Getting Started Guide
# Table of Contents

- What is a graph database? ................................................................. 2
- Why graph databases? ................................................................. 2
- The property graph model ............................................................ 3
- What is Neo4j? ............................................................... 4
- Where and How is Neo4j Used? .................................................. 4

- Get started with Neo4j .............................................................. 6
  - Installing Neo4j ............................................................... 6
  - Documentation ............................................................... 6

- Graph database concepts ......................................................... 7
  - Example graph ............................................................... 7
  - Node ............................................................... 8
  - Node labels ............................................................... 9
  - Relationship ............................................................. 10
  - Relationship type ......................................................... 11
  - Properties ............................................................. 12
  - Traversals and paths ........................................................ 13
  - Schema ............................................................. 14
  - Indexes ............................................................. 14
  - Constraints ............................................................. 14
  - Naming conventions ....................................................... 14

- Concepts: Relational to Graph .................................................. 15
  - Translating Relational Knowledge to Graphs ...................... 16
  - Data Model Differences ................................................... 17
  - Data Storage and Retrieval ............................................. 18
  - Resources ............................................................. 19

- Introduction to Cypher .......................................................... 20
  - Patterns ............................................................. 20
  - Patterns in practice ...................................................... 23
  - Getting the correct results ............................................. 28
  - Composing large statements ........................................... 33
  - Defining a schema ...................................................... 34
  - Import data ............................................................. 37
Neo4j

Neo4j is the world’s leading graph database. The architecture is designed for optimal management, storage, and traversal of nodes and relationships. The graph database takes a property graph approach, which is beneficial for both traversal performance and operations runtime.

Cypher

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

Contents of this guide

The Neo4j Getting Started Guide covers the following areas:

- Get started with Neo4j — How to get started with Neo4j.
- Graph database concepts — Introduction to graph database concepts.
- Introduction to Cypher — Introduction to the graph query language Cypher.

Who should read this?

This guide is written for anyone who is exploring Neo4j and Cypher.
What is a graph database?

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

Why graph databases?

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

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

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

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

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

The property graph model

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

Building blocks of the property graph model

Nodes are the entities in the graph.

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

Relationships provide directed, named, connections between two node entities (e.g. Person LOVES Person).

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

If you’d like to learn more about any of these, you can read more about Graph Data Modeling.
What is Neo4j?

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

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

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

What makes Neo4j the easiest graph to work with?

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

Where and How is Neo4j Used?

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

- Recommendations
- Fraud Detection
- Dynamic Pricing
- IoT-applications

### Metadata and Advanced Analytics
*Generate Actionable Insights*

- Customer Engagement
- Data Lake Integration
- Risk Mitigation
- Knowledge Graphs for AI

### Internal Business Processes
*Improve Efficiency and Cut Costs*

- Identity and Access Management
- Network Management
- Supply Chain Efficiency
Get started with Neo4j

There are a number of options how to install Neo4j and how to get started running Cypher queries.

Installing Neo4j

The easiest way to set up an environment for developing an application with Neo4j and Cypher is to use Neo4j Desktop. Download Neo4j Desktop from https://neo4j.com/download/ and follow the installation instructions for your operating system.

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

Documentation

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

That is where you find the full manuals such as:

- **The Cypher manual** — This is the comprehensive manual for Cypher.
- **The Operations manual** — This manual describes how to deploy and maintain Neo4j.

The **Cypher Refcard** is a valuable asset when learning and writing Cypher.

Additionally, you can find more specialized documentation along with API documentation and documentation for older Neo4j releases.
Graph database concepts

Neo4j uses a property graph database model.

A graph data structure consists of nodes (discrete objects) that can be connected by relationships.

Example 1. Concept of a graph structure.

A graph with three nodes (the circles) and three relationships (the arrows).

