Get started with Neo4j

Table of Contents

Hub

Explore the capabilities of graph databases

With a property graph database at its core, Neo4j offers an ecosystem of tools, applications, and libraries which aim to help you get started with the technology.

What is Neo4j

Learn what a graph database is and how to use it in your projects.

What is Neo4j?

What is a graph database

Learn the principles of a graph database.

What is a graph database

What is Cypher

Learn Cypher, Neo4j's graph query language, and start thinking about graphs and patterns.

What is Cypher

Work with data

Model your data for Neo4j

Discover strategies to model your data and to improve your workflow.

/docs/model

Import data to Neo4j

Read more about the different ways of importing data to your Neo4j graph database.

/docs/import/

Create applications with Neo4j

Run through detailed examples of how to integrate Neo4j with your preferred programming language.

/docs/create-applications

Data science with Neo4j

Learn how to use Neo4j Graph Data Science, a library of graph algorithms for analysts and data scientists.

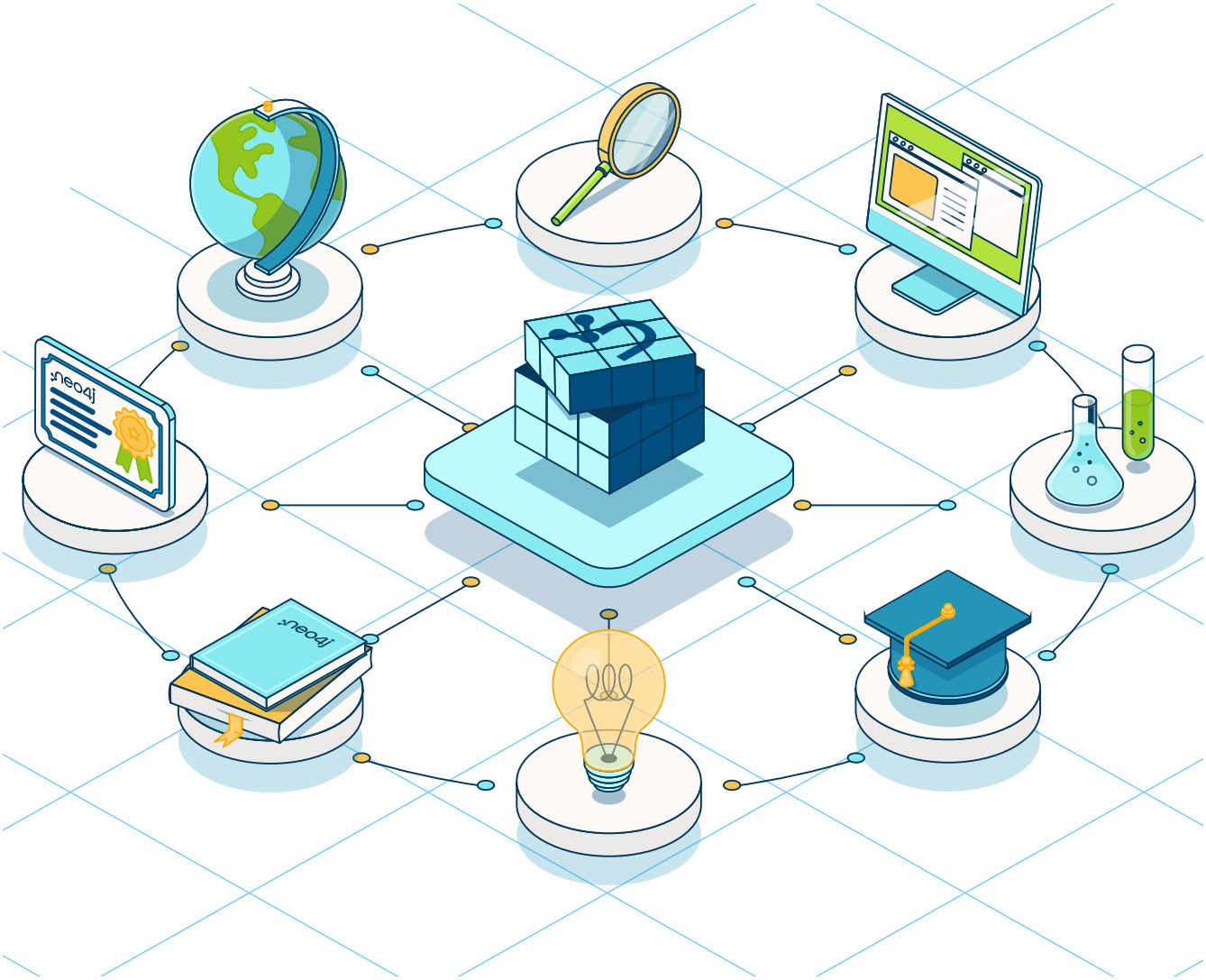
/docs/gds

Visualize data in Neo4j

Learn how to export your graph data in Neo4j for display as a visualization.

/docs/visualize/

GraphAcademy courses



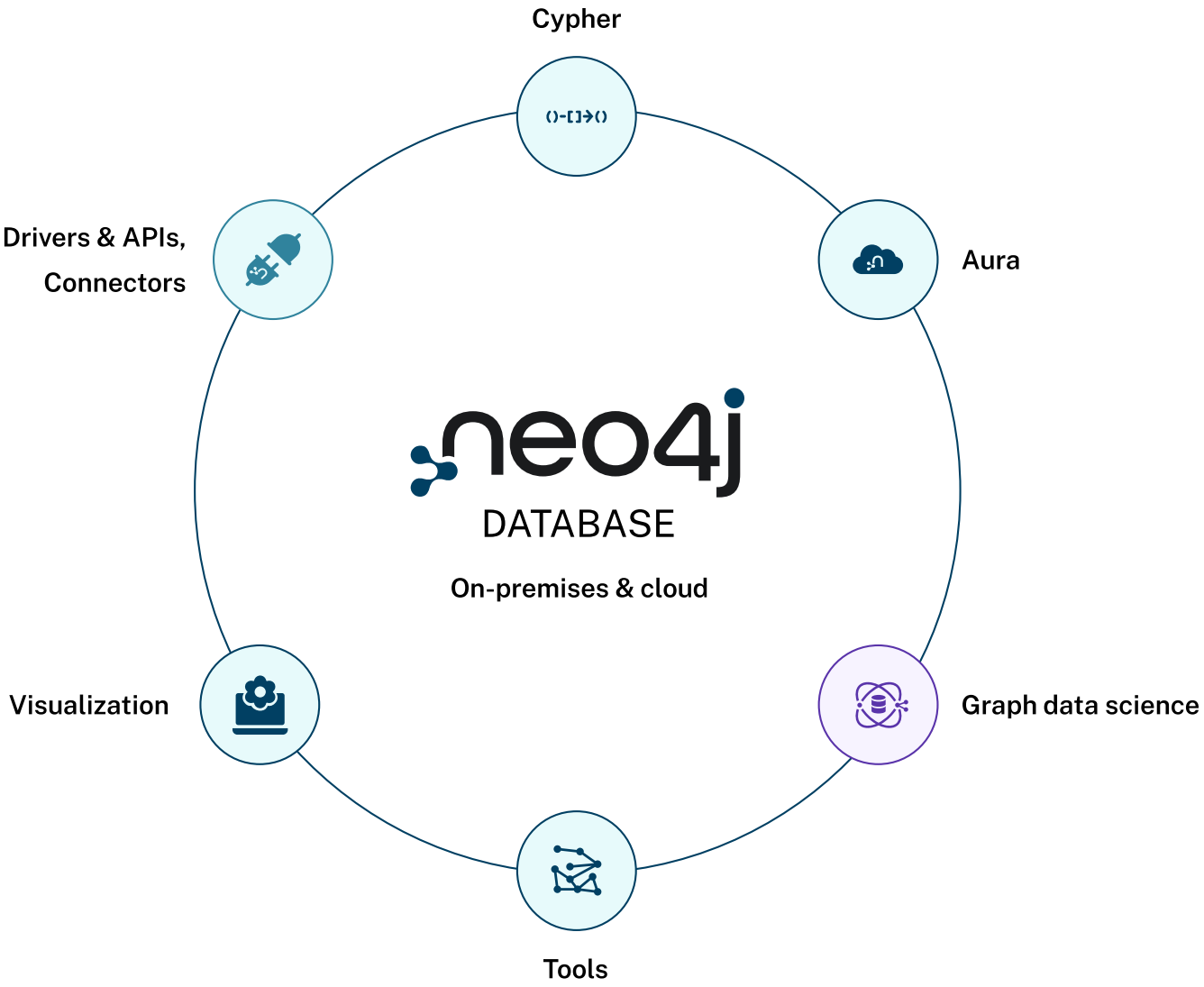
GraphAcademy courses

- Embedding & Vector Indexes Tutorial
- Import data from a relational database into Neo4j
- Build a Cypher recommendation engine
- Set up and use a composite database
- Capture and track changes in real-time
- Apply centrality algorithms to your graph

Licenses and disclaimers

Get started

What is Neo4j?



Neo4j is a *native graph database*, which means that it implements a true graph model all the way down to the storage level. Instead of using a "graph abstraction" on top of another technology, the data is stored in Neo4j in the same way you may whiteboard your ideas.

Since 2007, Neo4j has evolved into a rich ecosystem of tools, applications, and libraries. This ecosystem allows you to integrate graph technologies with your working environment in a number of ways which are here described.

Beyond the core graph, Neo4j also provides ACID transactions, cluster support, and runtime failover.

> **Note:**
>
> Neo4j is written in Java and Scala. You can check the source code on GitHub.

How to interact with Neo4j

Neo4j uses Cypher, a declarative query language similar to SQL, but optimized for graphs. The same

language is also used by other databases such as SAP HANA Graph via the openCypher project.

Another option is to use libraries. Neo4j currently supports Java, JavaScript, .NET, Python, Go, GraphQL, Spring, and more.

Create a Neo4j instance

Deploying a database is the first step towards exploring Neo4j. Select a deployment method that suits your project from the following options:

Fully managed cloud service

Neo4j AuraDB is a fully managed cloud service that allows you to start exploring Neo4j right from your browser.

If you are a data scientist, you might also want to check Neo4j AuraDS and get access to more than 65 pretuned graph algorithms.

Neo4j Aura has both free and subscription-based editions. See full comparison.

Self-managed cloud services

You can also deploy your graph database on a cloud platform of your choice. Neo4j works with Amazon Web Services (AWS), Google Cloud (GCP), and Microsoft Azure.

For self-managed cloud services, you need to install Neo4j locally or use Neo4j Desktop if your project is not in a production environment.

Neo4j is available for installation on Linux, macOs, and Windows.

Self-managed local deployment

If you prefer to work with a local deployment: install Neo4j Desktop if you are not working in a production environment or install Neo4j locally.

Neo4j on Docker

Neo4j can be run in a Docker container. An official Neo4j image that provides a standard, ready-to-run package of Neo4j Community Edition and Enterprise Edition for a variety of versions can be downloaded from the DockerHub. It is available for macOS, Windows, and Linux.

Neo4j on Kubernetes

With Neo4j Helm charts, you can deploy both a standalone and a cluster deployment of Neo4j on Kubernetes, and use configuration options suitable for the most common scenarios.

Neo4j has free and subscription-based licensing options. Read more about the available

editions.

Work with data

After creating your database, your learning can take different paths depending on whether you want to work with your own data or use Neo4j's example datasets:

- **Own data**: There are several ways to import data to Neo4j and to model it for a better experience.
- **Example datasets**: Both Aura and Neo4j Browser feature embedded guides that allow you to create example datasets and start querying. To access them, use the graduation cap icon on the top right section in Aura or write `:guide` in Neo4j Browser.
  You can also download the example datasets and then import them to your instance.

Neo4j tools

Neo4j has a catalogue of tools that can be used for various ends such as database administration, data visualization, and more. You can check all products in the Tools hub.

Supported libraries

Neo4j supports several of the most the popular query languages and also offers proprietary libraries for a customized experience:

- The Neo4j Graph Data Science (GDS) library provides implementations of common graph algorithms and machine learning pipelines to train predictive supervised models. You can use them to solve graph problems, such as predicting missing relationships, for example.
- The Object Graph Mapping (OGM) library, maps nodes and relationships in the graph to objects and references in a domain model. You can use this resource to start tracking changes and minimize necessary updates and transitive persistence (reading and updating neighborhoods of an object).

APIs

Neo4j currently offers three proprietary APIs:

- The Neo4j HTTP API allows you to execute a series of Cypher statements against a Neo4j instance through HTTP requests.
- The Change Data Capture (CDC) API allows you to capture and track changes to your database in real-time, as well as keep data sources up to date.
- The Neo4j Query API allows you to develop client applications in languages not currently supported by Neo4j.

Note:

At Neo4j Labs, you can find experimental projects including APIs, libraries, and visualization tools.

Keep learning

To learn more about what a graph database is and the concepts behind the technology, continue reading the documentation or browse other curated resources.

You can also reach out to other members of the Neo4j community on the Neo4j Community Site.

Glossary

label

Marks a node as a member of a named and indexed subset. A node may be assigned zero or more labels.

labels

A label marks a node as a member of a named and indexed subset. A node may be assigned zero or more labels.

node

A node represents an entity or discrete object in your graph data model. Nodes can be connected by relationships, hold data in properties, and are classified by labels.

nodes

A node represents an entity or discrete object in your graph data model. Nodes can be connected by relationships, hold data in properties, and are classified by labels.

relationship

A relationship represents a connection between nodes in your graph data model. Relationships connect a source node to a target node, hold data in properties, and are classified by type.

relationships

A relationship represents a connection between nodes in your graph data model. Relationships connect a source node to a target node, hold data in properties, and are classified by type.

property

Properties are key-value pairs that are used for storing data on nodes and relationships.

properties

Properties are key-value pairs that are used for storing data on nodes and relationships.

cluster

A Neo4j DBMS that spans multiple servers working together to increase fault tolerance and/or read scalability. Databases on a cluster may be configured to replicate across servers in the cluster thus achieving read scalability or high availability.

clusters

A Neo4j DBMS that spans multiple servers working together to increase fault tolerance and/or read scalability. Databases on a cluster may be configured to replicate across servers in the cluster thus

achieving read scalability or high availability.

graph

A logical representation of a set of nodes where some pairs are connected by relationships.

graphs

A logical representation of a set of nodes where some pairs are connected by relationships.

schema

The prescribed property existence and datatypes for nodes and relationships.

schemas

The prescribed property existence and datatypes for nodes and relationships.

[[database schema]]database schema

The prescribed property existence and datatypes for nodes and relationships.

indexes

Data structure that improves read performance of a database. Read more about supported categories of indexes.

indexed

Data structure that improves read performance of a database. Read more about supported categories of indexes.

constraints

Constraints are sets of data modeling rules that ensure the data is consistent and reliable. See what constraints are available in Cypher.

data model

A data model defines how information is organized in a database. A good data model will make querying and understanding your data easier. In Neo4j, the data models have a graph structure.

data models

A data model defines how information is organized in a database. A good data model will make querying and understanding your data easier. In Neo4j, the data models have a graph structure.

What is a graph database

A Neo4j graph database stores data as nodes, relationships, and properties instead of in tables or documents. This means you can organize your data in a similar way as when sketching ideas on a whiteboard.

And since graph databases are not restricted to a pre-defined model, you can take more flexible approaches and strategies when working with them.

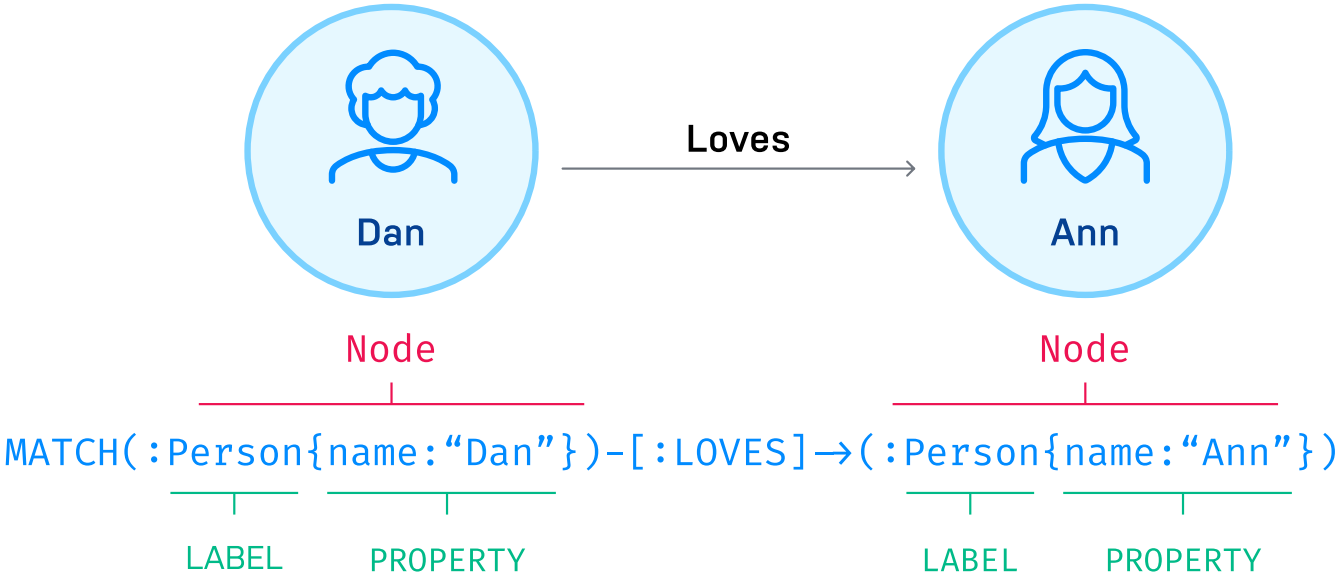


Figure 1. A visual way of matching patterns and relationships using Cypher

How it works

Graph databases are structured through nodes and relationships.

Nodes are entities in the graph which can:

- Be tagged with labels representing their different roles in a domain (e.g., `Person`).
- Hold any number of key-value pairs as properties (e.g., `name`).
- Be indexed and bound by constraints.

Relationships provide named connections between two nodes (e.g., *Person* - `LOVES` - *Person*) and they:

- Must always have a start node, an end node, and exactly one type.
- Must have a direction.
- Can have properties, like nodes.
- Nodes can have multiple relationships of various types without sacrificing performance.

In summary, nodes and relationships are an efficient and flexible way to store data since they allow you to:

- Create traversals in big graphs for both depth and breadth.

- Scale-up your database to billions of nodes.
- Design flexible property graph data models that can adapt over time.

Why use a graph database

Projects often deal with large amounts of complex data and graph databases can be a powerful tool.

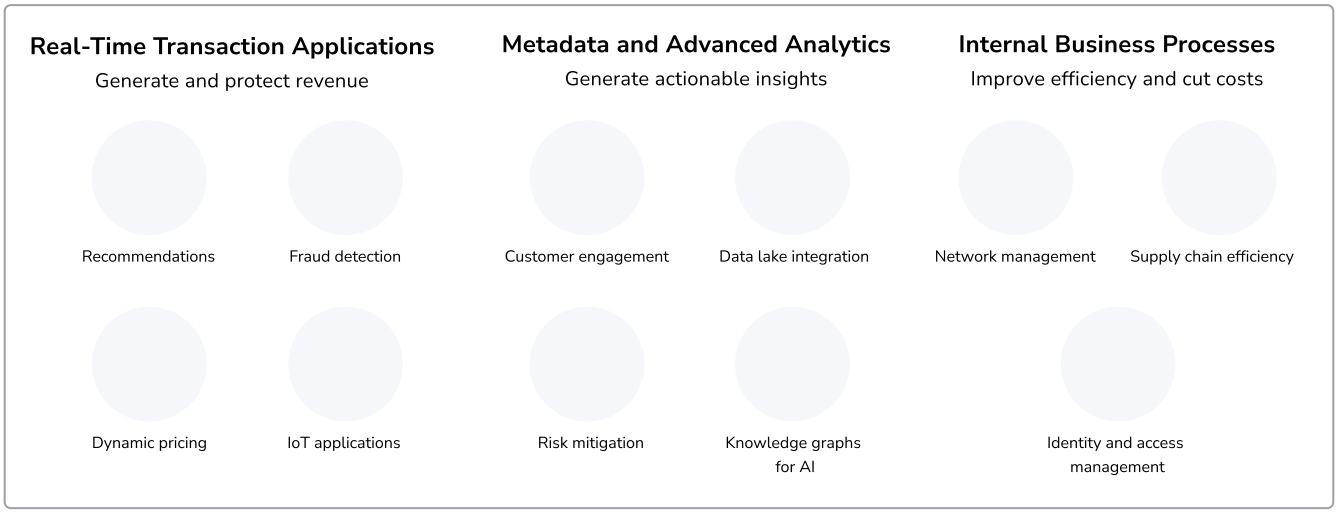
There are other ways to store data as objects and connections, such as relational databases, for example. However, relational databases use computing-wise expensive `JOIN` operations or cross-looks, which are often tied to a rigid data model.

Graph databases do not use JOINs. Rather, relationships are stored natively alongside the data elements (nodes) in a more flexible format, which allows the optimization of data traversing and millions of connections to be accessed per second.

Moreover, many daily challenges and tasks can be viewed from a graph perspective as it allows you to:

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

How to use



Whether it's a social or a road network, all networks can be structured as an interconnected graph of relationships. Many times, a project's questions and challenges revolve around the relationship between elements, not the elements themselves (e.g. how to get from A to B, instead of what A is and what B is). For this reason, graphs can be applied to many areas of society and to a wide variety of projects.

Neo4j is frequently used today by startups, educational institutions, and large enterprises in a variety of sectors including financial services, government, energy, technology, retail, and manufacturing. Graphs have been successful in helping them on the development of innovative new technology, business management, insight and revenue regenration, as well as overall improvements in efficiency.

You can find more information about use cases on Neo4j's main website.

Keep learning

If you want to get a better and deeper understanding of graph databases, you can read more about Graph database concepts, or enroll to the GraphAcademy course on Neo4j Fundamentals.

Glossary

label

Marks a node as a member of a named and indexed subset. A node may be assigned zero or more labels.

labels

A label marks a node as a member of a named and indexed subset. A node may be assigned zero or more labels.

node

A node represents an entity or discrete object in your graph data model. Nodes can be connected by relationships, hold data in properties, and are classified by labels.

nodes

A node represents an entity or discrete object in your graph data model. Nodes can be connected by relationships, hold data in properties, and are classified by labels.

relationship

A relationship represents a connection between nodes in your graph data model. Relationships connect a source node to a target node, hold data in properties, and are classified by type.

relationships

A relationship represents a connection between nodes in your graph data model. Relationships connect a source node to a target node, hold data in properties, and are classified by type.

property

Properties are key-value pairs that are used for storing data on nodes and relationships.

properties

Properties are key-value pairs that are used for storing data on nodes and relationships.

cluster

A Neo4j DBMS that spans multiple servers working together to increase fault tolerance and/or read scalability. Databases on a cluster may be configured to replicate across servers in the cluster thus achieving read scalability or high availability.

clusters

A Neo4j DBMS that spans multiple servers working together to increase fault tolerance and/or read scalability. Databases on a cluster may be configured to replicate across servers in the cluster thus achieving read scalability or high availability.

graph

A logical representation of a set of nodes where some pairs are connected by relationships.

graphs

A logical representation of a set of nodes where some pairs are connected by relationships.

schema

The prescribed property existence and datatypes for nodes and relationships.

schemas

The prescribed property existence and datatypes for nodes and relationships.

[[database schema]]database schema

The prescribed property existence and datatypes for nodes and relationships.

indexes

Data structure that improves read performance of a database. Read more about supported categories of indexes.

indexed

Data structure that improves read performance of a database. Read more about supported categories of indexes.

constraints

Constraints are sets of data modeling rules that ensure the data is consistent and reliable. See what constraints are available in Cypher.

data model

A data model defines how information is organized in a database. A good data model will make querying and understanding your data easier. In Neo4j, the data models have a graph structure.

data models

A data model defines how information is organized in a database. A good data model will make querying and understanding your data easier. In Neo4j, the data models have a graph structure.

Graph database concepts

Introduction

The guide covers graph database fundamentals.

Neo4j uses a *property graph* database model. A graph data structure consists of **nodes** (discrete objects) that can be connected by **relationships**. Below is the image of a graph with three nodes (the circles) and three relationships (the arrows).

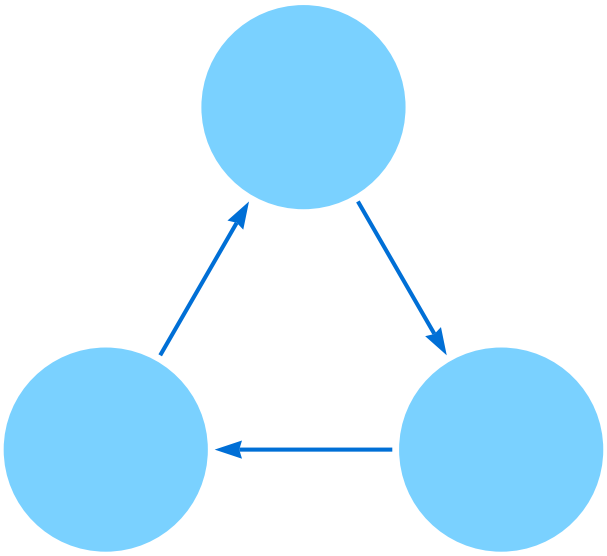


Figure 2. Concept of a graph structure

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** describe a connection between a *source node* and a *target node*.
- **Relationships** always have 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.

Note:

In mathematics, graph theory is the study of graphs.

In graph theory:

- Nodes are also referred to as vertices or points.
- Relationships are also referred to as edges, links, or lines.

Example graph

The example graph shown below introduces the basic concepts of the property graph:

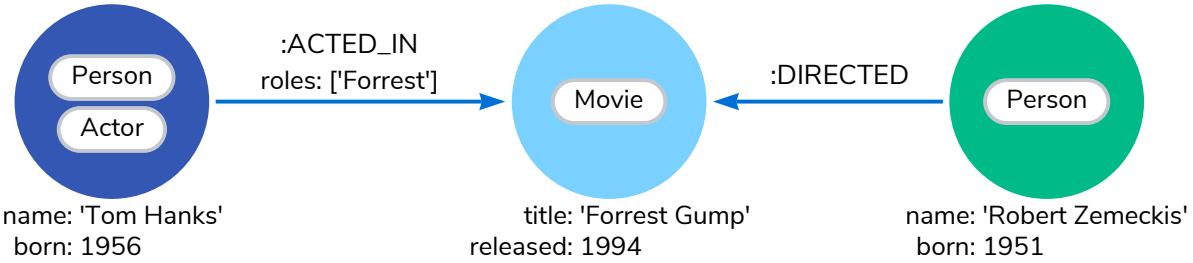


Figure 3. Example graph

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', released: 1994})<-[: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.

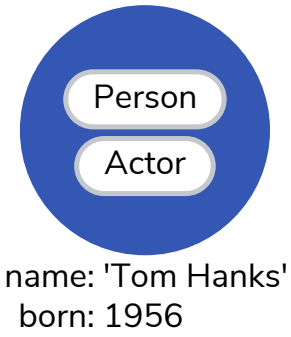


Figure 4. Node

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.

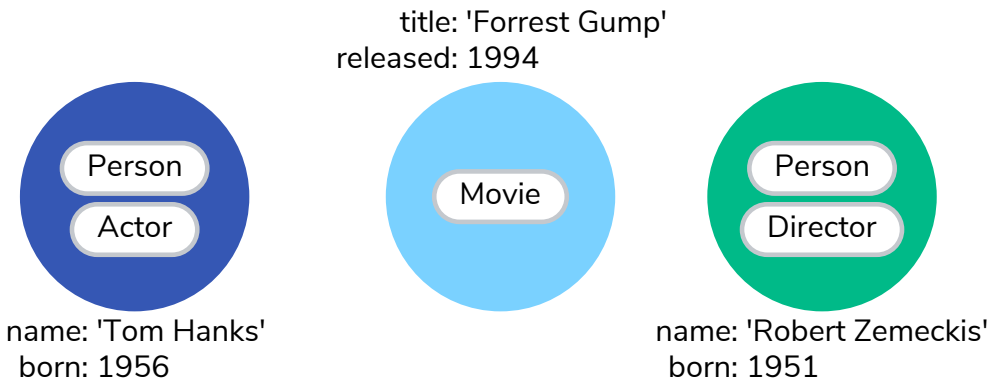


Figure 5. Multiple labels

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.

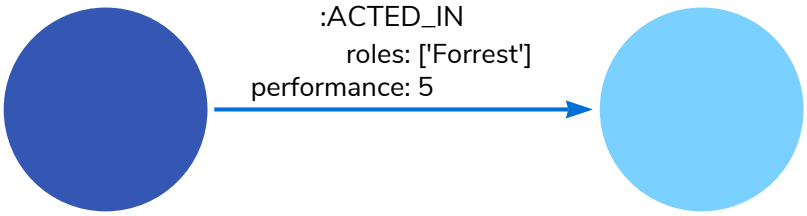


Figure 6. Relationship

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}]->()
```

> **Note:**
>
> 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:

Figure 7. 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.

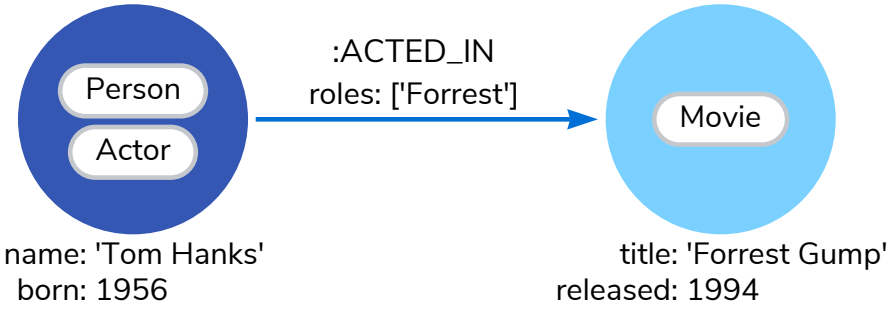


Figure 8. Relationship type

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 1. 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 2. 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 3. 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]`.

Tip:

For a thorough description of the available data types, refer to the Cypher manual → Values and types.

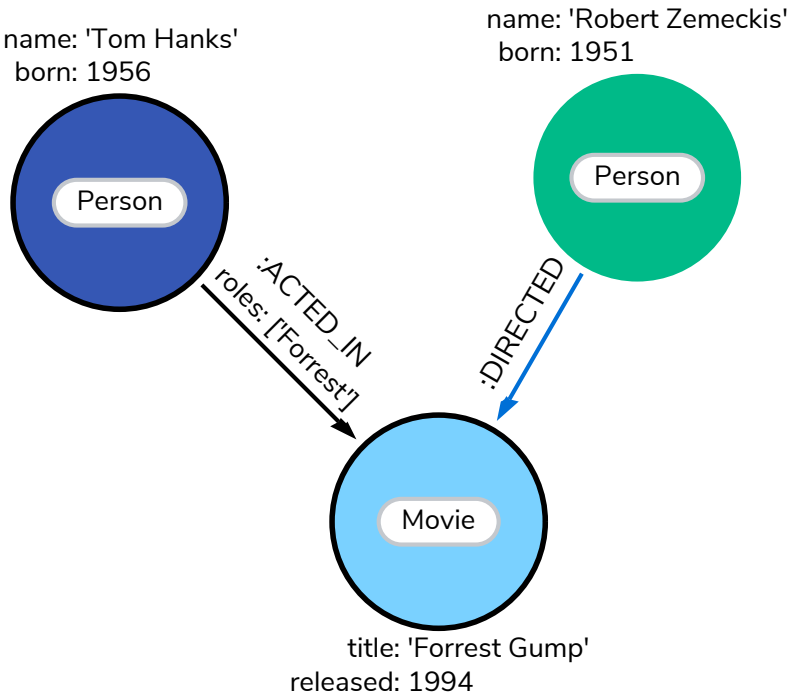
Traversals and paths

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?".

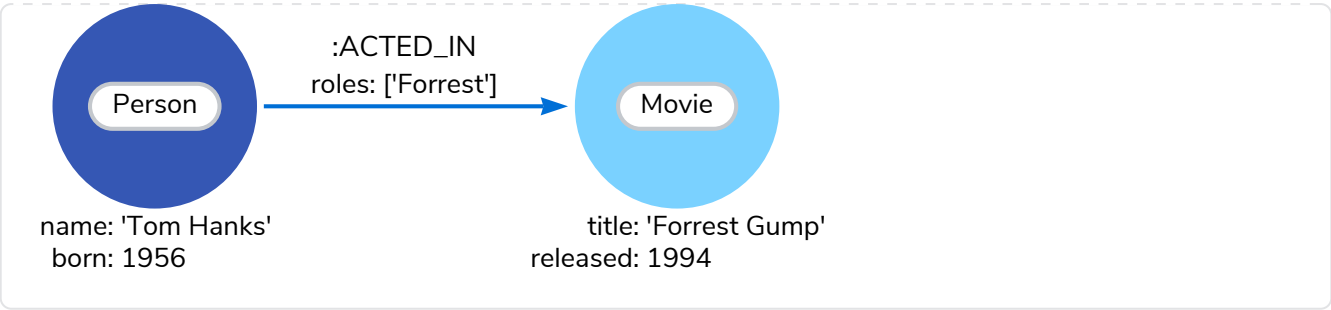
Traversing a graph means visiting nodes by following relationships according to some rules. In most cases only a subset of the graph is visited.

Example 4. 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 the `Movie` node `Forrest Gump` as the result (see the black 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.

A path containing only a single node has the length of `0`.

Figure 9. Path of length zero

A path containing one relationship has the length of `1`.

Figure 10. Path of length one

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 1. Naming conventions

Graph entity	Recommended style	Example
Node label	PascalCase	`:VehicleOwner` rather than `:vehicle_owner`
Relationship type	Screaming snake case	`:OWNS_VEHICLE` rather than `:ownsVehicle`
Property	camelCase	`firstName` rather than `first_name`

For the precise naming rules, refer to the Cypher manual → Naming rules and recommendations.

Comparing relational to graph database

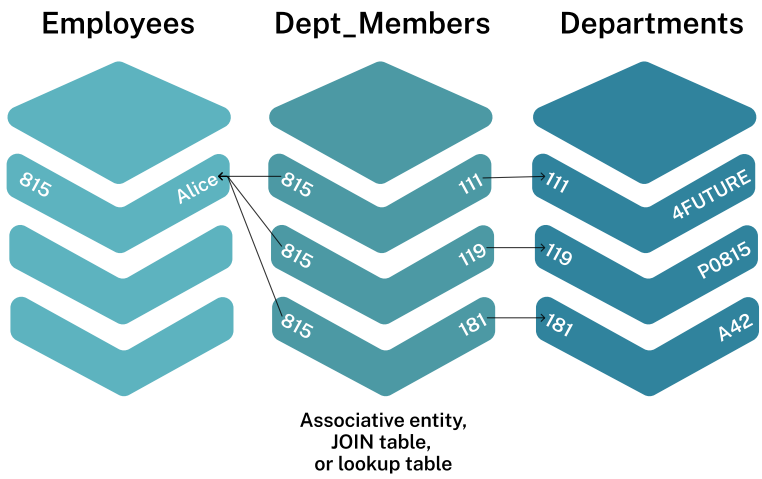
This page explores the conceptual differences between relational and graph database structures and data models. For a comparison between query languages, see Comparing Cypher with SQL.

Relational database overview

Relational databases store highly-structured data in tables with predetermined columns and rows of specific 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. `JOIN` s 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 diagram shows the concept of connecting an `Employee` (from the `Employee` table) to a `Department` (in the `Departments` table) by creating a `Dpt_Members` join table that contains the ID of the employee in one column and the ID of the associated department in another column.

This structure makes understanding the connections cumbersome, because you must know the `Employee` and the `Department` ID values (performing additional lookups to find them) in order to know which employee connects to which department.

Additionally, these types of costly `JOIN` operations are often addressed by denormalizing the data to reduce the number of `JOIN` s necessary, therefore breaking the data integrity of a relational database. Graph databases offer other ways to connect data.

Translating relational knowledge to graphs

Unlike other database management systems, relationships are of equal importance to the data itself in a graph data model. This means you 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 building models that map closely to a problem domain. With Cypher's equivalent of a `JOIN` operation, the graph database can directly access the connected nodes and eliminate the need for expensive search-and-match computations.

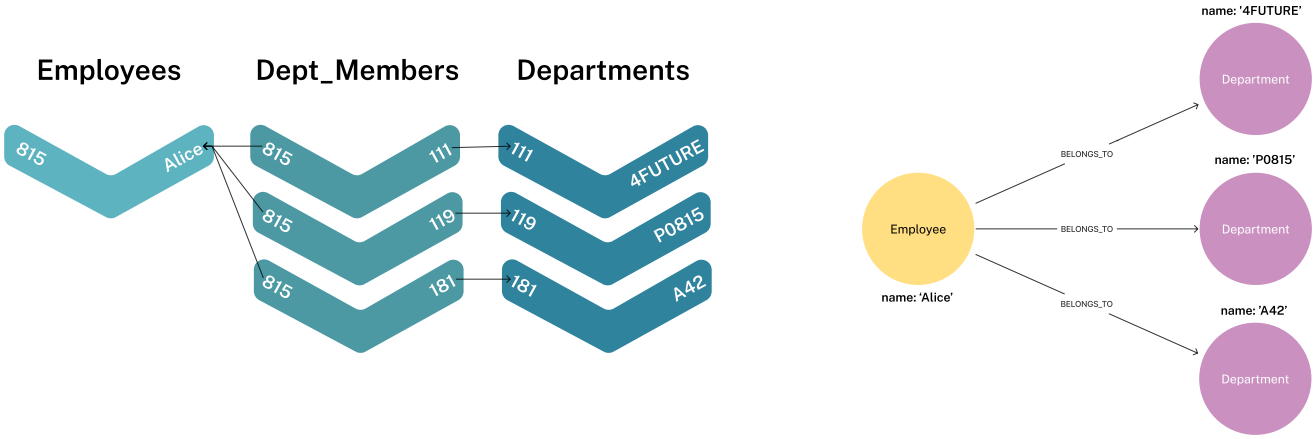
This ability to pre-materialize relationships into the database structure allows Neo4j to provide improved performance compared to others, especially for join-heavy queries.

Video: https://www.youtube.com/watch?v=o_6C27I5yeA

Data model differences

The data models for relational and graph databases are vastly different, as a result of the structural differences previously described. The graph model needs to consider access requirements, expected queries and performance, as well as business logic.

For example, if you want to know which departments Alice belongs to, this is how a relational and a graph databases structure the same data:



In the relational example, on the left, you need to:

1. Search the `Employees` table (potentially with thousands of rows) to find the user Alice and her ID of 815.
2. Search the `Dept_Members` table to locate all the rows that reference Alice's ID of 815.
3. Once the 3 relevant rows are found, you go for the `Departments` table to search for the actual values of the department IDs (111, 119, 181).
4. Only now you know that Alice is part of the 4Future, P0815, and A42 departments.

In the graph version, you need to:

1. Search for Alice's `Employee` node.
2. Traverse all of the `BELONGS_TO` relationships from Alice and find the `Department` nodes she is connected to.

If you want to learn how to create a data model, follow the Tutorial: Create a graph data model or see how to adapt an existing project with a relational model to a graph on Modeling: relational to graph.

Data storage and retrieval

SQL is a query language used to query relational databases. Cypher is Neo4j's declarative query language built on the basic concepts and clauses of SQL, but with additional functionalities that make working with graph databases more efficient.

For example, when writing an SQL statement with a large number of `JOIN` s, you can quickly lose sight of what the query actually does, since there is a lot of technical noise in SQL syntax. In Cypher, the syntax remains concise and focused on domain components and their connections, thus expressing the pattern to find or create data more visually and clearly.

Other clauses outside of the basic pattern matching still look very similar to SQL, as Cypher was built on the predecessor language's foundation. You can see the similarities and differences Comparing Cypher with SQL.

Keep learning

- Import: RDBMS to graph → Learn how to import data from a relational database to a graph.

- Modeling: relational to graph → Find more comparisons between relational and graph data modeling.
- The Definitive Guide to Graph Databases for the RDBMS Developer → Download the free e-book.

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.

Video: Evolution of DBs

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.

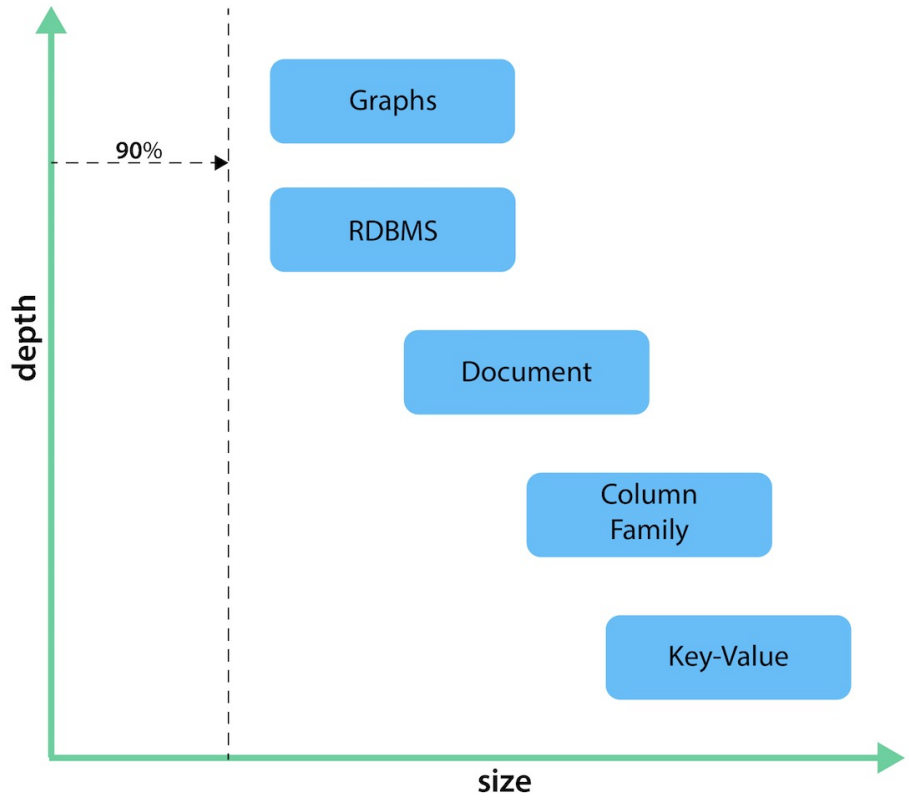
Let's compare the graph data model to other NoSQL models.

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.

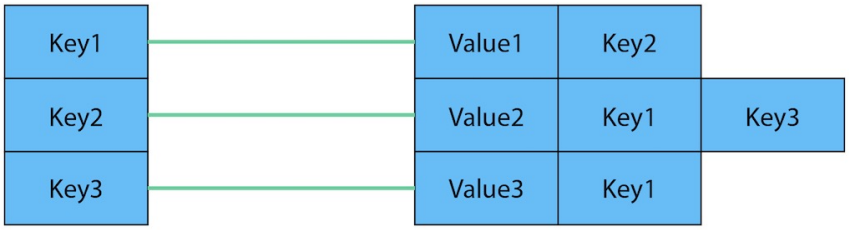


Figure 11. Key-Value Model (click to zoom)

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).

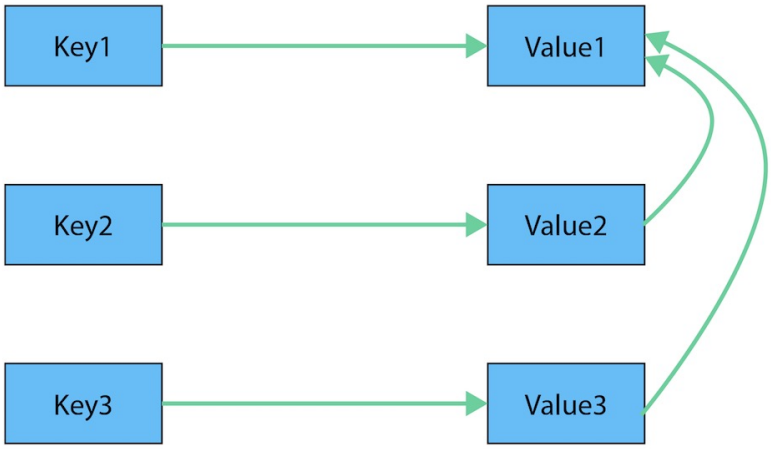


Figure 12. Key-Value as Graph (click to zoom)

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.

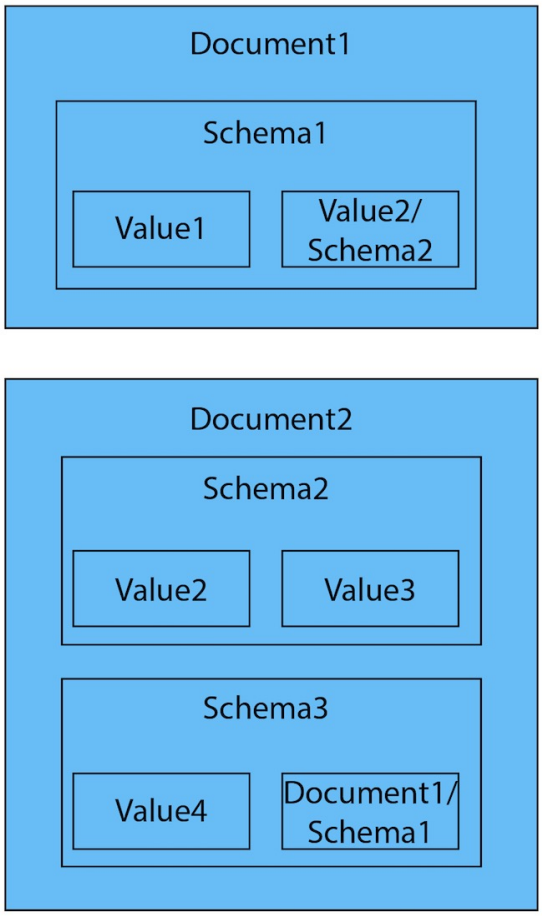


Figure 13. Document Model (click to zoom)

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.

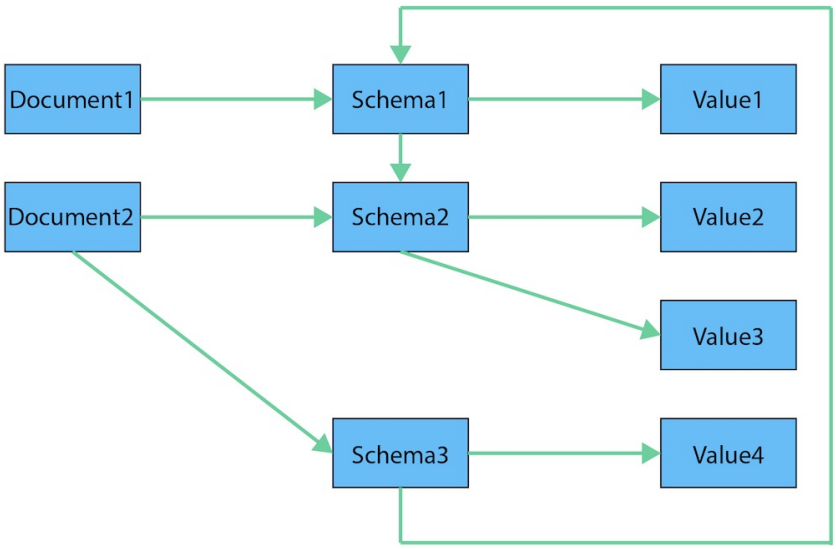


Figure 14. Document as Graph (click to zoom)

Resources

- DZone: NoSQL Database Types
- Blog post: Tour of Aggregate Stores

What is Cypher

> **Important:**
>
> This page covers the basics of Cypher. For the complete documentation, refer to the Cypher Manual.

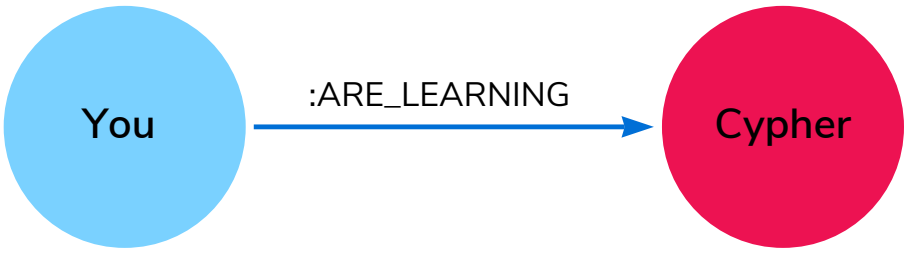


Figure 15. A visual representation of a Cypher query

Cypher is Neo4j's declarative and GQL conformant query language. Available as open source via The openCypher project, Cypher is similar to SQL, but optimized for graphs.

Intuitive and close to natural language, Cypher provides a visual way of matching patterns and relationships by having its own design based on ASCII-art type of syntax:

```
(:nodes)-[:ARE_CONNECTED_TO]->(:otherNodes)
```

Round brackets are used to represent `(:Nodes)`, and square brackets `-[]→` to represent a relationship between the `(:Nodes)`. With this query syntax, you can perform create, read, update, or delete (CRUD) operations on your graph.

> **Tip:**
>
> To try querying with Cypher, get a free Aura instance, no installation required. Use the graduation cap icon on the top right section to access the interactive guides. The "Query fundamentals" gives you a hands-on introduction to Cypher.

How does Cypher work?

The graph is composed of nodes and relationships, which may also have assigned properties. With nodes and relationships, you can build a graph that can express both simple and complex patterns.

Pattern recognition is a key fundamental cognitive process. With Cypher, you can use pattern matching, which in turn makes the learning process more intuitive.

Cypher syntax

What is Cypher?

Cypher's constructs are close to natural language and the syntax is designed to visually look like a graph.

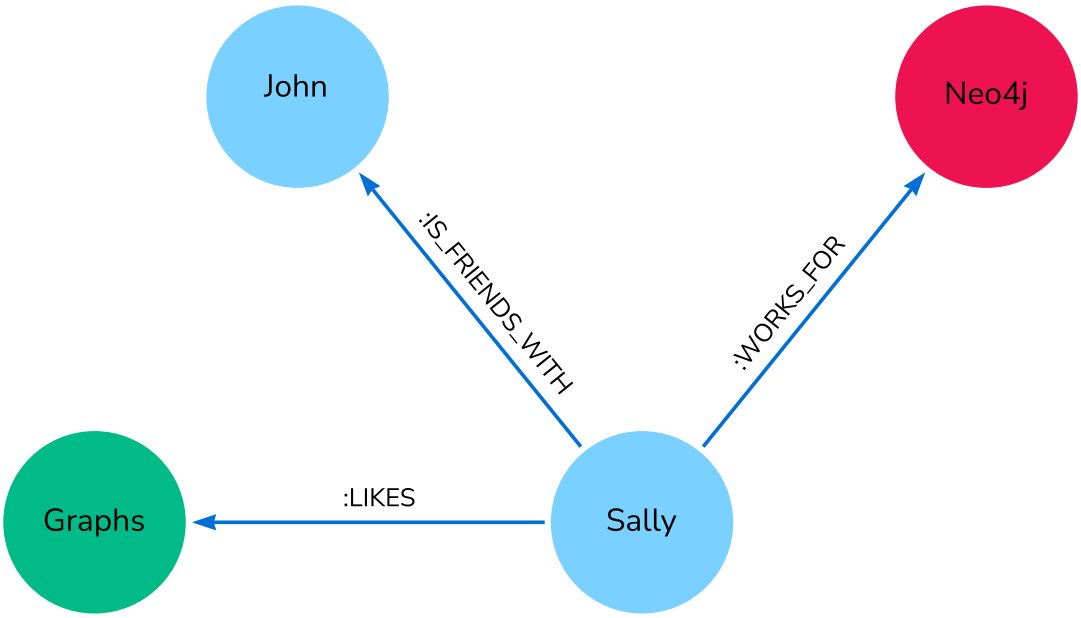


Figure 16. A graph example involving four nodes and three relationships.

If you want to represent the data in this graph in English, it would read as something like: *"Sally likes Graphs. Sally is friends with John. Sally works for Neo4j."*

Now, if you were to write this same information in Cypher, then it would look like this:

