient
Authors	Charlotte Skardon Tatham Oddie
Docs	https://github.com/DotNet4Neo4j/Neo4jClient/wiki
Example	https://github.com/neo4j-examples/movies-dotnet-neo4jclient
Protocol	Bolt, Http

Neo4j.Driver.Extensions

`Neo4j.Driver.Extensions` provides a set of extension methods to the official driver API, aiming at reducing boilerplate and easing mapping to entity classes.

[github] Source	https://github.com/DotNet4Neo4j/Neo4j.Driver.Extensions
NuGet Package	https://nuget.org/packages/neo4j.driver.extensions
Authors	Charlotte Skardon
Docs	Introduction blogpost

Neo4j from JavaScript

> *If you are a JavaScript developer, this guide provides an overview of options for connecting to Neo4j. While this guide is not comprehensive it introduces the different drivers and links to the relevant resources.*
>
> *You should be familiar with graph database concepts and the property graph model. You should have created an Neo4j AuraDB cloud instance, or installed Neo4j locally.*

Neo4j for JavaScript Developers

Neo4j provides drivers which allow you to make a connection to the database and develop applications which create, read, update, and delete information from the graph.

You can use the official driver for JavaScript (neo4j-driver) or connect via HTTP using the `request` module or any of our community drivers.

Learn with GraphAcademy Discrete.ad

[badge] | *//graphacademy.neo4j.com/courses/app-nodejs/badge/*

Building Neo4j Applications with Node.js Discrete

In this free course, we walk through the steps to integrate Neo4j into your Node.js projects. You will learn about the Neo4j JavaScript Driver, how sessions and transactions work and how to query Neo4j from an existing application.

Learn more

Neo4j Javascript Driver

The Neo4j JavaScript driver is **officially supported** by Neo4j and connects to the database using the binary protocol. It aims to be minimal, while being idiomatic to JavaScript, allowing you to subscribe to a stream of responses, errors and completion events.

```
npm install neo4j-driver
```

```
const neo4j = require('neo4j-driver')

    const driver = neo4j.driver(uri, neo4j.auth.basic(user, password))
    const session = driver.session()
    const personName = 'Alice'

    try {
      const result = await session.run(
        'CREATE (a:Person {name: $name}) RETURN a',
        { name: personName }
      )

      const singleRecord = result.records[0]
      const node = singleRecord.get(0)

      console.log(node.properties.name)
    } finally {
      await session.close()
    }

    // on application exit:
    await driver.close()
```

Driver Configuration

From Neo4j version **4.0** and onwards, the default encryption setting is **off** by default and Neo4j will no longer generate self-signed certificates. This applies to default installations, installations through Neo4j Desktop and Docker images. You can verify the encryption level of your server by checking the `dbms.connector.bolt.enabled` key in `neo4j.conf`.

Table 21. Table Scheme Usage

Certificate Type	Neo4j Causal Cluster	Neo4j Standalone Server	Direct Connection to Cluster Member
Unencrypted	`neo4j`	`neo4j`	`bolt`
Encrypted with Full Certificate	`neo4j+s`	`neo4j+s`	`bolt+s`
Encrypted with Self-Signed Certificate	`neo4j+ssc`	`neo4j+ssc`	`bolt+ssc`
Neo4j AuraDB	`neo4j+s`	N/A	N/A

Please review your SSL Framework settings when going into production. If necessary, you can also generate certificates for Neo4j with Letsencrypt

Name	Version	Authors
neo4j-driver	4.4.7	The Neo4j Team
Package		Neo4j Online Community
Docs	API	[github Source]

The Example Project

The Neo4j example project is a small, one page webapp for the movies database built into the Neo4j tutorial. The front-end page is the same for all drivers: movie search, movie details, and a graph visualization of actors and movies. Each backend implementation shows you how to connect to Neo4j from each of the different languages and drivers.

You can learn more about our small, consistent example project across many different language drivers here. You will find the implementations for all drivers as individual GitHub repositories, which you can clone and deploy directly.

Neo4j Community Drivers

Members of the each programming language community have invested a lot of time and love to develop each one of the community drivers for Neo4j, so if you use any one of them, please provide feedback to the authors.

> ℹ The community drivers have been graciously contributed by the Neo4j community. Many of them are fully featured and well-maintained, but some may not be. Neo4j does not take any responsibility for their usability.

js2neo

As a demonstration of a minimal JavaScript based bolt driver, Nigel Small put together js2neo.

Using the HTTP-Endpoint directly

You can use something as simple as the `request` node-module to send queries to and receive responses from Neo4j. The endpoint protocol and formats are explained in detail in the Neo4j Manual. It enables you do to much more, e.g. sending many statements per request or keeping transactions open across multiple requests.

Here is a very simple example:

Simple Function Accessing the Remote Endpoint

```
var r=require("request");
var txUrl = "http://localhost:7474/db/data/transaction/commit";
function cypher(query,params,cb) {
  r.post({uri:txUrl,
          json:{statements:[{statement:query,parameters:params}]}},
         function(err,res) { cb(err,res.body)})
}
```

Running the Function

```
var query="MATCH (n:User) RETURN n, labels(n) as l LIMIT {limit}"
var params={limit: 10}
var cb=function(err,data) { console.log(JSON.stringify(data)) }

cypher(query,params,cb)

{"results":[
  {"columns":["n","l"],
   "data":[
     {"row":[{"name":"Aran"},["User"]]}
    ]
  }],
 "errors":[]}
```

Using Neo4j from Python

> *This guide provides an overview of how to connecting to Neo4j from Python.*

You should be familiar with graph database concepts and the property graph model. You should have created an Neo4j AuraDB cloud instance, or installed Neo4j locally.

Neo4j and Python

Neo4j provides drivers which allow you to make a connection to the database and develop applications which create, read, update, and delete information from the graph.

Learn with GraphAcademy Discrete.ad

[badge] | *//graphacademy.neo4j.com/courses/app-python/badge/*

Building Neo4j Applications with Python Discrete

In this free course, we walk through the steps to integrate Neo4j into your Python projects. You will learn about the Neo4j Python Driver, how sessions and transactions work and how to query Neo4j from an existing application.

Learn more

Neo4j Python Driver

The Neo4j Python driver is **officially supported** by Neo4j and connects to the database using the binary protocol. It aims to be minimal, while being idiomatic to Python.

!	Support for Python 2 was removed in the 2.0 release of the driver.

```
pip install neo4j
```

```
Unresolved directive in languages-guides/neo4j-python.adoc - include::https://github.com/neo4j/neo4j-
python-driver/raw/4.4.5/tests/integration/examples/test_hello_world_example.py[tag=hello-world-import]

Unresolved directive in languages-guides/neo4j-python.adoc - include::https://github.com/neo4j/neo4j-
python-driver/raw/4.4.5/tests/integration/examples/test_hello_world_example.py[tag=hello-world,indent=0]
```

Driver configuration

From Neo4j version **4.0** and onwards, the default encryption setting is **off** by default and Neo4j will no longer generate self-signed certificates. This applies to default installations, installations through Neo4j Desktop and Docker images. You can verify the encryption level of your server by checking the `dbms.connector.bolt.enabled` key in `neo4j.conf`.

Table 22. Table Scheme Usage

Certificate Type	Neo4j Causal Cluster	Neo4j Standalone Server	Direct Connection to Cluster Member
Unencrypted	`neo4j`	`neo4j`	`bolt`
Encrypted with Full Certificate	`neo4j+s`	`neo4j+s`	`bolt+s`
Encrypted with Self-Signed Certificate	`neo4j+ssc`	`neo4j+ssc`	`bolt+ssc`
Neo4j AuraDB	`neo4j+s`	N/A	N/A

Please review your SSL Framework settings when going into production. If necessary, you can also generate certificates for Neo4j with Letsencrypt

Name	Version	Authors
neo4j-driver	4.4.5	The Neo4j Team
Package		Neo4j Online Community
Docs	</> API	[github Source]

The Example project

The Neo4j example project is a small, one page webapp for the movies database built into the Neo4j tutorial. The front-end page is the same for all drivers: movie search, movie details, and a graph visualization of actors and movies. Each backend implementation shows you how to connect to Neo4j from each of the different languages and drivers.

You can learn more about our small, consistent example project across many different language drivers here. You will find the implementations for all drivers as individual GitHub repositories, which you can clone and deploy directly.

Neo4j Community drivers

Members of the each programming language community have invested a lot of time and love to develop each one of the community drivers for Neo4j, so if you use any one of them, please provide feedback to the authors.

> The community drivers have been graciously contributed by the Neo4j community. Many of them are fully featured and well-maintained, but some may not be. Neo4j does not take any responsibility for their usability.

For anyone working with Python, the Neo4j community have contributed a range of driver options. These range from lightweight to comprehensive driver packages as well as libraries designed for use with web frameworks such as Django.

While Python 3 is preferred, some drivers still support Python 2, thus check with the individual project if you need it.

While we do not provide a specific web framework recommendation, both the lightweight Flask and the more comprehensive Django frameworks are known to work well.

Py2neo

Py2neo is a client library and comprehensive toolkit for working with Neo4j from within Python applications and from the command line. It has been carefully designed to be easy and intuitive to use.

Author	Nigel Small
Package	https://pypi.python.org/pypi/py2neo
[github] Source	https://github.com/technige/py2neo
Example	https://github.com/neo4j-examples/movies-python-py2neo
Docs	http://py2neo.org/
[code fork] Python	2.7 / 3.4+
Protocols	Bolt, Http

Installation

```
pip install py2neo
```

Example

```
from py2neo import Graph
graph = Graph()

tx = graph.begin()
for name in ["Alice", "Bob", "Carol"]:
    tx.append("CREATE (person:Person name: $name) RETURN person", name=name)
alice, bob, carol = [result.one for result in tx.commit()]
```

Neomodel

[neomodel.200x80] | *https://dist.neo4j.com/wp-content/uploads/neomodel.200x80.png*

An Object Graph Mapper built on top of the Neo4j python driver. Familiar Django style node definitions with a powerful query API, thread safe and full transaction support. A Django plugin django_neomodel is also available.

Author	Athanasios Anastasiou and Robin Edwards
Package	https://pypi.python.org/pypi/neomodel
[github] Source	http://github.com/neo4j-contrib/neomodel
Docs	https://neomodel.readthedocs.io/en/latest/
[code fork] Python	2.7 / 3.3+
Protocols	Bolt
Example App	https://github.com/neo4j-examples/neo4j-movies-python-neomodel

Installation

```
pip install neomodel
```

Example

```
from neomodel import StructuredNode, StringProperty, RelationshipTo, RelationshipFrom, config

config.DATABASE_URL = 'bolt://neo4j:test@localhost:7687'

class Book(StructuredNode):
    title = StringProperty(unique_index=True)
    author = RelationshipTo('Author', 'AUTHOR')

class Author(StructuredNode):
    name = StringProperty(unique_index=True)
    books = RelationshipFrom('Book', 'AUTHOR')

harry_potter = Book(title='Harry potter and the..').save()
rowling =  Author(name='J. K. Rowling').save()
harry_potter.author.connect(rowling)
```

Resources

- The Neo4j Python Driver Manual
- Blog post: Python 4.0 driver
- Blog Post: Neo4j for Django Developers: From zero to deployment with Django-Neomodel and Neo4j's Paradise Papers dataset
- Blog Post: Running Django-Neomodel within a Docker Container

Using Neo4j from Go

> *If you are a Go developer, this guide provides an overview of options for connecting to Neo4j. While this guide is not comprehensive it will introduce the different drivers and link to the relevant resources.*

You should be familiar with graph database concepts and the property graph model. You should have created an Neo4j AuraDB cloud instance, or installed Neo4j locally.

Neo4j for Go Developers

Neo4j provides drivers which allow you to make a connection to the database and develop applications which create, read, update, and delete information from the graph.

Neo4j Go Driver

The Neo4j Go driver is **officially supported** by Neo4j and connects to the database using the binary bolt protocol. It aims to be minimal, while being idiomatic to Go.

Module version

Make sure your application has been set up to use go modules (there should be a go.mod file in your application root). Add the driver with:

```
go get github.com/neo4j/neo4j-go-driver/v4
```

If you need to pin a specific 4.x version, you can run this instead:

```
go get github.com/neo4j/neo4j-go-driver/v4@<4.x tag>
```

where `<4.x tag>` is one of the available tag (e.g.: `v4.2.4`).

```
Unresolved directive in languages-guides/neo4j-go.adoc - include::https
://raw.githubusercontent.com/neo4j/neo4j-go-driver/v4.4.3/neo4j/test-
integration/examples_test.go[tag=hello-world,indent=0]
```

Driver Configuration

From Neo4j version **4.0** and onwards, the default encryption setting is **off** by default and Neo4j will no longer generate self-signed certificates. This applies to default installations, installations through Neo4j Desktop and Docker images. You can verify the encryption level of your server by checking the `dbms.connector.bolt.enabled` key in `neo4j.conf`.

Table 23. Table Scheme Usage

Certificate Type	Neo4j Causal Cluster	Neo4j Standalone Server	Direct Connection to Cluster Member
Unencrypted	`neo4j`	`neo4j`	`bolt`
Encrypted with Full Certificate	`neo4j+s`	`neo4j+s`	`bolt+s`
Encrypted with Self-Signed Certificate	`neo4j+ssc`	`neo4j+ssc`	`bolt+ssc`

Certificate Type	Neo4j Causal Cluster	Neo4j Standalone Server	Direct Connection to Cluster Member
Neo4j AuraDB	`neo4j+s`	N/A	N/A

Please review your SSL Framework settings when going into production. If necessary, you can also generate certificates for Neo4j with Letsencrypt

Name	Version	Authors
neo4j-driver	4.4.3	The Neo4j Team
Neo4j Online Community	Docs	API

The Example Project

The Neo4j example project is a small, one page webapp for the Movies database built into the Neo4j tutorial. The front-end page is the same for all drivers: movie search, movie details, and a graph visualization of actors and movies. Each backend implementation shows you how to connect to Neo4j from each of the different languages and drivers.

You can learn more about our small, consistent example project across many different language drivers here. You can find the implementations for all drivers as individual GitHub repositories, which you can clone and deploy directly.

Neo4j Community Drivers

Members of the each programming language community have invested a lot of time and love to develop each one of the community drivers for Neo4j, so if you use any one of them, please provide feedback to the authors.

> ℹ The community drivers have been graciously contributed by the Neo4j community. Many of them are fully featured and well-maintained, but some may not be. Neo4j does not take any responsibility for their usability.

GoGM: Golang Object Graph Mapper

Author	Eric Solender, CTO and co-founder of Mindstand
[github] Source	https://github.com/mindstand/gogm
Docs	https://github.com/mindstand/gogm/blob/master/README.md

Using Neo4j from Ruby

> *This section provides an overview of options for connecting to Neo4j using Ruby. It introduces the different drivers and links to relevant resources.*

Neo4j Community Drivers

Members of the each programming language community have invested a lot of time and love to develop each one of the community drivers for Neo4j, so if you use any one of them, please provide feedback to the authors.

> The community drivers have been graciously contributed by the Neo4j community. Many of them are fully featured and well-maintained, but some may not be. Neo4j does not take any responsibility for their usability.

Neo4j for Rubyists

Neo4j has been accessible from Ruby long before version 1.0, first using JRuby on the JVM and then on MRI via the HTTP and now Bolt protocols.

Neo4j.rb

The Neo4j.rb project is made up of the following Ruby gems:

neo4j-ruby-driver

A neo4j driver for ruby with an api consistent with the official drivers. It is based on seabolt and ffi. Available on all rubies (including jruby) and all platforms supported by seabolt.

neo4j-java-driver

A neo4j driver for ruby based on the official java implementation. It provides a thin wrapper over the java driver (only in jruby).

activegraph

A Object-Graph-Mapper (OGM) for the Neo4j graph database. It tries to follow API conventions established by ActiveRecord but with a Neo4j flavor. It requires one of the above drivers.

neo4j-rake_tasks

A set of rake tasks for installing and managing a Neo4j database within your project.

[browser] Website	http://neo4jrb.io/
Authors	Heinrich, Amit, Brian, Chris, Andreas
Package	neo4j-ruby-driver, neo4j-java-driver, activegraph
[github] Source	https://github.com/neo4jrb
Example	https://github.com/neo4j-examples?q=movies-ruby
Docs	http://neo4jrb.readthedocs.org/en/latest/
Tutorial	Ruby Tutorial

Blog	http://blog.brian-underwood.codes/
Protocols	bolt

History:

1. Andreas Ronge, one of our Swedish friends from Malmö, started writing his canonical Neo4j Ruby driver since before we hit 1.0.
2. Brian Underwood and Chris Grigg joined the project and together released version 3.0 in September 2014.
3. Starting in 2017, the team around Heinrich Klobuczek contributed to the project and from **fall 2018 Heinrich took over the Neo4j.rb project as the primary maintainer.**

Gem: neo4j-ruby-driver

A ruby driver for neo4j based on seabolt protocol. It provides database connection, manages sessions and transactions.

```
Neo4j::Driver::GraphDatabase.driver('bolt://localhost:7687',
                                    Neo4j::Driver::AuthTokens.basic('neo4j', 'password')) do |driver|
  driver.session do |session|
    greeting = session.write_transaction do |tx|
      result = tx.run("CREATE (a:Greeting) SET a.message = $message RETURN a.message + ', from node ' +
id(a)",
                      message: 'hello, world')
      result.single.first
    end # session auto closed at the end of the block if one given
    puts greeting
  end
end # driver auto closed at the end of the block if one given
```

Check Examples for more usage.

Gem: activegraph

The activegraph gem uses as an API to connect to the server and provides an ActiveRecord-like experience for use in frameworks. It adds modules allowing the creation of models that look and feel very similar to those found in vanilla Ruby on Rails. The gem builds on neo4j-ruby-driver's foundation to streamline all aspects of node and relationship CRUD and provides an extremely advanced, intuitive, flexible DSL for generating Cypher.

How to get running:

```
# See the documentation for setup instructions

class Person
  include ActiveGraph::Node

  property :name

  has_many :out, :books, type: :OWNS_BOOK
  has_many :both, :friends, type: :HAS_FRIEND
end

person = Person.find_by(name: 'Jim')

# Get the books owned by Jim's friends:
person.friends.books.to_a
```

Deployment considerations

Very often, your gem choice may come down to how you want to deploy:

server

Using Ruby with a separate Neo4j Server over the http or bolt

embedded

Connecting directly to the Neo4j database files from within your Ruby process (this requires JRuby)

A separate Neo4j server is go-to choice for most developers, especially if they are used to other relational or NoSQL databases in frameworks such as Ruby on Rails.

It allows you to have separate web and database servers and allows compatibility with popular PaaSes such as Heroku. If this sounds good to you, any of the popular gems are solid choices and you are free to consider whether you want a thin wrapper or the full framework experience.

Neo4j Embedded and JRuby is less common but offers blazing fast performance and access to our core Java API. The downside is that JRuby has its own configuration and compatibility demands and hosting a Java app can be difficult. Thankfully, modern app servers such as Torquebox and a strong community provide far more options and resources now than ever before. If this is right for you, the neo4j-ruby and activegraph will equally offer support via the bolt protocol.

ActiveGraph gem integrations

There are many common gems that you'll want to use with your Neo4j database. Many are supported for the Neo4j.rb project:

Authentication

- devise-activegraph

Authorization

- cancancan-activegraph

File Attachment

- neo4jrb-paperclip (*)
- carrierwave-neo4j (*)

Pagination

- neo4j-will_paginate_redux (*)
- kaminari-neo4j (*)

ElasticSearch Integration

- neo4j-searchkick (*)

Admin User Interface

- rails_admin (*)

Integration With the Neo4j Spatial Plugin

- neo4jrb_spatial (*)

Ruby Object Manager

- rom-neo4j

Misc.

- neo4j-even_easier_id (BSON UUIDs)

Note: (*) not (yet) compatible with activegraph

The Example Project

The Neo4j example project is a small, one page webapp for the movies database built into the Neo4j tutorial. The front-end page is the same for all drivers: movie search, movie details, and a graph visualization of actors and movies. Each backend implementation shows you how to connect to Neo4j from each of the different languages and drivers.

You can learn more about our small, consistent example project across many different language drivers here. You will find the implementations for all drivers as individual GitHub repositories, which you can clone and deploy directly.

Using Neo4j from PHP

> *If you are a PHP developer, this guide provides an overview of options for connecting to Neo4j. While this guide is not comprehensive it introduces the different drivers and links to the relevant resources.*

You should be familiar with graph database concepts and the property graph model. You should have created an Neo4j AuraDB cloud instance, or installed Neo4j locally When developing with Neo4j, please use Java 8 or 11 and your favorite IDE.

Neo4j for PHP Developers

Alternatively, Neo4j can be installed on any system and then accessed via its bolt and HTTP APIs. We recommend the PHP Client for easiest development over bolt and HTTP APIs. You can also directly access the bolt protocol via the PHP Bolt library.

Neo4j Community Drivers

Members of the each programming language community have invested a lot of time and love to develop each one of the community drivers for Neo4j, so if you use any one of them, please provide feedback to the authors.

> The community drivers have been graciously contributed by the Neo4j community. Many of them are fully featured and well-maintained, but some may not be. Neo4j does not take any responsibility for their usability.

Full client

Neo4j-php-client is a client supporting multiple protocols. **Http** and **bolt** are supported, starting from neo4j 3.5 up until the most recent version.

It boasts many features such as multiple connections, transactions functions, authentication and auto-routing. It also integrates with the psalm static analysis tool for complete type safety.

It is being actively developed and has a big readme file on the Github page.

Author	Ghlen Nagels
[github] Source	https://github.com/neo4j-php/neo4j-php-client
Package	https://packagist.org/packages/laudis/neo4j-php-client
[code fork] Php	7.4 / 8.0+
[code fork] Neo4j	3.5 / 4.0+
Protocols	Bolt, HTTP
Example App	https://github.com/neo4j-examples/movies-neo4j-php-client

Installation

```
composer require laudis/neo4j-php-client
```

Example

```
$client = Laudis\Neo4j\ClientBuilder::create()->withDriver('default', 'bolt://neo4j:password@localhost')-
>build();

$result = $client->run(<<<'CYPHER'
MERGE (neo4j:Database {name: $dbName}) - [:HasRating] - (rating:Rating {value: 10})
RETURN neo4j, rating
CYPHER, ['dbName' => 'neo4j'])->first();

$neo4j = $result->get('neo4j');
$rating = $result->get('rating');

// Outputs "neo4j is 10 out of 10"
echo $neo4j->getProperty('name').' is '.$rating->getProperty('value') . ' out of 10!';
```

Bolt

A low level driver for the Bolt protocol in PHP.

Author	Michal Stefanak
[github] Source	https://github.com/neo4j-php/Bolt
[code fork] Php	7.1+ / 8.0+
[code fork] Neo4j	3.0+ / 4.0+
Protocols	Bolt

Installation

```
composer require stefanak-michal/bolt
```

Example

```
$bolt = new \Bolt\Bolt( new \Bolt\connection\StreamSocket() );
$protocol = $bolt->setProtocolVersions(4.1)->build();
$protocol->hello( \Bolt\helpers\Auth::basic('neo4j', 'neo4j') );

$protocol->run(<<<'CYPHER'
MERGE (neo4j:Database {name: $dbName})-[:HasRating]-(rating:Rating {value: 10})
RETURN neo4j, rating
CYPHER, ['dbName' => 'neo4j']);

$result = $protocol->pull()[0];

$neo4j = $result[0];
$rating = $result[1];

// Outputs "neo4j is 10 out of 10"
echo $neo4j->properties()['name'] . ' is ' . $rating->properties()['value'] . ' out of 10!';
```

Erlang and Elixir: Neo4j Community Driver

> *If you are an Erlang, or today an Elixir developer, this guide provides an overview of options for connecting to Neo4j. While this guide is not comprehensive it will introduce the different drivers and link to the relevant resources.*

You should be familiar with graph database concepts and the property graph model. You should have created an Neo4j AuraDB cloud instance, or installed Neo4j locally.

Neo4j Community Drivers

Members of the each programming language community have invested a lot of time and love to develop each one of the community drivers for Neo4j, so if you use any one of them, please provide feedback to the authors.

> The community drivers have been graciously contributed by the Neo4j community. Many of them are fully featured and well-maintained, but some may not be. Neo4j does not take any responsibility for their usability.

Neo4j for Erlang/Elixir Developers

Members of the Erlang and Elixir community have invested a lot of time and love to develop these drivers, so if you use them, please provide feedback to the authors.

The Example Project

The Neo4j example project is a small, one page webapp for the movies database built into the Neo4j tutorial. The front-end page is the same for all drivers: movie search, movie details, and a graph visualization of actors and movies. Each backend implementation shows you how to connect to Neo4j from each of the different languages and drivers.

You can learn more about our small, consistent example project across many different language drivers here. You will find the implementations for all drivers as individual GitHub repositories, which you can clone and deploy directly.

Elixir Bolt.Sips Driver

Neo4j driver for Elixir wrapped around the Bolt protocol.

Features:

- It uses Bolt, Neo4j's standard network protocol, designed for high-performance
- Supports transactions, as well as simple and complex Cypher queries with or without parameters
- Connection pool implementation using "A hunky Erlang worker pool factory", aka: Poolboy :)
- Supports Neo4j 3.0.x and forward

Author	Florin Patrascu
Example Project	https://github.com/neo4j-examples/bolt_movies_elixir_phoenix
Package	https://hex.pm/packages/bolt_sips

[github] Source	https://github.com/florinpatrascu/bolt_sips
Docs	http://hexdocs.pm/bolt_sips/

Neo4j.Sips

A simple Elixir wrapper around the Neo4j graph database REST API. It aims to help Elixir developers to play with Neo4j and to eventually become the main support for a future Ecto adapter.

Author	Florin Patrascu
Example Project	https://github.com/neo4j-examples/movies-elixir-phoenix
Package	https://hex.pm/packages/neo4j_sips
[github] Source	https://github.com/florinpatrascu/neo4j_sips
Docs	https://github.com/florinpatrascu/neo4j_sips/blob/master/README.md

Perl: Neo4j Community Drivers

> *This guide provides an overview of options for connecting to Neo4j using the Perl programming language. While this guide is not comprehensive, it will introduce some prominent drivers and link to the relevant resources.*

You should be familiar with graph database concepts and the property graph model. You should have created an Neo4j AuraDB cloud instance, or installed Neo4j locally

Neo4j Community Drivers

Members of the each programming language community have invested a lot of time and love to develop each one of the community drivers for Neo4j, so if you use any one of them, please provide feedback to the authors.

> The community drivers have been graciously contributed by the Neo4j community. Many of them are fully featured and well-maintained, but some may not be. Neo4j does not take any responsibility for their usability.

Neo4j for Perl developers

Members of the Perl community have invested a lot of time and love to develop drivers for Neo4j, so if you use them, please provide feedback to the authors.

The Example Project

The Neo4j example project is a small, one page webapp for the movies database built into the Neo4j tutorial. The front-end page is the same for all drivers: movie search, movie details, and a graph visualization of actors and movies. Each backend implementation shows you how to connect to Neo4j from each of the different languages and drivers.

You can learn more about our small, consistent example project across many different language drivers here. You will find the implementations for all drivers as individual GitHub repositories, which you can clone and deploy directly.

REST::Neo4p

This Perl driver from Mark Jensen works with Neo4j's REST API by using Perl5 objects in a consistent, idiomatic Perl-style. There is also a related DBI-compliant wrapper (DBD::Neo4p)

Author	Mark A. Jensen
Example Project	https://github.com/neo4j-examples/movies-perl-neo4p
Package	http://metacpan.org/release/REST-Neo4p
[github] Source	https://github.com/majensen/rest-neo4p
Docs	http://slideshare.net/majensen1/neo4p-dcbpw2015-46990541
Community Site	https://community.neo4j.com/c/drivers-stacks/perl

Neo4j::Driver

This is a Perl driver by Arne Johannessen. It enables interacting with a Neo4j server using the same classes and method calls as the official Neo4j drivers. It also has (currently experimental) support for HTTPS and Bolt.

Author	Arne Johannessen
Package	https://metacpan.org/release/Neo4j-Driver
[github] Source	https://github.com/johannessen/neo4j-driver-perl

Neo4j::Bolt

This is another driver from Mark Jensen. It's implemented as a Perl wrapper around the libneo4j-client C library, which implements the Bolt network protocol.

Author	Mark A. Jensen
[github] Source	https://github.com/majensen/perlbolt

Neo4j Connectors

Neo4j is supported by a rich ecosystem of libraries, tools, drivers, and guides provided by partners, users, and community contributors. We want to give an overview about what's available and link to the original sources.

Neo4j Connectors are integrations provided by Neo4j and are supported for existing Enterprise customers.

- Neo4j Connector for Apache Spark
- Neo4j Connector for Apache Kafka (also known as Neo4j-Streams)
- Neo4j Connector for Business Intelligence

Graph visualization

> *This section explains graph visualization tool options, and how to get insights from your data using visualization tools.*

Neo4j is designed to be very visual in nature. Native graph databases like Neo4j focus on relationships. Visualizing these relationships can give a unique "big picture" to your data that is difficult or impossible to get with traditional tables and business intelligence packages.

Graph visualization takes these capabilities one step further by drawing the graph in various formats so users can interact with the data in a more user-friendly way. With visualization tools, a full or partial graph can come to life and allow the user to explore it, setting various rules or views in order to analyze it from different perspectives.

This section is designed to help you understand how to export your graph data in Neo4j for display as a visualization, and list options so you can choose what suits your needs.

Why visualize a graph?

You have many options for working with data; JSON, XML, tabular, and others. Why represent it visually?

Anyone reviewing a graph can see the connections, determine areas of interest, or quickly assess the current state and organization of the data. As you can imagine, this can provide insight where other types of data formats cannot, bringing enormous value. Visualizations help make anomalies or relevant patterns stand out to help human eyes and brains detect them, where other types of data formats might not highlight hidden structures as well.

Graph

Let us look at a very rudimentary example of this using our Movie data from the earlier data modeling section guides. In the graph view above, we can easily pick out that `Lana Wachowski` directed both `Cloud Atlas` and `The Matrix` movies, where in the tabular representation, that information is not as clear or easy-to-find.

Table

Even if you feel that the relationship is not hard to find in the tabular format, imagine if we were looking at a graph that contained these individuals' entire filmography careers, as well as hundreds of other actors, directors, and film crew members. The connections could easily be lost in a non-visual presentation.

Neo4j visualization tools and products

Neo4j has two main visualization tools that are built and designed to work specifically with data in Neo4j's graph database: Neo4j Browser and Neo4j Bloom. We will briefly discuss the key details of each here.

Neo4j Bloom

Bloom is a standalone product focused on visualization that comes with every AuraDB Instance, and is available with a Neo4j Desktop. It was designed for business analysts annd nother non-developers to interact with graph data without writing any code.

Users can use natural language to query the database and explore patterns, clusters, and traversals in their graph data. It is also possible to create different dissections of the graph (called perspectives) that allow users to view different aspects and slices of graph data for further analysis.

Neo4j Browser

Neo4j Browser is an interactive cypher command shell for developers that allows you to interact with your graph and visualize the information in it. Neo4j Browser is bundled with Neo4j and is available in all editions and versions of Neo4j.

Its visualization functionality is designed to display a node-graph representation of the underlying data stored in the database in response to a given Cypher query, showing circles for nodes and lines for relationships. Neo4j Browser also provides some functionality for styling with color and size based on node labels and relationship types, or you can customize your own styles by importing a GRASS (graph-stylesheet) file for Neo4j Browser to reference. You can also use the built-in drop-down buttons on query result panes to easily export the data to PNG, SVG, or CSV formats.

Alternative visualizations of graph data

Not all graph visualizations represent data in circles and lines for nodes and relationships. Users may want to view data in various chart-based, map-based, or 3D formats.

Chart-based visualizations

Viewing data in familiar chart formats such as bar charts, histograms, pie charts, dials, meters and other representations might be preferred for various users and business needs. There are tools that support these types of charts for metrics and dashboarding.

There are several open source tools available, but we will mention a few with links that we have used before. Feel free to explore others!

Tableau

Tableau is a data analysis tool that can take data from a variety of sources and blend or split the data based on user specification. Using the Neo4j Connector for BI you can make a connection between Neo4j and Tableau as you would any other SQL databases, and visualize data directly.

Once the data is in Tableau, the user can interact with a drag-and-drop Tableau GUI to aggregate, splice, and style various combinations of the data into colorized visualizations in countless formats.

amCharts

Blog post: Charts for Neo4j query results with amCharts+Structr

Chart.js

Blog post: Charting Neo4j

Nivo

Blog post: Neo4j Spatial with Nivo charts

Map-based visualizations

Graph data is an excellent fit for mapping and representing geographic data, as it is laid out by entities and connections (locations/points and routes to get to those locations). Neo4j can help plot latitude and longitude, polygon geometries, routes, as well as distances, so a tool to overlay a map visualization on the front-end of this data provides a great deal of value for interacting and exploring an area.

Commercial tools by Tom Sawyer and Keylines both also support this type of visualization.

Leaflet.js / Mapbox

Leaflet.js is an open source library that allows us to create multiple layers and show/hide various layers. It is designed to be interactive and function on mobile phones, as well as traditional devices. You can extend functionality with a variety of plugins, including Mapbox. With these tools, you can create a base map layer (such as map tiles) and data visualizations live in map layers that are plotted on top of the map tiles. Mapbox also gives you the capability to add an interactive map.

Leaflet.js Resources

- Leaflet.js website: Leaflet.js
- Blog post: Leaflet.js to visualize Paradise Papers data
- Blog post: Using Leaflet.js and Mapbox to visualize spatial data in Neo4j
- Example source code: Leaflet/Mapbox spatial Neo4j
- Example source code: Leaflet/Mapbox interactive map
- Video: GraphConnect spatial Neo4j with Leaflet/Mapbox

Heatmap visualizations

A heatmap is a data visualization where colors are used to represent data values. It is often imposed on a map, but could also be on a matrix as well. When heatmaps are used on a map, pockets of activity may be spread out, so some form of interpolation is often used.

We will list the tool(s) we have encountered so far, but we will add to this as we interact with more.

- **Leaflet.js plugins:**

    - Blog post: Leaflet.js heatcanvas plugin

3D visualizations

[graph vis 3d] | *https://dist.neo4j.com/wp-content/uploads/graph_vis_3d.jpg*

Adding a third dimension may increase some complexity in the visualization, but also adds value. Exploring your data in 3D can help navigate through large amounts of data better and more clearly. Clustering should also be more apparent in a 3D visualization because data can be more spread out when using the third dimension, where 2D can cause groups to overlap or display more closely.

Kineviz (commercial tool) also supports this type of visualization.

3d-force-graph

With this open source library, there are a couple of different components for handling the physics behind three dimensions and for actually rendering the visualization. It uses an iterative approach for rendering in 3D and creates stunning, interactive visualizations. The tool includes features for customizing styles of nodes and relationships, as well as container layouts, rendering controls, configuring simulation, and user interaction. The data structure required is similar to previous tools we have seen, with collections for nodes and relationships. 3d-force-graph also offers functionality for visualizations to use with virtual reality.

3d-force-graph Resources

- Source code: 3d-force-graph Github
- Author post: Example
- Blog post: Visualizing Graphs in 3D

Other categories

There are still other tools for visualization that may not necessarily fit into the categories we have discussed so far. Instead, they expand the current boundaries and find uniquely powerful ways to utilize graph technologies. Thinking outside the box increases the possibilities of graph even further!

Graphileon

Graphileon is a platform for building graphy applications by composing functions and UI elements. It can be harnessed by users such as consultants and designers for styling and dashboards. Developers can also integrate with other technologies to customize applications, embed views, or extend functionality.

Partner and community visualization tools

Outside of Neo4j’s offerings, partners and community members have built tools and integrations to connect graph data in Neo4j with more graph visualizations. Learn more about options and functionality of these tools in the next section.

Resources

- Neo4j Browser
- Blog post: Neo4j Bloom
- Blog post: 15 Tools for Visualizing Your Neo4j Graph Database

Graph visualization tools

Types of graph visualization

There are three architectural categories into which most of our graph visualization tools fall. We will discuss how each of these categories handles the exported data and provide some pros and cons of the different architectures. Depending on the visualization needs, one of these categories may define the set of tools you can choose to implement as a solution to your business needs.

1. Standalone product tools

Certain tools and products are designed as standalone applications that can connect to Neo4j and interact with the stored data without involving any code. These applications are built with non-developers in mind - for business analysts, data scientists, managers, and other users to interact with Neo4j in a node-graph format.

Many of these tools involve commercial licenses and support but can be configured specifically to your use case and custom requirements. They also require little or no developer integration hours and setup.

The next paragraphs help us get a feel for the types of products in this area.

Neo4j Bloom

Neo4j Bloom is a data exploration tool that visualizes data in the graph and allows users to navigate and query the data without any query language or programming.

Users can write patterns similar to natural language questions to retrieve data and traverse layers of the graph. Bloom also allows appropriate users to edit, update, or correct the graph when missing information or bad data is found.

Neo4j Bloom is available in the following formats:

- Neo4j Bloom local with users accessing Bloom via Neo4j Desktop (free for local database instances)
- Neo4j Bloom server with users accessing Bloom via a web browser
- Neo4j Bloom through the sandbox
- Neo4j Bloom through Neo4j Database as a Service, AuraDB
- Included in Neo4j Startup Program

Bloom Resources

- Developer Guide: Neo4j Bloom User Interface Guide
- Blog post: Bloom-ing marvellous! Introducing Bloom 1.3
- Product information: Neo4j Bloom landing page

NeoDash

[neodash] | *//neo4j.com/labs/neodash/_images/neodash.png*

NeoDash is an open-source, low-code Dashboard Builder for Neo4j. As a part of Neo4j Labs, NeoDash is developed and supported via the online Community.

NeoDash lets you build an interactive dashboard with tables, graphs, bar charts, line charts, maps and more. Dashboards can be saved and shared directly from your Neo4j database.

- A low-code dashboard builder with a drag-and-drop interface
- Create visualizations directly from Cypher
- The ability to add customization and interactivity to dashboards
- Build and publish dashboards for read-only access

NeoDash Resources

- User Guide: NeoDash User Guide
- Blog Post: NeoDash 2.0 – A Brand New Way to Visualize Neo4j
- Try NeoDash: NeoDash Online Demo

GraphXR

GraphXR is a start-to-finish web-based visualization platform for interactive analytics. For technical users, it's a highly flexible and extensible environment for conducting ad hoc analysis. For business users, it's an intuitive tool for code-free investigation and insight.

- Collect data from Neo4j, SQL dbs, CSVs, and Json.
- Cleanse and enrich with built-in tools as well as API calls.
- Analyze links, properties, time series, and spatial data within a unified, animated context.
- Save back to Neo4j, output as a report, or embed in your webpage.

GraphXR supports a wide range of applications including law enforcement, medical research, and knowledge management.

Kineviz also has a graph app version of this tool that can be installed in Neo4j Desktop. The blog post about the graph app is included in the resources below.

GraphXR Resources

- Blog post: Adding GraphXR as a Graph App in Neo4j Desktop
- Blog post: Evaluating Investor Performance Using Neo4j, GraphXR and MLI
- Product information: GraphXR Datasheet

yFiles

yWorks provides sophisticated solutions for the visualization of graphs, diagrams, and networks with yFiles, a family of high-quality, commercial software programming libraries. The yFiles libraries enable you to easily create sophisticated graph-based applications powered by Neo4j. They support the widest range of desktop and web technologies and layout algorithms with the highest quality and performance. With the wide-ranging extensibility and large feature set, all your visualization needs can be satisfied.

yWorks also provides a free graph explorer app that is based on the yFiles technology. It can be installed in Neo4j Desktop.

yFiles Resources

- Blog post: Custom Visualization Solutions with yFiles and Neo4j
- Blog post: Visualizing Neo4j Database Content Like a Pro
- Webinar: Technical intro to yFiles with Neo4j
- Product information: yFiles Visualization Libraries

Linkurious Enterprise

Linkurious Enterprise is an on-premises and browser-based platform that works on top of graph databases. It brings graph visualization and analysis capabilities to analysts tasked to detect and analyze threats in large volumes of connected data. Organizations such as the French Ministry of Economy and Finance, Zurich Insurance or Bank of Montreal use Linkurious Enterprise to fight financial crime, terror networks or cyber threats.