The Neo4j property graph database model consists of:

- **Nodes** describe entities (discrete objects) of a domain.
- **Nodes** can have zero or more labels to define (classify) what kind of nodes they are.
- **Relationships** describes a connection between a source node and a target node.
- **Relationships** always has a direction (one direction).
- **Relationships** must have a type (one type) to define (classify) what type of relationship they are.
- **Nodes and relationships** can have properties (key-value pairs), which further describe them.

In mathematics, graph theory is the study of graphs.

In graph 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:
Example 2. Example graph.

Example 3. Cypher.

To create the example graph, use the Cypher clause `CREATE`.

```cypher
```

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.

The node labels are:

- Person
- Actor

The properties are:

- name: Tom Hanks
- born: 1956

The node can be created with Cypher using the query:

```
CREATE (:Person:Actor {name: 'Tom Hanks', born: 1956})
```

Node labels

Labels shape the domain by grouping (classifying) nodes into sets where all nodes with a certain label belong to the same set.

For example, all nodes representing users could be labeled with the label User. With that in place, you can ask Neo4j to perform operations only on your user nodes, such as finding all users with a given name.

Since labels can be added and removed during runtime, they can also be used to mark temporary states for nodes. A Suspended label could be used to denote bank accounts that are suspended, and a Seasonal label can denote vegetables that are currently in season.

A node can have zero to many labels.

In the example graph, the node labels, Person, Actor, and Movie, are used to describe (classify) the nodes. More labels can be added to express different dimensions of the data.

The following graph shows the use of multiple labels.
Example 5. Multiple labels.

<table>
<thead>
<tr>
<th>Person Actor</th>
<th>Movie</th>
<th>Person Director</th>
</tr>
</thead>
<tbody>
<tr>
<td>name = 'Tom Hanks'</td>
<td>title = 'Forrest Gump'</td>
<td>name = 'Robert Zemeckis'</td>
</tr>
<tr>
<td>born = 1956</td>
<td>released = 1994</td>
<td>born = 1951</td>
</tr>
</tbody>
</table>

Relationship

A relationship describes how a connection between a source node and a target node are related. It is possible for a node to have a relationship to itself.

A relationship:

- Connects a source node and a target node.
- Has a direction (one direction).
- Must have a type (one type) to define (classify) what type of relationship it is.
- Can have properties (key-value pairs), which further describe the relationship.

Relationships organize nodes into structures, allowing a graph to resemble a list, a tree, a map, or a compound entity — any of which may be combined into yet more complex, richly inter-connected structures.
Example 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}]->()
```

You must create or reference a source node and a target node to be able to create a relationship.

Relationships always have a direction. However, the direction can be disregarded where it is not useful. This means that there is no need to add duplicate relationships in the opposite direction unless it is needed to describe the data model properly.

A node can have relationships to itself. To express that **Tom Hanks** **KNOWS** himself would be expressed as:

Example 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.
Example 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 9. Number

```graphql
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 10. String and boolean

```graphql
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 11. Lists

```graphql
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].
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 12. Path matching.

To find out which movies Tom Hanks acted in according to the tiny example database, the traversal would start from the Tom Hanks node, follow any ACTED_IN relationships connected to the node, and end up with Forrest Gump as the result (see the dashed lines):

![Graph diagram showing Tom Hanks, Forrest Gump, and related relationships]

The traversal result could be returned as a path with the length 1:

![Path diagram showing Tom Hanks to Forrest Gump]

The shortest possible path has length zero. It contains a single node and no relationships.

Example 13. Path of length zero.

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

![Single node diagram]
Example 14. Path of length one.

A path containing one relationship has the length of 1.