```
(:Sally)-[:LIKES]->(:Graphs)
(:Sally)-[:IS_FRIENDS_WITH]->(:John)
(:Sally)-[:WORKS_FOR]->(:Neo4j)
```

With this query, you turn the information into nodes and relationships, which are the core elements of Cypher.

Nodes

The main components in a graph are nodes and relationships. Nodes are often used to represent nouns or objects in your data model. In the previous example, `Sally`, `John`, `Graphs`, and `Neo4j` are the nodes:

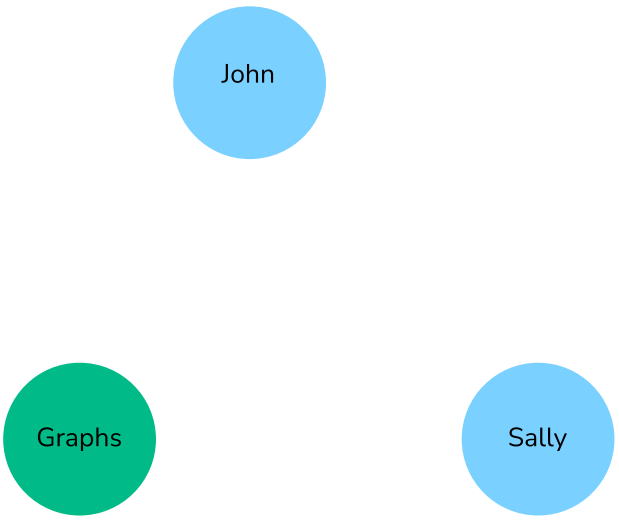


Figure 17. A visual representation of nodes.

As mentioned previously, nodes are represented as round brackets `(node)` in Cypher. The parentheses are

a representation of the circles that compose the nodes in the visualization.

Node labels

Nodes can be grouped together through a label, which works like a tag and allows you to specify certain types of entities in your queries. Labels help Cypher distinguish between nodes and optimize execution.

In the example, both `Sally` and `John` are persons, so they get a `Person` label, `Graphs` gets a `Technology` label, and `Neo4j` is a `Company`:

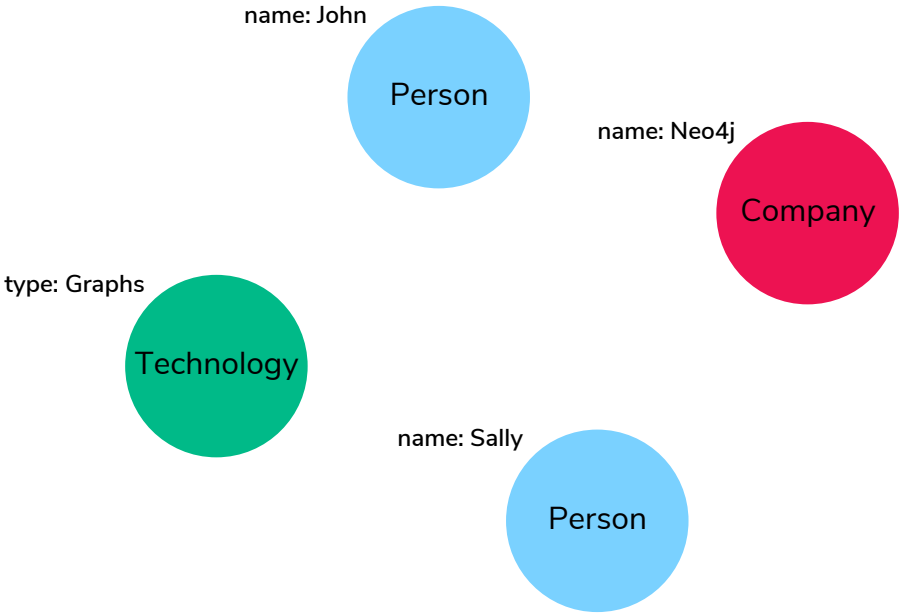


Figure 18. Nodes grouped by labels. Note that `Sally`, `John`, `Graphs`, and `Neo4j` are now properties instead.

In a relational database context, this would be the same as using SQL to refer to a particular row in a table. The same way you can use SQL to query a person's information from a `Person` table, you can also use the `Person` label for that information in Cypher.

> **Caution:**
>
> If you do not specify a label for Cypher to filter out non-matching node categories, the query will check all of the nodes in the database. This can affect performance in very large graphs.

Node variables

If part of your query matches nodes that you need to reference in a later part of your query (i.e. in a subclause), you can use **node variables**.

Variables can be single letters or words, and should be written in lower-case. For example, if you want to bind all nodes labeled `Person` to the variable `p`, you write `(p:Person)`. If you want to use a full word as a variable, `(person:Person)` works exactly the same.

Retrieve all Person nodes

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

Relationships

In a graph database, both nodes and relationships are first-class citizens and they have equal value. In a relational database, relationships are only implied via foreign keys and join tables.

In Cypher, relationships are represented as square brackets with an optional arrow to indicate the direction (e.g. `(Node1)-[]→(Node2)`).

In the example, the arrows connecting the nodes represent the relationship between the nodes:

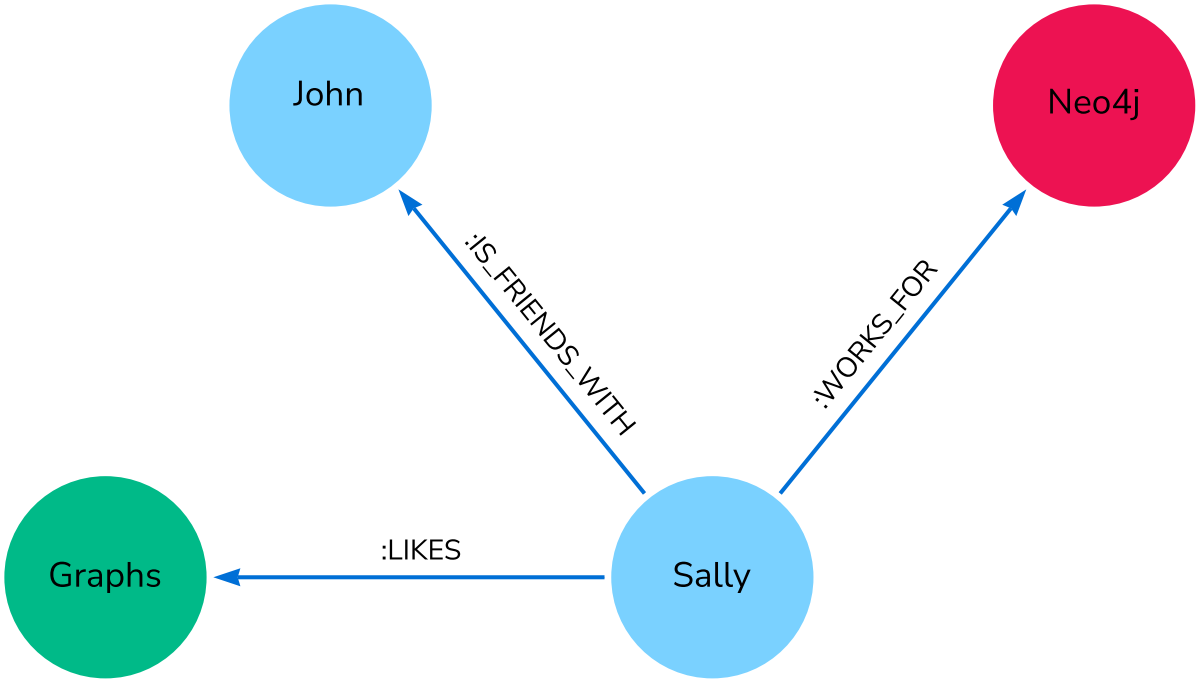


Figure 19. Graph featuring nodes and relationships.

Relationship directions

Relationships **always** have a direction which is indicated by an arrow.

They can go from left to right:

```
(p:Person)-[:LIKES]->(t:Technology)
```

From right to left:

```
(p:Person)<-[:LIKES]-(t:Technology)
```

Or be undirected (where the direction is **not** specified):

```
MATCH (p:Person)-[:LIKES]-(t:Technology)
```

Undirected relationships

An undirected relationship does not mean that it doesn't have a direction, but that it can be traversed in

either direction. While you can't **create** relationships without a direction, you can **query** them undirected (in the example, using the `MATCH` clause).

Since Cypher won't return anything if you write a query with the wrong direction, you can use undirected relationships in queries when you don't know the direction. This way, Cypher will retrieve **all** nodes connected by the specified relationship type, regardless of direction.

Note:

Because undirected relationships in queries are traversed twice (once for each direction), the same pattern will be returned twice. This may impact the performance of the query.

Relationship types

Relationship types categorize and add meaning to a relationship, similar to how labels group nodes together. It is considered best practice to use verbs or derivatives for the relationship type. The type describes how the nodes relate to each other. This way, Cypher is almost like natural language, where nodes are the subjects and objects (nouns), and the relationships (verbs) are the predicates that relate them.

In the previous example, the relationship types are:

- `[:LIKES]` - communicates that Sally (a node) *likes* graphs (another node).
- `[:IS_FRIENDS_WITH]` - communicates that Sally *is friends with* John.
- `[:WORKS_FOR]` - communicates that Sally *works for* Neo4j.

Important:

Remember to always put a colon in front of a relationship type. If you write `(Person)-[LIKES]→(Technology)`, `[LIKES]` will represent a relationship **variable**, not a relationship **type**. In this case, since no relationship type is declared, Cypher's `RETURN` clause will search for all types of relationships in order to retrieve a result to your query.

Relationship variables

Variables can be used for relationships in the same way as for nodes. Once you specify a variable, you can use it later in the query to reference the relationship.

Take this example:

```
MATCH (p:Person)-[r:LIKES]->(t:Technology)
RETURN p,r,t
```

This query specifies variables for both the node labels (`p` for `Person` and `t` for `Technology`) and the relationship type (`r` for `:LIKES`). In the return clause, you can then use the variables (i.e. `p`, `r`, and `t`) to return the bound entities.

This would be your result:

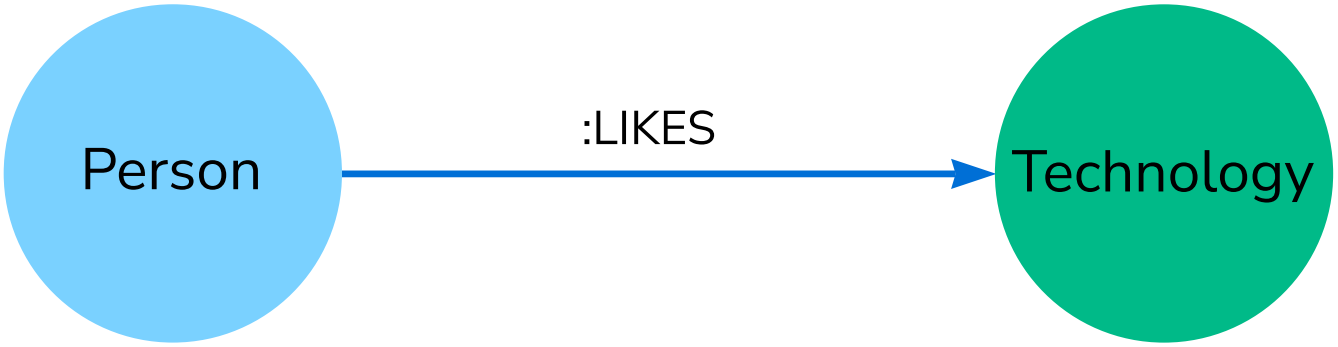


Figure 20. Result for the example query using node and relationship variables.

Table 2. Result

p	r	t
`(:Person)`	`[:LIKES]`	`(:Technology)`
Rows: 1		

Properties

Properties are used to store additional information and can be added both to nodes and relationships and be of a variety of data types. For a full list of values and types, see Cypher manual → Values and types.

In the following example, `sally` and `john` are variables for `Person` nodes which contain a `name` property with the **property values** "Sally" and "John":

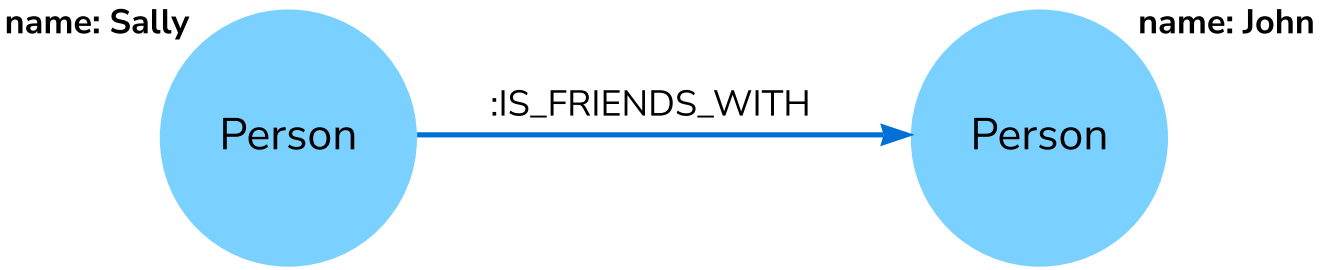


Figure 21. Graph example with node and relationship properties.

To add this info to the graph, you can use the following query:

```
CREATE (sally:Person {name:'Sally'})-[r:IS_FRIENDS_WITH]->(john:Person {name:'John'})
RETURN sally, r, john
```

Properties are enclosed by curly brackets (`{}`), the key is followed by a colon, and the value is enclosed by single or double quotation marks.

In case you have already added Sally and John as node labels, but want to change them into node properties, you need to refactor your graph. Refactoring is a strategy in data modeling that you can learn more about in this tutorial.

Patterns in Cypher

Graph pattern matching sits at the very core of Cypher. It is the mechanism used to navigate, describe, and extract data from a graph by applying a declarative pattern.

Consider this example:

```
(sally:Person {name:'Sally'})-[l:LIKES]->(g:Technology {type: "Graphs"})
```

This bit of Cypher represents a pattern. It expresses that a `Person` node with *Sally* as its `name` property has a `LIKES` relationship to the `Technology` node with *Graphs* as its `type` property.

You can use this pattern in different queries to the database by adding a keyword to make it a **clause**.

For example, you can add this information to the database with the `CREATE` clause:

```
CREATE (sally:Person {name: "Sally"})-[r:LIKES]->(t:Technology {type: "Graphs"})
```

And once this data is written to the database, you can retrieve it with this pattern:

```
MATCH (sally:Person {name: "Sally"})-[r:LIKES]->(t:Technology {type: "Graphs"})
RETURN sally,r,t
```

Pattern variables

In the same way as nodes and relationships, you can also use variables for patterns. Considering the previous example, you can turn the whole pattern (`(Sally)-[:LIKES]→(Technology)`) into a variable (`p`):

```
MATCH p = (sally:Person {name: "Sally"})-[r:LIKES]->(t:Technology {type: "Graphs"})
RETURN p
```

For more information, refer to Cypher manual → Patterns → Syntax and Semantics.

Keep learning

If you want to learn more about writing Cypher queries, you can take the tutorial on how to Get started with Cypher. In the Cypher manual, you can find more information on:

- How to write basic queries and what clauses you can use to read data from the database.
- How patterns work and how you can use them to navigate, describe and extract data from a graph.
- What values and types, and functions are available in Cypher.

From SQL to Cypher

In case you have a background in SQL and are new to graph databases, these are some resources for more information on the key differences and the transition to graphs:

- Key differences between Cypher and SQL
- Transition from relational to graph database

- Reference: Comparing Cypher with SQL
- How-to: Import from RDBMS into graph
- Tutorial: Import data from a relational database into Neo4j
- How-to: Model data from relational to graph

From NoSQL to Graphs

If you are familiar with NoSQL ("Not only SQL") system, you can also learn more on how to make the transition to a graph database.

GraphAcademy

With the Cypher Fundamentals course, you can learn Cypher in 60 minutes and practice using a sandbox.

Other resources

For more suggestions on how to expand your knowledge about Cypher, refer to Resources.

Glossary

ROOT:glossary.adoc

Get started with Cypher

This tutorial shows the basic concepts of Cypher, Neo4j's query language, including how to create and query graphs. You will create a graph, find nodes, query patterns, and solve questions about the graph.

Create the Movie Graph

After you create a free Aura instance, use the "Connect" button and select "Query". In the Cypher editor, copy and paste the following Cypher and execute the query:

```
CREATE CONSTRAINT movie_title IF NOT EXISTS FOR (m:Movie) REQUIRE m.title IS UNIQUE;
CREATE CONSTRAINT person_name IF NOT EXISTS FOR (p:Person) REQUIRE p.name IS UNIQUE;

MERGE (TheMatrix:Movie {title:'The Matrix'}) ON CREATE SET TheMatrix.released=1999, TheMatrix.tagline
='Welcome to the Real World'

MERGE (Keanu:Person {name:'Keanu Reeves'}) ON CREATE SET Keanu.born=1964
MERGE (Carrie:Person {name:'Carrie-Anne Moss'}) ON CREATE SET Carrie.born=1967
MERGE (Laurence:Person {name:'Laurence Fishburne'}) ON CREATE SET Laurence.born=1961
MERGE (Hugo:Person {name:'Hugo Weaving'}) ON CREATE SET Hugo.born=1960
MERGE (LillyW:Person {name:'Lilly Wachowski'}) ON CREATE SET LillyW.born=1967
MERGE (LanaW:Person {name:'Lana Wachowski'}) ON CREATE SET LanaW.born=1965
MERGE (JoelS:Person {name:'Joel Silver'}) ON CREATE SET JoelS.born=1952

MERGE (Keanu)-[:ACTED_IN {roles:['Neo']}]->(TheMatrix)
MERGE (Carrie)-[:ACTED_IN {roles:['Trinity']}]->(TheMatrix)
MERGE (Laurence)-[:ACTED_IN {roles:['Morpheus']}]->(TheMatrix)
MERGE (Hugo)-[:ACTED_IN {roles:['Agent Smith']}]->(TheMatrix)
MERGE (LillyW)-[:DIRECTED]->(TheMatrix)
MERGE (LanaW)-[:DIRECTED]->(TheMatrix)
MERGE (JoelS)-[:PRODUCED]->(TheMatrix)

MERGE (Emil:Person {name:'Emil Eifrem'}) ON CREATE SET Emil.born=1978
MERGE (Emil)-[:ACTED_IN {roles:["Emil"]}]->(TheMatrix);


MERGE (TheMatrixReloaded:Movie {title:'The Matrix Reloaded'}) ON CREATE SET TheMatrixReloaded.released
=2003, TheMatrixReloaded.tagline='Free your mind'

MERGE (Keanu:Person {name:'Keanu Reeves'}) ON CREATE SET Keanu.born=1964
MERGE (Carrie:Person {name:'Carrie-Anne Moss'}) ON CREATE SET Carrie.born=1967
MERGE (Laurence:Person {name:'Laurence Fishburne'}) ON CREATE SET Laurence.born=1961
MERGE (Hugo:Person {name:'Hugo Weaving'}) ON CREATE SET Hugo.born=1960
MERGE (LillyW:Person {name:'Lilly Wachowski'}) ON CREATE SET LillyW.born=1967
MERGE (LanaW:Person {name:'Lana Wachowski'}) ON CREATE SET LanaW.born=1965
MERGE (JoelS:Person {name:'Joel Silver'}) ON CREATE SET JoelS.born=1952

MERGE (Keanu)-[:ACTED_IN {roles:['Neo']}]->(TheMatrixReloaded)
MERGE (Carrie)-[:ACTED_IN {roles:['Trinity']}]->(TheMatrixReloaded)
MERGE (Laurence)-[:ACTED_IN {roles:['Morpheus']}]->(TheMatrixReloaded)
MERGE (Hugo)-[:ACTED_IN {roles:['Agent Smith']}]->(TheMatrixReloaded)
MERGE (LillyW)-[:DIRECTED]->(TheMatrixReloaded)
MERGE (LanaW)-[:DIRECTED]->(TheMatrixReloaded)
MERGE (JoelS)-[:PRODUCED]->(TheMatrixReloaded);


MERGE (TheMatrixRevolutions:Movie {title:'The Matrix Revolutions'}) ON CREATE SET
TheMatrixRevolutions.released=2003, TheMatrixRevolutions.tagline='Everything that has a beginning has an
end'

MERGE (Keanu:Person {name:'Keanu Reeves'}) ON CREATE SET Keanu.born=1964
MERGE (Carrie:Person {name:'Carrie-Anne Moss'}) ON CREATE SET Carrie.born=1967
MERGE (Laurence:Person {name:'Laurence Fishburne'}) ON CREATE SET Laurence.born=1961
MERGE (Hugo:Person {name:'Hugo Weaving'}) ON CREATE SET Hugo.born=1960
MERGE (LillyW:Person {name:'Lilly Wachowski'}) ON CREATE SET LillyW.born=1967
MERGE (LanaW:Person {name:'Lana Wachowski'}) ON CREATE SET LanaW.born=1965
MERGE (JoelS:Person {name:'Joel Silver'}) ON CREATE SET JoelS.born=1952

MERGE (Keanu)-[:ACTED_IN {roles:['Neo']}]->(TheMatrixRevolutions)
MERGE (Carrie)-[:ACTED_IN {roles:['Trinity']}]->(TheMatrixRevolutions)
```

```
MERGE (Laurence)-[:ACTED_IN {roles:['Morpheus']}]->(TheMatrixRevolutions)
MERGE (Hugo)-[:ACTED_IN {roles:['Agent Smith']}]->(TheMatrixRevolutions)
MERGE (LillyW)-[:DIRECTED]->(TheMatrixRevolutions)
MERGE (LanaW)-[:DIRECTED]->(TheMatrixRevolutions)
MERGE (JoelS)-[:PRODUCED]->(TheMatrixRevolutions);


MERGE (TheDevilsAdvocate:Movie {title:"The Devil's Advocate", released:1997, tagline:'Evil has its winning
ways'})

MERGE (Keanu:Person {name:'Keanu Reeves'}) ON CREATE SET Keanu.born=1964
MERGE (Charlize:Person {name:'Charlize Theron'}) ON CREATE SET Charlize.born=1975
MERGE (Al:Person {name:'Al Pacino'}) ON CREATE SET Al.born=1940
MERGE (Taylor:Person {name:'Taylor Hackford'}) ON CREATE SET Taylor.born=1944

MERGE (Keanu)-[:ACTED_IN {roles:['Kevin Lomax']}]->(TheDevilsAdvocate)
MERGE (Charlize)-[:ACTED_IN {roles:['Mary Ann Lomax']}]->(TheDevilsAdvocate)
MERGE (Al)-[:ACTED_IN {roles:['John Milton']}]->(TheDevilsAdvocate)
MERGE (Taylor)-[:DIRECTED]->(TheDevilsAdvocate);


MERGE (AFewGoodMen:Movie {title:'A Few Good Men'}) ON CREATE SET AFewGoodMen.released=1992,
AFewGoodMen.tagline='In the heart of the nation\'s capital, in a courthouse of the U.S. government, one
man will stop at nothing to keep his honor, and one will stop at nothing to find the truth.'

MERGE (TomC:Person {name:'Tom Cruise'}) ON CREATE SET TomC.born=1962
MERGE (JackN:Person {name:'Jack Nicholson'}) ON CREATE SET JackN.born=1937
MERGE (DemiM:Person {name:'Demi Moore'}) ON CREATE SET DemiM.born=1962
MERGE (KevinB:Person {name:'Kevin Bacon'}) ON CREATE SET KevinB.born=1958
MERGE (KieferS:Person {name:'Kiefer Sutherland'}) ON CREATE SET KieferS.born=1966
MERGE (NoahW:Person {name:'Noah Wyle'}) ON CREATE SET NoahW.born=1971
MERGE (CubaG:Person {name:'Cuba Gooding Jr.'}) ON CREATE SET CubaG.born=1968
MERGE (KevinP:Person {name:'Kevin Pollak'}) ON CREATE SET KevinP.born=1957
MERGE (JTW:Person {name:'J.T. Walsh'}) ON CREATE SET JTW.born=1943
MERGE (JamesM:Person {name:'James Marshall'}) ON CREATE SET JamesM.born=1967
MERGE (ChristopherG:Person {name:'Christopher Guest'}) ON CREATE SET ChristopherG.born=1948
MERGE (RobR:Person {name:'Rob Reiner'}) ON CREATE SET RobR.born=1947
MERGE (AaronS:Person {name:'Aaron Sorkin'}) ON CREATE SET AaronS.born=1961

MERGE (TomC)-[:ACTED_IN {roles:['Lt. Daniel Kaffee']}]->(AFewGoodMen)
MERGE (JackN)-[:ACTED_IN {roles:['Col. Nathan R. Jessup']}]->(AFewGoodMen)
MERGE (DemiM)-[:ACTED_IN {roles:['Lt. Cdr. JoAnne Galloway']}]->(AFewGoodMen)
MERGE (KevinB)-[:ACTED_IN {roles:['Capt. Jack Ross']}]->(AFewGoodMen)
MERGE (KieferS)-[:ACTED_IN {roles:['Lt. Jonathan Kendrick']}]->(AFewGoodMen)
MERGE (NoahW)-[:ACTED_IN {roles:['Cpl. Jeffrey Barnes']}]->(AFewGoodMen)
MERGE (CubaG)-[:ACTED_IN {roles:['Cpl. Carl Hammaker']}]->(AFewGoodMen)
MERGE (KevinP)-[:ACTED_IN {roles:['Lt. Sam Weinberg']}]->(AFewGoodMen)
MERGE (JTW)-[:ACTED_IN {roles:['Lt. Col. Matthew Andrew Markinson']}]->(AFewGoodMen)
MERGE (JamesM)-[:ACTED_IN {roles:['Pfc. Louden Downey']}]->(AFewGoodMen)
MERGE (ChristopherG)-[:ACTED_IN {roles:['Dr. Stone']}]->(AFewGoodMen)
MERGE (AaronS)-[:ACTED_IN {roles:['Man in Bar']}]->(AFewGoodMen)
MERGE (RobR)-[:DIRECTED]->(AFewGoodMen)
MERGE (AaronS)-[:WROTE]->(AFewGoodMen);


MERGE (TopGun:Movie {title:'Top Gun'}) ON CREATE SET TopGun.released=1986, TopGun.tagline='I feel the
need, the need for speed.'

MERGE (TomC:Person {name:'Tom Cruise'}) ON CREATE SET TomC.born=1962
MERGE (KellyM:Person {name:'Kelly McGillis'}) ON CREATE SET KellyM.born=1957
MERGE (ValK:Person {name:'Val Kilmer'}) ON CREATE SET ValK.born=1959
MERGE (AnthonyE:Person {name:'Anthony Edwards'}) ON CREATE SET AnthonyE.born=1962
MERGE (TomS:Person {name:'Tom Skerritt'}) ON CREATE SET TomS.born=1933
MERGE (MegR:Person {name:'Meg Ryan'}) ON CREATE SET MegR.born=1961
MERGE (TonyS:Person {name:'Tony Scott'}) ON CREATE SET TonyS.born=1944
MERGE (JimC:Person {name:'Jim Cash'}) ON CREATE SET JimC.born=1941

MERGE (TomC)-[:ACTED_IN {roles:['Maverick']}]->(TopGun)
MERGE (KellyM)-[:ACTED_IN {roles:['Charlie']}]->(TopGun)
MERGE (ValK)-[:ACTED_IN {roles:['Iceman']}]->(TopGun)
MERGE (AnthonyE)-[:ACTED_IN {roles:['Goose']}]->(TopGun)
MERGE (TomS)-[:ACTED_IN {roles:['Viper']}]->(TopGun)
MERGE (MegR)-[:ACTED_IN {roles:['Carole']}]->(TopGun)
MERGE (TonyS)-[:DIRECTED]->(TopGun)
MERGE (JimC)-[:WROTE]->(TopGun);
```

```
MERGE (JerryMaguire:Movie {title:'Jerry Maguire'}) ON CREATE SET JerryMaguire.released=2000,
JerryMaguire.tagline='The rest of his life begins now.'

MERGE (TomC:Person {name:'Tom Cruise'}) ON CREATE SET TomC.born=1962
MERGE (CubaG:Person {name:'Cuba Gooding Jr.'}) ON CREATE SET CubaG.born=1968
MERGE (ReneeZ:Person {name:'Renee Zellweger'}) ON CREATE SET ReneeZ.born=1969
MERGE (KellyP:Person {name:'Kelly Preston'}) ON CREATE SET KellyP.born=1962
MERGE (JerryO:Person {name:'Jerry O\'Connell'}) ON CREATE SET JerryO.born=1974
MERGE (JayM:Person {name:'Jay Mohr'}) ON CREATE SET JayM.born=1970
MERGE (BonnieH:Person {name:'Bonnie Hunt'}) ON CREATE SET BonnieH.born=1961
MERGE (ReginaK:Person {name:'Regina King'}) ON CREATE SET ReginaK.born=1971
MERGE (JonathanL:Person {name:'Jonathan Lipnicki'}) ON CREATE SET JonathanL.born=1996
MERGE (CameronC:Person {name:'Cameron Crowe'}) ON CREATE SET CameronC.born=1957

MERGE (TomC)-[:ACTED_IN {roles:['Jerry Maguire']}]->(JerryMaguire)
MERGE (CubaG)-[:ACTED_IN {roles:['Rod Tidwell']}]->(JerryMaguire)
MERGE (ReneeZ)-[:ACTED_IN {roles:['Dorothy Boyd']}]->(JerryMaguire)
MERGE (KellyP)-[:ACTED_IN {roles:['Avery Bishop']}]->(JerryMaguire)
MERGE (JerryO)-[:ACTED_IN {roles:['Frank Cushman']}]->(JerryMaguire)
MERGE (JayM)-[:ACTED_IN {roles:['Bob Sugar']}]->(JerryMaguire)
MERGE (BonnieH)-[:ACTED_IN {roles:['Laurel Boyd']}]->(JerryMaguire)
MERGE (ReginaK)-[:ACTED_IN {roles:['Marcee Tidwell']}]->(JerryMaguire)
MERGE (JonathanL)-[:ACTED_IN {roles:['Ray Boyd']}]->(JerryMaguire)
MERGE (CameronC)-[:DIRECTED]->(JerryMaguire)
MERGE (CameronC)-[:PRODUCED]->(JerryMaguire)
MERGE (CameronC)-[:WROTE]->(JerryMaguire);

MERGE (StandByMe:Movie {title:'Stand By Me'}) ON CREATE SET StandByMe.released=1986, StandByMe.tagline
='For some, it\'s the last real taste of innocence, and the first real taste of life. But for everyone,
it\'s the time that memories are made of.'

MERGE (RiverP:Person {name:'River Phoenix'}) ON CREATE SET RiverP.born=1970
MERGE (CoreyF:Person {name:'Corey Feldman'}) ON CREATE SET CoreyF.born=1971
MERGE (JerryO:Person {name:'Jerry O\'Connell'}) ON CREATE SET JerryO.born=1974
MERGE (WilW:Person {name:'Wil Wheaton'}) ON CREATE SET WilW.born=1972
MERGE (KieferS:Person {name:'Kiefer Sutherland'}) ON CREATE SET KieferS.born=1966
MERGE (JohnC:Person {name:'John Cusack'}) ON CREATE SET JohnC.born=1966
MERGE (MarshallB:Person {name:'Marshall Bell'}) ON CREATE SET MarshallB.born=1942
MERGE (RobR:Person {name:'Rob Reiner'}) ON CREATE SET RobR.born=1947

MERGE (WilW)-[:ACTED_IN {roles:['Gordie Lachance']}]->(StandByMe)
MERGE (RiverP)-[:ACTED_IN {roles:['Chris Chambers']}]->(StandByMe)
MERGE (JerryO)-[:ACTED_IN {roles:['Vern Tessio']}]->(StandByMe)
MERGE (CoreyF)-[:ACTED_IN {roles:['Teddy Duchamp']}]->(StandByMe)
MERGE (JohnC)-[:ACTED_IN {roles:['Denny Lachance']}]->(StandByMe)
MERGE (KieferS)-[:ACTED_IN {roles:['Ace Merrill']}]->(StandByMe)
MERGE (MarshallB)-[:ACTED_IN {roles:['Mr. Lachance']}]->(StandByMe)
MERGE (RobR)-[:DIRECTED]->(StandByMe);

MERGE (AsGoodAsItGets:Movie {title:'As Good as It Gets'}) ON CREATE SET AsGoodAsItGets.released=1997,
AsGoodAsItGets.tagline='A comedy from the heart that goes for the throat.'

MERGE (JackN:Person {name:'Jack Nicholson'}) ON CREATE SET JackN.born=1937
MERGE (HelenH:Person {name:'Helen Hunt'}) ON CREATE SET HelenH.born=1963
MERGE (GregK:Person {name:'Greg Kinnear'}) ON CREATE SET GregK.born=1963
MERGE (JamesB:Person {name:'James L. Brooks'}) ON CREATE SET JamesB.born=1940
MERGE (CubaG:Person {name:'Cuba Gooding Jr.'}) ON CREATE SET CubaG.born=1968

MERGE (JackN)-[:ACTED_IN {roles:['Melvin Udall']}]->(AsGoodAsItGets)
MERGE (HelenH)-[:ACTED_IN {roles:['Carol Connelly']}]->(AsGoodAsItGets)
MERGE (GregK)-[:ACTED_IN {roles:['Simon Bishop']}]->(AsGoodAsItGets)
MERGE (CubaG)-[:ACTED_IN {roles:['Frank Sachs']}]->(AsGoodAsItGets)
MERGE (JamesB)-[:DIRECTED]->(AsGoodAsItGets);

MERGE (WhatDreamsMayCome:Movie {title:'What Dreams May Come'}) ON CREATE SET WhatDreamsMayCome.released
=1998, WhatDreamsMayCome.tagline='After life there is more. The end is just the beginning.'

MERGE (AnnabellaS:Person {name:'Annabella Sciorra'}) ON CREATE SET AnnabellaS.born=1960
MERGE (MaxS:Person {name:'Max von Sydow'}) ON CREATE SET MaxS.born=1929
MERGE (WernerH:Person {name:'Werner Herzog'}) ON CREATE SET WernerH.born=1942
MERGE (Robin:Person {name:'Robin Williams'}) ON CREATE SET Robin.born=1951
MERGE (VincentW:Person {name:'Vincent Ward'}) ON CREATE SET VincentW.born=1956
MERGE (CubaG:Person {name:'Cuba Gooding Jr.'}) ON CREATE SET CubaG.born=1968

MERGE (Robin)-[:ACTED_IN {roles:['Chris Nielsen']}]->(WhatDreamsMayCome)
MERGE (CubaG)-[:ACTED_IN {roles:['Albert Lewis']}]->(WhatDreamsMayCome)
MERGE (AnnabellaS)-[:ACTED_IN {roles:['Annie Collins-Nielsen']}]->(WhatDreamsMayCome)
```

```
MERGE (MaxS)-[:ACTED_IN {roles:['The Tracker']}]->(WhatDreamsMayCome)
MERGE (WernerH)-[:ACTED_IN {roles:['The Face']}]->(WhatDreamsMayCome)
MERGE (VincentW)-[:DIRECTED]->(WhatDreamsMayCome);

MERGE (SnowFallingonCedars:Movie {title:'Snow Falling on Cedars'}) ON CREATE SET
SnowFallingonCedars.released=1999, SnowFallingonCedars.tagline='First loves last. Forever.'

MERGE (EthanH:Person {name:'Ethan Hawke'}) ON CREATE SET EthanH.born=1970
MERGE (RickY:Person {name:'Rick Yune'}) ON CREATE SET RickY.born=1971
MERGE (JamesC:Person {name:'James Cromwell'}) ON CREATE SET JamesC.born=1940
MERGE (ScottH:Person {name:'Scott Hicks'}) ON CREATE SET ScottH.born=1953
MERGE (MaxS:Person {name:'Max von Sydow'}) ON CREATE SET MaxS.born=1929

MERGE (EthanH)-[:ACTED_IN {roles:['Ishmael Chambers']}]->(SnowFallingonCedars)
MERGE (RickY)-[:ACTED_IN {roles:['Kazuo Miyamoto']}]->(SnowFallingonCedars)
MERGE (MaxS)-[:ACTED_IN {roles:['Nels Gudmundsson']}]->(SnowFallingonCedars)
MERGE (JamesC)-[:ACTED_IN {roles:['Judge Fielding']}]->(SnowFallingonCedars)
MERGE (ScottH)-[:DIRECTED]->(SnowFallingonCedars);

MERGE (YouveGotMail:Movie {title:'You\'ve Got Mail'}) ON CREATE SET YouveGotMail.released=1998,
YouveGotMail.tagline='At odds in life... in love on-line.'

MERGE (TomH:Person {name:'Tom Hanks'}) ON CREATE SET TomH.born=1956
MERGE (MegR:Person {name:'Meg Ryan'}) ON CREATE SET MegR.born=1961
MERGE (GregK:Person {name:'Greg Kinnear'}) ON CREATE SET GregK.born=1963
MERGE (ParkerP:Person {name:'Parker Posey'}) ON CREATE SET ParkerP.born=1968
MERGE (DaveC:Person {name:'Dave Chappelle'}) ON CREATE SET DaveC.born=1973
MERGE (SteveZ:Person {name:'Steve Zahn'}) ON CREATE SET SteveZ.born=1967
MERGE (NoraE:Person {name:'Nora Ephron'}) ON CREATE SET NoraE.born=1941

MERGE (TomH)-[:ACTED_IN {roles:['Joe Fox']}]->(YouveGotMail)
MERGE (MegR)-[:ACTED_IN {roles:['Kathleen Kelly']}]->(YouveGotMail)
MERGE (GregK)-[:ACTED_IN {roles:['Frank Navasky']}]->(YouveGotMail)
MERGE (ParkerP)-[:ACTED_IN {roles:['Patricia Eden']}]->(YouveGotMail)
MERGE (DaveC)-[:ACTED_IN {roles:['Kevin Jackson']}]->(YouveGotMail)
MERGE (SteveZ)-[:ACTED_IN {roles:['George Pappas']}]->(YouveGotMail)
MERGE (NoraE)-[:DIRECTED]->(YouveGotMail);

MERGE (SleeplessInSeattle:Movie {title:'Sleepless in Seattle'}) ON CREATE SET SleeplessInSeattle.released
=1993, SleeplessInSeattle.tagline='What if someone you never met, someone you never saw, someone you never
knew was the only someone for you?'

MERGE (TomH:Person {name:'Tom Hanks'}) ON CREATE SET TomH.born=1956
MERGE (MegR:Person {name:'Meg Ryan'}) ON CREATE SET MegR.born=1961
MERGE (RitaW:Person {name:'Rita Wilson'}) ON CREATE SET RitaW.born=1956
MERGE (BillPull:Person {name:'Bill Pullman'}) ON CREATE SET BillPull.born=1953
MERGE (VictorG:Person {name:'Victor Garber'}) ON CREATE SET VictorG.born=1949
MERGE (RosieO:Person {name:'Rosie O\'Donnell'}) ON CREATE SET RosieO.born=1962
MERGE (NoraE:Person {name:'Nora Ephron'}) ON CREATE SET NoraE.born=1941

MERGE (TomH)-[:ACTED_IN {roles:['Sam Baldwin']}]->(SleeplessInSeattle)
MERGE (MegR)-[:ACTED_IN {roles:['Annie Reed']}]->(SleeplessInSeattle)
MERGE (RitaW)-[:ACTED_IN {roles:['Suzy']}]->(SleeplessInSeattle)
MERGE (BillPull)-[:ACTED_IN {roles:['Walter']}]->(SleeplessInSeattle)
MERGE (VictorG)-[:ACTED_IN {roles:['Greg']}]->(SleeplessInSeattle)
MERGE (RosieO)-[:ACTED_IN {roles:['Becky']}]->(SleeplessInSeattle)
MERGE (NoraE)-[:DIRECTED]->(SleeplessInSeattle);

MERGE (JoeVersustheVolcano:Movie {title:'Joe Versus the Volcano'}) ON CREATE SET
JoeVersustheVolcano.released=1990, JoeVersustheVolcano.tagline='A story of love, lava and burning desire.'

MERGE (TomH:Person {name:'Tom Hanks'}) ON CREATE SET TomH.born=1956
MERGE (MegR:Person {name:'Meg Ryan'}) ON CREATE SET MegR.born=1961
MERGE (JohnS:Person {name:'John Patrick Stanley'}) ON CREATE SET JohnS.born=1950
MERGE (Nathan:Person {name:'Nathan Lane'}) ON CREATE SET Nathan.born=1956

MERGE (TomH)-[:ACTED_IN {roles:['Joe Banks']}]->(JoeVersustheVolcano)
MERGE (MegR)-[:ACTED_IN {roles:['DeDe', 'Angelica Graynamore', 'Patricia Graynamore']}]->
(JoeVersustheVolcano)
MERGE (Nathan)-[:ACTED_IN {roles:['Baw']}]->(JoeVersustheVolcano)
MERGE (JohnS)-[:DIRECTED]->(JoeVersustheVolcano);

MERGE (WhenHarryMetSally:Movie {title:'When Harry Met Sally'}) ON CREATE SET WhenHarryMetSally.released
=1998, WhenHarryMetSally.tagline='Can two friends sleep together and still love each other in the
morning?'

MERGE (MegR:Person {name:'Meg Ryan'}) ON CREATE SET MegR.born=1961
```

```
MERGE (BillyC:Person {name:'Billy Crystal'}) ON CREATE SET BillyC.born=1948
MERGE (CarrieF:Person {name:'Carrie Fisher'}) ON CREATE SET CarrieF.born=1956
MERGE (BrunoK:Person {name:'Bruno Kirby'}) ON CREATE SET BrunoK.born=1949
MERGE (RobR:Person {name:'Rob Reiner'}) ON CREATE SET RobR.born=1947
MERGE (NoraE:Person {name:'Nora Ephron'}) ON CREATE SET NoraE.born=1941

MERGE (BillyC)-[:ACTED_IN {roles:['Harry Burns']}]->(WhenHarryMetSally)
MERGE (MegR)-[:ACTED_IN {roles:['Sally Albright']}]->(WhenHarryMetSally)
MERGE (CarrieF)-[:ACTED_IN {roles:['Marie']}]->(WhenHarryMetSally)
MERGE (BrunoK)-[:ACTED_IN {roles:['Jess']}]->(WhenHarryMetSally)
MERGE (RobR)-[:DIRECTED]->(WhenHarryMetSally)
MERGE (RobR)-[:PRODUCED]->(WhenHarryMetSally)
MERGE (NoraE)-[:PRODUCED]->(WhenHarryMetSally)
MERGE (NoraE)-[:WROTE]->(WhenHarryMetSally);

MERGE (ThatThingYouDo:Movie {title:'That Thing You Do'}) ON CREATE SET ThatThingYouDo.released=1996,
ThatThingYouDo.tagline='In every life there comes a time when that thing you dream becomes that thing you
do'

MERGE (TomH:Person {name:'Tom Hanks'}) ON CREATE SET TomH.born=1956
MERGE (LivT:Person {name:'Liv Tyler'}) ON CREATE SET LivT.born=1977
MERGE (Charlize:Person {name:'Charlize Theron'}) ON CREATE SET Charlize.born=1975

MERGE (TomH)-[:ACTED_IN {roles:['Mr. White']}]->(ThatThingYouDo)
MERGE (LivT)-[:ACTED_IN {roles:['Faye Dolan']}]->(ThatThingYouDo)
MERGE (Charlize)-[:ACTED_IN {roles:['Tina']}]->(ThatThingYouDo)
MERGE (TomH)-[:DIRECTED]->(ThatThingYouDo);

MERGE (TheReplacements:Movie {title:'The Replacements'}) ON CREATE SET TheReplacements.released=2000,
TheReplacements.tagline='Pain heals, Chicks dig scars... Glory lasts forever'

MERGE (Keanu:Person {name:'Keanu Reeves'}) ON CREATE SET Keanu.born=1964
MERGE (Brooke:Person {name:'Brooke Langton'}) ON CREATE SET Brooke.born=1970
MERGE (Gene:Person {name:'Gene Hackman'}) ON CREATE SET Gene.born=1930
MERGE (Orlando:Person {name:'Orlando Jones'}) ON CREATE SET Orlando.born=1968
MERGE (Howard:Person {name:'Howard Deutch'}) ON CREATE SET Howard.born=1950

MERGE (Keanu)-[:ACTED_IN {roles:['Shane Falco']}]->(TheReplacements)
MERGE (Brooke)-[:ACTED_IN {roles:['Annabelle Farrell']}]->(TheReplacements)
MERGE (Gene)-[:ACTED_IN {roles:['Jimmy McGinty']}]->(TheReplacements)
MERGE (Orlando)-[:ACTED_IN {roles:['Clifford Franklin']}]->(TheReplacements)
MERGE (Howard)-[:DIRECTED]->(TheReplacements);

MERGE (RescueDawn:Movie {title:'RescueDawn'}) ON CREATE SET RescueDawn.released=2006, RescueDawn.tagline
='Based on the extraordinary true story of one man\'s fight for freedom'

MERGE (ChristianB:Person {name:'Christian Bale'}) ON CREATE SET ChristianB.born=1974
MERGE (ZachG:Person {name:'Zach Grenier'}) ON CREATE SET ZachG.born=1954
MERGE (MarshallB:Person {name:'Marshall Bell'}) ON CREATE SET MarshallB.born=1942
MERGE (SteveZ:Person {name:'Steve Zahn'}) ON CREATE SET SteveZ.born=1967
MERGE (WernerH:Person {name:'Werner Herzog'}) ON CREATE SET WernerH.born=1942

MERGE (MarshallB)-[:ACTED_IN {roles:['Admiral']}]->(RescueDawn)
MERGE (ChristianB)-[:ACTED_IN {roles:['Dieter Dengler']}]->(RescueDawn)
MERGE (ZachG)-[:ACTED_IN {roles:['Squad Leader']}]->(RescueDawn)
MERGE (SteveZ)-[:ACTED_IN {roles:['Duane']}]->(RescueDawn)
MERGE (WernerH)-[:DIRECTED]->(RescueDawn);

MERGE (TheBirdcage:Movie {title:'The Birdcage'}) ON CREATE SET TheBirdcage.released=1996,
TheBirdcage.tagline='Come as you are'

MERGE (MikeN:Person {name:'Mike Nichols'}) ON CREATE SET MikeN.born=1931
MERGE (Robin:Person {name:'Robin Williams'}) ON CREATE SET Robin.born=1951
MERGE (Nathan:Person {name:'Nathan Lane'}) ON CREATE SET Nathan.born=1956
MERGE (Gene:Person {name:'Gene Hackman'}) ON CREATE SET Gene.born=1930

MERGE (Robin)-[:ACTED_IN {roles:['Armand Goldman']}]->(TheBirdcage)
MERGE (Nathan)-[:ACTED_IN {roles:['Albert Goldman']}]->(TheBirdcage)
MERGE (Gene)-[:ACTED_IN {roles:['Sen. Kevin Keeley']}]->(TheBirdcage)
MERGE (MikeN)-[:DIRECTED]->(TheBirdcage);

MERGE (Unforgiven:Movie {title:'Unforgiven'}) ON CREATE SET Unforgiven.released=1992, Unforgiven.tagline
='It\'s a hell of a thing, killing a man'

MERGE (Gene:Person {name:'Gene Hackman'}) ON CREATE SET Gene.born=1930
MERGE (RichardH:Person {name:'Richard Harris'}) ON CREATE SET RichardH.born=1930
MERGE (ClintE:Person {name:'Clint Eastwood'}) ON CREATE SET ClintE.born=1930
```

```
MERGE (RichardH)-[:ACTED_IN {roles:['English Bob']}]->(Unforgiven)
MERGE (ClintE)-[:ACTED_IN {roles:['Bill Munny']}]->(Unforgiven)
MERGE (Gene)-[:ACTED_IN {roles:['Little Bill Daggett']}]->(Unforgiven)
MERGE (ClintE)-[:DIRECTED]->(Unforgiven);

MERGE (JohnnyMnemonic:Movie {title:'Johnny Mnemonic'}) ON CREATE SET JohnnyMnemonic.released=1995,
JohnnyMnemonic.tagline='The hottest data on earth. In the coolest head in town'

MERGE (Keanu:Person {name:'Keanu Reeves'}) ON CREATE SET Keanu.born=1964
MERGE (Takeshi:Person {name:'Takeshi Kitano'}) ON CREATE SET Takeshi.born=1947
MERGE (Dina:Person {name:'Dina Meyer'}) ON CREATE SET Dina.born=1968
MERGE (IceT:Person {name:'Ice-T'}) ON CREATE SET IceT.born=1958
MERGE (RobertL:Person {name:'Robert Longo'}) ON CREATE SET RobertL.born=1953

MERGE (Keanu)-[:ACTED_IN {roles:['Johnny Mnemonic']}]->(JohnnyMnemonic)
MERGE (Takeshi)-[:ACTED_IN {roles:['Takahashi']}]->(JohnnyMnemonic)
MERGE (Dina)-[:ACTED_IN {roles:['Jane']}]->(JohnnyMnemonic)
MERGE (IceT)-[:ACTED_IN {roles:['J-Bone']}]->(JohnnyMnemonic)
MERGE (RobertL)-[:DIRECTED]->(JohnnyMnemonic);

MERGE (CloudAtlas:Movie {title:'Cloud Atlas'}) ON CREATE SET CloudAtlas.released=2012, CloudAtlas.tagline
='Everything is connected'

MERGE (TomH:Person {name:'Tom Hanks'}) ON CREATE SET TomH.born=1956
MERGE (Hugo:Person {name:'Hugo Weaving'}) ON CREATE SET Hugo.born=1960
MERGE (HalleB:Person {name:'Halle Berry'}) ON CREATE SET HalleB.born=1966
MERGE (JimB:Person {name:'Jim Broadbent'}) ON CREATE SET JimB.born=1949
MERGE (TomT:Person {name:'Tom Tykwer'}) ON CREATE SET TomT.born=1965
MERGE (DavidMitchell:Person {name:'David Mitchell'}) ON CREATE SET DavidMitchell.born=1969
MERGE (StefanArndt:Person {name:'Stefan Arndt'}) ON CREATE SET StefanArndt.born=1961
MERGE (LillyW:Person {name:'Lilly Wachowski'}) ON CREATE SET LillyW.born=1967
MERGE (LanaW:Person {name:'Lana Wachowski'}) ON CREATE SET LanaW.born=1965

MERGE (TomH)-[:ACTED_IN {roles:['Zachry', 'Dr. Henry Goose', 'Isaac Sachs', 'Dermot Hoggins']}]->
(CloudAtlas)
MERGE (Hugo)-[:ACTED_IN {roles:['Bill Smoke', 'Haskell Moore', 'Tadeusz Kesselring', 'Nurse Noakes',
'Boardman Mephi', 'Old Georgie']}]->(CloudAtlas)
MERGE (HalleB)-[:ACTED_IN {roles:['Luisa Rey', 'Jocasta Ayrs', 'Ovid', 'Meronym']}]->(CloudAtlas)
MERGE (JimB)-[:ACTED_IN {roles:['Vyvyan Ayrs', 'Captain Molyneux', 'Timothy Cavendish']}]->(CloudAtlas)
MERGE (TomT)-[:DIRECTED]->(CloudAtlas)
MERGE (LillyW)-[:DIRECTED]->(CloudAtlas)
MERGE (LanaW)-[:DIRECTED]->(CloudAtlas)
MERGE (DavidMitchell)-[:WROTE]->(CloudAtlas)
MERGE (StefanArndt)-[:PRODUCED]->(CloudAtlas);

MERGE (TheDaVinciCode:Movie {title:'The Da Vinci Code'}) ON CREATE SET TheDaVinciCode.released=2006,
TheDaVinciCode.tagline='Break The Codes'

MERGE (TomH:Person {name:'Tom Hanks'}) ON CREATE SET TomH.born=1956
MERGE (IanM:Person {name:'Ian McKellen'}) ON CREATE SET IanM.born=1939
MERGE (AudreyT:Person {name:'Audrey Tautou'}) ON CREATE SET AudreyT.born=1976
MERGE (PaulB:Person {name:'Paul Bettany'}) ON CREATE SET PaulB.born=1971
MERGE (RonH:Person {name:'Ron Howard'}) ON CREATE SET RonH.born=1954

MERGE (TomH)-[:ACTED_IN {roles:['Dr. Robert Langdon']}]->(TheDaVinciCode)
MERGE (IanM)-[:ACTED_IN {roles:['Sir Leight Teabing']}]->(TheDaVinciCode)
MERGE (AudreyT)-[:ACTED_IN {roles:['Sophie Neveu']}]->(TheDaVinciCode)
MERGE (PaulB)-[:ACTED_IN {roles:['Silas']}]->(TheDaVinciCode)
MERGE (RonH)-[:DIRECTED]->(TheDaVinciCode);

MERGE (VforVendetta:Movie {title:'V for Vendetta'}) ON CREATE SET VforVendetta.released=2006,
VforVendetta.tagline='Freedom! Forever!'