Linkurious Resources

- Blog post: Panama Papers Discovery with Neo4j and Linkurious
- Blog post: Fraud detection with Neo4j and Linkurious
- Blog post: Detect and Investigate Financial Crime with Neo4j and Linkurious
- Webinar: How to visualize Neo4j with Linkurious
- Solution: Linkurious Enterprise + Neo4j
- Product datasheet Linkurious Enterprise

Graphistry

Graphistry brings a human interface to the age of big and complex data. It automatically transforms your data into interactive, visual investigation maps built for the needs of analysts. Quickly surface relationships between events and entities without writing queries or wrangling data. Harness all of your data without worrying about scale, and pivot on the fly to follow anywhere your investigation leads you.

Ideal for everything from security, fraud, and IT investigations to 3600 views of customers and supply chains, Graphistry turns the potential of your data into human insight and value.

Graphistry Resources

- Source code: Graphistry on Github
- Product information: Graphistry graph visualization

Graphlytic

Graphlytic is a highly customizable web application for graph visualization and analysis. Users can interactively explore the graph, look for patterns with the Cypher language, or use filters to find answers to any graph question. Graph rendering is done with the Cytoscape.js library which allows Graphlytic to render tens of thousands of nodes and hundreds of thousands of relationships.

The application is provided in three ways: Desktop, Cloud, and Server. Graphlytic Desktop is a free Neo4j Desktop application installed in just a few clicks. Cloud instances are ideal for small teams that need them get up and running in very little time. Graphlytic Server is used by corporations and agencies with highly sensitive data typically in closed networks.

Graphlytic Resources

- Product webpage: https://graphlytic.biz
- Online Demo: Graphlytic Demo
- Free Desktop Installation: How To Install And Use Graphlytic In Neo4j Desktop
- Features: Graphlytic Feature Clips
- Blog post: Parallel Relationship Models with Graphlytic

Perspectives

Tom Sawyer Perspectives is a robust platform for building enterprise-class graph and data visualization and analysis applications. It is a complete graph visualization software development kit (SDK) with a graphics-based design and preview environment. The platform integrates enterprise data sources with the powerful graph visualization, layout, and analysis technology to solve big data problems. Enterprises, system integrators, technology companies, and government agencies use Tom Sawyer Perspectives to build a wide range of applications.

Perspectives Resources

- Product information: Perspectives graph visualization

Keylines

KeyLines makes it easy to build and deploy high-performance network visualization tools quickly. Every aspect of your application can be tailored to suit you, your data and the questions you need to answer. KeyLines applications work on any device and in all common browsers, to reach everyone who needs to use them. It is also compatible with any IT environment, letting you deploy your network visualization application to an unlimited number of diverse users. You can build a custom application that is scalable and easy to use.

Keylines Resources

- Product information: Keylines graph visualization

Semspect

SemSpect is a highly scalable knowledge graph exploration tool that uses visual aggregation to solve the hairball problem faced by standard graph visualization approaches. The data guided construction of the exploration tree empowers the users to build complex requests intuitively without query syntax. Its meta level approach is very effective for grasping the overall structure of the graph data, while flexible access to node and relationship details ensures easy inspection and filtering. SemSpect furthermore allows to define query-based node labels during exploration to refine the graph data schema.

SemSpect is available as follows:

- SemSpect as Graph App for Neo4j Desktop (free for local database instances)
- SemSpect as Web App for Neo4j database servers

Semspect Resources

- Product information: SemSpect for Neo4j
- Blog post: A Different Approach to Graph Visualization

Visualization Resources

- Blog series: Neo4j Visualization
- Blog: Max de Marzi on Visualization with Neo4j
- Neo4j Visualization: YouTube videos

2. Embeddable tools with built-in Neo4j connections

These kinds of tools can be included as a dependency within an application and can easily be configured and styled for your application and Neo4j. Each is easily connected to an instance of the graph database using configuration properties and allows you to style the visualization based on nodes, relationships, or specific properties.

Embedding the visualization within the application allows the developer to create applications that include the visualization as part of the user interface. This also means that the developer can write other components and customize the application experience and other components involved in the application to the exact business requirements.

On the downside, these libraries don't often support extremely complex or heavy workloads and do not have vendor support or SLAs for functionality requests. Because they are managed by the community, the tools depend on the community for support and feature improvements. Also, this typically means that our client application is connecting directly to the database, which might not always be the desired architecture.

Let us look at some of the tools in this category.

Neovis.js

This library was designed to combine JavaScript visualization and Neo4j in a seamless integration. Connection to Neo4j is simple and straightforward, and because it is built with Neo4j's property graph model in mind, the data format Neovis expects aligns with the database. Customizing and coloring styles based on labels, properties, nodes, and relationships is defined in a single configuration object. Neovis.js can be used without writing Cypher and with minimal JavaScript for integrating into your project.

> The Neovis library is one of our Neo4j Labs projects. To learn more about Neo4j Labs, visit our Labs page.

To maximize functionality and data analysis capabilities through visualization, you can also combine this library with the graph algorithms library in Neo4j to style the visualization to align with results of algorithms such as page rank, centrality, communities, and more. Below, we see a graph visualization of Game Of Thrones character interactions rendered by neovis.js, and enhanced using Neo4j graph algorithms by applying pagerank and community detection algorithms to the styling of the visualization.

An advantage of enhancing graph visualization with these algorithms is that we can visually interpret the results of these algorithms.

Neovis.js Resources

- Blog post: Neovis.js
- Download neovis.js: npm package

Popoto.js

Popoto.js is a JavaScript library that is built upon D3.js. Popoto.js will help users build queries in a visual way to execute against Neo4j. Users can also customize the results and the visual display. Along with the visualization, you can include auto-complete searches for potential queries, see the Cypher translations that are generated from the visualization, review text results from queries, and more.

To use Popoto.js in your application, you simply need to include each component independently bound to a container id in an HTML page. The rest of the content will be generated from that.

Popoto.js Resources

- Documentation: Popoto.js
- Website: popoto.js

3. Embeddable libraries without direct Neo4j connection

These libraries offer the ability to embed graph visualization in an application, but without connecting directly to Neo4j. An advantage here is that we can populate our visualization with data sent from an API application that connects to the database, ensuring the client application is not querying the database directly. The downside, however, is that we often must transform the results to export from Neo4j into the format expected by these libraries.

We can get a closer look at these tools in the next paragraphs.

D3.js

As the first line on D3’s website states "D3.js is a JavaScript library for manipulating documents based on data." You can bind different kinds of data to a DOM and then execute different kinds of functions on it. One of those functions includes generating an SVG, canvas, or HTML visualization from the data in the DOM.

Neo4j’s movie example applications use d3.js, and you can find a variety of other projects using Neo4j and d3. The complicated part of D3 (or any embeddable library that doesn’t have direct Neo4j connection) is converting your graph data into the expected map format for export. D3 expects two different collections of graph data - one for nodes[] and one for links[] (relationships). Each of these maps includes arrays of properties for each node and relationship that d3 then converts into circles and lines. Version 4 and 5 of d3.js also support force-directed graphs, where the visualization adjusts to the user’s view pane.

D3.js Resources

- Website: D3.js
- D3 and graphs example: D3 Examples
- Neo4j Github examples with d3: Examples with Neo4j

Vis.js

This library offers a variety of visualizations designed to handle large, dynamic data sets. There are a variety of formats to style your data, including timeline, dataset, graph2d, graph3d, and network. The most common format seen with Neo4j is the network visualization.

Even with the network format, there are numerous customizations available for styling nodes, labels, animations, coloring, grouping, and others. For additional information and to see everything that is available, check out their docs and examples linked in the resources below.

Vis.js Resources

- Vis.js website: Vis.js
- Network format examples: Format Examples
- Source code project: Vis.js Github

Sigma.js

While some libraries are meant to include all the capabilities in one bundle, Sigma.js touts a highly-extensible environment where users can add extension libraries or plugins to provide additional capability. This library takes exported data in either JSON or GEXF formats.

Users can start from a very basic visualization right out of the box, and then begin adding custom functions and rendering for styling preferences. Once the requirements surpass what is possible there, users can write and use their own custom plugins for specific functionality. Be sure to check out the repository, though, for any existing extensions!

Sigma.js Resources

- Website: Sigma.js
- Source code: Sigma.js Github
- Blog post: Sigma.js+Neo4j

Vivagraph.js

Vivagraph.js was built to handle different types of layout algorithms for arranging nodes and edges. It manages data set sizes from very small to very large and also renders in WebGL, SVG, and CSS-based formats. Customizations and styling are available through CSS modifications and extension libraries. It also can track changes in the graph that update the visualization accordingly.

Vivagraph.js Resources

- Source code: Vivagraph.js Github
- Blog post: Viavgraph.js+Neo4j

Cytoscape.js

This library is also meant to visualize and render network node graphs and offers customization and extensibility for additional features. Cytoscape.js responds to user interaction and works on touch screen interfaces, allowing users to zoom, tap, and explore in the method that is relevant to them. You can customize styling and web page view with a variety of style components.

Cytoscape.js Resources

- Website: Cytoscape.js
- Source code: Cytoscape.js Github

Appendix

In this section, you can find:

- Tutorials on how to create and explore graphs using Cypher - Neo4j graph query language.
- Tutorials on how to import data from a relational database into Neo4j using `LOAD CSV` Cypher command.
- A list of all available example datasets used across doc sets and learning courses. In the corresponding section links to the example datasets are provided, and ways of data import are explained.
- A list of resources which may inspire you to learn more about the Neo4j environment and graph technology.

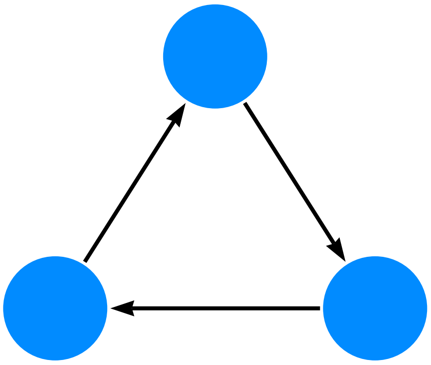
Graph database concepts

Neo4j uses a *property graph* database model.

A graph data structure consists of **nodes** (discrete objects) that can be connected by **relationships**.

Example 3. Concept of a graph structure.

A graph with three nodes (the circles) and three relationships (the arrows).



The Neo4j property graph database model consists of:

- **Nodes** describe entities (discrete objects) of a domain.
- **Nodes** can have zero or more **labels** to define (classify) what kind of nodes they are.
- **Relationships** describes a connection between a *source node* and a *target node*.
- **Relationships** always has a direction (one direction).
- **Relationships** must have a **type** (one type) to define (classify) what type of relationship they are.
- Nodes and relationships can have **properties** (key-value pairs), which further describe them.

In mathematics, graph theory is the study of graphs.

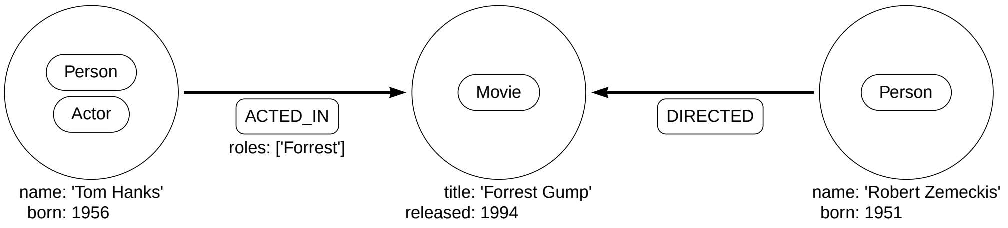
In graph therory:

- Nodes are also refered to as vertices or points.
- Relationships are also refered to as edges, links, or lines.

Example graph

The example graph shown below, introduces the basic concepts of the property graph:

Example 4. Example graph.



Example 5. Cypher.

To create the example graph, use the Cypher clause CREATE.

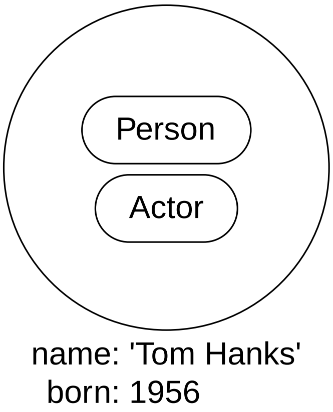
```
CREATE (:Person:Actor {name: 'Tom Hanks', born: 1956})-[:ACTED_IN {roles: ['Forrest']}]->(:Movie
{title: 'Forrest Gump'})<-[:DIRECTED]-(:Person {name: 'Robert Zemeckis', born: 1951})
```

Node

Nodes are used to represent *entities* (discrete objects) of a domain.

The simplest possible graph is a single node with no relationships. Consider the following graph, consisting of a single node.

Example 6. Node.



The node labels are:

- `Person`
- `Actor`

The properties are:

- `name: Tom Hanks`
- `born: 1956`

The node can be created with Cypher using the query:

```
CREATE (:Person:Actor {name: 'Tom Hanks', born: 1956})
```

Node labels

Labels shape the domain by grouping (classifying) nodes into sets where all nodes with a certain label belong to the same set.

For example, all nodes representing users could be labeled with the label `User`. With that in place, you can ask Neo4j to perform operations only on your user nodes, such as finding all users with a given name.

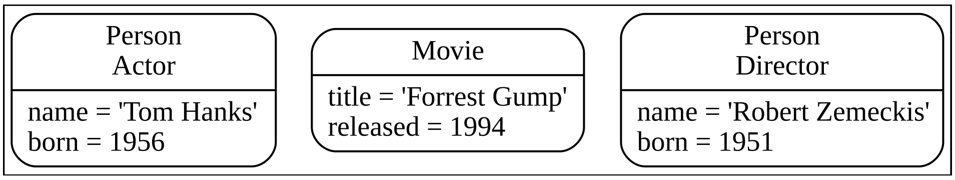
Since labels can be added and removed during runtime, they can also be used to mark temporary states for nodes. A `Suspended` label could be used to denote bank accounts that are suspended, and a `Seasonal` label can denote vegetables that are currently in season.

A node can have zero to many labels.

In the example graph, the node labels, `Person`, `Actor`, and `Movie`, are used to describe (classify) the nodes. More labels can be added to express different dimensions of the data.

The following graph shows the use of multiple labels.

Example 7. Multiple labels.



Relationship

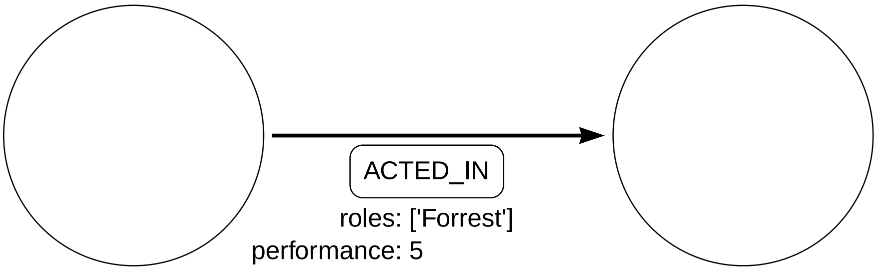
A relationship describes how a connection between a *source node* and a *target node* are related. It is possible for a node to have a relationship to itself.

A relationship:

- Connects a *source node* and a *target node*.
- Has a direction (one direction).
- Must have a **type** (one type) to define (classify) what type of relationship it is.
- Can have properties (key-value pairs), which further describe the relationship.

Relationships organize nodes into structures, allowing a graph to resemble a list, a tree, a map, or a compound entity — any of which may be combined into yet more complex, richly inter-connected structures.

Example 8. Relationship.



The relationship type: ACTED_IN

The properties are:

- roles: ['Forrest']
- performance: 5

The roles property has an array value with a single item ('Forrest') in it.

The relationship can be created with Cypher using the query:

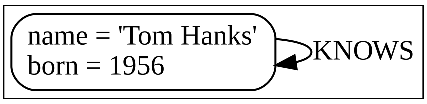
```
CREATE ()-[:ACTED_IN {roles: ['Forrest'], performance: 5}]->()
```

ℹ You must create or reference a *source node* and a *target node* to be able to create a relationship.

Relationships always have a direction. However, the direction can be disregarded where it is not useful. This means that there is no need to add duplicate relationships in the opposite direction unless it is needed to describe the data model properly.

A node can have relationships to itself. To express that Tom Hanks KNOWS himself would be expressed as:

Example 9. Relationship to a single node.

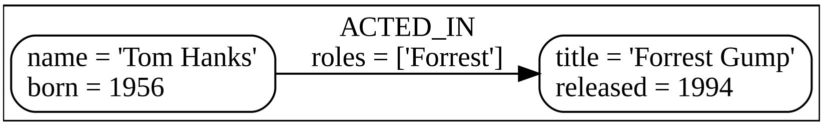


Relationship type

A relationship must have exactly one relationship type.

Below is an ACTED_IN relationship, with the Tom Hanks node as the *source node* and Forrest Gump as the *target node*.

Example 10. Relationship type.



Observe that the `Tom Hanks` node has an *outgoing* relationship, while the `Forrest Gump` node has an *incoming* relationship.

Properties

Properties are key-value pairs that are used for storing data on nodes and relationships.

The value part of a property:

- Can hold different data types, such as `number`, `string`, or `boolean`.
- Can hold a homogeneous list (array) containing, for example, strings, numbers, or boolean values.

Example 11. Number

```
CREATE (:Example {a: 1, b: 3.14})
```

- The property `a` has the type `integer` with the value `1`.
- The property `b` has the type `float` with the value `3.14`.

Example 12. String and boolean

```
CREATE (:Example {c: 'This is an example string', d: true, e: false})
```

- The property `c` has the type `string` with the value `'This is an example string'`.
- The property `d` has the type `boolean` with the value `true`.
- The property `e` has the type `boolean` with the value `false`.

Example 13. Lists

```
CREATE (:Example {f: [1, 2, 3], g: [2.71, 3.14], h: ['abc', 'example'], i: [true, true, false]})
```

- The property `f` contains an array with the value `[1, 2, 3]`.
- The property `g` contains an array with the value `[2.71, 3.14]`.
- The property `h` contains an array with the value `['abc', 'example']`.
- The property `i` contains an array with the value `[true, true, false]`.

For a thorough description of the available data types, refer to the Cypher manual → Values and types.

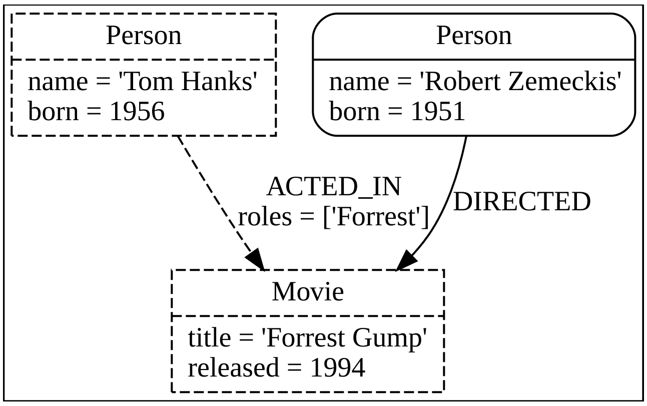
Traversals and paths

A traversal is how you query a graph in order to find answers to questions, for example: "What music do my friends like that I don't yet own?", or "What web services are affected if this power supply goes down?".

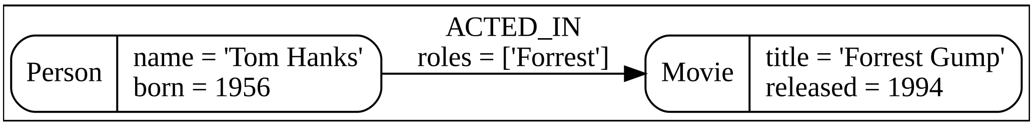
Traversing a graph means visiting nodes by following relationships according to some rules. In most cases only a subset of the graph is visited.

Example 14. Path matching.

To find out which movies Tom Hanks acted in according to the tiny example database, the traversal would start from the `Tom Hanks` node, follow any `ACTED_IN` relationships connected to the node, and end up with `Forrest Gump` as the result (see the dashed lines):



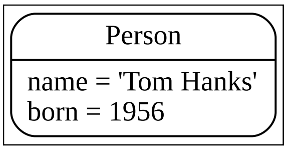
The traversal result could be returned as a path with the length `1`:



The shortest possible path has length zero. It contains a single node and no relationships.

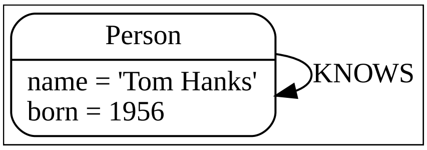
Example 15. Path of length zero.

A path containing only a single node has the length of `0`.



Example 16. Path of length one.

A path containing one relationship has the length of 1.



Schema

A *schema* in Neo4j refers to indexes and constraints.

Neo4j is often described as *schema optional*, meaning that it is not necessary to create indexes and constraints. You can create data — nodes, relationships and properties — without defining a schema up front. Indexes and constraints can be introduced when desired, in order to gain performance or modeling benefits.

Indexes

Indexes are used to increase performance. To see examples of how to work with indexes, see Using indexes. For detailed descriptions of how to work with indexes in Cypher, see Cypher Manual → Indexes.

Constraints

Constraints are used to make sure that the data adheres to the rules of the domain. To see examples of how to work with constraints, see Using constraints. For detailed descriptions of how to work with constraints in Cypher, see the Cypher manual → Constraints.

Naming conventions

Node labels, relationship types, and properties (the key part) are case sensitive, meaning, for example, that the property `name` is different from the property `Name`.

The following naming conventions are recommended:

Table 24. Naming conventions

Graph entity	Recommended style	Example
Node label	Camel case, beginning with an upper-case character	`:VehicleOwner` rather than `:vehicle_owner`
Relationship type	Upper case, using underscore to separate words	`:OWNS_VEHICLE` rather than `:ownsVehicle`
Property	Lower camel case, beginning with a lower-case character	`firstName` rather than `first_name`

For the precise naming rules, refer to the Cypher manual → Naming rules and recommendations.

Transition from relational to graph database

> *This chapter explores the concepts of graph databases from a relational developer's point of view. It aims to explain the conceptual differences between relational and graph database structures and data models. It also gives a high-level overview of how working with each database type is similar or different - from the relational and graph query languages to interacting with the database from applications.*

Relational databases have been the work horse of software applications since the 80's, and continue as such to this day. They store highly-structured data in tables with predetermined columns of specific types and many rows of those defined types of information. Due to the rigidity of their organization, relational databases require developers and applications to strictly structure the data used in their applications.

In relational databases, references to other rows and tables are indicated by referring to primary key attributes via foreign key columns. Joins are computed at query time by matching primary and foreign keys of all rows in the connected tables. These operations are compute-heavy and memory-intensive and have an exponential cost.

When many-to-many relationships occur in the model, you must introduce a *JOIN* table (or associative entity table) that holds foreign keys of both the participating tables, further increasing join operation costs. The image below shows this concept of connecting a Person (from Person table) to a Department (in Department table) by creating a Person-Department join table that contains the ID of the person in one column and the ID of the associated department in the next column.

As you can probably see, this makes understanding the connections very cumbersome because you must know the person ID and department ID values (performing additional lookups to find them) in order to know which person connects to which departments. Those types of costly join operations are often addressed by denormalizing the data to reduce the number of joins necessary, therefore breaking the data integrity of a relational database.

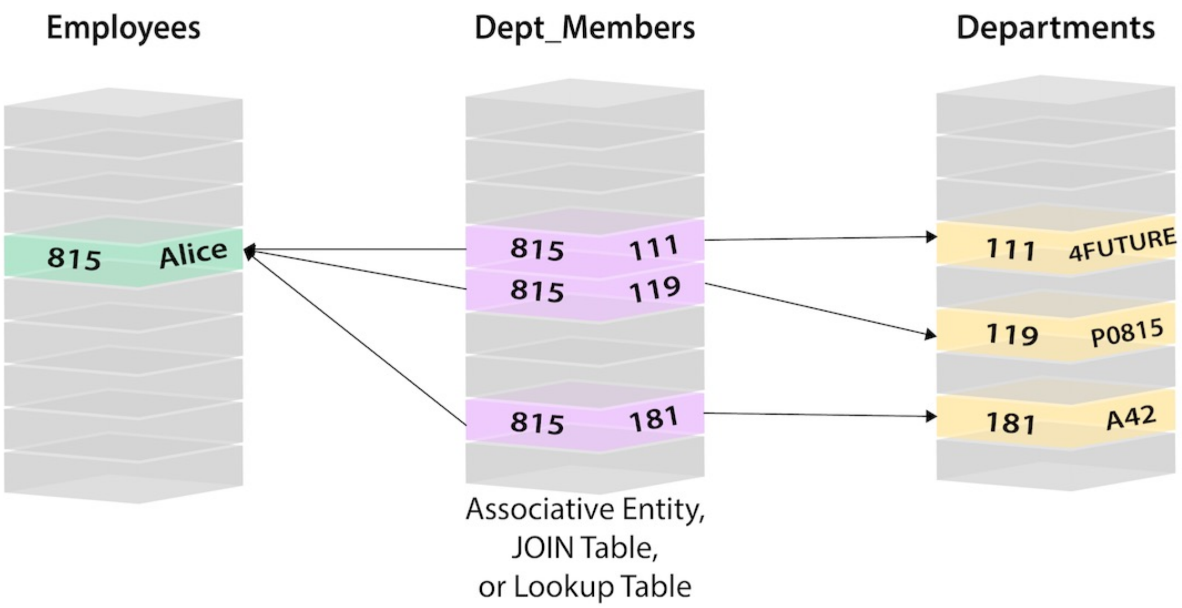


Figure 1. Relational Model (click to zoom)

Although not every use case is a good fit for this stringent data model, the lack of viable alternatives and the wide support for relational databases made it difficult for alternative models to break into the mainstream. However, the NoSQL era arrived in the market, filling some needs for users and businesses, but still missing the importance of the connections between data. This is how graph databases were born. They were designed to provide the greatest advantage in the connected world we live in today.

Translating Relational Knowledge to Graphs

Unlike other database management systems, relationships are of equal importance in the graph data model to the data itself. This means we are not required to infer connections between entities using special properties such as foreign keys or out-of-band processing like map-reduce.

By assembling nodes and relationships into connected structures, graph databases enable us to build simple and sophisticated models that map closely to our problem domain. The data stays remarkably similiar to its form in the real world - small, normalized, yet richly connected entities. This allows you to query and view your data from any imaginable point of interest, supporting many different use cases.

Each node (entity or attribute) in the graph database model directly and physically contains a list of relationship records that represent the relationships to other nodes. These relationship records are organized by type and direction and may hold additional attributes. Whenever you run the equivalent of a *JOIN* operation, the graph database uses this list, directly accessing the connected nodes and eliminating the need for expensive search-and-match computations.

This ability to pre-materialize relationships into the database structure allows Neo4j to provide performance of several orders of magnitude above others, especially for join-heavy queries, allowing users to leverage a *minutes to milliseconds* advantage.

Data Model Differences

As you can probably imagine from the structural differences discussed above, the data models for relational versus graph are very different. The straightforward graph structure results in much simpler and more expressive data models than those produced using traditional relational or other NoSQL databases.

If you are used to modeling with relational databases, remember the ease and beauty of a well-designed, normalized entity-relationship diagram - a simple, easy-to-understand model you can quickly whiteboard with your colleagues and domain experts. A graph is exactly that - a clear model of the domain, focused on the use cases you want to efficiently support.

Let’s compare the two data models to show how the structure differs between relational and graph.

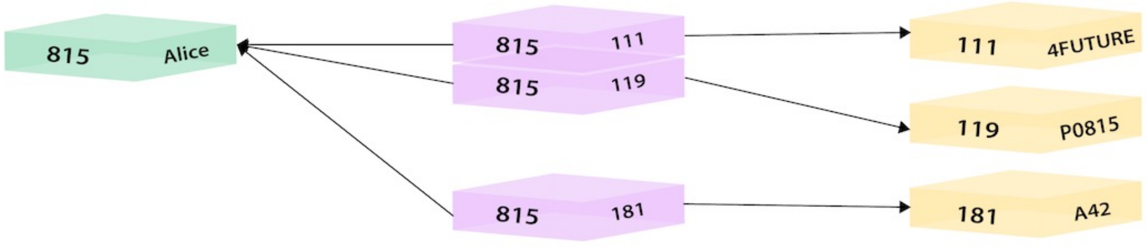


Figure 2. Relational - Person and Department tables (click to zoom)

In the above relational example, we search the Person table on the left (potentially millions of rows) to find the user Alice and her person ID of 815. Then, we search the Person-Department table (orange middle table) to locate all the rows that reference Alice’s person ID (815). Once we retrieve the 3 relevant rows, we go to the Department table on the right to search for the actual values of the department IDs (111, 119, 181). Now we know that Alice is part of the 4Future, P0815, and A42 departments.

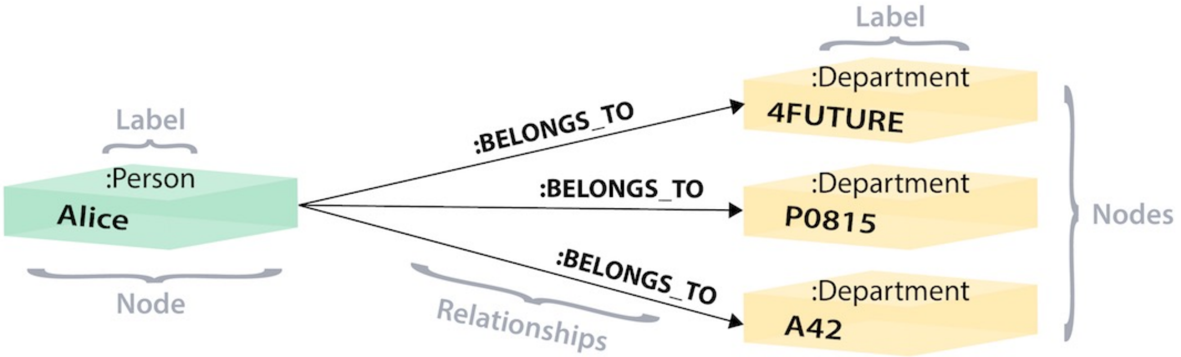


Figure 3. Graph - Alice and 3 Departments as nodes (click to zoom)

In the above graph version, we have a single node for Alice with a label of Person. Alice belongs to 3 different departments, so we create a node for each one and with a label of Department. To find out which departments Alice belongs to, we would search the graph for Alice’s node, then traverse all of the BELONGS_TO relationships from Alice to find the Department nodes she is connected to. That’s all we need - a single hop with no lookups involved.

More information on this topic can be found in the Data Modeling section.

Data Storage and Retrieval

Querying relational databases is easy with SQL - a declarative query language that allows both easy ad-hoc querying in a database tool, as well as use-case-specific querying from application code. Even object-relational mappers (ORMs) use SQL under the hood to talk to the database.

Do graph databases have something similar? Yes!

Cypher, Neo4j's declarative graph query language, is built on the basic concepts and clauses of SQL but has a lot of additional graph-specific functionality to make it easy to work with your graph model.

If you have ever tried to write a SQL statement with a large number of joins, you know that you quickly lose sight of what the query actually does because of all the technical noise in SQL syntax. In Cypher, the syntax remains concise and focused on domain components and the connections among them, expressing the pattern to find or create data more visually and clearly. Other clauses outside of the basic pattern matching look very similar to SQL, as Cypher was built on the predecessor language's foundations.

We will cover Cypher query language syntax in an upcoming guide, but let us look at a brief example of how a SQL query differs from a Cypher query. In the organizational domain from our data modeling example above, what would a SQL statement that **lists the employees in the IT Department** look like, and how does it compare to the Cypher statement?

SQL Statement

```
SELECT name FROM Person
LEFT JOIN Person_Department
  ON Person.Id = Person_Department.PersonId
LEFT JOIN Department
  ON Department.Id = Person_Department.DepartmentId
WHERE Department.name = "IT Department"
```

Cypher Statement

```
MATCH (p:Person)-[:WORKS_AT]->(d:Dept)
WHERE d.name = "IT Department"
RETURN p.name
```

You can find more about Cypher syntax in the upcoming chapters for Cypher Query Language and transitioning from SQL to Cypher.

Transitioning from Relational to Graph - In Practice

If you do decide to move your data from a relational to a graph database, the steps to transition your applications to use Neo4j are actually quite simple. You can connect to Neo4j with a driver or connector library designed for your stack or programing language, just as you can with other databases. Thanks to Neo4j and its community, there are Neo4j drivers that mimic existing database driver idioms and approaches for nearly any popular programing language.

For instance, the Neo4j JDBC driver would be used like this to query the database for *John's departments*:

```
Connection con = DriverManager.getConnection("jdbc:neo4j://localhost:7474/");

String query =
    "MATCH (:Person {name:{1}})-[:EMPLOYEE]-(d:Department) RETURN d.name as dept";
try (PreparedStatement stmt = con.prepareStatement(QUERY)) {
    stmt.setString(1,"John");
    ResultSet rs = stmt.executeQuery();
    while(rs.next()) {
        String department = rs.getString("dept");
        ....
    }
}
```

For more information, you can visit our pages for Building Applications to see how to connect to Neo4j using different programming languages.

Resources

- Free eBook: Relational to Graph
- DZone Refcard: From Relational to Graph
- Data Modeling: Relational to Graph

Transition from NoSQL to graph database

Although unhelpfully named, the NoSQL ("Not only SQL") space brings together many interesting solutions offering different data models and database systems, each more suitable than traditional SQL solutions for certain use cases and shapes of data.

With the advent of the NoSQL movement, the "one-size-fits-all" proposition of large relational systems was replaced by conscious decisions about finding the right tool for the job.

```
<div class="responsive-embed">
<iframe width="680" height="425" src="https://www.youtube.com/embed/5Tl8WcaqZoc" frameborder="0"
allowfullscreen></iframe>
</div>
```

Most NoSQL systems are **aggregate-oriented**, grouping the data based on a particular criterion and the database type (such as document store, key-value pair, etc). This model provides only simple, limited operations and only forms one dedicated view of your data. Focusing on one aggregate at a time allows users to easily spread many chunks of data across a network of machines along the aggregate dimension (for instance, the **Document** in document databases), but that means that other projections and perspectives have to be computed by crunching or duplicating your data.

> Most NoSQL databases store sets of disconnected aggregates. This makes it difficult to use them for connected data and graphs. One well-known strategy for adding relationships to such stores is to embed an aggregate's identifier inside the field belonging to another aggregate — effectively introducing foreign keys. But, this requires joining aggregates at the application level, which quickly becomes prohibitively expensive.
>
> — Graph Databases, O'Reilly

Other NoSQL databases lack relationships. Graph databases, on the other hand, handle fine-grained networks of information, providing **any perspective** on your data that fits your use case. The well-known and trusted transactional guarantees from relational systems also protect updates of the graph data in Neo4j, conforming to ACID standards.

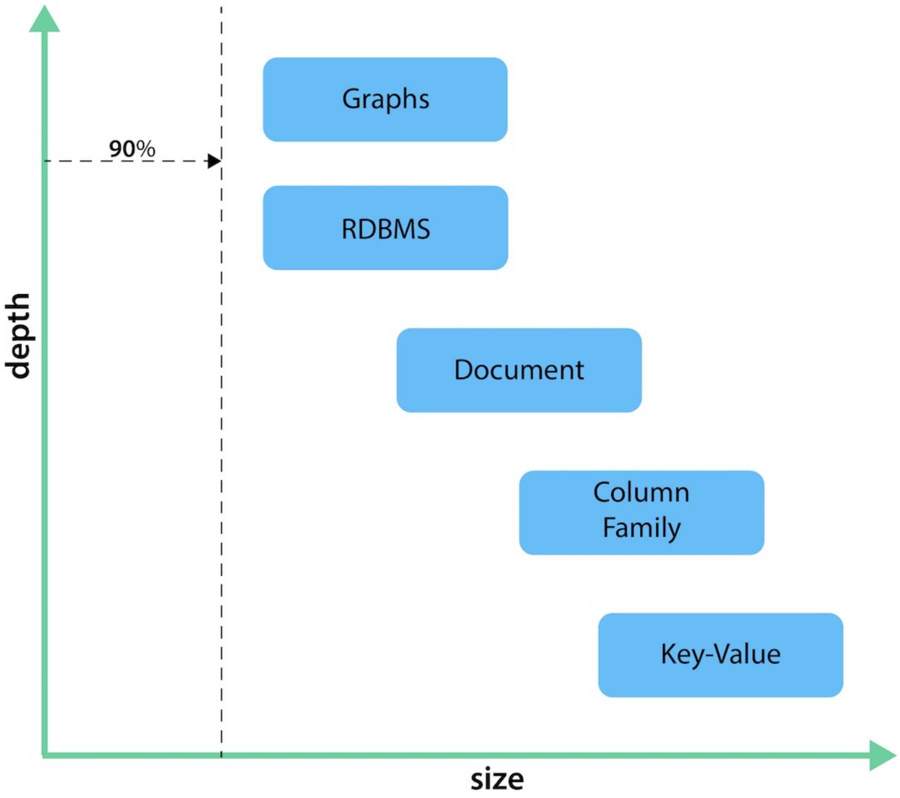
Let's compare the graph data model to other NoSQL models.

Translating NoSQL Knowledge to Graphs

With the advent of the NoSQL movement, businesses of all sizes have a variety of modern options from which to build solutions relevant to their use cases.

- Calculating average income? Ask a **relational database**.
- Building a shopping cart? Use a **key-value Store**.
- Storing structured product information? Store as a **document**.
- Describing how a user got from point A to point B? Follow a **graph**.

The chart below shows how each database type stacks up on a spectrum measuring depth and size. While key-value stores can handle massive sizes, they are designed for a high-level view (low depth) of the data. Graph databases retain minimum sizing, even at a greater depth of data than other types of databases. The other types of databases fall somewhere in between those ranges.



Key-Value vs. Graph: Data Model Differences

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

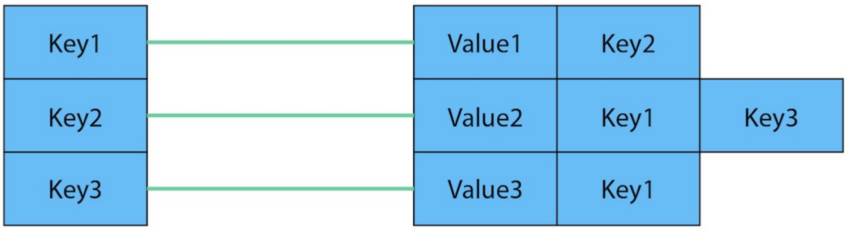


Figure 4. Key-Value Model (click to zoom)

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 below 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).

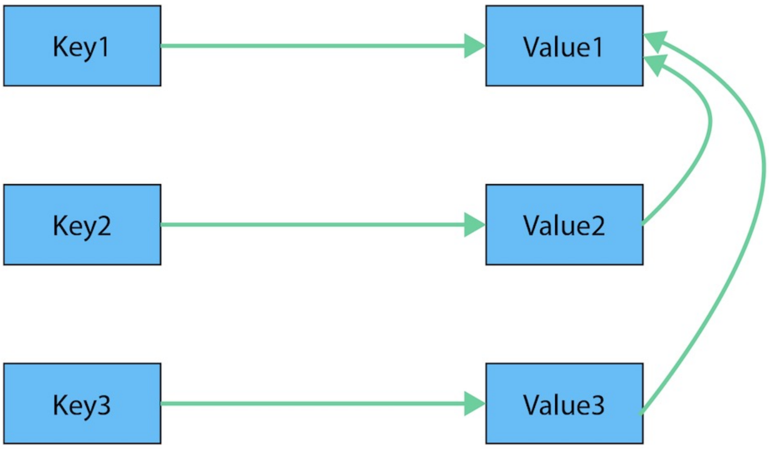


Figure 5. Key-Value as Graph (click to zoom)

Document vs. Graph: Data Model Differences

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. The image below demonstrates how a document store hierarchy is structured as pieces within larger components.