![Diagram of a path with a single relationship]

**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>
<thead>
<tr>
<th>Graph entity</th>
<th>Recommended style</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node label</td>
<td>Camel case, beginning with an upper-case character</td>
<td>:VehicleOwner rather than :vehice_owner</td>
</tr>
<tr>
<td>Relationship type</td>
<td>Upper case, using underscore to separate words</td>
<td>:OWNS_VEHICLE rather than :ownsVehicle</td>
</tr>
<tr>
<td>Property</td>
<td>Lower camel case, beginning with a lower-case character</td>
<td>firstName rather than first_name</td>
</tr>
</tbody>
</table>

For the precise naming rules, refer to the Cypher manual → Naming rules and recommendations.
This chapter explores the concepts of graph databases from a relational developer’s point of view. It aims to explain the conceptual differences between relational and graph database structures and data models. It also gives a high-level overview of how working with each database type is similar or different - from the relational and graph query languages to interacting with the database from applications.

Relational databases have been the work horse of software applications since the 80’s, and continue as such to this day. They store highly-structured data in tables with predetermined columns of specific types and many rows of those defined types of information. Due to the rigidity of their organization, relational databases require developers and applications to strictly structure the data used in their applications.

In relational databases, references to other rows and tables are indicated by referring to primary key attributes via foreign key columns. Joins are computed at query time by matching primary and foreign keys of all rows in the connected tables. These operations are compute-heavy and memory-intensive and have an exponential cost.

When many-to-many relationships occur in the model, you must introduce a JOIN table (or associative entity table) that holds foreign keys of both the participating tables, further increasing join operation costs. The image below shows this concept of connecting a Person (from Person table) to a Department (in Department table) by creating a Person-Department join table that contains the ID of the person in one column and the ID of the associated department in the next column.

As you can probably see, this makes understanding the connections very cumbersome because you must know the person ID and department ID values (performing additional lookups to find them) in order to know which person connects to which departments. Those types of costly join operations are often addressed by denormalizing the data to reduce the number of joins necessary, therefore breaking the data integrity of a relational database.
Although not every use case is a good fit for this stringent data model, the lack of viable alternatives and the wide support for relational databases made it difficult for alternative models to break into the mainstream. However, the NoSQL era arrived in the market, filling some needs for users and businesses, but still missing the importance of the connections between data. This is how graph databases were born. They were designed to provide the greatest advantage in the connected world we live in today.

Translating Relational Knowledge to Graphs

Unlike other database management systems, relationships are of equal importance in the graph data model to the data itself. This means we are not required to infer connections between entities using special properties such as foreign keys or out-of-band processing like map-reduce.

By assembling nodes and relationships into connected structures, graph databases enable us to build simple and sophisticated models that map closely to our problem domain. The data stays remarkably similar to its form in the real world - small, normalized, yet richly connected entities. This allows you to query and view your data from any imaginable point of interest, supporting many different use cases.

Each node (entity or attribute) in the graph database model directly and physically contains a list of relationship records that represent the relationships to other nodes. These relationship records are organized by type and direction and may hold additional attributes. Whenever you run the equivalent of a JOIN operation, the graph database uses this list, directly accessing the connected nodes and eliminating the need for expensive search-and-match computations.

This ability to pre-materialize relationships into the database structure allows Neo4j to provide performance of several orders of magnitude above others, especially for join-heavy queries, allowing users to leverage a minutes to milliseconds advantage.
Data Model Differences

As you can probably imagine from the structural differences discussed above, the data models for relational versus graph are very different. The straightforward graph structure results in much simpler and more expressive data models than those produced using traditional relational or other NoSQL databases.

If you are used to modeling with relational databases, remember the ease and beauty of a well-designed, normalized entity-relationship diagram - a simple, easy-to-understand model you can quickly whiteboard with your colleagues and domain experts. A graph is exactly that - a clear model of the domain, focused on the use cases you want to efficiently support.

Let’s compare the two data models to show how the structure differs between relational and graph.

Figure 2. Relational - Person and Department tables (click to zoom)

In the above relational example, we search the Person table on the left (potentially millions of rows) to find the user Alice and her person ID of 815. Then, we search the Person-Department table (orange middle table) to locate all the rows that reference Alice’s person ID (815). Once we retrieve the 3 relevant rows, we go to the Department table on the right to search for the actual values of the department IDs (111, 119, 181). Now we know that Alice is part of the 4Future, P0815, and A42 departments.