MERGE (Hugo:Person {name:'Hugo Weaving'}) ON CREATE SET Hugo.born=1960
MERGE (NatalieP:Person {name:'Natalie Portman'}) ON CREATE SET NatalieP.born=1981
MERGE (StephenR:Person {name:'Stephen Rea'}) ON CREATE SET StephenR.born=1946
MERGE (JohnH:Person {name:'John Hurt'}) ON CREATE SET JohnH.born=1940
MERGE (BenM:Person {name:'Ben Miles'}) ON CREATE SET BenM.born=1967
MERGE (LillyW:Person {name:'Lilly Wachowski'}) ON CREATE SET LillyW.born=1967
MERGE (LanaW:Person {name:'Lana Wachowski'}) ON CREATE SET LanaW.born=1965
MERGE (JamesM:Person {name:'James Marshall'}) ON CREATE SET JamesM.born=1967
MERGE (JoelS:Person {name:'Joel Silver'}) ON CREATE SET JoelS.born=1952

MERGE (Hugo)-[:ACTED_IN {roles:['V']}]->(VforVendetta)
MERGE (NatalieP)-[:ACTED_IN {roles:['Evey Hammond']}]->(VforVendetta)
MERGE (StephenR)-[:ACTED_IN {roles:['Eric Finch']}]->(VforVendetta)
```

```
MERGE (JohnH)-[:ACTED_IN {roles:['High Chancellor Adam Sutler']}]->(VforVendetta)
MERGE (BenM)-[:ACTED_IN {roles:['Dascomb']}]->(VforVendetta)
MERGE (JamesM)-[:DIRECTED]->(VforVendetta)
MERGE (LillyW)-[:PRODUCED]->(VforVendetta)
MERGE (LanaW)-[:PRODUCED]->(VforVendetta)
MERGE (JoelS)-[:PRODUCED]->(VforVendetta)
MERGE (LillyW)-[:WROTE]->(VforVendetta)
MERGE (LanaW)-[:WROTE]->(VforVendetta);

MERGE (SpeedRacer:Movie {title:'Speed Racer'}) ON CREATE SET SpeedRacer.released=2008, SpeedRacer.tagline
='Speed has no limits'

MERGE (EmileH:Person {name:'Emile Hirsch'}) ON CREATE SET EmileH.born=1985
MERGE (JohnG:Person {name:'John Goodman'}) ON CREATE SET JohnG.born=1960
MERGE (SusanS:Person {name:'Susan Sarandon'}) ON CREATE SET SusanS.born=1946
MERGE (MatthewF:Person {name:'Matthew Fox'}) ON CREATE SET MatthewF.born=1966
MERGE (ChristinaR:Person {name:'Christina Ricci'}) ON CREATE SET ChristinaR.born=1980
MERGE (Rain:Person {name:'Rain'}) ON CREATE SET Rain.born=1982
MERGE (BenM:Person {name:'Ben Miles'}) ON CREATE SET BenM.born=1967
MERGE (LillyW:Person {name:'Lilly Wachowski'}) ON CREATE SET LillyW.born=1967
MERGE (LanaW:Person {name:'Lana Wachowski'}) ON CREATE SET LanaW.born=1965
MERGE (JoelS:Person {name:'Joel Silver'}) ON CREATE SET JoelS.born=1952

MERGE (EmileH)-[:ACTED_IN {roles:['Speed Racer']}]->(SpeedRacer)
MERGE (JohnG)-[:ACTED_IN {roles:['Pops']}]->(SpeedRacer)
MERGE (SusanS)-[:ACTED_IN {roles:['Mom']}]->(SpeedRacer)
MERGE (MatthewF)-[:ACTED_IN {roles:['Racer X']}]->(SpeedRacer)
MERGE (ChristinaR)-[:ACTED_IN {roles:['Trixie']}]->(SpeedRacer)
MERGE (Rain)-[:ACTED_IN {roles:['Taejo Togokahn']}]->(SpeedRacer)
MERGE (BenM)-[:ACTED_IN {roles:['Cass Jones']}]->(SpeedRacer)
MERGE (LillyW)-[:DIRECTED]->(SpeedRacer)
MERGE (LanaW)-[:DIRECTED]->(SpeedRacer)
MERGE (LillyW)-[:WROTE]->(SpeedRacer)
MERGE (LanaW)-[:WROTE]->(SpeedRacer)
MERGE (JoelS)-[:PRODUCED]->(SpeedRacer);

MERGE (NinjaAssassin:Movie {title:'Ninja Assassin'}) ON CREATE SET NinjaAssassin.released=2009,
NinjaAssassin.tagline='Prepare to enter a secret world of assassins'

MERGE (NaomieH:Person {name:'Naomie Harris'})
MERGE (Rain:Person {name:'Rain'}) ON CREATE SET Rain.born=1982
MERGE (BenM:Person {name:'Ben Miles'}) ON CREATE SET BenM.born=1967
MERGE (LillyW:Person {name:'Lilly Wachowski'}) ON CREATE SET LillyW.born=1967
MERGE (LanaW:Person {name:'Lana Wachowski'}) ON CREATE SET LanaW.born=1965
MERGE (RickY:Person {name:'Rick Yune'}) ON CREATE SET RickY.born=1971
MERGE (JamesM:Person {name:'James Marshall'}) ON CREATE SET JamesM.born=1967
MERGE (JoelS:Person {name:'Joel Silver'}) ON CREATE SET JoelS.born=1952

MERGE (Rain)-[:ACTED_IN {roles:['Raizo']}]->(NinjaAssassin)
MERGE (NaomieH)-[:ACTED_IN {roles:['Mika Coretti']}]->(NinjaAssassin)
MERGE (RickY)-[:ACTED_IN {roles:['Takeshi']}]->(NinjaAssassin)
MERGE (BenM)-[:ACTED_IN {roles:['Ryan Maslow']}]->(NinjaAssassin)
MERGE (JamesM)-[:DIRECTED]->(NinjaAssassin)
MERGE (LillyW)-[:PRODUCED]->(NinjaAssassin)
MERGE (LanaW)-[:PRODUCED]->(NinjaAssassin)
MERGE (JoelS)-[:PRODUCED]->(NinjaAssassin);

MERGE (TheGreenMile:Movie {title:'The Green Mile'}) ON CREATE SET TheGreenMile.released=1999,
TheGreenMile.tagline='Walk a mile you\'ll never forget.'

MERGE (TomH:Person {name:'Tom Hanks'}) ON CREATE SET TomH.born=1956
MERGE (JamesC:Person {name:'James Cromwell'}) ON CREATE SET JamesC.born=1940
MERGE (BonnieH:Person {name:'Bonnie Hunt'}) ON CREATE SET BonnieH.born=1961
MERGE (MichaelD:Person {name:'Michael Clarke Duncan'}) ON CREATE SET MichaelD.born=1957
MERGE (DavidM:Person {name:'David Morse'}) ON CREATE SET DavidM.born=1953
MERGE (SamR:Person {name:'Sam Rockwell'}) ON CREATE SET SamR.born=1968
MERGE (GaryS:Person {name:'Gary Sinise'}) ON CREATE SET GaryS.born=1955
MERGE (PatriciaC:Person {name:'Patricia Clarkson'}) ON CREATE SET PatriciaC.born=1959
MERGE (FrankD:Person {name:'Frank Darabont'}) ON CREATE SET FrankD.born=1959

MERGE (TomH)-[:ACTED_IN {roles:['Paul Edgecomb']}]->(TheGreenMile)
MERGE (MichaelD)-[:ACTED_IN {roles:['John Coffey']}]->(TheGreenMile)
MERGE (DavidM)-[:ACTED_IN {roles:['Brutus "Brutal" Howell']}]->(TheGreenMile)
MERGE (BonnieH)-[:ACTED_IN {roles:['Jan Edgecomb']}]->(TheGreenMile)
MERGE (JamesC)-[:ACTED_IN {roles:['Warden Hal Moores']}]->(TheGreenMile)
MERGE (SamR)-[:ACTED_IN {roles:['"Wild Bill" Wharton']}]->(TheGreenMile)
MERGE (GaryS)-[:ACTED_IN {roles:['Burt Hammersmith']}]->(TheGreenMile)
```

```
MERGE (PatriciaC)-[:ACTED_IN {roles:['Melinda Moores']}]->(TheGreenMile)
MERGE (FrankD)-[:DIRECTED]->(TheGreenMile);

MERGE (FrostNixon:Movie {title:'Frost/Nixon'}) ON CREATE SET FrostNixon.released=2008, FrostNixon.tagline
='400 million people were waiting for the truth.'

MERGE (FrankL:Person {name:'Frank Langella'}) ON CREATE SET FrankL.born=1938
MERGE (MichaelS:Person {name:'Michael Sheen'}) ON CREATE SET MichaelS.born=1969
MERGE (OliverP:Person {name:'Oliver Platt'}) ON CREATE SET OliverP.born=1960
MERGE (KevinB:Person {name:'Kevin Bacon'}) ON CREATE SET KevinB.born=1958
MERGE (SamR:Person {name:'Sam Rockwell'}) ON CREATE SET SamR.born=1968
MERGE (RonH:Person {name:'Ron Howard'}) ON CREATE SET RonH.born=1954

MERGE (FrankL)-[:ACTED_IN {roles:['Richard Nixon']}]->(FrostNixon)
MERGE (MichaelS)-[:ACTED_IN {roles:['David Frost']}]->(FrostNixon)
MERGE (KevinB)-[:ACTED_IN {roles:['Jack Brennan']}]->(FrostNixon)
MERGE (OliverP)-[:ACTED_IN {roles:['Bob Zelnick']}]->(FrostNixon)
MERGE (SamR)-[:ACTED_IN {roles:['James Reston, Jr.']}]->(FrostNixon)
MERGE (RonH)-[:DIRECTED]->(FrostNixon);

MERGE (Hoffa:Movie {title:'Hoffa'}) ON CREATE SET Hoffa.released=1992, Hoffa.tagline='He didn\'t want law.
He wanted justice.'

MERGE (DannyD:Person {name:'Danny DeVito'}) ON CREATE SET DannyD.born=1944
MERGE (JohnR:Person {name:'John C. Reilly'}) ON CREATE SET JohnR.born=1965
MERGE (JackN:Person {name:'Jack Nicholson'}) ON CREATE SET JackN.born=1937
MERGE (JTW:Person {name:'J.T. Walsh'}) ON CREATE SET JTW.born=1943

MERGE (JackN)-[:ACTED_IN {roles:['Hoffa']}]->(Hoffa)
MERGE (DannyD)-[:ACTED_IN {roles:['Robert "Bobby" Ciaro']}]->(Hoffa)
MERGE (JTW)-[:ACTED_IN {roles:['Frank Fitzsimmons']}]->(Hoffa)
MERGE (JohnR)-[:ACTED_IN {roles:['Peter "Pete" Connelly']}]->(Hoffa)
MERGE (DannyD)-[:DIRECTED]->(Hoffa);

MERGE (Apollo13:Movie {title:'Apollo 13'}) ON CREATE SET Apollo13.released=1995, Apollo13.tagline
='Houston, we have a problem.'

MERGE (TomH:Person {name:'Tom Hanks'}) ON CREATE SET TomH.born=1956
MERGE (EdH:Person {name:'Ed Harris'}) ON CREATE SET EdH.born=1950
MERGE (BillPax:Person {name:'Bill Paxton'}) ON CREATE SET BillPax.born=1955
MERGE (KevinB:Person {name:'Kevin Bacon'}) ON CREATE SET KevinB.born=1958
MERGE (GaryS:Person {name:'Gary Sinise'}) ON CREATE SET GaryS.born=1955
MERGE (RonH:Person {name:'Ron Howard'}) ON CREATE SET RonH.born=1954

MERGE (TomH)-[:ACTED_IN {roles:['Jim Lovell']}]->(Apollo13)
MERGE (KevinB)-[:ACTED_IN {roles:['Jack Swigert']}]->(Apollo13)
MERGE (EdH)-[:ACTED_IN {roles:['Gene Kranz']}]->(Apollo13)
MERGE (BillPax)-[:ACTED_IN {roles:['Fred Haise']}]->(Apollo13)
MERGE (GaryS)-[:ACTED_IN {roles:['Ken Mattingly']}]->(Apollo13)
MERGE (RonH)-[:DIRECTED]->(Apollo13);

MERGE (Twister:Movie {title:'Twister'}) ON CREATE SET Twister.released=1996, Twister.tagline='Don\'t
Breathe. Don\'t Look Back.'

MERGE (PhilipH:Person {name:'Philip Seymour Hoffman'}) ON CREATE SET PhilipH.born=1967
MERGE (JanB:Person {name:'Jan de Bont'}) ON CREATE SET JanB.born=1943
MERGE (BillPax:Person {name:'Bill Paxton'}) ON CREATE SET BillPax.born=1955
MERGE (HelenH:Person {name:'Helen Hunt'}) ON CREATE SET HelenH.born=1963
MERGE (ZachG:Person {name:'Zach Grenier'}) ON CREATE SET ZachG.born=1954

MERGE (BillPax)-[:ACTED_IN {roles:['Bill Harding']}]->(Twister)
MERGE (HelenH)-[:ACTED_IN {roles:['Dr. Jo Harding']}]->(Twister)
MERGE (ZachG)-[:ACTED_IN {roles:['Eddie']}]->(Twister)
MERGE (PhilipH)-[:ACTED_IN {roles:['Dustin "Dusty" Davis']}]->(Twister)
MERGE (JanB)-[:DIRECTED]->(Twister);

MERGE (CastAway:Movie {title:'Cast Away'}) ON CREATE SET CastAway.released=2000, CastAway.tagline='At the
edge of the world, his journey begins.'

MERGE (TomH:Person {name:'Tom Hanks'}) ON CREATE SET TomH.born=1956
MERGE (HelenH:Person {name:'Helen Hunt'}) ON CREATE SET HelenH.born=1963
MERGE (RobertZ:Person {name:'Robert Zemeckis'}) ON CREATE SET RobertZ.born=1951

MERGE (TomH)-[:ACTED_IN {roles:['Chuck Noland']}]->(CastAway)
MERGE (HelenH)-[:ACTED_IN {roles:['Kelly Frears']}]->(CastAway)
MERGE (RobertZ)-[:DIRECTED]->(CastAway);
```

```
MERGE (OneFlewOvertheCuckoosNest:Movie {title:'One Flew Over the Cuckoo\'s Nest'}) ON CREATE SET
OneFlewOvertheCuckoosNest.released=1975, OneFlewOvertheCuckoosNest.tagline='If he\'s crazy, what does that
make you?'

MERGE (MilosF:Person {name:'Milos Forman'}) ON CREATE SET MilosF.born=1932
MERGE (JackN:Person {name:'Jack Nicholson'}) ON CREATE SET JackN.born=1937
MERGE (DannyD:Person {name:'Danny DeVito'}) ON CREATE SET DannyD.born=1944

MERGE (JackN)-[:ACTED_IN {roles:['Randle McMurphy']}]->(OneFlewOvertheCuckoosNest)
MERGE (DannyD)-[:ACTED_IN {roles:['Martini']}]->(OneFlewOvertheCuckoosNest)
MERGE (MilosF)-[:DIRECTED]->(OneFlewOvertheCuckoosNest);

MERGE (SomethingsGottaGive:Movie {title:'Something\'s Gotta Give'}) ON CREATE SET
SomethingsGottaGive.released=2003

MERGE (JackN:Person {name:'Jack Nicholson'}) ON CREATE SET JackN.born=1937
MERGE (DianeK:Person {name:'Diane Keaton'}) ON CREATE SET DianeK.born=1946
MERGE (NancyM:Person {name:'Nancy Meyers'}) ON CREATE SET NancyM.born=1949
MERGE (Keanu:Person {name:'Keanu Reeves'}) ON CREATE SET Keanu.born=1964

MERGE (JackN)-[:ACTED_IN {roles:['Harry Sanborn']}]->(SomethingsGottaGive)
MERGE (DianeK)-[:ACTED_IN {roles:['Erica Barry']}]->(SomethingsGottaGive)
MERGE (Keanu)-[:ACTED_IN {roles:['Julian Mercer']}]->(SomethingsGottaGive)
MERGE (NancyM)-[:DIRECTED]->(SomethingsGottaGive)
MERGE (NancyM)-[:PRODUCED]->(SomethingsGottaGive)
MERGE (NancyM)-[:WROTE]->(SomethingsGottaGive);

MERGE (BicentennialMan:Movie {title:'Bicentennial Man'}) ON CREATE SET BicentennialMan.released=1999,
BicentennialMan.tagline='One robot\'s 200 year journey to become an ordinary man.'

MERGE (ChrisC:Person {name:'Chris Columbus'}) ON CREATE SET ChrisC.born=1958
MERGE (Robin:Person {name:'Robin Williams'}) ON CREATE SET Robin.born=1951
MERGE (OliverP:Person {name:'Oliver Platt'}) ON CREATE SET OliverP.born=1960

MERGE (Robin)-[:ACTED_IN {roles:['Andrew Marin']}]->(BicentennialMan)
MERGE (OliverP)-[:ACTED_IN {roles:['Rupert Burns']}]->(BicentennialMan)
MERGE (ChrisC)-[:DIRECTED]->(BicentennialMan);

MERGE (CharlieWilsonsWar:Movie {title:'Charlie Wilson\'s War'}) ON CREATE SET CharlieWilsonsWar.released
=2007, CharlieWilsonsWar.tagline='A stiff drink. A little mascara. A lot of nerve. Who said they couldn\'t
bring down the Soviet empire.'

MERGE (TomH:Person {name:'Tom Hanks'}) ON CREATE SET TomH.born=1956
MERGE (PhilipH:Person {name:'Philip Seymour Hoffman'}) ON CREATE SET PhilipH.born=1967
MERGE (JuliaR:Person {name:'Julia Roberts'}) ON CREATE SET JuliaR.born=1967
MERGE (MikeN:Person {name:'Mike Nichols'}) ON CREATE SET MikeN.born=1931

MERGE (TomH)-[:ACTED_IN {roles:['Rep. Charlie Wilson']}]->(CharlieWilsonsWar)
MERGE (JuliaR)-[:ACTED_IN {roles:['Joanne Herring']}]->(CharlieWilsonsWar)
MERGE (PhilipH)-[:ACTED_IN {roles:['Gust Avrakotos']}]->(CharlieWilsonsWar)
MERGE (MikeN)-[:DIRECTED]->(CharlieWilsonsWar);

MERGE (ThePolarExpress:Movie {title:'The Polar Express'}) ON CREATE SET ThePolarExpress.released=2004,
ThePolarExpress.tagline='This Holiday Season... Believe'

MERGE (TomH:Person {name:'Tom Hanks'}) ON CREATE SET TomH.born=1956
MERGE (RobertZ:Person {name:'Robert Zemeckis'}) ON CREATE SET RobertZ.born=1951

MERGE (TomH)-[:ACTED_IN {roles:['Hero Boy', 'Father', 'Conductor', 'Hobo', 'Scrooge', 'Santa Claus']}]->
(ThePolarExpress)
MERGE (RobertZ)-[:DIRECTED]->(ThePolarExpress);

MERGE (ALeagueofTheirOwn:Movie {title:'A League of Their Own'}) ON CREATE SET ALeagueofTheirOwn.released
=1992, ALeagueofTheirOwn.tagline='Once in a lifetime you get a chance to do something different.'

MERGE (TomH:Person {name:'Tom Hanks'}) ON CREATE SET TomH.born=1956
MERGE (Madonna:Person {name:'Madonna'}) ON CREATE SET Madonna.born=1954
MERGE (GeenaD:Person {name:'Geena Davis'}) ON CREATE SET GeenaD.born=1956
MERGE (LoriP:Person {name:'Lori Petty'}) ON CREATE SET LoriP.born=1963
MERGE (PennyM:Person {name:'Penny Marshall'}) ON CREATE SET PennyM.born=1943
MERGE (RosieO:Person {name:'Rosie O\'Donnell'}) ON CREATE SET RosieO.born=1962
MERGE (BillPax:Person {name:'Bill Paxton'}) ON CREATE SET BillPax.born=1955

MERGE (TomH)-[:ACTED_IN {roles:['Jimmy Dugan']}]->(ALeagueofTheirOwn)
MERGE (GeenaD)-[:ACTED_IN {roles:['Dottie Hinson']}]->(ALeagueofTheirOwn)
MERGE (LoriP)-[:ACTED_IN {roles:['Kit Keller']}]->(ALeagueofTheirOwn)
MERGE (RosieO)-[:ACTED_IN {roles:['Doris Murphy']}]->(ALeagueofTheirOwn)
```

```
MERGE (Madonna)-[:ACTED_IN {roles:['"All the Way" Mae Mordabito']}]->(ALeagueofTheirOwn)
MERGE (BillPax)-[:ACTED_IN {roles:['Bob Hinson']}]->(ALeagueofTheirOwn)
MERGE (PennyM)-[:DIRECTED]->(ALeagueofTheirOwn);


MATCH (CloudAtlas:Movie {title:'Cloud Atlas'})
MATCH (TheReplacements:Movie {title:'The Replacements'})
MATCH (Unforgiven:Movie {title:'Unforgiven'})
MATCH (TheBirdcage:Movie {title:'The Birdcage'})
MATCH (TheDaVinciCode:Movie {title:'The Da Vinci Code'})
MATCH (JerryMaguire:Movie {title:'Jerry Maguire'})

MERGE (PaulBlythe:Person {name:'Paul Blythe'})
MERGE (AngelaScope:Person {name:'Angela Scope'})
MERGE (JessicaThompson:Person {name:'Jessica Thompson'})
MERGE (JamesThompson:Person {name:'James Thompson'})

MERGE (JamesThompson)-[:FOLLOWS]->(JessicaThompson)
MERGE (AngelaScope)-[:FOLLOWS]->(JessicaThompson)
MERGE (PaulBlythe)-[:FOLLOWS]->(AngelaScope)

MERGE (JessicaThompson)-[:REVIEWED {summary:'An amazing journey', rating:95}]->(CloudAtlas)
MERGE (JessicaThompson)-[:REVIEWED {summary:'Silly, but fun', rating:65}]->(TheReplacements)
MERGE (JamesThompson)-[:REVIEWED {summary:'The coolest football movie ever', rating:100}]->
(TheReplacements)
MERGE (AngelaScope)-[:REVIEWED {summary:'Pretty funny at times', rating:62}]->(TheReplacements)
MERGE (JessicaThompson)-[:REVIEWED {summary:'Dark, but compelling', rating:85}]->(Unforgiven)
MERGE (JessicaThompson)-[:REVIEWED {summary:"Slapstick redeemed only by the Robin Williams and Gene
Hackman's stellar performances", rating:45}]->(TheBirdcage)
MERGE (JessicaThompson)-[:REVIEWED {summary:'A solid romp', rating:68}]->(TheDaVinciCode)
MERGE (JamesThompson)-[:REVIEWED {summary:'Fun, but a little far fetched', rating:65}]->(TheDaVinciCode)
MERGE (JessicaThompson)-[:REVIEWED {summary:'You had me at Jerry', rating:92}]->(JerryMaguire);
```

You should now have a graph with 171 nodes and 253 relationships.

Also note that when you run Cypher queries in the Cypher editor, a reusable result frame is created. You can edit the query directly in the result frame or use a fresh one by using the main Cypher editor.

Find actors and movies

A specific actor

If you want to find a specific actor in the graph, you can `MATCH` the `Person` node and refer to the actor using the `name` property:

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

Note that the `WHERE` clause refers to the variable you employed for the `Person` node (`tom`), and not the node label.

A specific movie

The same goes when querying for a specific movie, for example, "Cloud Atlas". Now, instead of matching the `Person` node, you query the `Movie` nodes and use the `WHERE` clause to locate the movie with the property `title`:

```
MATCH (cloudAtlas:Movie)
WHERE cloudAtlas.title = "Cloud Atlas"
RETURN cloudAtlas
```

Limit returned results

If you want to limit the number of returned results without any specific order, you can add `LIMIT` to the `RETURN` clause. This query finds all the `Person` nodes in the graph and returns 10 of them using the `name` property.

```
MATCH (p:Person)
RETURN p.name LIMIT 10
```

For this query, property values are returned and you can only view the results as a table.

The `WHERE` subclause

If you want to return nodes by a specific property, you can use the `WHERE` subclause. You can either return a specific value, in this case movies released in one specific year:

```
MATCH (nineties:Movie)
WHERE nineties.released = 1990
RETURN nineties.title
```

Or within a range:

```
MATCH (nineties:Movie)
WHERE nineties.released > 1990 AND nineties.released < 2000
RETURN nineties.title
```

For the first query, the result is "Joe Versus the Volcano", and for the second:

nineties.title
"The Matrix"
"The Devil's Advocate"
"A Few Good Men"
"As Good as It Gets"
"What Dreams May Come"
"Snow Falling on Cedars"
"You've Got Mail"
"Sleepless in Seattle"
"When Harry Met Sally"
"That Thing You Do"
"The Birdcage"
"Unforgiven"
"Johnny Mnemonic"

nineties.title
"The Green Mile"
"Hoffa"
"Apollo 13"
"Twister"
"Bicentennial Man"
"A League of Their Own"
Rows: 19

Find patterns

In the previous step, you queried the graph for nodes. Now you are going to retrieve related nodes by finding the patterns within the graph, that is, how these nodes are related to each other.

All Tom Hanks movies

To list the movies Tom Hanks have been part of, you need to query his `Person` node and how it is connected to the `Movie` nodes through a relationship.

```
MATCH (tom:Person {name: "Tom Hanks"})-[r]->(tomHanksMovies:Movie)
RETURN tom,r,tomHanksMovies
```

This is the result:

Note that Tom Hanks' node is connected to `Movie` nodes through two different relationships: `ACTED_IN` and `DIRECTED`, which means a `Person` node can be both an actor and a director.

Directors of "Cloud Atlas"

As mentioned previously, you can find information about directors through the relationship `DIRECTED`. To learn who directed "Cloud Atlas", execute the following query:

```
MATCH (cloudAtlas:Movie {title: "Cloud Atlas"})<-[:DIRECTED]-(directors)
RETURN directors.name
```

Here is the result:

directors.name
"Lilly Wachowski"
"Lana Wachowski"
"Tom Tykwer"
Rows: 3

Tom Hanks' co-actors

If you want to expand the pattern and find actors who worked on the same movie, you can use the following query:

```
MATCH (tom:Person {name:"Tom Hanks"})-[a:ACTED_IN]->(m:Movie)<-[b:ACTED_IN]-(coActors:Person)
RETURN tom, m, a, b, coActors
```

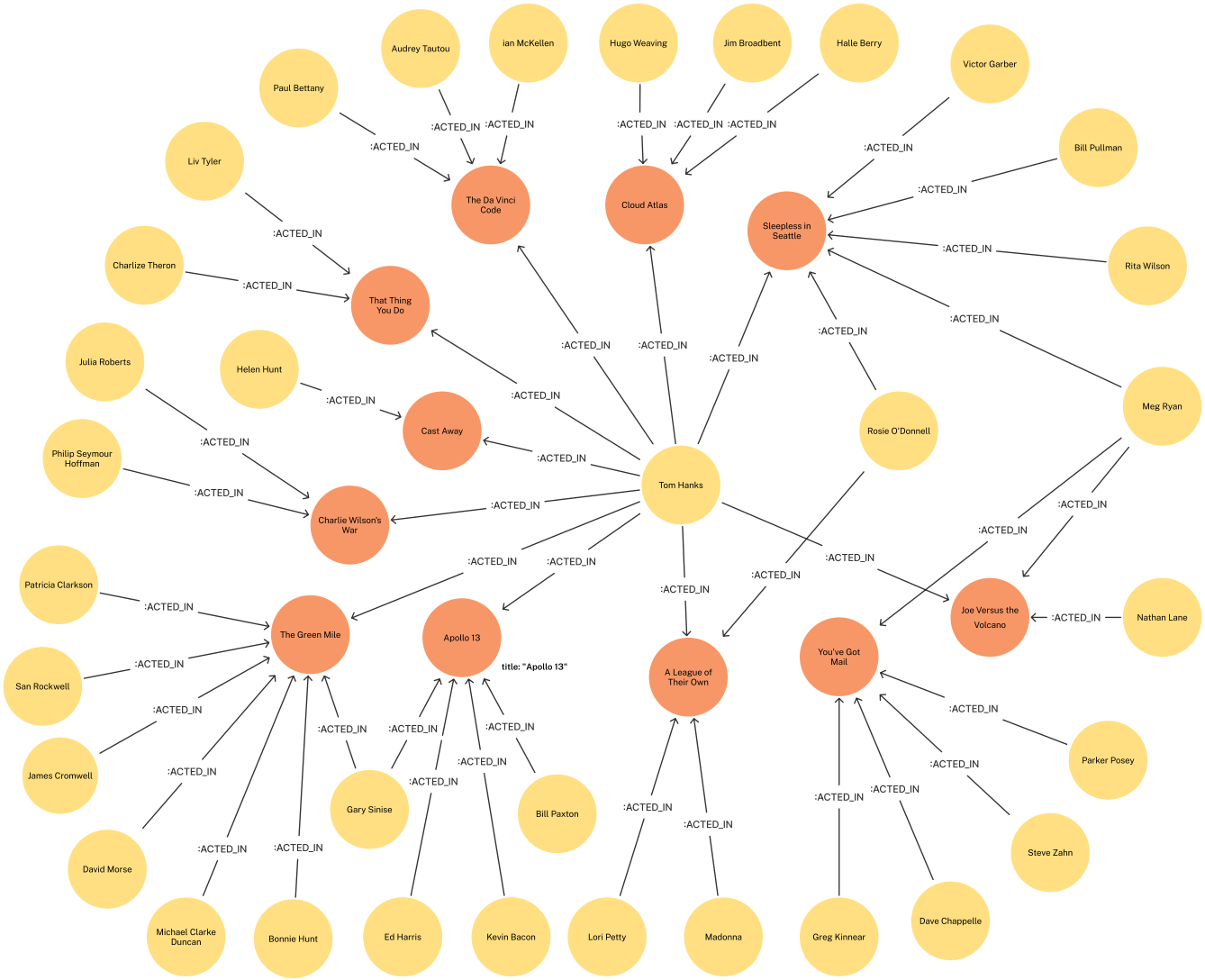
To better understand the query above, break it up. First, the query finds which movies Tom Hanks acted in:

```
MATCH (tom:Person {name: "Tom Hanks"})-[a:ACTED_IN]->(m:Movie)
```

Then, find other `Person` nodes who acted in the same movie:

```
MATCH (m)<-[b:ACTED_IN]-(coActors:Person)
```

This is what your graph should look like:



People related to "Cloud Atlas"

If you don't know what relationships connect the `Person` and `Movie` nodes, you can query the graph to discover that:

```
MATCH (people:Person)-[relatedTo]-(:Movie {title: "Cloud Atlas"})
RETURN people.name, type(relatedTo), relatedTo
```

In this query, the relationship is undirected and unspecified, because you want to retrieve all relationships between these two nodes, no matter the direction and what relationships exist in the graph. Then, in the `RETURN` line, you can specify what property you want to retrieve instead of getting all available ones, as seen in the previous steps.

In Cypher, you can define what specific propety you want to retrieve by combining the variable and the name of the property. So, in this case, `people` is the variable for `Person` nodes and `name` is the property that stores the name of the people, thus `people.name`.

Then, since the relationships are not specified, you can use the `type()` function to retrieve the relationship type, and then `relatedTo` to also retrieve all the properties associated to the relationship (e.g. `roles` in the case of `:ACTED_IN`).

Here is the result:

people.name	type(relatedTo)	relatedTo
"Hugo Weaving"	"ACTED_IN"	[:ACTED_IN {roles: ["Bill Smoke", "Haskell Moore", "Tadeusz Kesselring", "Nurse Noakes", "Boardman Mephi", "Old Georgie"]}]
"Tom Hanks"	"ACTED_IN"	[:ACTED_IN {roles: ["Zachry", "Dr. Henry Goose", "Isaac Sachs", "Dermot Hoggins"]}]
"Halle Berry"	"ACTED_IN"	[:ACTED_IN {roles: ["Luisa Rey", "Jocasta Ayrs", "Ovid", "Meronym"]}]
"Jim Broadbent"	"ACTED_IN"	[:ACTED_IN {roles: ["Vyvyan Ayrs", "Captain Molyneux", "Timothy Cavendish"]}]
"Lilly Wachowski"	"DIRECTED"	[:DIRECTED]
"Lana Wachowski"	"DIRECTED"	[:DIRECTED]
"Tom Tykwer"	"DIRECTED"	[:DIRECTED]
"Stefan Arndt"	"PRODUCED"	[:PRODUCED]
"David Mitchell"	"WROTE"	[:WROTE]
"Jessica Thompson"	"REVIEWED"	[:REVIEWED {summary: "An amazing journey", rating: 95}]
Rows: 10		

Movies and actors connected to Kevin Bacon

The premise of the game "Six Degrees of Kevin Bacon" is to link any actor to Kevin Bacon through their film co-stars in six or fewer steps. This game can be played in the Movie graph using different strategies.

Two possible strategies:

- Variable length patterns
- Built-in shortestPath() algorithm

> **Note:**
>
> For the purpose of simplicity, the example goes through only three hops from Kevin Bacon instead of six.

Variable length pattern

This query finds all `Person` and `Movie` nodes one to three hops away from Kevin Bacon:

```
MATCH (bacon:Person {name:"Kevin Bacon"})-[*1..3]-(a:Person)
RETURN DISTINCT bacon, a
```

The `RETURN` clause includes a `DISTINCT` operator to ensure that results are not duplicated. In this query, it is possible that Kevin Bacon worked with the same actor in different movies, but you only want unique results.

Here is the result:

The downside of this query is that you don't know *how* Kevin Bacon is related to these people and movies. If you want to retrieve the path that connects these nodes, `MATCH` the **path** and return it along with the `Person` or `Movie` nodes involved:

```
MATCH p = (bacon:Person {name:"Kevin Bacon"})-[*1..3]-(a:Person)
RETURN DISTINCT bacon, a, p
```

The `p` variable refers to the entire path between Kevin Bacon and the `Person` nodes he is related to in one to three hops. The `p` variable is then referred to in the `RETURN` clause.

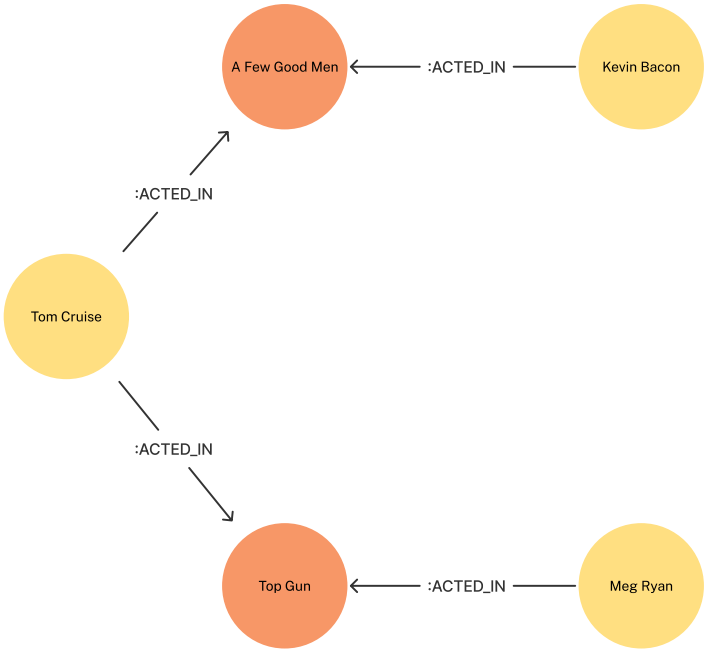
Now the result includes `Movie` nodes too, because this is how Kevin Bacon is connected to other `Person` nodes:

`shortestPath` algorithm

To find the shortest path between two actors, use the algorithm as follows and return the path:

```
MATCH p=shortestPath(
  (bacon:Person {name:"Kevin Bacon"})-[*]-(meg:Person {name:"Meg Ryan"})
)
RETURN p
```

Here is the result of this query:



Clean up

When you're done experimenting, you can remove the movie data set and clean up your Aura instance. You can delete everything with one single query:

```
MATCH (n)
DETACH DELETE n
```

The result is the following message: "Deleted 171 nodes, deleted 253 relationships".

Verify if the graph 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:

```
MATCH (n)
RETURN count(*)
```

That means all nodes and relationships were deleted.

Conclusion

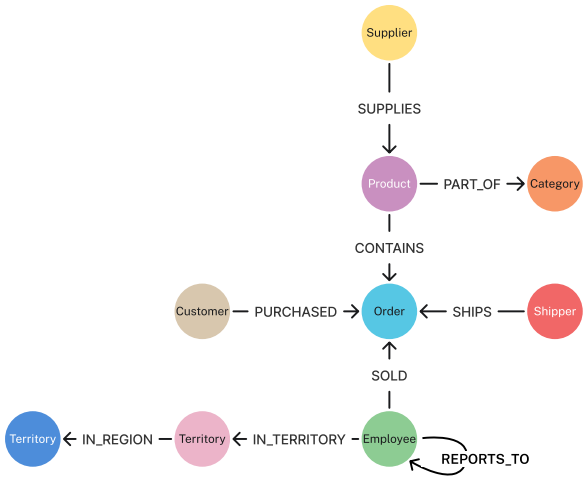
In this tutorial, you have seen how to retrieve specific information from the graph: from single nodes to the relationships between them. You have also performed a shortest path query, so you can see how different nodes are related to each other in different degrees.

Keep learning

If you want to learn more about Cypher and what you can do with Neo4j's propretary query language, refer to:

- Cypher docs → A complete docset with all information about Cypher.
- Cyper Cheat Sheet → A reference document for quick access.

Comparing Cypher with SQL



While there are key differences between Cypher and SQL, it is still possible to compare both languages and write equivalent SQL statements using Cypher. The Northwind dataset is used here to better illustrate the comparison.

> **Tip:**
>
> For a more in-depth explanation on the differences and similarities between graph and relational databases, see Transition from relational to graph database.

Indexing

Indexes are available both in SQL and Cypher. They make searching for a specific node label and attribute combination more efficient.

Indexes in Cypher are only used for finding the starting points of a query; all subsequent pattern matching is done through the graph structure. Cypher supports range, text, point, lookup, full-text, and vector indexes.

In the Northwind dataset, adding indexes on `productName` and `unitPrice` makes searching for a product and its price quicker:

SQL	Cypher
`CREATE INDEX Product_productName ON products ( product_name);` `CREATE INDEX Product_unitPrice ON products (unit_price);`	`CREATE INDEX Product_productName IF NOT EXISTS FOR ( p:Product) ON p.productName;` `CREATE INDEX Product_unitPrice IF NOT EXISTS FOR ( p:Product) ON p.unitPrice;`

Query examples

Select and return records

SQL	Cypher
To select and return records in SQL, select everything from the `products` table: `SELECT p.* FROM products as p;`	In Cypher, you `MATCH` a simple pattern: all nodes with the label `:Product`, and `RETURN` them: `MATCH (p:Product) RETURN p;`

Field access, ordering, and paging

Rather than returning all attributes, you can filter out the ones you are interested in — `ProductName` and `UnitPrice`, for example.

SQL	Cypher
In SQL, this is how you order items by price and return the 10 most expensive items: `SELECT p.ProductName, p.UnitPrice FROM products as p ORDER BY p.UnitPrice DESC LIMIT 10;`	The statement is similar in Cypher, except for the pattern matching part: `MATCH (p:Product) RETURN p.productName, p.unitPrice ORDER BY p.unitPrice DESC LIMIT 10;`

Result

p.productName	p.unitPrice
"Côte de Blaye"	263.5
"Thüringer Rostbratwurst"	123.79
"Mishi Kobe Niku"	97.0
"Sir Rodney's Marmalade"	81.0
"Carnarvon Tigers"	62.5
"Raclette Courdavault"	55.0
"Manjimup Dried Apples"	53.0
"Tarte au sucre"	49.3
"Ipoh Coffee"	46.0
"Rössle Sauerkraut"	45.6

Important:

Remember that labels, relationship types, and property names are **case sensitive** in Neo4j. For more details on naming rules, see the Cypher Manual → Naming rules and recommendations.

Find a single product by name

There are different ways to query the database and retrieve a single item, for example, a product named `Chocolade`. You can do that, for example, by filtering by equality:

SQL	Cypher

In SQL, you can filter data using the `WHERE` clause:
 	
`SELECT p.ProductName, p.UnitPrice`
`FROM products AS p`
`WHERE p.ProductName = 'Chocolade';`
	In Cypher, the `WHERE` clause belongs to the `MATCH` statement:
	
	`MATCH (p:Product)`
	`WHERE p.productName = 'Chocolade'`
	`RETURN p.productName, p.unitPrice;`
	
	A shorter option is to use the label `productName` to specify the product in the `MATCH` statement:
	
	`MATCH (p:Product {productName:'Chocolade'})`
	`RETURN p.productName, p.unitPrice;`

Result

p.productName	p.unitPrice
"Chocolade"	12.75

Filter products

Filter by list/range

SQL	Cypher

In SQL, you can use the operator `IN`:
 	
`SELECT p.ProductName, p.UnitPrice`
`FROM products as p`
`WHERE p.ProductName IN ('Chocolade','Chai');`
	Cypher has full collection support, including `IN` and other collection functions, predicates, and transformations:
	
	`MATCH (p:Product)`
	`WHERE p.productName IN ['Chocolade','Chai']`
	`RETURN p.productName, p.unitPrice;`

Result

p.productName	p.unitPrice
"Chocolade"	12.75
"Chai"	18.0

Filter by multiple numeric and textual predicates

SQL	Cypher

This query retrieves products with a name starting with "C" and a price larger than 100:
 	
`SELECT p.ProductName, p.UnitPrice`
`FROM products AS p`
`WHERE p.ProductName LIKE 'C%' AND p.UnitPrice > 100;`
	In Cypher, the `LIKE` operator is replaced by the `STARTS WITH`, `CONTAINS`, and `ENDS WITH` operators:
	
	`MATCH (p:Product)`
	`WHERE p.productName STARTS WITH 'C' AND p.unitPrice > 100`
	`RETURN p.productName, p.unitPrice;`
	
	You can also use a regular expression to get all products with name starting with "C" and their prices:
	
	`MATCH (p:Product)`
	`WHERE p.productName =~ '^C.*'`
	`RETURN p.productName, p.unitPrice`

Result

p.productName	p.unitPrice
"Côte de Blaye"	263.5

Joining products with customers

SQL	Cypher

In SQL, if you want to see who bought `Chocolade`, you can join the four tables together:
 	
`SELECT DISTINCT c.CompanyName`
`FROM customers AS c`
`JOIN orders AS o ON (c.CustomerID = o.CustomerID)`
`JOIN order_details AS od ON (o.OrderID = od.OrderID)`
`JOIN products AS p ON (od.ProductID = p.ProductID)`
`WHERE p.ProductName = 'Chocolade';`
	In Cypher, there is no need to `JOIN` tables. You can express connections as graph patterns instead:
	
	`MATCH (p:Product {productName:'Chocolade'})<-[:ORDERS]-(:Order)<-[:PURCHASED]-(c:Customer)`
	`RETURN DISTINCT c.companyName;`

Result

c.companyName
"Victuailles en stock"
"Ernst Handel"
"Antonio Moreno Taquería"
"Furia Bacalhau e Frutos do Mar"
"Around the Horn"
"Queen Cozinha"

Total spent in each product

By summing up product prices and ordered quantities, an aggregated view per product for the customer is

provided. You can use aggregation functions like `sum`, `count`, `avg`, and `max` in both SQL and Cypher.

SQL	Cypher

If you want to see **what** a company (e.g. Drachenblut Delikatessen) paid in total per product, including where they had no orders for products, you have to use `OUTER JOINS` to make sure that results are returned even if there are no matching rows in other tables:
 	
`SELECT p.Product_Name, sum(od.Unit_Price * od.Quantity) AS TotalPrice`
`FROM customers AS c`
`LEFT OUTER JOIN orders AS o ON (c.Customer_ID = o.Customer_ID)`
`LEFT OUTER JOIN order_details AS od ON (o.Order_ID = od.Order_ID)`
`LEFT OUTER JOIN products AS p ON (od.Product_ID = p.Product_ID)`
`WHERE c.Company_Name = 'Drachenblut Delikatessen'`
`GROUP BY p.Product_Name;`
	In Cypher, you need to turn the `unitPrice` property of the `ORDERS` relationship into an interger in order to do the calculation between quantity ordered and amount spent:
	
	`MATCH (p:Product)<-[o:ORDERS]-(order:Order)`
	`SET o.unitPrice = toInteger(o.unitPrice)`
	`RETURN o`
	
	Then you `MATCH` the company you want to gather information from, and use `OPTIONAL MATCH` to find their purchases, and products acquired, and `RETURN` the sum:
	
	`MATCH (c:Customer {companyName:'Drachenblut Delikatessen'})`
	`OPTIONAL MATCH (c)-[:PURCHASED]->(:Order)-[o:ORDERS]->(p:Product)`
	`RETURN p.productName, toInteger(sum(o.unitPrice * o.quantity)) AS totalPrice`

Result

p.productName	totalPrice
"Gumbär Gummibärchen"	372
"Perth Pasties"	640
"Konbu"	114
"Jack's New England Clam Chowder"	81
"Queso Cabrales"	420
"Raclette Courdavault"	1650
"Lakkalikööri"	168
"Rhönbräu Klosterbier"	72
"Gorgonzola Telino"	200

Amount of products supplied

The previous example mentioned aggregation and used the `SUM` function to find out how much a company has spent when purchasing specific products. You can use the `COUNT` function in Cypher to also count how many products are offered by a supplier, for example.

SQL	Cypher

In SQL, aggregation is explicit, so you have to provide all grouping keys again in the `GROUP BY` clause.
 	
`SELECT s.CompanyName AS Supplier, COUNT(p.ProductID) AS NumberOfProducts`
`FROM Suppliers s`
`JOIN Products p ON s.SupplierID = p.SupplierID`
`GROUP BY s.CompanyName`
`ORDER BY NumberOfProducts DESC`
`LIMIT 5;`
	In Cypher, grouping for aggregation is implicit. As soon as you use the first aggregation function, all non-aggregated columns automatically become grouping keys:
	
	`MATCH (s:Supplier)<-[:SUPPLIED_BY]-(p:Product)`
	`RETURN s.companyName AS Supplier, COUNT(p) AS NumberOfProducts`
	`ORDER BY NumberOfProducts DESC`
	`LIMIT 5`
	
	Note:
	
	Additional aggregation functions like `collect`, `percentileCont`, `stdDev` are also available.

Result

Supplier	NumberOfProducts
"Pavlova"	5
"Plutzer Lebensmittelgroßmärkte AG"	5
"Specialty Biscuits"	4
"New Orleans Cajun Delights"	4
"Grandma Kelly's Homestead"	3

List of products supplied

In Cypher you can use the `COLLECT` function to gather all nodes connected to others, but SQL doesn't have a direct equivalent to it.

SQL	Cypher

In SQL, if you want a lit of what products the suppliers offer, you use `STRING_AGG`:
 	
`SELECT s.CompanyName AS Supplier, STRING_AGG(p.ProductName, ', ' ORDER BY p.ProductName) AS ProductsSupplied`
`FROM Suppliers s`
`JOIN Products p ON s.SupplierID = p.SupplierID`
`GROUP BY s.CompanyName`
`ORDER BY s.CompanyName`
`LIMIT 5;`
	In Cypher, you can either return the structure like in SQL or use the `collect()` aggregation function, which aggregates values into a collection (list, array). This way, only one row per parent, containing an inlined collection of child values, is returned:
	
	`MATCH (s:Supplier)-[:SUPPLIES]->(p:Product)`
	`RETURN s.companyName AS Supplier, COLLECT(p.productName) AS ProductsSupplied`
	`ORDER BY Supplier`
	`LIMIT 5`
	
	Note:
	
	This also works for nested values.

Result

Supplier	ProductsSupplied
"Aux joyeux ecclésiastiques"	["Côte de Blaye", "Chartreuse verte"]
"Bigfoot Breweries"	["Sasquatch Ale", "Laughing Lumberjack Lager", "Steeleye Stout"]
"Cooperativa de Quesos 'Las Cabras'"	["Queso Manchego La Pastora", "Queso Cabrales"]
"Escargots Nouveaux"	["Escargots de Bourgogne"]
"Exotic Liquids"	["Aniseed Syrup", "Chang", "Chai"]

Defining a schema

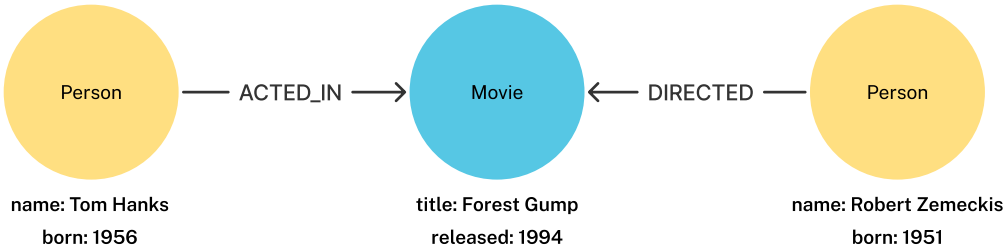
This guide explains how to define a schema by using indexes and constraints. While Neo4j is often described as *schema optional*, indexes and constraints can be added when needed, either for performance optimization or modeling benefits.

Example graph

Use the following example graph to follow the guide.

```
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

Indexes are used for speeding up data retrieval and thus improve performance. They can be added at any time.

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

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

Cypher's query planner automatically selects the appropriate index to use for your query. Only in

exceptional cases do you need to specify which index to use. See Cypher → Index hints for the Cypher planner for more information.

For example, the following query automatically uses the `example_index_1` previously created:

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

The result is:

actor
(:Person:Actor {born: 1956, name: "Tom Hanks"})
Rows: 1

Composite index

A composite index is an index on multiple properties for all nodes that have a particular label. For example, the following statement creates a composite index on all nodes labeled with `Actor` and which have both a `name` and a `born` property:

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

Note that, for example, if you had a node with the `Actor` label that did not contain the `born` property, it wouldn't be indexed.

Show indexes

You can use `SHOW INDEXES` to list indexes in a database:

```
SHOW INDEXES
```

With the current example dataset, the result is:

id	name	state	populationPercent	type	entityType	labelsOrTypes	properties	indexProvider	owningConstraint	lastRead	readCount
1	"index_1b9dcc97"	"ONLINE"	100.0	"LOOKUP"	"RELATIONSHIP"	null	null	"token-lookup-1.0"	null	null	0
0	"index_460996c0"	"ONLINE"	100.0	"LOOKUP"	"RELATIONSHIP"	null	null	"token-lookup-1.0"	null	null	0
Rows: 2											

Using constraints

Constraints are used to make sure that the data adheres to the rules of the domain. They can be property uniqueness, property existence, property type, or key constraints.

Constraint type	Description
Property uniqueness	Ensure that the combined property values are unique for all nodes with a specific label or all relationships with a specific type.
Property existence	Ensure that a property exists either for all nodes with a specific label or for all relationships with a specific type. Enterprise Edition
Property type	Ensure that a property has the required property type for all nodes with a specific label or for all relationships with a specific type. Enterprise Edition
Key	Ensure that all properties exist and that the combined property values are unique for all nodes with a specific label or all relationships with a specific type. Enterprise Edition

Constraints can be added to the database at any time. It's recommended to do it early (or even when the database is still empty), since adding constraints requires that the existing data complies with the constraint that is being added.

This is an example of how to create a constraint:

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

Show constraints

You can use `SHOW CONSTRAINTS` to find all existing constraints in a graph:

```
SHOW CONSTRAINTS
```

With the current example, this is the result you get:

id	name	type	entityType	labelsOrTypes	properties	ownedIndex	propertyType
5	"constraint_example_1"	"UNIQUENESS"	"NODE"	["Movie"]	["title"]	"constraint_example_1"	null
Rows: 1							

Keep learning

Refer to the Cypher documentation for more information on indexes and constraints:

- Search-performance indexes → See how to get quicker retrievals of exact matches between an index and the primary data storage.

- Semantic indexes → Read more on how semantic indexes capture the semantic meaning or context of the data in a database.
- Create, show, and drop indexes → Discover how to create, show, and drop indexes.
- Create, show, and drop constraints → Learn how to create, show, and drop constraints.