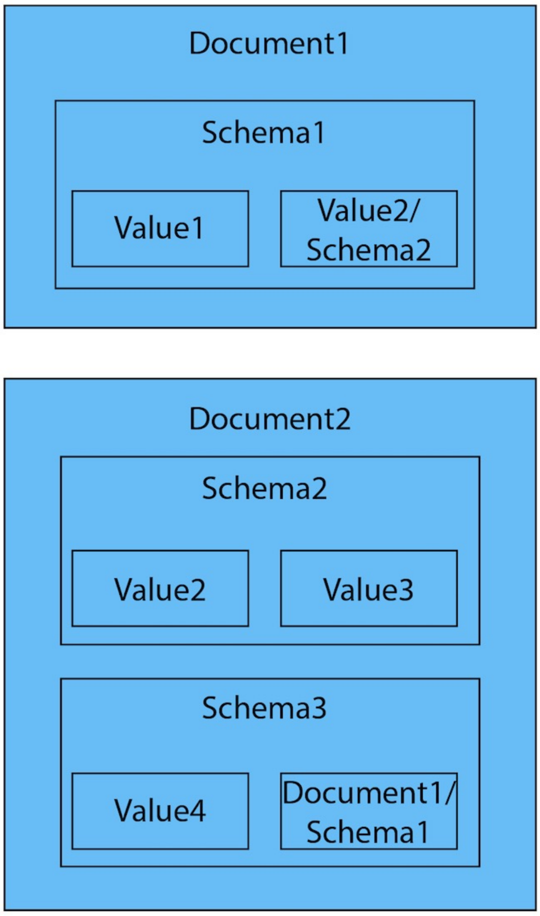


Figure 6. Document Model (click to zoom)

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. The graph version below demonstrates how moving this data to a graph structure allows you to view different levels and details of the tree in different combinations.

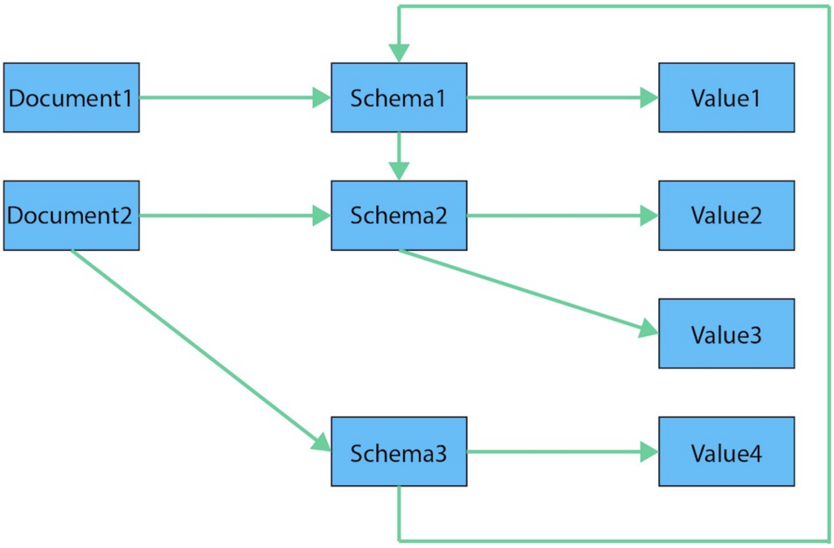


Figure 7. Document as Graph (click to zoom)

Resources

- DZone: NoSQL Database Types
- Blog post: Tour of Aggregate Stores

Example datasets

> *Here you can find a list of available example datasets for Neo4j and learn how to import and explore them.*

Datasets

For getting started with Neo4j, it's helpful to use example datasets relevant to your domain and use case. For each we want to provide a description, the graph model and some use case queries.

Built-in examples

Neo4j Browser comes with two built-in databases, which you can create and explore using interactive slideshows.

The **"Movies"** example, is launched via the `:guide movie-graph` command and contains a small graph of movies and people related to those movies as actors, directors, producers etc.

The **"Northwind"** example, is run via `:guide northwind-graph` and contains a traditional retail-system with products, orders, customers, suppliers and employees. It walks you through the import of the data and incrementally complex queries using the available data.

Other Browser guides

Other example datasets that you can run within your own Neo4j Browser are:

- Game of Thrones Interactions — `:play got`
- UK company registration, property ownership, political donations — `:play ukcompanies`
- Stack Overflow users, tags and Q&A data — `:play stackoverflow`
- BBC Good Foods recipe data — `:play recipes`
- Airbnb listings data — `:play listings`
- Football (Soccer) transfer data — `:play football_transfers`

AuraDB Free example datasets

When creating a new database in Neo4j AuraDB, besides the default empty you can also select one of the starting datasets:

- Movies
- Graph based Recommendations
- Graphs for Cybersecurity
- StackOverflow Data

You can explore them following the Browser guides instructions and test data with suggested Cypher queries.

In addition, you have few options to download graph data into Aura from other Neo4j instances:

- Load a dump from Neo4j Sandbox backup.
- Load a dump from Neo4j Graph Example repository.
- Load a dump from Neo4j Desktop.

For more information, you can read the blog post Week 10 — Getting Dumps and Example Projects into Aura Free and watch the corresponding video from the series ***Discover Neo4j Aura Free with Michael and Alex***.

Neo4j Sandbox

To explore a wide variety of datasets in an **online setup** without a local installation, you can use the Neo4j sandbox.

Each sandbox is available for at least three days after creation and can also be remotely accessed from applications using any Neo4j driver.

Except for the "blank" sandbox, all other sandboxes come prepopulated with the domain data and focus on use case specific queries.

All sandboxes provide access to Neo4j Browser, Neo4j Bloom, APOC, Graph Data Science, neosemantics (n10s) and a GraphQL integration.

Neo4j Graph Example repository on Github

The data, browser guides, code examples (JavaScript, Java, Python, Go, C#), Cypher queries, Bloom perspectives for each sandbox are all available on GitHub repository.

The use cases range from

- movie recommendations (Repo)
- network management (Repo)
- investigative data from the ICIJ (Panama Papers) (Repo)
- crime investigation (Repo)
- social networks optionally using your own Twitter account (Repo).

Neo4j dataset demo server

Access information

If you need to explore more graph databases you can access the server on
https://demo.neo4jlabs.com:7473
This server hosts a number of datasets with read-only access for public use.
The username and password are the same as the database name.
For instance, for `recommendations` database the username is `recommendations` and password is `recommendations` too.

Hosted databases

You can open any of the following databases by clicking the link. Don't forget to copy the username and password.

- recommendations
- movies
- northwind
- fincen
- twitter
- stackoverflow
- gameofthrones
- neoflix
- wordnet

- slack

Means of data import

Loading data from source data

The most reliable way to get a dataset into Neo4j is to import it from the raw sources. Then you are independent of database versions, which you otherwise might have to upgrade. That's why we provided raw data (CSV, JSON, XML) for several of the datasets, accompanied by import scripts in Cypher.

You could run the Cypher script using a command-line client like `cypher-shell`.

Run Cypher Shell from the "Terminal" of your Graph Database in Neo4j Desktop

```
./bin/cypher-shell -u neo4j -p "password" -f import-file.cypher
```

You can also drag and drop or paste the script into Neo4j Browser (check that `multi-statement editor` is enabled in the settings) and run it from there.

CSV data can be imported using either `LOAD CSV` clause in Cypher or `neo4j-admin import` for initial bulk imports of large datasets.

For loading JSON, XML, XLS you need to have the APOC utility library installed, which comes with a number of procedures for importing data also from other databases.

Using a dump of a Neo4j database

Other datasets are provided as dump of a Neo4j datastore.

Community Edition (replace the default database)

1. Stop your Neo4j server.
2. Then you can import the file using the `./bin/neo4j-admin load --force true --from file.dump` command.
3. Start the Neo4j server.

Enterprise Edition (also Neo4j Desktop)

1. Import the file using the `./bin/neo4j-admin load --force true --from file.dump --database <dbname>` command.
2. Make the new database known to the system database with `CREATE DATABASE dbname` which will also automatically start it.

> ⚠ The Neo4j version of some of the datasets might be older than your Neo4j version. Then you might need to configure Neo4j to upgrade your database automatically, by setting `dbms.allow_upgrade=true` in your Neo4j settings, or directly in `$NEO4J_HOME/conf/neo4j.conf`

Large data dumps

Stack Overflow

This is a graph-import of the Stack Overflow archive with 16.4M questions, 52k tags and 8.9M users (Stack Overflow Dump (6.2GB)). This graph is pretty big, for global graph queries you'd need a page-cache of 6G and heap of 16G to work with it.

Here is an article explaining the data model and some exploratory analysis we ran on the data.

[0*IOrKWCLdILGG4BXe] | *//cdn-images-1.medium.com/max/1600/0*IOrKWCLdILGG4BXe.jpg*

The database is available in the Demo Server as outlined above.

Getting started resources

To help you along your path of learning more about Neo4j, we want to provide you with the resources which may encourage you to dive deeper into Neo4j products and graph technology.

Graph database concepts

- Video Series: Under the Hood
- Video Series: Intro to Graph Databases
- Free eBook: O'Reilly Graph Databases
- DZone: Graph Databases for Beginners

Graph for relational developers

- Free eBook: Relational to Graph
- DZone Refcard: From Relational to Graph
- Relational to Graph Data Modeling
- Neo4j ETL Tool
- New Data Importer and Neo4j Browser Updates

Cypher Query Language

- Cypher Query Language
- Cypher Cheat Sheet
- From SQL to Cypher

Graph for NoSQL developers

- DZone: NoSQL Database Types

Graph data models

- Featured GraphGists
- Start a Sandbox online

Training, courses, and webinars

Free, self-paced courses at GraphAcademy Online Training:

- Hands-on Neo4j Courses for Beginners
  - Neo4j Fundamentals Learn the basics of Neo4j and the property graph model (1 hour)
  - Cypher Fundamentals Learn Cypher in 60 minutes (1 hour)
  - Graph Data Modeling Fundamentals Learn how to design a Neo4j graph using best practices (2 hours)
  - Importing CSV Data into Neo4j Learn the basics of importing data into Neo4j (2 hours)
- Neo4j Courses for Developers
  - Building Neo4j Applications with Go Learn how to interact with Neo4j from your Go application using the Neo4j Go Driver
  - Building Neo4j Applications with Node.js Learn how to interact with Neo4j from Node.js using the Neo4j JavaScript Driver
  - Building Neo4j Applications with Python Learn how to interact with Neo4j from Python using the Neo4j Python Driver
- Neo4j Certifications
  - Neo4j 4.0 Certified Professional
  - Neo4j Graph Data Science Certification

Classroom and virtual training classes, webinars:

- Register for public virtual and classroom training
- Neo4j webinars: upcoming live and on-demand

Companies can also choose the instuctor-led training courses and request private, custom training events for internal staff to solve specific problems or understand concepts and architecture specific to their use case.

Neo4j events

- Check Neo4j Events calendar for upcoming live online events, trainings, and demos. You can select an event by its category, country, city, or language.

Neo4j Community resources

- Neo4j Community Forums are designed for learning and seeking guidance.

- Neo4j Discord Chat is a live chat environment (requires signup) for conversations with other Neo4j users.

When to use which? Read more about the differences between our forums and Discord.

Other resources

- Developer Blog
- Weekly Twitch stream sessions
- openCypher project
- Neo4j Customers
- Neo4j Features^
- Neo4j Editions

Tutorials

In this section, you find how-to guides and tutorials on different topics.

Overview

- Tutorial: Getting Started with Cypher explains the basic concepts of Cypher, Neo4j's query language, including how to create and query graphs. This tutorial is based on *the Movie Graph*. You'll find out how to create, query, and delete data in Neo4j.
- Tutorial: Build a Cypher Recommendation Engine uses examples from *the Movie Graph* and shows how to create recommendation algorithms with Cypher statements.
- Tutorial: Import data from a relational database into Neo4j shows the process for moving the data from a relational database into a graph database by translating the schema and using import tools.
- How-To: Import CSV data with Neo4j Desktop walks through how to import the data into a graph with **Neo4j Desktop** — a user-friendly interface for starting and creating Neo4j instances, adding or removing plugins, changing configurations, and other functionality.

Tutorial: Getting Started with Cypher

> *This tutorial explains the basic concepts of Cypher, Neo4j's query language, including how to create and query graphs. You should be able to read and understand Cypher queries after finishing this tutorial.*

Pop culture connections

The Movie Graph is a mini graph application containing actors and directors that are related through the movies they've collaborated on.

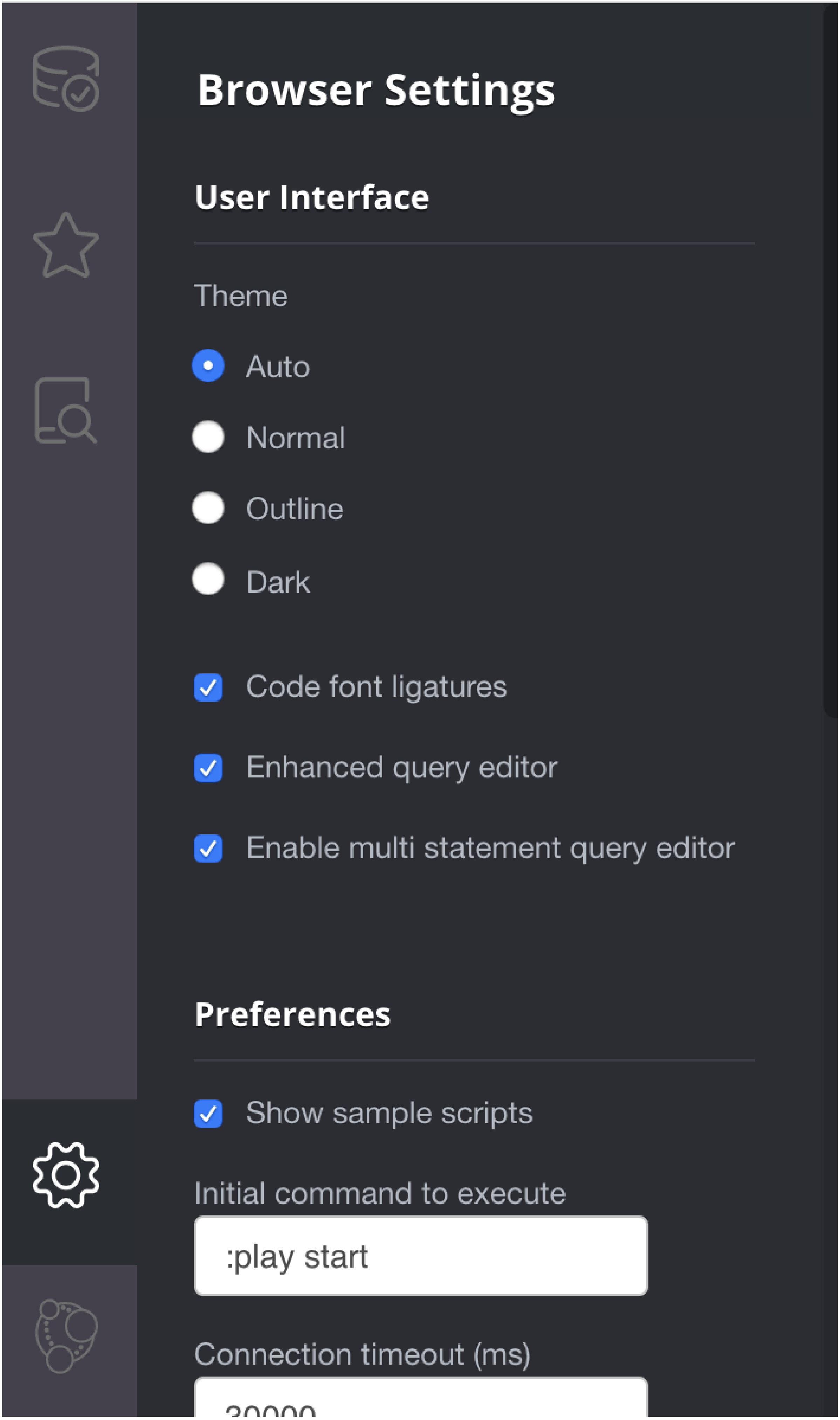
It is helpful if you run the queries and cypher code to create data as you follow this tutorial.

This tutorial will show you how to:

1. Create: Insert movie data into the graph.
2. Find: Retrieve individual movies and actors.
3. Query: Find patterns in the graph.
4. Solve: Answer some questions about the graph.

Create the Movie Graph

1. Create and start a new Neo4j database.
    a. Create a blank sandbox at https://sandbox.neo4j.com or..
    b. Create a new database in Neo4j Desktop:
        i. Create a new project.
        ii. Add a database to the project.
        iii. Start the database.
2. Open Neo4j Browser.
3. Set the browser settings to allow multi-statements:



Browser Settings

User Interface

Theme

● Auto

○ Normal

○ Outline

○ Dark

☑ Code font ligatures

☑ Enhanced query editor

☑ Enable multi statement query editor

Preferences

☑ Show sample scripts

Initial command to execute

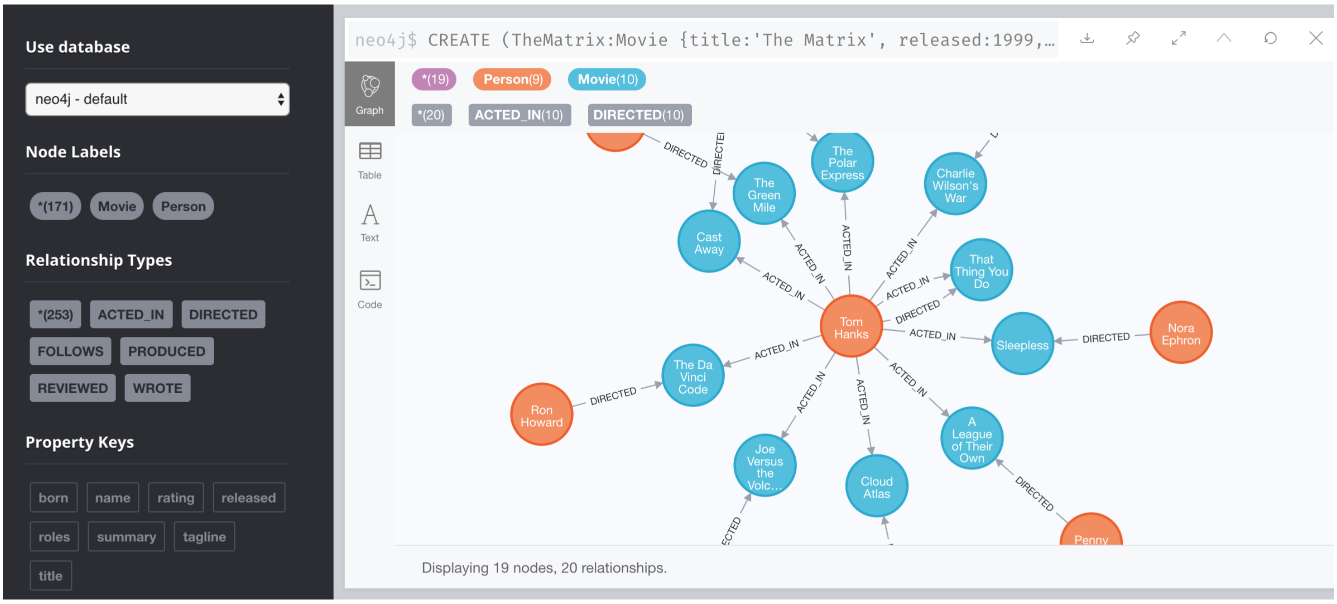
:play start

Connection timeout (ms)

30000

4. Enter `:guide movie-graph` in the query pane and click the "Play" button on the right. A new window opens below the query pane with the browser guide.
5. Go to page 2 of the browser guide.
6. Click on the Cypher code block which will bring it into the query pane and click the "Play" button.

This is what you should see in Neo4j Browser after loading the movie graph:

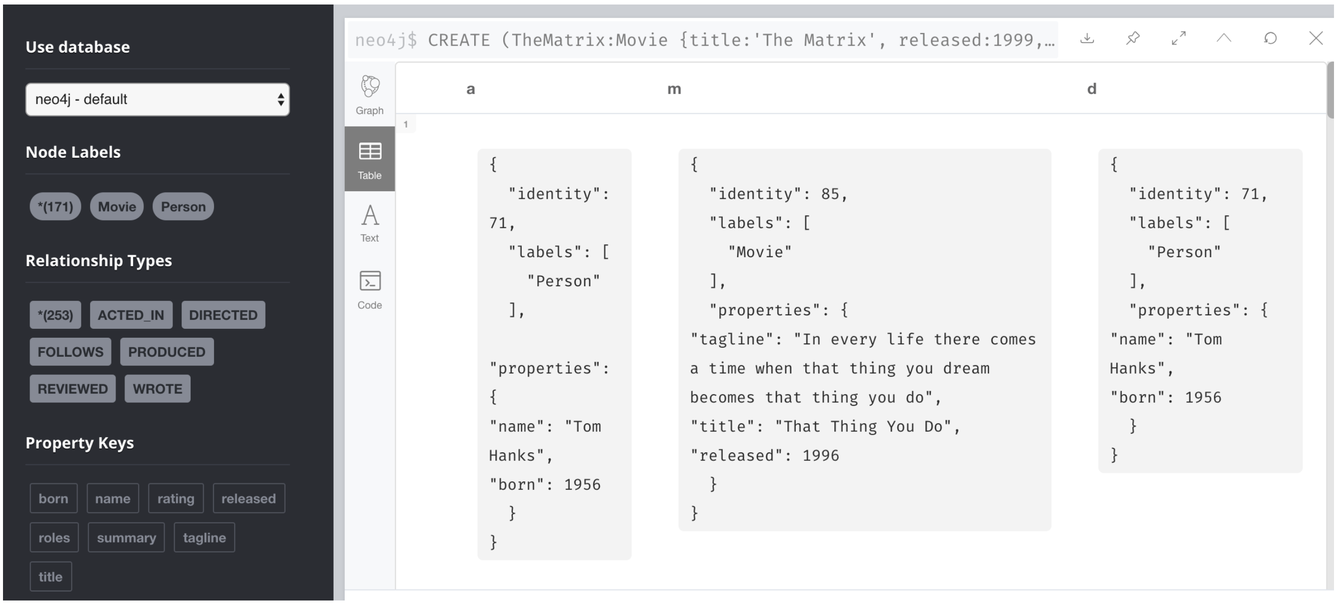


```
neo4j$ CREATE (TheMatrix:Movie {title:'The Matrix', released:1999,…
```

Displaying 19 nodes, 20 relationships.

This is the graph view of some of the data returned.

If you want to see the table view of the data returned, you click the table icon on the left:



```
neo4j$ CREATE (TheMatrix:Movie {title:'The Matrix', released:1999,…
```

a	m	d

{
"identity": 71,
"labels": [
"Person"
],
"properties": {
"name": "Tom Hanks",
"born": 1956
}
}
	{
	"identity": 85,
	"labels": [
	"Movie"
	],
	"properties": {
	"tagline": "In every life there comes a time when that thing you dream becomes that thing you do",
	"title": "That Thing You Do",
	"released": 1996
	}
	}
		{
		"identity": 71,
		"labels": [
		"Person"
		],
		"properties": {
		"name": "Tom Hanks",
		"born": 1956
		}
		}

How you view the results will also depend on the data returned. If the query returns nodes, then you can view the data as a graph. If the query returns property values, you can only view the data as a table.

If you need help:

```
:help cypher
```

When you run Cypher code in the query pane, it always creates a new pane with the results below the query pane.

Find actors and movies

Next, you will learn about queries for finding individual nodes.

1. Look at every query example
2. Run the query with the play button
3. Notice the syntax pattern
4. Try looking for other movies or actors

If you need help with syntax:

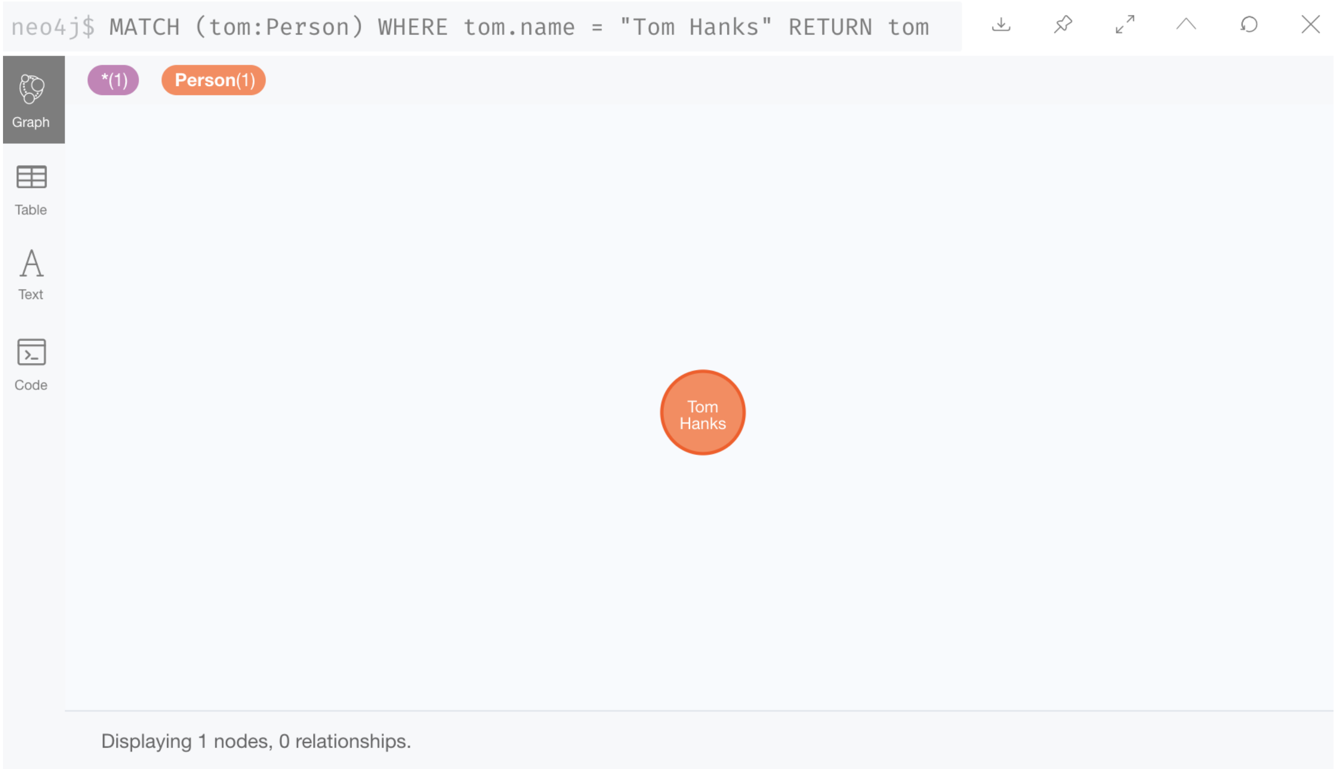
`:help MATCH`, `:help WHERE`, and `:help RETURN`

Find the person named "Tom Hanks"...

Copy and paste this code into the query pane and execute it:

```
MATCH (tom:Person)
WHERE tom.name = "Tom Hanks"
RETURN tom
```

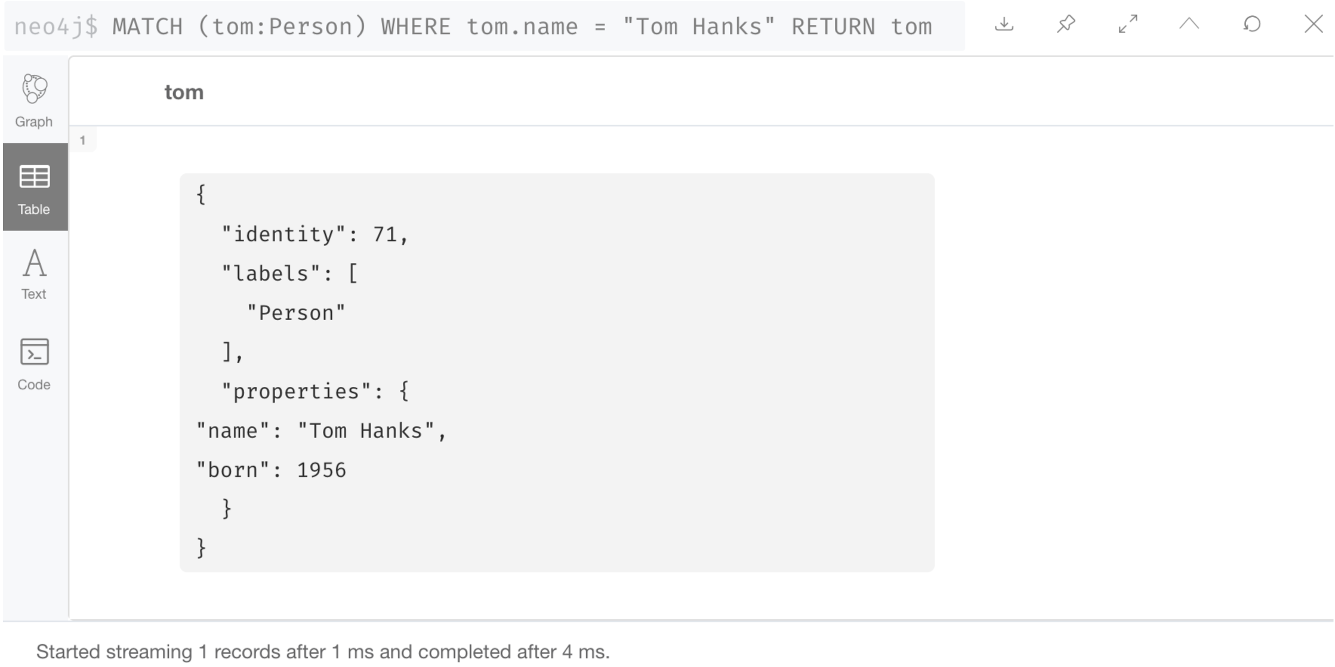
The graph result should look as follows:



```
neo4j$ MATCH (tom:Person) WHERE tom.name = "Tom Hanks" RETURN tom
```

Displaying 1 nodes, 0 relationships.

You can also view the properties of the node with the table view:



```
neo4j$ MATCH (tom:Person) WHERE tom.name = "Tom Hanks" RETURN tom
```

tom

1

```
{
  "identity": 71,
  "labels": [
    "Person"
  ],
  "properties": {
"name": "Tom Hanks",
"born": 1956
  }
}
```

Started streaming 1 records after 1 ms and completed after 4 ms.

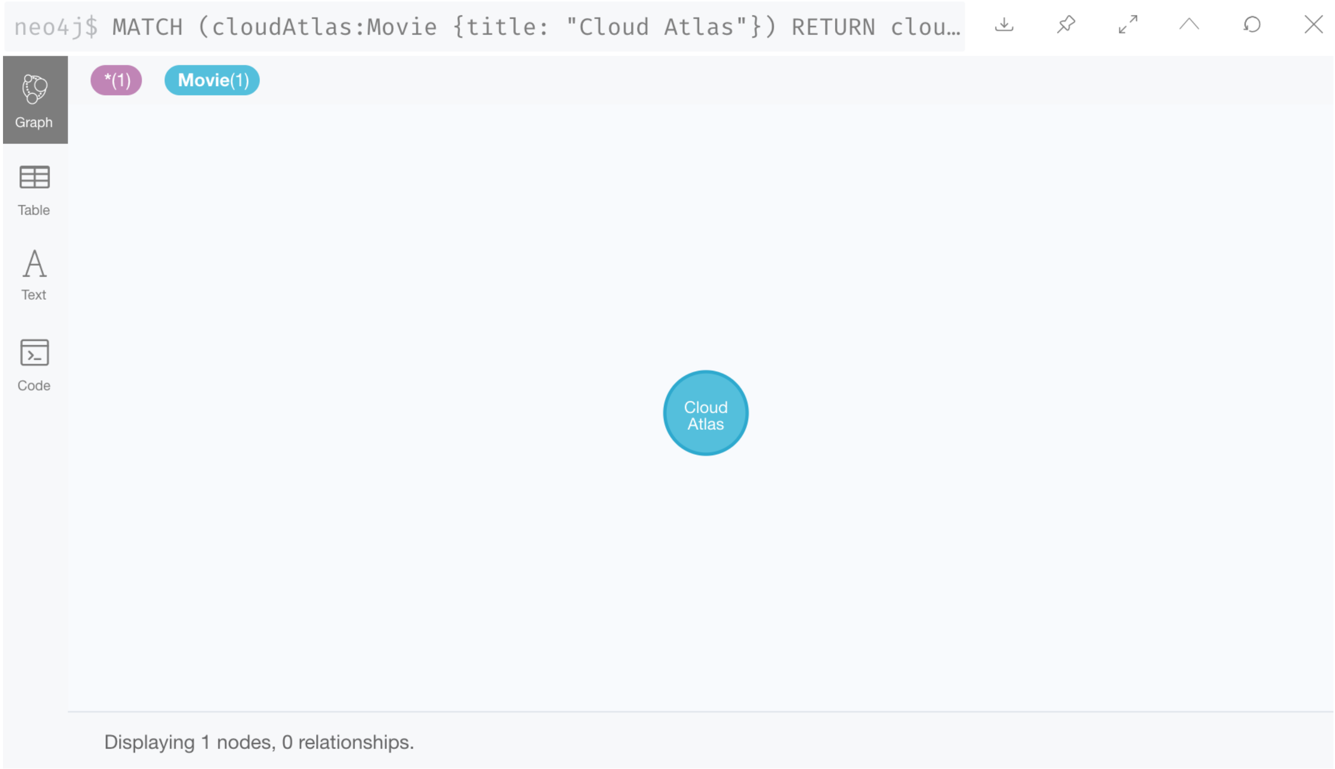
Find the movie titled "Cloud Atlas"...

Here we filter the query a different way where we specify the value in the node specification, rather than using a `WHERE` clause.

Copy and paste this code into the query pane and execute it:

```
MATCH (cloudAtlas:Movie {title: "Cloud Atlas"})
RETURN cloudAtlas
```

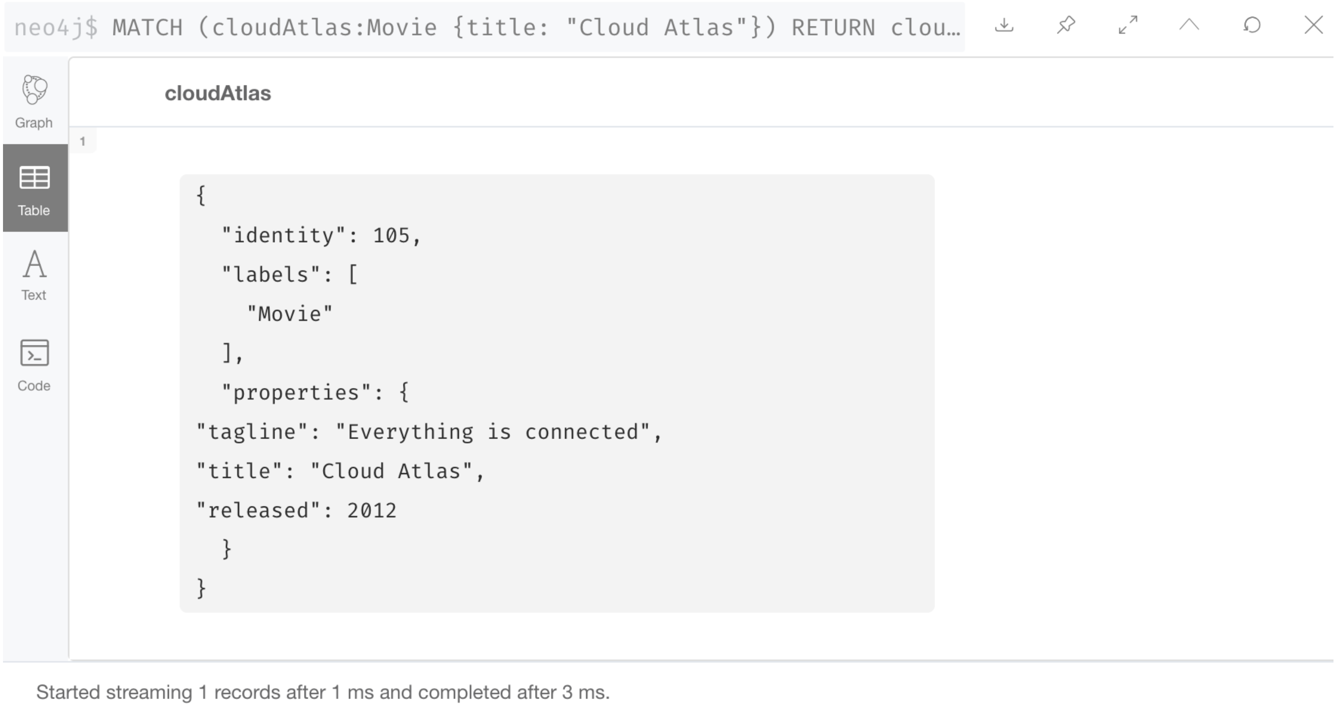
Here is the result of this query:



```
neo4j$ MATCH (cloudAtlas:Movie {title: "Cloud Atlas"}) RETURN clou…
```

Displaying 1 nodes, 0 relationships.

And here is the table view:



```
neo4j$ MATCH (cloudAtlas:Movie {title: "Cloud Atlas"}) RETURN clou…
```

cloudAtlas

1

```
{
  "identity": 105,
  "labels": [
    "Movie"
  ],
  "properties": {
"tagline": "Everything is connected",
"title": "Cloud Atlas",
"released": 2012
  }
}
```

Started streaming 1 records after 1 ms and completed after 3 ms.