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

In the above graph version, we have a single node for Alice with a label of Person. Alice belongs to 3 different departments, so we create a node for each one and with a label of Department. To find out which departments Alice belongs to, we would search the graph for Alice’s node, then traverse all of the BELONGS_TO relationships from Alice to find the Department nodes she is connected to. That’s all we
need - a single hop with no lookups involved.

More information on this topic can be found in Graph Data Modeling.

Data Storage and Retrieval

Querying relational databases is easy with SQL - a declarative query language that allows both easy ad-hoc querying in a database tool, as well as use-case-specific querying from application code. Even object-relational mappers (ORMs) use SQL under the hood to talk to the database.

Do graph databases have something similar? Yes!

Cypher,Neo4j’s declarative graph query language, is built on the basic concepts and clauses of SQL but has a lot of additional graph-specific functionality to make it easy to work with your graph model.

If you have ever tried to write a SQL statement with a large number of joins, you know that you quickly lose sight of what the query actually does because of all the technical noise in SQL syntax. In Cypher, the syntax remains concise and focused on domain components and the connections among them, expressing the pattern to find or create data more visually and clearly. Other clauses outside of the basic pattern matching look very similar to SQL, as Cypher was built on the predecessor language’s foundations.

We will cover Cypher query language syntax in an upcoming guide, but let us look at a brief example of how a SQL query differs from a Cypher query. In the organizational domain from our data modeling example above, what would a SQL statement that lists the employees in the IT Department look like, and how does it compare to the Cypher statement?

SQL Statement

```sql
SELECT name FROM Person
LEFT JOIN Person_Department
  ON Person.Id = Person_Department.PersonId
LEFT JOIN Department
  ON Department.Id = Person_Department.DepartmentId
WHERE Department.name = "IT Department"
```

Cypher Statement

```cypher
MATCH (p:Person)-[:WORKS_AT]->(d:Dept)
WHERE d.name = "IT Department"
RETURN p.name
```

You can find more about Cypher syntax in the upcoming chapters for Cypher Query Language and transitioning from SQL to Cypher.

Transitioning from Relational to Graph - In Practice

If you do decide to move your data from a relational to a graph database, the steps to transition your applications to use Neo4j are actually quite simple. You can connect to Neo4j with a driver or connector library designed for your stack or programming language, just as you can with other databases. Thanks to Neo4j and its community, there are Neo4j drivers that mimic existing database driver idioms and approaches for nearly any popular programming language.
For instance, the Neo4j JDBC driver would be used like this to query the database for John’s departments:

```java
Connection con = DriverManager.getConnection("jdbc:neo4j://localhost:7474/");

String query =
    "MATCH (:Person {name:{1}})-[\:EMPLOYEE]-(:Department) RETURN d.name as dept";

try (PreparedStatement stmt = con.prepareStatement(query)) {
    stmt.setString(1,"John");
    ResultSet rs = stmt.executeQuery();
    while (rs.next()) {
        String department = rs.getString("dept");
        ....
    }
}
```

For more information, you can visit our pages for Building Applications to see how to connect to Neo4j using different programming languages.

### Resources

- Free eBook: Relational to Graph
- DZone Refcard: From Relational to Graph
- Data Modeling: Relational to Graph
Introduction to Cypher

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

For the full reference of Cypher, see the Cypher manual.