Updating the graph

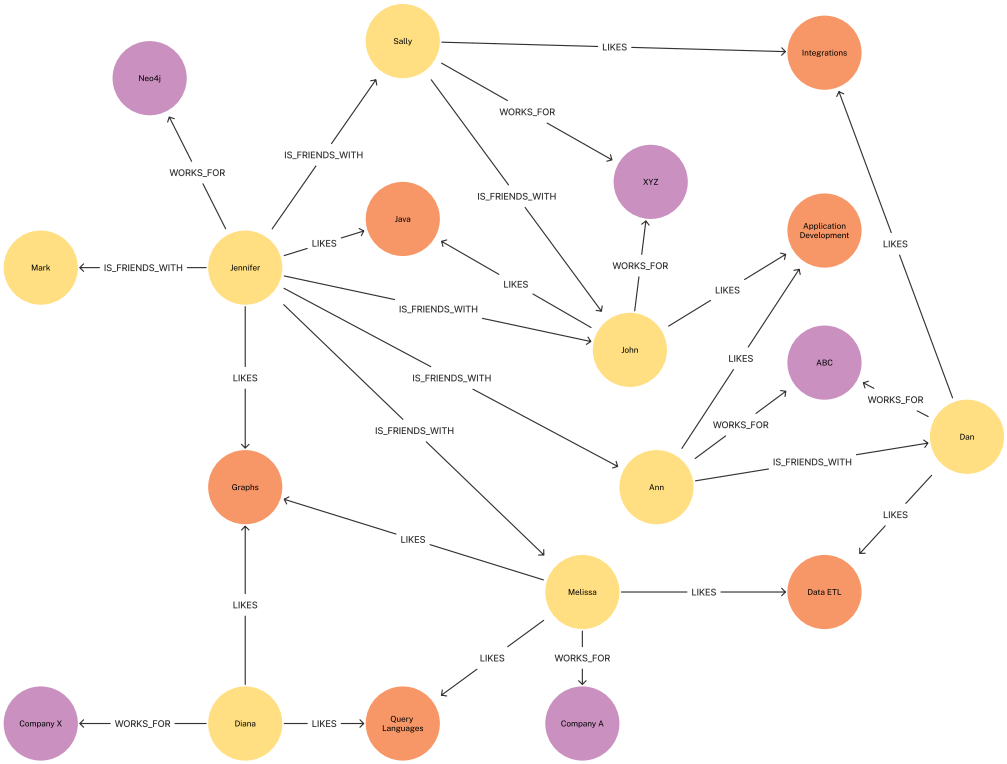
This tutorial shows how to update information in the graph by changing, removing, and adding nodes, relationships, and properties. It also addresses how to avoid duplication.

Create the dataset

After you create a free Aura instance, use the "Connect" button and select "Query". In the Cypher editor, copy and paste the following Cypher and execute the query:

```
CREATE (diana:Person {name: "Diana"})
CREATE (melissa:Person {name: "Melissa", twitter: "@melissa"})
CREATE (dan:Person {name: "Dan", twitter: "@dan", yearsExperience: 6})
CREATE (sally:Person {name: "Sally", yearsExperience: 4})
CREATE (john:Person {name: "John", yearsExperience: 5})
CREATE (jennifer:Person {name: "Jennifer", twitter: "@jennifer", yearsExperience: 5})
CREATE (joe:Person {name: "Joe"})
CREATE (mark:Person {name: "Mark", twitter: "@mark"})
CREATE (ann:Person {name: "Ann"})
CREATE (xyz:Company {name: "XYZ"})
CREATE (x:Company {name: "Company X"})
CREATE (a:Company {name: "Company A"})
CREATE (Neo4j:Company {name: "Neo4j"})
CREATE (abc:Company {name: "ABC"})
CREATE (query:Technology {type: "Query Languages"})
CREATE (etl:Technology {type: "Data ETL"})
CREATE (integrations:Technology {type: "Integrations"})
CREATE (graphs:Technology {type: "Graphs"})
CREATE (dev:Technology {type: "Application Development"})
CREATE (java:Technology {type: "Java"})
CREATE (diana)-[:LIKES]->(query)
CREATE (melissa)-[:LIKES]->(query)
CREATE (dan)-[:LIKES]->(etl)<-[:LIKES]-(melissa)
CREATE (xyz)<-[:WORKS_FOR]-(sally)-[:LIKES]->(integrations)<-[:LIKES]-(dan)
CREATE (sally)<-[:IS_FRIENDS_WITH]-(john)-[:LIKES]->(java)
CREATE (john)<-[:IS_FRIENDS_WITH]-(jennifer)-[:LIKES]->(java)
CREATE (john)-[:WORKS_FOR]->(xyz)
CREATE (sally)<-[:IS_FRIENDS_WITH]-(jennifer)-[:IS_FRIENDS_WITH]->(melissa)
CREATE (joe)-[:LIKES]->(query)
CREATE (x)<-[:WORKS_FOR]-(diana)<-[:IS_FRIENDS_WITH]-(joe)-[:IS_FRIENDS_WITH]->(mark)-[:LIKES]->(graphs)<-
[:LIKES]-(jennifer)-[:WORKS_FOR]->(Neo4j)
CREATE (ann)<-[:IS_FRIENDS_WITH]-(jennifer)-[:IS_FRIENDS_WITH]->(mark)
CREATE (john)-[:LIKES]->(dev)<-[:LIKES]-(ann)-[:IS_FRIENDS_WITH]->(dan)-[:WORKS_FOR]->(abc)
CREATE (ann)-[:WORKS_FOR]->(abc)
CREATE (a)<-[:WORKS_FOR]-(melissa)-[:LIKES]->(graphs)<-[:LIKES]-(diana)
```

You should now have a graph with 20 nodes, 31 relationships, 28 properties, and 20 labels.



Add a property to a node

If you want to add a birthday for the person named Jennifer, you can add this information as a property with this query:

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

This query does the following:

1. Find Jennifer's node with the `MATCH` clause.
2. Use `SET` to create the new property `birthday` (with syntax `variable.property`) and set its value.
3. Use `RETURN` to return Jennifer's node and so you can ensure that the information was updated correctly.

This is the tabular result:

p
(:Person {twitter: "@jennifer", birthdate: 1980-01-01, name: "Jennifer", yearsExperience: 5})
Rows: 1

Update a property

To change Jennifer's birthdate, you can use the same query from the previous example to find Jennifer's node again, and change the property value by adding a different date with the `SET` clause:

```
MATCH (p:Person {name: 'Jennifer'})
```

```
SET p.birthdate = date('1980-01-02')
RETURN p
```

This is the tabular result:

p
(:Person {twitter: "@jennifer", birthdate: 1980-01-02, name: "Jennifer", yearsExperience: 5})
Rows: 1

Delete properties

To remove a property, either use the `REMOVE` clause, or set its value to `null`. Neo4j doesn't store properties without values, so doing this removes the property from a node.

Using the `REMOVE` clause

```
MATCH (n:Person {name: 'Jennifer'})
REMOVE n.birthdate
```

Using the `SET` clause

```
MATCH (n:Person {name: 'Jennifer'})
SET n.birthdate = null
```

With both options, the result is "Set 1 property", and when you `MATCH` it, you see that the result shows the value `null`. This means that the property doesn't hold any value and doesn't exist in the graph anymore.

Add a property to a relationship

If you want to add information about Jennifer's employment, you can add a property to the `[:WORKS_FOR]` relationship. The statement is similar to the one used to add a property to a node:

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

This is the result:

rel
[:WORKS_FOR {startYear: 2018-01-01}]
Rows: 1

Delete a relationship

You can use the `DELETE` clause to delete both relationships and nodes. However, because Neo4j is ACID-compliant, you cannot delete a node if it still has relationships.

Therefore, if you want to delete a node, for example Jennifer's node, you need to first delete the relationships appended to it:

```
MATCH (j:Person {name:"Jennifer"})-[r]-(n)
DELETE r
```

Note that the arrow is undirected because you want to delete **all** relationships connected to Jennifer, no matter what direction.

The result is the following message: "Deleted 8 relationships".

Delete a node

If you have deleted all relationships in the previous step, you can delete Jennifer's node with the following query:

```
MATCH (j:Person {name:"Jennifer"})
DELETE j
```

The result is the following message: "Deleted 1 node".

It is also possible to delete a node and all its relationships with a single query using `DETACH DELETE`. This is faster but allows less control:

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

This removes all relationships attached to Mark's node and the node itself. The result is the following message: "Deleted 1 node, deleted 2 relationships".

Add new data

If you want to add new data to your graph, it's important to avoid duplication. For example, if you try to create a node that already exists in the graph using `CREATE`:

```
CREATE (ann:Person {name: 'Ann'})
RETURN ann
```

You create a duplication. If you `MATCH` Ann's node:

```
MATCH (ann:Person {name: 'Ann'})
RETURN ann
```

You see there are two nodes containing the same information:

ann
(:Person {name: "Ann"})

ann
(:Person {name: "Ann"})
Rows: 2

However, if you use `MERGE`:

```
MERGE (ann:Person {name: 'Ann'})
RETURN ann
```

Cypher returns the existing node without creating a new one:

ann
(:Person {name: "Ann"})
Rows: 1

You can try using `MERGE` to see how it works when you don't have the data in the graph:

```
MERGE (j:Person {name: 'Jack'})
RETURN j
```

You get the message "Created 1 node, set 1 property, added 1 label" and the following result:

j
(:Person {name: "Jack"})
Rows: 1

This means Jack wasn't found in the graph, thus `MERGE` adds it.

If you want to add a relationship between two existing nodes, you need to first `MATCH` them or you will create the same duplication. If you use `CREATE` to link Ann and Mark:

```
CREATE (a:Person {name: 'Ann'})-[r:IS_FRIENDS_WITH]->(m:Person {name: 'Mark'})
RETURN a, r, m
```

The result is that you duplicate Ann's node and create a new Mark node (since it was removed in the previous step) and add a new `IS_FRIENDS_WITH` relationship between them. You get the following message confirming it: "Created 2 nodes, created 1 relationship, set 2 properties, added 2 labels".

If you use `MERGE` instead:

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

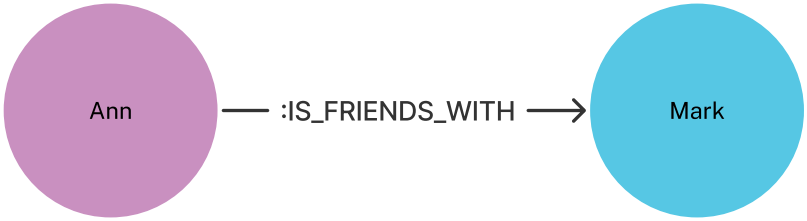
The result is more duplication. `MERGE` tries to match the **entire** pattern, and if it doesn't exist, it is created.

Even though the graph already has Ann and Mark's nodes, the pattern `(Ann)-[:IS_FRIENDS_WITH]→(Mark)` doesn't exist, so all elements are created anew.

> **Note:**
>
> If you created the previous duplication, you don't need to fix it before proceeding with the tutorial. However, if you didn't do it, you need to create a new node for Mark, as it was removed in the previous step. You can do it using either `CREATE` or `MERGE`.

In order to create the new relationship, without duplications, you need to first `MATCH` Ann and Mark's nodes and then `MERGE` (or `CREATE`) the relationship:

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



Conclusion

In this tutorial, you have seen how to update nodes, relationships, and properties. You have also seen how to delete data from the graph as well as how to avoid duplication when adding new data.

Keep learning

If you want to learn more about how to update your graph and find the best way to model your data, refer to:

- Data modeling → A complete section on how to get started with data modeling through tutorials and reference material.

Subqueries in Cypher

Recap our example graph

All of our code examples will continue with one of the graph examples we have been using before, but include some more data for the queries on this page. Below is an image of the new graph, a network of people, the companies they work for, and the technologies they like.

We have added one more `Person` node (blue) who `WORKS_FOR` a `Company` node (red) and `LIKES` a `Technology` (green) node: **Ryan** works for *Company Z* and likes **Python**. You can find this data on the right hand side of the graph.

To recap, each person could also have multiple `IS_FRIENDS_WITH` relationships to other people.

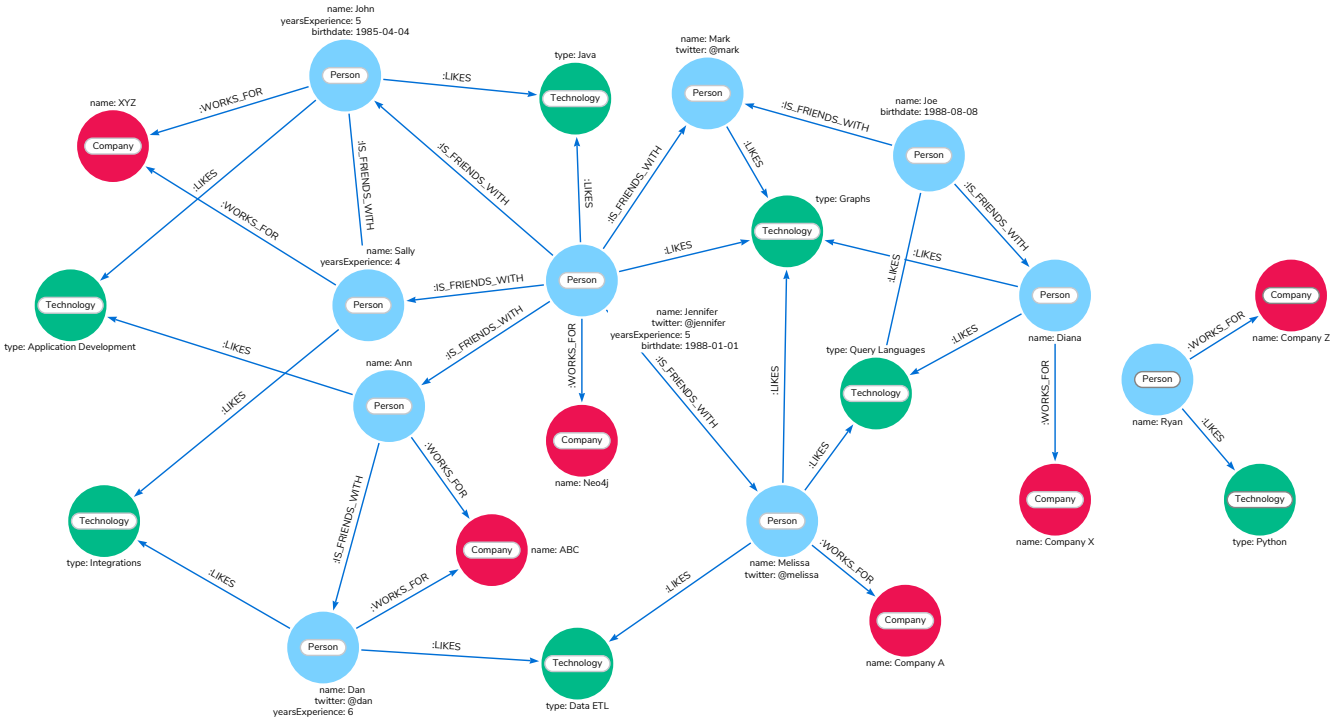


Figure 22. Network of people, the companies they work for, and the technologies they like

You can create this dataset using the following Cypher query:

```
CREATE (diana:Person {name: "Diana"})-[:LIKES]->(query:Technology {type: "Query Languages"})
CREATE (melissa:Person {name: "Melissa", twitter: "@melissa"})-[:LIKES]->(query)
CREATE (dan:Person {name: "Dan", twitter: "@dan", yearsExperience: 6})-[:LIKES]->(etl:Technology {type:
"Data ETL"})<-[:LIKES]-(melissa)
CREATE (xyz:Company {name: "XYZ"})<-[:WORKS_FOR]-(sally:Person {name: "Sally", yearsExperience: 4})-
[:LIKES]->(integrations:Technology {type: "Integrations"})<-[:LIKES]-(dan)
CREATE (sally)<-[:IS_FRIENDS_WITH]-(john:Person {name: "John", yearsExperience: 5, birthdate: "1985-04-
04"})-[:LIKES]->(java:Technology {type: "Java"})
CREATE (john)<-[:IS_FRIENDS_WITH]-(jennifer:Person {name: "Jennifer", twitter: "@jennifer",
yearsExperience: 5, birthdate: "1988-01-01"})-[:LIKES]->(java)
CREATE (john)-[:WORKS_FOR]->(xyz)
CREATE (sally)<-[:IS_FRIENDS_WITH]-(jennifer)-[:IS_FRIENDS_WITH]->(melissa)
CREATE (joe:Person {name: "Joe", birthdate: "1988-08-08"})-[:LIKES]->(query)
CREATE (mark:Person {name: "Mark", twitter: "@mark"})
CREATE (ann:Person {name: "Ann"})
CREATE (x:Company {name: "Company X"})<-[:WORKS_FOR]-(diana)<-[:IS_FRIENDS_WITH]-(joe)-[:IS_FRIENDS_WITH]
->(mark)-[:LIKES]->(graphs:Technology {type: "Graphs"})<-[:LIKES]-(jennifer)-[:WORKS_FOR]->(:Company
{name: "Neo4j"})
CREATE (ann)<-[:IS_FRIENDS_WITH]-(jennifer)-[:IS_FRIENDS_WITH]->(mark)
CREATE (john)-[:LIKES]->(:Technology {type: "Application Development"})<-[:LIKES]-(ann)-[:IS_FRIENDS_WITH]
->(dan)-[:WORKS_FOR]->(abc:Company {name: "ABC"})
CREATE (ann)-[:WORKS_FOR]->(abc)
CREATE (a:Company {name: "Company A"})<-[:WORKS_FOR]-(melissa)-[:LIKES]->(graphs)<-[:LIKES]-(diana)
CREATE (:Technology {type: "Python"})<-[:LIKES]-(:Person {name: "Ryan"})-[:WORKS_FOR]->(:Company {name:
"Company Z"})
```

An introduction to subqueries

Subqueries were introduced in Neo4j 4.0.

Go to the Cypher manual → Subqueries for detailed information on how to use them.

The following types of subqueries are possible in Neo4j:

- `EXISTS` subquery
- `COUNT` subquery
- `CALL {…}` subquery clause
- `CALL {…} IN TRANSACTIONS` subquery clause
- `COLLECT` subqueries Introduced in 5.6

The `EXISTS`, `COUNT`, and `CALL {…}` subqueries are covered in this section.

To learn more about using `CALL {…} IN TRANSACTIONS`, see the code examples in the following tutorials on how to import CSV data into a Neo4j database:

- Working with CSV files
- Import CSV data using `LOAD CSV`

The `COLLECT` subqueries were introduced in Neo4j 5.6. It is a new kind of subquery for collecting results of a subquery into a list so that further operations like `DISTINCT`, `ORDER BY`, `LIMIT`, and `SKIP` can be performed. `COLLECT` subqueries differ from `COUNT` and `EXISTS` subqueries in that the final `RETURN` clause is mandatory. The `RETURN` clause in a `COLLECT` subquery must return exactly one column.

Cypher subqueries

A subquery is a set of Cypher statements that execute within their own scope. A subquery is typically called from an outer enclosing query.

Here are some important things to know about a subquery:

- A subquery returns values referred to by the variables in the `RETURN` clause.
- A subquery cannot return variables with the same name used in the enclosing query.
- You must explicitly pass in variables from the enclosing query to a subquery.

Subqueries are demarcated by braces (`{ }`).

In the *Filtering on patterns* section of the ***Getting the correct results*** chapter, you learnt how to filter based on patterns. For example, you can write 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 you run this query in Neo4j Browser, the following graph is returned:

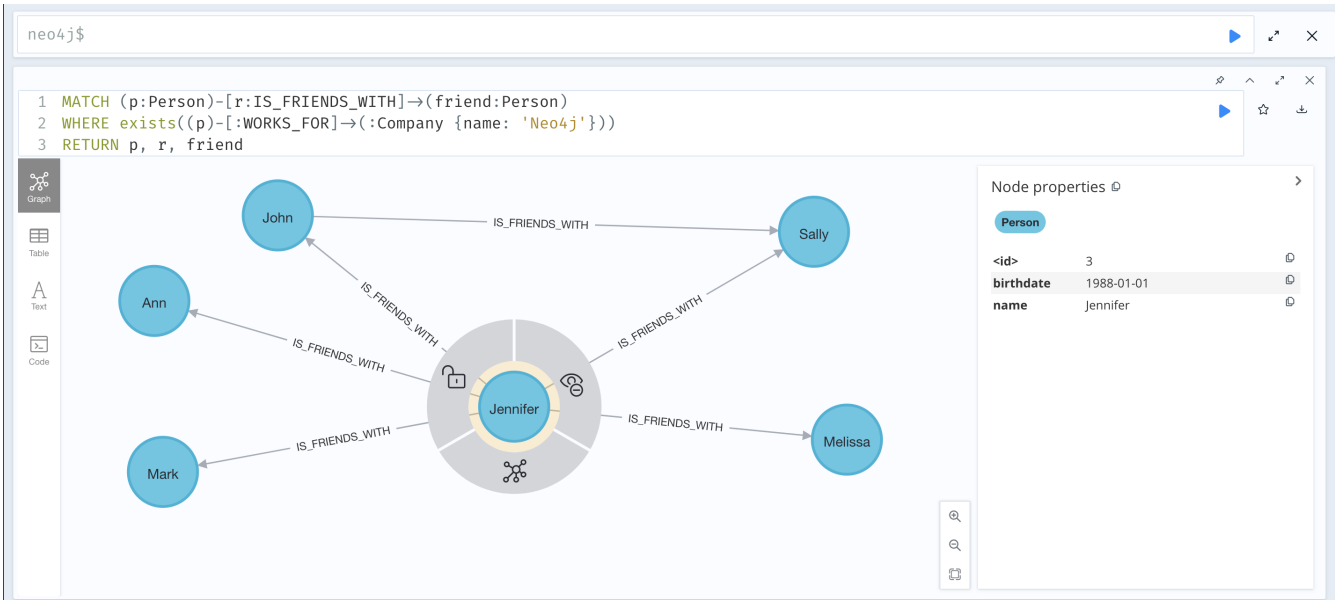


Figure 23. Output in the graph format

Cypher subqueries enable more powerful pattern filtering. Instead of using the `exists` function in the `WHERE` clause, you can use the `EXISTS` subquery. You 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
```

You will get the same result, which is nice, but so far all you've achieved is the same thing with more code!

Next, let's write a subquery that filters more powerfully than what can be achieved with the `WHERE` clause or the `exists` function alone.

Assume that:

- You want to find people who work for a company whose name starts with 'Company' and who like at least one technology that's liked by three or more people.
- You aren't interested in knowing what those technologies are.

You 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 COUNT { (t)<-[:LIKES]-() } >= 3
RETURN person.name as person, company.name AS company;
```

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

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

You can find people that like a technology, but you cannot check that at least three other 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` subquery 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 COUNT { (t)<-[:LIKES]-() } >= 3
}
RETURN person.name as person, company.name AS company;
```

Now you can successfully run the query, which returns the following results:

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

If you recall the graph visualisation from the start of this guide, **Ryan** is the only other person who works for a company which 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 three other people who like Python.

Result returning subqueries

So far you have learnt how to use subqueries to filter out results, but this doesn't fully show their power. You can also use subqueries to return results as well.

Let's say you want to write a query that finds people who like Java or have more than one friend. Apart from that, you want to return the results ordered by date of birth in descending order. This can be partially achieved using the `UNION` clause and the `COUNT` subquery:

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

If you run that query, you see the following output:

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

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

You can try another approach, where you execute each of the 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 you need to use `DISTINCT` operator in the `RETURN` clause to remove the 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 COUNT { (p)-[:IS_FRIENDS_WITH]->() } > 1
WITH collect(p) AS popularPeople, peopleWhoLikeJava
WITH popularPeople + peopleWhoLikeJava AS people

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

If you run that query, you will 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, as you have to keep passing through parts of the query to its next part.

The `CALL {…}` clause gives you the best of both worlds:

- You can use the UNION approach to run the individual queries and remove duplicates.
- You 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 COUNT { (p)-[:IS_FRIENDS_WITH]->() } > 1
    RETURN p
}
RETURN p.name AS person, p.birthdate AS dob
ORDER BY dob DESC;
```

If you run that query, you will get the following output:

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

You could extend the 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 COUNT { (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"]

You can also apply aggregation functions to the results of the 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 COUNT { (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

Summary

You have seen how to use the `EXISTS {}` subquery to write complex filtering patterns and the `CALL {}` clause to execute result-returning subqueries.

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
```

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
```

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
```

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:

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
```

Results

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

title	created	datePublished	readingTime
"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
```

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
```

Results

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

title	created	datePublished	readingTime
"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 library. Below are some resources for learning more about using Temporal types in Neo4j:

- Temporal (Date/Time) values in Cypher
- APOC Library
  - Date and Time Conversions
  - Temporal Functions
- Developer Blog: Cypher Sleuthing: Dealing with Dates, Part 1

Refining results

This guide describes how to refine the results from Cypher queries by filtering, ordering, comparing, aliasing, and more.

Example dataset

After you create a free Aura instance, use the "Connect" button and select "Query". In the Cypher editor, copy and paste the following Cypher and execute the query:

```
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 (larryCrowne:Movie {title: 'Larry Crowne', released: 2011})
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 (tom)-[:ACTED_IN {roles: ['Larry Crowne']}]->(larryCrowne)
CREATE (tom)-[:DIRECTED]->(larryCrowne)
CREATE (robert)-[:DIRECTED]->(forrestGump)

CREATE (diana:Person {name: "Diana"})
CREATE (melissa:Person {name: "Melissa", twitter: "@melissa"})
CREATE (dan:Person {name: "Dan", twitter: "@dan", yearsExperience: 6})
CREATE (sally:Person {name: "Sally", yearsExperience: 4})
CREATE (john:Person {name: "John", yearsExperience: 5})
CREATE (jennifer:Person {name: "Jennifer", twitter: "@jennifer", yearsExperience: 5})
CREATE (joe:Person {name: "Joe"})
CREATE (mark:Person {name: "Mark", twitter: "@mark"})
CREATE (ann:Person {name: "Ann"})
CREATE (xyz:Company {name: "XYZ"})
CREATE (x:Company {name: "Company X"})
CREATE (a:Company {name: "Company A"})
CREATE (Neo4j:Company {name: "Neo4j"})
CREATE (abc:Company {name: "ABC"})
CREATE (query:Technology {type: "Query Languages"})
CREATE (etl:Technology {type: "Data ETL"})
CREATE (integrations:Technology {type: "Integrations"})
CREATE (graphs:Technology {type: "Graphs"})
CREATE (dev:Technology {type: "Application Development"})
CREATE (java:Technology {type: "Java"})
```

```
CREATE (diana)-[:LIKES]->(query)
CREATE (melissa)-[:LIKES]->(query)
CREATE (dan)-[:LIKES]->(etl)<-[:LIKES]-(melissa)
CREATE (xyz)<-[:WORKS_FOR]-(sally)-[:LIKES]->(integrations)<-[:LIKES]-(dan)
CREATE (sally)<-[:IS_FRIENDS_WITH]-(john)-[:LIKES]->(java)
CREATE (john)<-[:IS_FRIENDS_WITH]-(jennifer)-[:LIKES]->(java)
CREATE (john)-[:WORKS_FOR]->(xyz)
CREATE (sally)<-[:IS_FRIENDS_WITH]-(jennifer)-[:IS_FRIENDS_WITH]->(melissa)
CREATE (joe)-[:LIKES]->(query)
CREATE (x)<-[:WORKS_FOR]-(diana)<-[:IS_FRIENDS_WITH]-(joe)-[:IS_FRIENDS_WITH]->(mark)-[:LIKES]->(graphs)<-
[:LIKES]-(jennifer)-[:WORKS_FOR {startYear: 2017}]->(Neo4j)
CREATE (ann)<-[:IS_FRIENDS_WITH]-(jennifer)-[:IS_FRIENDS_WITH]->(mark)
CREATE (john)-[:LIKES]->(dev)<-[:LIKES]-(ann)-[:IS_FRIENDS_WITH]->(dan)-[:WORKS_FOR]->(abc)
CREATE (ann)-[:WORKS_FOR]->(abc)
CREATE (a)<-[:WORKS_FOR]-(melissa)-[:LIKES]->(graphs)<-[:LIKES]-(diana)
```

Filter

You can filter results and return only a subset of data by using the `WHERE` subclause or the `FILTER` (available only in Cypher 25) clause.

Property

You can filter results by **property**. For example, to find which people in the example dataset have Twitter handles, you can use the following query:

```
MATCH (p:Person)
RETURN p.name, p.twitter
```

Since you `MATCH` **all** `Person` nodes, they are all returned, regardless of whether they have a `twitter` property or not. Out of the 12 `Person` nodes in the graph, only four (Melissa, Dan, Jennifer, Mark) contain a value for the `twitter` property. Since the query matches and returns **all** `Person` nodes, you get another eight `null` results.

In Neo4j, properties only exist (are stored) if they have a value. A `null` property value is not stored. This ensures that only valuable, necessary information is retained for your nodes and relationships.

To return only results that contain a value, you can use the `WHERE` subclause and specify that it returns only results that are not `null`:

```
MATCH (p:Person)
WHERE p.twitter IS NOT NULL
RETURN p.name, p.twitter
```

Or, alternatively, use `FILTER` with Cypher 25:

```
CYPHER 25
MATCH (p:Person)
FILTER p.twitter IS NOT NULL
RETURN p.name, p.twitter
```

Important:

If you are using any other Cypher version, you need to add `CYPHER 25` to the first line of the

query. This does **not** change the default language version, it only ensures that this query runs in Cypher 25. Otherwise, the clause `FILTER`, which is exclusive to this Cypher version, won't work.

To see a comparison between `FILTER` and `WHERE`, refer to Cypher → `FILTER`.

The result is:

p.name	p.twitter
"Melissa"	"@melissa"
"Dan"	"@dan"
"Jennifer"	"@jennifer"
"Mark"	"@mark"
Rows: 4	

Strings and partial values

You can filter results based on parts of the value stored in a node or relationship using specific predicates together with `WHERE`:

`STARTS WITH`

The `STARTS WITH` operator searches for property values that begin with a string you specify, for example, the letter "M":

```
MATCH (p:Person)
WHERE p.name STARTS WITH 'M'
RETURN p.name;
```

The result is:

p.name
"Melissa"
"Mark"
Rows: 2

`CONTAINS`

The `CONTAINS` operator checks if a specified string is part of a property value, for example, if it contains the letter "a":

```
MATCH (p:Person)
WHERE p.name CONTAINS 'a'
RETURN p.name;
```

The result is:

p.name
"Keanu Reeves"
"Tom Hanks"
"Diana"
"Melissa"
"Dan"
"Sally"
"Mark"
Rows: 7

`ENDS_WITH`

The `ENDS_WITH` operator searches for a specified string at the end of a property value, for example, a `name` that ends with "n":

```
MATCH (p:Person)
WHERE p.name ENDS WITH 'n'
RETURN p.name;
```

The result is:

p.name
"Dan"
"John"
"Ann"
Rows: 3

Alternatively, you can use regular expressions, without string operators, to find a property value. For example, if you want to find any names that include "Jo" while respecting case-sensitivity (queries are case sensitive by default):

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

p.name
"John"
"Joe"
Rows: 2

If you don't want your query to be case sensitive, you can use `toLower()` with `CONTAINS`:

```
MATCH (p:Person)
WHERE toLower(p.name) CONTAINS "jo"
RETURN p.name
```

Or, alternatively, use `toUpper()`:

```
MATCH (p:Person)
WHERE toUpper(p.name) CONTAINS "JO"
RETURN p.name
```

Both options return all `name` property values that contain "jo" regardless of the letter case.

`IN`

You can use the `IN` operator to test whether a property exists in a given list of values. For example, if any `Person` node in the graph has "1", "5" or "6" values for `yearsExperience`:

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

The result shows that no one has 1 year of experience, but three people have 5 or 6:

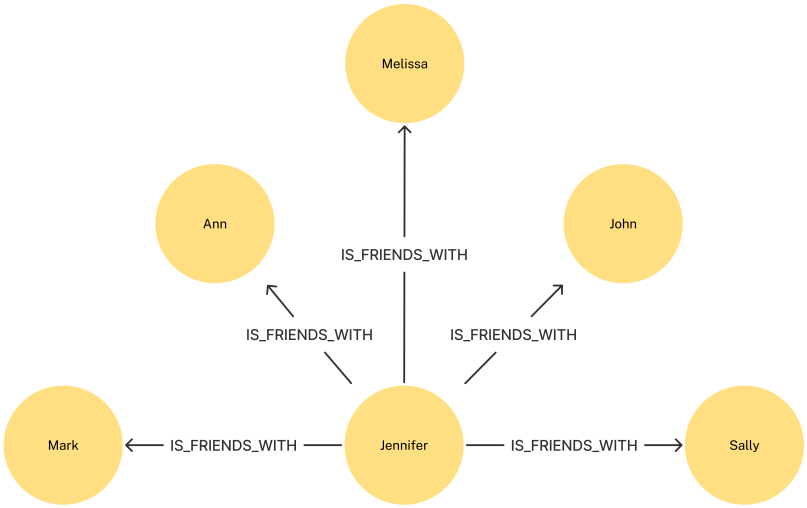
p.name	p.yearsExperience
"Dan"	5
"John"	6
"Jennifer"	5
Rows: 3	

Patterns

For example, to find people who are friends with someone who works for Neo4j, you can filter on whether there exists a `[:WORKS_FOR]->(:Company {name:'Neo4j'})` relationship from the `p:Person` in the friendship:

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

Which gives you the result:



Alternatively, you can also filter the result by excluding certain patterns from the results: For example, if you want to find which of Jennifer's friends do not work for any company, use the following query:

```
MATCH (p:Person)-[r:IS_FRIENDS_WITH]->(friend:Person)
WHERE p.name = 'Jennifer'
AND NOT exists((friend)-[:WORKS_FOR]->(:Company))
RETURN friend.name;
```

You get only "Mark" as the result.

`OPTIONAL` patterns

Sometimes you want to return results from queries even if they don't meet all the criteria. For that, you can use an `OPTIONAL MATCH` which returns null values where there is no match for the optional clause.

For example, if you search for people whose name starts with a "J" and who optionally 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;
```

You get the following result:

p.name	other.name
"Jennifer"	"Neo4j"
"John"	"XYZ"
"Joe"	null
Rows: 3	

Notice that Joe is returned because his name starts with the letter 'J', but his company's name is `null`. This 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 optional match is not found, so a `null` result is returned for `other.name`.

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

More complex patterns

You can query for patterns that have more than one relationship. For example, if you want to know who works for the same company and are also friends:

```
MATCH (p1:Person)-[:WORKS_FOR]->(company:Company)<-[:WORKS_FOR]-(p2:Person),
(p1)-[:IS_FRIENDS_WITH]->(p2)
RETURN p1.name AS people1, p2.name AS people2, company.name AS company
```

The result is:

people1	people2	company
"John"	"Sally"	"XYZ"
"Ann"	"Dan"	"ABC"
Rows: 2		

Note that the query has a comma at the end of the first line, and another pattern is added to `MATCH` on the next line. This allows you to chain different patterns together, similar to when you used `WHERE exists(<pattern>)` previously.

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.

For more information on how to query for patterns, see Cypher → Patterns.

Compare values

Equality

You can use comparisons to filter values, starting with equality:

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

The result is:

m
(:Movie {title: "The Matrix", released: 1997})
Rows: 1

Numerical

Another option is the numerical comparison to confirm the existence of values within a list.

The `WHERE` clause in the following example includes a greater-than comparison to find which movies were released after the year "2000":

```
MATCH (m:Movie)
WHERE m.released > 2000
RETURN m.title, m.released
```

The result is:

m	m.release
"Cloud Atlas"	2012
"Larry Crowne"	2011
Rows: 2	

Cypher has other mathematical operators that allow for other types of numerical comparison. See Cypher → Mathematical operators for the complete list of operators and examples.

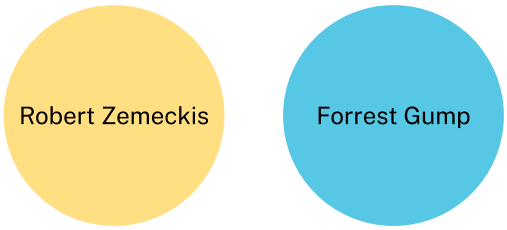
Boolean

Cypher's boolean operators allow you to combine or evaluate logical conditions.

For example, to find who in the graph has directed a movie but didn't act in it:

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

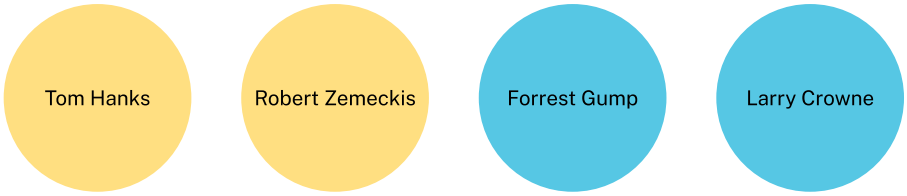
The result is:



This is because the query looks for `Person` nodes that are connected to `Movie` nodes through the `DIRECTED` relationship **only** and **not** `ACTED_IN`. If the `ACTED_IN` relationship wasn't a restriction anymore:

```
MATCH (p:Person)-[:DIRECTED]->(m)
RETURN p, m
```

Then the result includes Tom Hanks and the movie "Larry Crowne" because Tom Hanks **both** directed **and** acted in the movie:



See Cypher → Boolean operators for the complete list of operators and examples.

Range of values

You can query data within a certain range in number or date, for example, if you want to find events within a certain timeline, age values, etc.

If you want to know who has experience within the range of three to seven years:

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

The result is:

Aliasing results

By default, Cypher returns a label as the name of the column in the tabular results. However, you can use an alias to make the results more understandable by using `AS`.

For example, if you want to have the tabular result with a column titled `name` instead of the label `p.name`:

```
MATCH (p:Person)
RETURN p.name AS name
```

The result is:

name
"Keanu Reeves"
"Robert Zemeckis"
"Tom Hanks"
"Diana"
"Melissa"
"Dan"
"Sally"

name
"John"
"Jennifer"
"Joe"
"Mark"
"Ann"
Rows: 12

Avoiding duplication

For example, to find which people are connected through a `IS_FRIENDS_WITH` relationship:

```
MATCH (p:Person)-[:IS_FRIENDS_WITH]-(friend:Person)
RETURN friend.name;
```

The result is:

friend.name
"Joe"
"Jennifer"
"Ann"
"John"
"Jennifer"
"Sally"
"Jennifer"
"Melissa"
"Sally"
"John"
"Mark"
"Ann"
"Diana"
"Mark"
"Jennifer"
"Joe"
"Dan"
"Jennifer"

friend.name
Rows: 18

Note that some names appear more than once. For example, "Sally" appears twice because she is connected to two people through two `IS_FRIENDS_WITH` relationships. To avoid this, add `DISTINCT` after `RETURN`:

```
MATCH (p:Person)-[:IS_FRIENDS_WITH]->(friend:Person)
RETURN DISTINCT friend.name;
```

The result is now:

friend.name
"Sally"
"Melissa"
"John"
"Mark"
"Ann"
"Diana"
"Dan"
Rows: 7

Order

If you want to have the results ordered, you can use the clause `ORDER BY` after the `RETURN` clause. For example, to list how many years of experience each person has in descending order:

```
MATCH (p:Person)
RETURN p.yearsExperience AS yearsExperience
ORDER BY yearsExperience DESC
```

This is the result:

yearsExperience
null
null
null
null
null
null

yearsExperience
null
null
6
5
5
4
Rows: 12

Note that the first 8 rows contain `null` as result. This means that not all returned `Person` nodes have the property `yearsExperience`.

Since `null` is considered the highest possible value when sorting in descending order, they are returned before the numerical values. If the order was ascending, `null` values would appear last.

Optionally, you can avoid returning `null` values by using `WHERE` and `IS NOT NULL`:

```
MATCH (p:Person)
WHERE p.yearsExperience IS NOT NULL
RETURN p.yearsExperience AS yearsExperience
ORDER BY yearsExperience DESC
```

Limit

You can use `LIMIT` when you want to restrict the number of results returned. For example, the current example graph has twelve `Person` nodes and if you want to retrieve only three:

```
MATCH (p:Person)
RETURN p
LIMIT 3
```

The result is:

p
(:Person {born: 1964, name: "Keanu Reeves"})
(:Person {born: 1951, name: "Robert Zemeckis"})
(:Person {born: 1956, name: "Tom Hanks"})
Rows: 3

Since you didn't use `ORDER BY` to specify an order to the results, the query retrieves the first three `Person` nodes that were added to the graph. To get random results, you can add `ORDER BY rand()` to the query:

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

```
ORDER BY rand()
LIMIT 3
```

Now every time you run this query, you get three random `Person` nodes.

Aggregate

Cypher provides several aggregating functions that let you calculate a single value from a set of values.

You can use them with the `RETURN` or the `WITH` clauses to aggregate results.

Function	Description
`avg()`	Returns the average of a set of `INTEGER`, `FLOAT` or `DURATION` values.
`collect()`	Returns a list containing the values returned by an expression.
`count()`	Returns the number of values or rows.
`max()`	Returns the maximum value in a set of values.
`min()`	Returns the minimum value in a set of values.
`percentileCont()`	Returns the percentile of a value over a group using linear interpolation.
`percentileDisc()`	Returns the nearest `INTEGER` or `FLOAT` value to the given percentile over a group using a rounding method.
`stDev()`	Returns the standard deviation for the given value over a group for a sample of a population.
`stDevP()`	Returns the standard deviation for the given value over a group for an entire population.
`sum()`	Returns the sum of a set of INTEGER, FLOAT or DURATION values.

Refer to Cypher → Aggregating for the complete list of all aggregating functions and examples.

Unwind

If you have a list that you want to inspect or separate the values, Cypher offers the `UNWIND` function for that.

For example, if you want to see which people in the example graph like "Graphs" and/or "Query Languages":

```
WITH ['Graphs','Query Languages'] AS likedTech
UNWIND likedTech AS technology
MATCH (p:Person)-[r:LIKES]-(t:Technology {type: technology})
RETURN t.type, p.name AS people;
```

The result is:

t.type	people
"Graphs"	"Diana"

t.type	people
"Graphs"	"Melissa"
"Graphs"	"Jennifer"
"Graphs"	"Mark"
"Query Languages"	"Diana"
"Query Languages"	"Melissa"
"Query Languages"	"Joe"
Rows: 7	

You can use the `collect()` function to group results together. If you rewrite the query like this:

```
WITH ['Graphs','Query Languages'] AS likedTech
UNWIND likedTech AS technology
MATCH (p:Person)-[r:LIKES]-(t:Technology {type: technology})
RETURN t.type, collect(p.name) AS people;
```

The result looks tidier:

t.type	people
"Graphs"	["Diana", "Melissa", "Jennifer", "Mark"]
"Query Languages"	["Diana", "Melissa", "Joe"]
Rows: 2	

Number of items

If you have a list of elements, you can also return the number of elements in that list using the `size()` function.

For example, if you want to know how many friends each person in the graph has:

```
MATCH (p:Person)-[:IS_FRIENDS_WITH]-(friend:Person)
RETURN p.name, size(collect(friend.name)) AS numberOfFriends;
```

Note that the `collect()` function is used **inside** the `size()` function. If you didn't have it, that is, only `size(friend.name)`, what the function would do is to count how many characters the string values of the `friend.name` properties have. Since you want to know how many friends each person has, then `collect()` works as an aggregating function that aggregates all `friend.name` values and `size()` returns the amount in number.

The result is:

p.name	numberOfFriends
"Diana"	1
"Melissa"	1
"Dan"	1
"Sally"	2
"John"	2
"Jennifer"	5
"Joe"	2
"Mark"	2
"Ann"	1
Rows: 9	

Another point to note is that if you add a direction to the relationship:

```
MATCH (p:Person)-[:IS_FRIENDS_WITH]->(friend:Person)
RETURN p.name, size(collect(friend.name)) AS numberOfFriends;
```

You get a different result:

p.name	numberOfFriends
"John"	1
"Jennifer"	5
"Joe"	2
"Ann"	1
Rows:4	

This is because the `IS_FRIENDS_WITH` relationship is directed.

For example, Jennifer is connected to Mark through an outgoing `IS_FRIENDS_WITH` relationship from Jennifer to Mark (i.e. `(jennifer)-[:IS_FRIENDS_WITH]→(mark)`), not `(mark)-[:IS_FRIENDS_WITH]→(jennifer)`.

In practice, the fact that Jennifer and Mark are friends is what matters, so the relationship's direction is irrelevant. However, when adding this information to the graph, the relationship must have a direction. On the other hand, querying for it doesn't require direction.

Creating a new pattern with the opposite direction (i.e. `(mark)-[:IS_FRIENDS_WITH]→(jennifer)`) wouldn't be incorrect from a syntax perspective, but it's duplicate information when the direction is irrelevant to answer the question "who is friends with whom?". This is why a more accurate result is achieved by using a query with an undirected relationship.

Keep learning

Cypher has many other resources for refining query results. If you want to keep learning, see the following pages:

1. Composed queries → See how to use `UNION`, `WHEN`, and `NEXT` to combine queries, create linear compositions and different query branches.
2. `SKIP` → Defines from which row to start including the rows in the output.
3. Expressions → Operators and expressions that can be used to evaluate values.
4. Functions → Full list of existing Cypher functions.
5. Indexes → Strategies for speeding up data retrieval.

How to extend Cypher

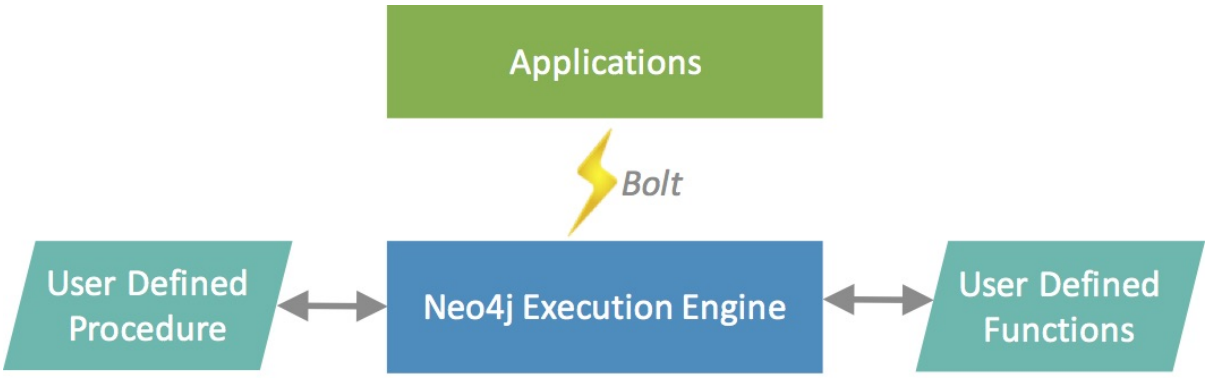
This guide explains how to create, use and deploy user-defined procedures and functions, the extension mechanism of Cypher, Neo4j's query language.

Extending Cypher

Cypher is a 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.

Cypher can be extended with *User-defined procedures and functions*, as described in Java Reference → User-defined procedures and Java Reference → User-defined functions.

Neo4j itself provides and utilizes custom procedures. Many of the monitoring, introspection and security features exposed by **Neo4j Browser** are implemented using procedures.



Procedures and functions in Neo4j

- 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 can generate, fetch, or compute data to make it available to later processing steps in your

Cypher query.

- To call a procedure deployed in the database, use the `CALL` clause (for more details, see Cypher Manual → CALL procedure).

Listing and using functions and procedures in Neo4j

Neo4j comes with a number of built-in procedures. To learn more about them, see Operations Manual → Procedures.

To list all available functions and procedures in DBMS, use the following Cypher commands:

- `SHOW FUNCTIONS`
- `SHOW PROCEDURES`

You can refer to the Cypher Cheat Sheet to get a quick reference on how to use these commands.

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

Like other Cypher administrative commands, `SHOW PROCEDURES` can be used with a subset of Cypher clauses, as shown below where we filter by 'db.' prefix and return the results ordered by name.

```
SHOW PROCEDURES
YIELD name, signature, description as text
WHERE name STARTS WITH 'db.'
RETURN * ORDER BY name ASC
```

Below you can find one more example on how to group available procedures by a chosen category, for example by package.