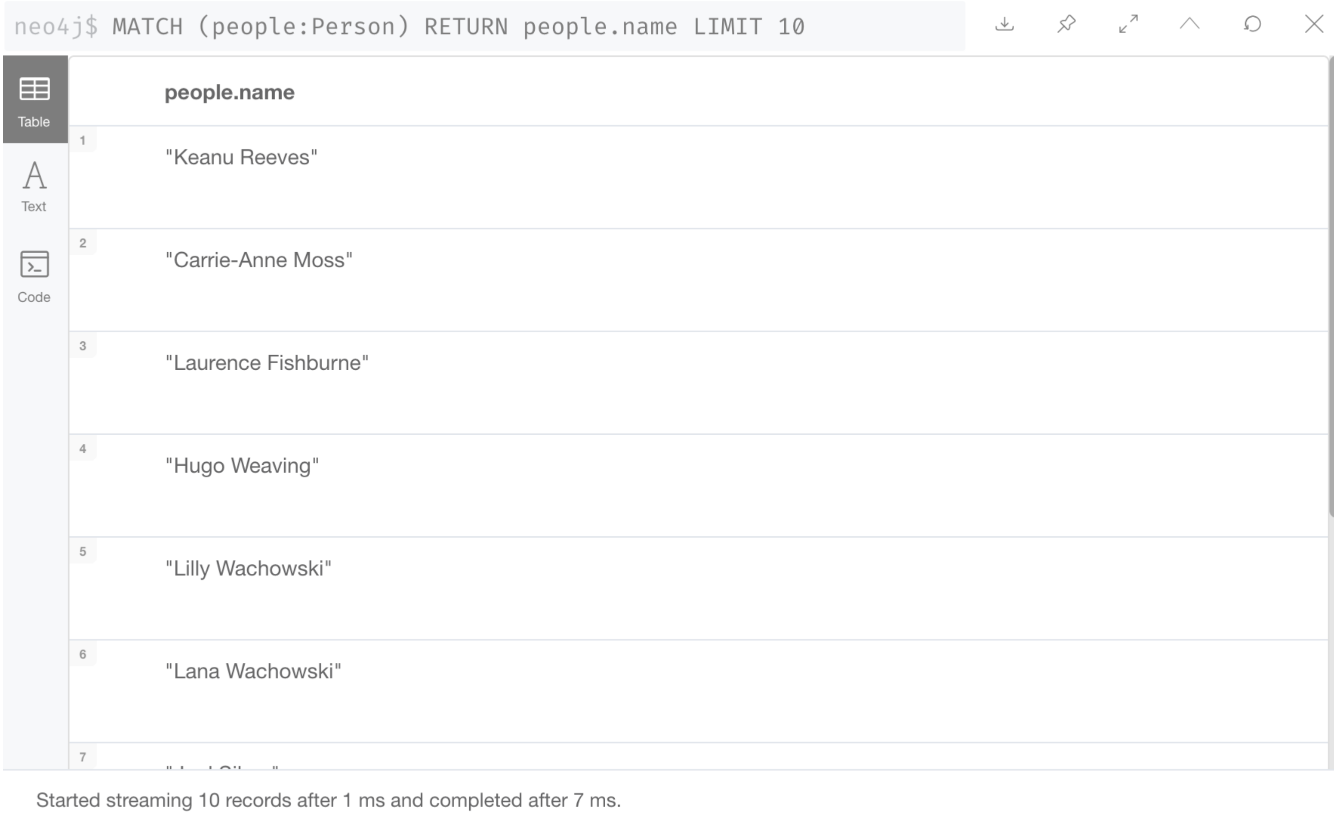
Find 10 people...

Next we want to find the names of 10 people in the graph. This code finds all *Person* nodes in the graph but just returns the *name* property value for 10 of them.

Copy and paste this code into the query pane and execute it:

```
MATCH (people:Person)
RETURN people.name LIMIT 10
```

Here is the result of this query:



```
neo4j$ MATCH (people:Person) RETURN people.name LIMIT 10
```

	people.name
1	"Keanu Reeves"
2	"Carrie-Anne Moss"
3	"Laurence Fishburne"
4	"Hugo Weaving"
5	"Lilly Wachowski"
6	"Lana Wachowski"
7	"Joel Silver"

Started streaming 10 records after 1 ms and completed after 7 ms.

For this query, property values are returned and you can only view the results as a table.

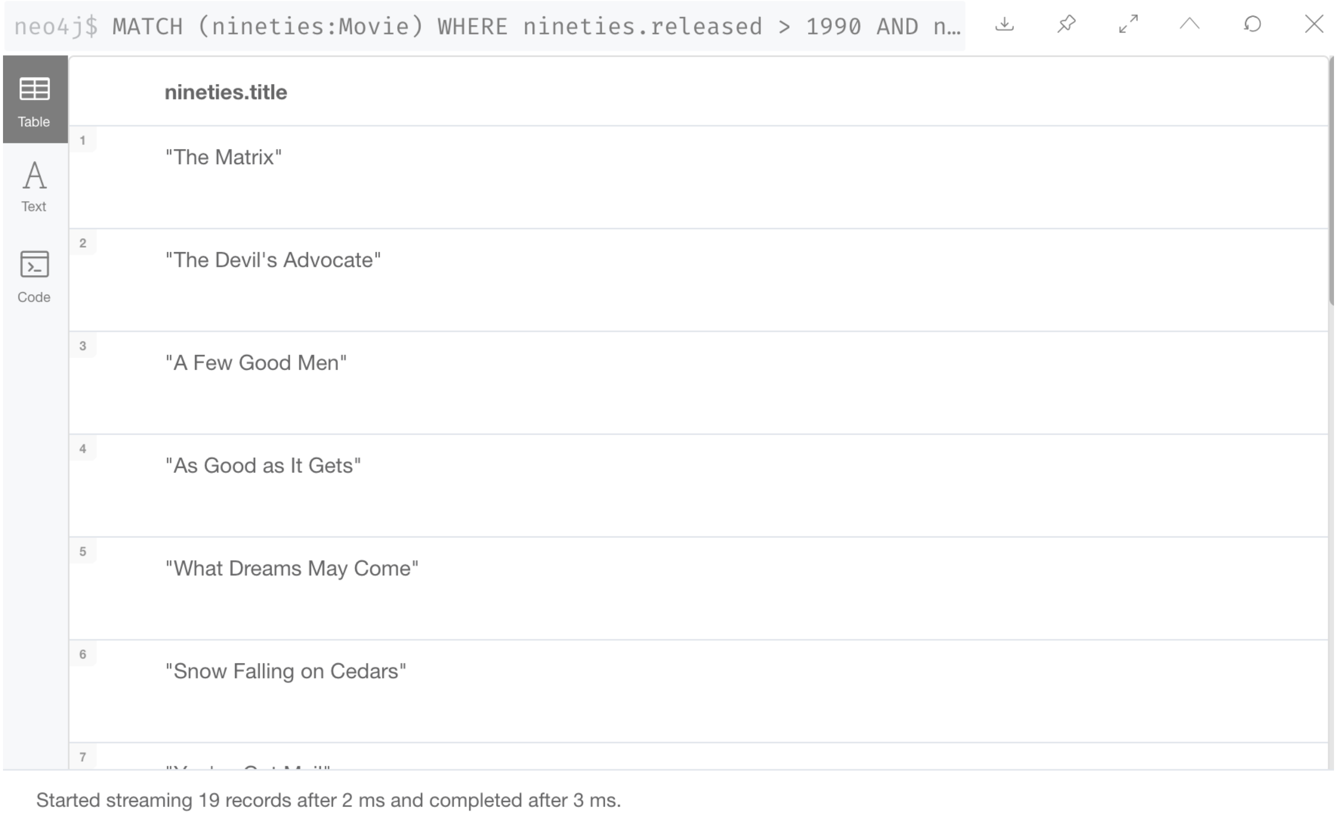
Find movies released in the 1990s…

Here is a query where we specify a range of values for selecting the *Movie* nodes to retrieve. Then we return the titles of these *Movie* nodes.

Copy and paste this code into the query pane and execute it:

```
MATCH (nineties:Movie)
WHERE nineties.released > 1990 AND nineties.released < 2000
RETURN nineties.title
```

Here is the result of this query:



```
neo4j$ MATCH (nineties:Movie) WHERE nineties.released > 1990 AND n…
```

	nineties.title
1	"The Matrix"
2	"The Devil's Advocate"
3	"A Few Good Men"
4	"As Good as It Gets"
5	"What Dreams May Come"
6	"Snow Falling on Cedars"
7	"You've Got Mail"

Started streaming 19 records after 2 ms and completed after 3 ms.

Find patterns in the graph

Thus far, you have queried the graph for nodes. Next, you will gain experience retrieving related nodes.

You will execute Cypher code to find patterns within the graph.

1. Actors are people who acted in movies.
2. Directors are people who directed a movie.
3. What other relationships exist?

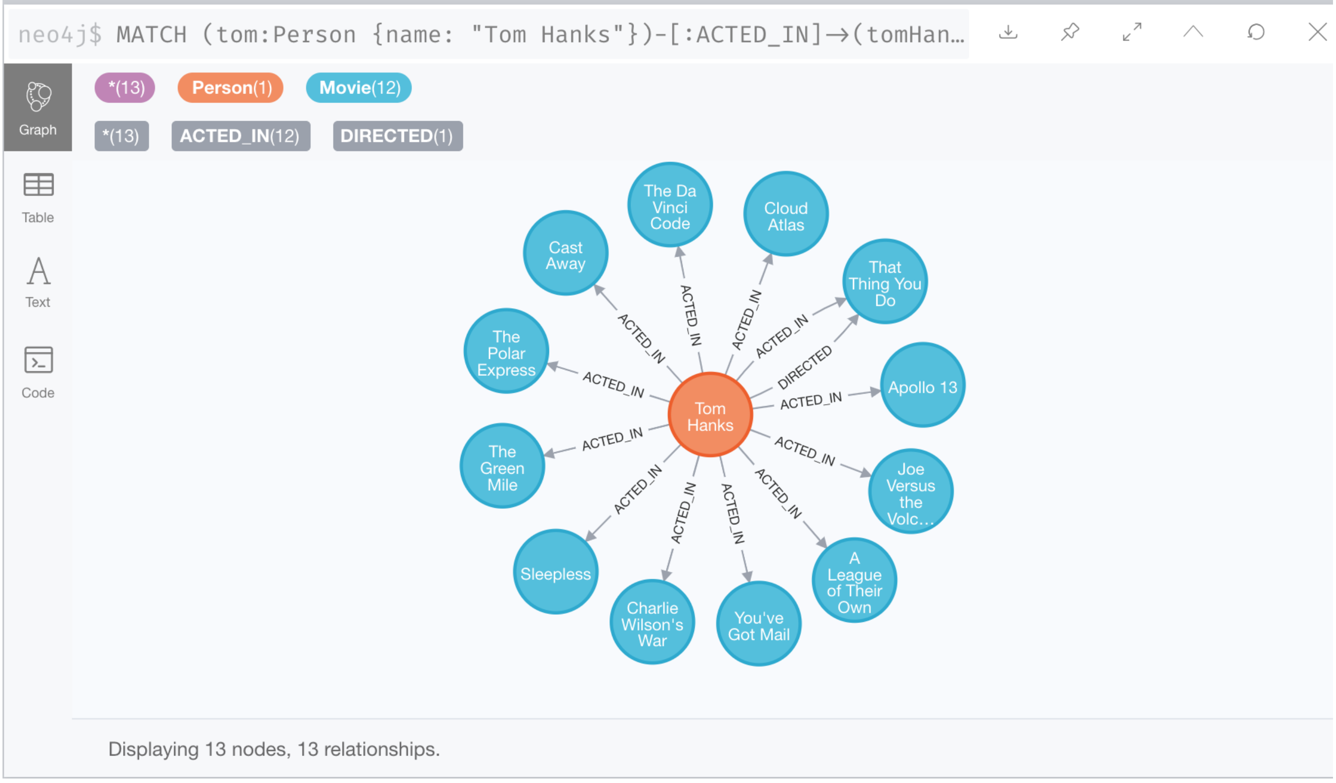
List all Tom Hanks movies...

Here is a query where we want to return the *Person* node for the actor Tom Hanks and we also want to return all *Movie* nodes that have the *ACTED_IN* relationship to Tom Hanks. That is, all movies that Tom Hanks acted in.

Copy and paste this code into the query pane and execute it:

```
MATCH (tom:Person {name: "Tom Hanks"})-[:ACTED_IN]->(tomHanksMovies)
RETURN tom,tomHanksMovies
```

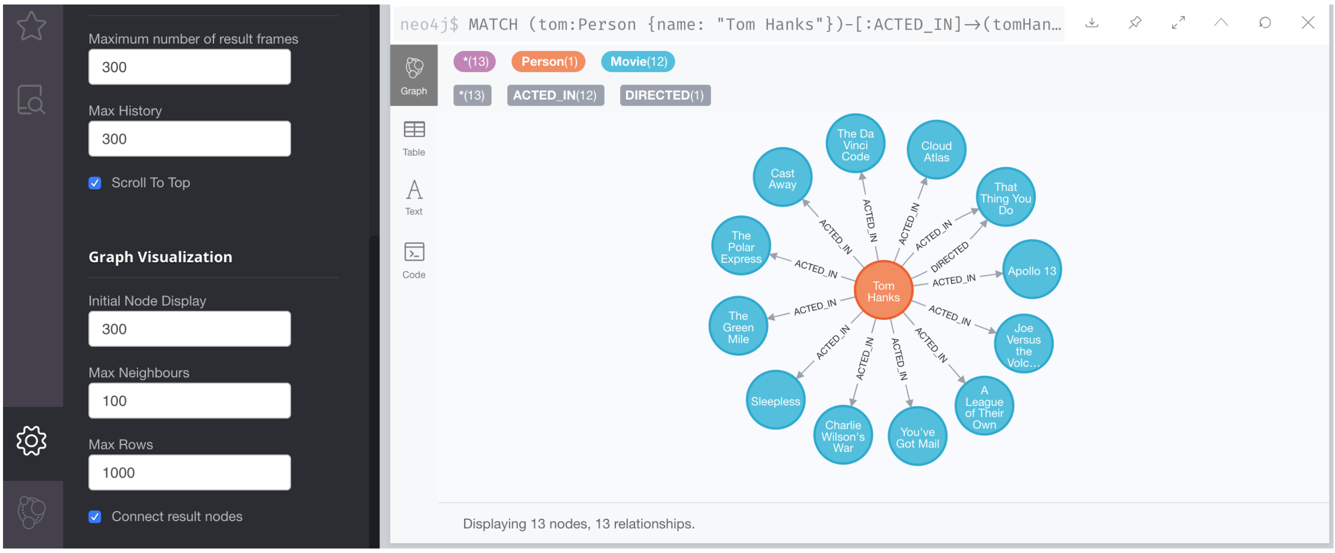
Here is the result of this query:



```
neo4j$ MATCH (tom:Person {name: "Tom Hanks"})-[:ACTED_IN]→(tomHan…
```

Displaying 13 nodes, 13 relationships.

Notice here that we also see the *DIRECTED* relationships between the Tom Hanks node and the *Movie* nodes. This is because we have a setting in our Neo4j Browser where result nodes will be connected:



Maximum number of result frames: 300

Max History: 300

☑ Scroll To Top

Graph Visualization

Initial Node Display: 300

Max Neighbours: 100

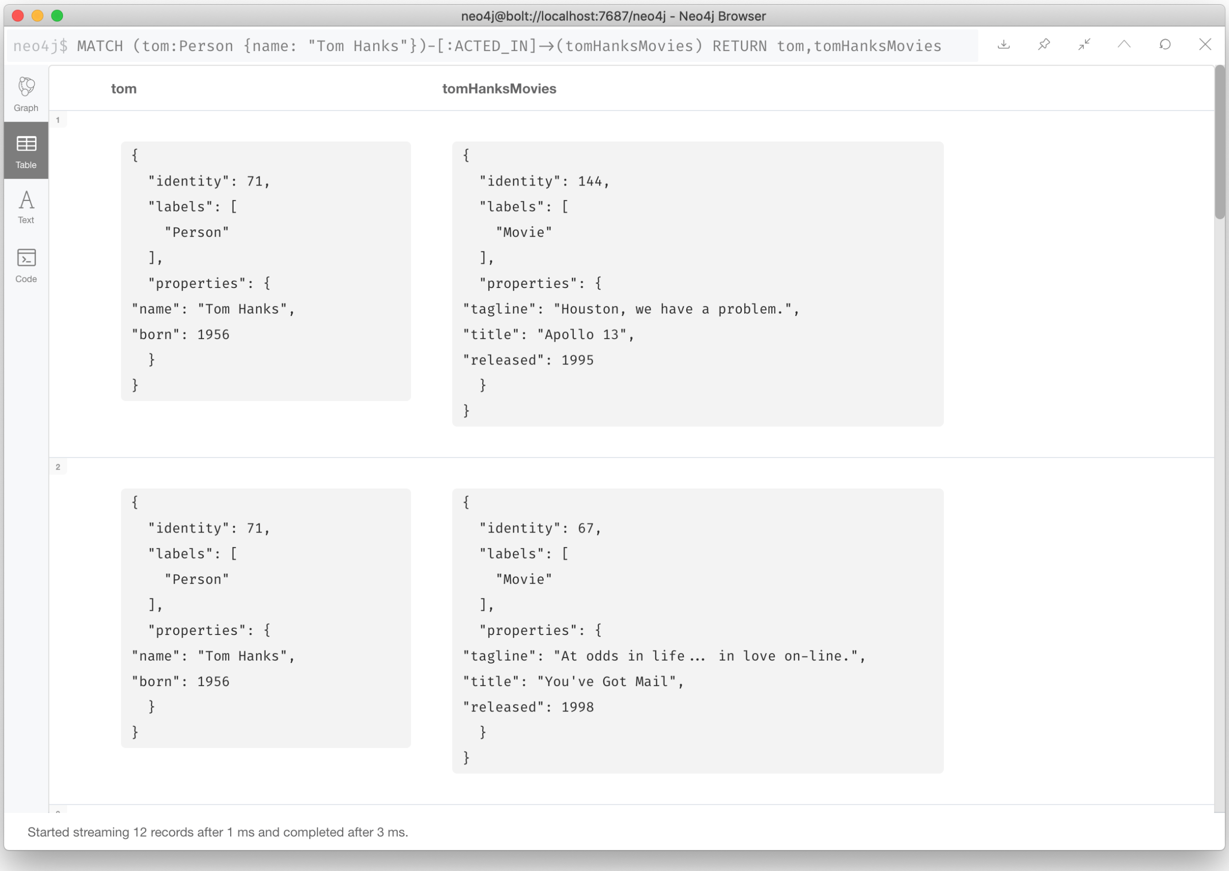
Max Rows: 1000

☑ Connect result nodes

```
neo4j$ MATCH (tom:Person {name: "Tom Hanks"})-[:ACTED_IN]→(tomHan…
```

Displaying 13 nodes, 13 relationships.

And here is the table view:



```
neo4j$ MATCH (tom:Person {name: "Tom Hanks"})-[:ACTED_IN]→(tomHanksMovies) RETURN tom,tomHanksMovies
```

	tom	tomHanksMovies
1	{ "identity": 71, "labels": ["Person"], "properties": { "name": "Tom Hanks", "born": 1956 } }	{ "identity": 144, "labels": ["Movie"], "properties": { "tagline": "Houston, we have a problem.", "title": "Apollo 13", "released": 1995 } }
2	{ "identity": 71, "labels": ["Person"], "properties": { "name": "Tom Hanks", "born": 1956 } }	{ "identity": 67, "labels": ["Movie"], "properties": { "tagline": "At odds in life … in love on-line.", "title": "You've Got Mail", "released": 1998 } }

Started streaming 12 records after 1 ms and completed after 3 ms.

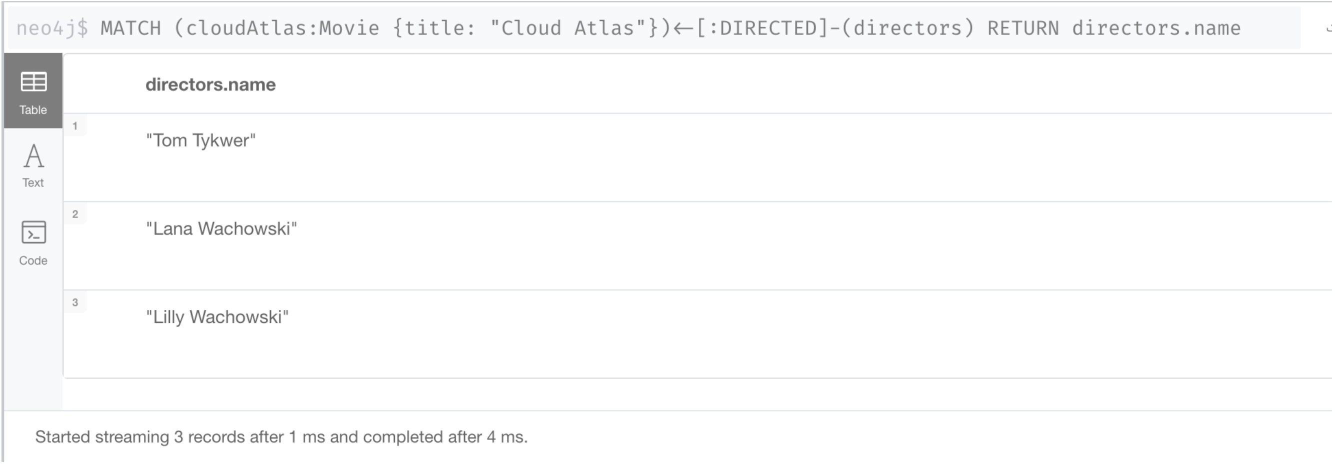
Who directed "Cloud Atlas"?

Here is a query where we want to return the nodes that have the *DIRECTED* relationship to the Cloud Atlas *Movie* node. It will return the names of the people who directed the movie.

Copy and paste this code into the query pane and execute it:

```
MATCH (cloudAtlas:Movie {title: "Cloud Atlas"})<-[:DIRECTED]-(directors)
RETURN directors.name
```

Here is the result of this query:



```
neo4j$ MATCH (cloudAtlas:Movie {title: "Cloud Atlas"})←[:DIRECTED]-(directors) RETURN directors.name
```

	directors.name
1	"Tom Tykwer"
2	"Lana Wachowski"
3	"Lilly Wachowski"

Started streaming 3 records after 1 ms and completed after 4 ms.

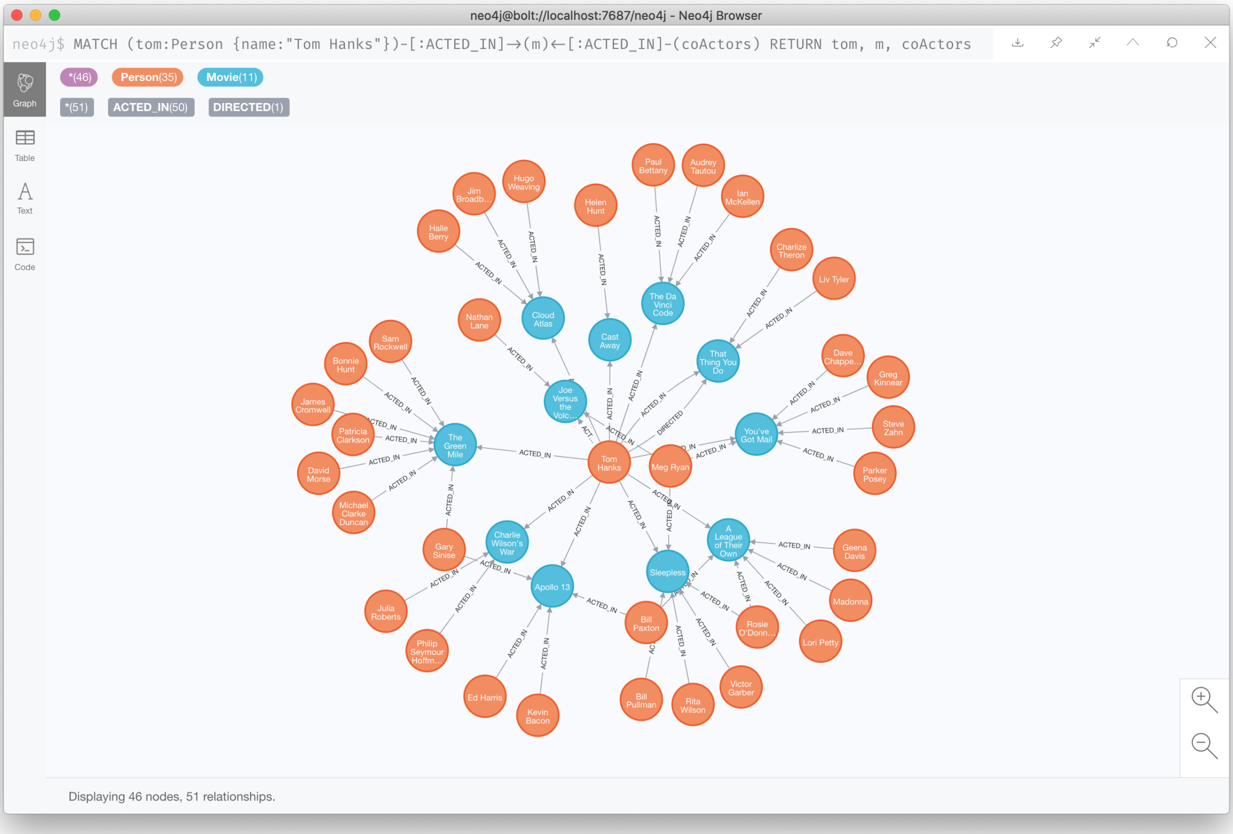
Tom Hanks' co-actors...

Next, we want to find all movies that Tom Hanks acted in and for each movie retrieved, also find the people who acted in that movie.

Copy and paste this code into the query pane and execute it:

```
MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors)
RETURN tom, m, coActors
```

Here is the result of this query:



neo4j@bolt://localhost:7687/neo4j - Neo4j Browser

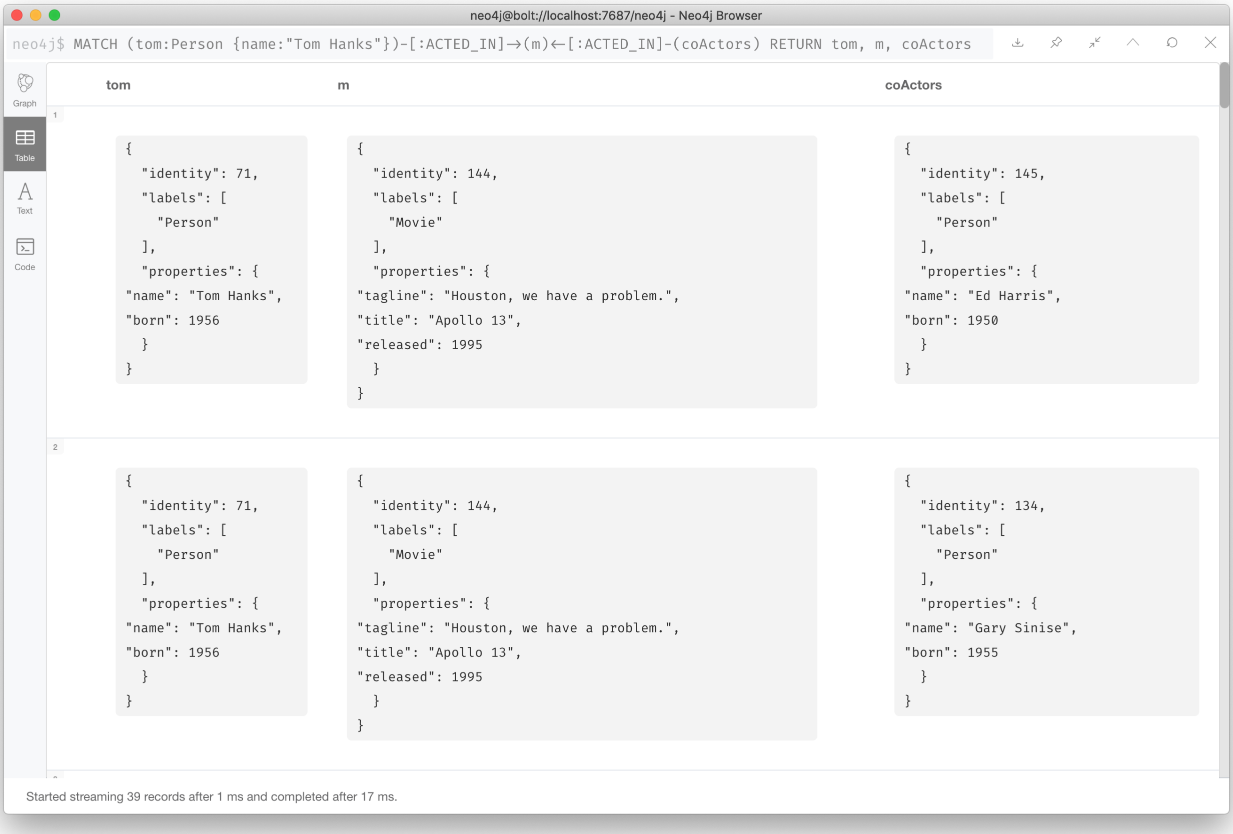
```
neo4j$ MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]→(m)←[:ACTED_IN]-(coActors) RETURN tom, m, coActors
```

*(46) Person(35) Movie(11)

*(51) ACTED_IN(50) DIRECTED(1)

Displaying 46 nodes, 51 relationships.

And here is the table view:



```
neo4j$ MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]→(m)←[:ACTED_IN]-(coActors) RETURN tom, m, coActors
```

```
tom                         m                                           coActors

1
{                           {                                           {
  "identity": 71,             "identity": 144,                            "identity": 145,
  "labels": [                 "labels": [                                 "labels": [
    "Person"                    "Movie"                                     "Person"
  ],                          ],                                          ],
  "properties": {             "properties": {                             "properties": {
"name": "Tom Hanks",        "tagline": "Houston, we have a problem.",   "name": "Ed Harris",
"born": 1956                "title": "Apollo 13",                       "born": 1950
  }                         "released": 1995                              }
}                             }                                         }
                            }

2
{                           {                                           {
  "identity": 71,             "identity": 144,                            "identity": 134,
  "labels": [                 "labels": [                                 "labels": [
    "Person"                    "Movie"                                     "Person"
  ],                          ],                                          ],
  "properties": {             "properties": {                             "properties": {
"name": "Tom Hanks",        "tagline": "Houston, we have a problem.",   "name": "Gary Sinise",
"born": 1956                "title": "Apollo 13",                       "born": 1955
  }                         "released": 1995                              }
}                             }                                         }
                            }
```

Started streaming 39 records after 1 ms and completed after 17 ms.

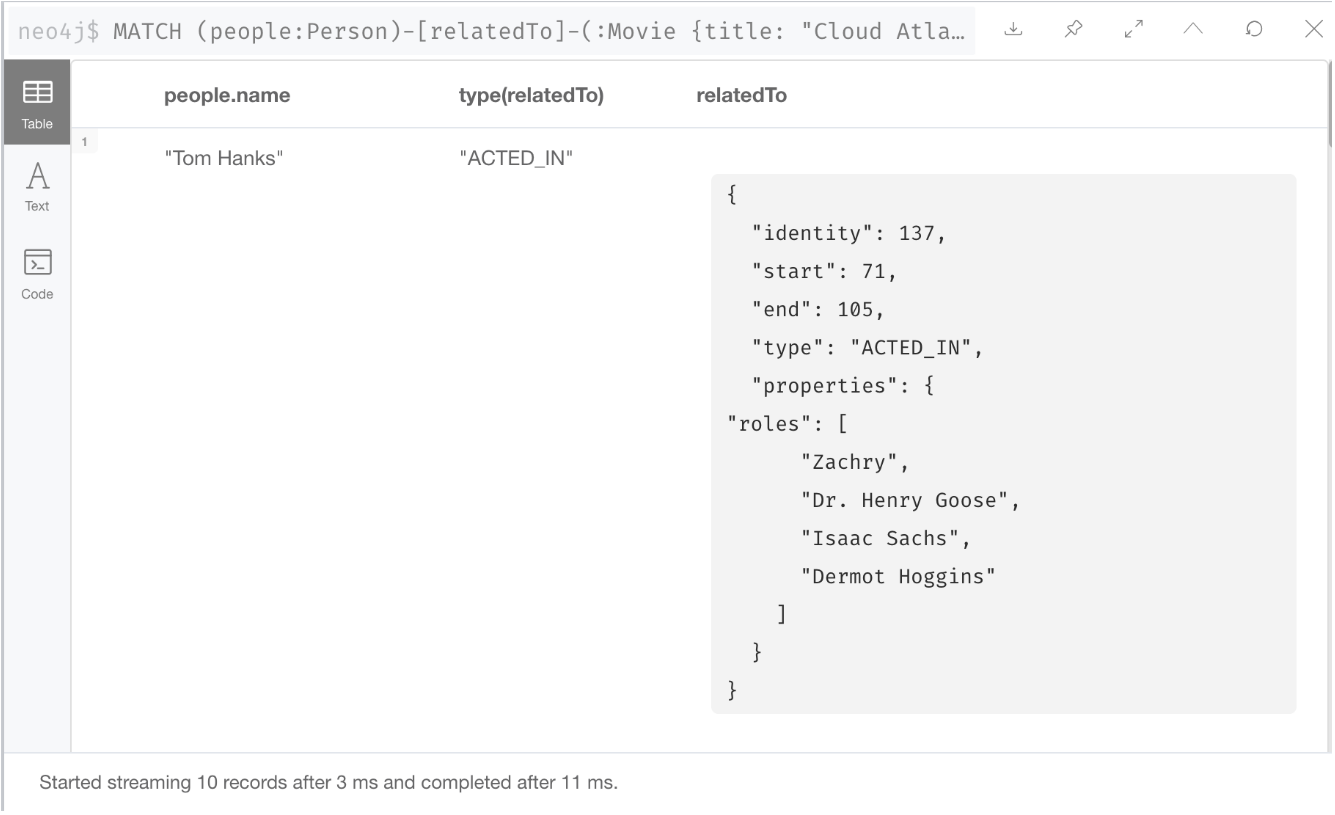
How people are related to "Cloud Atlas"...

Here is a query where we want to return information about the relationships to and from the Cloud Atlas movie. We find the related nodes and then we return the name of the person, the type of relationship, and the properties for that relationship.

Copy and paste this code into the query pane and execute it:

```
MATCH (people:Person)-[relatedTo]-(:Movie {title: "Cloud Atlas"})
RETURN people.name, type(relatedTo), relatedTo
```

Here is the result of this query:



```
neo4j$ MATCH (people:Person)-[relatedTo]-(:Movie {title: "Cloud Atla…
```

	people.name	type(relatedTo)	relatedTo
1	"Tom Hanks"	"ACTED_IN"	see below

```
{
  "identity": 137,
  "start": 71,
  "end": 105,
  "type": "ACTED_IN",
  "properties": {
"roles": [
      "Zachry",
      "Dr. Henry Goose",
      "Isaac Sachs",
      "Dermot Hoggins"
    ]
  }
}
```

Started streaming 10 records after 3 ms and completed after 11 ms.

Answer some questions about the graph

You've heard of the classic "Six Degrees of Kevin Bacon"? That is, find all people who are up to 6 hops away from Kevin Bacon in the graph. This is simply a shortest path query called the "Bacon Path". To perform this type of query, you need to specify:

- Variable length patterns: variable length relationships
- Built-in shortestPath() algorithm: shortestPath

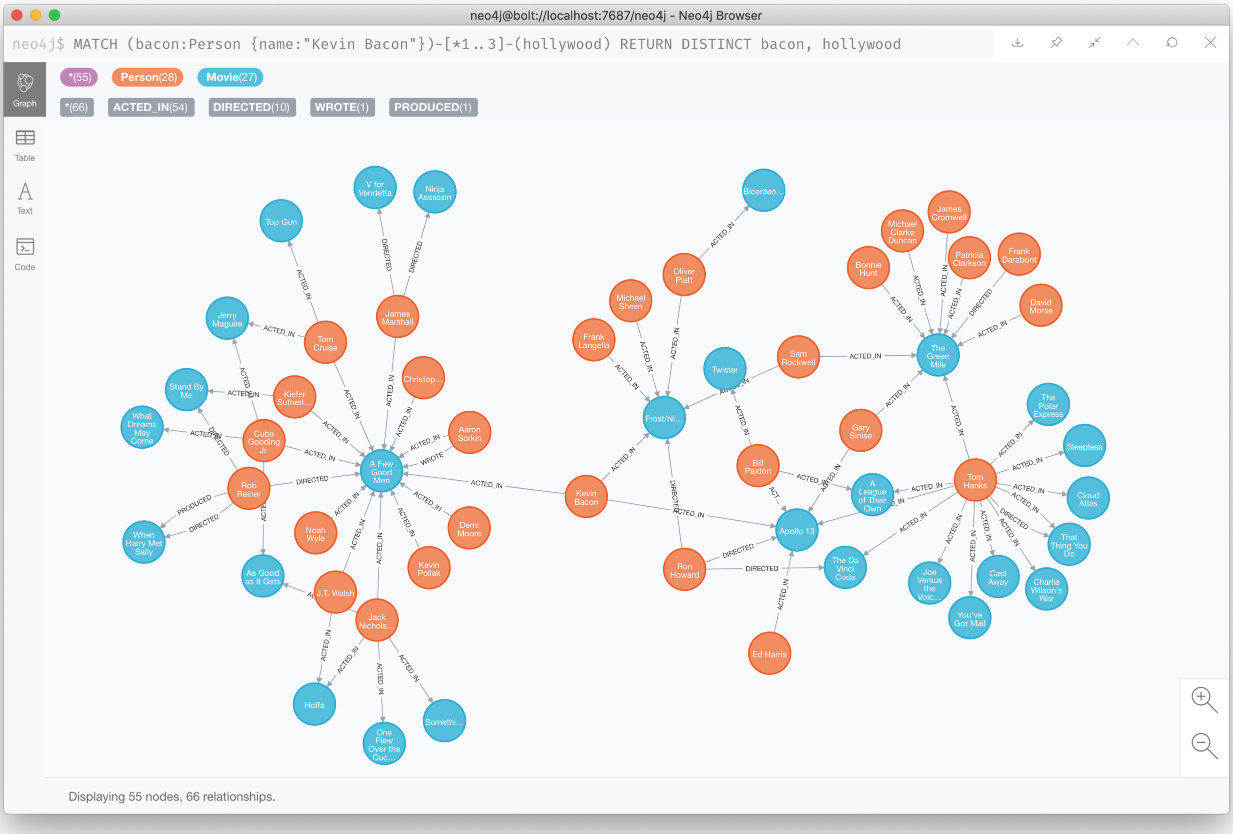
Movies and actors up to three hops away from Kevin Bacon

In our first query, we want to find all movies and/or people who are up to 3 hops away from Kevin Bacon in the graph.

Copy and paste this code into the query pane and execute it:

```
MATCH (bacon:Person {name:"Kevin Bacon"})-[*1..3]-(hollywood)
RETURN DISTINCT bacon, hollywood
```

Here is the result of this query:



```
neo4j$ MATCH (bacon:Person {name:"Kevin Bacon"})-[*1..3]-(hollywood) RETURN DISTINCT bacon, hollywood
```

```
Displaying 55 nodes, 66 relationships.
```

Find the Bacon Path to Meg Ryan

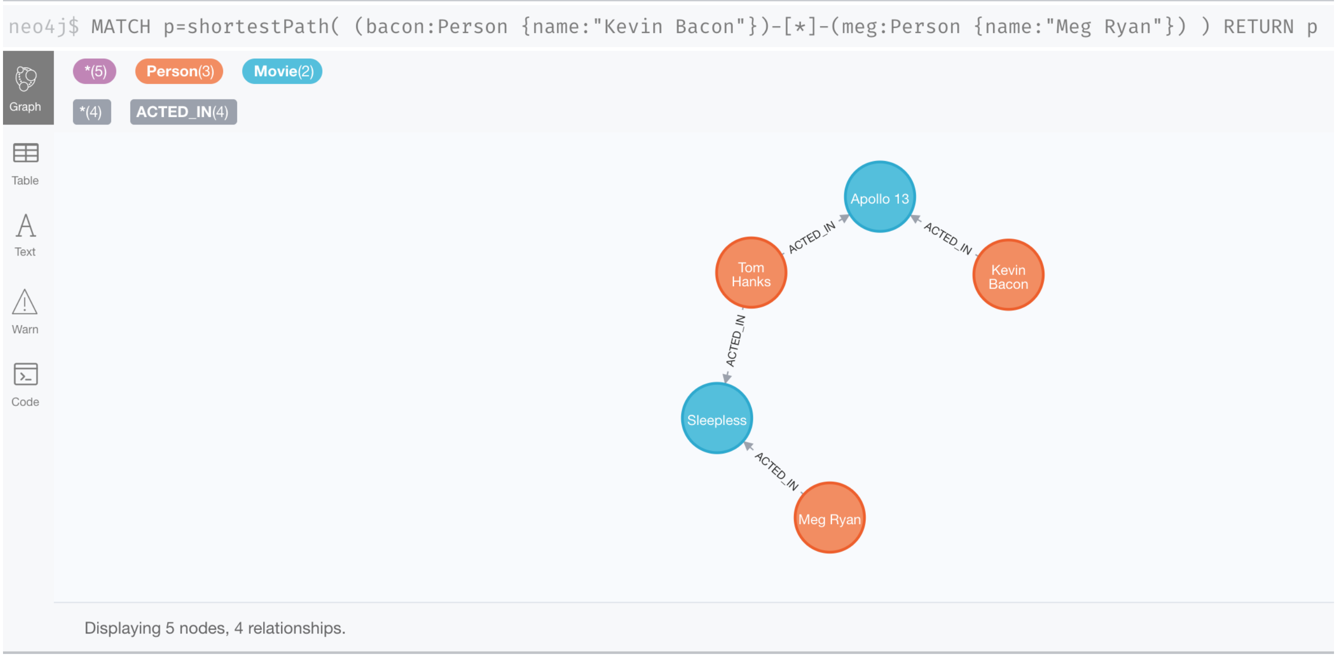
What is the shortest path between Kevin Bacon and Meg Ryan in the graph? In this Cypher, we are returning the path that includes nodes and relationships.