- Patterns
  - Node syntax
  - Relationship syntax
  - Pattern syntax
  - Pattern variables
  - Clauses
- Patterns in practice
  - Creating data
  - Matching patterns
  - Attaching structures
  - Completing patterns
- Getting the correct results
  - Filtering results
  - Returning results
  - Aggregating information
  - Ordering and pagination
  - Collecting aggregation
- Composing large statements
  - UNION
  - WITH
- Defining a schema
  - Using indexes
  - Using constraints
- Import data

Patterns

Neo4j’s Property Graphs are composed of nodes and relationships, either of which may have properties. Nodes represent entities, for example concepts, events, places and things. Relationships connect pairs of nodes.
However, nodes and relationships can be considered as low-level building blocks. The real strength of the property graph lies in its ability to encode patterns of connected nodes and relationships. A single node or relationship typically encodes very little information, but a pattern of nodes and relationships can encode arbitrarily complex ideas.

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

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

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

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

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

### Node syntax

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

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

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

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

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

### Relationship syntax

Cypher uses a pair of dashes (--) to represent an undirected relationship. Directed relationships have an arrowhead at one end (<-- , -->). Bracketed expressions ([...]) can be used to add details. This may include variables, properties, and type information:
The syntax and semantics found within a relationship’s bracket pair are very similar to those used between a node’s parentheses. A variable (e.g., `role`) can be defined, to be used elsewhere in the statement. The relationship’s type (e.g., `:ACTED_IN`) is analogous to the node’s label. The properties (e.g., `roles`) are entirely equivalent to node properties.

**Pattern syntax**

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

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

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

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

**Pattern variables**

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

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

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

**Clauses**

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

- create and match patterns in the graph
- filter, project, sort, or paginate results
- compose partial statements

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

Creating data

We'll start by looking into the clauses that allow us to create data.

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

Obviously the simplest clause is called `CREATE`. It will just go ahead and directly create the patterns that you specify.

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

```
```

If we execute this statement, Cypher returns the number of changes, in this case adding 1 node, 1 label and 2 properties.

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

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

![Movie]

```
title = 'The Matrix'
released = 1997
```

If case we also want to return the created data we can add a `RETURN` clause, which refers to the variable we have assigned to our pattern elements.

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

This is what gets returned:

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

[![Created Nodes: 1 Added Labels: 1 Set Properties: 2 Rows: 1](image)]

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

If we want to create more than one element, we can separate the elements with commas or use multiple `CREATE` statements.
We can of course also create more complex structures, like an `ACTED_IN` relationship with information about the character, or `DIRECTED` ones for the director.

```
CREATE (d:Person {name: 'Robert Zemeckis', born: 1951})-[r:DIRECTED]->(m)
RETURN a, d, r, m
```

This is the part of the graph we just updated:

![Graph visualization]

In most cases, we want to connect new data to existing structures. This requires that we know how to find existing patterns in our graph data, which we will look at next.

### Matching patterns

Matching patterns is a task for the `MATCH` statement. We pass the same kind of patterns we’ve used so far to `MATCH` to describe what we’re looking for. It is similar to query by example, only that our examples also include the structures.

A `MATCH` statement will search for the patterns we specify and return one row per successful pattern match.

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

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

Here’s the result:

<table>
<thead>
<tr>
<th>Movie</th>
<th>Movie</th>
</tr>
</thead>
<tbody>
<tr>
<td>title = 'The Matrix'</td>
<td>title = 'Forrest Gump'</td>
</tr>
<tr>
<td>released = 1997</td>
<td>released = 1994</td>
</tr>
</tbody>
</table>

This should show both The Matrix and Forrest Gump.

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

```
MATCH (p:Person {name: 'Keanu Reeves'})
RETURN p
```
This query returns the matching node:

<table>
<thead>
<tr>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>name = 'Keanu Reeves'</td>
</tr>
<tr>
<td>born = 1964</td>
</tr>
</tbody>
</table>

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

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

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

Rows: 1

<table>
<thead>
<tr>
<th>m.title</th>
<th>r.roles</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Forrest Gump'</td>
<td>['Forrest']</td>
</tr>
</tbody>
</table>

In this case we only returned the properties of the nodes and relationships that we were interested in. You can access them everywhere via a dot notation `identifier.property`.