```
SHOW PROCEDURES
YIELD name, signature, description
RETURN split(name,".")[0..-1] AS package,  count(*) AS count,
       collect(split(name,".")[-1]) AS names
ORDER BY count DESC
```

Set of the available procedures depends on the type of installation you have and your configuration settings. The result can be the following:

package	count	names
["dbms"]	20	["checkConfigValue", "components", "info",...]
["db"]	16	["awaitIndex", "awaitIndexes", "checkpoint",...]
["db","stats"]	6	["clear", "collect", "retrieve",...]
["dbms", "cluster"]	6	["checkConnectivity", "cordonServer", "protocols",...]

package	count	names
["db", "index", "fulltext"]	4	["awaitEventuallyConsistentIndexRefresh", "listAvailableAnalyzers",...]

User-defined functions are written in Java, deployed into the database and are called in the same way as any other Cypher functions. There are two main types of functions that can be developed and used:

- user-defined scalar functions,
- user-defined aggregation functions.

For more details, see Cypher Manual → User-defined functions.

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

Also take a look at the procedure section in the Neo4j Java Reference.

Deploying procedures and functions

If you build your own procedures or download 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.

Warning:

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.

See Operations Manual → Securing extensions for best practices on how to ensure the security of these additions.

Important:

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.
In Neo4j AuraDB, the set of available procedures and functions is limited to the built-in ones and a subset of the APOC Core library.

Procedure and function gallery

The APOC Core library offers you a set of useful procedures on Cypher to increase functionality in areas of data integration, graph algorithms and data conversion.

For example, 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

In our Neo4j Labs projects, you can find a 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

Note:

Community and Neo4j Labs projects are not supported officially and we don't provide any SLAs or guarantees around backwards compatibility and deprecation.

Developing your own procedures and functions

You can find details on writing and testing procedures in the Neo4j Java Reference.

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`

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 Cheat Sheet
- 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

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

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

Work with data

What is graph data modeling?

Data modeling is a practice that defines the logic of queries and the structure of the data in storage. A well-designed model is the key to leveraging the strengths of a graph database as it improves query performance, supports flexible queries, and optimizes storage.

In summary, the process of creating a data model includes the following:

1. Understand the domain and define specific use cases (questions) for the application.
2. Develop an initial graph data model by extracting entities and decide how they relate to each other.
3. Test the use cases against the initial data model.
4. Create the graph with test data using Cypher.
5. Test the use cases, including performance against the graph.
6. Refactor the graph data model due to changes in the key use cases or for performance reasons.

For a full tutorial, refer to Create a data model.

Keep learning

For a more hands-on approach to data modeling, try the following resources:

- GraphAcademy: Data Modeling Fundamentals: enroll to an interactive course.
- From relational to graph: learn how to adapt data from a relational to a graph data model.
- Data modeling tools: see a list of tools you can use to create your data model.
- Data modeling tips: check tips on how to improve your data modeling skills.
- Modeling designs: see examples of data modeling designs that can be used as strategy for your project.
- Neo4j GraphGists: find examples of graph data modeling shared by the Neo4j community.

Glossary

label

Marks a node as a member of a named and indexed subset. A node may be assigned zero or more labels.

labels

A label marks a node as a member of a named and indexed subset. A node may be assigned zero or more labels.

node

A node represents an entity or discrete object in your graph data model. Nodes can be connected by relationships, hold data in properties, and are classified by labels.

nodes

A node represents an entity or discrete object in your graph data model. Nodes can be connected by relationships, hold data in properties, and are classified by labels.

relationship

A relationship represents a connection between nodes in your graph data model. Relationships connect a source node to a target node, hold data in properties, and are classified by type.

relationships

A relationship represents a connection between nodes in your graph data model. Relationships connect a source node to a target node, hold data in properties, and are classified by type.

property

Properties are key-value pairs that are used for storing data on nodes and relationships.

properties

Properties are key-value pairs that are used for storing data on nodes and relationships.

cluster

A Neo4j DBMS that spans multiple servers working together to increase fault tolerance and/or read scalability. Databases on a cluster may be configured to replicate across servers in the cluster thus achieving read scalability or high availability.

clusters

A Neo4j DBMS that spans multiple servers working together to increase fault tolerance and/or read scalability. Databases on a cluster may be configured to replicate across servers in the cluster thus achieving read scalability or high availability.

graph

A logical representation of a set of nodes where some pairs are connected by relationships.

graphs

A logical representation of a set of nodes where some pairs are connected by relationships.

schema

The prescribed property existence and datatypes for nodes and relationships.

schemas

The prescribed property existence and datatypes for nodes and relationships.

[[database schema]]database schema

The prescribed property existence and datatypes for nodes and relationships.

indexes

Data structure that improves read performance of a database. Read more about supported categories of indexes.

indexed

Data structure that improves read performance of a database. Read more about supported categories of indexes.

constraints

Constraints are sets of data modeling rules that ensure the data is consistent and reliable. See what constraints are available in Cypher.

data model

A data model defines how information is organized in a database. A good data model will make querying and understanding your data easier. In Neo4j, the data models have a graph structure.

data models

A data model defines how information is organized in a database. A good data model will make querying and understanding your data easier. In Neo4j, the data models have a graph structure.

Tutorial: Create a graph data model

This tutorial is designed to help you understand how to model your data based on what you intend to use it for. You will use the Movies example dataset as the main resource.

Tip:

For an interactive course on the fundamentals of data modeling, see GraphAcademy.

Define the domain

In this tutorial, you will use the Movies example dataset, the domain includes movies, people who acted or directed movies, and users who rated movies. It is in the connections (relationships) between these entities that you find insights about your domain.

Define the use case

With the domain defined, you need to identify your application use cases. In other words, what questions are you trying to answer?

You can make a **list of questions** to help you identify the application use cases. The questions will help you define what you need from the application, and what data must be included in the graph.

For this tutorial, your application should be able to answer these questions:

- Which people acted in a movie?
- Which person directed a movie?
- Which movies did a person act in?
- How many users rated a movie?
- Who was the youngest person to act in a movie?
- Which role did a person play in a movie?
- Which is the highest rated movie in a particular year according to imDB?
- Which drama movies did an actor act in?
- Which users gave a movie a rating of 5?

Define the purpose

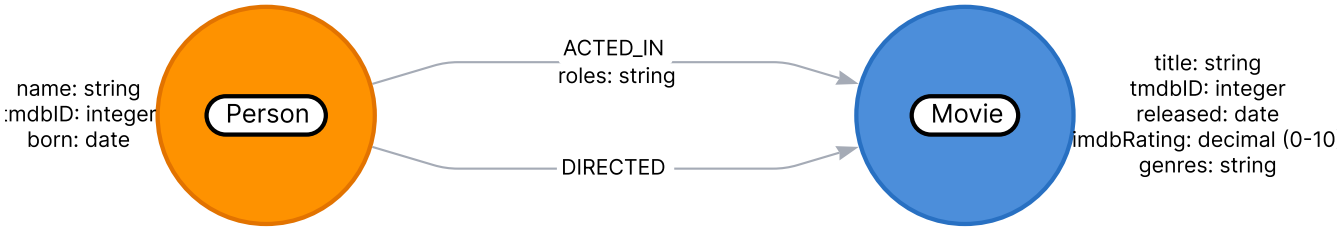
When designing a graph data model for an application, you may need both a data model and an instance model.

Data model

The data model describes the nodes and relationships in the domain and includes labels, types, and properties. It doesn't contain any data but shows what information might be needed to answer the use

cases.

At this stage, you can opt to use no-code tools to visualize your plan. With Arrows.app, for example, you can draft a data model that includes node labels, relationship types, and properties:



With your example domain and initial questions in mind, you can list the information you need to have in your first model:

- Differentiation between a person who acted in a movie, who directed a movie, and who rated a movie.
- What ratings were given, how many there are, and when they were submitted.
- Which role an actor played in a movie and what their age is.
- The genres of the movies.
- Etc.

Note that in the model, labels, relationship types, and property keys follow a certain syntax. In Cypher, these are called identifiers and they are case-sensitive, as are string values.

See the Cypher style guide for more information. While not mandatory, it is recommended that:

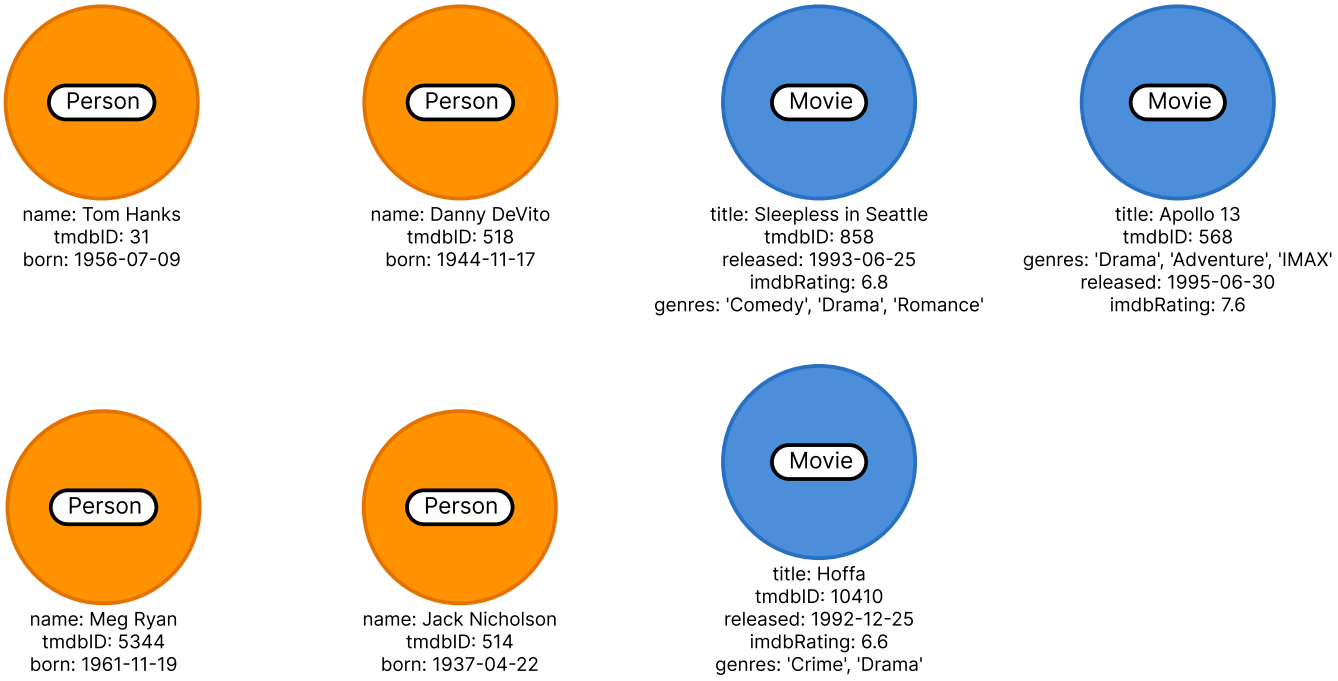
- Labels are capitalized and should be CamelCase (e.g., `Person`, `Movie`, `ImdbUser`).
- Relationship types are written with all capital letters and an underscore character as a separator (e.g., `DIRECTED`, `ACTED_IN`).
- Property keys for nodes or relationships are **not** capitalized and can be camelCase (e.g. `name`, `userID`).

> **Note:**
>
> At this stage of creating your initial model, focus on the high-level design of your model, i.e. how your entities connect. The Define entities step describes a more detailed view on how to allocate certain information in a graph (e.g., as a node, relationship, property, etc).

Instance model

An instance model is a representation of the data that is stored and processed in the actual model. You can use an instance model to test against your use cases.

To create an instance model, you need to have some sample data and load it to a deployment of your choice. The current example is a small but representative dataset:




```
CREATE (Apollo13:Movie {title: 'Apollo 13', tmdbID: 568, released: '1995-06-30', imdbRating: 7.6, genres:
['Drama', 'Adventure', 'IMAX']})
CREATE (TomH:Person {name: 'Tom Hanks', tmdbID: 31, born: '1956-07-09'})
CREATE (MegR:Person {name: 'Meg Ryan', tmdbID: 5344, born: '1961-11-19'})
CREATE (DannyD:Person {name: 'Danny DeVito', tmdbID: 518, born: '1944-11-17'})
CREATE (JackN:Person {name: 'Jack Nicholson', tmdbID: 514, born: '1937-04-22'})
CREATE (SleeplessInSeattle:Movie {title: 'Sleepless in Seattle', tmdbID: 858, released: '1993-06-25',
imdbRating: 6.8, genres: ['Comedy', 'Drama', 'Romance']})
CREATE (Hoffa:Movie {title: 'Hoffa', tmdbID: 10410, released: '1992-12-25', imdbRating: 6.6, genres:
['Crime', 'Drama']})
```

> **Note:**
>
> The data used here can be found in the Movies example dataset, also available in Browser and Aura guides. However, in order to practice data modeling, it is recommended that you add the data manually using Cypher.

Define entities

An instance model helps you preview how the data will be stored as nodes, relationships, and properties. The next step is to refine your model with more details.

Labels

The dominant nouns in your application use case are represented as nodes in your model and can be used as **node labels**. For example:

- Which person acted in a movie?
- How many users rated a movie?

The nodes in your initial model are thus **Person**, **Movie**, and **User**. Note that creating a model is an iterative process and, after refactoring, your model may look different.

> **Note:**
>
> Read more about labels in Graph database concepts.

Node properties

You can use node properties to:

Anchor (where to begin the query)

```
MATCH (p:Person {name: 'Tom Hanks'})-[:ACTED_IN]-(m:Movie)
RETURN m
```

Traverse the graph (navigation)

```
MATCH (p:Person)-[:ACTED_IN]-(m:Movie {title: 'Apollo 13'})-[:RATED]-(u:User)
RETURN p,u
```

Return data from the query

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

With these properties, it is easier to visualize what you need from the graph to answer the use case questions. For example:

Use case	Steps required	Query example
Which people acted in a movie?	• Retrieve a movie by its **title**. • Return the **names** of the actors.	`MATCH (m:Movie {title:'Hoffa'})<-[r:ACTED_IN]-(p:Person)` `RETURN p.name`
Which person directed a movie?	• Retrieve a movie by its **title**. • Return the **name** of the director.	`MATCH (m:Movie {title:'Hoffa'})<-[r:DIRECTED]-(p:Person)` `RETURN p.name`
Which movies did a person act in?	• Retrieve a person by their **name**. • Return the **titles** of the movies.	`MATCH (p:Person {name:'Tom Hanks'})-[:ACTED_IN]->(m:Movie)` `RETURN m.title`

Who was the youngest person to act in a movie?	• Retrieve a movie by its **title**.
	• Evaluate the **ages** of the actors.
	• Return the **name** of the actor with the lowest age.
		`MATCH (m:Movie {title:'Sleepless in Seattle'})<-[r:ACTED_IN]-(p:Person)`
		`RETURN p.name, p.born`
		`ORDER BY p.born DESC`
		`LIMIT 1`

What is the highest rated movie in a particular year according to imDB?	• Retrieve all movies **released** in a particular year.
	• Evaluate the **imDB ratings**.
	• Return the movie **title** for the movie with the highest rating.
		`MATCH (m:Movie {release:date('1995')})`
		`RETURN m.title, m.imdbRating`
		`ORDER BY m.imdbRating DESC`
		`LIMIT 1`

Unique identifiers

In Cypher, it is possible to create two different nodes with the exact same data. However, from a data management and model perspective, different nodes should contain different data. You can use **unique identifiers** to make sure that every node is a separate and distinguished entity.

In the initial instance model, these are the properties set for the `Movies` nodes:

- `Movie.title` (string)
- `Movie.tmdbID` (integer)
- `Movie.released` (date)
- `Movie.imdbRating` (decimal between 0-10)
- `Movie.genres` (list of strings)

And for the `Person` nodes:

- `Person.name` (string)
- `Person.tmdbID` (integer)
- `Person.born` (date)

The `Movie` node property `tmdbID` is a good example of a unique identifier, as there might be different movies with the same title in the database, but the property will be different and thus function as a unique identifier.

Tip:

It is strongly suggested that you enforce unique identifiers by using uniqueness constrains. Read more about this topic in Cypher → Create property uniqueness constraints.

Relationships

Relationships are connections between nodes, and these connections are the verbs in your use cases:

- Which person acted in a movie?
- Which person directed a movie?

At a glance, connections seem straightforward, but their micro- and macro-design are arguably the most critical factors in graph performance. To get started, thinking of relationships from the perspective that "connections are verbs" works well, but there are other important considerations that you will learn as you advance with your model.

Naming

It is important to choose good names (types) for the relationships in the graph and be as specific as possible in order to allow Neo4j to traverse only relevant connections.

For example, instead of connecting two nodes with a generic relationship type (e.g. `CONNECTED_TO`), prefer

to be more specific and intuitive about the way those entities connect.

For this tutorial sample, you could define relationships as:

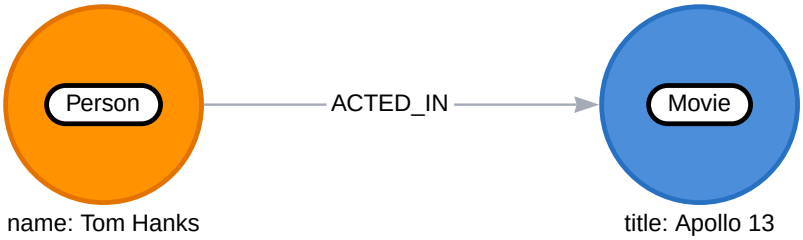
- `ACTED_IN`
- `DIRECTED`

With these options, you can already plan the direction of the relationships.

Relationship direction

All relationships must have a direction. When created, relationships need to specify their direction explicity or be inferred by the left-to-right order of the pattern.

In the example use cases, the `ACTED_IN` relationship must be created to go from a `Person` node to a `Movie` node:

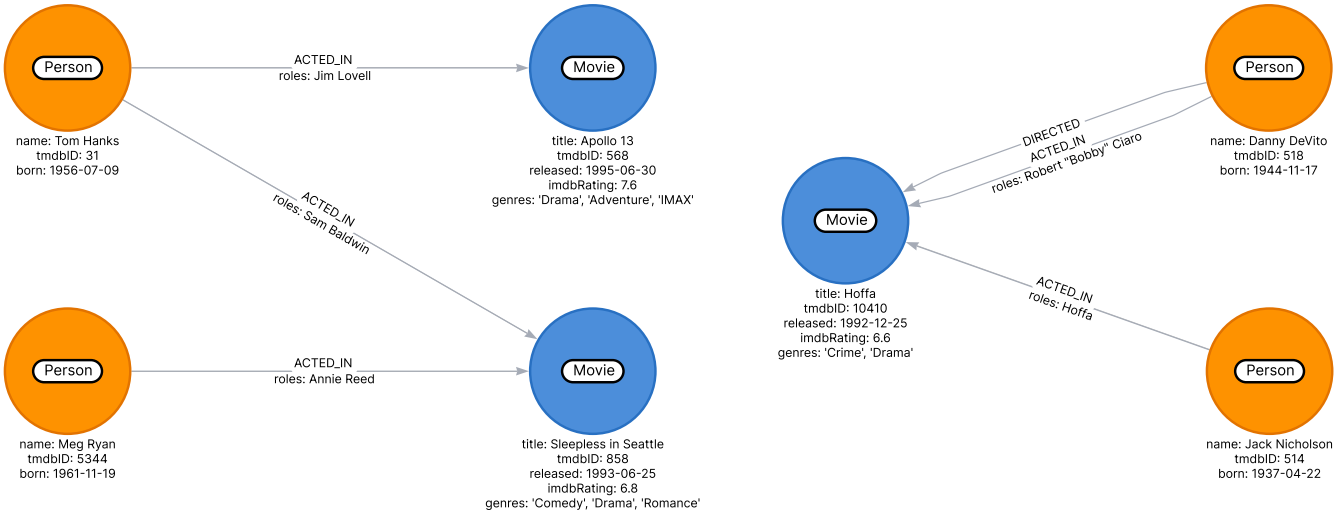


To add all `ACTED_IN` and `DIRECTED` relationships, you can use this statement:

```
MATCH (TomH:Person {name:'Tom Hanks'})
MATCH (MegR:Person {name:'Meg Ryan'})
MATCH (DannyD:Person {name:'Danny DeVitto'})
MATCH (JackN:Person {name:'Jack Nicholson'})
MATCH (Apollo13:Movie {title:'Apollo 13'})
MATCH (SleeplessInSeattle:Movie {title:'Sleepless in Seattle'})
MATCH (Hoffa:Movie {title:'Hoffa'})

MERGE (TomH)-[:ACTED_IN]->(Apollo13)
MERGE (TomH)-[:ACTED_IN]->(SleeplessInSeattle)
MERGE (MegR)-[:ACTED_IN]->(SleeplessInSeattle)
MERGE (DannyD)-[:ACTED_IN]->(Hoffa)
MERGE (DannyD)-[:DIRECTED]->(Hoffa)
MERGE (JackN)-[:ACTED_IN]->(Hoffa)
```

And your graph should now look like this:

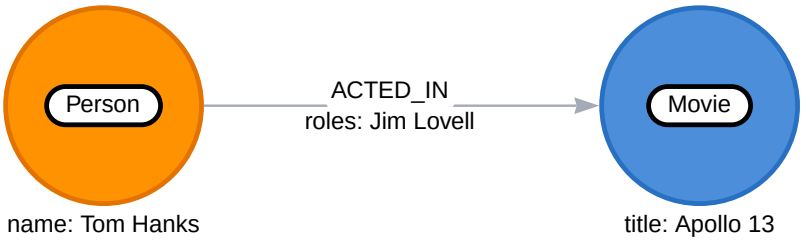


> **Tip:**
>
> You can always use the query `MATCH (n) RETURN n` to see what your graph looks like.

Relationship properties

Properties for a relationship are used to enrich how two nodes are related. When you need to know **how** two nodes are related and not just that they are related, you can use relationship properties to further define the relationship.

The example question "Which role did a person play in a movie?" can be asked with the help of the property `roles` in the `ACTED_IN` relationship:



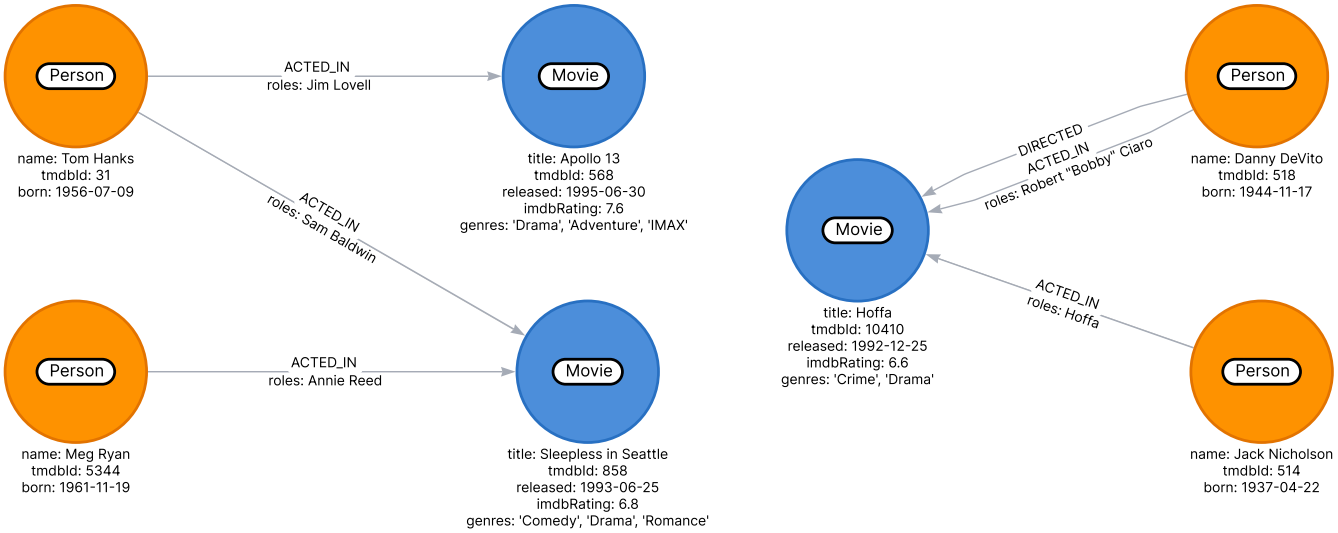
Note that the information about roles needs to be added to the graph before being retrieved. You can use this Cypher statement for that:

```
MERGE (TomH)-[:ACTED_IN {roles:'Jim Lovell'}]->(Apollo13)
MERGE (TomH)-[:ACTED_IN {roles:'Sam Baldwin'}]->(SleeplessInSeattle)
MERGE (MegR)-[:ACTED_IN {roles:'Annie Reed'}]->(SleeplessInSeattle)
MERGE (DannyD)-[:ACTED_IN {roles:'Robert "Bobby" Ciaro'}]->(Hoffa)
MERGE (JackN)-[:ACTED_IN {roles:'Hoffa'}]->(Hoffa)
```

Then, in order to find which role Tom Hanks played in Apollo 13, you use the following statement:

```
MATCH (p:Person {name:'Tom Hanks'})-[r:ACTED_IN]->(m:Movie {title:'Apollo 13'})
RETURN r.roles
```

With the addition of the new relationship property, your graph should now look like this:

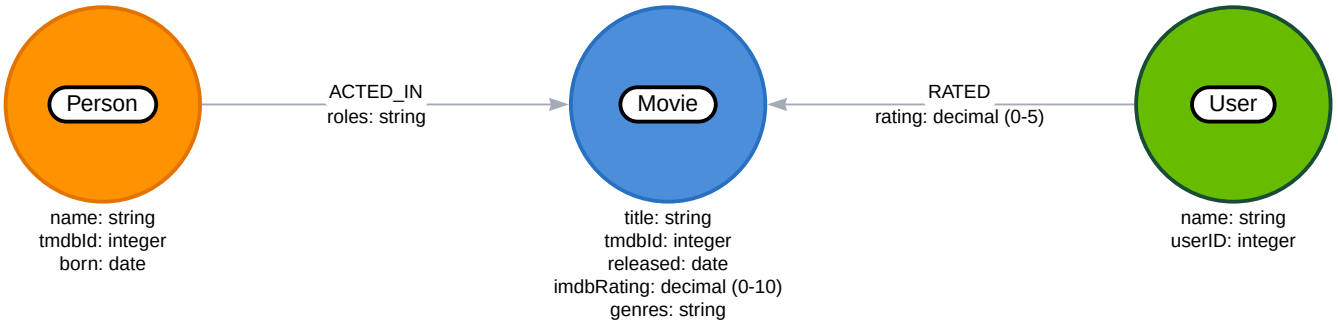


Add more data

Now that you have created the first connections between the nodes, it's time to add more information to the graph. This way, you can answer more questions, such as:

- How many users rated a movie?
- Which users gave a movie a rating of 5?

To answer these questions, you need information about users and their ratings in your graph, which means a change in your data model. Note that, with the addition of new data such as the property `roles` in the `ACTED_IN` relationship, your initial data model has already been updated along the way:



You can start by adding the users to your graph:

```
MERGE (Sandy:User {name: 'Sandy Jones', userID: 1})
MERGE (Clinton:User {name: 'Clinton Spencer, userID: 2'})
```

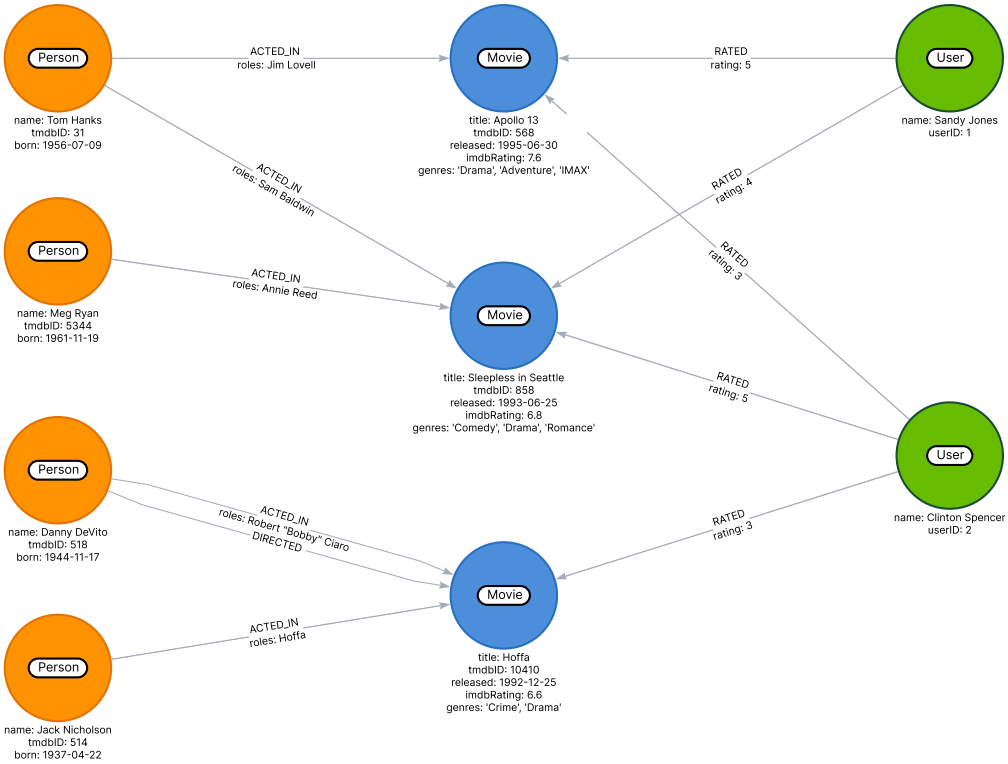
> **Note:**
>
> While it is possible to add user information as a `Person` node, it is advisable to separate them from actors and directors as they relate to the `Movie` nodes differently.

Then, connect the `User` nodes to the `Movie` nodes through a `RATED` relationship which contains the `rating` property:

```
MERGE (Sandy)-[:RATED {rating:5}]->(Apollo13)
```

```
MERGE (Sandy)-[:RATED {rating:4}]->(SleeplessInSeattle)
MERGE (Clinton)-[:RATED {rating:3}]->(Apollo13)
MERGE (Clinton)-[:RATED {rating:3}]->(SleeplessInSeattle)
MERGE (Clinton)-[:RATED {rating:3}]->(Hoffa)
```

Your graph should now look like this:



Test the model

After populating the graph to implement the data model with a small set of test data, you should now test it to ensure that it satisfies every use case.

For example, if you want to test the use case "Which people acted in a movie?", you can run the following query:

```
MATCH (p:Person)-[:ACTED_IN]-(m:Movie)
WHERE m.title = 'Sleepless in Seattle'
RETURN p.name
```

This is just a simple example of testing. As you go through the use cases, you may think of more data to be added to the graph in order to complete the testing.

Additionally, make sure that the Cypher statements used to test the use cases are correct. A query written incorrectly could lead to the assumption that the data model has failed.

For example, using an incorrect node label in a test may lead you to believe that the data doesn't exist in the graph.

At this point, you can also start considering the scalability of your graph and how performant it would be if you write the same queries in a graph with millions of nodes and relationships.

Refactoring

The next step, refactoring, is about making adjustments after you are finished testing your graph. Refer to Tutorial: Refactor a graph data model for instructions.

Tutorial: Refactor a graph data model

Refactoring is the process of changing the data model and the graph. The main reasons why you would need to refactor a data model include:

- The graph as modeled does not cover all of the use cases.
- A new use case has come up.
- The Cypher for the use cases does not perform optimally, especially when the graph scales.

In order to address these demands, this tutorial guides you through the design, implementation, and testing of a refactored data model with updated Cypher.

Pre-requisites

This tutorial is a follow-up to Tutorial: Create a graph data model. You need the data model created there before proceeding.

Optionally, you can also create it from scratch now. Choose your preferred deployment method and use this code to add the data:

```
CREATE (Apollo13:Movie {title: 'Apollo 13', tmdbID: 568, released: '1995-06-30', imdbRating: 7.6, genres:
['Drama', 'Adventure', 'IMAX']})
CREATE (TomH:Person {name: 'Tom Hanks', tmdbID: 31, born: '1956-07-09'})
CREATE (MegR:Person {name: 'Meg Ryan', tmdbID: 5344, born: '1961-11-19'})
CREATE (DannyD:Person {name: 'Danny DeVito', tmdbID: 518, born: '1944-11-17'})
CREATE (JackN:Person {name: 'Jack Nicholson', tmdbID: 514, born: '1937-04-22'})
CREATE (SleeplessInSeattle:Movie {title: 'Sleepless in Seattle', tmdbID: 858, released: '1993-06-25',
imdbRating: 6.8, genres: ['Comedy', 'Drama', 'Romance']})
CREATE (Hoffa:Movie {title: 'Hoffa', tmdbID: 10410, released: '1992-12-25', imdbRating: 6.6, genres:
['Crime', 'Drama']})

MERGE (TomH)-[:ACTED_IN {roles:'Jim Lovell'}]->(Apollo13)
MERGE (TomH)-[:ACTED_IN {roles:'Sam Baldwin'}]->(SleeplessInSeattle)
MERGE (MegR)-[:ACTED_IN {roles:'Annie Reed'}]->(SleeplessInSeattle)
MERGE (DannyD)-[:DIRECTED]->(Hoffa)
MERGE (DannyD)-[:ACTED_IN {roles:'Robert "Bobby" Ciaro'}]->(Hoffa)
MERGE (JackN)-[:ACTED_IN {roles:'Hoffa'}]->(Hoffa)

CREATE (Sandy:User {name: 'Sandy Jones', userID: 1})
CREATE (Clinton:User {name: 'Clinton Spencer', userID: 2})

MERGE (Sandy)-[:RATED {rating:5}]->(Apollo13)
MERGE (Sandy)-[:RATED {rating:4}]->(SleeplessInSeattle)
MERGE (Clinton)-[:RATED {rating:3}]->(Apollo13)
MERGE (Clinton)-[:RATED {rating:3}]->(SleeplessInSeattle)
MERGE (Clinton)-[:RATED {rating:3}]->(Hoffa)
```

Remaining or new use cases

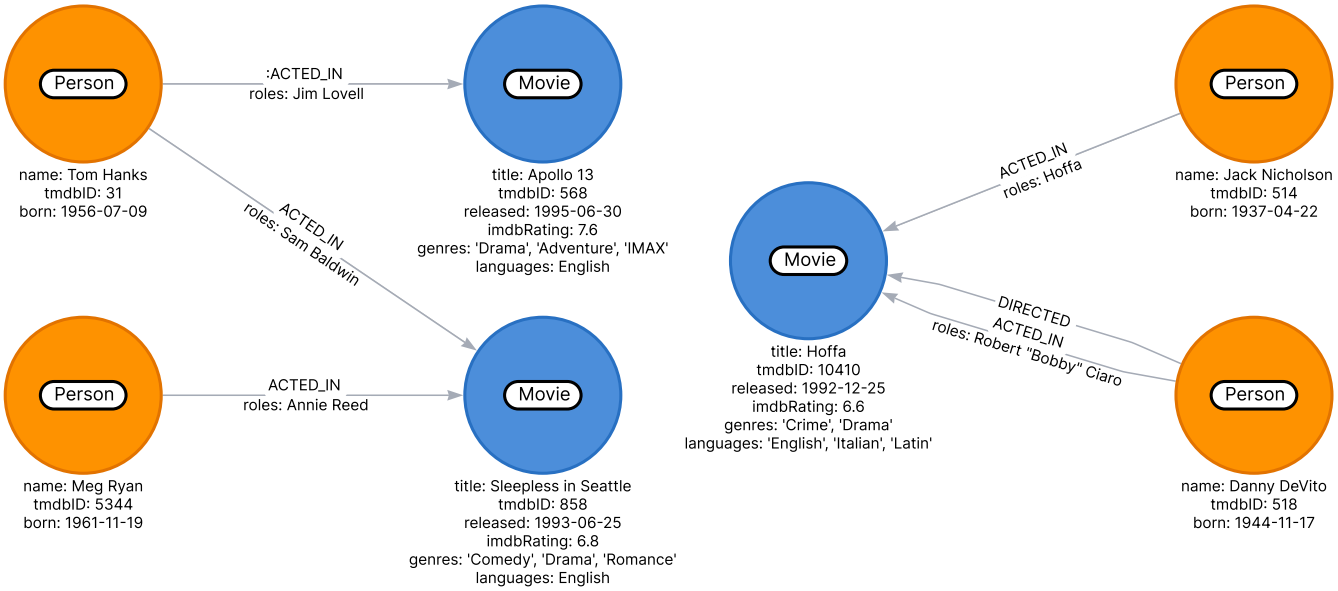
Suppose that you want to know what movies are available in a particular language.

To answer that question, you need to first add this information to the graph the same way you did when adding information about users in the Tutorial: Create a graph data model. However, adding new data poses the risk of duplication, which, in turn, affects the performance of your graph.

To illustrate this situation, add the new property `languages` to the 'Movie' nodes and its corresponding values:

```
MATCH (Apollo13:Movie {title:'Apollo 13'})
MATCH (SleeplessInSeattle:Movie {title:'Sleepless in Seattle'})
MATCH (Hoffa:Movie {title:'Hoffa'})
SET Apollo13.languages = ['English']
SET SleeplessInSeattle.languages = ['English']
SET Hoffa.languages = ['English', 'Italian', 'Latin']
```

Your updated graph should look like this:



If you want to retrieve all movies in English, execute this query:

```
MATCH (m:Movie)
WHERE 'English' IN m.languages
RETURN m.title
```

The result for this query answers the question and returns the movies "Apollo 13", "Sleepless in Seattle", and "Hoffa":

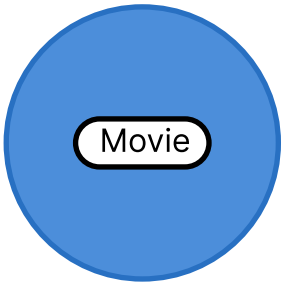
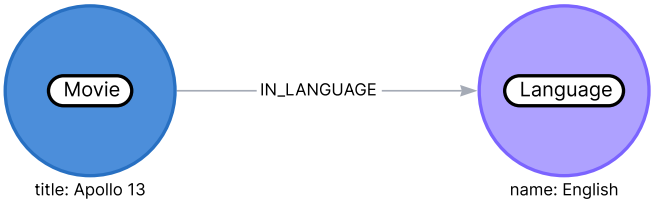
Result

m.title
"Apollo 13"
"Sleepless in Seattle"
"Hoffa"

This query retrieves all `Movie` nodes and then test whether the `languages` property contains the value `English`. This isn't wrong, but as the graph scales, you may encounter two issues:

- **In order to perform the query, all `Movie` nodes must be retrieved** → As the graph scales, the performance of such queries is diminished by modelling your data this way. The alternative to avoid this would be to create an index.
- **The same property value of the `language` property is duplicated across many `Movie` nodes (in this case, all of them)** → If many nodes share a same property value, it is a sign that this property value can be made into a new entity, like a node or a relationship, for example.

The solution for these issues is to refactor the property `languages` into a node and connect it to the `Movie` nodes with a new relationship.

Now	Refactored
 title: Apollo 13 languages: English	

Eliminating duplicated data

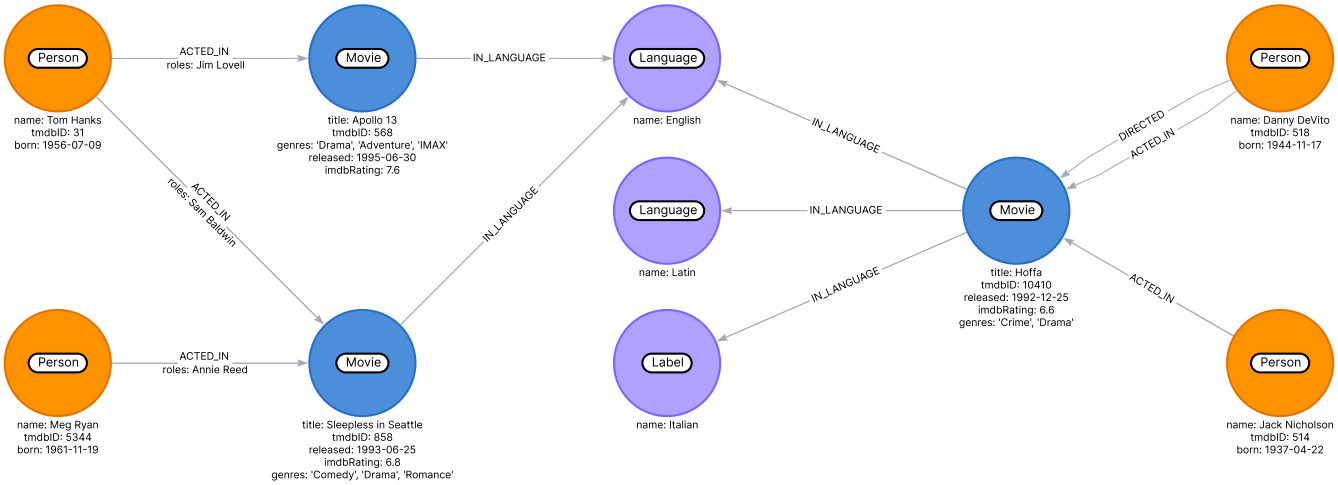
In order to refactor the node property `languages` into a node, you can use the following query:

```
MATCH (m:Movie)
WITH m, m.languages AS languages
UNWIND languages AS language
MERGE (l:Language {name: language})
MERGE (m)-[:IN_LANGUAGE]->(l)
REMOVE m.languages
```

By breaking down the query, this is what you should be doing:

1. `UNWIND` the `languages` property from the `Movie` node and turn their entries into new `Language` nodes.
2. Create the `IN_LANGUAGE` relationship to connect the `Movie` nodes to their respective `Language` nodes:
3. Remove the languages property from the `Movie` node.

Your graph should now look like this:



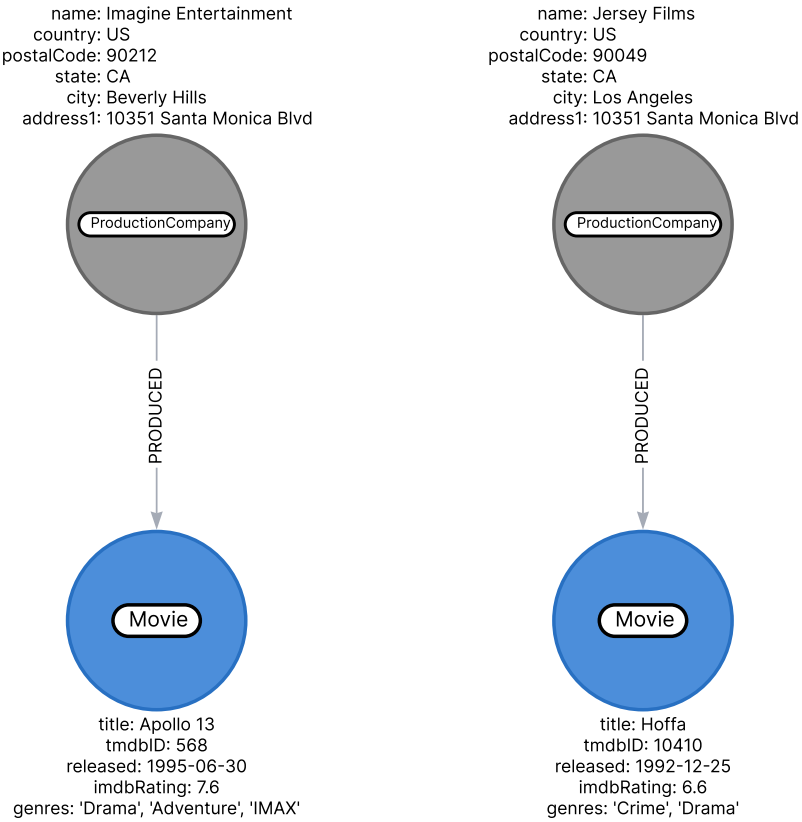
After this refactoring, you should have only one `Language` node with the value "English" and the equivalent movies connected to it. This eliminates a lot of duplication in the graph and improves performance when the graph scales.

Dealing with complex data

Suppose a new use case has come up that requires information about the producers of each film. Part of the data about the producers include their physical address, which is what can be considered complex data.

You can add this information to the graph by creating a `ProductionCompany` node and an `address` property:

```
CREATE (p:ProductionCompany {name:'Imagine Entertainment', country:'US', postalCode:90212, state:'CA',
city:'Beverly Hills', address1:'10351 Santa Monica Blvd'})
MERGE (Apollo13:Movie {title:'Apollo 13'})
CREATE (p)-[:PRODUCED]->(Apollo13)
CREATE (jerseyFilms:ProductionCompany {name:'Jersey Films', country:'US', postalCode:90049, state:'CA',
city:'Los Angeles', address1:'10351 Santa Monica Blvd'})
MERGE (hoffa:Movie {title:'Hoffa'})
CREATE (jerseyFilms)-[:PRODUCED]->(hoffa)
```



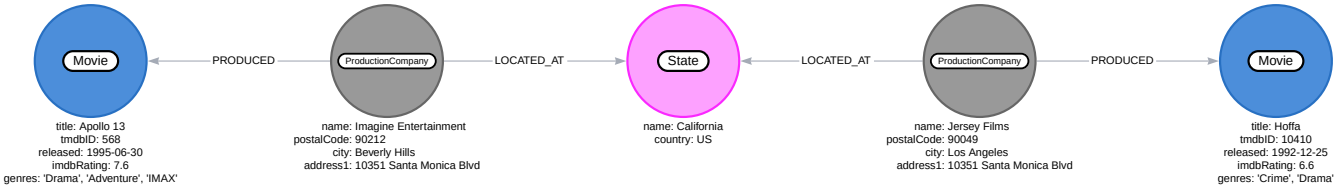
However, storing complex data on nodes this way may not be beneficial for different reasons, including:

- **Duplicate data**: There may exist several production companies in the same location, and the same information is then repeated on multiple nodes.
  - Example: In the previous step, you refactored the property 'languages' to become a node to avoid having the entry "English" duplicated on all `Movie` nodes.
- **Over-fetching**: Queries related to the information on the nodes require that more nodes in a category be retrieved unnecessarily.
  - Example: If you want to return production companies that are located in California, all the properties of the `ProductionCompany` nodes need to be scanned to retrieve the property value `California` from the `state` key. Instead, a node for `California` could be a shorter path to this information and you wouldn't need to retrieve more information than what you need.
  - Alternatively, you can also create an index.

The goal in data modeling is to reduce the size of the graph that is touched by a query. If the graph contains a large amount of duplicate data or if your queries still over-fetch data, you may need to refactor your model again.

In the current model, you have added more information in the form of a new node label `ProductionCompany` with a number of address properties. The property values contain a lot of duplicate data, which is not desirable. To make the model more efficient, check for duplicate key values and see if you can turn them into another entity, like a node or a relationship.

In this case, both production companies are based in California, so the state could be turned into a node for `State` and be connected to the producer companies via a new relationship `LOCATED_AT`:



After this refactoring, queries that retrieve production companies by their state can now be filtered based on the `State.name` value, rather than evaluating all `ProductionCompany` nodes for the `ProductionCompany.state` property.

How you refactor your graph to handle complex data depends on the questions you'd like to answer and the performance of the queries when your graph scales. The next step is to measure performance in your graph by testing it.

Using specific relationships

Specific relationships are a refactor strategy that you can use when your project has a recurrent use case that needs a certain piece of information to be constantly retrieved. The benefits of using them include:

- Reducing the number of nodes that need to be retrieved.
- Improving query performance.

Suppose that you frequently need to retrieve information about actors in reference to the year 1995. The query for that could be:

```
MATCH (p:Person)-[:ACTED_IN]-(m:Movie)
WHERE p.name = 'Tom Hanks' AND m.released STARTS WITH '1995'
RETURN DISTINCT m.title AS Movie
```

But if you create a specific relationship, for example, `ACTED_IN_1995`, when you query for this same information, you will write the code like this instead:

```
MATCH (p:Person)-[:ACTED_IN_1995]-(m:Movie)
WHERE p.name = 'Tom Hanks'
RETURN m.title AS Movie
```

This way, the query won't need to retrieve all the `Movie` nodes connected to Tom Hanks and read all their `m.released` properties, but only retrieve the title of those that are connected with Tom Hanks by the specific relationship `ACTED_IN_1995`. You can therefore avoid overfetching and improve query performance.

Retest the graph

After you have refactored the graph, you should revisit all queries for your use cases and determine whether any of them can be rewritten to take advantage of the refactoring. Here is a list:

Use case	Query example
Which people acted in a movie?	`MATCH (p:Person)-[:ACTED_IN]->(m:Movie {title:'Hoffa'})` `RETURN p`

Use case	Query example
Which person directed a movie?	`MATCH (p:Person)-[:DIRECTED]->(m:Movie {title:'Hoffa'})` `RETURN p`
Which movies did a person act in?	`MATCH (p:Person {name:'Tom Hanks'})-[:ACTED_IN]->(m:Movie)` `RETURN m`
How many users rated a movie?	`MATCH (m:Movie {title: 'Apollo 13'})` `RETURN COUNT {(:User)-[:RATED]->(m)} AS `Number of reviewers``

Who was the youngest person to act in a movie?	`MATCH (p:Person)-[:ACTED_IN]-(m:Movie)`
	`WHERE m.title = 'Hoffa'`
	`RETURN  p.name AS Actor, p.born as `Year Born` ORDER BY p.born DESC LIMIT 1`

Which role did a person play in a movie?	`MATCH (p:Person {name:'Tom Hanks'})-[a:ACTED_IN]->(m:Movie {title: 'Apollo 13'})` `RETURN a.roles`

Which is the highest rated movie in a particular year according to imDB?	`MATCH (m:Movie)`
	`WHERE m.released STARTS WITH '1995'`
	`RETURN  m.title as Movie, m.imdbRating as Rating ORDER BY m.imdbRating DESC LIMIT 1`

Which drama movies did an actor act in?	`MATCH (p:Person)-[:ACTED_IN]-(m:Movie)`
	`WHERE p.name = 'Tom Hanks' AND`
	`'Drama' IN m.genres`
	`RETURN m.title AS Movie`

Which users gave a movie a rating of 5?	`MATCH (u:User)-[r:RATED]-(m:Movie)`
	`WHERE m.title = 'Apollo 13' AND`
	`r.rating = 5`
	`RETURN u.name as Reviewer`

Which movies are in English?	`MATCH (m:Movie)`
	`WHERE m.languages = 'English'`
	`RETURN m.title as Movie in English`

With this considered, you should now determine if any of the queries need to be rewritten to take advantage of the refactoring and rewrite them when applicable. For example, for the use case "Which movies are in English?":

Old query	Query after refactoring

`MATCH (m:Movie)`
`WHERE m.languages = 'English'`
`RETURN m.title as Movie in English`
	`MATCH (m:Movie)-[:IN_LANGUAGE]->(l:Language)`
	`WHERE l.name = 'English'`
	`RETURN m.title as Movie in English`

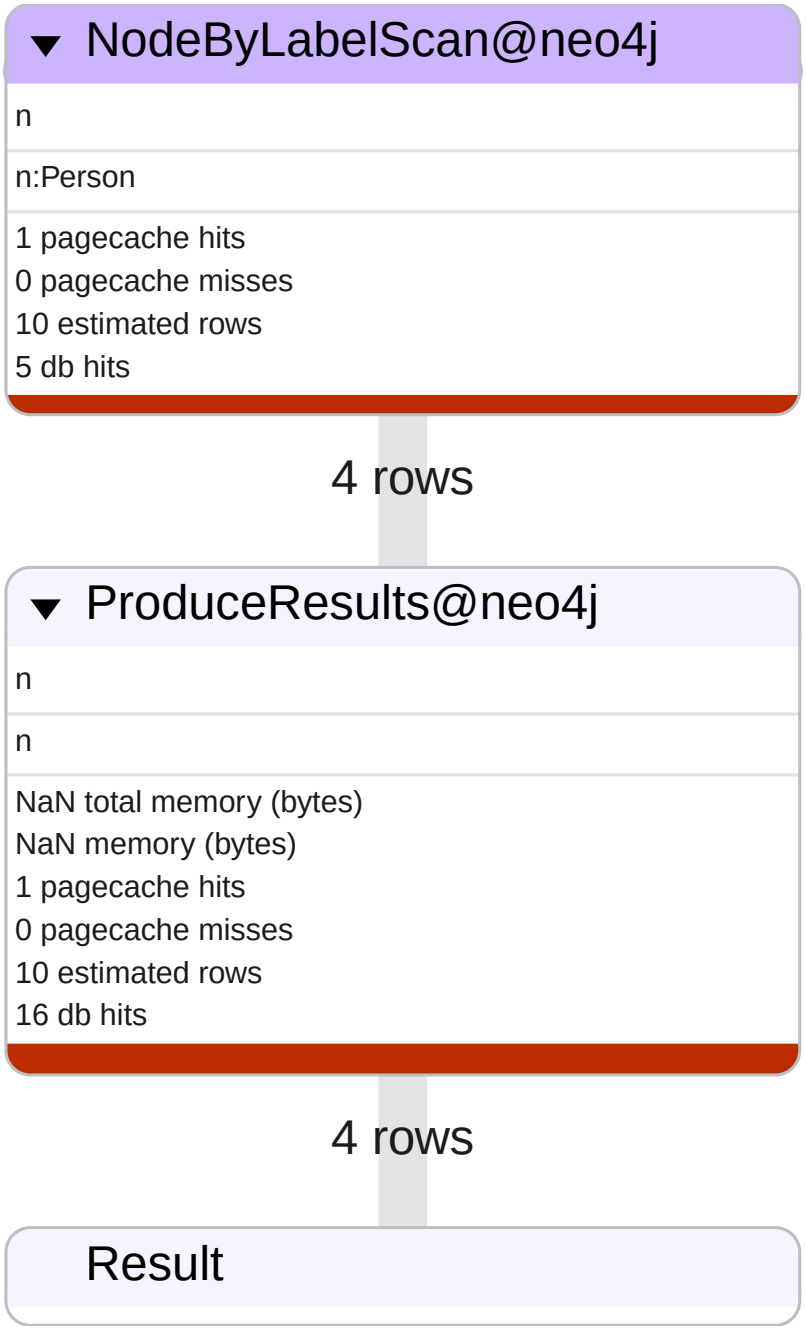
Performance check

When testing on a real application and, especially with a fully-scaled graph, you can also profile the new queries to see if it improves performance. On a small instance model such as the example in this tutorial, you will not see significant improvements, but you may see differences in the number of rows retrieved.

As an example, if you want to see the number of database hits for a query to retrieve all `Person` nodes, you need to add the clause `PROFILE` before it:

```
PROFILE MATCH (n:Person)
RETURN n
```

This should be the result:



You can find more detailed information on query tuning and planning at Cypher manual → Execution plans and query tuning.

Keep learning

Most of the refactoring that you can keep doing on your model is about repurposing or adding more information to your graph.

You can see more examples on how to split the node `Person` into `Actor` and `Director` nodes, how to turn the `Movie` node property `genre` into nodes, and other refactoring strategies by following the interactive course Graph Data Modeling Fundamentals on GraphAcademy.

Modeling designs

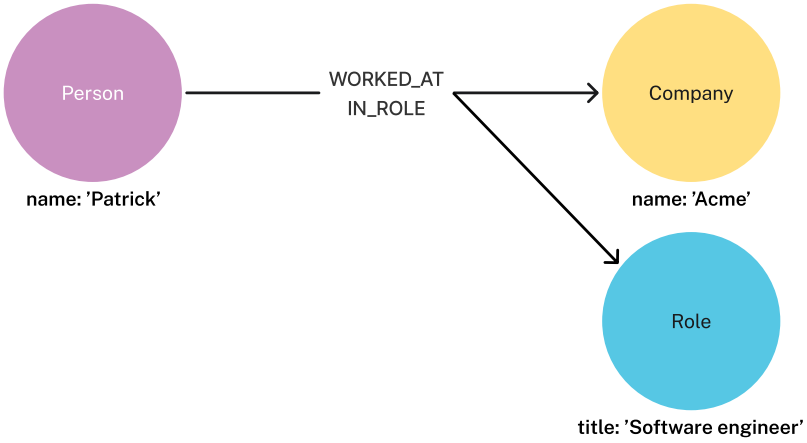
This page features examples of graph data modeling patterns and designs that are commonly used with Neo4j. The purpose is to get an overview of the options available for building graph data models and how known strategies can be adapted to your project.

Intermediate nodes

Intermediary nodes are nodes that contain data that need to be in the graph but don't seem to fit neatly into the initial model.

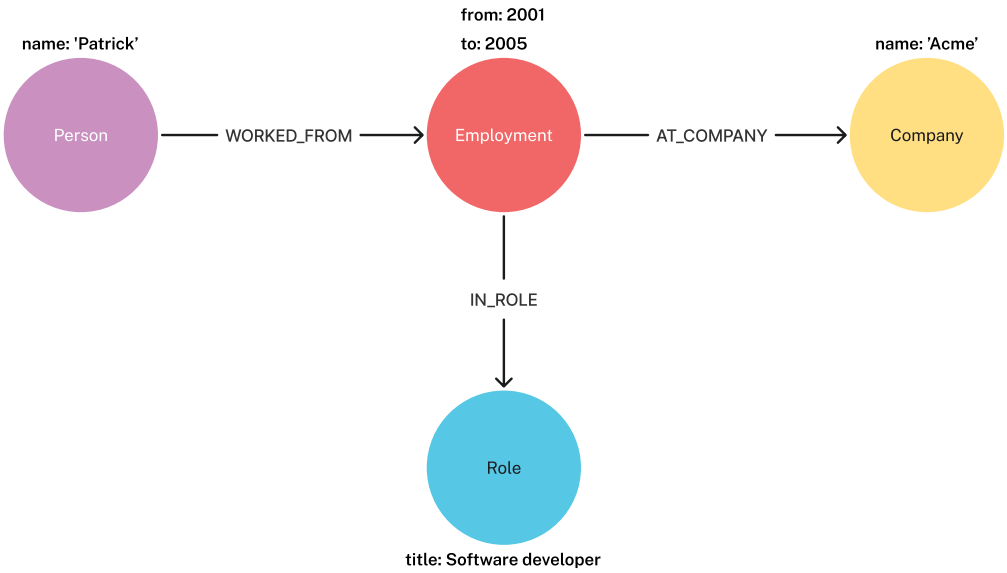
Sometimes you need to convey a lot of information in a relationship. In a mathematical graph, this can be solved with a **hyperedge**, i.e. a relationship that connects more than two nodes. This is not supported in Neo4j but can be solved by using an intermediary node.

For example, consider a person who works at a company and you need to convey information about their role:



In a mathematical graph, you could use the same relationship `WORKED_AT` to connect the `Person` node with both `Role` and `Company` nodes. However this is not supported in Neo4j.

Instead, you could either turn the `Role` node into a property of the `WORKED_AT` relationship or use an **intermediate node** between the `Person`, `Company`, and `Role` nodes:

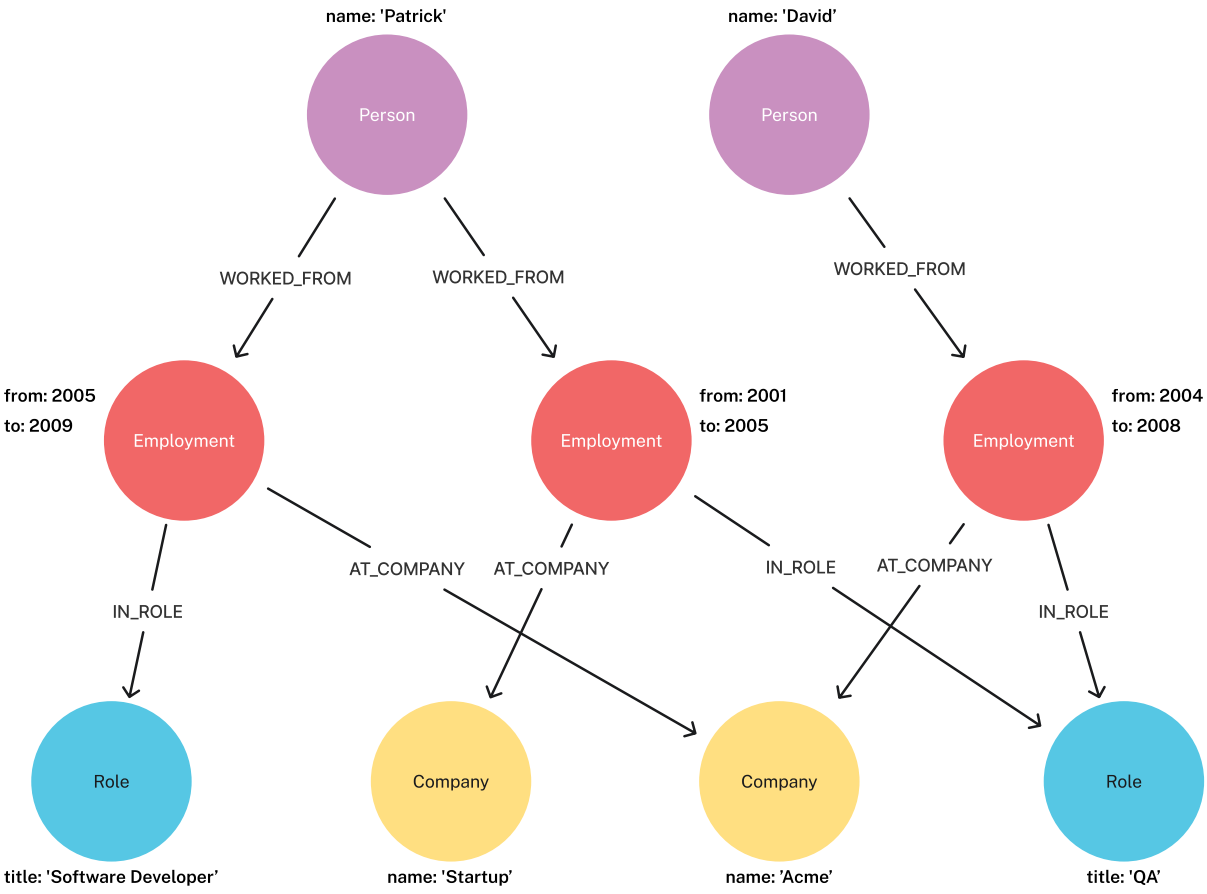


In this new graph, instead of saying Patrick works at company Acme, Patrick has an **employment event**, which becomes a new node. The employment event holds the employment start and end dates, and logically relates to the other three nodes.

Despite the fact that an employment event is an abstract idea, it is a good way to link related additional information.

Sharing context

In this expanded version of the previous example, a new `Person` node with the name David is added:



This expanded example highlights the ability to show shared context between multiple nodes using a

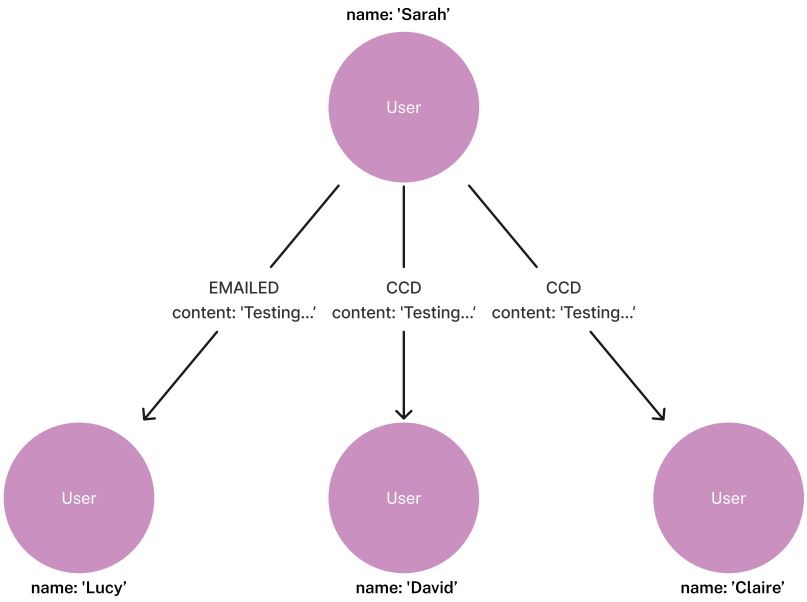
common event (the intermediary node). Specifically in this example, the `Person` nodes share context through `Role` and `Company` nodes. The `Employment` nodes provide a way to trace details such as a person's career, or the overlap between different individuals at the same `Company`, or those who had the same `Role`.

The use of intermediary nodes can also answer the question "Who worked at the same company at the same time?" as the added employment event contains information about when each individual worked at a certain company. A `MATCH` clause would show that Patrick and David both worked at Acme, being colleagues from 2004 to 2005 since their employment events overlap during that time:

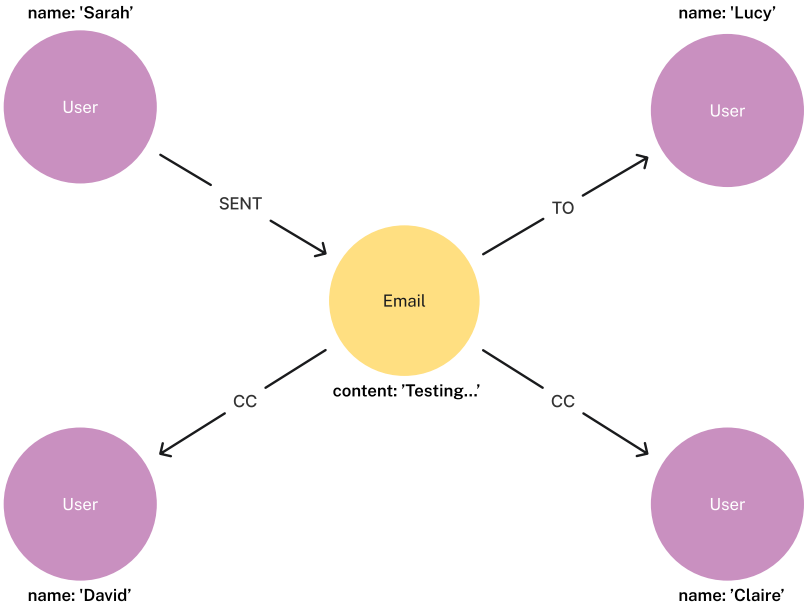
```
MATCH (p1:Person)-[w1:WORKED_AT]->(c:Company {name: "Acme"}),
      (p2:Person)-[w2:WORKED_AT]->(c)
WHERE p1 <> p2
  AND w1.startDate <= w2.endDate
  AND w2.startDate <= w1.endDate
RETURN p1.name AS Person1, p2.name AS Person2
ORDER BY Person1, Person2
```

Sharing data

Intemediate nodes can also add value to a model by providing a way to share data and thus reduce duplicate information. In this example, Sarah sends an email to Lucy and copies David and Claire to it. The content of each email is a property on every relationship:



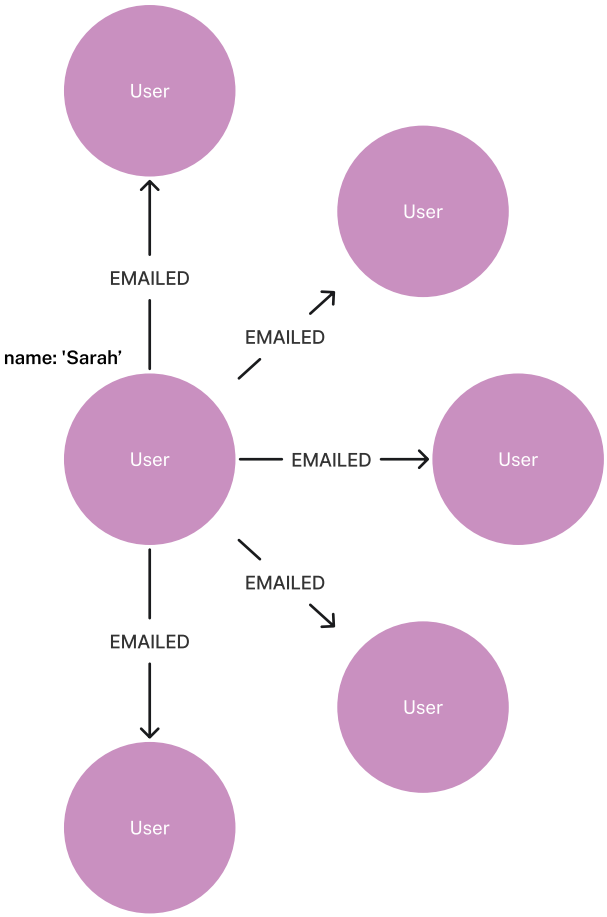
If you instead fan out the model, you reduce duplication by breaking out the property `content` from all relationships and turning it into the intermediary node `Email` instead:



Once the property value `content` is moved to a single node `Email`, it can be referenced via relationships with the `User` nodes that previously held that value. Now there are no duplications.

Organizing data

Intermediate nodes can also help organize structures. In the previous example, Sarah sent the same email message to several people. If Sarah sends more email messages to more people, without using intermediary nodes, the graph quickly grows to this:

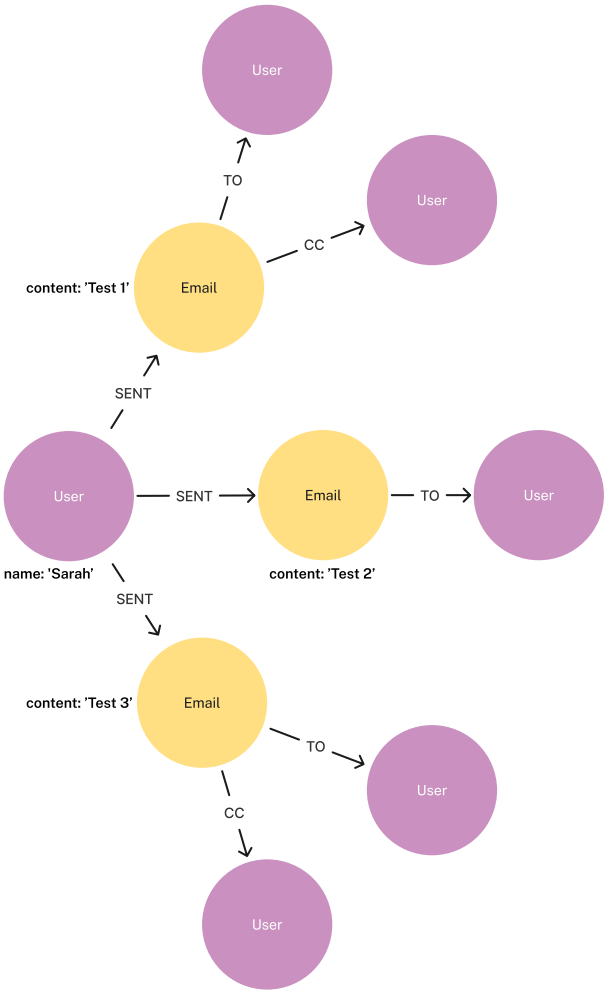


When every `EMAILED` relationship includes a property with the content of the message, in addition to

duplication, two other problems can arise:

- **Sarah's node is becoming very dense**: For every email she sends, including CC's, her node gains another relationship.
- **It's expensive to retrieve the content of the email**: With the data modeled like this, it's very expensive to determine who in Sarah's recipient network has received a given message by searching for the content in multiple 'EMAILED' relationships.

When you fan out and add intermediate nodes to represent each email message, Sarah's node has only one relationship per email message, regardless of the number of recipients:



With this model, you can find the recepients by locating the specific `Email` node that now contains the content of the message in the `content` property, and then see which users are connected to it via `TO` relationships.

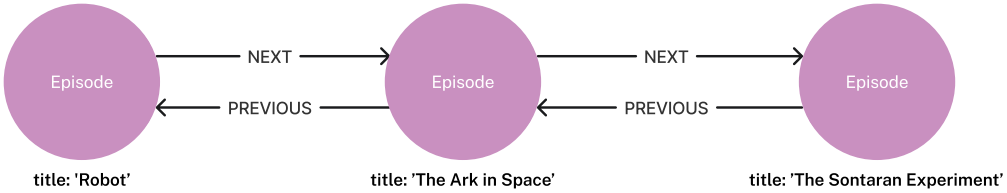
While both models use a gather-and-inspect approach, the scope of the problem is reduced significantly after the refactoring. In the first iteration, if you want to see who received a certain email, you need to find all users connected to Sarah via the `EMAILED` relationship. In the second iteration, you only need to locate the correct `Email` node, then traverse from it to all of the connected recipients.

In summary, you're likely to find many uses for intermediate nodes during refactoring since you rarely recognize the need for them at the outset of the data modeling.

Linked list

Linked lists are commonly used in computer science and they are particularly useful whenever the sequence of objects matters. A **simple-linked list** is where each node links to the next node only:

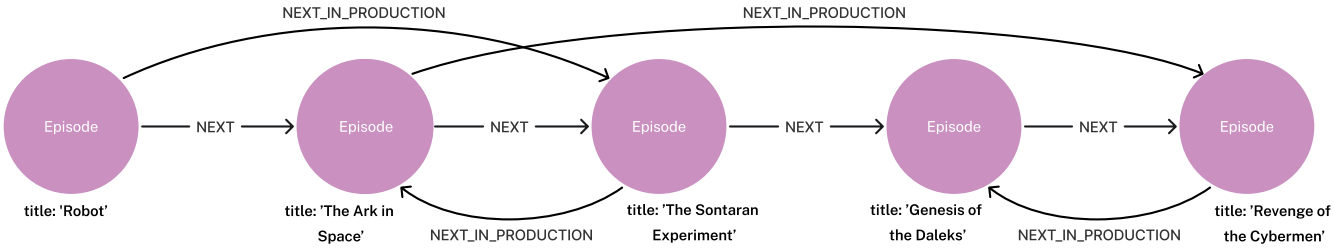
In a **double-linked list**, each node links both to the next and the previous node:



Double-linked lists are not recommended because one relationship becomes redundant (if one is the next, then the other is the previous) and Cypher also allows bi-directional matches. Moreover, while it is common practice to use verbs as relationship types, with linked lists, it is acceptable to connect sequential items using terms such as "next" and "previous" instead.

Interleaved linked list

Interleaved lists are used when you want to sequence a set of items based on context, not on chronology. This example combines a linked list with an interleaved linked list of Dr. Who episodes:



The order in which TV episodes are aired is often different than the order in which they are produced. This example contains five episodes of Dr. Who from season 12 and it shows:

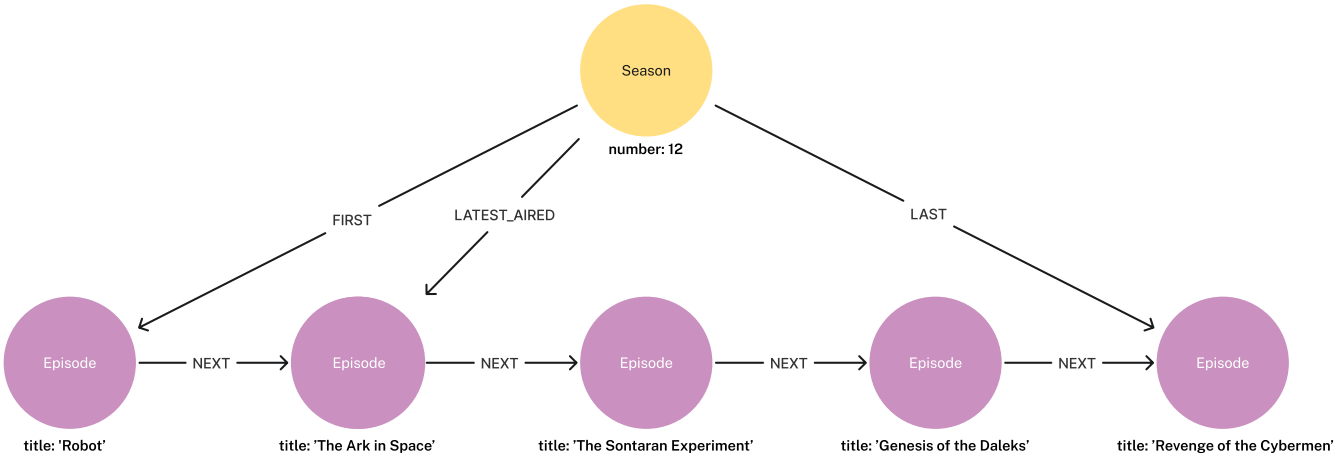
- The order in which the episodes were aired using the `NEXT` relationship and through a simple-linked list.
- The order in which the episodes were produced using the `NEXT_IN_PRODUCTION` relationship, which creates an interleaved linked list. It is not a linear list, as it goes 1, 3, 2, 5, 4.

Note that this example is **not** a double-linked list because the relationships are not mutually exclusive.

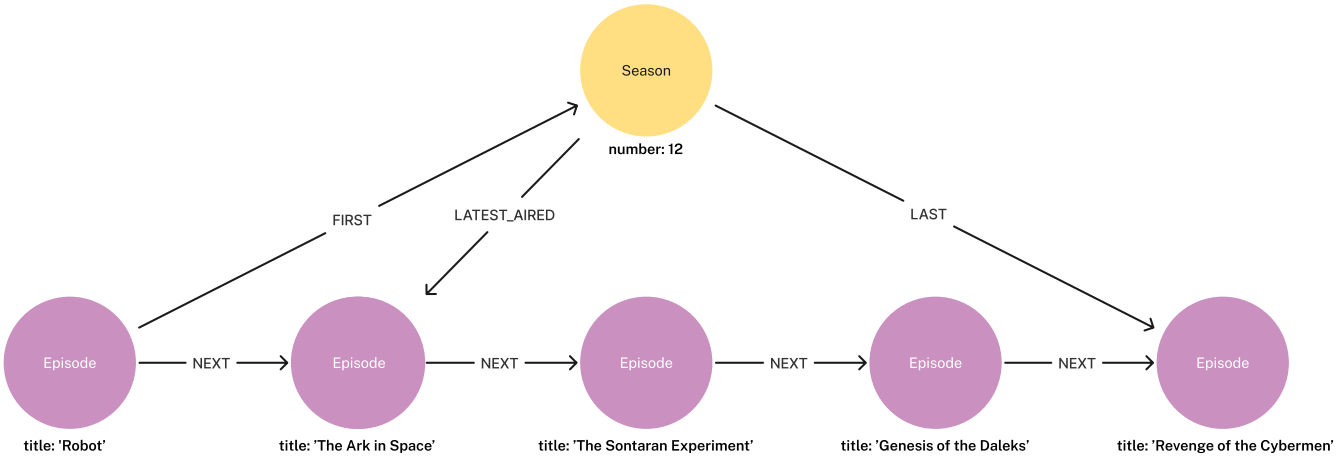
Head and tail of a linked list

When working with linked lists, there is often a “parent” node that is used as the entry point. The parent almost always points to the first item in the sequence, using an appropriately named relationship. Sometimes, another relationship points to the last item in a list.

In this example, you can see a `FIRST` and a `LAST` relationship, referring to their places in the sequence:



Some implementations also have a "progress" pointer that is used to keep track of the current node of interest. This can be done through a relationship, as such:



The progress pointer here is the `LATEST_AIRED` relationship and it shows which was the most recently aired episode (i.e. "The Ark in Space"). When the `NEXT` episode ("The Sontaran Experiment") airs, the relationship is updated by deleting the current one and creating a new `LATEST_AIRED` pointer, so that it always points to the current item.

Versioning

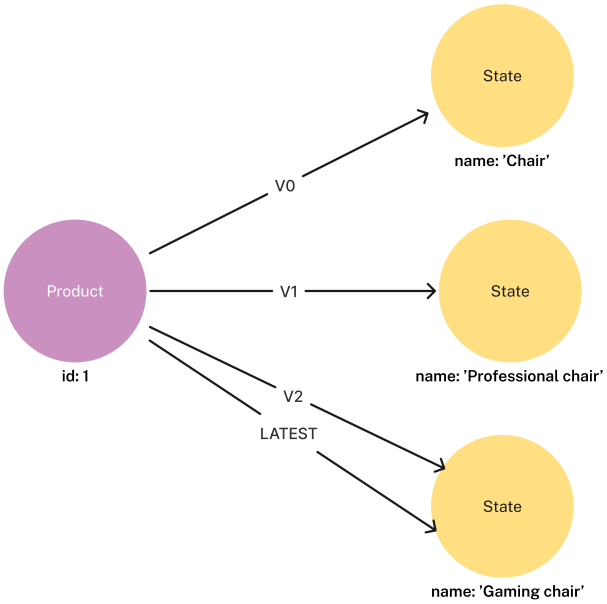
Every time you refactor your data model, you create new versions of it. Tracking changes in the data structure or showing a current and past value can be valuable for auditing purposes, trend analysis, etc. This page gives an overview of the different ways you could model data in order to keep track of changes over time.

Versioning of entities

You can keep track of changes in data by versioning relevant entities. This strategy is useful when you need to:

- Access the many versions of specific entities (nodes, for instance) in a graph (e.g. the different names a product has had throughout time).

- Retrieve the latest version only (e.g. the current name of a product).



With entities versioning:

- The entity `Product` is linked to its different versions by an explicit relationship.
- The entity `Product` is immutable. Only the properties that are stored in the different versions (`State` nodes) change.
- The `LATEST` relationship links the entity `Product` to its most recent version (`State`), which also happens to be version 2 (`V2`).

Pros and cons

Pros	Cons
Simple in terms of modeling, querying, and maintenance.	Updating nodes requires the deletion of the `LATEST` relationship, and the creation of a new relationship between the entity and its latest version.
Explicit for end users without any transformation.	Can be limited if not using other versioning patterns, as it can be hard to know which version you want to retrieve if it's not the latest.

Query examples

These are examples of common queries that are useful with the entity versioning strategy:

Get the name of the version 2 of a `Product` *with the id '1'*

```
MATCH (:Product {id:1})-[:V2]->(s:State)
RETURN s.name
```

Get the name of the latest version of a `Product` *with the id '1'*

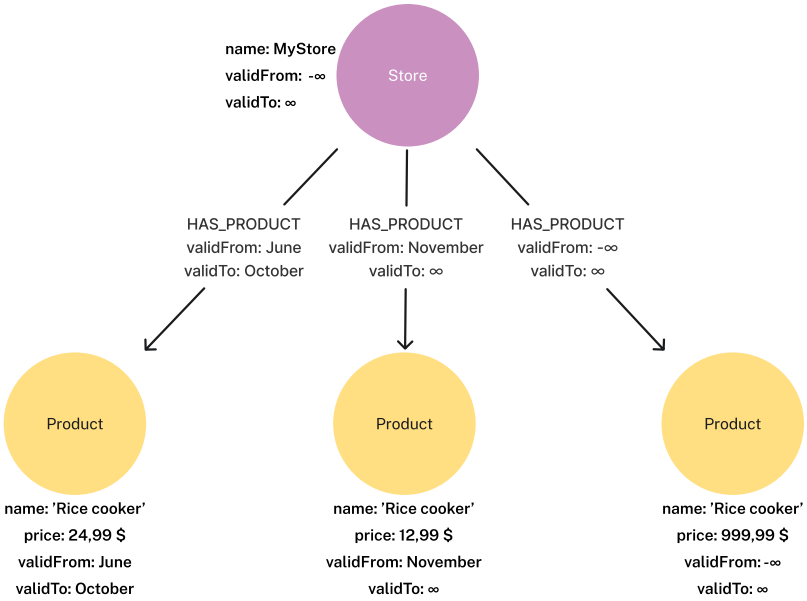
```
MATCH (:Product {id:1})-[:LATEST]->(s:State)
```

```
RETURN s.name
```

Time-based versioning of entities

A variation of the entity versioning is a time-based approach. It is useful when you are interested in:

- **Graph snapshot** by retrieving all valid elements (nodes and relationships) of the graph to a specific point in time (e.g. which products are available on Monday the 12.06.23).
- **Graph difference** by comparing two graph snashots of different time stamps (e.g. which nodes are added, which are deleted, and which remain the same).
- **Temporal traversal** by traversing only valid elements (node or relationships) of the graph to a specific point in time in order to find the chronological sequence of relationships which connect time-based events (e.g. bike sharing graph with trip relationships between stations as nodes).
- **Graph history** by modeling the history of data changes.



With time-based versioning of entities:

- Each element has dedicated `validFrom`/`validTo` time properties.
- Nodes can only share a relationship if their validity timespan overlap.
- Duplication of information is possible.
- Complete history of the graph is usable.

Pros and cons

Pros	Cons
Every element has a well defined time interval in which the element is valid.	If the state of a node changes, the node has to be duplicated and a new valid time interval should be assigned.

Pros	Cons
States are bound to the specific element (no additional relationship required).	Updating nodes requires the creation of a new relationship connecting to the new node/state and the assigning of A new valid interval to the relationship.
Aggregation of all elements (or only valid ones at a certain time) is possible.	Duplications of data cannot be avoided.

Query examples

These are examples of common queries that are useful with the time-based entity versioning strategy:

Get the current price of the `Product` *Rice Cooker*

```
MATCH (p:Product)
WHERE p.name = “Rice Cooker” AND p.validTo = ∞
RETURN p.price
```

Get the price of the `Product` *Rice Cooker in November*

```
MATCH (p:Product)
WHERE p.name = “Rice Cooker”
AND datetime(p.validFrom) <= datetime(“November”) <= datetime(p.validTo)
RETURN p.price
```

Get the current product catalogue and the prices

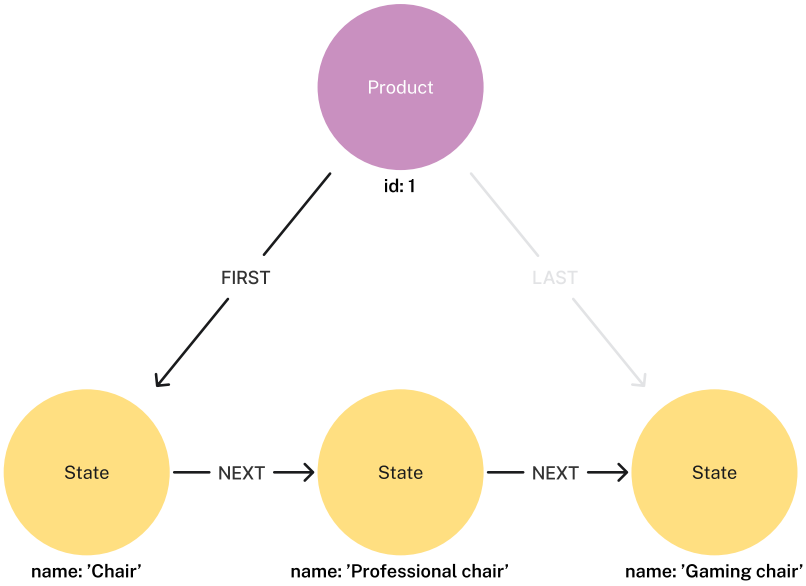
```
MATCH ()-[r:HAS_PRODUCT]->(p)
WHERE r.validTo = ∞
RETURN p.name, p.price
```

Linked list

A linked list is another modeling strategy that can be useful when the sequence of objects matters.

Linked lists are useful when:

- The order of events is of interest, e.g. getting the order of transactions executed on a bank account.
- You need the previous and next elements in a list, based on the relationship between them (e.g. what song is the next on a playlist, or undo an action on a text document) are .



With a linked list:

- The entity `Product` is linked to the first element of the sequence, and can be linked to the last one.
- As with the the versioning of entities, the entity `Product` is also immutable here.
- Each element of the sequence is linked to the next one through a `NEXT` relationship.

Pros and cons

Pros	Cons
Efficient by using relationships to get the next/previous element.	Limited to very specific use cases without using other versioning patterns.
Simple modeling and maintenance.	Difficult to find a specific version which is not the first or the last.
Explicit for end users.	

Query examples

These are examples of common queries that are useful with the linked-list versioning strategy:

Get the next name of the product named "Professional chair"

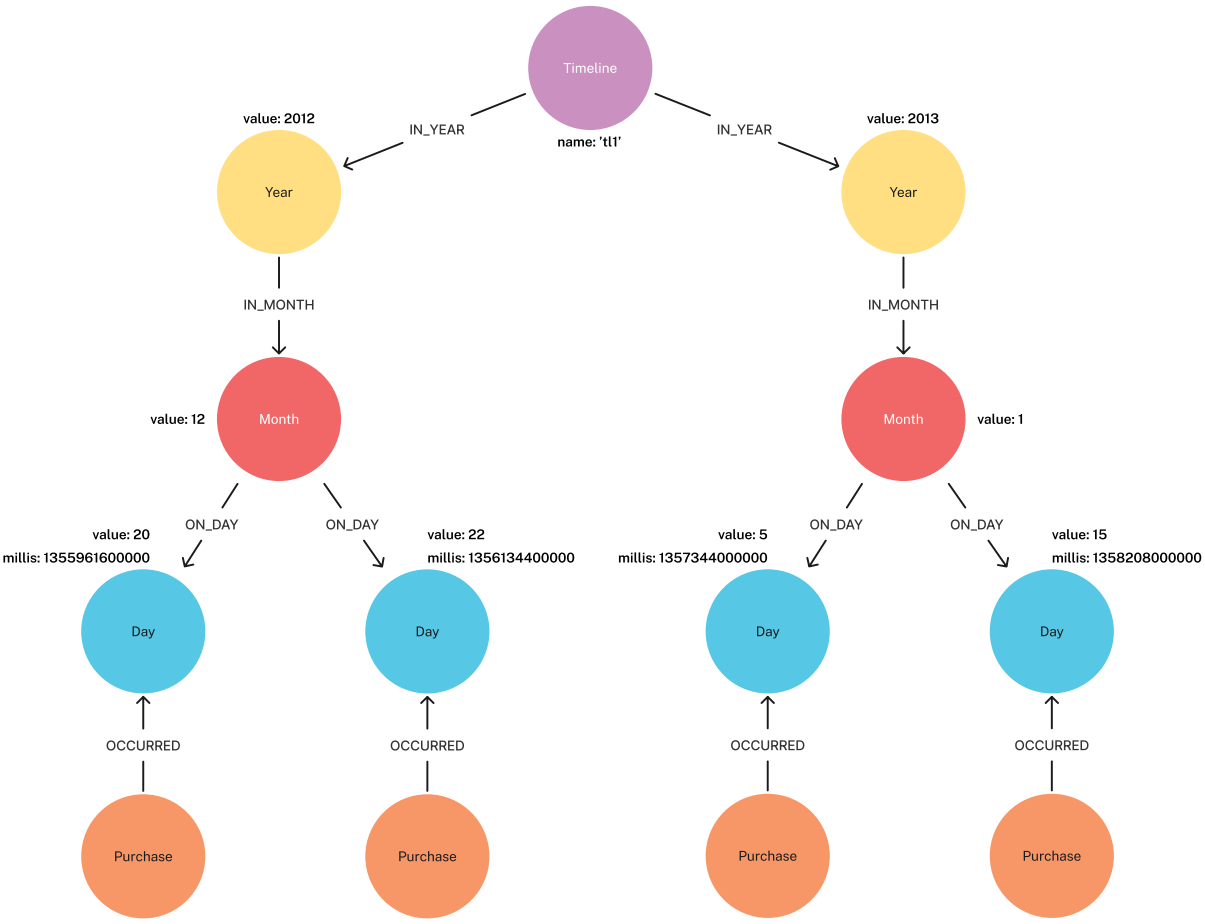
```
MATCH (:State{name: "Professional chair"})-[:NEXT]->(s:State)
RETURN s.name
```

Get the previous name of the product with the id '1'

```
MATCH (:Product {id:1})-[:LAST]->(:State)<-[:NEXT]-(s:State)
RETURN s.name
```

Timeline tree

As mentioned in Modeling designs, the timeline tree is a common modeling design. It can be a useful strategy when you want to track change. In this example, the timeline structure spans from years to days, and the rest of the non-time data nodes are the nodes that contain the important pieces of data in the graph:



Query examples

If you want to find all purchases that happened in a given time period, such as every purchase in the month of December 2012, the timeline tree can be navigated from 2012, to December, and then fetch everything from the connected leaf nodes (nodes with no descendants) under that branch:

```
MATCH (root:Timeline)-[:IN_YEAR]->(year:Year {value:2012})-[:IN_MONTH]->(month:Month {value:12})
WITH month
MATCH (month)-[:ON_DAY]->(day)
MATCH (purchase:Purchase)-[:OCCURRED]->(day)
RETURN purchase
```

Combined approach

Some complex use-cases require the combination of one or more of the previously mentioned modeling techniques since each has advantages and disadvantages.

The right combination depends on the specific use-case. Factors such as query times and the frequency of transactions should be considered as well.

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.

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.

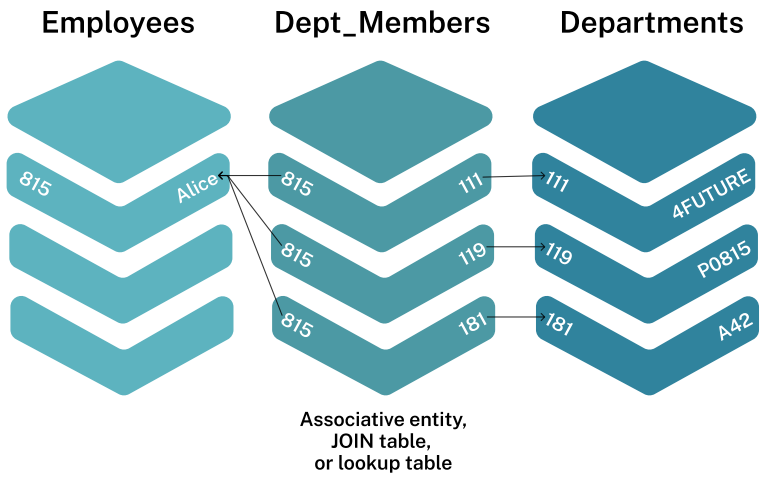


Figure 24. Relational - Person and Department tables

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.

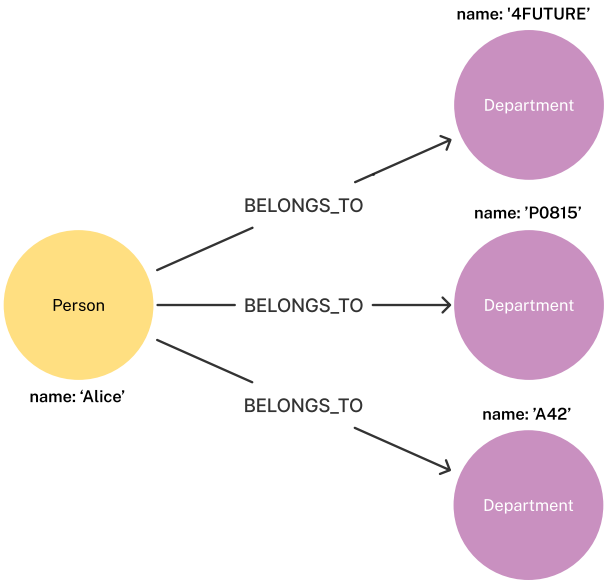


Figure 25. Graph - Alice and three departments as nodes

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.

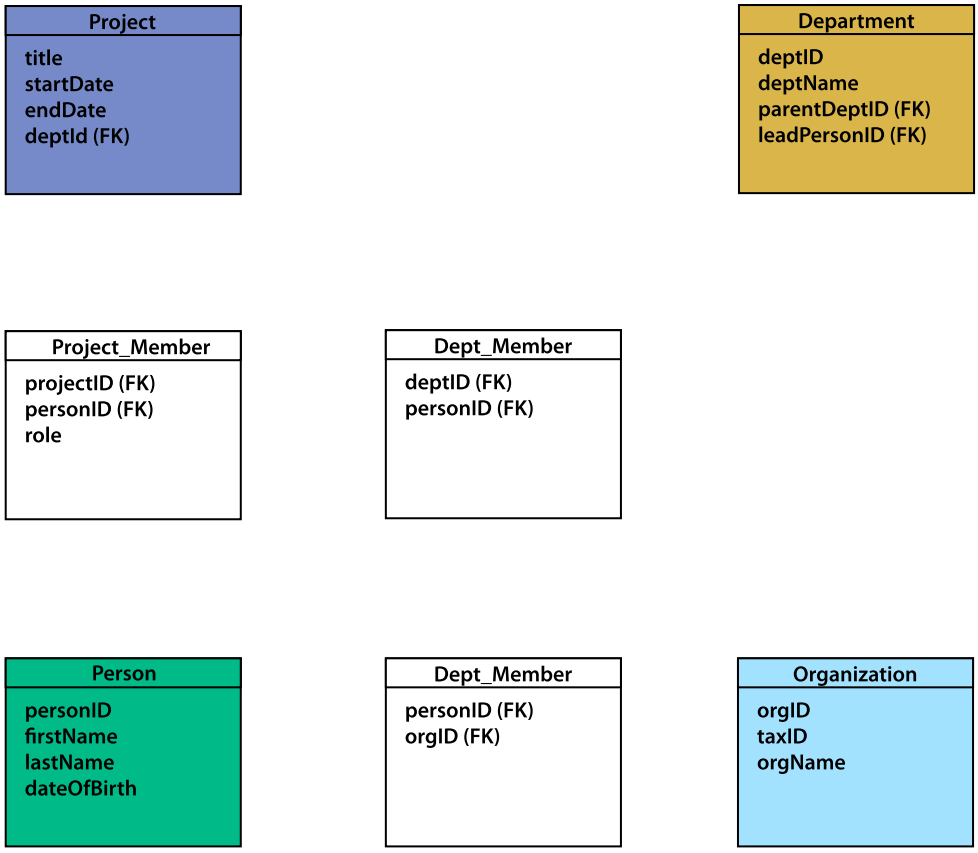


Figure 26. Organizational domain - Relational model

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.

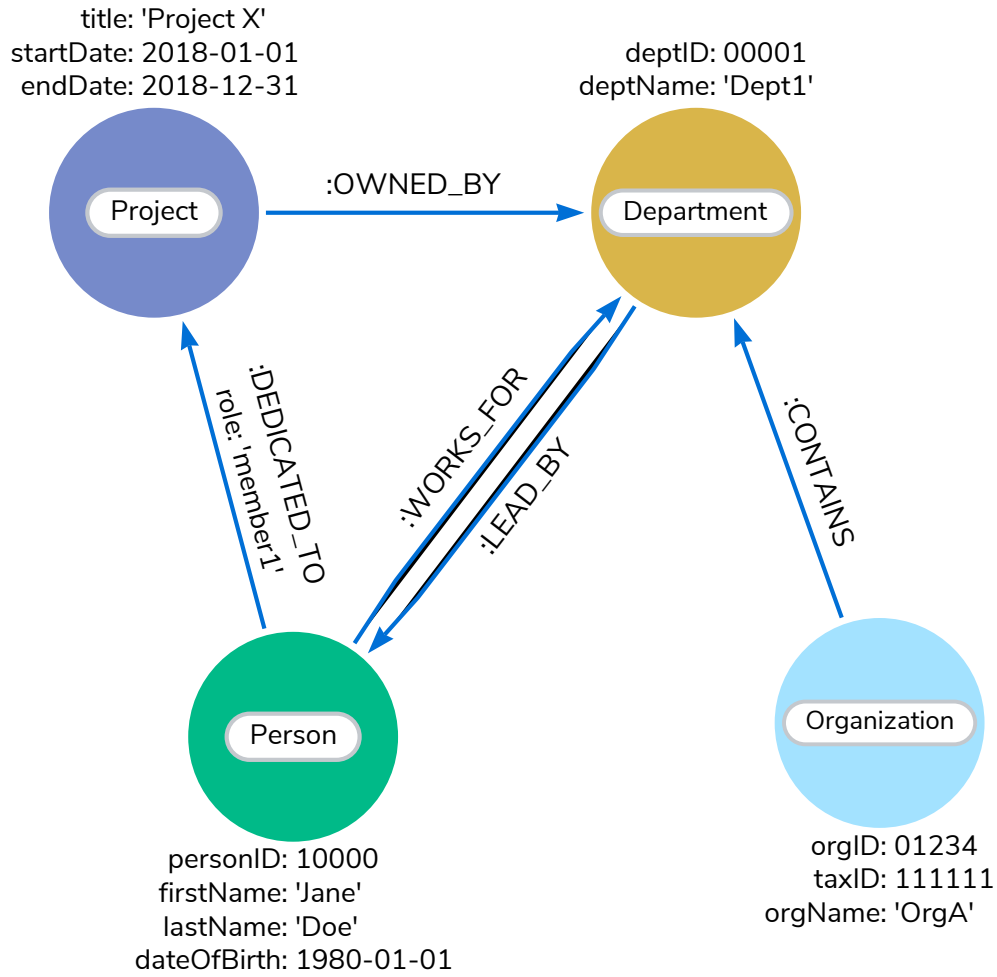


Figure 27. Organizational domain - Graph model

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 on this topic, check out the APOC 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.

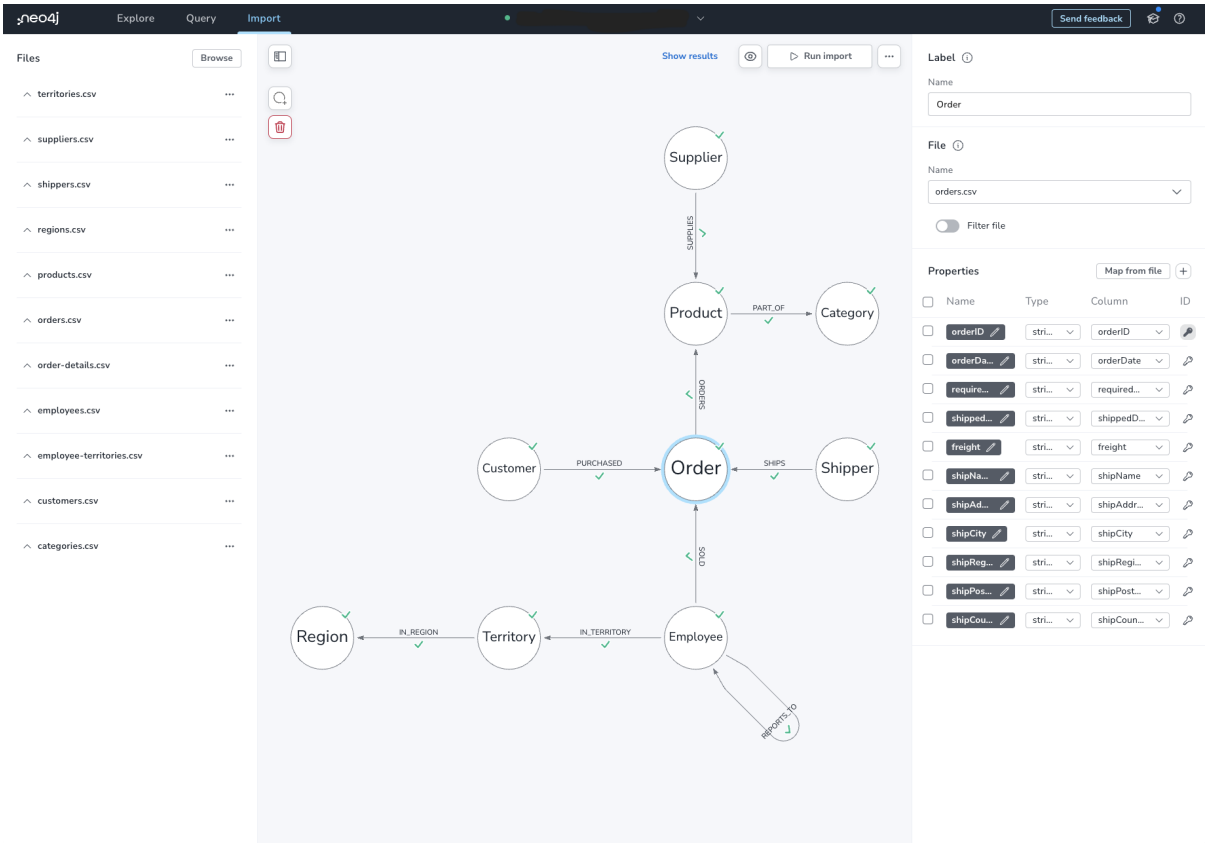
Resources

- Ask Questions on the Neo4j Community Site!

Data modeling tools

There is a range of no-code tools to get you started with data modeling. Some available tools are listed here.

Neo4j Data Importer

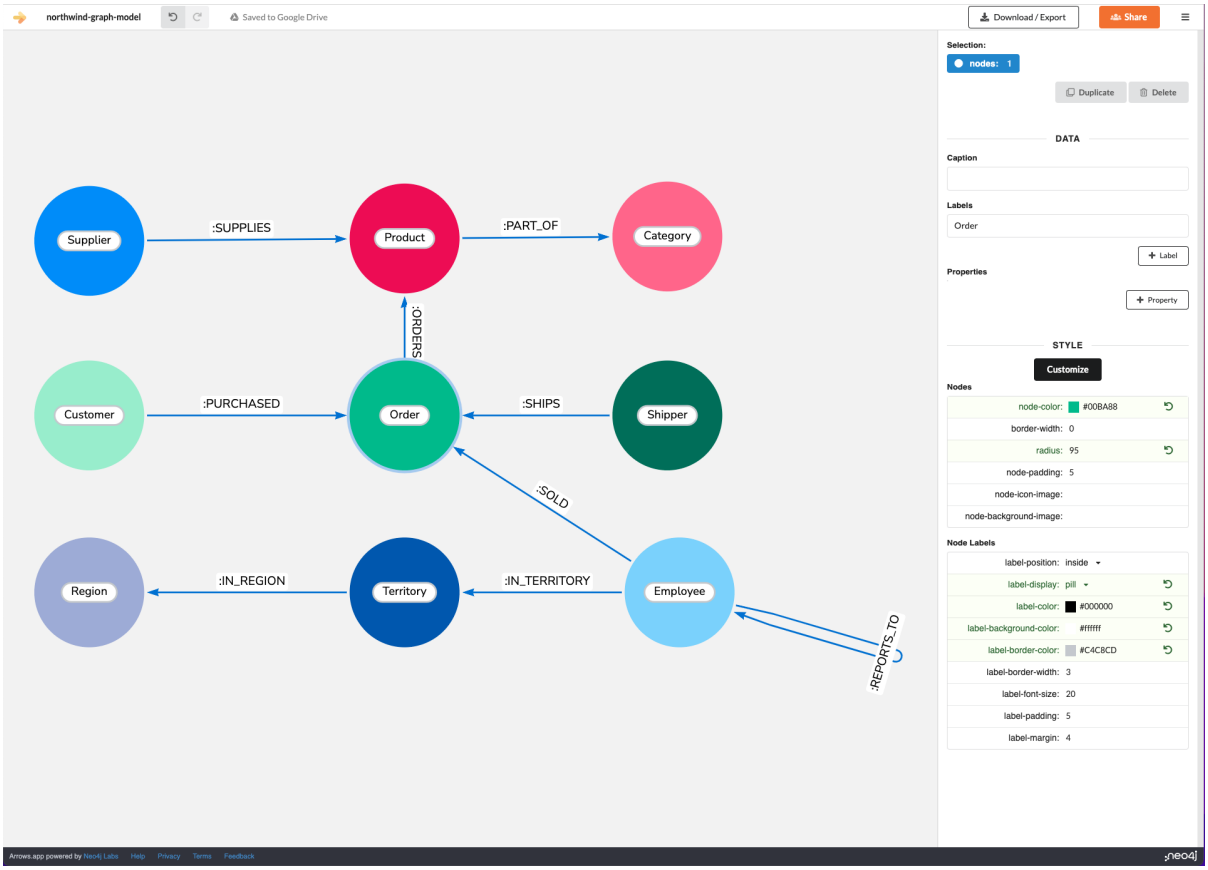


Neo4j Data Importer integrates data modeling into the import process and lets you sketch a graph model and map your data to it. You can access it:

- Via **Import** in Neo4j Aura.
- As a standalone version with secure connection only or with both secure and insecure connections.

For more information, refer to Neo4j Data Importer's documentation.

Arrows.app



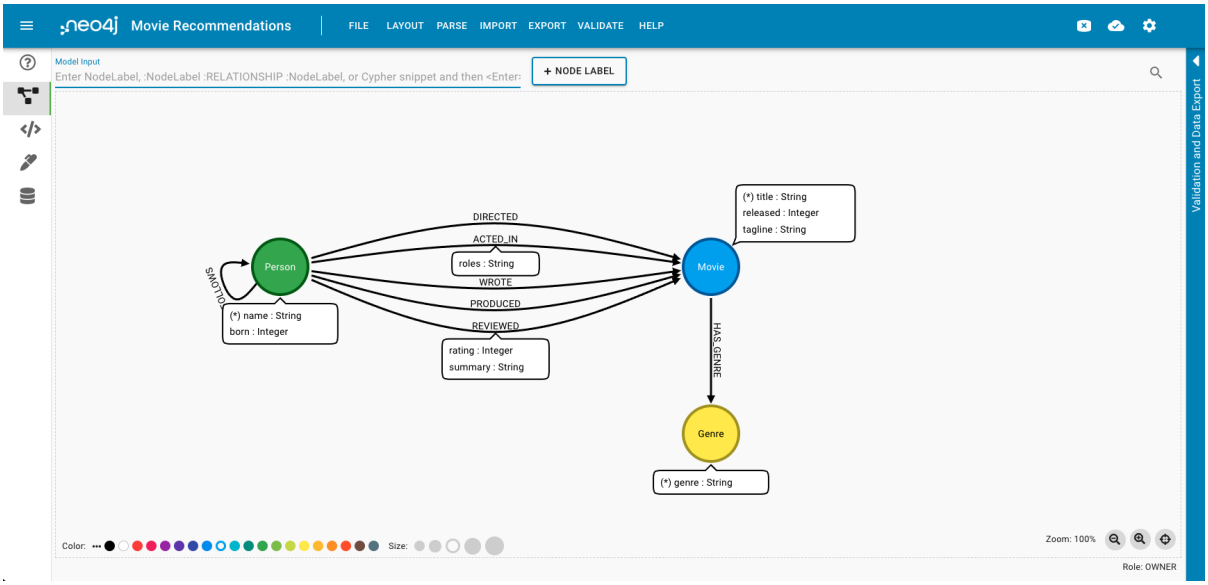
Arrows.app is a no-code visualization platform which allows whiteboarding ideas into a graph model. It is ideal for designing a domain model for your data.

With this platform, you can:

- Draft your own whiteboard from scratch or import data from JSON files and plain text.
- Create, modify, and delete nodes and relationships with their labels and properties without writing any code.
- Export the visualization as a Cypher statement and load it into a Neo4j database.

Cypher Workbench

Labs



Cypher Workbench is a cloud-based tool that assists Neo4j developers in creating and maintaining solutions built on top of Neo4j. It combines no-code visual solutions as the ones available in Arrows.app while also offering importing options similar to Neo4j Data Importer.

With this platform, you can:

- Create a data model from scratch or import data from JSON files.
- Reverse-engineer data models from existing Neo4j databases.
- Use Cypher statements to augment the current data model, including node labels, relationship types, and properties.
- Validate your model (naming conventions, constraints, data, common mistakes, etc).
- Use a business scenarios tool for capturing questions and scenarios of use cases.
- Import data from Excel, Google Sheets, or plain text.

For instructions on how to install it, refer to Cypher Workbench's documentation.

Other tools

There are other non-Neo4j tools that can be used for data modeling:

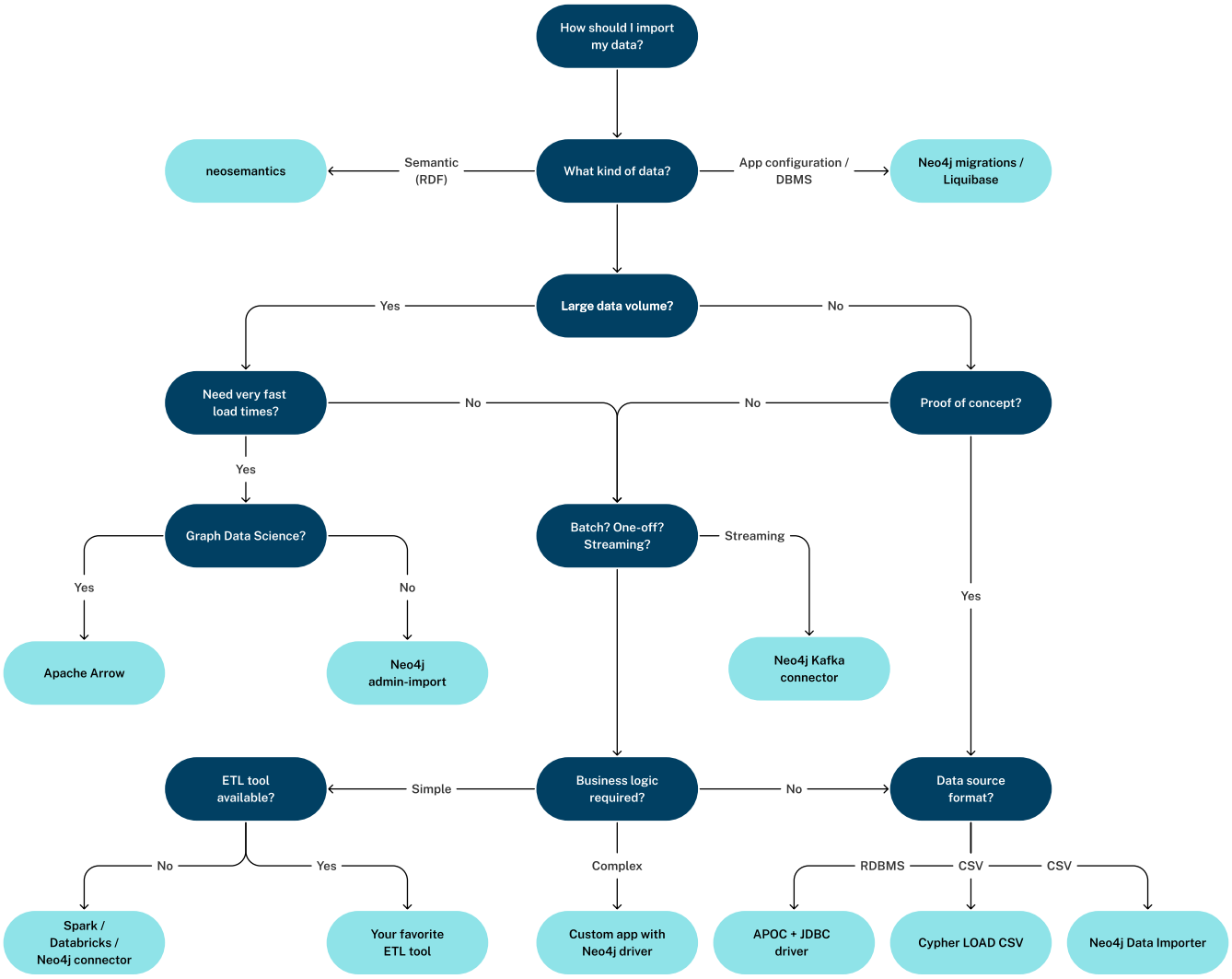
- Mermaid: general data modeling tool (not specifically for graph databases), based on Markdown. Ideal for documenting modeling strategies.
- PlantUML: application for creating diagrams from plain text. This is more for version control than model design.
- Hackolade: a tool to design, document, and communicate data models and schemas. Built to support the kind of data modeling specific to Neo4j with node labels and relationship types.

Tools comparison

Tool	Free	Import	Export
Data Importer	✔	.csv, .tsv	-
Arrows	✔	JSON	Image, Cypher, JSON,URL, GraphQL
Cypher Workbench	✖	Cypher Workbench JSON, Apoc.meta.schema, Arrows JSON	JSON
PlantUML	✔	PUML, JSON	PNG, SVG, LaTeX format and ASCII art diagrams
Mermaid	✖	MarkDown	MarkDown
Hackolade	✖	Hackolade JSON, YAML, DDL, XSD, Excel Template, Cloud Storage, Collibra Data Dictionary	Cypher, HTML

Import your data into Neo4j

Neo4j provides different tools for importing data stored in various formats, e.g. .csv, .tsv, and .json. Depending on the kind of data you are working with, there are different options to choose from, as shown in the diagram:



> **Tip:**
>
> For a more hands-on guidance, see the available GraphAcademy courses on data import.

Methods comparison

The following table shows all supported methods for importing data into Neo4j:

Method	Description	Available on Aura	Available on self-managed	Supported file formats and data sources
Import	A service for importing data into your Aura instance.	✔	✖	CSV, PostgreSQL, MySQL, SQL Server, Oracle, Snowflake
Data Importer	UI-based tool for importing flat files into Neo4j.	✔	✔	CSV, TSV
Neo4j Desktop	Import service for local files.	✖	✔	CSV
`LOAD CSV`	Cypher command to import small- to medium-sized datasets (up to 10 million records) from local and remote files, including from cloud URIs.	✔	✔	CSV
APOC import	A library of user-defined procedures and functions that extends the use of Cypher.	✔	✔	CSV, JSON, XML, XLS
Language libraries	Use Python, Java, JavaScript, Go, .NET, and JCBD to import files.	✔	✔	Language-independent columnar memory format for flat and nested data.
`neo4j-admin database import full`	Initial import into a non-existent empty database.	✖	✔	CSV, Parquet
`neo4j-admin database import incremental`	Used when import cannot be completed in a single full process. It allows the import to be performed as a series of smaller batches.	✖	✔ [1]	CSV, Parquet
Apache Arrow	Projecting graphs via Apache Arrow allows importing graph data which is stored outside of Neo4j. Apache Arrow is a language-agnostic in-memory, columnar data structure specification.	✔	✔ [1]	Language-independent columnar memory format for flat and nested data.
Neo4j Connector for Kafka	Stream data between Neo4j and platforms based on Apache Kafka using the Kafka Connect framework.	✔	✔	Language-independent columnar memory format for flat and nested data.
Neo4j Connector for Apache Spark	Process and transfer data between Neo4j and other platforms such as Databricks and several data warehouses.	✔	✔	Language-independent columnar memory format for flat and nested data.

Method	Description	Available on Aura	Available on self-managed	Supported file formats and data sources
Apache Hop	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.	✔	✔	hwf, hpl, JSON, CSV, TXT, XML, Markdown, SVG, Log, SAS 7 BDAT files
Neo4j ETL Tool	Neo4j Labs' interactive tool for the initial import of data from relational database management systems into Neo4j.	✖	✔	CSV
Neosemantics	Neo4j Labs' plugin that enables the use of RDF and its associated vocabularies.	✖	✔ [1]	RDF, OWL, RDFS, SKOS
Neo4j Migrations	Neo4j Labs' set of tools that provides a uniform way for applications, the command line, and build tools alike to track, manage and apply changes to your database.	✔	✔	See compatibility and features.

Working with CSV files

This page presents some points to consider to optimize your CSV files before importing them to Neo4j.

Structure of a CSV file

There are several elements in a CSV file that can help you understand the data that you are about to import and eventually model.

Data format

Neo4j reads all data from the CSV file as a `string`. For other data types, you need to use `toInteger()`, `toFloat()`, `toBoolean()`, or similar functions to convert data to the appropriate type. Remember also that labels, property names, relationship types, and variables are **case-sensitive**.

Field terminator

Also known as delimiter, a field terminator is a character used to separate each field in a CSV file. In this example, a comma (`,`) is used, but other characters, such as a tab (`\t`) or a pipe (`|`) also work and they can be blended:

```
personId,name,birthYear
23945,"Gerard Pires"|1942
553509,"Helen Reddy"|1941
113934,"Susan Flannery"|1939
```

Tip:

You can copy and paste the example into any text editor (e.g. Notepad or TextEdit) or a spreadsheet application (e.g. Excel or Google Sheets) and then save it as a CSV file.

Header

The header is typically the first line in the CSV file. They are not mandatory, but adding them is a good practice. In the previous example, the header is:

```
personId,name,birthYear
```

If your CSV files have no header row, you need to know the order of the columns and refer to them using indexes instead. This can make working with the data more complicated than it needs to be.

Quotes

Quotation marks (`"`) define what text should be stored as a single value. For example:

```
personId,name,birthYear
23945,"Pires, Gerard",1942
553509,"Reddy, Helen",1941
113934,"Flannery, Susan",1939
```

This CSV file uses commas as the field terminator character and, in the second row `names`, entries such as `Pires, Gerard` also contain a comma. If the entry `Pires, Gerard` wasn't enclosed with quotation marks, it would count as two different entries (i.e. one for `Pires` and another for `Gerard`).

Normalized data

If the source data is normalized (e.g. when exported from a relational data model), there are typically multiple CSV files. Each CSV file represents a table in the relational data model, and the files are related to each other by unique IDs.

In this normalized data example, there are three files for people, movies, and roles:

`person.csv`

```
personId,name,birthYear
23945,Gerard Pires,1942
553509,Helen Reddy,1941
113934,Susan Flannery,1939
```

`movies.csv`

```
movieId,title,avgVote,releaseYear,genres
189,Sin City,8.000000,2005,Crime|Thriller
2300,The Fifth Element,7.700000,1997,Action|Adventure|Sci-Fi
11969,Tombstone,7.800000,1993,Action|Romance|Western
```

`roles.csv`

```
personId,movieId,character
2295,189,Marv
56731,189,Nancy
16851,189,Dwight
```

Note that the `person.csv` file has a unique ID for every person, and the `movies.csv` file has a unique ID for every movie. The `roles.csv` file relates a person to a movie and provides the characters. It corresponds to the relationships in the graph as it contains the links needed to bind nodes together.

De-normalized data

De-normalized data typically represents data from multiple tables. If the source data is de-normalized, there is typically a single CSV file which contains all the data, often duplicated where there are relationships between entities. For example:

`movies-n.csv`

```
movieId,title,avgVote,releaseYear,genres,personType,name,birthYear,character
2300,The Fifth Element,7.700000,1997,Action|Adventure|Sci-Fi,ACTOR,Bruce Willis,1955,Korben Dallas
2300,The Fifth Element,7.700000,1997,Action|Adventure|Sci-Fi,ACTOR,Gary Oldman,1958,Jean-Baptiste Emanuel
Zorg
2300,The Fifth Element,7.700000,1997,Action|Adventure|Sci-Fi,ACTOR,Ian Holm,1931,Father Vito Cornelius
11969,Tombstone,7.800000,1993,Action|Romance|Western,ACTOR,Kurt Russell,1951,Wyatt Earp
11969,Tombstone,7.800000,1993,Action|Romance|Western,ACTOR,Val Kilmer,1959,Doc Holliday
11969,Tombstone,7.800000,1993,Action|Romance|Western,ACTOR,Sam Elliott,1944,Virgil Earp
```

Here, the movie and person data (including the IDs) is repeated in different rows every time new information about a particular actor's role is featured. This sort of duplication compromises the graph data structure. In this case, it is then advisable to prepare your file before importing.

File location

When using the `LOAD CSV` command to load your CSV data, the CSV file is accessed via URL, either over the internet:

```
LOAD CSV WITH HEADERS
FROM 'https://data.neo4j.com/importing-cypher/people.csv' AS row
RETURN row
```

Or from a local folder, if you use an on-premise deployment. In this case, you need to add a `file:///` prefix before the file name:

```
LOAD CSV WITH HEADERS
FROM 'file:///people.csv' AS row
RETURN row
```

Due to security reasons, local files, by default, can only be read from the Neo4j import directory, which is located differently based on your operating system. See Operations → File locations for more information.

If you want to open your CSV file from another location, you need to change the `server.directories.import` settings.

Security

It is **strongly recommended** to permit resource loading only over secure protocols such as HTTPS instead of insecure protocols like HTTP. This can be done by limiting the load privileges to only trusted sources that use secure protocols.

If allowing an insecure protocol is unavoidable, you need to add the JVM argument `-Dsun.net.http.allowRestrictedHeaders=true` to the configuration setting `server.jvm.additional` to avoid automatic blocking from Neo4j's built-in security checks.

File preparation

Before you import CSV data you should consider the **source** of the data. It can come from:

- Relational databases
- Web APIs
- Public data directories
- BI tools
- Speadsheets (e.g. Excel or Google Sheets)

Most data systems have an option for exporting data as CSV files as it is a common format for data exchange. However, real-world data is often messy, which means some values need to be cleaned up or transformed before imported to another system.

These are some common issues you may encounter:

1. **The source files contains more data than you need**

   For example, if you are interested in only one director and the movies they were involved in, the Movies dataset contains a lot of data which is irrelevant to you. To make the import process more efficient, you need to remove the unnecessary data *before* you import the CSV files.

2. **Inconsistency between headers and data**

   Headers can be inconsistent with data. They could be missing or be lost in too many columns.

   To avoid this problem:

   - Check if headers match the data in the file.
   - Adjust formatting, columns, etc *before* you import for a smooth process.

3. **Extra or missing quotes**

   Standalone double (`"`) or single quotes (`'`) in the middle of non-quoted text or non-escaped quotes in quoted text can cause issues when reading the file for loading. It is best to either escape **or** remove stray quotes. Find documentation for proper escaping in the Cypher style guide.

4. **Special or newline characters**

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.

5. **Inconsistent line breaks**

   Ensure line breaks are consistent throughout the file. For Linux users, the recommendation is to use the Unix style for compatibility.

6. **Binary zeros, BOM byte order mark (2 UTF-8 bytes) or other non-text characters**

   Unusual characters or tool-specific formatting are sometimes hidden in application tools. You can use a basic editor to detect and remove this type of characters from your files or use backticks to escape them.

Data types

Since Neo4j reads all imported values as a `string`, you need to convert any values that are not `string`. You can do it using functions in Cypher:

- **`toInteger()`**: converts a value to an `integer`.
- **`toFloat()`**: converts a value to a `float` (e.g. for monetary amounts).
- **`datetime()`**: converts a value to a `DateTime`.

Depending on what sort of data you have in your CSV file, you need to convert values according to their type. See Cypher → Values and types for more information on what values and types are available in Cypher.

Cleaning up

Some issues in the CSV files need to be addressed **before** you load them, but others can be addressed **while** you load them by adding additional clauses to the `LOAD CSV` command.

Null values

Neo4j does not store null values, but you can skip or replace them with default values by adding clauses or functions to the `LOAD CSV` command. 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 third and the fourth lines have no entry for some of the headers, which means that they have null values that need to be skipped. You can use the `WHERE` clause to specify it:

```
LOAD CSV WITH HEADERS FROM 'file:///companies.csv' AS row
WITH row WHERE row.Id IS NOT NULL
MERGE (c:Company {companyId: row.Id});
```

Or have a default value (e.g. "Unknown") set for them and use the `coalesce` function:

```
LOAD CSV WITH HEADERS FROM 'file:///companies.csv' AS row
MERGE (c:Company {companyId: row.Id, hqLocation: coalesce(row.Location, "Unknown")})
```

You can also change the empty `strings` to null values which will not be stored using the `SET` clause:

```
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
```

Conditional conversions

Conditional conversions can be achieved with `CASE`. The previous example checked for null values or empty `strings`, but you can also set a property in this cleaning stage based on a value in the CSV file.

For example, you can set the `businessType` property based on an abbreviated value in the CSV file:

```
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 *
```

Lists as entries

If you have a field in the CSV file that is a list of items that you want to split into separate rows, you can use the Cypher `split()` function to separate arrays in a cell. For example:

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
```

Both Joe and Trevor have multiple skills listed in this file. You can split them using the `split()` function with the `UNWIND` clause like this:

```
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)
```

Clean-up tools

You can use the following third-party tools to make sure your CSV file is in good shape to allow you to import data efficiently:

- CSVKit a set of Python tools that provides statistics (csvstat), search (csvgrep), and more.
- 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.

File size

You can use most Neo4j's import methods for importing small or medium-sized datasets (up to 10 million records). If you want to import larger datasets, it is recommended to use `neo4j-admin database import`. See the tutorial for Neo4j-admin import to learn more.

Optimization

Performance can be a problem when working with large amounts of data or complex loading. Some strategies can, however, improve the processing of large amounts of information.

For example, if you want to create a graph using the preceding **companies.csv** file and the following one:

people.csv

```
employeeId,Name,companyId
1,Bob Smith,1
2,Joe Jones,3
3,Susan Scott,2
4,Karen White,1
```

In this case, you should separate node and relationship creation on a separate part of the processing. For example, instead of the following:

```
LOAD CSV WITH HEADERS FROM 'file:///people.csv' AS row
MERGE (e:Employee {employeeId: row.employeeId})
MERGE (c:Company {companyId: row.companyId})
MERGE (e)-[r:WORKS_FOR]->(c)
```

You can write it like this:

Load the `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 the `Company` *nodes*

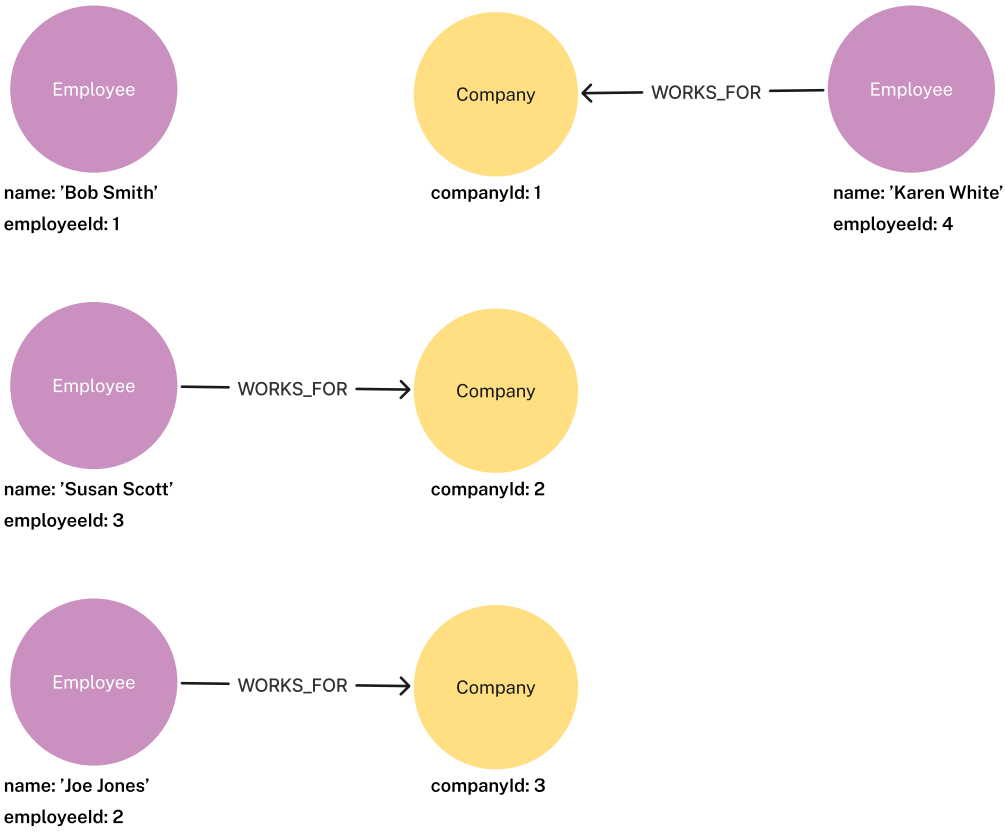
```
LOAD CSV WITH HEADERS FROM 'file:///people.csv' AS row
MERGE (c:Company {companyId: row.companyId})
RETURN count(c);
```

Create relationships

```
LOAD CSV WITH HEADERS FROM 'file:///people.csv' AS row
MATCH (e:Employee {employeeId: row.employeeId, name: row.Name})
MATCH (c:Company {companyId: row.companyId})
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.

The result is this graph:



Import CSV data using `LOAD CSV`

Neo4j supports different methods of importing data. Which method to use depends on factors like dataset size, deployment method, and familiarity with Cypher. This guide shows how to import CSV data into Neo4j using `LOAD CSV` command.

> **Tip:**
>
> For a more hands-on experience, enroll in the GraphAcademy course Importing CSV data into Neo4j. To import larger CSV files into Neo4j, see the tutorial for Neo4j-admin import.

Loading the file

The `LOAD CSV` command can be used to load data into any deployment of Neo4j, whether it is an Aura instance or a local installation. See deployment options for information. The command looks like this:

```
LOAD CSV [WITH HEADERS] FROM url [AS alias] [FIELDTERMINATOR char]
```

What the command does is:

- `WITH HEADERS` **(optional)** → the first line of the CSV file is treated as a header and each row is treated as a map of key-value pairs rather than a list of values.
- `FROM` **(mandatory)** → specifies the location whether it is local or over the internet.
- `AS alias` **(optional)** → names each row for reference.
- `FIELDTERMINATOR` **(optional)** → the default field terminator in CSV files is the comma, but others are supported and can be specified here.

You can import local files. However, in this guide, you will use a link to load the CSV file:

```
LOAD CSV WITH HEADERS
FROM 'https://data.neo4j.com/importing-cypher/people.csv' AS row
RETURN row
```

The result is:

row

{
"birthYear": "1942",
"name": "Gerard Pires",
"personId": "23945"
}

{
"birthYear": "1941",
"name": "Helen Reddy",
"personId": "553509"
}

{
"birthYear": "1939",
"name": "Susan Flannery",
"personId": "113934"
}

Filtering loaded data

Alternatively, you can also use `LOAD CSV` to load only a subset of the data in the CSV file. For example, you can import information only about people born in 1942:

```
LOAD CSV WITH HEADERS
FROM 'https://data.neo4j.com/importing-cypher/people.csv'
AS row
WITH row WHERE row.birthYear = '1942'
```

```
RETURN row
```

The result is:

row
{ "birthYear": "1942", "name": "Gerard Pires", "personId": "23945" }

Note that in the query, you need to refer to the year of 1942 as the `string` datatype (encased by single quotes `'`). This is because `LOAD CSV` reads all values as a `string`. In the next step, you will learn how to convert the values to their respective types.

Convert data types

In the **people.csv** file, you have data such as the birth year and the ID number for each person. `LOAD CSV` only reads **all** values as `string`. For better processing of the data, you can convert these values into temporal and numerical types.

`personId` can be turned into an interger by using the `toInteger()` function, and `birthYear` can be turned into a date format by using the `date()` function:

```
LOAD CSV WITH HEADERS
FROM 'https://data.neo4j.com/importing-cypher/people.csv'
AS row
WITH toInteger(row.personId) AS personId, date(row.birthYear) AS birthYear
RETURN personId, birthYear
```

Result

personId	birthYear
23945	"1942-01-01"
553509	"1941-01-01"
113934	"1939-01-01"

Note that the property value for the `birthYear` only contains the year, but when converted into a `date` type, Cypher automatically adds January 1st as the date of the specified year. This is to comply with the format of the `date` data type, `YYYY-MM-DD`.

Check the imported data

You may want to verify that your data has been imported correctly. These are some things you can do to ensure that the imported data is accurate:

1. Count the number of rows in the CSV file and compare it to the number of rows returned by the `LOAD`

`CSV` clause by using the `COUNT` function in a new query:

```
LOAD CSV WITH HEADERS
FROM 'https://data.neo4j.com/importing-cypher/people.csv'
AS row
RETURN count(row)
```

2. Make sure header names match those in the CSV file.

Keep learning

Regardless of where your data comes from, it is likely that it needs some preparation before it is ready to be imported. See Working with CSV files to learn more about the structure of data, how to clean it up, and optimize it.

Alternatively, you can follow the the Tutorial: Create a graph data model to learn what to do next after importing data into Neo4j.

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 running the following command:

```
:params {stravaToken: "Bearer <insert-strava-token>"}
```

Important:

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: Load JSON
- APOC Documentation: StackOverflow JSON Data Example

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.

- 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.
- Data Importer: UI-based tool for importing data into Neo4j. Available both as a standalone tool and as

a service in Aura.

- 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.
- 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.
- Apache Hop: 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.

Note that this import option is not supported officially.

- 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.
- 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.

> **Note:**
>
> 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

Neo4j Data Importer

Refer to Neo4j Data Importer documentation for a complete guide.

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

Apache Hop

Apache Hop, an abbreviation for Hop Orchestration Platform, is a data orchestration and data engineering

platform. It was designed to facilitate creation and management of data flow.

The source code from the Neo4j Apache Hop project has been integrated into the Apache Hop framework. Recent versions include the Neo4j plugins as well.

With Apache Hop, you can load large datasets in Neo4j, update graphs, and get logging information.

Note:

Note that this tool is community-contributed and not supported officially.

Apache Hop resources

- **Documentation:** Apache Hop Neo4j
- **Neo4j plugins:** Use Apache Hop with Neo4j
- **Tutorial:** 5 minutes to load data to Neo4j and other graph databases with Apache Hop

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

[1] Enterprise only

Create an application

This page lists the officially supported libraries and APIs that you can use to create an application.

Libraries

Documentation	GraphAcademy course
Python	Building Neo4j Applications with Python
JavaScript	Building Neo4j Applications with TypeScript
Java	Building Neo4j Applications with Java
.NET	Building Neo4j Applications with .NET
Go	Building Neo4j Applications with Go
GraphQL	Introduction to Neo4j & GraphQL
Node.js	Building Neo4j Applications with Node.js
Spring Data	Building Neo4j Applications with Spring Data
Neo4j OGM	-

APIs

Neo4j supports the following APIs:

- HTTP API **(Not available on Aura)** and Query API → allow executing Cypher statements against a Neo4j server through HTTP requests. The main use case of these APIs is for developing client applications in languages for which there is no supported library.
- Change Data Capture → allows capturing and tracking changes to your database in real-time, enabling you to keep your other data sources up to date with Neo4j.

Using Neo4j from Java

To connect to Neo4j from a Java application, you can use either the Java driver or the JDBC driver. Both are officialy supported.

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.

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

```
<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>1.17.6</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>neo4j</artifactId>
    <version>1.17.6</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.

```
spring.neo4j.uri=neo4j+s://abcd.databases.neo4j.io
spring.neo4j.authentication.username=neo4j
spring.neo4j.authentication.password=secret
```

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.

```
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

Projects	Spring Data Neo4j
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

> *For Java developers who use Quarkus and want to take advantage of a pre-configured Java driver instance as well as the integration with OGM. Please consult the linked documentations for more information.*

- 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 Quarkus.
- For this library, please use JDK 17 or later, a recent Maven installation, and your favorite IDE.
- If you want to run a Neo4j database locally, you also need Docker installed.

What we will cover

- Movies example application
- Quarkus Neo4j driver integration
- Neo4j OGM integration
- Health & metrics support

Getting started

Over the next few sections, we will walk through all of the steps for creating a Quarkus application with Neo4j.

You can fetch the Movies example project from GitHub:

```
git clone https://github.com/sdaschner/movies-java-quarkus/
cd movies-java-quarkus/
```

Building & running locally

You can build the project via Maven:

```
mvn package
```

Additionally, you can execute the integration test, which fires up a local Neo4j instance via Testcontainers:

```
mvn test-compile failsafe:integration-test failsafe:verify
```

You run the Quarkus application via executable JAR:

```
java -jar target/quarkus-app/quarkus-run.jar
```

This starts up your Quarkus application which then is available under http://localhost:8080/

Now, you can give it a try and explore the movies queries examples.

Run the database locally

Per default, the example runs with a generally-available database under https://demo.neo4jlabs.com:7473

You can also run a local Neo4j database with the example project as Docker container. For this, execute the `./run-graph-db.sh` script, and change the Quarkus application properties under *src/main/resources/application.properties*.

```
# run in a separate shell, starts up a Docker container from neo4j:4.4.12
./run-graph-db.sh
```

Quarkus application.properties

```
quarkus.neo4j.uri=bolt://localhost:7687
quarkus.neo4j.authentication.username=neo4j
quarkus.neo4j.authentication.password=test
```

Quarkus dev mode

In order to improve the development experience and to quickly change code, you can use Quarkus' dev mode, which is compatible with the Neo4j and Neo4j OGM extensions:

```
mvn quarkus:dev
```

This also starts up your application with port 8080, but keeps the connection to your source code and allows for quick redeploys.

Congratulations! Now you have everything to develop Quarkus applications with Neo4j.

The following explains how our Quarkus application accesses the Neo4j database, and how the OGM mapping is integrated.

Understanding the examples

Our Quarkus application uses the Neo4j-OGM Quarkus extension that we recommend to use in case you want to make use of the Object Graph Mapper.

The *pom.xml* includes this dependency which transitively adds the Neo4j Quarkus extension and OGM dependencies:

Neo4j-OGM Quarkus dependency

```
<dependency>
  <groupId>org.neo4j</groupId>
  <artifactId>neo4j-ogm-quarkus</artifactId>
  <version>1.5.1</version>
</dependency>
```

With that being included, your Quarkus application configures the Neo4j driver and sets up the OGM mapping session factory as injectable bean.

You can get an idea of the OGM session factory usage in the following classes:

```
@ApplicationScoped
public class Searches {

    @Inject
    SessionFactory sessionFactory;

    public List<Movie> searchMoviesByTitle(String title) {
        Session session = sessionFactory.openSession();
        Iterable<Movie> iterable = session.query(Movie.class, "MATCH (movie:Movie) WHERE movie.title
CONTAINS $title RETURN movie", Map.of("title", title));

        // [...]
    }
}
```

The domain entities, such as *Movie* are declared as OGM node entity classes:

```
@NodeEntity
public class Movie {

    @Id
    public String title;
    public String tagline;
    public Integer released;
    public int votes;

    @Relationship(value = "DIRECTED", direction = INCOMING)
    @JsonbTypeSerializer(PersonNamesSerializer.class)
```

```
    public Set<Person> directors = new HashSet<>();

    @Relationship(value = "WROTE", direction = INCOMING)
    @JsonbTypeSerializer(PersonNamesSerializer.class)
    public Set<Person> writers = new HashSet<>();

    @Relationship(value = "PRODUCED", direction = INCOMING)
    @JsonbTypeSerializer(PersonNamesSerializer.class)
    public Set<Person> producers = new HashSet<>();

    @Relationship(value = "REVIEWED", direction = INCOMING)
    @JsonbTypeSerializer(ReviewsSerializer.class)
    public Set<Review> reviewers = new HashSet<>();

    @Relationship(value = "ACTED_IN", direction = INCOMING)
    @JsonbTypeSerializer(ActsSerializer.class)
    public Set<Act> actors = new HashSet<>();
}
```

Have a look at the Object Graph Mapper docs for a more detailed explanation. The `@JsonbTypeSerializer` annotation controls how the entity objects are mapped to JSON for the JAX-RS REST endpoints.

You can follow the code in the `MovieResource`, `SearchResource`, `ActorsResource`, and `GraphResource` JAX-RS classes to comprehend the individual use cases.

Another helpful OGM feature added in `4.0` is the mapping of DTO classes and Java records. These types are mapped from arbitrary query results and the corresponding classes don't have to be annotated. For an example see the `Persons` class and the usage of the `session.queryDto()` method:

```
@ApplicationScoped
public class Persons {

    @Inject
    SessionFactory sessionFactory;

    public List<ActorRecommendation> recommendCoActor(String name) {
        Session session = sessionFactory.openSession();
        return session.queryDto(" MATCH (actor:Person {name: $name}) [...] " +
                  " [...] " +
              " RETURN cocoActors.name AS actor, count(*) AS strength ORDER BY strength DESC",
              Map.of("name", name), ActorRecommendation.class);
    }
}
```

```
public record ActorRecommendation(String actor, long strength) {

}
```

Quarkus Neo4j features

In the following, we have a look at the integration features that are available with Quarkus and Neo4j.

Driver integration

The goal of the Quarkus Neo4j integration is to provide support for getting a managed instance of the Neo4j driver. You can provide the driver properties via the Quarkus configuration mechanisms, usually in the *application.properties* file, 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.

You probably noticed that we're not using this injection in our Movies example, but instead injected the OGM session factory. Both works and it depends on your use case and application setup, which way you choose.

Additional to the managed driver bean creation, the integrations also expose health metrics for the driver and connection to your Neo4j instance.

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>
```

If you're using the Neo4j-OGM Extension, this dependency will be included transitively and shouldn't be declared explicitly.

You can 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
```

Health check integration

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>
```

Metrics integration

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
```

OGM Integration

Your Quarkus application can be integrated with Neo4j OGM in order to provide declarative object mappings for your domain entities. There is an official Quarkus extension available, that we recommend to use unless you have a reason not to do so.

Neo4j-OGM Quarkus dependency

```
<dependency>
  <groupId>org.neo4j</groupId>
  <artifactId>neo4j-ogm-quarkus</artifactId>
  <version>1.5.1</version>
</dependency>
<!-- with this, you can remove io.quarkus:quarkus-neo4j from your pom.xml again -->
```

This dependency transitively includes Neo4j OGM and the Quarkus Neo4j dependency, so it's the only Neo4j dependency you need in your *pom.xml*.

The Neo4j-OGM Quarkus dependency configures the OGM session factory and makes it injectable as bean:

```
@Inject
SessionFactory sessionFactory;
```

Resources

Quarkus Documentation	Neo4j integration, Configuration properties, Guide
Neo4j-OGM Quarkus Extension	GitHub
▶ Examples	Quarkus examples, Quarkus Movies example application

Helidon, Micronaut

> *For Java developers who use 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 both 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.

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

Helidon Documentation	Reference, Helidon Neo4j
Micronaut Documentation	Neo4j integration, Guide
Examples	Helidon, Micronaut and more examples

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 capabilities using procedures and functions. Below 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.

Using Neo4j from .NET

Using Neo4j from JavaScript

Using Neo4j from Python

Using Neo4j from Go

Neo4j OGM

Community-contributed libraries

Introduction

In addition to the officially supported drivers, you can find their Community alternatives. Besides Java, .NET, JavaScript, Go, and Python drivers, Neo4j Community offers support for Ruby, PHP, Perl, and Rust. Links to their relevant resources are provided below.

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.

Important:

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 from Ruby

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.

Website	https://neo4jrb.io/
Authors	Heinrich, Amit, Brian, Chris, Andreas
Package	neo4j-ruby-driver, neo4j-java-driver, activegraph

Source	https://github.com/neo4jrb
Docs	https://neo4jrb.readthedocs.org/en/latest/
Blog	https://blog.brian-underwood.codes/
Protocols	Bolt

Using Neo4j from PHP

Alternatively, Neo4j can be installed on any system and then accessed via its Bolt and HTTP APIs. We recommend the Neo4j PHP client for easiest development over Bolt and HTTP APIs. You can also directly access the Bolt protocol via the PHP Bolt library.

Neo4j PHP 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 is being actively developed. For more details, refer to a README file on the Github page.

Author	Ghlen Nagels
Source	https://github.com/neo4j-php/neo4j-php-client
Package	https://packagist.org/packages/laudis/neo4j-php-client
PHP	7.4 / 8.0+
Neo4j	3.5 / 4.0+
Protocols	Bolt, HTTP
Example App	https://github.com/neo4j-examples/movies-neo4j-php-client

PHP Bolt

A low level driver for the Bolt protocol in PHP.

Author	Michal Stefanak
Source	https://github.com/neo4j-php/Bolt
PHP	7.4+ / 8.0+
Neo4j	3.0+ / 4.0+ / 5.0+
Protocols	Bolt

Using Neo4j from Perl

Neo4j::Driver

This Perl driver 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
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
Source	https://github.com/majensen/perlbolt

Java Community drivers

Scala: neotypes

Author	Dmitry Fedosov
Source	https://github.com/neotypes/neotypes
Docs	https://neotypes.github.io/neotypes/
Blog Post	http://dimafeng.com/2018/12/27/neotypes-1/
Example	https://github.com/neotypes/examples

.NET Community drivers

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.

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.

Source	https://github.com/DotNet4Neo4j/Neo4j.Driver.Extensions
NuGet Package	https://nuget.org/packages/neo4j.driver.extensions
Authors	Charlotte Skardon
Docs	Introduction blogpost

Python Community drivers

Neomodel

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
Source	http://github.com/neo4j-contrib/neomodel
Docs	https://neomodel.readthedocs.io/en/latest/
Python	2.7 / 3.3+
Protocols	Bolt
Example	https://github.com/neo4j-examples/neo4j-movies-python-neomodel

Go Community drivers

GoGM: Golang Object Graph Mapper

Author	Eric Solender, CTO and co-founder of Mindstand
Source	https://github.com/z5labs/gogm
Docs	https://github.com/mindstand/gogm/blob/master/README.md

Using Neo4j from Rust

neo4rs

Neo4j can be used from Rust using the neo4rs driver. **neo4rs** supports using Neo4j via **Bolt**, starting from Neo4j 4.4 up until the most recent version.

You can also ask questions on the Neo4j Community Discord in the `#drivers` channel.

Authors	knutwalker
Source	https://github.com/neo4j-labs/neo4rs
Package	neo4rs on crates.io
Docs	neo4rs on docs.rs
Example	https://github.com/neo4j-examples/movies-rust-bolt
Rust	1.75+
Neo4j	4.4+
Protocols	Bolt

Connect data sources

Data science with Neo4j

Introduction

With a native graph database at the core, Neo4j offers Neo4j Graph Data Science — a library of graph algorithms for analysts and data scientists.

The library includes algorithms for community detection, centrality, node similarity, pathfinding, and link prediction. Graph Data Science (GDS) is designed to support data science workflows and machine learning tasks over your graphs.

Graph algorithms are exposed through Cypher procedures. Cypher is a declarative graph query language created by Neo4j. For more information about Cypher, visit the Cypher Manual.

In Neo4j GDS, the typical workflow is the following:

- Read the graph data from the Neo4j database.
- Create a graph projection — loading the data into an in-memory graph.
- Run a graph algorithm on a projection.
- Write the results back to the projected graph and/or to the Neo4j database.

To learn more about the GDS library capabilities, go to the official documentation.

Setting up the environment

There are several options on how to get started with GDS in Neo4j. You can select a self-hosted or a fully managed cloud edition.

- Neo4j AuraDS is the data science solution as a fully managed **cloud** service that unifies the ML surface and graph database into a single workspace. AuraDS documentation tells more about its features and provides **usage examples**.
- If you prefer to use on-premises solutions, you can:
  - install GDS as a plugin in Neo4j Desktop, client-side application to work with Neo4j,
  - download `neo4j-graph-data-science-[version].zip` from the Neo4j Download Center and follow instructions described in the Neo4j GDS Library Manual → Neo4j Server,
  - configure the GDS library as a Neo4j Docker plugin if you run Neo4j in a Docker container.

  Two GDS editions are available: Community and Enterprise. To compare their features, visit Neo4j Pricing page.

  Important:

  Note that the GDS library has to be compatible with Neo4j. Before installing the GDS library, consult the compatibility matrix in the Neo4j GDS Manual → The supported Neo4j versions.

If you run Neo4j in a cluster, you can follow the same instructions for the Neo4j Server with some additional considerations.

- If you are new to graph technology and Neo4j, GraphAcademy courses could be a great starting point. They are free of charge, interactive, and hands-on. You can select a learning path specifically designed for data scientists.

  Besides, you can use Neo4j Sandbox for learning graph concepts and Cypher.

Integrating Neo4j GDS with your data ecosystem

Minimizing friction around data movement makes the adoption of any product much easier. Bearing that in mind, Neo4j provides multiple connectors, drivers, and libraries that allow easy data integration:

- GDS Python client to call all Neo4j GDS procedures straight from Python.
- Data Warehouse Connector for moving data to and from Snowflake, Google BigQuery, Amazon Redshift, or Microsoft Azure Synapse Analytics.
- BI Connector for direct access to BI tools like Microsoft Power BI, Tableau, etc.

GDS Python client

If you are a Python oriented person, you can use the GDS Python client package called `graphdatascience`. It enables users to write pure Python code to project graphs, run algorithms, use ML pipelines, and train ML models with GDS. To avoid naming confusion with the server-side GDS library, we refer to the Neo4j GDS client as the *GDS Python client*.

1. To import and set up the GDS Python client, follow instructions in the GDS Client manual → Getting started.
2. To install the GDS Python client, run:

   ```
   pip install graphdatascience
   ```

3. Keep in mind compatibility requirements between the GDS Python client, the Neo4j Python Driver, a server-side installation of the GDS library. Check them in the GDS Client manual.
4. If you use the GDS Python client on AuraDS, run the following:

   ```
   # Replace with the actual URI, username, and password
   AURA_CONNECTION_URI = "neo4j+s://xxxxxxxx.databases.neo4j.io"
   AURA_USERNAME = "neo4j"
   AURA_PASSWORD = "..."

   # Client instantiation
   gds = GraphDataScience(
       AURA_CONNECTION_URI,
       auth=(AURA_USERNAME, AURA_PASSWORD),
       aura_ds=True
   )
   ```

The source code of the GDS Python client is available at GitHub.

Data visualization with Neo4j Bloom

Data visualization is an essential part of data science workflow. That allows data specialists not only to analyze massive amounts of information but also to represent them efficiently. Data visualization tools and technologies can influence data-driven decisions.

Neo4j offers a low-code visualization tool — Neo4j Bloom, designed to explore and dynamically visualize big graphs.

For instructions on how to use Bloom with the GDS library, see Neo4j Bloom Manual → Graph Data Science integration.

Graph Data Science use cases

You can apply the graph data science in all industries to make recommendations, identify anomalies and find fraudsters, improve customer knowledge, and optimize supply chains.

The documentation provides instructions on how to apply graph algorithms from GDS to real-life use cases.

- In the GDS Client Manual → Tutorials, Jupyter ready-to-run notebooks showcase features of the GDS Python client.
- Usage examples in the AuraDS docs show the GDS workflow components and answer the frequently asked questions on how to estimate memory usage, monitor the progress of a running algorithm, or how to share ML models.

Besides the manuals, you can look for information on the Neo4j YouTube channel.

Ask a Data Scientist about Graph series answers many questions and provide significant insights.

Resources

- Official documentation: The Neo4j Graph Data Science Library Manual
- The Neo4j Graph Data Science Client Manual
- Neo4j AuraDS Documentation
- GraphAcademy: Free online courses
- Use cases and recommendations on how to select a specific algorithm
- Video Series on YouTube: Ask a Data Scientist about Graph

Visualize your data in Neo4j

> *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 product focused on visualization that comes with every AuraDB Instance. It is available both

with Neo4j Desktop and standalone. It was designed for business analysts and other 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.

Neo4j Visualization Library

NVL is a collection of libraries that you can use if you want to build your own graph visualizations. The same visualizations are used in Neo4j Bloom and Explore. For more information about NVL, see NVL documentation and the NVL API documentation.

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 graph visualization tools fall. You will see how each of these categories handles the exported data and be provided with 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

GraphAware Hume

GraphAware Hume is a government-grade data analytics platform for fast, intuitive intelligence analysis. From the user interface to the database, GraphAware Hume is uniquely graph-based, showcasing the full power of Neo4j through code-free UIs.

GraphAware Hume ingests and connects structured and unstructured data sources via configurable data workflows. It supports both streaming and batch processing to establish a unified and up-to-date `single source of truth`.

As more intelligence sources are integrated, specialized graphs can be created to support different types

of analysis. Analysts can leverage graph data science to understand node importance within networks, predict links, and set up automated graph-based alerts.

GraphAware Hume comes with robust deployment options (including containerized and air-gapped setups), strong security features (such as SSO, RBAC, and audit logging), high availability, backup capabilities, and open standards and API, ensuring no vendor lock-in and full ownership of your graph-enabled solutions.

GraphAware Hume Resources

- Product: GraphAware Hume
- Features:
  - Intelligence Analysis
  - Data Ingestion
  - Advanced Intelligence Analysis
  - Intelligence Sharing

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.

Tip:

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 you can populate your 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 you often must transform the results to export from Neo4j into the format expected by these libraries.

You will get a closer look at these tools in the next paragraphs.

Neo4j Visualization Library (NVL)

NVL is a collection of libraries that can be used to build custom graph visualizations like those used in Neo4j Bloom and Explore in the Aura console. NVL is written in TypeScript and can be used in any JavaScript project. It is also available as a React component that can be used in React applications.

NVL Resources

- Documentation: Neo4j Visualization Library
- NVL API documentation: NVL API
- NVL code examples: NVL Code examples

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

Reference

Example datasets

Neo4j offers a variety of example datasets to get started with the products. Some of them are built-in, others can be retrieved from a GitHub repository. This page shows how to access them.

Available datasets

This list includes all available datasets, descriptions, and how to access them from different platforms.

Important:

Some datasets are not continuously maintained. Information may be outdated.

GitHub	Aura	Neo4j Browser	Demo server	Description
Movies	✔	`:guide movies`	`movies`	A small graph containing actors and directors that are related through the movies they have collaborated on. It includes the year when the actors, producers, and directors were born, as well as the year when the movie was released.
Northwind	✔	`:guide northwind`	`northwind`	A graph representing a traditional retail system with products, orders, customers, suppliers, and employees.
StackOverflow	✔	`:guide stackoverflow`	`stackoverflow`	A graph including users, tags, and Q&A data retrieved from the website StackOverflow.
Movie recommendations	✔	✖	`recommendations`	A graph example using a dataset of movie reviews for generating personalized, real-time recommendations.
Crime investigation (POLE)	✔	✖	✖	A Persons Objects Locations Events example data model focused on the relationships between people, objects, locations, and events.
Healthcare analysis	✔	✖	✖	An example graph using FDA Adverse Event Reporting System (FAERS) datasets. It contains information on adverse event and medication error reports submitted to FDA.

GitHub	Aura	Neo4j Browser	Demo server	Description
Game of Thrones	✖	`:guide got`	`gameofthrones`	A graph based on George R. R. Martin's series Game of Thrones. It contains information on the interaction between the characters throughout the books.
FinCEN	✖	✖	`fincen`	A graph with data from the global investigation FinCEN Files concerning money laundering.
Twitter	✖	✖	`twitter`	An example graph based on the structure of a social network, with Neo4j's Twitter data.
BBC recipes	✖	`:guide recipes`	✖	A graph using data from BBC Good Foods. It contains information on ingredients, diet types, recipes, and author of the recipes.
UK companies	✖	`:guide ukcompanies`	✖	A graph containing information on UK company registration, land ownership, and political donation data.
Airbnb listings	✖	`:guide listings`	✖	A graph containing information on Airbnb listings, hosts, and reviews.
Football transfers	✖	`:guide football_transfers`	✖	A graph with transfers data. It includes information on players, clubs, transfers, and countries.
Neoflix	✖	✖	`neoflix`	A graph with over 300 movie nodes including information on release status and date, revenue, budget, language, and synopsis.
WordNet	✖	✖	`wordnet`	A graph using data from WordNet, a large lexical database of English. It groups nouns, verbs, adjectives, and adverbs into sets of cognitive synonyms (synsets), each expressing a distinct concept.
Panama Papers	✖	✖	✖	The Paradise Papers dataset and guide from the International Consortium of Investigative Journalists (ICIJ).
London public transportation network	✖	✖	✖	A graph of the London public transportation network containing information on statios and tube lines.

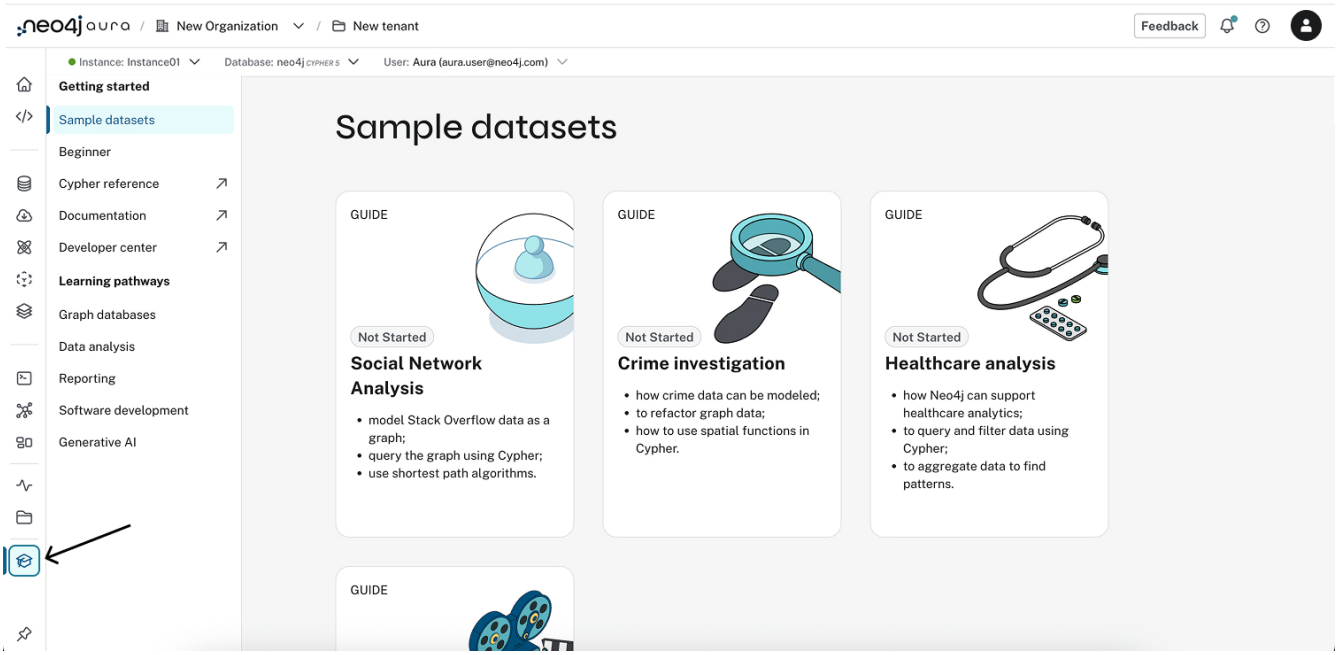
GitHub	Aura	Neo4j Browser	Demo server	Description
IT management	✖	✖	✖	An example graph representing a network and IT management. It contains information for dependency and root cause analysis, and more.

Built-in examples

Some example datasets can be retrieved directly from Aura and Neo4j Browser in the form of guides. With them, you can create or import a dataset and learn how to explore it.

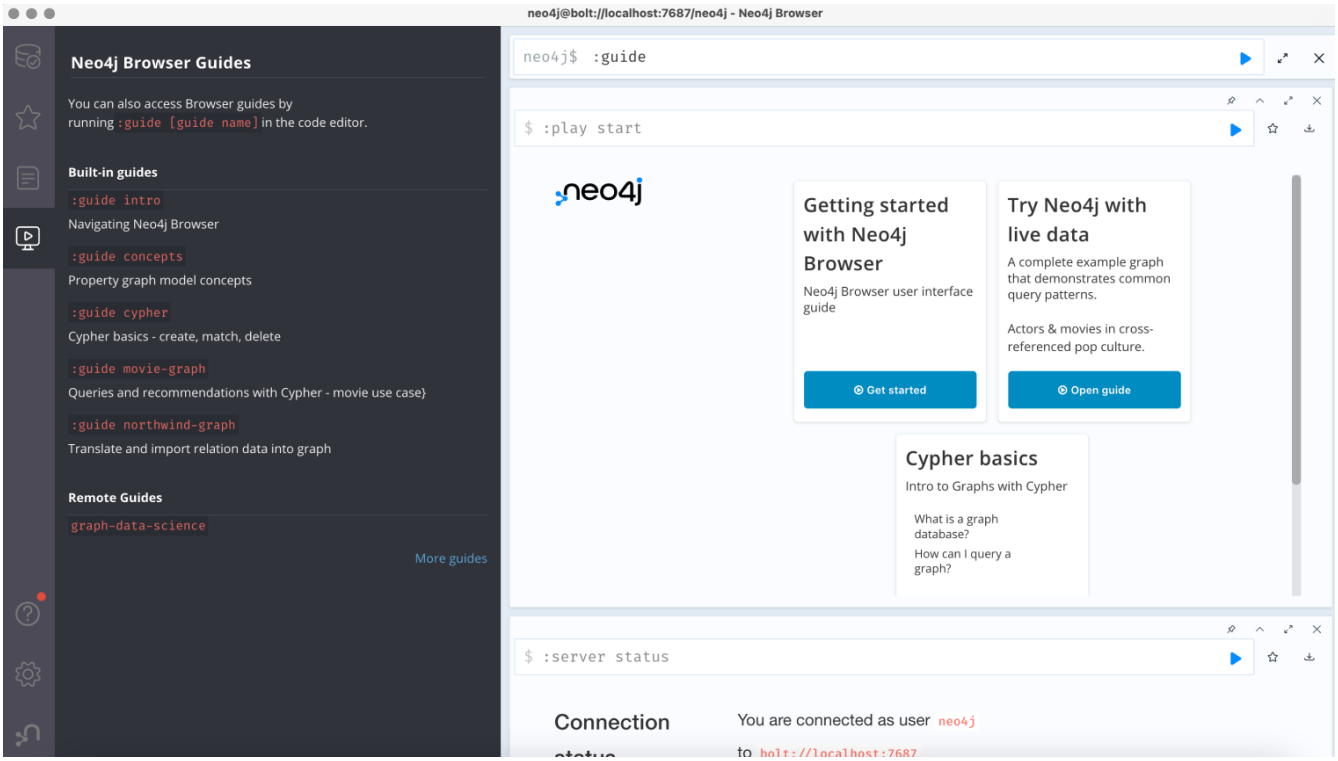
Aura Learning Center

You can access the example datasets through the Learning Center in the Aura console. Select the cap icon 🎓 in the left-hand navigation to see a list of datasets available as learning guides.



Neo4j Browser

Use the command `:guide` to access a list of built-in interactive guides or access them directly using these previously listed commands.



Demo server

Optionally, you can access the demo server on https://demo.neo4jlabs.com:7473 to explore a number of datasets with read-only access for public use.

The username and password are the same as the dataset name. For instance, for the `recommendations` dataset the username is `recommendations` and password is `recommendations` too.

Find the full list of datasets and the username/password entries to use in the available datasets table.

Database dump files

In the GitHub repository Neo4j graph examples, you find dump files for several graph example datasets, including the ones listed previously in the available datasets table.

There are several ways to load them, depending on the environment that is being used:

- Aura
- Neo4j Desktop
- Kubernetes
- Docker
- Neo4j Admin

You can also refer to the Importing your data section to learn more ways to load a dataset to your instance, including other supported file formats.

Important:

The Neo4j version of some of the dump files may be older than your Neo4j version. In this case, you need to upgrade your database dump using the `neo4j-admin database migrate` command before loading it into Neo4j. Note that this command can only be run on a stopped database. For more details, see Upgrade and Migration guide → Migrate your databases.

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.
- 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.

Tutorial: Build a Cypher Recommendation Engine

Graphs are everywhere. By following the meaningful relationships between the people and movies, you can determine occurrences 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*

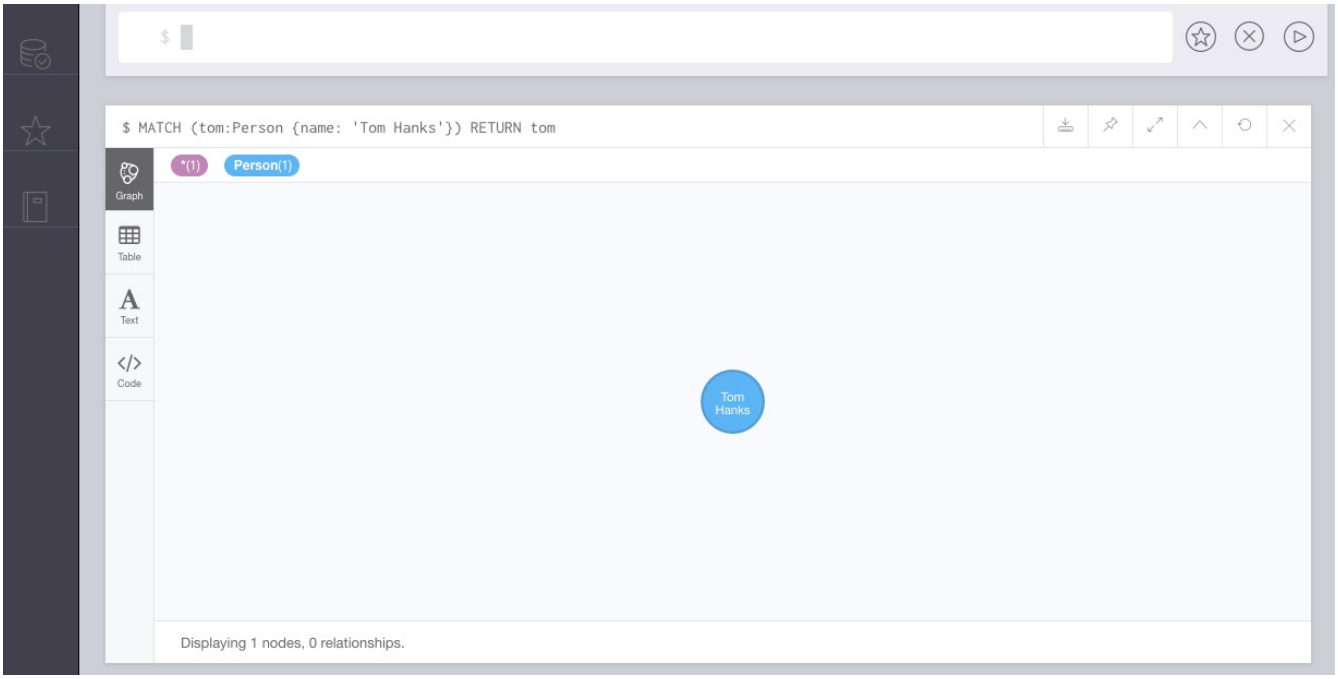
This first query just shows a few movies in the database to prove there's something there. Congratulations, you've got some data in a new database, and we're ready to get started.

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
```