Copy and paste this code into the query pane and execute it:

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

Before you execute the query, you will see a warning that a relationship of '*' could take a long time to execute. Our movie graph is small, so you can ignore this warning.

Here is the result of this query:



```
neo4j$ MATCH p=shortestPath( (bacon:Person {name:"Kevin Bacon"})-[*]-(meg:Person {name:"Meg Ryan"}) ) RETURN p
```

Displaying 5 nodes, 4 relationships.

Clean up

When you're done experimenting, you can remove the movie data set.

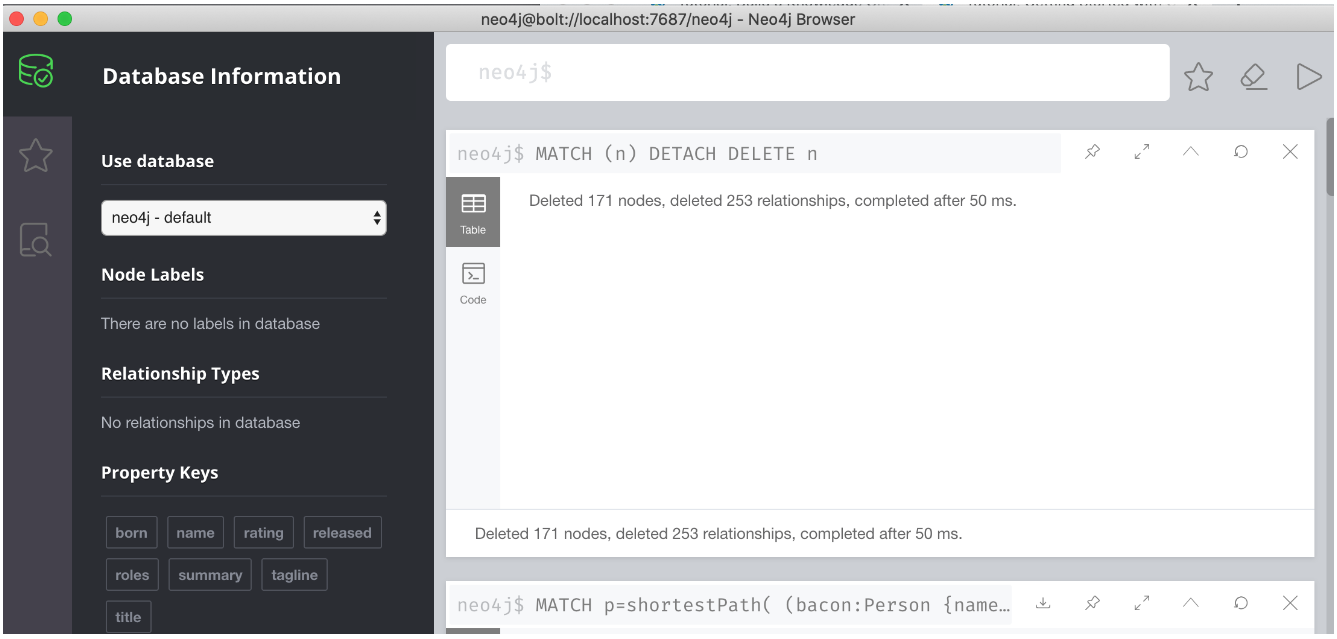
1. Nodes can't be deleted if relationships to them exist.
2. Delete both nodes and relationships together.

⚠ This will remove all nodes and relationships in the graph!

Copy and paste this code into the query pane and execute it:

```
MATCH (n)
DETACH DELETE n
```

Here is the result of this query:



neo4j@bolt://localhost:7687/neo4j - Neo4j Browser

Database Information

Use database

neo4j - default

Node Labels

There are no labels in database

Relationship Types

No relationships in database

Property Keys

born name rating released roles summary tagline title

```
neo4j$ MATCH (n) DETACH DELETE n
```

Deleted 171 nodes, deleted 253 relationships, completed after 50 ms.

Deleted 171 nodes, deleted 253 relationships, completed after 50 ms.

```
neo4j$ MATCH p=shortestPath( (bacon:Person {name…
```

Notice that although the database information in the left panel shows no nodes or relationships in the graph, the property key names remain.

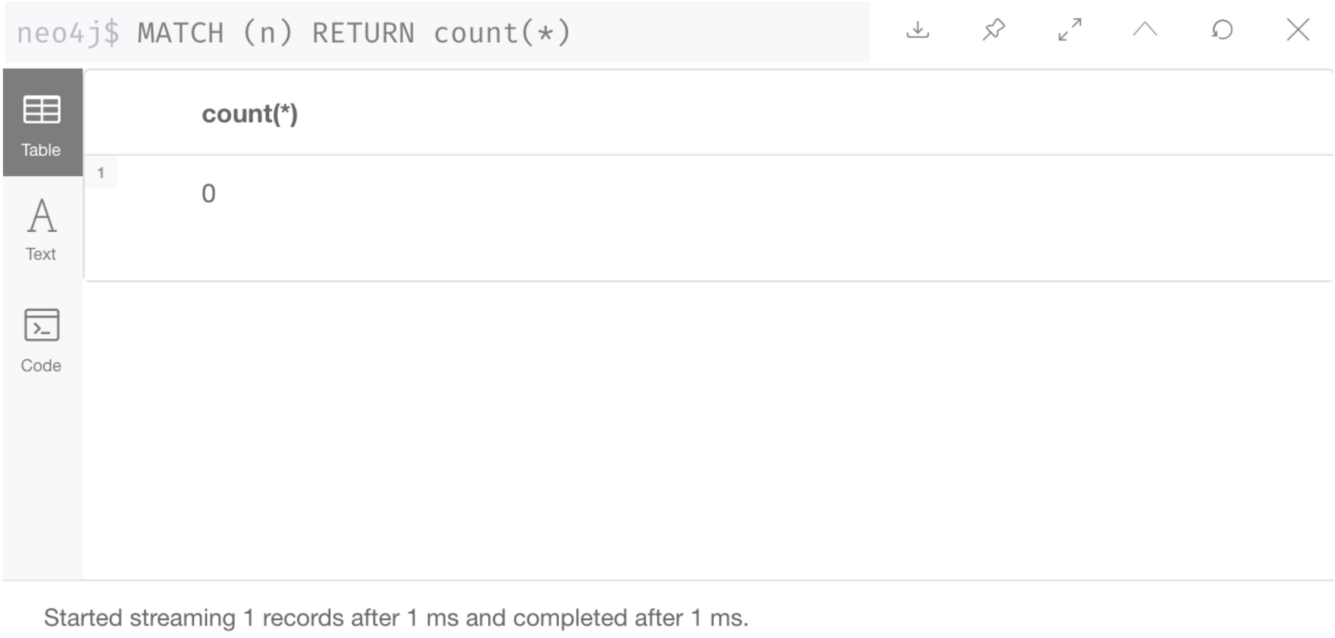
Verify that the movie graph data is gone

If you perform this query to retrieve all nodes in the graph and return the count, you should see a value of 0 returned.

Copy and paste this code into the query pane and execute it:

```
MATCH (n)
RETURN count(*)
```

Here is the result of this query:



```
neo4j$ MATCH (n) RETURN count(*)
```

	count(*)
1	0

Started streaming 1 records after 1 ms and completed after 1 ms.

Congratulations! You have learned how to use Cypher to query a Neo4j database.

Tutorial: Build a Cypher Recommendation Engine

Graphs are everywhere. By following the meaningful relationships between the people and movies, you can determine occurences of actors working together, the frequency of actors working with one another, and the movies they have in common in the graph. This is one way we can recommend movies to users, based on what they liked before, and their favorite actors. We will step you through everything you need to get started with AuraDB and Cypher, to solve a real-world problem.

Setting Up

When you've created your AuraDB account, click "Create a Database" and select a free database

[free database type] | *//dist.neo4j.com/wp-content/uploads/free-database-type.png*

Then, fill out the name, and choose a cloud region for your database and click "Create Database". Make sure "Learn about graphs with a movie dataset" is selected, so you’ll start with a dataset.

[recommendation engine free database] **|** *//dist.neo4j.com/wp-content/uploads/recommendation-engine-*

free-database.png

AuraDB will prompt you with the password for your new instance while it being set up. **Make sure to save the password for later steps.**

Once your database is running, open browser as shown below.

[open auradb browser] | *//dist.neo4j.com/wp-content/uploads/open-auradb-browser.png*

Now you've arrived inside of Neo4j Browser. Use your username and password (the one you captured above) to log in. You'll immediately notice a guide on the left-hand side that you can tab through to start out with some experimental queries. Any of these queries you see can be automatically put into the query execution box and run on the right hand side of the screen by clicking the little "play" button.

[first movies query] | *//dist.neo4j.com/wp-content/uploads/first-movies-query.png*

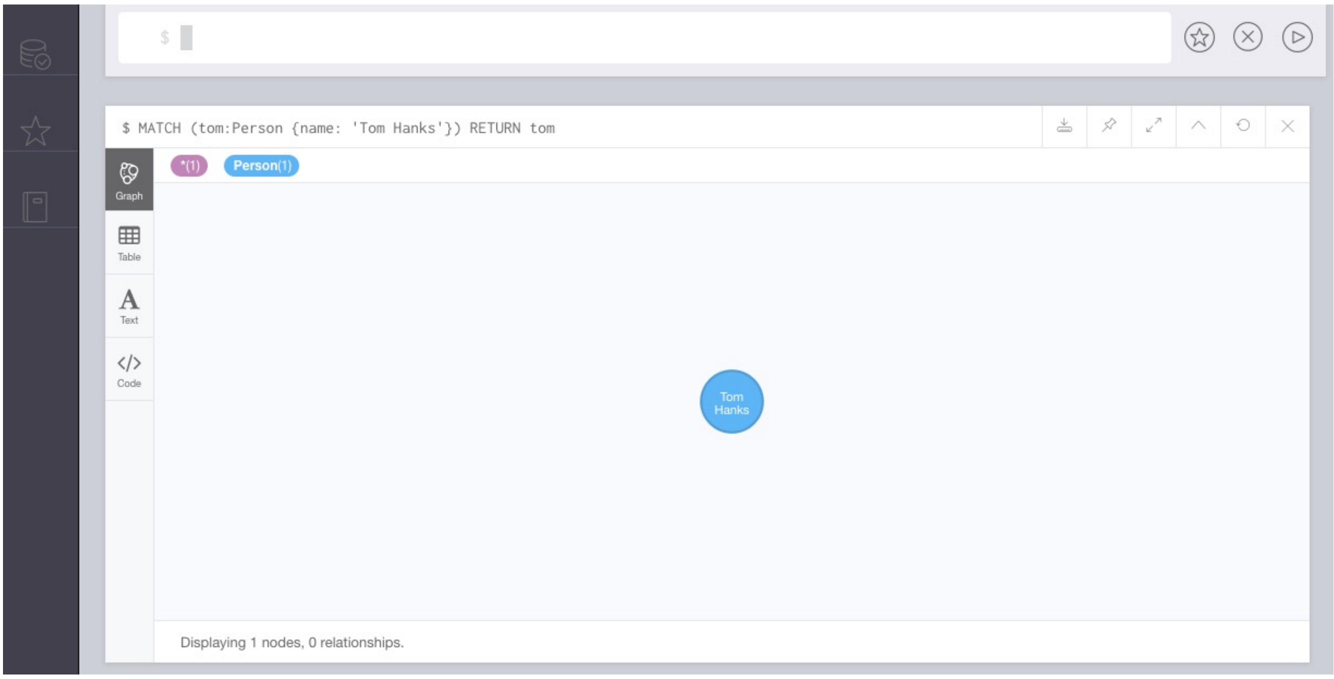
This first query just shows a few movies in the database to prove there's something there. Congratulations, you've got some data in a new database, and we're ready to get started.

The next section will show you how to write some queries to explore the data you just created.

Basic queries

Before we start recommending things, we need to find out what is interesting in our data to see what kinds of things we can and want to recommend. To start, let us run a query like this to find a single actor like *Tom Hanks*.

```
MATCH (tom:Person {name: 'Tom Hanks'})
RETURN tom
```

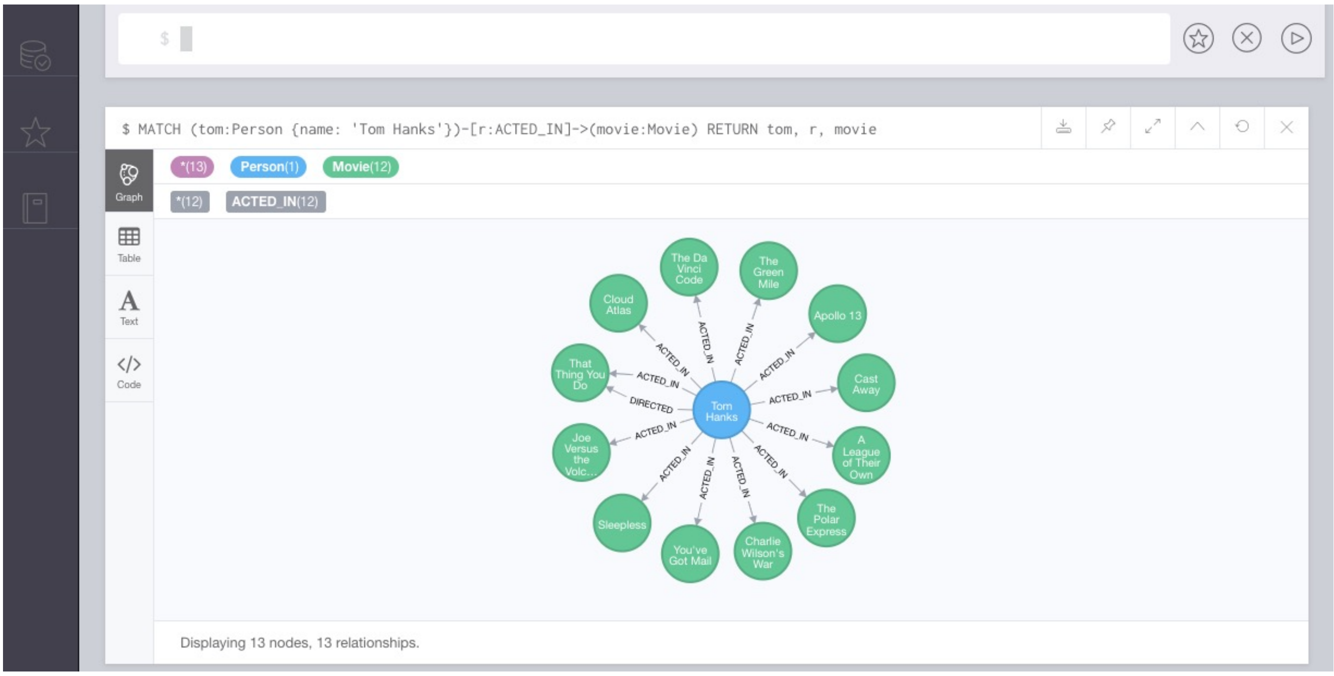


```
$ MATCH (tom:Person {name: 'Tom Hanks'}) RETURN tom
```

Displaying 1 nodes, 0 relationships.

Now that we found an actor we are interested in, we can retrieve all his movies by starting from the `Tom Hanks` node and following the `ACTED_IN` relationships. Your results should look like a graph.

```
MATCH (tom:Person {name: 'Tom Hanks'})-[r:ACTED_IN]->(movie:Movie)
RETURN tom, r, movie
```

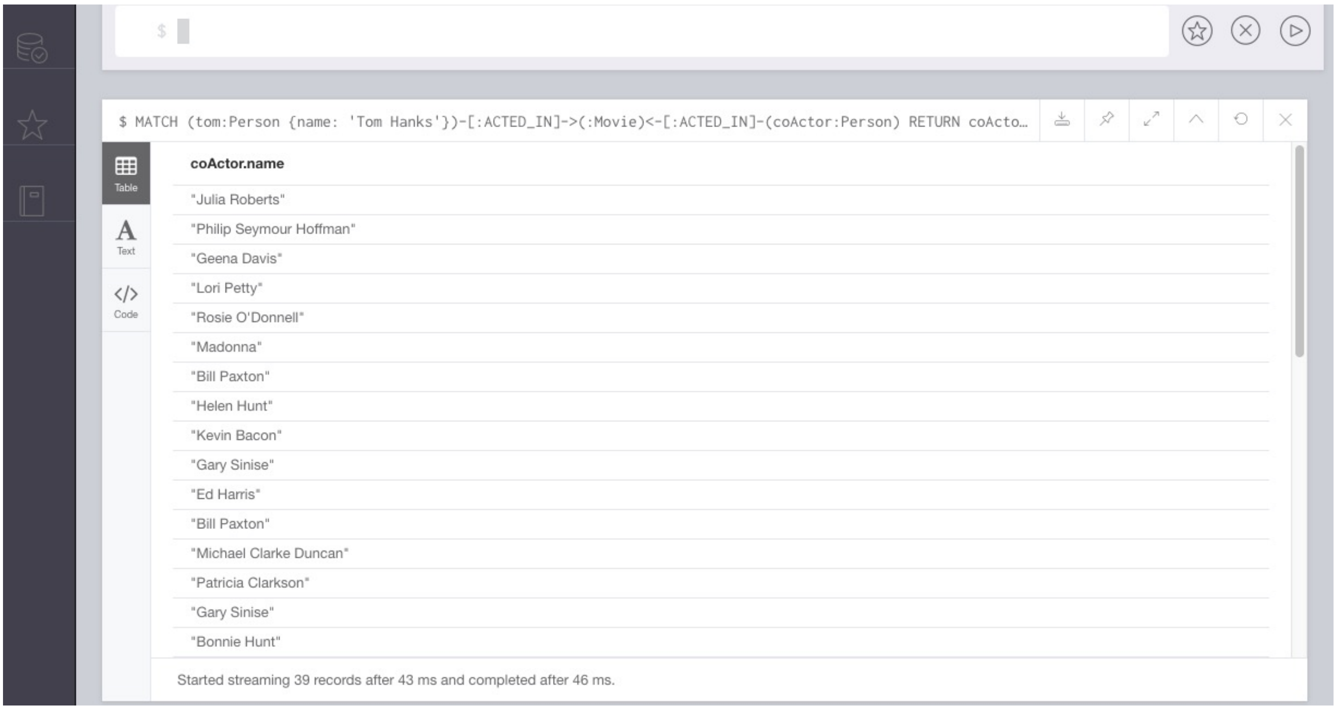


```
$ MATCH (tom:Person {name: 'Tom Hanks'})-[r:ACTED_IN]->(movie:Movie) RETURN tom, r, movie
```

Displaying 13 nodes, 13 relationships.

Of course, Tom has colleagues who acted with him in his movies. A statement to find Tom's co-actors looks like this:

```
MATCH (tom:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(:Movie)<-[:ACTED_IN]-(coActor:Person)
RETURN coActor.name
```



```
$ MATCH (tom:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(:Movie)<-[:ACTED_IN]-(coActor:Person) RETURN coActo…
```

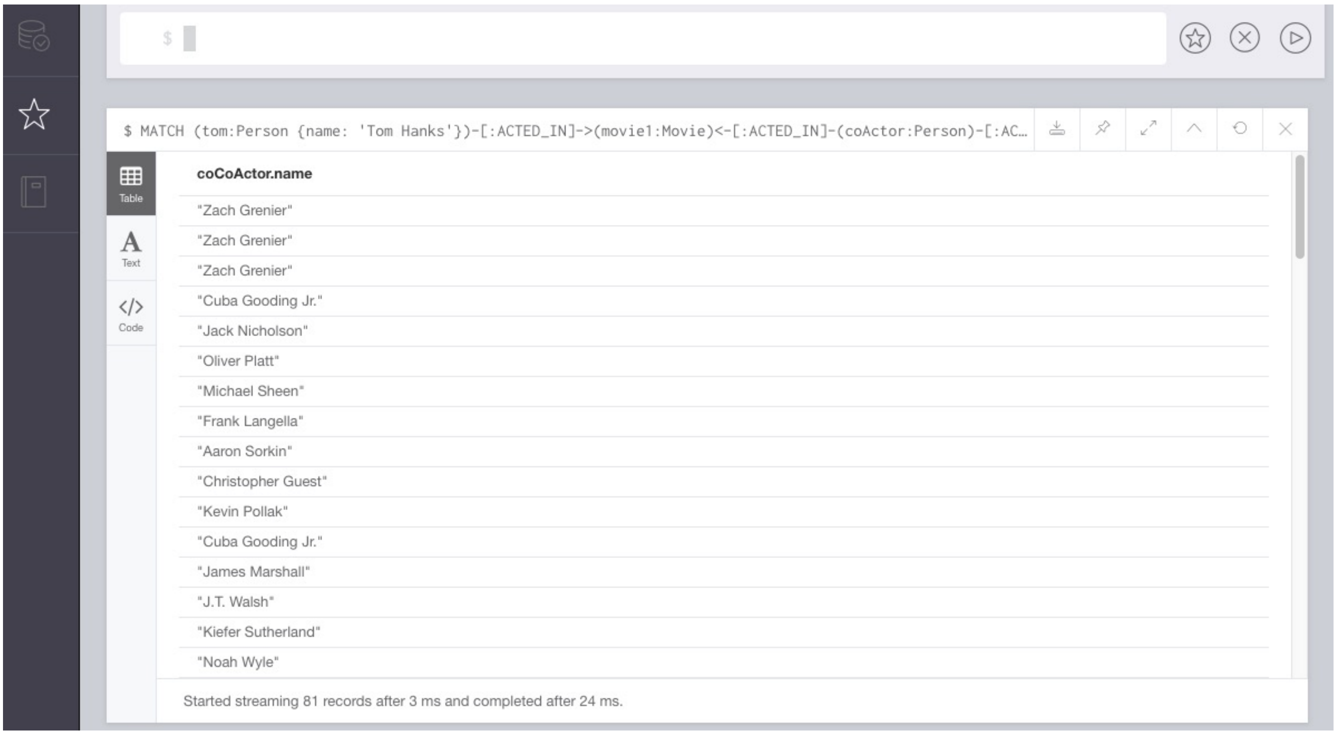
coActor.name
"Julia Roberts"
"Philip Seymour Hoffman"
"Geena Davis"
"Lori Petty"
"Rosie O'Donnell"
"Madonna"
"Bill Paxton"
"Helen Hunt"
"Kevin Bacon"
"Gary Sinise"
"Ed Harris"
"Bill Paxton"
"Michael Clarke Duncan"
"Patricia Clarkson"
"Gary Sinise"
"Bonnie Hunt"

Started streaming 39 records after 43 ms and completed after 46 ms.

Recommendations with collaborative filtering

We can now turn the co-actor query above into a recommendation query by following those relationships another step out to find the "co-co-actors", i.e. the second-degree actors in Tom's network. This will show us all the actors Tom may not have worked with yet, and we can specify a criteria to be sure he hasn't directly acted with that person.

```
MATCH (tom:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(movie1:Movie)<-[:ACTED_IN]-(coActor:Person)-
[:ACTED_IN]->(movie2:Movie)<-[:ACTED_IN]-(cocoActor:Person)
WHERE tom <> cocoActor
AND NOT (tom)-[:ACTED_IN]->(:Movie)<-[:ACTED_IN]-(cocoActor)
RETURN cocoActor.name
```



```
$ MATCH (tom:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(movie1:Movie)<-[:ACTED_IN]-(coActor:Person)-[:AC…
```

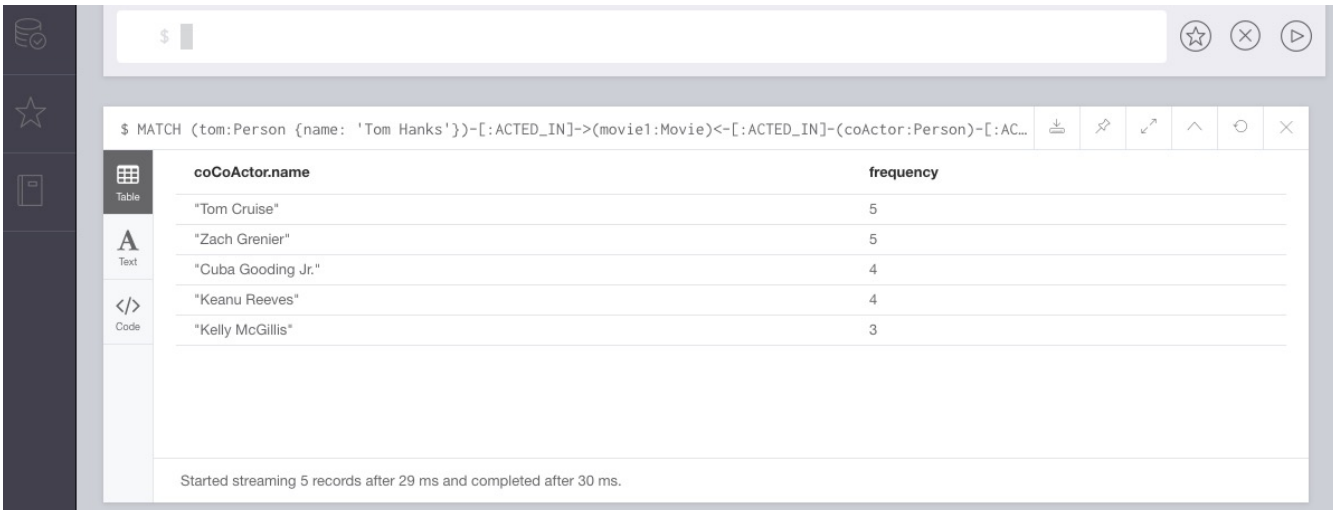
coCoActor.name
"Zach Grenier"
"Zach Grenier"
"Zach Grenier"
"Cuba Gooding Jr."
"Jack Nicholson"
"Oliver Platt"
"Michael Sheen"
"Frank Langella"
"Aaron Sorkin"
"Christopher Guest"
"Kevin Pollak"
"Cuba Gooding Jr."
"James Marshall"
"J.T. Walsh"
"Kiefer Sutherland"
"Noah Wyle"

Started streaming 81 records after 3 ms and completed after 24 ms.

You probably noticed that a few names appear multiple times. This is because there are multiple paths to follow from *Tom Hanks* to these actors.

To see which co-co-actors appear most often in Tom's network, we can take frequency of occurrences into account by counting the number of paths between *Tom Hanks* and each coCoActor and ordering them by highest to lowest value.

```
MATCH (tom:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(movie1:Movie)<-[:ACTED_IN]-(coActor:Person)-
[:ACTED_IN]->(movie2:Movie)<-[:ACTED_IN]-(cocoActor:Person)
WHERE tom <> cocoActor
AND NOT (tom)-[:ACTED_IN]->(:Movie)<-[:ACTED_IN]-(cocoActor)
RETURN cocoActor.name, count(cocoActor) as frequency
ORDER BY frequency DESC
LIMIT 5
```



$ MATCH (tom:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(movie1:Movie)<-[:ACTED_IN]-(coActor:Person)-[:AC…

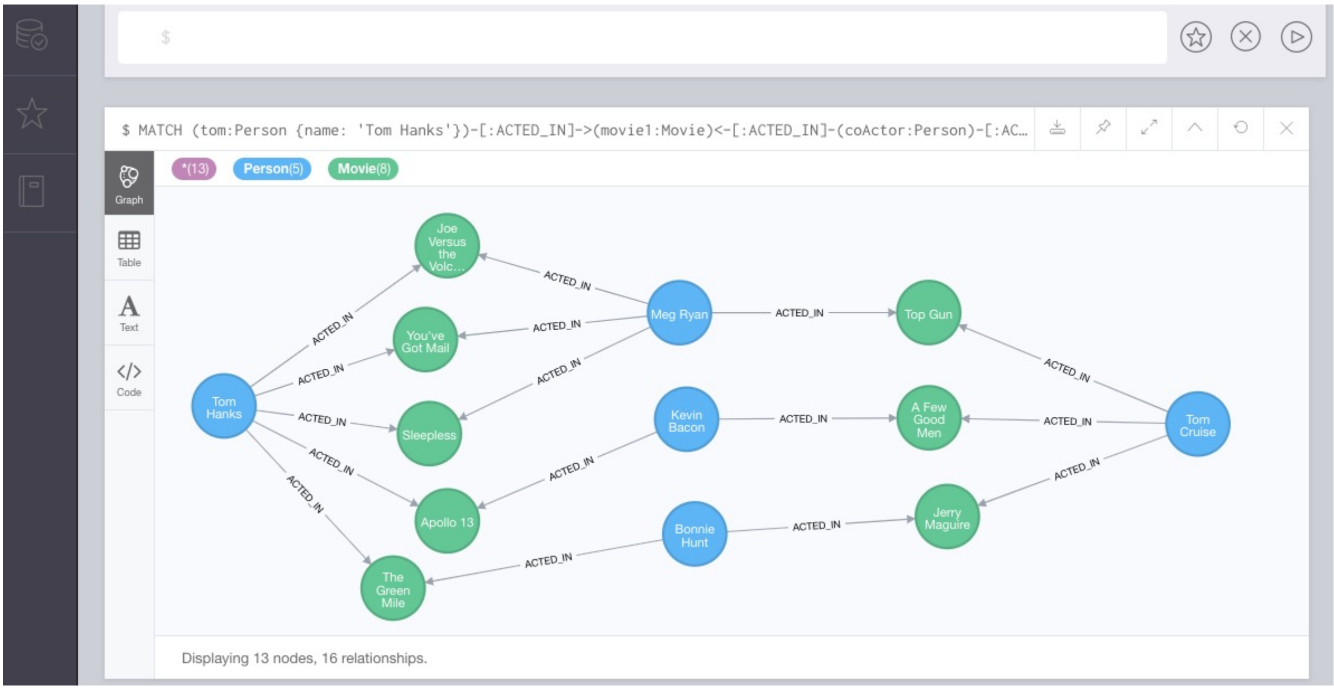
coCoActor.name	frequency
"Tom Cruise"	5
"Zach Grenier"	5
"Cuba Gooding Jr."	4
"Keanu Reeves"	4
"Kelly McGillis"	3

Started streaming 5 records after 29 ms and completed after 30 ms.

One of those "co-co-actors" is *Tom Cruise*. Now let’s see which movies and actors are between the two Toms so we can find out who can introduce them.

Exploring the paths

```
MATCH (tom:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(movie1:Movie)<-[:ACTED_IN]-(coActor:Person)-
[:ACTED_IN]->(movie2:Movie)<-[:ACTED_IN]-(cruise:Person {name: 'Tom Cruise'})
WHERE NOT (tom)-[:ACTED_IN]->(:Movie)<-[:ACTED_IN]-(cruise)
RETURN tom, movie1, coActor, movie2, cruise
```



$ MATCH (tom:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(movie1:Movie)<-[:ACTED_IN]-(coActor:Person)-[:AC…

Displaying 13 nodes, 16 relationships.

As you can see, this returns multiple paths. If you have ever played the six degrees of Kevin Bacon game, this concept of seeing how many hops exist between people is exactly what graphs depict. You will notice that our results even return a path with *Kevin Bacon* himself.

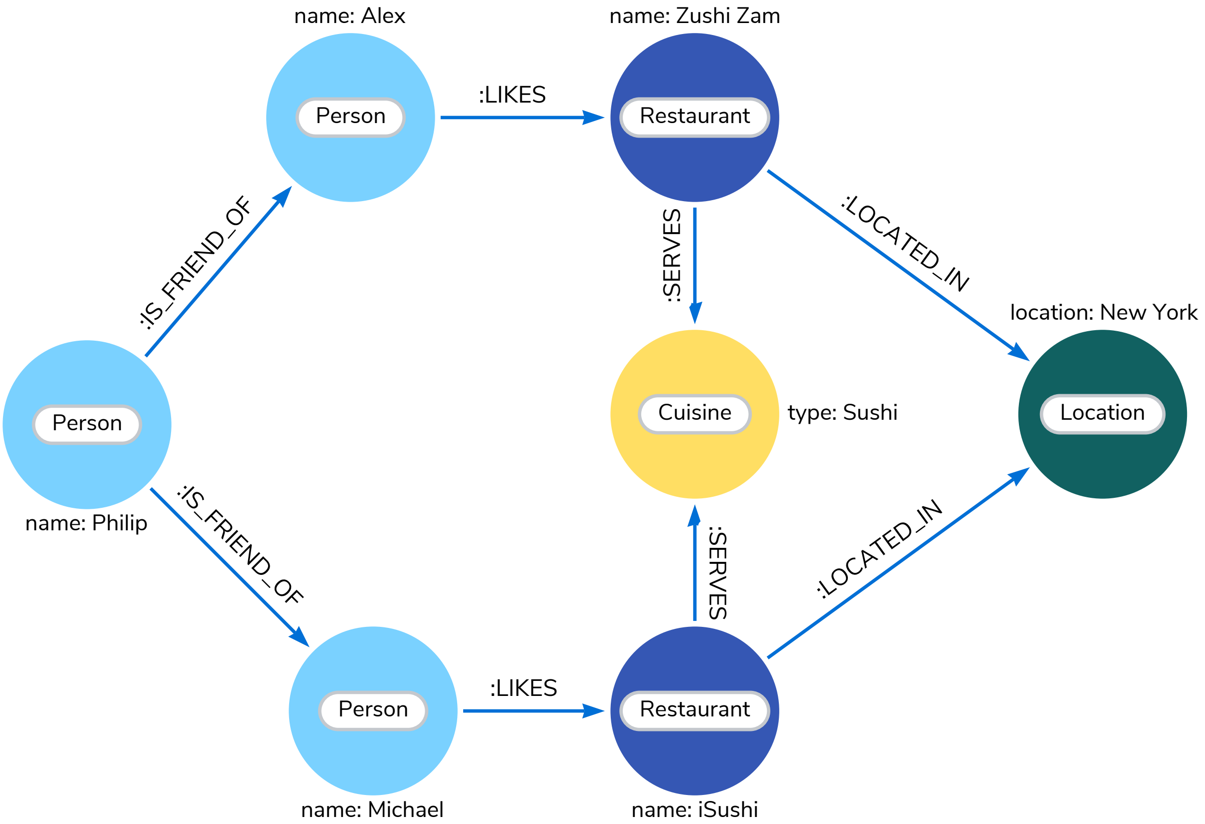
With these two simple Cypher statements, we already created two recommendation algorithms - **who to meet/work with next** and **how to meet them.**

Other recommendations

You could apply the same ideas you learned here to many other uses for recommending products and services, finding restaurants or activities you might like, or connecting with other colleagues who share similar interests of skills. We will mention a few specifically here with resources you can use to find more information.

Restaurant recommendations

We have a graph of a few friends with their favorite restaurants, cuisines, and locations.



A practical question to answer here, formulated as a graph search, is:

```
What Sushi restaurants are in New York that my friends like?
```

How to translate that into the appropriate Cypher statement?

```
MATCH (person:Person {name: 'Philip'})-[:IS_FRIEND_OF]->(friend)-[:LIKES]->(restaurant:Restaurant)-
[:LOCATED_IN]->(loc:Location {location: 'New York'}),
      (restaurant)-[:SERVES]->(type:Cuisine {type: 'Sushi'})
RETURN restaurant.name, count(*) AS occurrence
ORDER BY occurrence DESC
LIMIT 5
```

Other factors that can be easily integrated in this query are favorites, allergies, ratings, and distance from current position.

More recommendation solutions

- Recipe and Food Recommendations
- Sandbox: Recommend Movies by Reviews
- GraphGist: Beer and Breweries Recommendations
- GraphGist: Northwind Product Recommendations

Resources

- Neo4j Videos: Building Recommendation Engines
- Recommendation Use Cases
- Online Training: Learn Cypher with Intro to Neo4j
- Michal Bachman Slides: Recommendation Engines with Neo4j
- GraphGists: Recommendation Engine Examples

Tutorial: Import data from a relational database into Neo4j

Introduction

This tutorial shows the process for exporting data from a relational database (PostgreSQL) and importing into a graph database (Neo4j). You will learn how to take data from the relational system and to the graph by translating the schema and using import tools.

Alternatively, you can:

- Create AuraDB cloud instance.
- Start a blank Neo4j Sandbox.
- Download and install Neo4j Desktop.

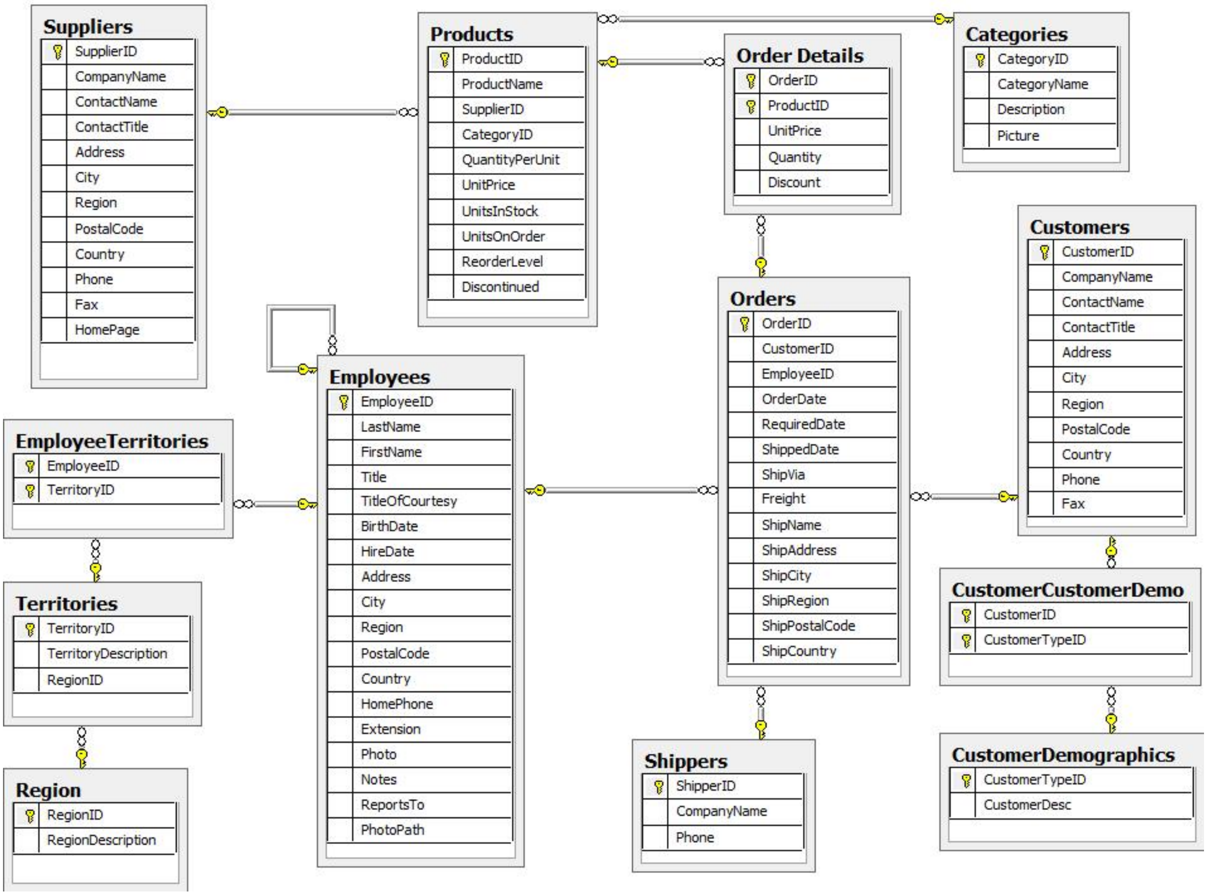
This guide uses a specific dataset, but its principles can be applied and reused with any data domain.

You should have a basic understanding of the property graph model and know how to model data as a graph.

About the data domain

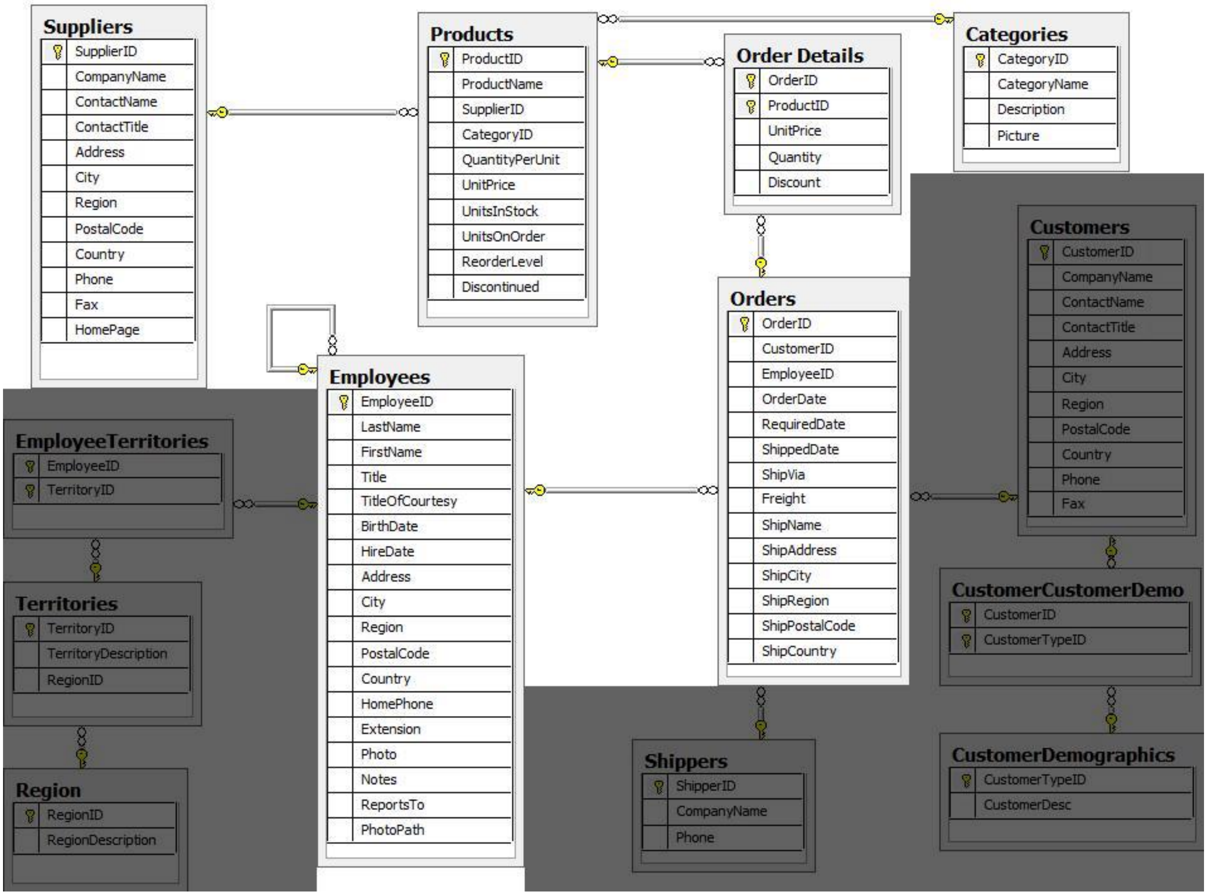
In this guide, we will be using the NorthWind dataset, an often-used SQL dataset. This data depicts a product sale system - storing and tracking customers, products, customer orders, warehouse stock, shipping, suppliers, and even employees and their sales territories. Although the NorthWind dataset is often used to demonstrate SQL and relational databases, the data also can be structured as a graph.

An entity-relationship diagram (ERD) of the Northwind dataset is shown below.



First, this is a rather large and detailed model. We can scale this down a bit for our example and choose the entities that are most critical for our graph - in other words, those that might benefit most from seeing the connections. For our use case, we really want to optimize the relationships with orders - what products were involved (with the categories and suppliers for those products), which employees worked on them and those employees' managers.

Using these business requirements, we can narrow our model down to these essential entities.



Developing a graph model

The first thing you will need to do to get data from a relational database into a graph is to translate the relational data model to a graph data model. Determining how you want to structure tables and rows as nodes and relationships may vary depending on what is most important to your business needs.

For more information on adapting your graph model to different scenarios, check out our modeling designs guide.

When deriving a graph model from a relational model, you should keep a couple of general guidelines in mind.

1. A *row* is a *node*.
2. A *table name* is a *label name*.
3. A *join or foreign key* is a *relationship*.

With these principles in mind, we can map our relational model to a graph with the following steps:

Rows to nodes, table names to labels

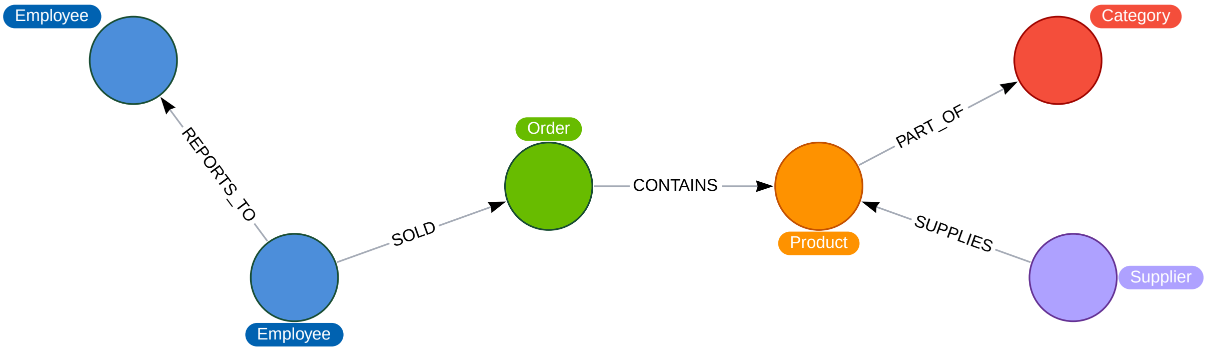
1. Each row on our `Orders` table becomes a node in our graph with `Order` as the label.
2. Each row on our `Products` table becomes a node with `Product` as the label.
3. Each row on our `Suppliers` table becomes a node with `Supplier` as the label.

4. Each row on our `Categories` table becomes a node with `Category` as the label.
5. Each row on our `Employees` table becomes a node with `Employee` as the label.

Joins to relationships

1. Join between `Suppliers` and `Products` becomes a relationship named `SUPPLIES` (where supplier supplies product).
2. Join between `Products` and `Categories` becomes a relationship named `PART_OF` (where product is part of a category).
3. Join between `Employees` and `Orders` becomes a relationship named `SOLD` (where employee sold an order).
4. Join between `Employees` and itself (unary relationship) becomes a relationship named `REPORTS_TO` (where employees have a manager).
5. Join with join table (`Order Details`) between `Orders` and `Products` becomes a relationship named `CONTAINS` with properties of `unitPrice`, `quantity`, and `discount` (where order contains a product).

If we draw our translation out on the whiteboard, we have this graph data model.



Now, we can, of course, decide that we want to include the rest of the entities from our relational model, but for now, we will keep to this smaller graph model.

How does the graph model differ from the relational model?

- There are no nulls. Non-existing value entries (properties) are just not present.
- It describes the relationships in more detail. For example, we know that an employee SOLD an order rather than having a foreign key relationship between the Orders and Employees tables. We could also choose to add more metadata about that relationship, should we wish.
- Either model can be more normalized. For example, addresses have been denormalized in several of the tables, but could have been in a separate table. In a future version of our graph model, we might also choose to separate addresses from the `Order` (or `Supplier` or `Employee`) entities and create separate `Address` nodes.

Exporting relational tables to CSV

Thankfully, this step has already been done for you with the Northwind data you will use later on in this guide.

However, if you are working with another data domain, you need to take the data from the relational tables and put it in another format for loading to the graph. A common format that many systems can handle a flat file of comma-separated values (CSV).

Here is an example script we already ran to export the northwind data into CSV files for you.

export_csv.sql

```
COPY (SELECT * FROM customers) TO '/tmp/customers.csv' WITH CSV header;
COPY (SELECT * FROM suppliers) TO '/tmp/suppliers.csv' WITH CSV header;
COPY (SELECT * FROM products)  TO '/tmp/products.csv' WITH CSV header;
COPY (SELECT * FROM employees) TO '/tmp/employees.csv' WITH CSV header;
COPY (SELECT * FROM categories) TO '/tmp/categories.csv' WITH CSV header;

COPY (SELECT * FROM orders
      LEFT OUTER JOIN order_details ON order_details.OrderID = orders.OrderID) TO '/tmp/orders.csv' WITH
CSV header;
```

If you want to create the CSV files yourself using your own northwind RDBMS, you can run this script against your RDBMS with the command `psql -d northwind < export_csv.sql`.

Note: You need not run this script unless you want to execute it against your own northwind RDBMS.

Importing the data using Cypher

You can use Cypher's `LOAD CSV` command to transform the contents of the CSV file into a graph structure.

When you use `LOAD CSV` to create nodes and relationships in the database, you have two options for where the CSV files reside:

- In the **import** folder for the Neo4j instance that you can manage.
- From a publicly-available location such as an S3 bucket or a github location. You must use this option if you are using Neo4j AuraDB or Neo4j Sandbox.

If you want to use the CSV files for your Neo4j instance that you manage, you can copy the CSV files from Northwind zip from github and place them in the **import** folder for your Neo4j DBMS.

We have already placed these CSV files in Gihub for your access to them.

You use use Cypher's `LOAD CSV` statement to read each file and add Cypher clauses after it to take the row/column data and transform it to the graph.

Next you will run Cypher code to:

1. Load the nodes from the CSV files.
2. Create the indexes and constraint for the data in the graph.
3. Create the relationships between the nodes.

Creating **Order** nodes

Execute this Cypher block to create the Order nodes in the database:

```
// Create orders
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f41
9b5a0cb8ac4f/orders.csv' AS row
MERGE (order:Order {orderID: row.OrderID})
  ON CREATE SET order.shipName = row.ShipName;
```

If you have placed the CSV files in to the **import** folder, you should use this code syntax to load the CSV files from a local directory:

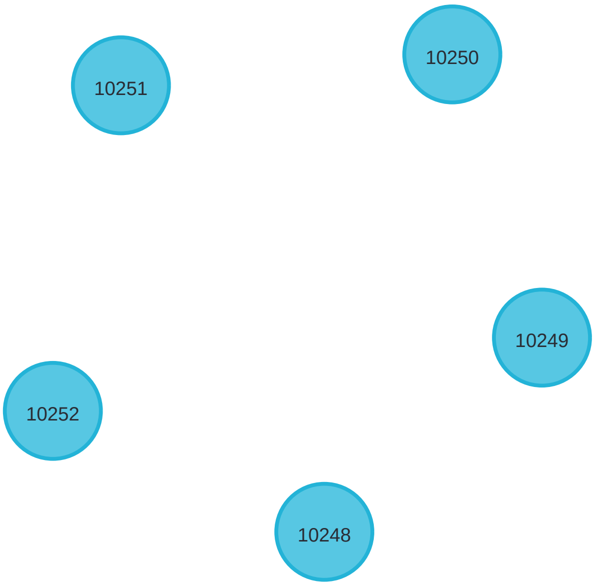
```
// Create orders
LOAD CSV WITH HEADERS FROM 'file:///orders.csv' AS row
MERGE (order:Order {orderID: row.OrderID})
  ON CREATE SET order.shipName = row.ShipName;
```

This code creates 830 `Order` nodes in the database.

You can view some of the nodes in the database by executing this code:

```
MATCH (o:Order) return o LIMIT 5;
```

The graph view is:



The table view contains these values for the node properties:

o
{"shipName":Vins et alcools Chevalier,"orderID":10248}
{"shipName":Toms Spezialitäten,"orderID":10249}
{"shipName":Hanari Carnes,"orderID":10250}
{"shipName":Victuailles en stock,"orderID":10251}

o
{"shipName":Suprêmes délices,"orderID":10252}

You might notice that you have not imported all of the field columns in the CSV file. With your statements, you can choose which properties are needed on a node, which can be left out, and which might need imported to another node type or relationship. You might also notice that you used the MERGE keyword, instead of CREATE. Though we feel pretty confident there are no duplicates in our CSV files, we can use MERGE as good practice for ensuring unique entities in our database.

Creating **Product** nodes

Execute this code to create the Product nodes in the database:

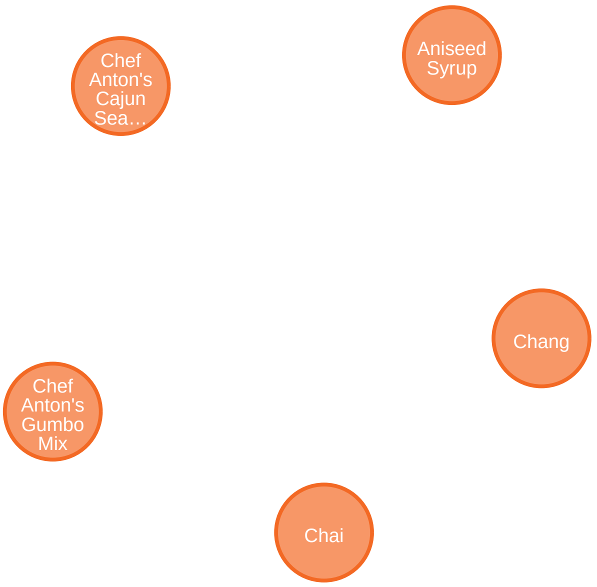
```
// Create products
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f41
9b5a0cb8ac4f/products.csv' AS row
MERGE (product:Product {productID: row.ProductID})
  ON CREATE SET product.productName = row.ProductName, product.unitPrice = toFloat(row.UnitPrice);
```

This code creates 77 Product nodes in the database.

You can view some of these nodes in the database by executing this code:

```
MATCH (p:Product) return p LIMIT 5;
```

The graph view is:



The table view contains these values for the node properties:

p
{"unitPrice":18.0,"productID":1,"productName":Chai}

p
{"unitPrice":19.0,"productID":2,"productName":Chang}
{"unitPrice":10.0,"productID":3,"productName":Aniseed Syrup}
{"unitPrice":22.0,"productID":4,"productName":Chef Anton’s Cajun Seasoning}
{"unitPrice":21.35,"productID":5,"productName":Chef Anton’s Gumbo Mix}

Creating **Supplier** nodes

Execute this code to create the Supplier nodes in the database:

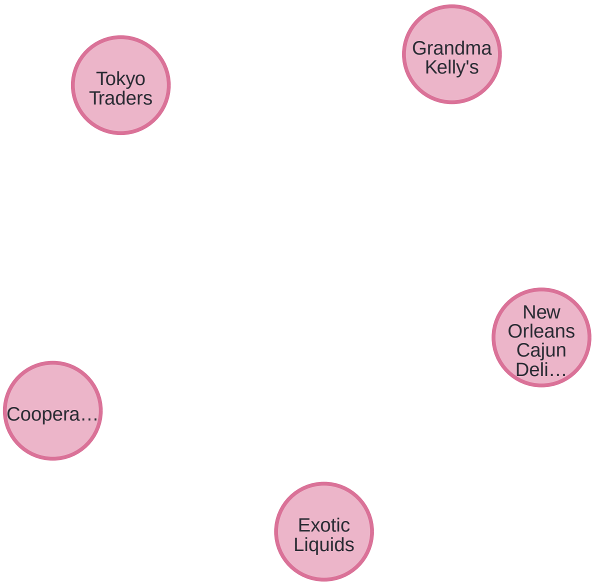
```
// Create suppliers
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f41
9b5a0cb8ac4f/suppliers.csv' AS row
MERGE (supplier:Supplier {supplierID: row.SupplierID})
  ON CREATE SET supplier.companyName = row.CompanyName;
```

This code creates 29 `Supplier` nodes in the database.

You can view some of these nodes in the database by executing this code:

```
MATCH (s:Supplier) return s LIMIT 5;
```

The graph view is:



The table view contains these values for the node properties:

s
{"supplierID":1,"companyName":Exotic Liquids}
{"supplierID":2,"companyName":New Orleans Cajun Delights}

s
{"supplierID":3,"companyName":Grandma Kelly’s Homestead}
{"supplierID":4,"companyName":Tokyo Traders}
{"supplierID":5,"companyName":Cooperativa de Quesos 'Las Cabras'}

Creating **Employee** nodes

Execute this code to create the Supplier nodes in the database:

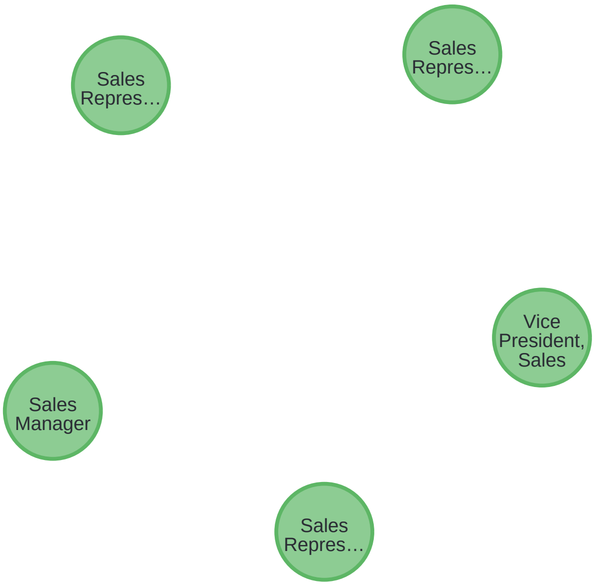
```
// Create employees
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f41
9b5a0cb8ac4f/employees.csv' AS row
MERGE (e:Employee {employeeID:row.EmployeeID})
  ON CREATE SET e.firstName = row.FirstName, e.lastName = row.LastName, e.title = row.Title;
```

This code creates 9 `Employee` nodes in the database.

You can view some of these nodes in the database by executing this code:

```
MATCH (e:Employee) return e LIMIT 5;
```

The graph view is:



The table view contains these values for the node properties:

e
{"lastName":Davolio,"firstName":Nancy,"employeeID":1,"title":Sales Representative}
{"lastName":Fuller,"firstName":Andrew,"employeeID":2,"title":Vice President, Sales}
{"lastName":Leverling,"firstName":Janet,"employeeID":3,"title":Sales Representative}

e
{"lastName":Peacock,"firstName":Margaret,"employeeID":4,"title":Sales Representative}
{"lastName":Buchanan,"firstName":Steven,"employeeID":5,"title":Sales Manager}

Creating **Category** nodes

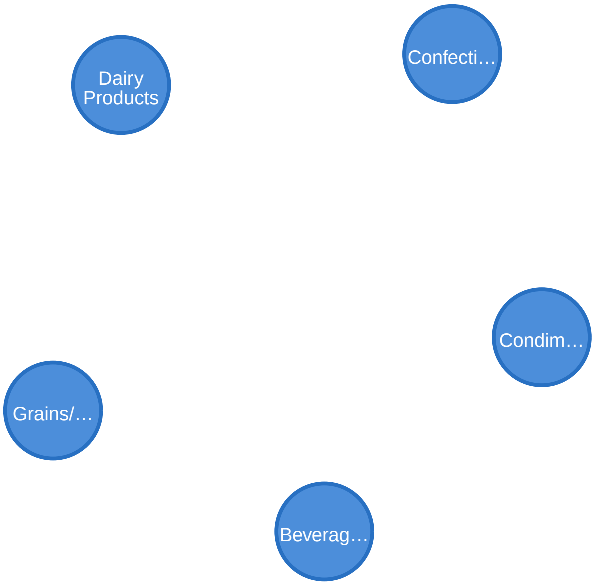
```
// Create categories
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f41
9b5a0cb8ac4f/categories.csv' AS row
MERGE (c:Category {categoryID: row.CategoryID})
  ON CREATE SET c.categoryName = row.CategoryName, c.description = row.Description;
```

This code creates 8 `Category` nodes in the database.

You can view some of these nodes in the database by executing this code:

```
MATCH (c:Category) return c LIMIT 5;
```

The graph view is:



The table view contains these values for the node properties:

c
{"description":Soft drinks, coffees, teas, beers, and ales,"categoryName":Beverages,"categoryID":1}
{"description":Sweet and savory sauces, relishes, spreads, and seasonings,"categoryName":Condiments,"categoryID":2}
{"description":Desserts, candies, and sweet breads,"categoryName":Confections,"categoryID":3}
{"description":Cheeses,"categoryName":Dairy Products,"categoryID":4}
{"description":Breads, crackers, pasta, and cereal,"categoryName":Grains/Cereals,"categoryID":5}

For very large commercial or enterprise datasets, you may find out-of-memory errors, especially on smaller machines. To avoid these situations, you can use `CALL IN {…} TRANSACTIONS` subquery to commit data in batches. Don't forget to prepend this query with `:auto` in Neo4j Browser. This practice is not standard recommendation for smaller datasets, but is only recommended when memory issues are threatened. More information on this subquery can be found in the Cypher manual → Subqueries in transactions.

Creating the indexes and constraints for the data in the graph

After the nodes are created, you need to create the relationships between them. Importing the relationships will mean looking up the nodes you just created and adding a relationship between those existing entities. To ensure the lookup of nodes is optimized, you will create indexes for any node properties used in the lookups (often the ID or another unique value).

We also want to create a constraint (also creates an index with it) that will disallow orders with the same id from getting created, preventing duplicates. Finally, as the indexes are created after the nodes are inserted, their population happens asynchronously, so we call `db.awaitIndexes()` to block until they are populated.

Execute this code block:

```
CREATE INDEX product_id FOR (p:Product) ON (p.productID);
CREATE INDEX product_name FOR (p:Product) ON (p.productName);
CREATE INDEX supplier_id FOR (s:Supplier) ON (s.supplierID);
CREATE INDEX employee_id FOR (e:Employee) ON (e.employeeID);
CREATE INDEX category_id FOR (c:Category) ON (c.categoryID);
CREATE CONSTRAINT order_id FOR (o:Order) REQUIRE o.orderID IS UNIQUE;
CALL db.awaitIndexes();
```

After you execute this code, you can execute this code to view the indexes (and constraint) in the database:

```
CALL db.indexes();
```

You should see these indexes (and constraint) in the database:

id	name	state	populationPercent	uniqueness	type	entityType	labelsOrTypes	properties	provider
5	category_id	ONLINE	100.0	NONUNIQUE	BTREE	NODE	[Category]	[categoryID]	native-btree-1.0
4	employee_id	ONLINE	100.0	NONUNIQUE	BTREE	NODE	[Employee]	[employeeID]	native-btree-1.0
6	order_id	ONLINE	100.0	UNIQUE	BTREE	NODE	[Order]	[orderID]	native-btree-1.0
1	product_id	ONLINE	100.0	NONUNIQUE	BTREE	NODE	[Product]	[productID]	native-btree-1.0

id	name	state	populationPercent	uniqueness	type	entityType	labelsOrTypes	properties	provider
2	product_name	ONLINE	100.0	NONUNIQUE	BTREE	NODE	[Product]	[productName]	native-btree-1.0
3	supplier_id	ONLINE	100.0	NONUNIQUE	BTREE	NODE	[Supplier]	[supplierID]	native-btree-1.0

Creating the relationships between the nodes

Next you have to create relationships:

1. Between Orders and Employees.
2. Between Products and Suppliers and between Products and Categories.
3. Between Employees.

Creating relationships between Orders and Employees

With the initial nodes and indexes in place, you can now create the relationships for orders to products and orders to employees.

Execute this code block:

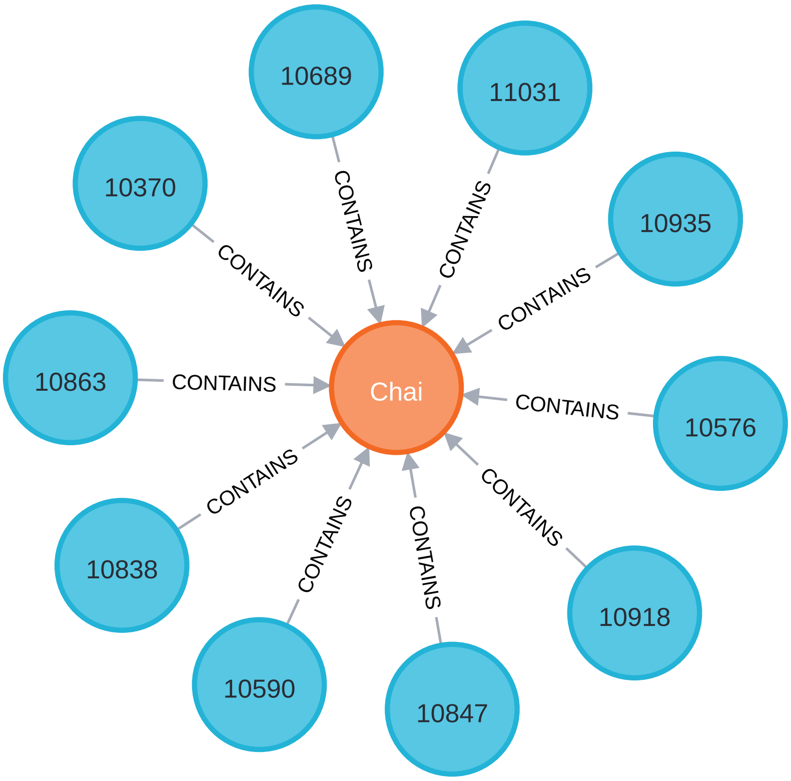
```
// Create relationships between orders and products
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f41
9b5a0cb8ac4f/orders.csv' AS row
MATCH (order:Order {orderID: row.OrderID})
MATCH (product:Product {productID: row.ProductID})
MERGE (order)-[op:CONTAINS]->(product)
  ON CREATE SET op.unitPrice = toFloat(row.UnitPrice), op.quantity = toFloat(row.Quantity);
```

This code creates 2155 relationships in the graph.

You can view some of them by executing this code:

```
MATCH (o:Order)-[]-(p:Product)
RETURN o,p LIMIT 10;
```

Your graph view should look something like this:



Then, execute this code block:

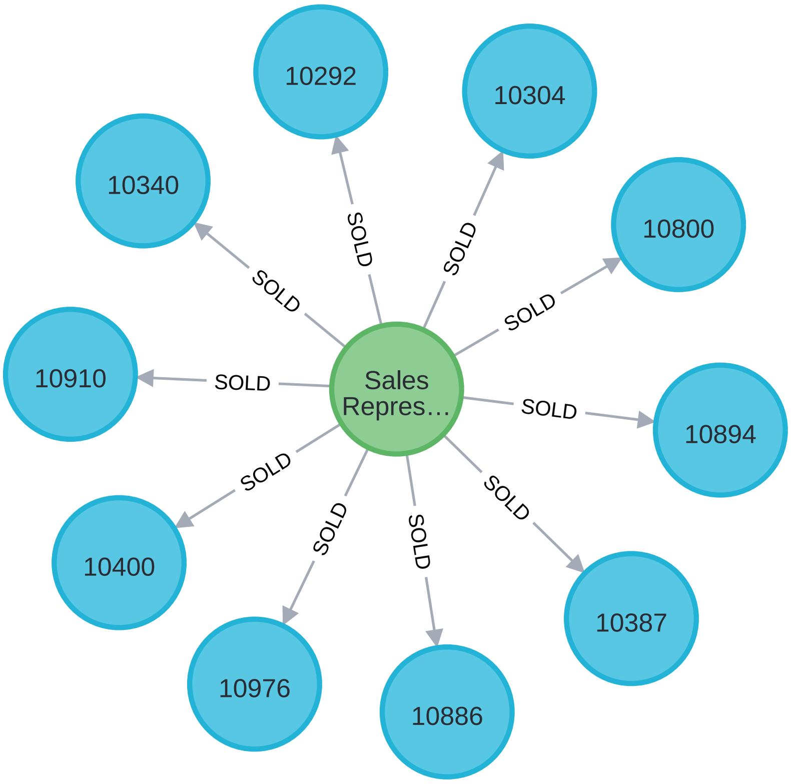
```
// Create relationships between orders and employees
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f41
9b5a0cb8ac4f/orders.csv' AS row
MATCH (order:Order {orderID: row.OrderID})
MATCH (employee:Employee {employeeID: row.EmployeeID})
MERGE (employee)-[:SOLD]->(order);
```

This code creates 830 relationships in the graph.

You can view some of them by executing this code:

```
MATCH (o:Order)-[]-(e:Employee)
RETURN o,e LIMIT 10;
```

Your graph view should look something like this:



Creating relationships between Products and Suppliers and between Products and Categories

Next, create relationships between Products, Suppliers, and Categories:

Execute this code block:

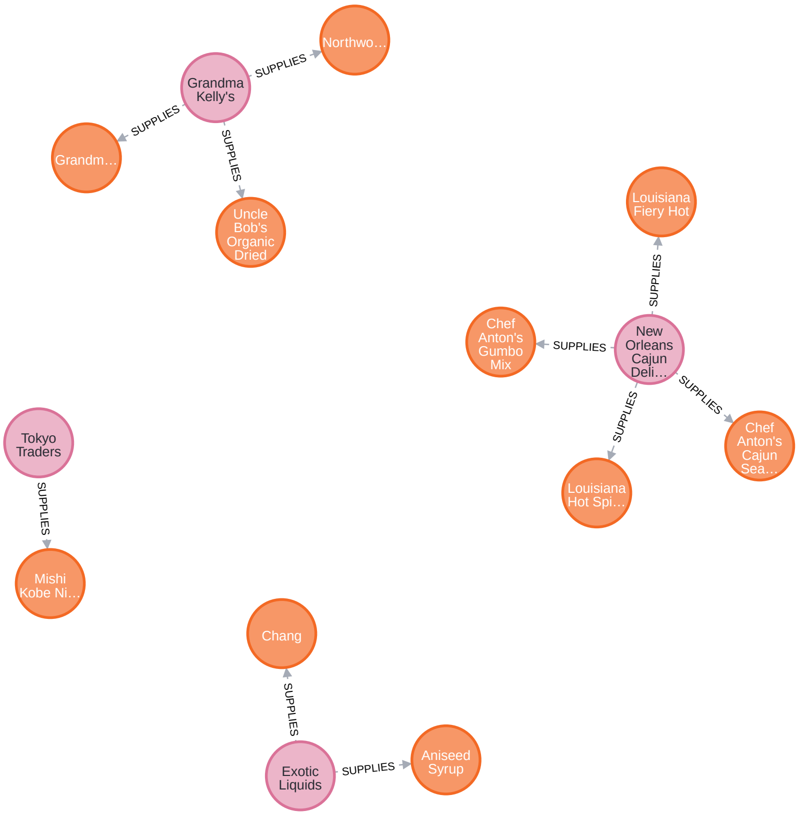
```
// Create relationships between products and suppliers
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f41
9b5a0cb8ac4f/products.csv
' AS row
MATCH (product:Product {productID: row.ProductID})
MATCH (supplier:Supplier {supplierID: row.SupplierID})
MERGE (supplier)-[:SUPPLIES]->(product);
```

This code creates 77 relationships in the graph.

You can view some of them by executing this code:

```
MATCH (s:Supplier)-[]-(p:Product)
RETURN s,p LIMIT 10;
```

Your graph view should look something like this:



Then, execute this code block:

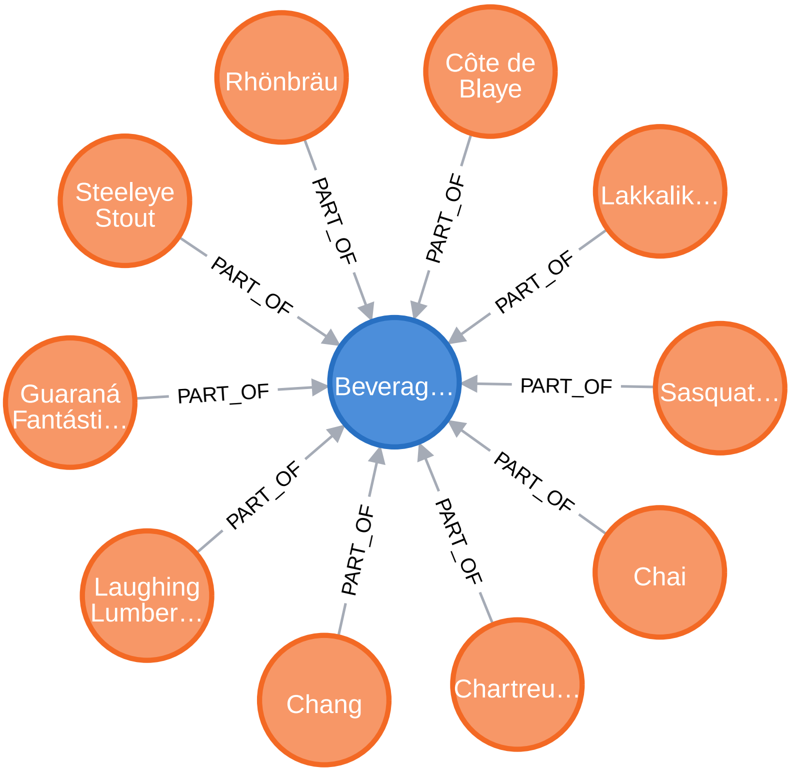
```
// Create relationships between products and categories
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f41
9b5a0cb8ac4f/products.csv
' AS row
MATCH (product:Product {productID: row.ProductID})
MATCH (category:Category {categoryID: row.CategoryID})
MERGE (product)-[:PART_OF]->(category);
```

This code creates 77 relationships in the graph.

You can view some of them by executing this code:

```
MATCH (c:Category)-[]-(p:Product)
RETURN c,p LIMIT 10;
```

Your graph view should look something like this:



Creating relationships between Employees

Lastly, you will create the 'REPORTS_TO' relationship between Employees to represent the reporting structure:

Execute this code block:

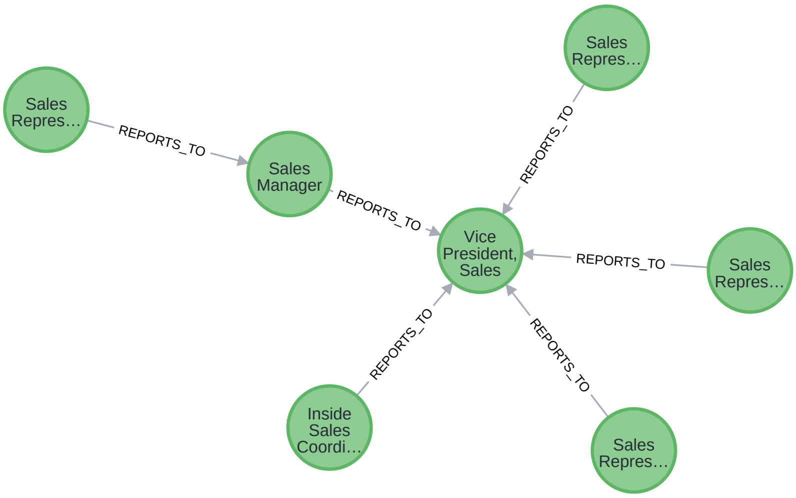
```
// Create relationships between employees (reporting hierarchy)
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f41
9b5a0cb8ac4f/employees.csv' AS row
MATCH (employee:Employee {employeeID: row.EmployeeID})
MATCH (manager:Employee {employeeID: row.ReportsTo})
MERGE (employee)-[:REPORTS_TO]->(manager);
```

This code creates 8 relationships in the graph.

You can view some of them by executing this code:

```
MATCH (e1:Employee)-[]-(e2:Employee)
RETURN e1,e2 LIMIT 10;
```

Your graph view should look something like this:



Next, you will query the resulting graph to find out what it can tell us about our newly-imported data.

Querying the graph

We might start with a couple of general queries to verify that our data matches the model we designed earlier in the guide. Here are some example queries.

Execute this code block:

```
//find a sample of employees who sold orders with their ordered products
MATCH (e:Employee)-[rel:SOLD]->(o:Order)-[rel2:CONTAINS]->(p:Product)
RETURN e, rel, o, rel2, p LIMIT 25;
```

Execute this code block:

```
//find the supplier and category for a specific product
MATCH (s:Supplier)-[r1:SUPPLIES]->(p:Product {productName: 'Chocolade'})-[r2:PART_OF]->(c:Category)
RETURN s, r1, p, r2, c;
```

Once you are comfortable that the data aligns with our data model and everything looks correct, you can start querying to gather information and insights for business decisions.

Which Employee had the highest cross-selling count of 'Chocolade' and another product?

Execute this code block:

```
MATCH (choc:Product {productName:'Chocolade'})<-[:CONTAINS]-(:Order)<-[:SOLD]-(employee),
      (employee)-[:SOLD]->(o2)-[:CONTAINS]->(other:Product)
RETURN employee.employeeID as employee, other.productName as otherProduct, count(distinct o2) as count
ORDER BY count DESC
LIMIT 5;
```

Looks like employee No. 4 was busy, though employee No. 1 also did well! Your results should look something like this:

employee	otherProduct	count
4	Gnocchi di nonna Alice	14
4	Pâté chinois	12
1	Flotemysost	12
3	Gumbär Gummibärchen	12
1	Pavlova	11

How are Employees organized? Who reports to whom?

Execute this code block:

```
MATCH (e:Employee)<-[:REPORTS_TO]-(sub)
RETURN e.employeeID AS manager, sub.employeeID AS employee;
```

Your results should look something like this:

manager	employee
2	3
2	4
2	5
2	1
2	8
5	9
5	7
5	6

Notice that employee No. 5 has people reporting to them but also reports to employee No. 2.

Next, let’s investigate that a bit more.

Which Employees report to each other indirectly?

Execute this code block:

```
MATCH path = (e:Employee)<-[:REPORTS_TO*]-(sub)
WITH e, sub, [person in NODES(path) | person.employeeID][1..-1] AS path
RETURN e.employeeID AS manager, path as middleManager, sub.employeeID AS employee
ORDER BY size(path);
```

Your results should look something like this:

manager	middleManager	employee
2	[]	3
2	[]	4
2	[]	5
2	[]	1
2	[]	8
5	[]	9
5	[]	7
5	[]	6
2	[5]	9
2	[5]	7
2	[5]	6

How many orders were made by each part of the hierarchy?

Execute this code block:

```
MATCH (e:Employee)
OPTIONAL MATCH (e)<-[:REPORTS_TO*0..]-(sub)-[:SOLD]->(order)
RETURN e.employeeID as employee, [x IN COLLECT(DISTINCT sub.employeeID) WHERE x <> e.employeeID] AS
reportsTo, COUNT(distinct order) AS totalOrders
ORDER BY totalOrders DESC;
```

Your results should look something like this:

employee	reportsTo	totalOrders
2	[8,1,5,6,7,9,4,3]	830
5	[6,7,9]	224
4	[]	156
3	[]	127
1	[]	123
8	[]	104
7	[]	72
6	[]	67
9	[]	43

What’s next?

If you followed along with each step through this guide, then you might want to explore the data set with more queries and try to answer additional questions you came up with for the data. You may also want to apply these same principles to your own or another data set for analysis.

If you used this as a process flow to apply to a different data set or you would like to do that next, feel free to start at the top and work through this guide again with another domain. The steps and processes still apply (though, of course, the data model, queries, and business questions will need adjusted).

If you have data that needs additional cleansing and manipulation than what is covered in this guide, the APOC library may be able to help. It contains hundreds of procedures and functions for handling large amounts of data, translating values, cleaning messy data sources, and more!

If you are interested in doing a one-time initial dump of relational data to Neo4j, then the Neo4j ETL Tool might be what you are looking for. The application is designed with a point-and-click user interface with the goal of fast, simple relational-to-graph loads that help new and existing users gain faster value from seeing their data as a graph without Cypher, import procedures, or other code.

Resources

- Northwind SQL, CSV and Cypher data files, also as zip file
- LOAD CSV: Cypher's command for importing CSV files
- APOC library: Neo4j's utility library
- Neo4j ETL Tool: Loading relational data without code
- Importing Data with Neo4j
- Graph Data Modeling

How-To: Import CSV data with Neo4j Desktop

Neo4j Desktop import

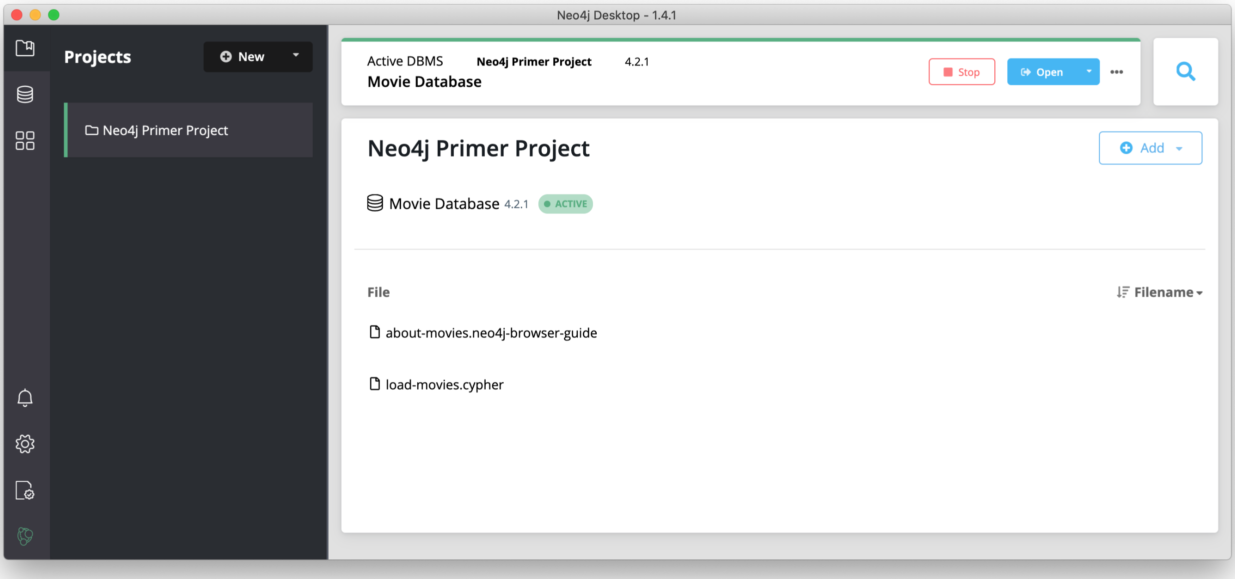
Neo4j Desktop provides a user-friendly interface for starting and creating Neo4j instances, adding or removing plugins, changing configurations, and other functionality. It also includes some shortcuts and easy access for importing files (such as CSVs) into Neo4j.

In this guide, you will work with a zipped folder containing three CSV files and import the data to a graph with Neo4j Desktop. The CSV files contain data for products, orders, and order line items. You will look at the data in the files later in this guide.

1. Creating and starting the Neo4j instance

If you already know how to create a project, create a Neo4j instance (DBMS), and start the DBMS, you can skip to Step 2.

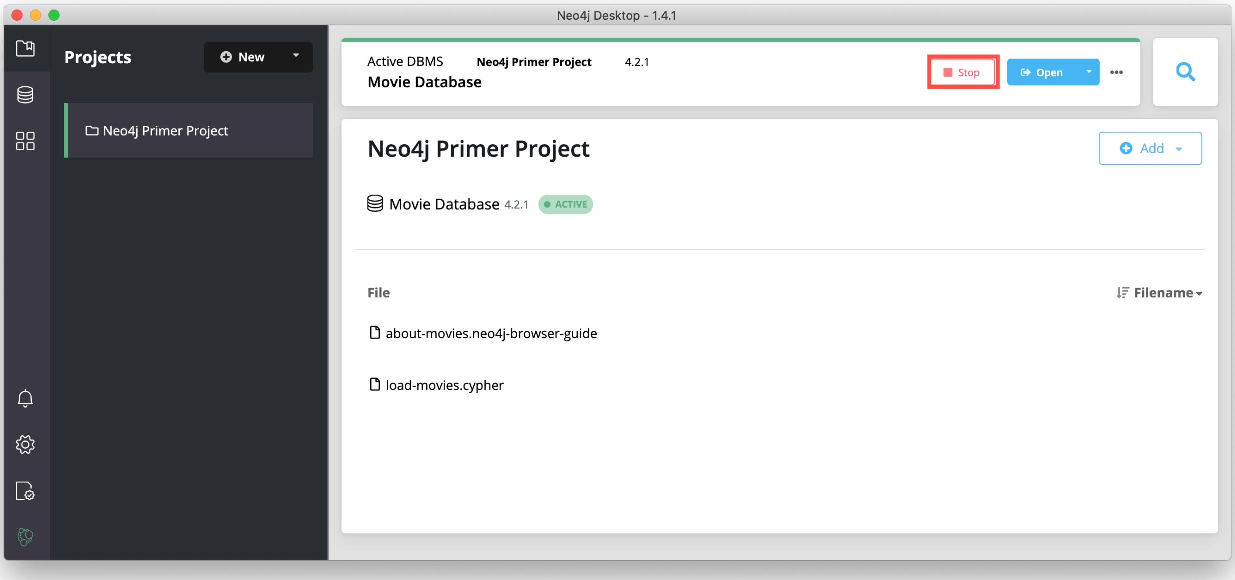
When you open Neo4j Desktop for the first time you will see a Neo4j Primer Project with the Movie Database already started as shown here:



Neo4j Desktop - 1.4.1

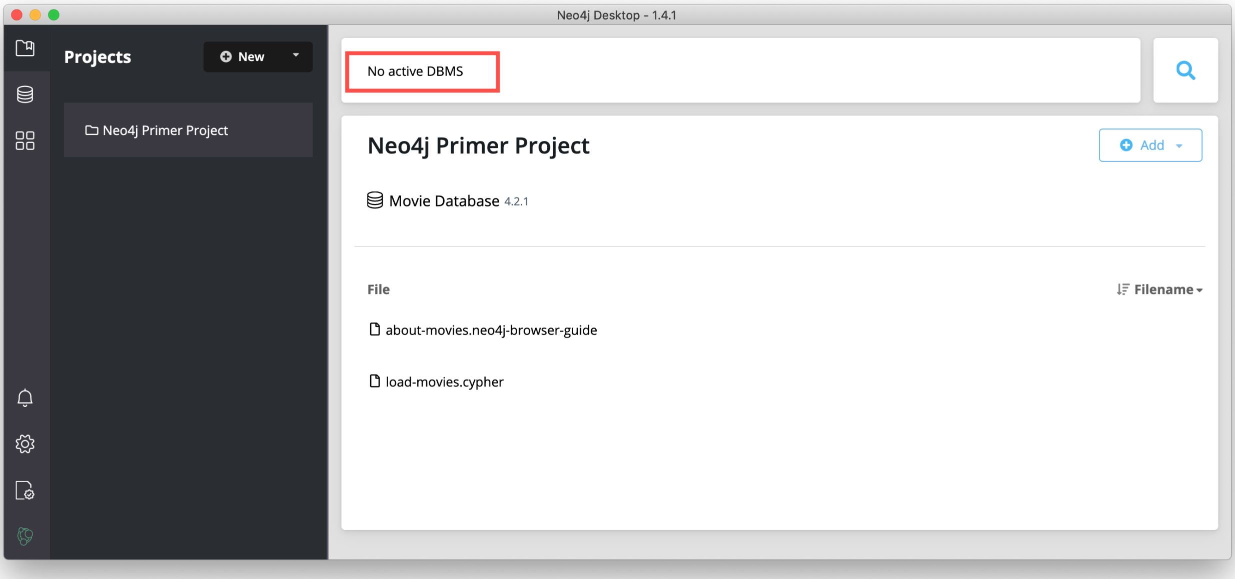
You can use this already-started DBMS if you want to begin learning about using Neo4j and Cypher. For your development, you typically create one or more projects where a project can contain one or more DBMSs.

Next, you will create a new project in Neo4j Desktop. You can only have a single DBMS running, so you must first stop the DBMS that is started by clicking the **Stop** button:

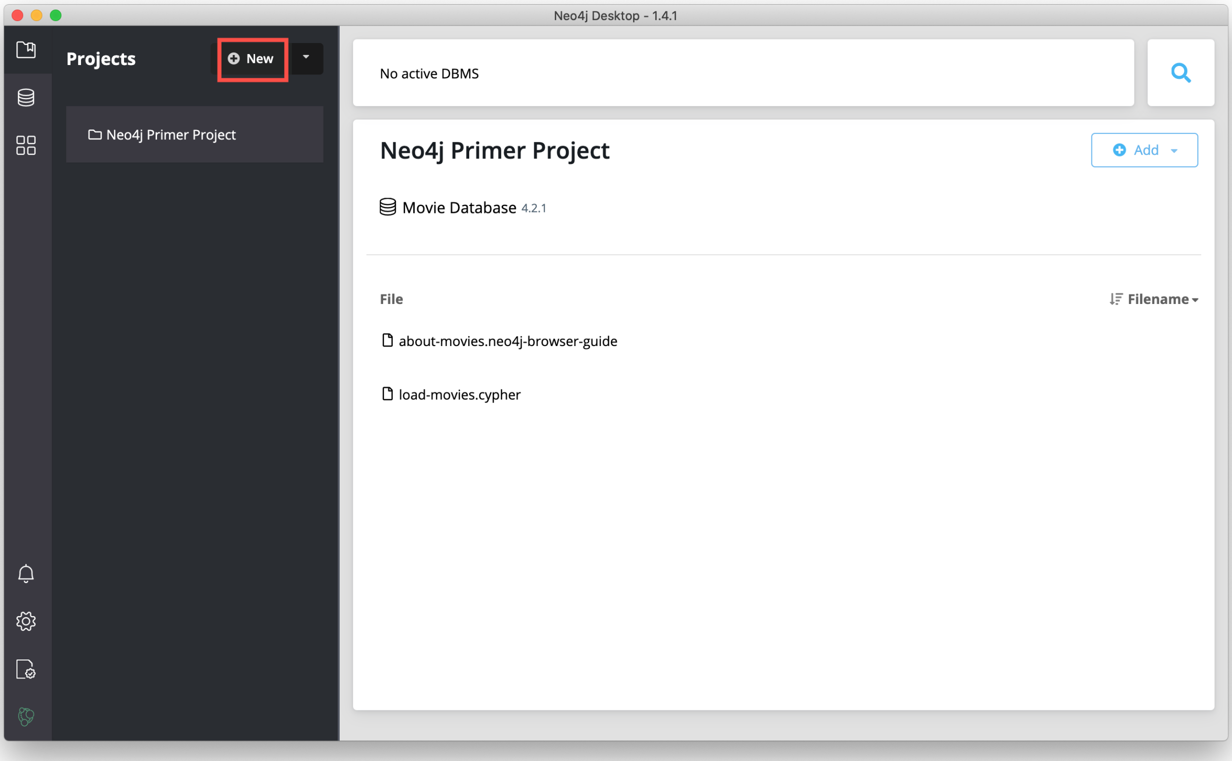


Neo4j Desktop - 1.4.1

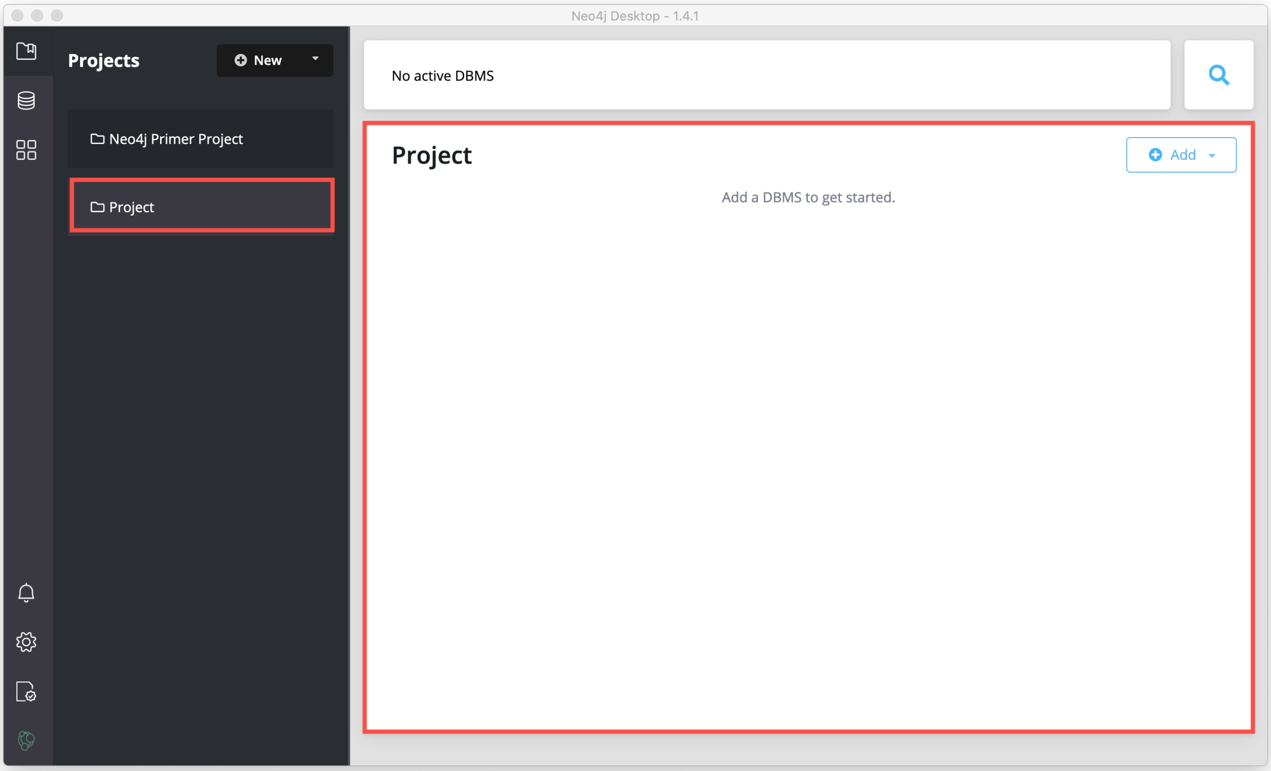
You should now see that there is no active DBMS:



Add a new project by clicking **New** at the top of the sidebar:

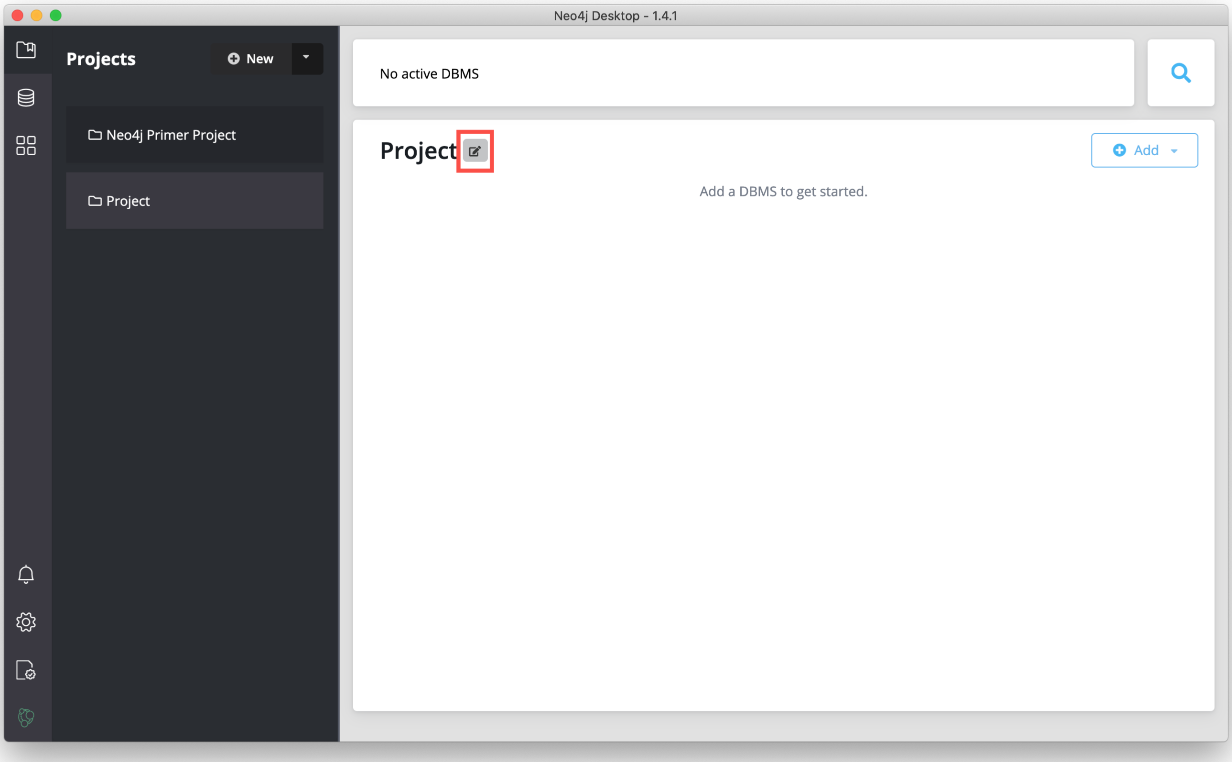


This creates a project named **Project**:



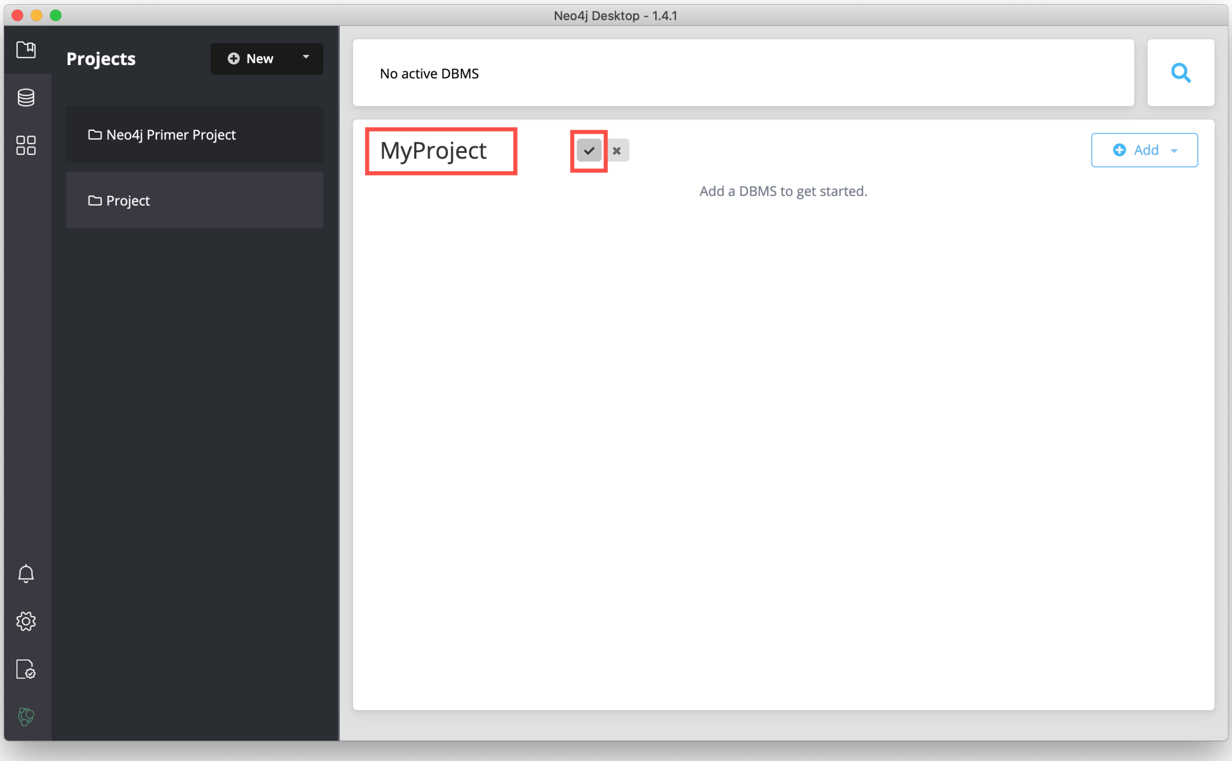
Neo4j Desktop - 1.4.1

You want your projects to be named something that describes the goal of the project. You can change the name of a project by hovering over the area to the right of the project name and selecting the **Edit** button:



Neo4j Desktop - 1.4.1

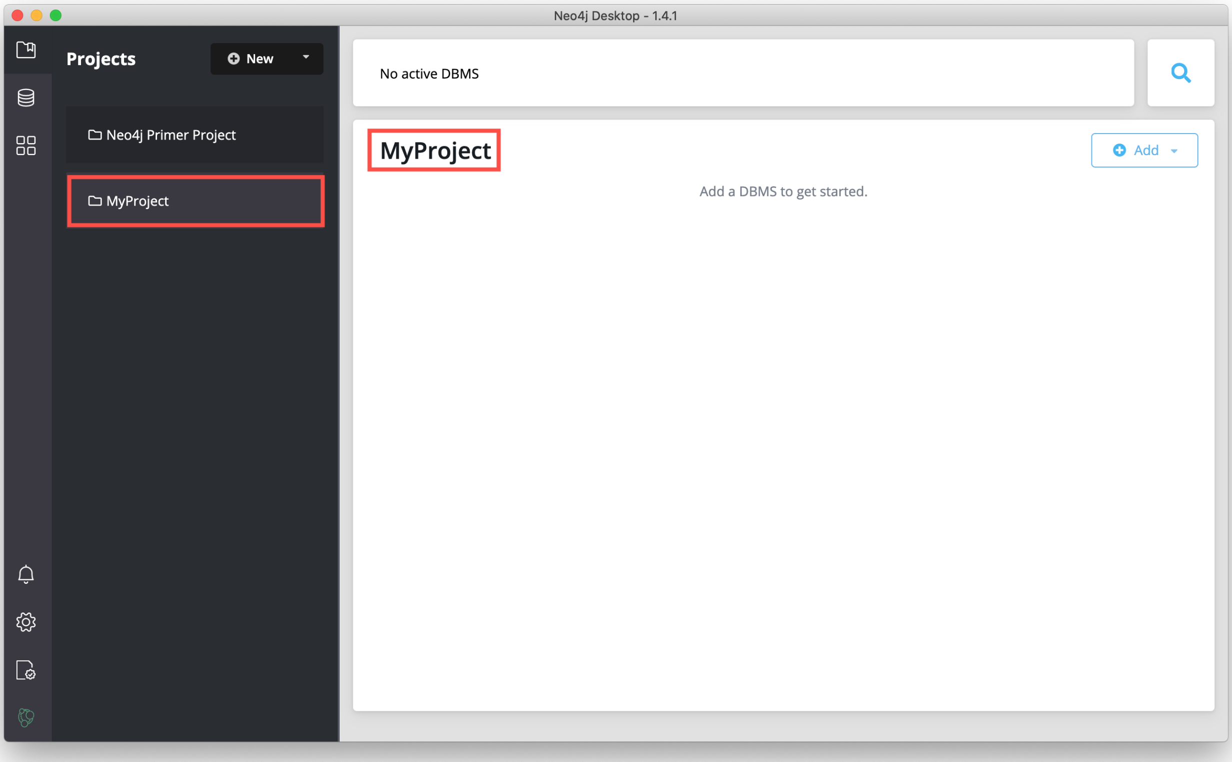
Then you type a name for the project and select the "Check" button to save the name.



No active DBMS

Add a DBMS to get started.

And here we see the project with the changed name:

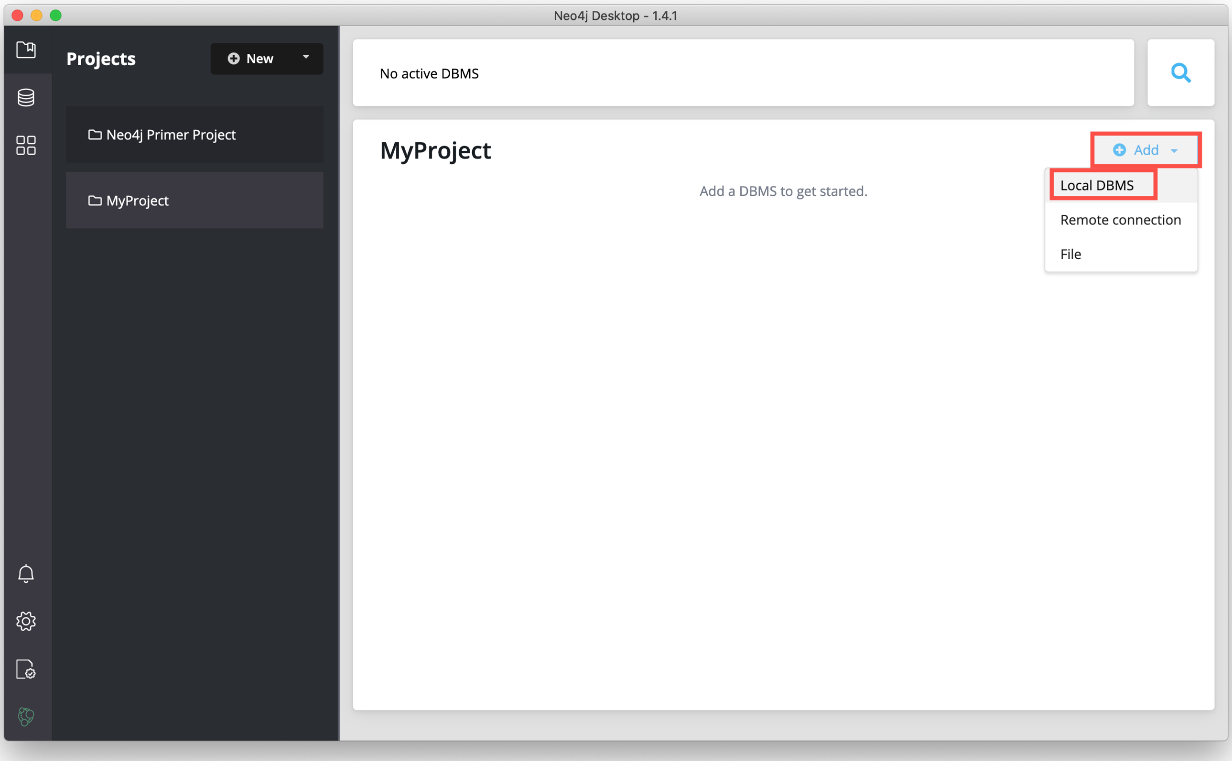


No active DBMS

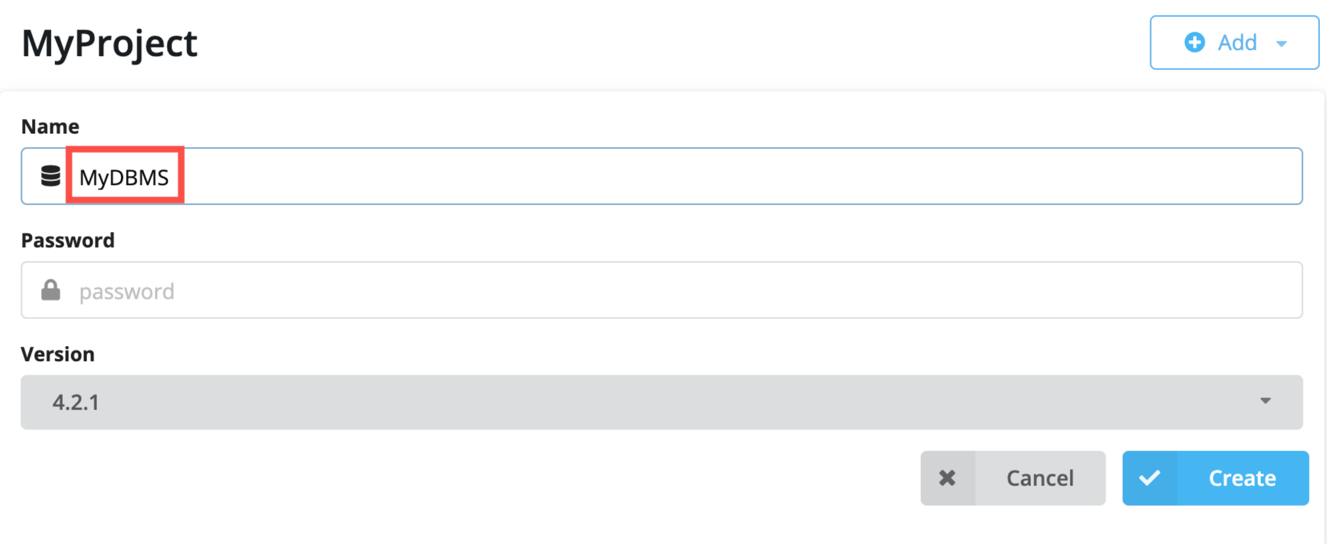
Add a DBMS to get started.

In a Neo4j Desktop project, you can create one or more DBMSs. Next, you will create a local DBMS in a project and start it.

In the project where you want to add the DBMS, click the **Add** button, and then select **Local DBMS**:



This opens a dialog where you will specify the details of the DBMS. You can use the default name for the DBMS, which is Graph DBMS, but you should name it something that helps to identify the use case for the DBMS. Here we specify MyDBMS as the name:

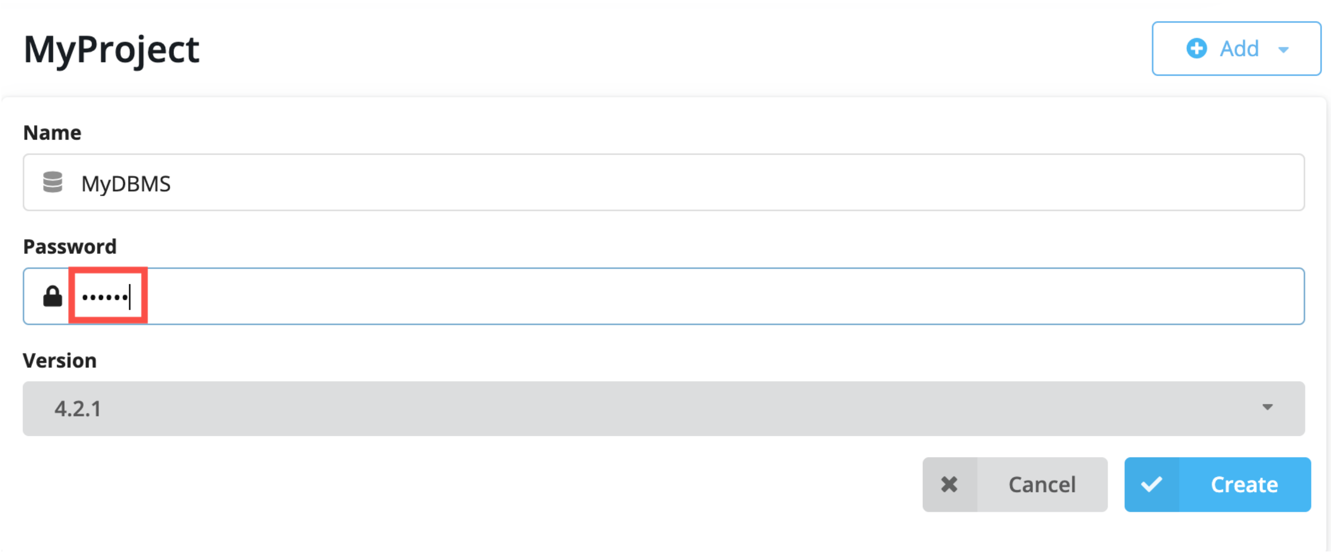


Name: MyDBMS

Password: password

Version: 4.2.1

You must specify a password for the DBMS so enter a password that you will remember:

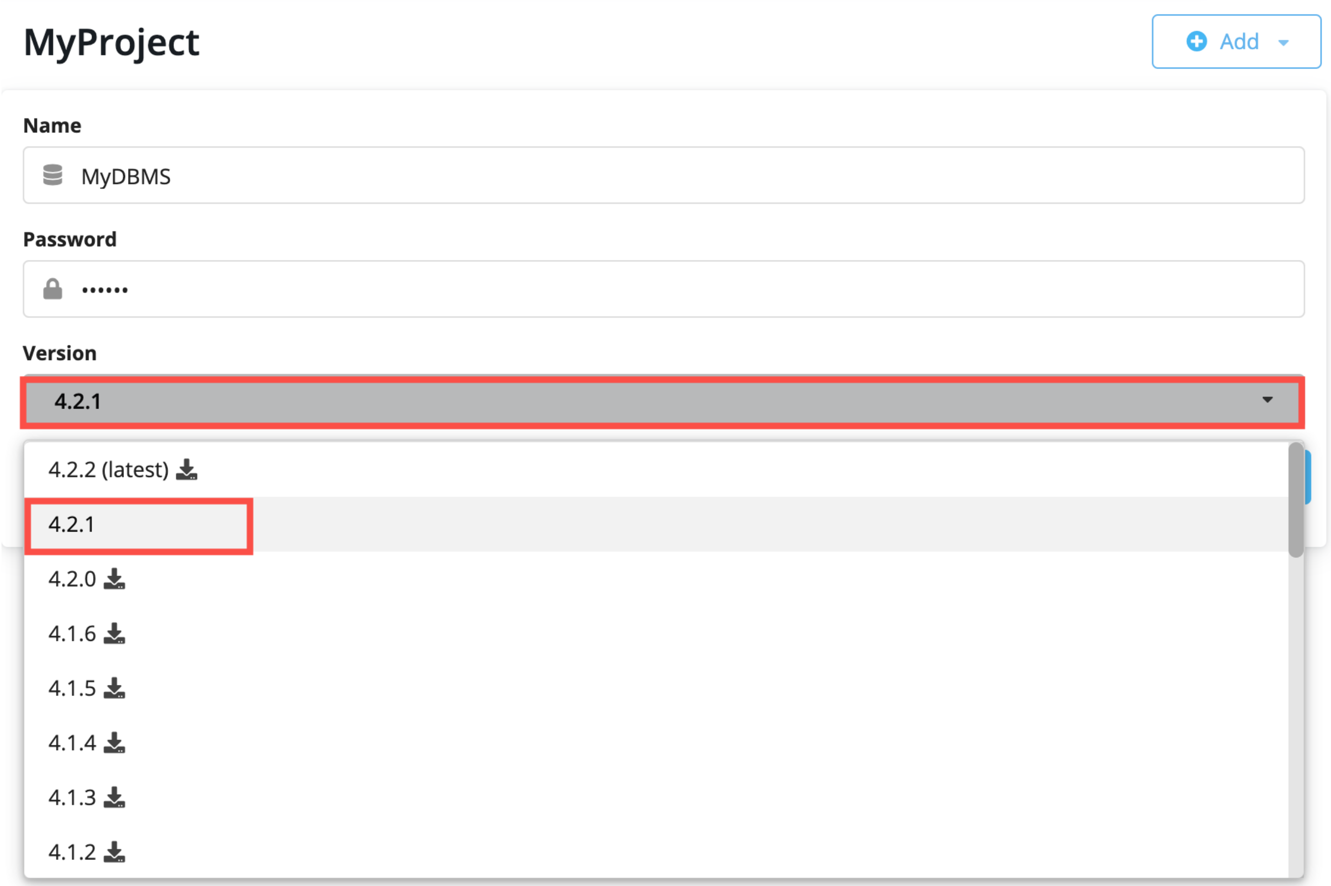


Name
MyDBMS

Password
••••••

Version
4.2.1

Neo4j Desktop will create a DBMS with the default version for Neo4j Desktop. However, you can select a different version. Keep in mind that if there is a down arrow shown next to the version, this means that Neo4j Desktop will need to download resources for that particular version of the DBMS. To do this you must be connected to the Internet:

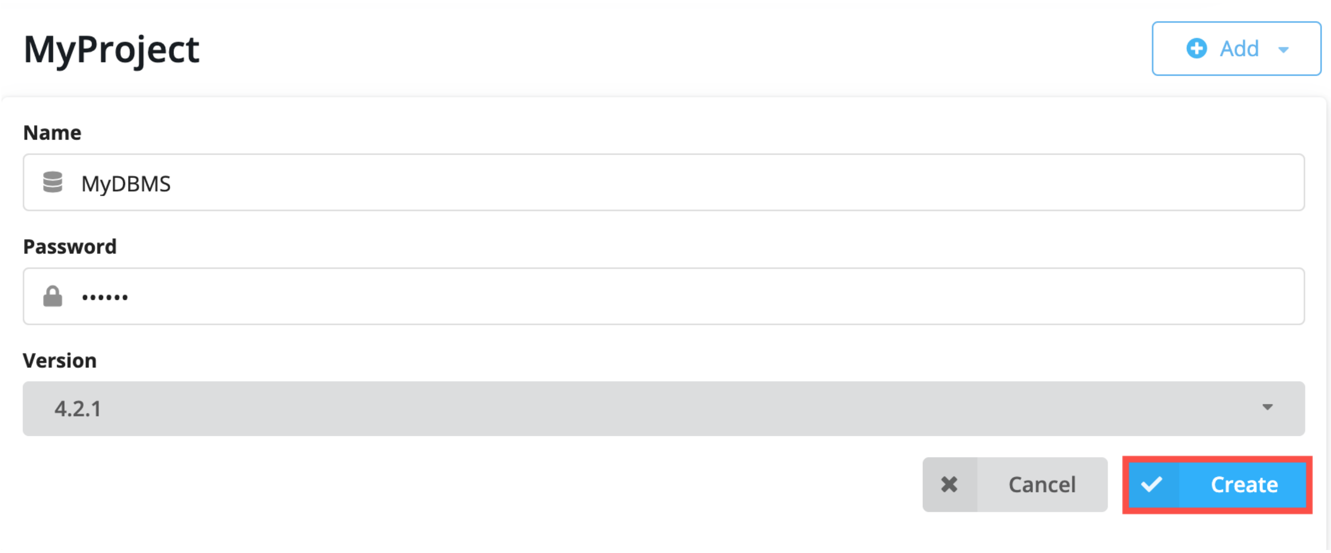


Name
MyDBMS

Password
••••••

Version
4.2.1

After you have specifed the details for the DBMS you want to create, you click the **Create** button:



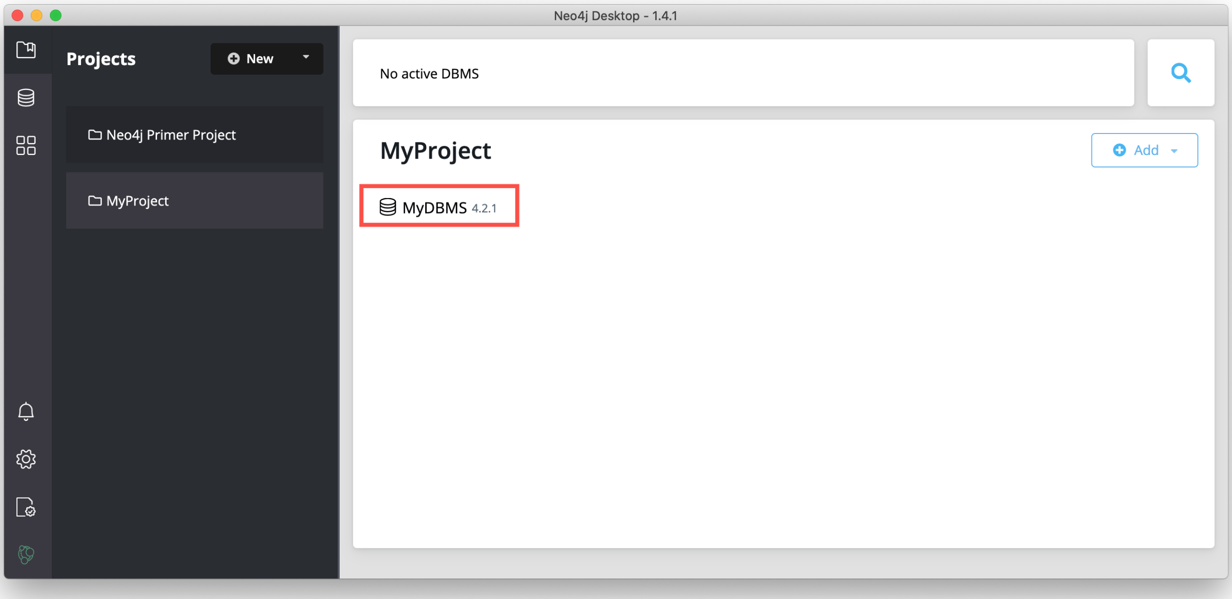
MyProject

Name: MyDBMS

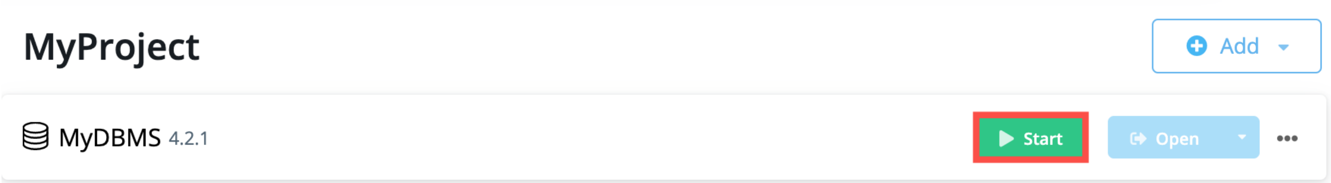
Password: ••••••

Version: 4.2.1

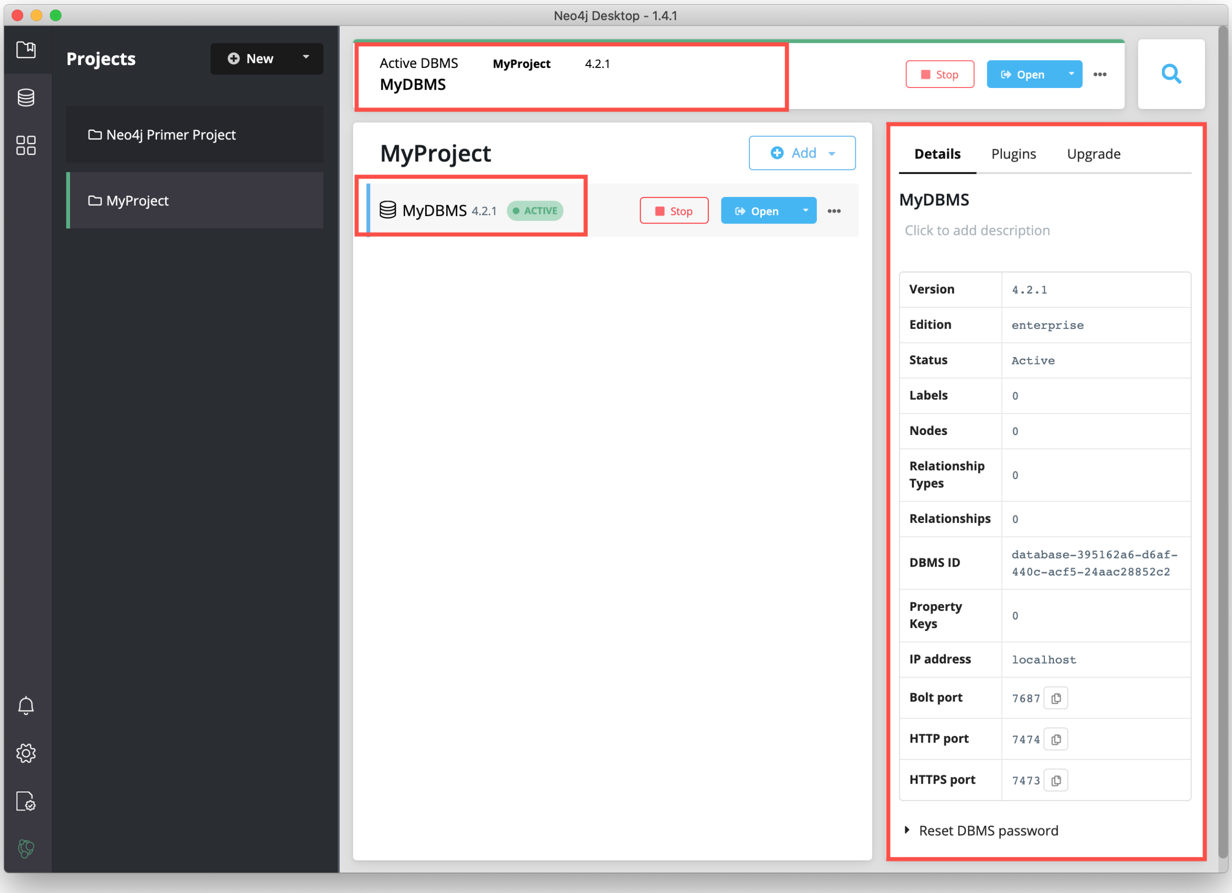
And here is what you should see after the DBMS is successfully created:



Recall that you cannot have more than one DBMS started. Provided you have no other DBMSs started in Neo4j Desktop, you can start your newly-created DBMS by hovering to the right of the DBMS name and clicking the **Start** button:



The DBMS will take a few seconds to start. After it is started, you should see something like this:



MyDBMS

Click to add description

Version	4.2.1
Edition	enterprise
Status	Active
Labels	0
Nodes	0
Relationship Types	0
Relationships	0
DBMS ID	database-395162a6-d6af-440c-acf5-24aac28852c2
Property Keys	0
IP address	localhost
Bolt port	7687
HTTP port	7474
HTTPS port	7473

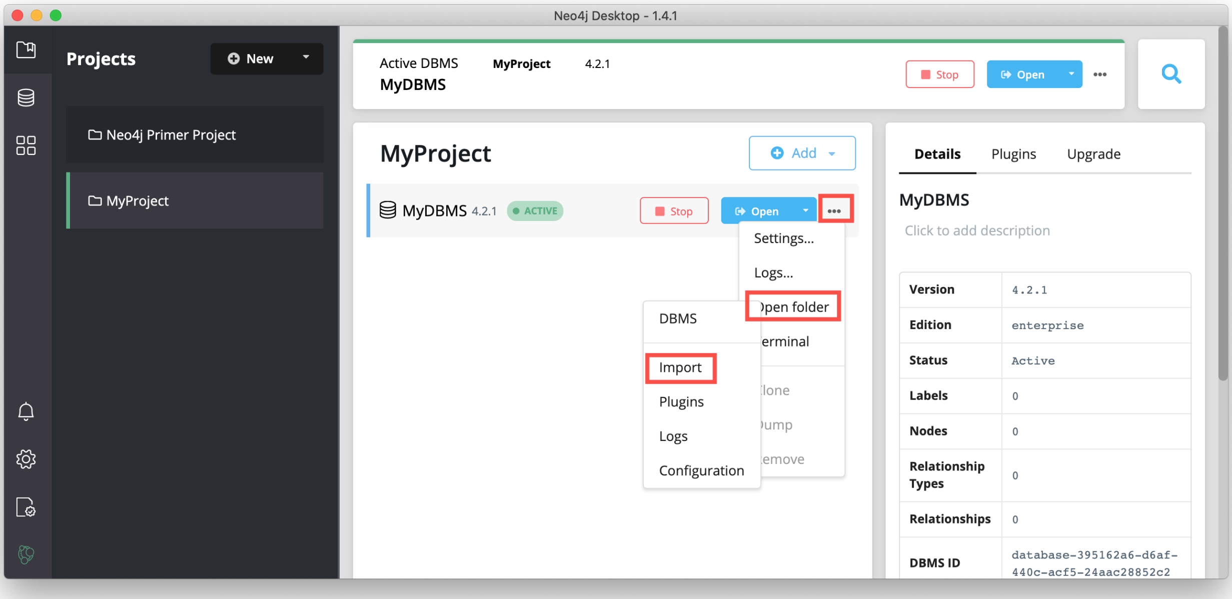
Reset DBMS password

After the DBMS is started, you can access it with clients running on your system such as Neo4j Browser, Neo4j Bloom, etc. In Neo4j Desktop, the DBMS is an Enterprise Server, but it can only be accessed locally.

2. Putting CSV files in the import folder

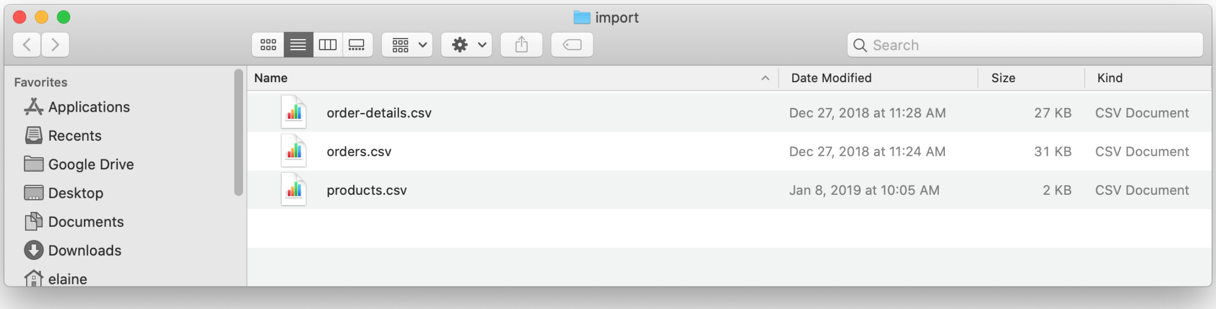
First, download this zip file. Uncompress/unzip this file which should yield three CSV files for products, orders, and order details. You will put these files in the **import** folder that Neo4j expects for imports.

Open a finder window on your system by hovering over the three dots to the right started DBMS and selecting **Open folder**, and then **Import**:



Version	4.2.1
Edition	enterprise
Status	Active
Labels	0
Nodes	0
Relationship Types	0
Relationships	0
DBMS ID	database-395162a6-d6af-440c-acf5-24aac28852c2

Copy or move the three CSV files into this **import** folder on your system:



Name	Date Modified	Size	Kind
order-details.csv	Dec 27, 2018 at 11:28 AM	27 KB	CSV Document
orders.csv	Dec 27, 2018 at 11:24 AM	31 KB	CSV Document
products.csv	Jan 8, 2019 at 10:05 AM	2 KB	CSV Document

Now that your files are in the **import** folder, you can import the data into the database managed by the DBMS. You will use the current table and column format in the CSV files and translate it into nodes and relationships. This can be done a few different ways, but you will use Cypher's `LOAD CSV` command in this guide.

LOAD CSV

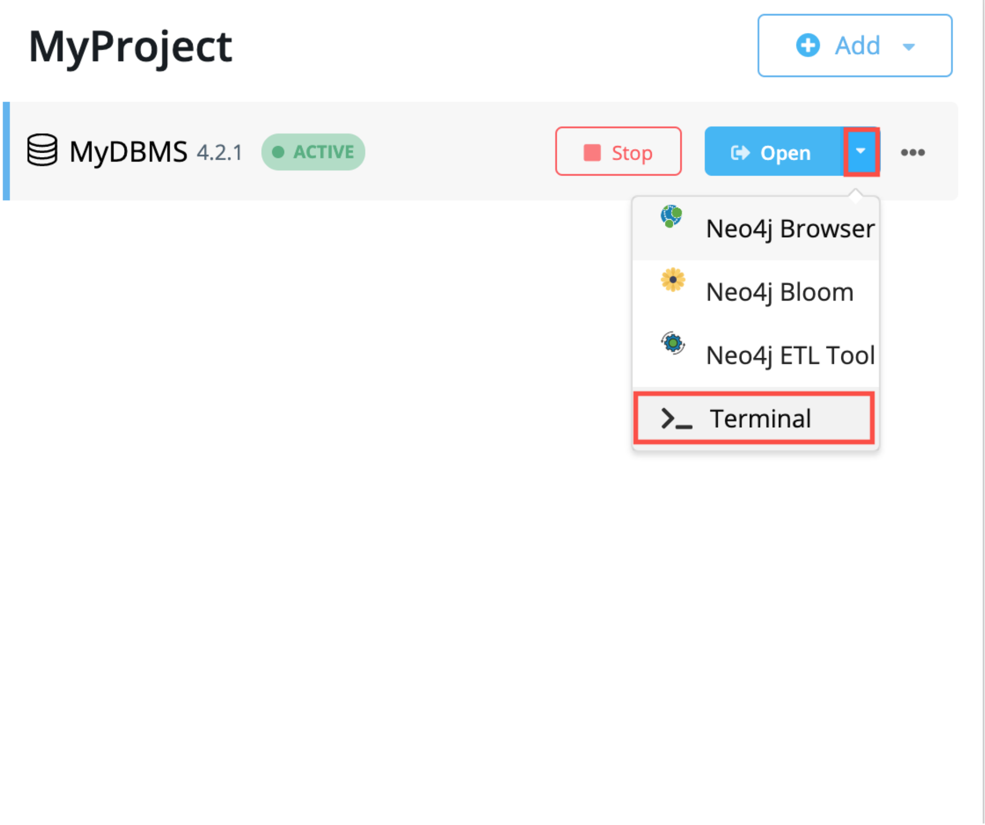
`LOAD CSV` is a built-in command in Cypher that allows you to read CSV files and append regular Cypher statements to create or update the data as a graph. You can also use `LOAD CSV` without creating the graph to output samples, counts, or distributions. This helps to detect incorrect header column counts, delimiters, quotes, escapes, or spelling of header names before the data is written and stored.

To enter and run Cypher statements on a started DBMS, you can:

1. Use Neo4j Browser:
    a. Click the **Open** button for the started DBMS
    b. Type or copy Cypher statements into the edit pane.
    c. Execute the Cypher with the `play` button on the right.

2. Use cypher-shell:

   a. Click the drop-down menu to the right of the **Open** button and select **Terminal**.



a. Enter `bin/cypher-shell`.

b. Enter **neo4j** for the user.

c. Enter the password you specified for the DBMS.

d. All Cypher statements must end with ";"

e. Use `:exit` to quit.

In Step 2, you downloaded the **.zip** file and copied its CSV files to the **import** folder for the DBMS. Before you insert anything into your graph database, you should inspect the data in the files a bit. To do this, you can use the `LOAD CSV` statement. If you opened the files previously, you may have noticed that two of the files have headers and one does not (**products.csv**). Let us see how to inspect each type of file.

First, you can check how many lines are in the CSV files to ensure they didn't get corrupted or cut off from a potential export process. For the files with headers, you simply add the `WITH HEADERS` clause after `LOAD CSV`, so that it excludes the header row in the count and only counts the rows of data.

You should execute this Cypher:

```
//count data rows in products.csv (no headers)
LOAD CSV FROM 'file:///products.csv' AS row
RETURN count(row);
```

```
//count data rows in orders.csv (headers)
LOAD CSV WITH HEADERS FROM 'file:///orders.csv' AS row
RETURN count(row);
```

```
//count data rows in order-details.csv (headers)
LOAD CSV WITH HEADERS FROM 'file:///order-details.csv' AS row
RETURN count(row);
```

Running these statements should return the following counts:

- 77 rows for **products.csv**
- 830 rows for **orders.csv**
- 2155 rows for **order-details.csv**

View data with `LOAD CSV`

Next, you can take a look at what the data looks like in the CSV files and how `LOAD CSV` sees it. The only line you need to change from the Cypher above is the `RETURN` statement. Since all of these files have several rows, you will use `LIMIT` to only get a sample.

```
//view data rows in products.csv
LOAD CSV FROM 'file:///products.csv' AS row
RETURN row
LIMIT 3;
```

Your results should look something like this:

row
["1", "Chai", "18"]
["2", "Chang", "19"]
["3", "Aniseed Syrup", "10"]

```
//count data rows in orders.csv (headers)
LOAD CSV WITH HEADERS FROM 'file:///orders.csv' AS row
RETURN row
LIMIT 5;
```

Your results should look something like this:

row
`{ "orderID": "10248", "orderDate": "1996-07-04 00:00:00.000", "shipCountry": "France" }`
`{ "orderID": "10249", "orderDate": "1996-07-05 00:00:00.000", "shipCountry": "Germany" }`
`{ "orderID": "10250", "orderDate": "1996-07-08 00:00:00.000", "shipCountry": "Brazil" }`
`{ "orderID": "10251", "orderDate": "1996-07-08 00:00:00.000", "shipCountry": "France" }`
`{ "orderID": "10252", "orderDate": "1996-07-09 00:00:00.000", "shipCountry": "Belgium" }`

```
//count data rows in order-details.csv (headers)
LOAD CSV WITH HEADERS FROM 'file:///order-details.csv' AS row
RETURN row
LIMIT 8;
```

Your results should look something like this:

row
{ "quantity": "12", "productID": "11", "orderID": "10248" }
{ "quantity": "10", "productID": "42", "orderID": "10248" }
{ "quantity": "5", "productID": "72", "orderID": "10248" }
{ "quantity": "9", "productID": "14", "orderID": "10249" }
{ "quantity": "40", "productID": "51", "orderID": "10249" }
{ "quantity": "10", "productID": "41", "orderID": "10250" }
{ "quantity": "35", "productID": "51", "orderID": "10250" }
{ "quantity": "15", "productID": "65", "orderID": "10250" }

Notice that the **orders.csv** and the **order-details.csv** return in a different format from the **products.csv**. This is because those files have headers, so the column names are returned with the values for those rows. Since the **products.csv** does not have column names, then `LOAD CSV` just returns the plain data row from the file.

Filtering what you load with `LOAD CSV`

After inspecting the data, you may only want to view or load a subset of the data in the CSV file. You can filter what you view (or load) as follows:

```
//count data rows in orders.csv (headers)
LOAD CSV WITH HEADERS FROM 'file:///orders.csv' AS row
WITH row WHERE row.shipCountry = 'Germany'
RETURN row
LIMIT 5;
```

Your results should look something like this:

row
{ "orderID": "10249", "orderDate": "1996-07-05 00:00:00.000", "shipCountry": "Germany" }
{ "orderID": "10260", "orderDate": "1996-07-19 00:00:00.000", "shipCountry": "Germany" }
{ "orderID": "10267", "orderDate": "1996-07-29 00:00:00.000", "shipCountry": "Germany" }
{ "orderID": "10273", "orderDate": "1996-08-05 00:00:00.000", "shipCountry": "Germany" }
{ "orderID": "10277", "orderDate": "1996-08-09 00:00:00.000", "shipCountry": "Germany" }

Data types

The `LOAD CSV` command reads all values as a string. No matter how the value appears in a file, it will be loaded as a string with `LOAD CSV`. So, before you import, you want to ensure you convert any values that are non-string.

There are a variety of conversion functions in Cypher. The ones you will use for this exercise are as follows:

- **toInteger():** converts a value to an integer.

- **toFloat():** converts a value to a float (in this case, for monetary amounts).
- **datetime():** converts a value to a datetime.

We look at the values in each CSV file to determine what needs to be converted.

Products.csv

The values in the products.csv files are for product ID, product name, and unit cost. Product ID looks like an integer value that increases with each row, so you can convert this to an integer using the `toInteger()` function in Cypher. Product name can remain a string since it consists of characters. The final column is the product unit cost. Though the sample values from your inspection are all whole numbers, we know that monetary amounts often have decimal place values, so we will convert these values to floats using the `toFloat()` function.

You can see the Cypher to handle all of these conversions below; however, you are still not loading the values into Neo4j yet. You will be just viewing the CSV files with converted values.

```
LOAD CSV FROM 'file:///products.csv' AS row
WITH toInteger(row[0]) AS productId, row[1] AS productName, toFloat(row[2]) AS unitCost
RETURN productId, productName, unitCost
LIMIT 3;
```

Your results should look something like this:

productId	productName	unitCost
1	"Chai"	18.0
2	"Chang"	19.0
3	"Aniseed Syrup"	10.0

Note that we are using collection positions (row[0], row[1], row[2]) to refer to the columns in the row and improve readability by using aliases to reference them in the return. In a file that has no headers, this is how to reference values in each position.

Orders.csv

The values in the orders.csv (per the column names) are for orderID, orderDate, and shipCountry. Again, you can evaluate the values and determine any conversions to apply.

OrderID looks like an integer, so you can convert that using the `toInteger()` function. The orderDate column is certainly in a date format and will require us to format it using the `datetime()` function. Finally, the shipCountry values are characters, so you can leave that as a string.

Just as you did with the last CSV files, let us look at the results of these conversions without importing the data.

```
LOAD CSV WITH HEADERS FROM 'file:///orders.csv' AS row
WITH toInteger(row.orderID) AS orderId, datetime(replace(row.orderDate,' ','T')) AS orderDate,
row.shipCountry AS country
RETURN orderId, orderDate, country
LIMIT 5;
```

Your results should look something like this:

orderId	orderDate	country
10248	"1996-07-04T00:00:00Z"	"France"
10249	"1996-07-05T00:00:00Z"	"Germany"
10250	"1996-07-08T00:00:00Z"	"Brazil"
10251	"1996-07-08T00:00:00Z"	"France"
10252	"1996-07-09T00:00:00Z"	"Belgium"

There was one tricky thing with this CSV in the `orderDate` column. Neo4j's datetime uses the ISO 8601 format which uses the delimiter `T` between the date and time values. The CSV file does not have the 'T' joining the date and time values but has a space between them instead. You used the `replace()` function to change the space to the character 'T' and get the string into the expected format. Then, you wrapped the `datetime()` function around that to convert the changed string to a datetime value.

Order-details.csv

The values in the order-details.csv (from column names) are for productID, orderID, and quantity. Let us look at which ones need to be converted.

product ID is also from the products.csv file, where you converted that value to an integer. You will do the same here to ensure you match formats. The order ID field contains values from the orders.csv file, so you will match your previous conversion and translate this field to an integer, as well. The quantity field in this file is a numeric value. You can convert this to an integer with the `toInteger()` function you have been using.

The results of these conversions are in the code below. Remember that you still are not loading any data yet.

```
LOAD CSV WITH HEADERS FROM 'file:///order-details.csv' AS row
WITH toInteger(row.productID) AS productId, toInteger(row.orderID) AS orderId, toInteger(row.quantity) AS
quantityOrdered
RETURN productId, orderId, quantityOrdered
LIMIT 8;
```

Your results should look something like this:

productId	orderId	quantityOrdered
11	10248	12
42	10248	10
72	10248	5
14	10249	9
51	10249	40
41	10250	10
51	10250	35

productId	orderId	quantityOrdered
65	10250	15

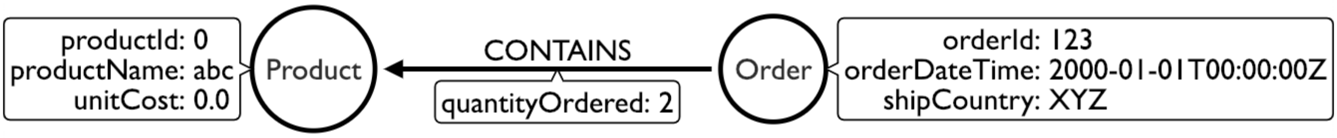
Loading the data!

Now that you have determined that the CSV file data looks OK, and you have verified how `LOAD CSV` sees the data and converted any non-string values, you are almost ready to create the data in our graph database! To do that, you will use Cypher statements alongside the `LOAD CSV` commands you used above. The `LOAD CSV` will read the files, and the Cypher statements will create the data in your database.

Graph data model

An important step you need before writing Cypher statements, though, is to determine what the graph structure should look like once you import your file data. After all, importing the data in the existing table and column data will not provide the value you want to achieve from a graph. To utilize the graph database fully, you need a graph data model.

Though there are a variety ways to organize the products and orders in your files, we will save that for another guide and use the below version of the model for this exercise.



We have two nodes - one for a product and one for an order. Each of those nodes have properties from the CSV files. For the `Product`, we have ID, name, and unit cost. For the `Order`, we have ID, date/time, and country where it is going.

The order-details.csv file defines the relationship between those two nodes. That file has the product ID, the order ID it belongs to, and the quantity of the product on the order. So, in the data model, this becomes the `CONTAINS` relationship between `Product` and `Order` nodes. We also include a property of `quantityOrdered` on the relationship because the product quantity value only exists when a product is related to an order.

Now that you know the types of nodes and relationships you will have and the properties involved, you can construct the Cypher statements to create the data for this model.

Avoiding duplicates and increasing performance

One final thing you need to think about before you create data in the graph is ensuring values are unique and performance is efficient. To handle this, you can use constraints. Just as with other databases, constraints ensure data integrity criteria is not violated, while simultaneously indexing the property with the constraint for faster query performance.

There are cases for applying indexes to a database before any data and with existing data. In this exercise, you will add two constraints before you create any data - one for `productId` and one for `orderId`. This will ensure that, when you create a new node of each of those types or a relationship to connect them, you know the entities are unique and indexed.

Below is the Cypher for adding indexes.

```
CREATE CONSTRAINT UniqueProduct ON (p:Product) ASSERT p.id IS UNIQUE;
CREATE CONSTRAINT UniqueOrder ON (o:Order) ASSERT o.id IS UNIQUE;
```

Cypher

Now you are ready to write the Cypher for creating the data in the graph! You could use `CREATE` statements where you are sure that you will not have duplicate rows in your CSV file and use `MATCH` to find existing data for updates. However, since it is hard to completely scrub all data and import perfectly clean data from any source, you will use `MERGE` statements to check if the data exists before inserting. If the node or relationship exists, Cypher will match and return (without any writes), but if it does not exist, Cypher will insert it. Using `MERGE` can have some performance overhead, but often it is the better approach to maintain high data integrity.

> **Why both constraints and MERGE?** Using constraints is different from using MERGE. Statements that create data in violation of the constraint will error, while statements that use `MERGE` will simply return existing values (no errors). If you use both, you avoid terminating your load statements due to constraint violations, and you also ensure you don't accidentally create duplicates in adhoc queries.

Products

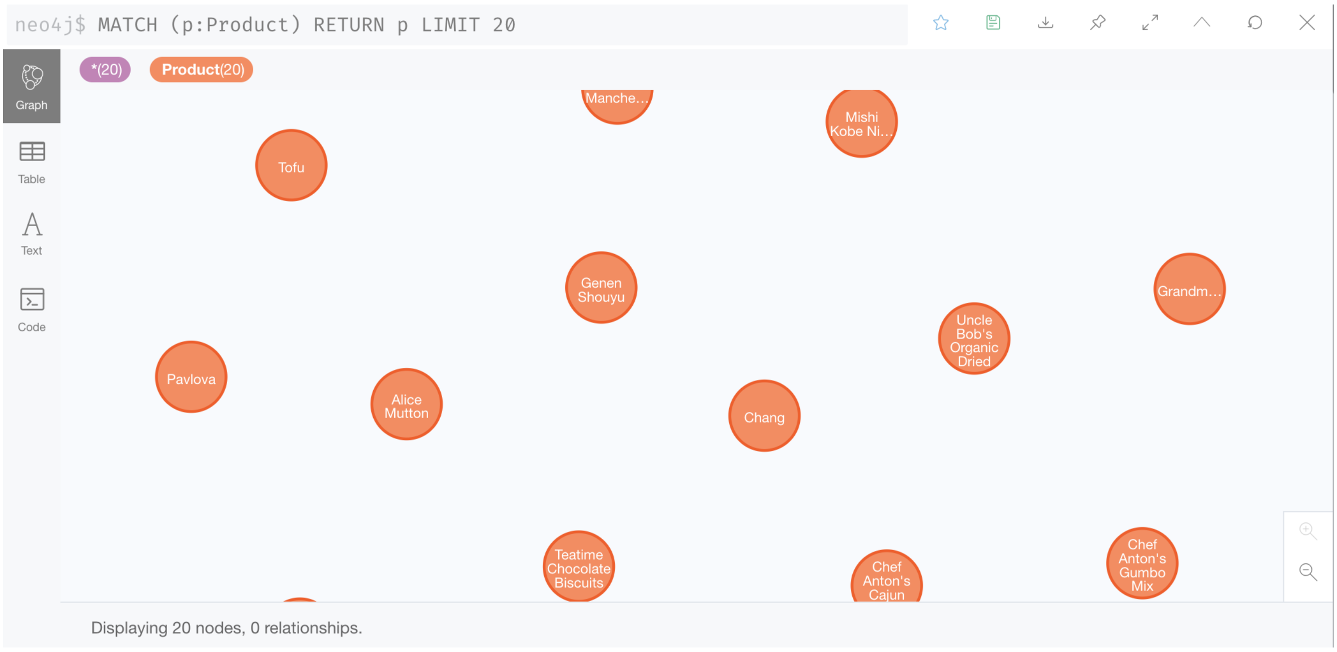
To start, you will load the products into the graph. You start with your `LOAD CSV` statement from above, then you add the Cypher to create the data from the CSV into your graph model. You will use the `MERGE` statement to check if the `Product` already exists before you create it. The properties will be set to the converted values you handled earlier in this guide.

```
LOAD CSV FROM 'file:///products.csv' AS row
WITH toInteger(row[0]) AS productId, row[1] AS productName, toFloat(row[2]) AS unitCost
MERGE (p:Product {productId: productId})
  SET p.productName = productName, p.unitCost = unitCost
RETURN count(p);
```

If you run that statement, it will return the number of product nodes (`count(p)`) that were created in the database. You can cross-check that number with the number of rows in the CSV file from earlier (77 rows in products.csv). You can also run a validation query to return a sample of nodes and review that the properties look accurate.

```
//validate products loaded correctly
MATCH (p:Product)
RETURN p LIMIT 20;
```

Here are the results in Neo4j Browser:



```
neo4j$ MATCH (p:Product) RETURN p LIMIT 20
```

Displaying 20 nodes, 0 relationships.

Orders

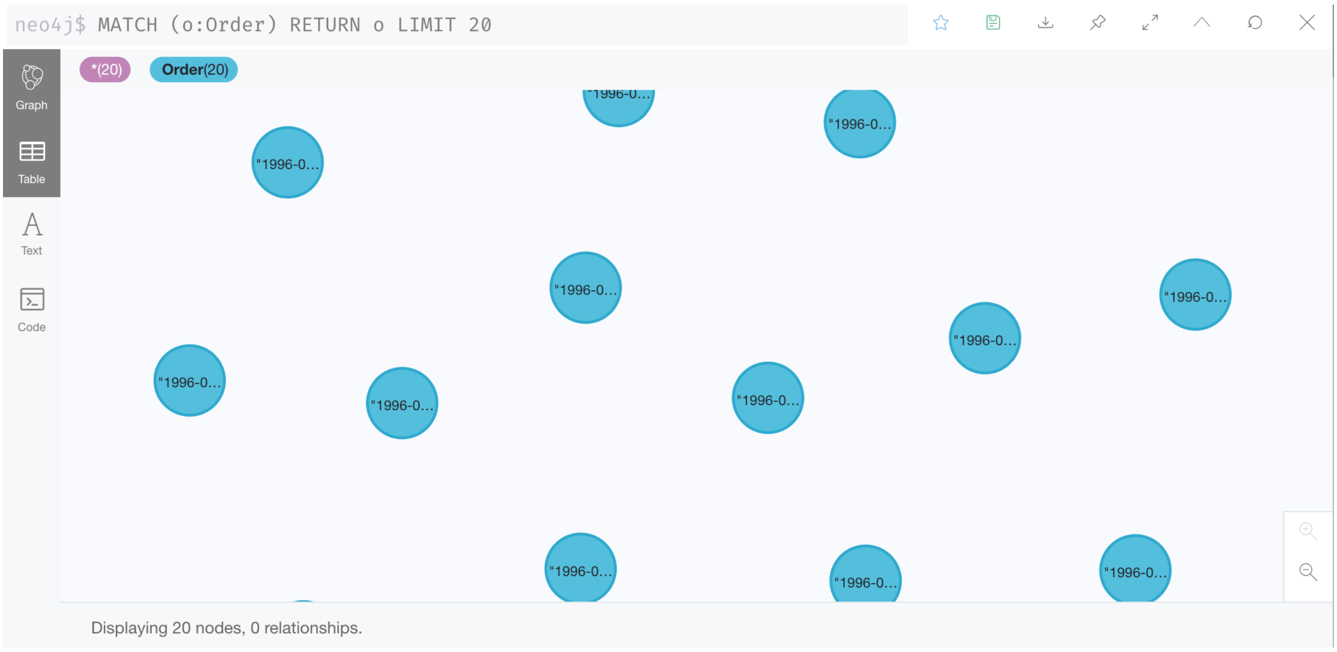
Next, you will load the orders. Again, since you want to verify you do not create duplicate `Order` nodes, you can use the `MERGE` statement. Just as with products, you start with the `LOAD CSV` command, then add Cypher statements and include your data conversions.

```
LOAD CSV WITH HEADERS FROM 'file:///orders.csv' AS row
WITH toInteger(row.orderID) AS orderId, datetime(replace(row.orderDate,' ','T')) AS orderDate,
row.shipCountry AS country
MERGE (o:Order {orderId: orderId})
  SET o.orderDateTime = orderDate, o.shipCountry = country
RETURN count(o);
```

You can also run a validation query, as before, to verify the graph data looks correct.

```
//validate orders loaded correctly
MATCH (o:Order)
RETURN o LIMIT 20;
```

Here are the results in Neo4j Browser:



neo4j$ MATCH (o:Order) RETURN o LIMIT 20

Displaying 20 nodes, 0 relationships.

Order-details

Last, but not least, you will create the relationship between the products and the orders. Since you expect all of your products and all of your orders to already exist in the graph (that data should have been loaded with the last two files), then you start with MATCH to find the existing Product and Order nodes. Then, the MERGE statement will add the new relationship or match an existing one.

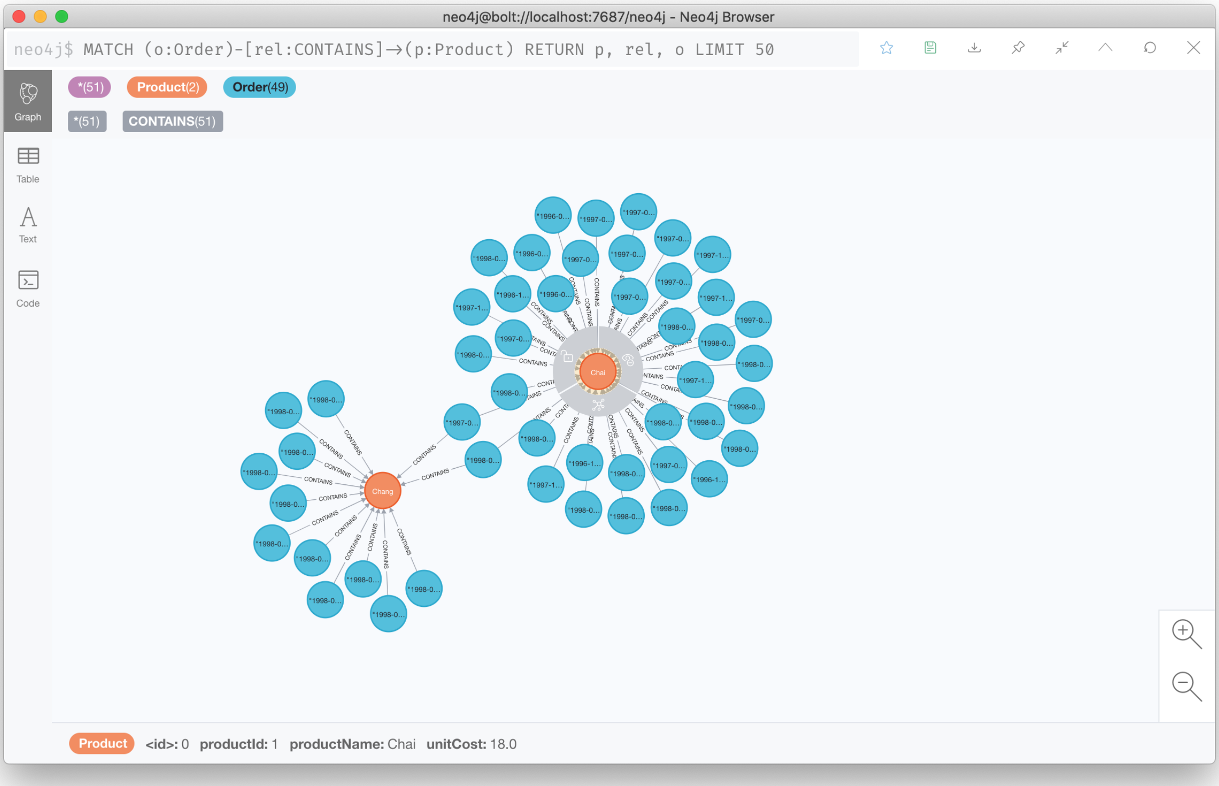
As you found when you ran a count on the order-details file above, there are 2,155 rows in the CSV. While this is not a huge number for file imports, you will have Cypher periodically commit the data to the database to reduce the memory overhead of the transaction state. For this, you can add the :auto USING PERIODIC COMMIT clause before the LOAD CSV command. The default value for periodic commit is 1,000, but for this exercise, you will ask Cypher to commit every **500 rows**. You could decrease this number if you have a lot of memory already allocated to other tasks, or if it is limited.

```
:auto USING PERIODIC COMMIT 500
LOAD CSV WITH HEADERS FROM 'file:///order-details.csv' AS row
WITH toInteger(row.productID) AS productId, toInteger(row.orderID) AS orderId, toInteger(row.quantity) AS
quantityOrdered
MATCH (p:Product {productId: productId})
MATCH (o:Order {orderId: orderId})
MERGE (o)-[rel:CONTAINS {quantityOrdered: quantityOrdered}]->(p)
RETURN count(rel);
```

Just as you did above, you can validate the data with the query below.

```
MATCH (o:Order)-[rel:CONTAINS]->(p:Product)
RETURN p, rel, o LIMIT 50;
```

Here are the results in Neo4j Browser:



neo4j@bolt://localhost:7687/neo4j - Neo4j Browser

```
neo4j$ MATCH (o:Order)-[rel:CONTAINS]→(p:Product) RETURN p, rel, o LIMIT 50
```

*(51) Product(2) Order(49)

*(51) CONTAINS(51)

Product <id>: 0 productId: 1 productName: Chai unitCost: 18.0

Wrapping up

Congratulations! You have successfully loaded three CSV files into a Neo4j graph database using Neo4j Desktop!

The `LOAD CSV` functionality, coupled with Cypher, is exceptionally useful for getting data from files into a graph structure. The best way to advance your skills in this area is to load a variety of files for various data sets and models. Practice makes perfect!

Increasing the challenge

If you work through this exercise again at a later time, feel free to increase the challenge by coming up with your own data model for these files or try to load some other CSV files to a graph.

If you have any questions or need assistance using `LOAD CSV`, reach out to us on the Community Site!

License