Of course this only lists his role as Forrest in Forrest Gump because that's all data that we have added.

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

### Attaching structures

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

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

Here's what the structure looks like in the database:
It is important to remember that we can assign variables to both nodes and relationships and use them later on, no matter if they were created or matched.

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

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

Completing patterns

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

If we would not know if our graph already contained Cloud Atlas we could merge it in again.

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

We get a result in any both cases: either the data (potentially more than one row) that was already in the graph or a single, newly created Movie node.
A **MERGE** clause without any previously assigned variables in it either matches the full pattern or creates the full pattern. It never produces a partial mix of matching and creating within a pattern. To achieve a partial match/create, make sure to use already defined variables for the parts that shouldn’t be affected.

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

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

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

In case the direction of a relationship is arbitrary, you can leave off the arrowhead. **MERGE** will then check for the relationship in either direction, and create a new directed relationship if no matching relationship was found.

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

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

This is the graph structure that gets created:
Here there is no global search for the two Month nodes; they are only searched for in the context of the 2014 Year node.

Getting the correct results

Example graph

First we create some data to use for our examples:

```javascript
cREATE (forrestGump:Movie {title: 'Forrest Gump', released: 1994})
cREATE (keanu:Person {name: 'Keanu Reeves', born: 1964})
cREATE (robert:Person {name: 'Robert Zemeckis', born: 1951})
cREATE (tom:Person {name: 'Tom Hanks', born: 1956})
cREATE (tom)-[:ACTED_IN {roles: ['Forrest']}]->(forrestGump)
cREATE (tom)-[:ACTED_IN {roles: ['Zachry']}]->(cloudAtlas)
cREATE (robert)-[:DIRECTED]->(forrestGump)
```

This is the resulting graph:

Filtering results

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

```javascript
MATCH (m:Movie)
WHERE m.title = 'The Matrix'
RETURN m
```
The query above, using the **WHERE** clause, is equivalent to this query which includes the condition in the pattern matching:

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

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

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

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

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

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

Here we find actors, because they sport an **ACTED_IN** relationship but then skip those that ever **DIRECTED**
any movie.

There are more advanced ways of filtering, for example list predicates, which we will discuss later in this section.

Returning results

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>p</th>
<th>name</th>
<th>toUpper(p.name)</th>
<th>nickname</th>
</tr>
</thead>
<tbody>
<tr>
<td>(:Person {name: 'Keanu Reeves', born: 1964})</td>
<td>'Keanu Reeves'</td>
<td>'KEANU REEVES'</td>
<td>'n/a'</td>
</tr>
<tr>
<td>(:Person {name: 'Robert Zemeckis', born: 1951})</td>
<td>'Robert Zemeckis'</td>
<td>'ROBERT ZEMECKIS'</td>
<td>'n/a'</td>
</tr>
<tr>
<td>(:Person {name: 'Tom Hanks', born: 1956})</td>
<td>'Tom Hanks'</td>
<td>'TOM HANKS'</td>
<td>'n/a'</td>
</tr>
</tbody>
</table>

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

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

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

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

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

Rows: 1

<table>
<thead>
<tr>
<th>people</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

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

Aggregation works implicitly in Cypher. We specify which result columns we wish to aggregate. Cypher will use all non-aggregated columns as grouping keys.

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

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

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

Rows: 1

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

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

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

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

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

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

For instance to find the most prolific actors we could do:

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

Collecting aggregation

A very helpful aggregation function is `collect()`, which collects all the aggregated values into a list. This is very useful in many situations, since no information of details is lost while aggregating.

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

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

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

Rows: 2

<table>
<thead>
<tr>
<th>movie</th>
<th>cast</th>
<th>actors</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Forrest Gump'</td>
<td>['Tom Hanks']</td>
<td>1</td>
</tr>
<tr>
<td>'Cloud Atlas'</td>
<td>['Tom Hanks']</td>
<td>1</td>
</tr>
</tbody>
</table>
The lists created by `collect()` can either be used from the client consuming the Cypher results, or directly within a statement with any of the list functions or predicates.

### Composing large statements

#### Example graph

We continue using the same example data as before:

```cypher
CREATE (forrestGump:Movie {title: 'Forrest Gump', released: 1994})
CREATE (keanu:Person {name: 'Keanu Reeves', born: 1964})
CREATE (robert:Person {name: 'Robert Zemeckis', born: 1951})
CREATE (tom:Person {name: 'Tom Hanks', born: 1956})
CREATE (tom)-[r:ACTED_IN {roles: ['Forrest', 'Zachry']}]-(forrestGump)
CREATE (robert)-[r:DIRECTED]->(forrestGump)
```

This is the resulting graph:

![Graph diagram]

**UNION**

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

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

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

<table>
<thead>
<tr>
<th></th>
<th>name</th>
<th>type</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>'Tom Hanks'</td>
<td>'ACTED_IN'</td>
<td>'Cloud Atlas'</td>
</tr>
<tr>
<td></td>
<td>'Tom Hanks'</td>
<td>'ACTED_IN'</td>
<td>'Forrest Gump'</td>
</tr>
<tr>
<td></td>
<td>'Robert Zemeckis'</td>
<td>'DIRECTED'</td>
<td>'Forrest Gump'</td>
</tr>
</tbody>
</table>
Note that the returned columns must be aliased in the same way in all the sub-clauses.

The query above is equivalent to this more compact query:

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

WITH

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

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

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

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

Rows: 1

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

Defining a schema

Example graph

First create some data to use for our examples:

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

This is the resulting graph:
Using indexes

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

Indexes can be added at any time.

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

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

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

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

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

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

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

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

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

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

```cypher
SHOW INDEXES YIELD name, labelsOrTypes, properties, type
```
Using constraints

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

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

Example 15. Uniqueness constraint

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

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

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

CREATE CONSTRAINT constraint_example_1 ON (movie:Movie) ASSERT movie.title IS UNIQUE

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

You can query a database to find out what constraints are defined with the SHOW CONSTRAINTS Cypher syntax.
Example 16. Constraints query

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

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

Rows: 1

```
+----------------------------------------------------------------------------------------------------+
<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>type</th>
<th>entityType</th>
<th>labelsOrTypes</th>
<th>properties</th>
<th>ownedIndexId</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>'constraint_example_1'</td>
<td>'UNIQUENESS'</td>
<td>'NODE'</td>
<td>['Movie']</td>
<td>['title']</td>
<td>3</td>
</tr>
</tbody>
</table>
+----------------------------------------------------------------------------------------------------+
```

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

Learn more about constraints in Cypher manual → Constraints.

Import data

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

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

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

The data files

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

- `persons.csv`
- `movies.csv`
- `roles.csv`

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

The content of the movies.csv file:

movies.csv

<table>
<thead>
<tr>
<th>id, title, country, year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, Wall Street, USA, 1987</td>
</tr>
<tr>
<td>2, The American President, USA, 1995</td>
</tr>
<tr>
<td>3, The Shawshank Redemption, USA, 1994</td>
</tr>
</tbody>
</table>

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

The content of the roles.csv file:

roles.csv

<table>
<thead>
<tr>
<th>personId, movieId, role</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 1, Bud Fox</td>
</tr>
<tr>
<td>4, 1, Carl Fox</td>
</tr>
<tr>
<td>3, 1, Gordon Gekko</td>
</tr>
<tr>
<td>4, 2, A.J. MacInerney</td>
</tr>
<tr>
<td>3, 2, President Andrew Shepherd</td>
</tr>
<tr>
<td>5, 3, Ellis Boyd 'Red' Redding</td>
</tr>
</tbody>
</table>

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

The graph model

The following simple data model shows what a graph model for this data set could look like:
This is the resulting graph, based on the data from the CSV files:

Prerequisites

This tutorial uses the Linux or macOS tarball installation.

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

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

Prepare the database

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

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

Creating a unique constraint also implicitly creates an index. By indexing the id property, node lookup (e.g. by MATCH) will be much faster.
Additionally, it is a good idea to index the country name for a fast lookup.

1. Start neo4j.

Run the command:

```
bin/neo4j start
```

The default user name is neo4j and password neo4j.

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

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

Using Neo4j Browser, run the following Cypher:

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

Or using Neo4j Cypher Shell, run the command:

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

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

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

Using Neo4j Browser, run the following Cypher:

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

Or using Neo4j Cypher Shell, run the command:

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

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

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

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

Using Neo4j Browser, run the following Cypher:
CREATE INDEX FOR (c:Country) ON (c.name)

Or using Neo4j Cypher Shell, run the command:

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

**Import data using LOAD CSV**

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

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

Using Neo4j Browser, run the following Cypher:

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

Or using Neo4j Cypher Shell, run the command:

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

Output:

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

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

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

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

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

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

Using Neo4j Browser, run the following Cypher:

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

Or using Neo4j Cypher Shell, run the command:
### 3. Load the data from the roles.csv file

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

- For larger data files, it is useful to use the hint `USING PERIODIC COMMIT` clause of `LOAD CSV`. This hint tells Neo4j that the query might build up inordinate amounts of transaction state, and thus needs to be periodically committed. For more information, see null.

Using Neo4j Browser, run the following Cypher:

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

Or using Neo4j Cypher Shell, run the command:

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

Output:

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

### Validate the imported data

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

Using Neo4j Browser, run the following Cypher:

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

Or using Neo4j Cypher Shell, run the command:

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

Output:

```
42
```
\[\begin{aligned}
\text{n} & \quad \text{r} \\
\text{m} & \\
\end{aligned}\]

\[
\begin{aligned}
\text{(:Movie \{id: 3, title: "The Shawshank Redemption", year: 1994\})} & \quad \text{[::ORIGIN]} \\
\text{(:Country \{name: "USA"\})} & \\
\text{(:Movie \{id: 2, title: "The American President", year: 1995\})} & \quad \text{[::ORIGIN]} \\
\text{(:Country \{name: "USA"\})} & \\
\text{(:Movie \{id: 1, title: "Wall Street", year: 1987\})} & \quad \text{[::ORIGIN]} \\
\text{(:Country \{name: "USA"\})} & \\
\text{(:Person \{name: "Morgan Freeman", id: 4\})} & \quad \text{[::ACTED_IN \{role: "Carl Fox"\}]} \\
\text{(:Movie \{id: 1, title: "Wall Street", year: 1987\})} & \\
\text{(:Person \{name: "Charlie Sheen", id: 1\})} & \quad \text{[::ACTED_IN \{role: "Bud Fox"\}]} \\
\text{(:Movie \{id: 1, title: "Wall Street", year: 1987\})} & \\
\text{(:Person \{name: "Martin Sheen", id: 3\})} & \quad \text{[::ACTED_IN \{role: "Gordon Gekko"\}]} \\
\text{(:Movie \{id: 1, title: "Wall Street", year: 1987\})} & \\
\text{(:Person \{name: "Martin Sheen", id: 3\})} & \quad \text{[::ACTED_IN \{role: "President Andrew Shepherd"\}]} \\
\text{(:Movie \{id: 2, title: "The American President", year: 1995\})} & \\
\text{(:Person \{name: "Morgan Freeman", id: 4\})} & \quad \text{[::ACTED_IN \{role: "A.J. MacInerney"\}]} \\
\text{(:Movie \{id: 2, title: "The American President", year: 1995\})} & \\
\end{aligned}\]
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.