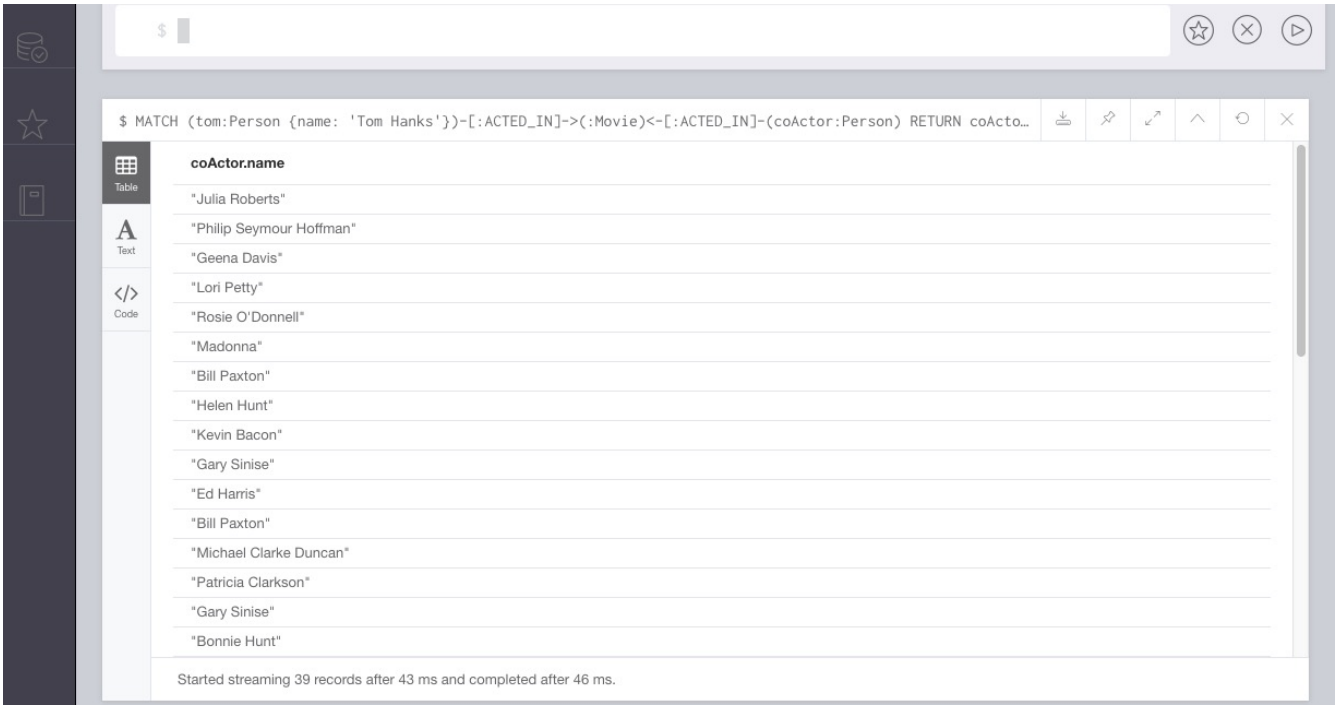


Now that we found an actor we are interested in, we can retrieve all his movies by starting from the `Tom Hanks` node and following the `ACTED_IN` relationships. Your results should look like a graph.

```
MATCH (tom:Person {name: 'Tom Hanks'})-[r:ACTED_IN]->(movie:Movie)
RETURN tom, r, movie
```

Of course, Tom has colleagues who acted with him in his movies. A statement to find Tom's co-actors looks like this:

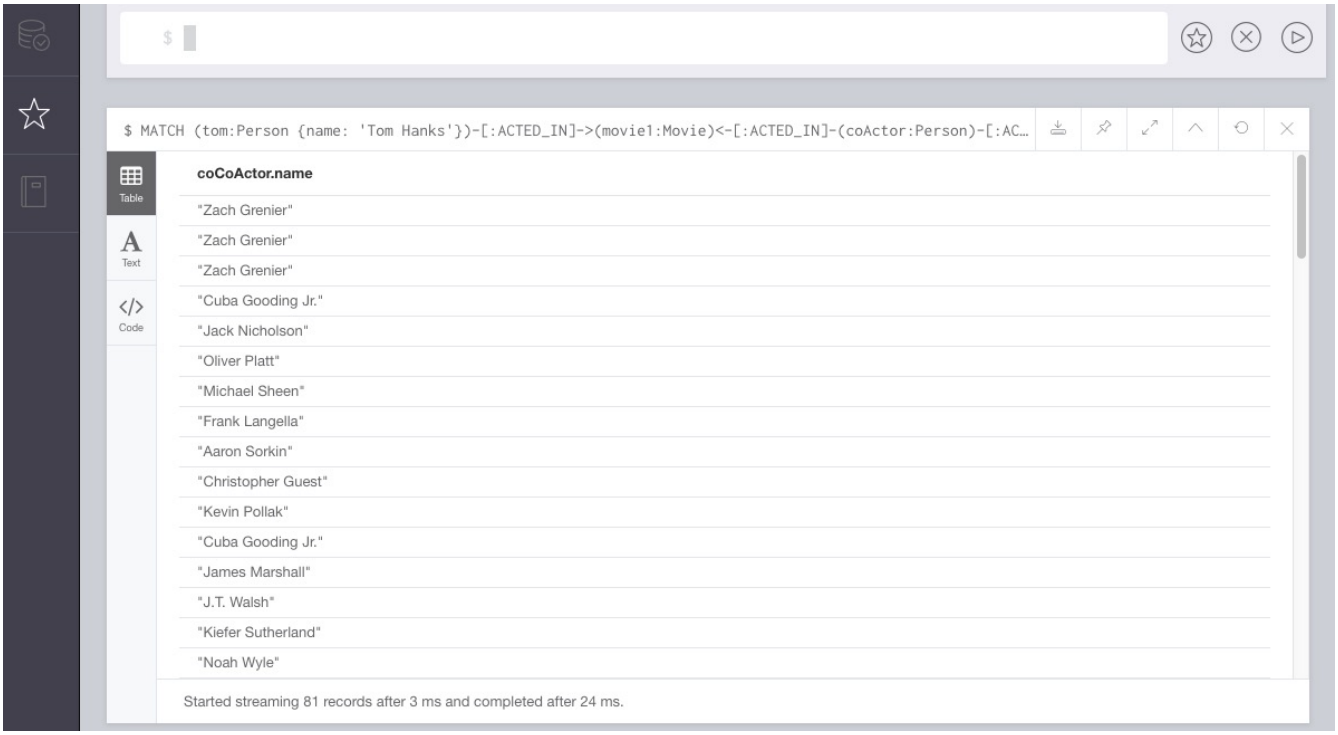
```
MATCH (tom:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(:Movie)<-[:ACTED_IN]-(coActor:Person)
RETURN coActor.name
```



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.

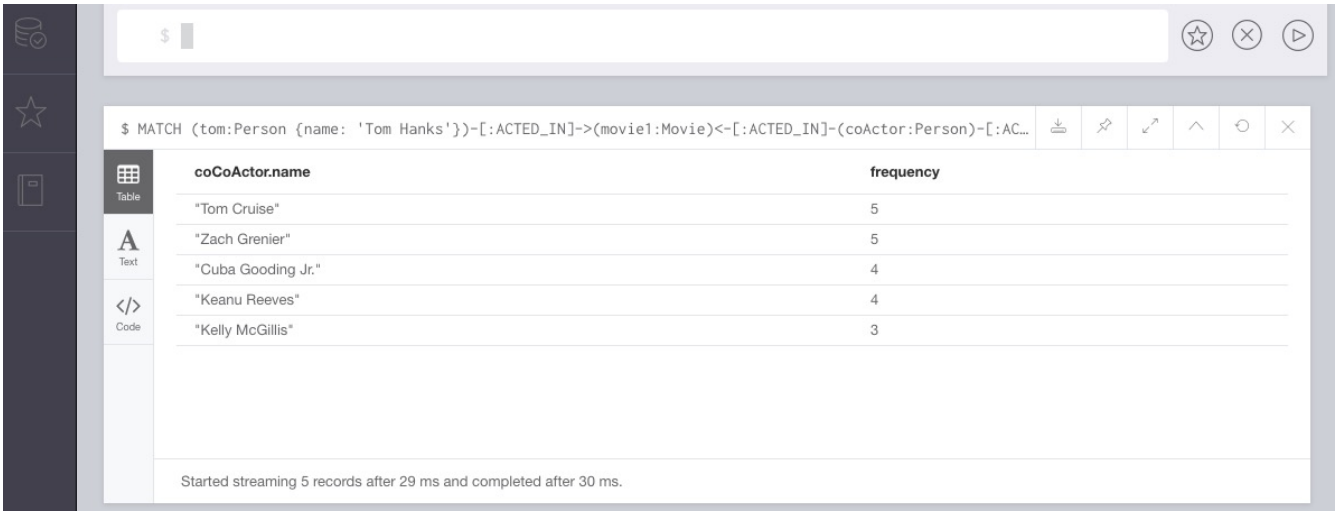
```
MATCH (tom:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(movie1:Movie)<-[:ACTED_IN]-(coActor:Person)-
[:ACTED_IN]->(movie2:Movie)<-[:ACTED_IN]-(coCoActor:Person)
WHERE tom <> coCoActor
AND NOT (tom)-[:ACTED_IN]->(:Movie)<-[:ACTED_IN]-(coCoActor)
RETURN coCoActor.name
```



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.

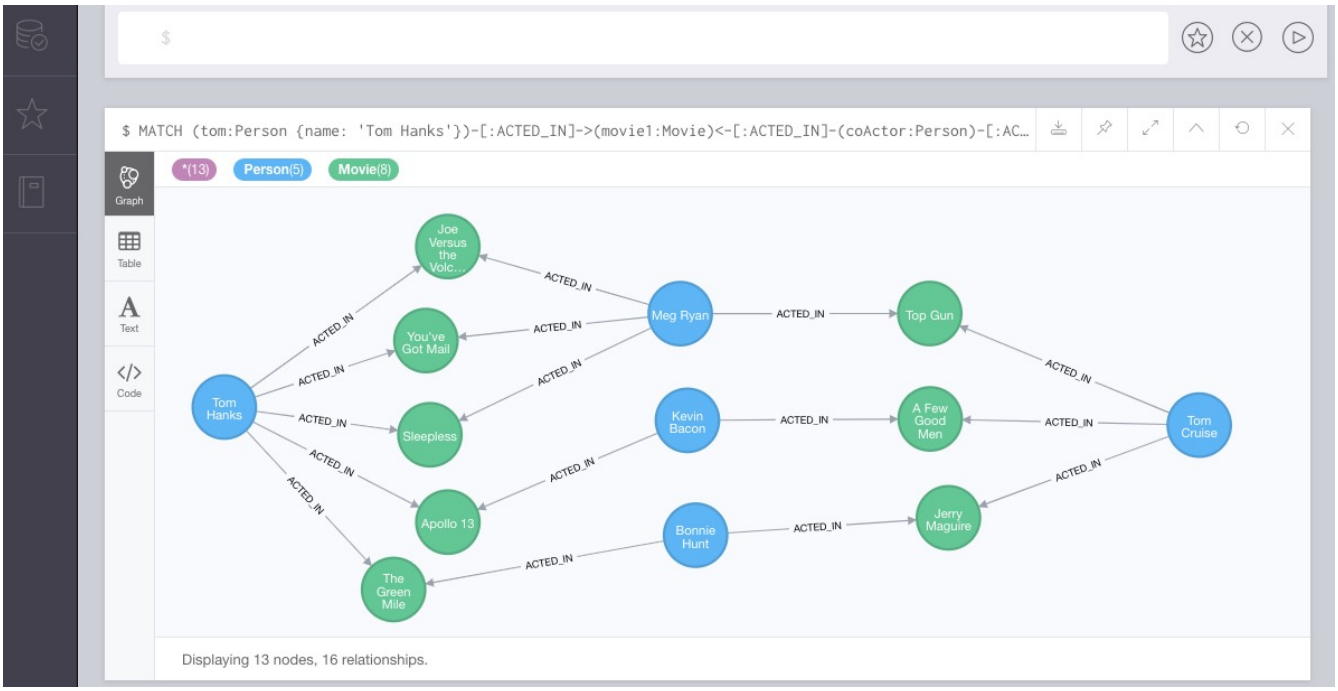
```
MATCH (tom:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(movie1:Movie)<-[:ACTED_IN]-(coActor:Person)-
[:ACTED_IN]->(movie2:Movie)<-[:ACTED_IN]-(coCoActor:Person)
WHERE tom <> coCoActor
AND NOT (tom)-[:ACTED_IN]->(:Movie)<-[:ACTED_IN]-(coCoActor)
RETURN coCoActor.name, count(coCoActor) as frequency
ORDER BY frequency DESC
LIMIT 5
```



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
```



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.

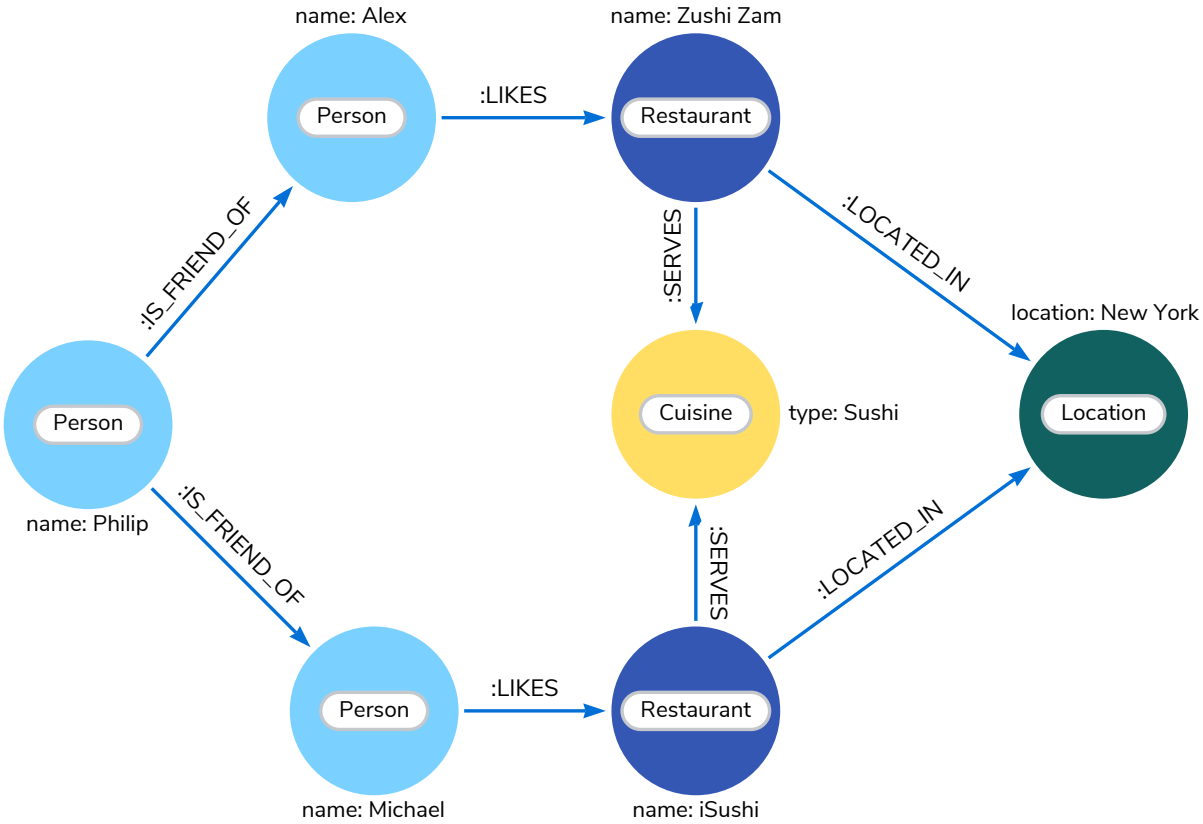
With these two simple Cypher statements, we already created two recommendation algorithms - **who to meet/work with next** and **how to meet them**.

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.

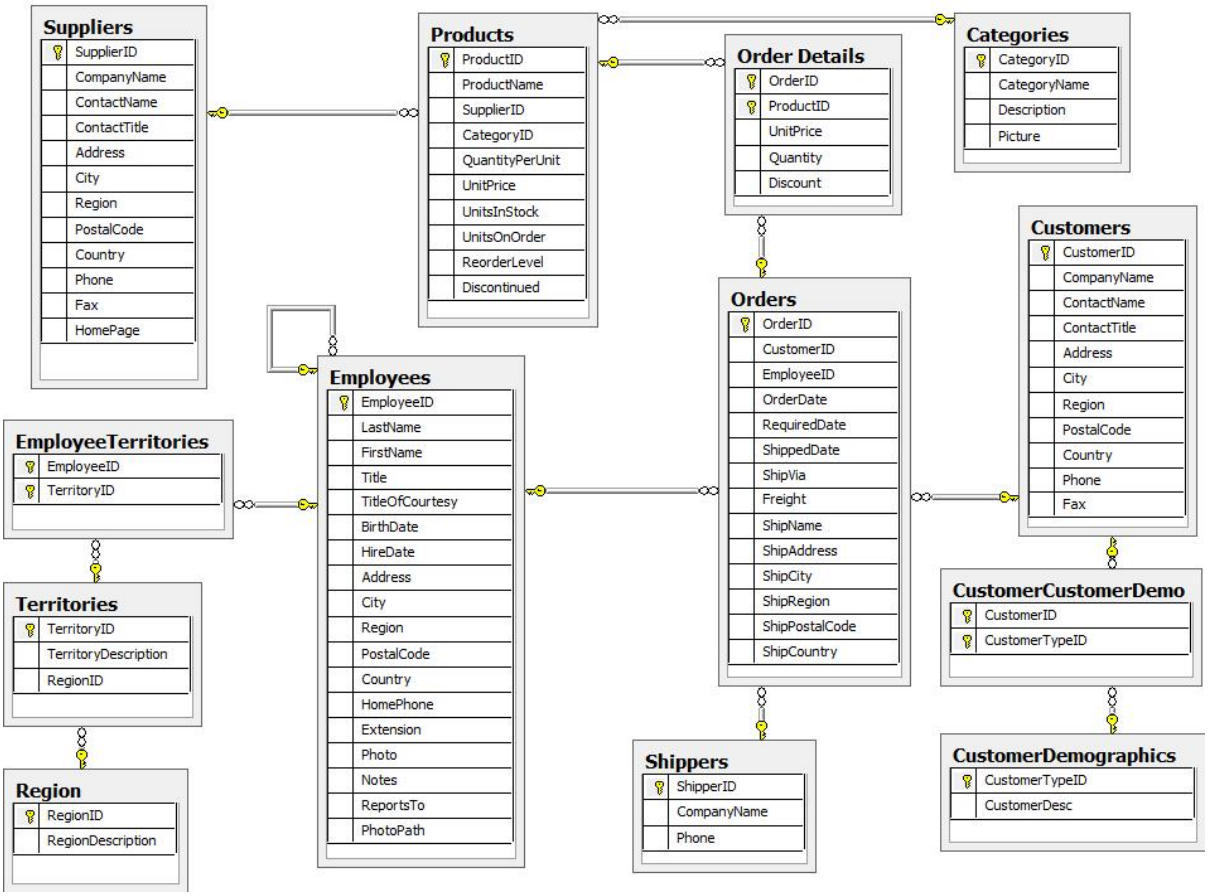
This guide uses a specific dataset, but its principles can be applied and reused with any data domain.

You should have a basic understanding of the property graph model and know how to model data as a graph.

About the data domain

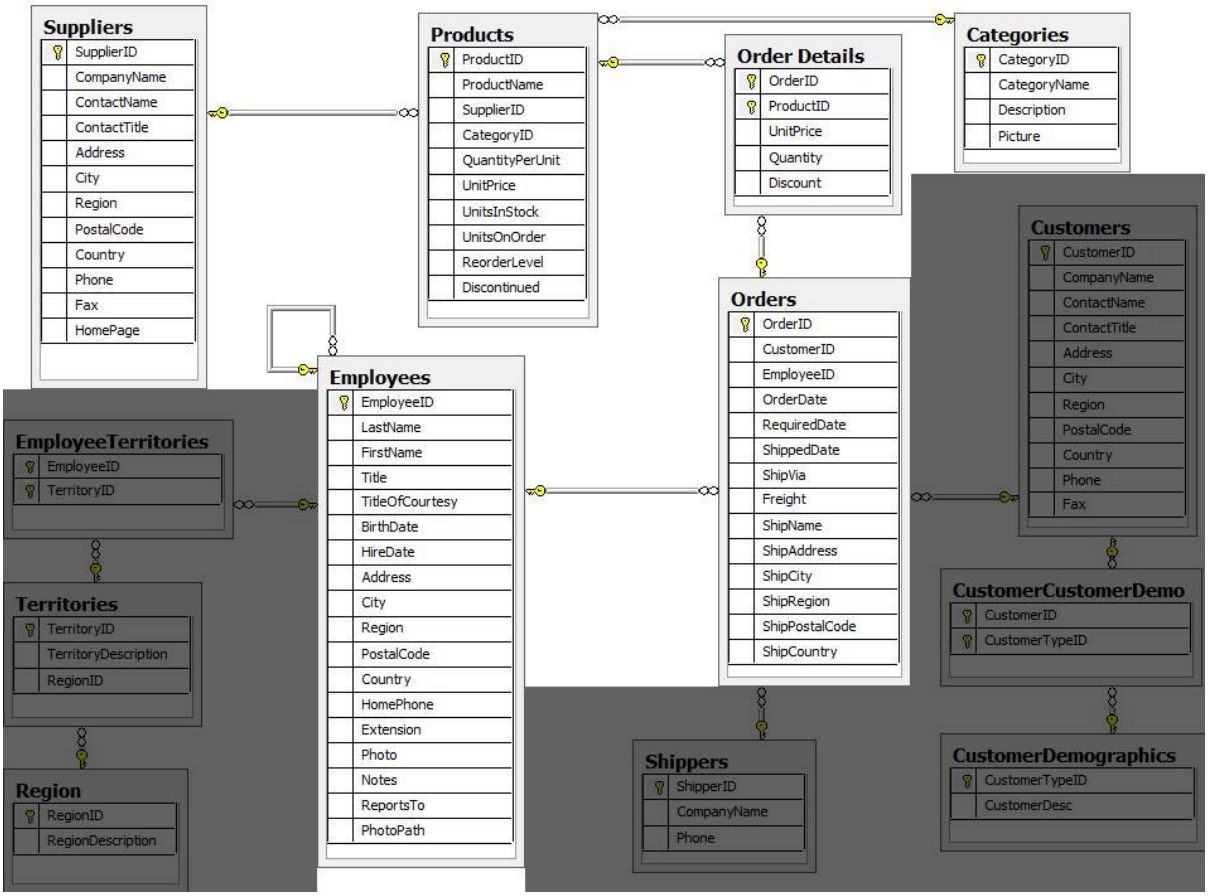
In this guide, we will be using the Northwind dataset, an often-used SQL dataset. This data depicts a product sale system - storing and tracking customers, products, customer orders, warehouse stock, shipping, suppliers, and even employees and their sales territories. Although the NorthWind dataset is 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.

> **Note:**
>
> 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

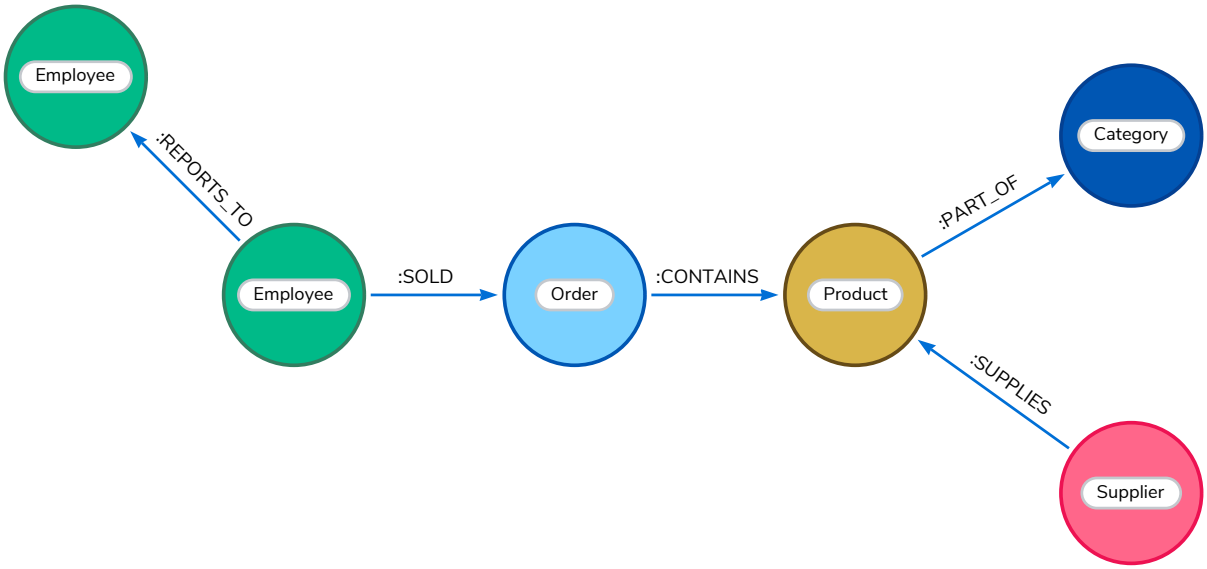
1. Each row on our `Orders` table becomes a node in our graph with `Order` as the label.
2. Each row on our `Products` table becomes a node with `Product` as the label.

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 the Neo4j instance you manage, you can copy the CSV files from Northwind files on GitHub and place them in the **import** folder for your Neo4j DBMS.

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 constraints 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/8bdd36dfc88381995e6823ff3f419b5a0cb8ac4f/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:

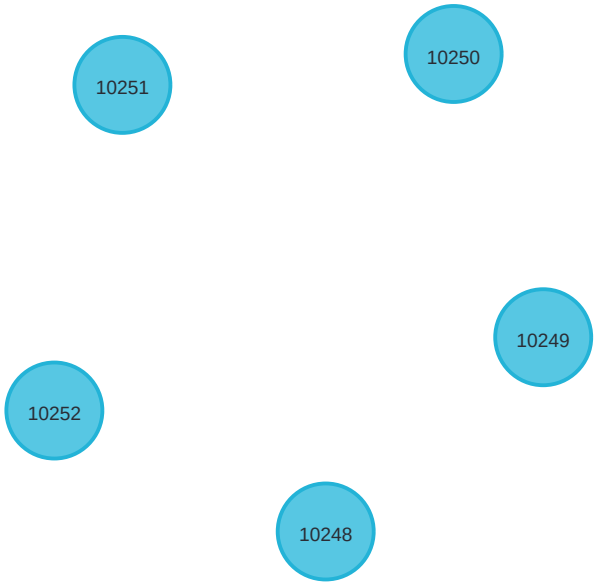
```
// Create orders
LOAD CSV WITH HEADERS FROM 'file:///orders.csv' AS row
MERGE (order:Order {orderID: row.OrderID})
  ON CREATE SET order.shipName = row.ShipName;
```

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}

o
{"shipName":Hanari Carnes,"orderID":10250}
{"shipName":Victuailles en stock,"orderID":10251}
{"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:

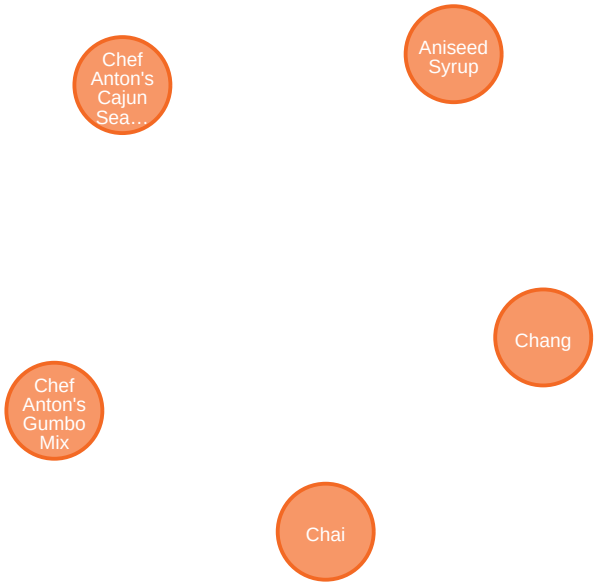
```
// Create products
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f419b5a0cb8ac4f/products.csv' AS row
MERGE (product:Product {productID: row.ProductID})
  ON CREATE SET product.productName = row.ProductName, product.unitPrice = 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}
{"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:

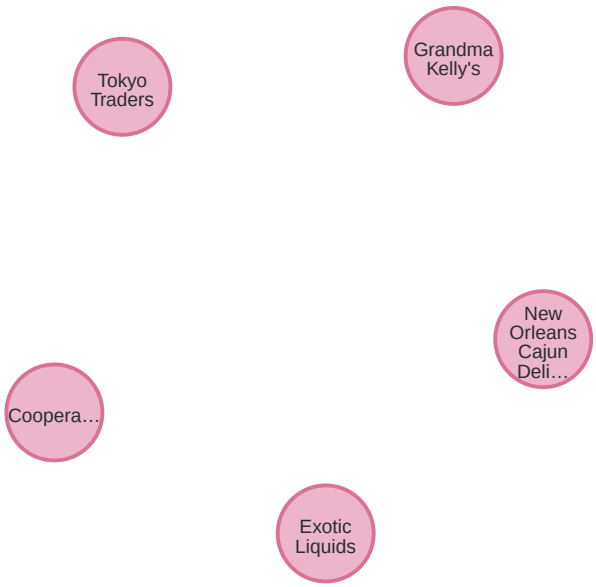
```
// Create suppliers
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f419b5a0cb8ac4f/suppliers.csv' AS row
MERGE (supplier:Supplier {supplierID: row.SupplierID})
  ON CREATE SET supplier.companyName = row.CompanyName;
```

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}

s
{"supplierID":2,"companyName":New Orleans Cajun Delights}
{"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:

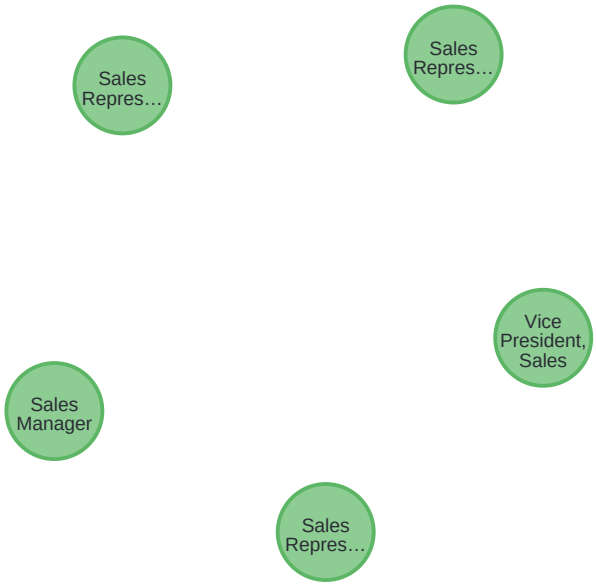
```
// Create employees
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f41
9b5a0cb8ac4f/employees.csv' AS row
MERGE (e:Employee {employeeID:row.EmployeeID})
  ON CREATE SET e.firstName = row.FirstName, e.lastName = row.LastName, 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}

e
{"lastName":Leverling,"firstName":Janet,"employeeID":3,"title":Sales Representative}
{"lastName":Peacock,"firstName":Margaret,"employeeID":4,"title":Sales Representative}
{"lastName":Buchanan,"firstName":Steven,"employeeID":5,"title":Sales Manager}

Creating **Category** nodes

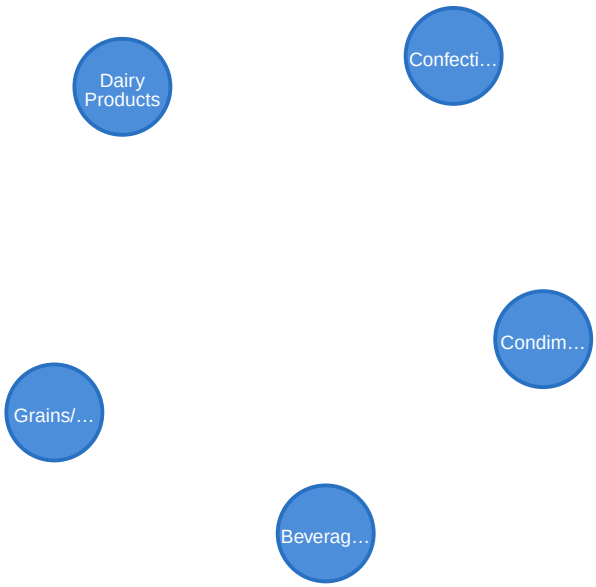
```
// Create categories
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f419b5a0cb8ac4f/categories.csv' AS row
MERGE (c:Category {categoryID: row.CategoryID})
  ON CREATE SET c.categoryName = row.CategoryName, c.description = 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}

c
{"description":Breads, crackers, pasta, and cereal,"categoryName":Grains/Cereals,"categoryID":5}

Note:

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 run the following Cypher command to view the indexes in the database:

```
SHOW INDEXES;
```

Two token lookup indexes (one for node labels and one for relationship types) are present by default when creating a Neo4j database. They exclusively solve node label and relationship type predicates and assist with the population of other indexes. Deleting them may have negative performance implications. You should see these indexes (and constraint) in the database:

```
+-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------+
|id|name          |state |populationPercent|type  |entityType  |labelsOrTypes|properties   |indexprovider
|owningConstraint|lastRead                    |readCount|
+-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------+
```

```
|7 |category_id   |ONLINE|100.0           |RANGE |NODE        |[Category]   |[categoryID] |range-1.0
|null            |null                          |0        |
+-------------------------------------------------------------------------------------------------------
-----------------------------------------------------------+
|6 |employee_id   |ONLINE|100.0           |RANGE |NODE        |[Employee]   |[employeeID] |range-1.0
|null            |null                          |0        |
+-------------------------------------------------------------------------------------------------------
-----------------------------------------------------------+
|1 |index_343aff4e|ONLINE|100.0           |LOOKUP|NODE        |null         |null         |token-lookup-
1.0|null            |2023-12-06T12:30:12.510000000Z|2286     |
+-------------------------------------------------------------------------------------------------------
-----------------------------------------------------------+
|2 |index_f7700477|ONLINE|100.0           |LOOKUP|RELATIONSHIP|null         |null         |token-lookup-
1.0|null            |null                          |0        |
+-------------------------------------------------------------------------------------------------------
-----------------------------------------------------------+
|8 |order_id      |ONLINE|100.0           |RANGE |NODE        |[Order]      |[orderID]    |range-1.0
|order_id        |2023-12-06T13:22:06.950000000Z|3815     |
+-------------------------------------------------------------------------------------------------------
-----------------------------------------------------------+
|3 |product_id    |ONLINE|100.0           |RANGE |NODE        |[Product]    |[productID]  |range-1.0
|null            |null                          |0        |
+-------------------------------------------------------------------------------------------------------
-----------------------------------------------------------+
|4 |product_name  |ONLINE|100.0           |RANGE |NODE        |[Product]    |[productName]|range-1.0
|null            |null                          |0        |
+-------------------------------------------------------------------------------------------------------
-----------------------------------------------------------+
|5 |supplier_id   |ONLINE|100.0           |RANGE |NODE        |[Supplier]   |[supplierID] |range-1.0
|null            |null                          |0        |
+-------------------------------------------------------------------------------------------------------
-----------------------------------------------------------+
```

For more information on indexes and their use in Neo4j, go to the Cypher Manual → The use of indexes.

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:

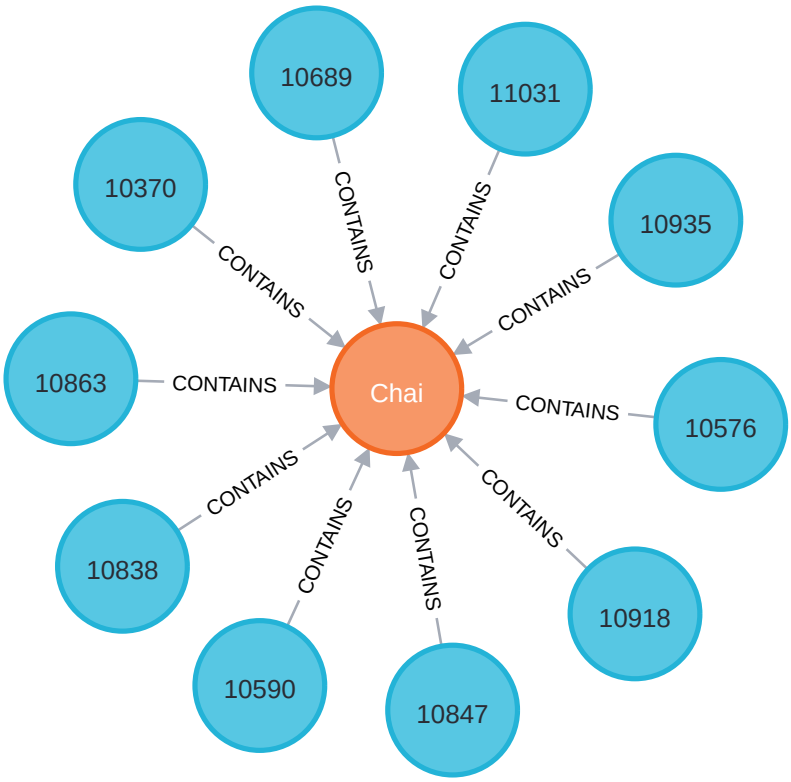
```
// Create relationships between orders and products
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f41
9b5a0cb8ac4f/orders.csv' AS row
MATCH (order:Order {orderID: row.OrderID})
MATCH (product:Product {productID: row.ProductID})
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:

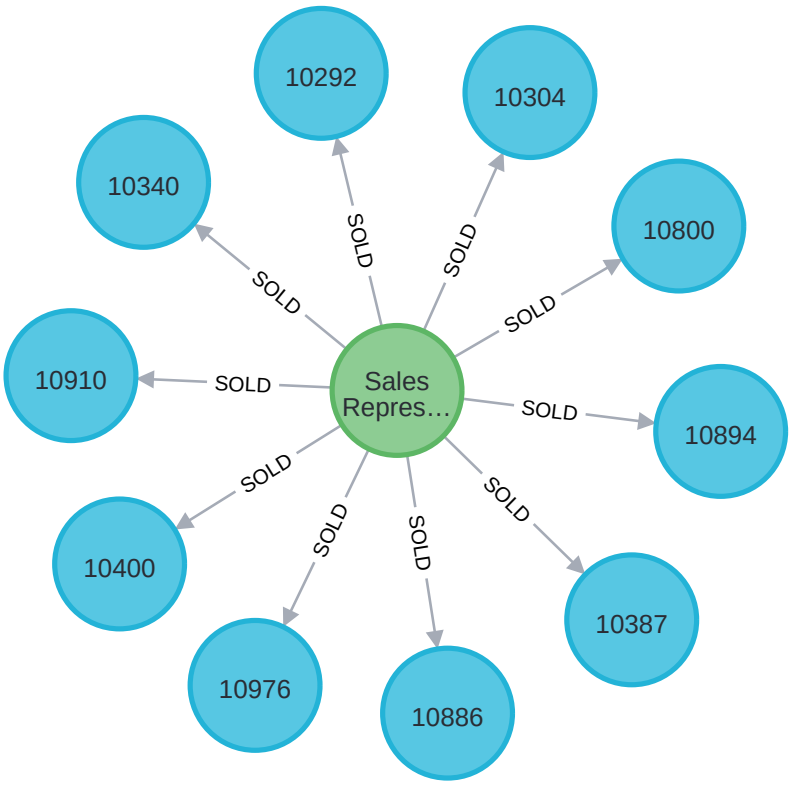
```
// Create relationships between orders and employees
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f41
9b5a0cb8ac4f/orders.csv' AS row
MATCH (order:Order {orderID: row.OrderID})
MATCH (employee:Employee {employeeID: 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:

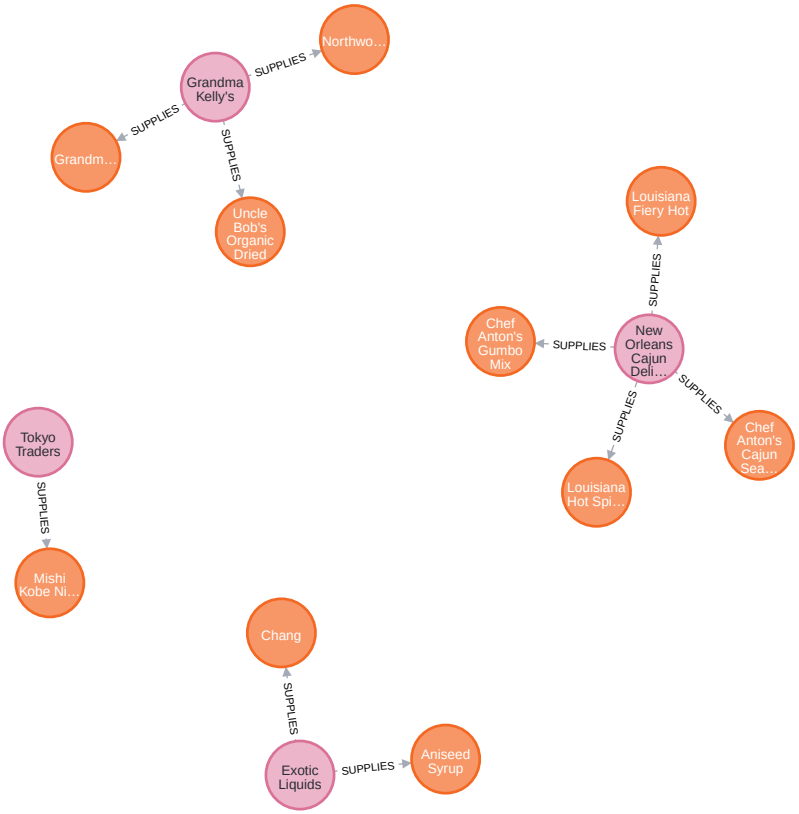
```
// Create relationships between products and suppliers
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f419b5a0cb8ac4f/products.csv
' AS row
MATCH (product:Product {productID: row.ProductID})
MATCH (supplier:Supplier {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:

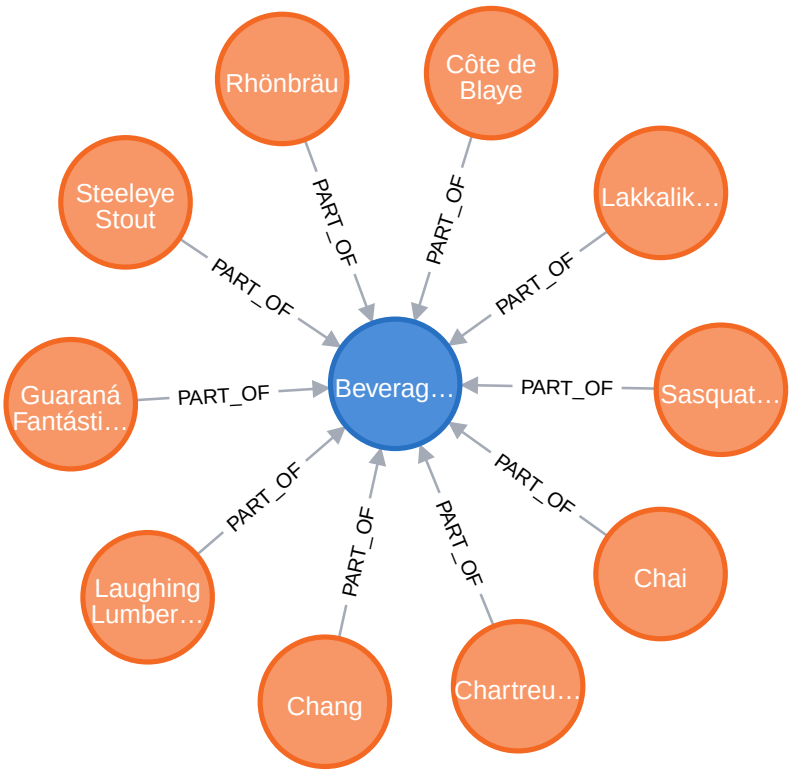
```
// Create relationships between products and categories
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f41
9b5a0cb8ac4f/products.csv
' AS row
MATCH (product:Product {productID: row.ProductID})
MATCH (category:Category {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:

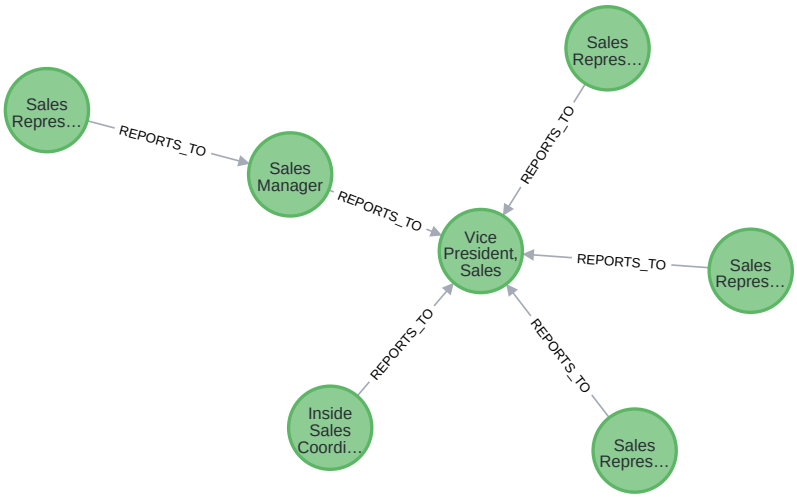
```
// Create relationships between employees (reporting hierarchy)
LOAD CSV WITH HEADERS FROM
'https://gist.githubusercontent.com/jexp/054bc6baf36604061bf407aa8cd08608/raw/8bdd36dfc88381995e6823ff3f41
9b5a0cb8ac4f/employees.csv' AS row
MATCH (employee:Employee {employeeID: row.EmployeeID})
MATCH (manager:Employee {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

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

License

Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)

You are free to

Share

copy and redistribute the material in any medium or format

Adapt

remix, transform, and build upon the material

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms

Attribution

You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

NonCommercial

You may not use the material for commercial purposes.

ShareAlike

If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions

You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

See https://creativecommons.org/licenses/by-nc-sa/4.0/ for further details. The full license text is available at https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode.