The Neo4j Getting Started Guide v4.0

Table of Contents

Get started with Neo4j 2
- Installing Neo4j 2
- Documentation guide 2

Graph database concepts 3
- Example graph 3
- Nodes 3
- Labels 3
- Relationships 4
- Relationship types 4
- Properties 5
- Traversals and paths 5
- Schema 5
- Naming rules and recommendations 6

Introduction to Cypher 7
- Patterns 7
- Patterns in practice 10
- Getting the correct results 14
- Composing large statements 19
- Defining a schema 21
- Import data 23

This is the Getting Started Guide for Neo4j version 4.0, authored by the Neo4j Team.

This 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 anybody who is exploring Neo4j and Cypher.

Get started with Neo4j

This chapter gives an orientation on how to get started with Neo4j.

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.

Documentation guide

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 Driver manual — This manual describes the officially supported drivers for Neo4j.
- 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

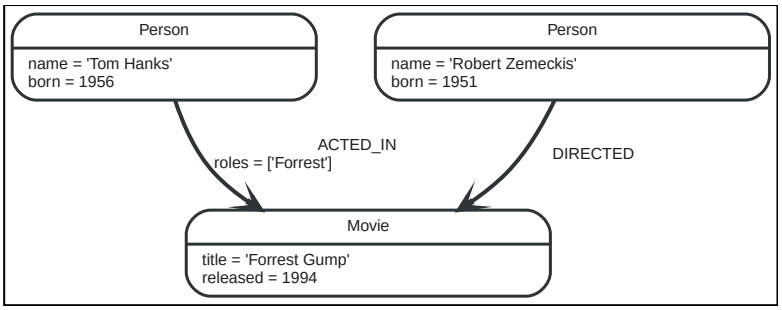
This chapter presents an introduction to graph database concepts.

This chapter includes the following sections:

- Example graph
- Nodes
- Labels
- Relationships
- Relationship types
- Properties
- Traversals and paths
- Schema
- Naming rules and recommendations

Example graph

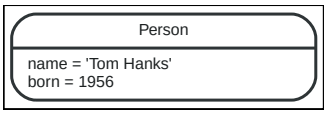
We will use the example graph below to introduce the basic concepts of the property graph:



Nodes

Nodes are often used to represent *entities*. The simplest possible graph is a single node.

Consider the graph below, consisting of a single node.



Labels

Labels are used to shape the domain by grouping nodes into sets where all nodes that have a certain label belongs to the same set.

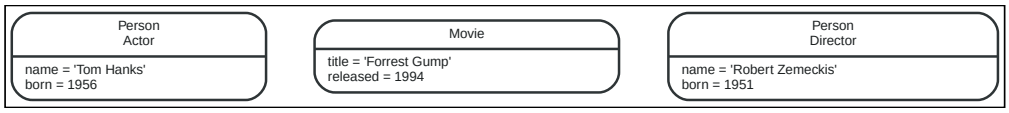
For example, all nodes representing users could be labeled with the label `:User`. With that in place, you can ask Neo4j to perform operations only on your user nodes, such as finding all users with a given name.

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 above, the nodes have the labels `Person` and `Movie`, which is one possible way of describing the data. But assume that we want to express different dimensions of the data. One way of doing that is to add more labels.

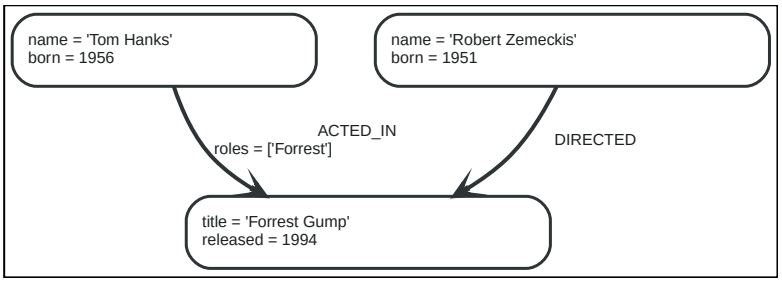
Below is an example showing the use of multiple labels:



Relationships

A relationship connects two nodes. 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.

Our example graph will make a lot more sense once we add relationships to it:

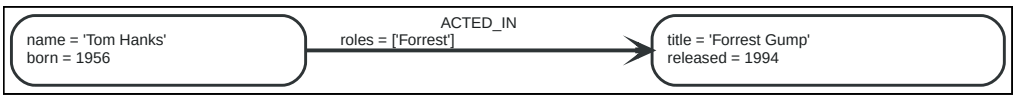


Relationship types

A relationship must have exactly one relationship type.

Our example uses `ACTED_IN` and `DIRECTED` as relationship types. The `roles` property on the `ACTED_IN` relationship has an array value with a single item in it.

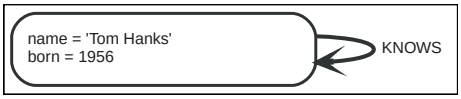
Below is an `ACTED_IN` relationship, with the `Tom Hanks` node as the *source node* and `Forrest Gump` as the *target node*.



We observe that the `Tom Hanks` node has an *outgoing* relationship, while the `Forrest Gump` node has an *incoming* relationship.

Relationships always have a direction. However, you only have to pay attention to the direction where it is useful. This means that there is no need to add duplicate relationships in the opposite direction unless it is needed in order to properly describe your use case.

Note that a node can have relationships to itself. If we want to express that `Tom Hanks` `KNOWS` himself, that would be expressed as:



Properties

Properties are name-value pairs that are used to add qualities to nodes and relationships.

In our example graphs, we have used the properties `name` and `born` on `Person` nodes, `title` and `released` on `Movie` nodes, and the property `roles` on the `:ACTED_IN` relationship.

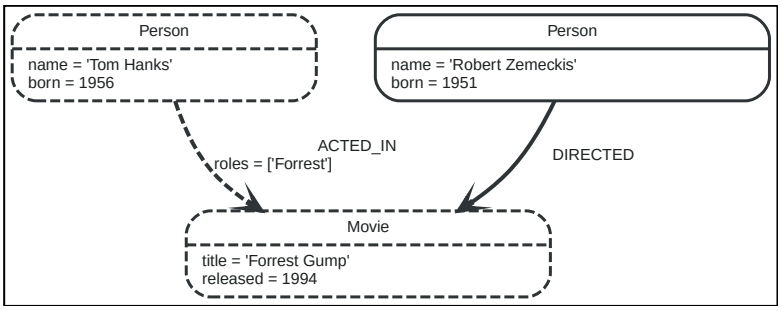
The value part of the property can hold different data types such as `number`, `string` and `boolean`. For a thorough description of the available data types, refer to the Cypher manual.

Traversals and paths

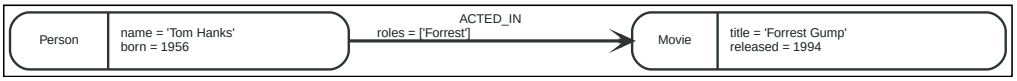
A traversal is how you query a graph in order to find answers to questions, for example: "What music do my friends like that I don't yet own?", or "What web services are affected if this power supply goes down?".

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

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

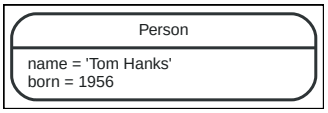


The traversal result could be returned as a path with the length one:

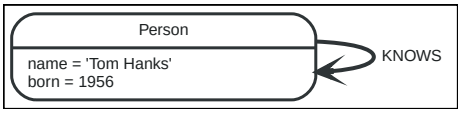


The path above has length one.

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



This path has 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 indexes, see Using constraints. For detailed descriptions of how to work with constraints in Cypher, see the Cypher manual Constraints.

Naming rules and recommendations

Node labels, relationship types and properties are case sensitive, meaning for example that the property `name` means something different than the property `Name`. It is recommended to follow the naming conventions described in the following table:

Table 1. Naming conventions

Graph entity	Recommended style	Example
Node label	Camel case, beginning with an upper-case character	`:VehicleOwner` rather than `:vehice_owner`
Relationship type	Upper case, using underscore to separate words	`:OWNS_VEHICLE` rather than `:ownsVehicle`
Property	Lower camel case, beginning with a lower-case character	`firstName` rather than `first_name`

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

Introduction to Cypher

This chapter gives a high-level overview of the graph query language Cypher.

This chapter 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

This chapter includes:

- 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

For a comprehensive guide to Cypher, see the Cypher manual.

Patterns

This section gives an introduction to the concept of patterns.

This section includes:

- Node syntax

- Relationship syntax
- Pattern syntax
- Pattern variables
- Clauses

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

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

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

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

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

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

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

Node syntax

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

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

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

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

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

Relationship syntax

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

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

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

This section describes how patterns are used in practice

This section includes:

- Creating data
- Matching patterns
- Attaching structures
- Completing patterns

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.

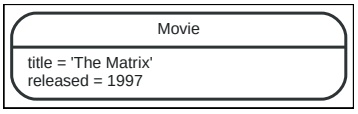
For the patterns we've looked at so far this could look like the following:

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

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

```
+-------------------+
| No data returned. |
+-------------------+
Nodes created: 1
Properties set: 2
Labels added: 1
```

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



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

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

This is what gets returned:

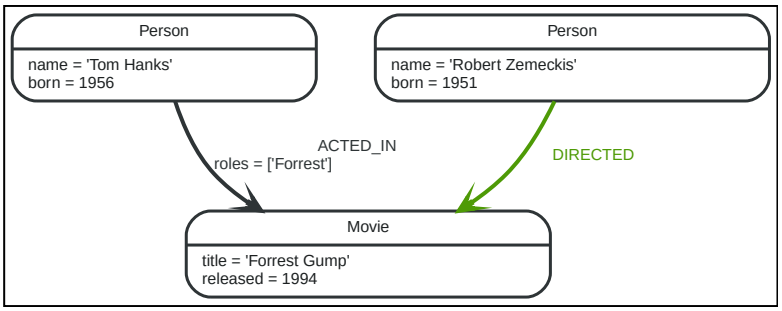
```
+--------------------------------------+
| p                                    |
+--------------------------------------+
| Node[1]{name:"Keanu Reeves",born:1964} |
+--------------------------------------+
1 row
Nodes created: 1
Properties set: 2
Labels added: 1
```

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

This is the part of the graph we just updated:



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

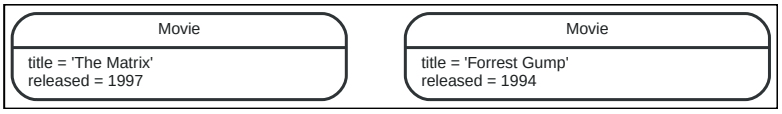
Matching patterns is a task for the `MATCH` statement. We pass the same kind of patterns we've used so far to `MATCH` to describe what we're looking for. It is similar to *query by example*, only that our examples also include the structures.

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

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

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

Here's the result:

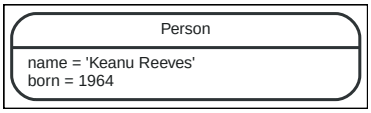


This should show both *The Matrix* and *Forrest Gump*.

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:



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
```

```
+-----------------------------+
| m.title        | r.roles    |
+-----------------------------+
| "Forrest Gump" | ["Forrest"] |
+-----------------------------+
1 row
```

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 `identifer.property`.

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

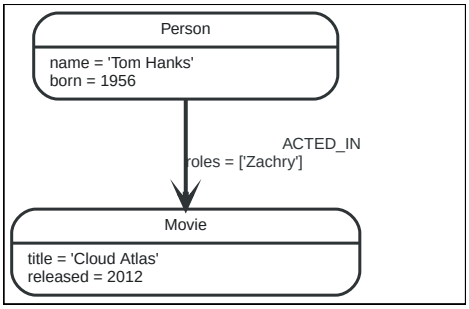
Now we know enough to connect new nodes to existing ones and can combine `MATCH` and `CREATE` to attach structures to the graph.

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 wouldn't know if our graph already contained *Cloud Atlas* we could merge it in again.

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

```
+-------------------------------------------+
| m                                         |
+-------------------------------------------+
| Node[5]{title:"Cloud Atlas",released:2012} |
+-------------------------------------------+
1 row
```

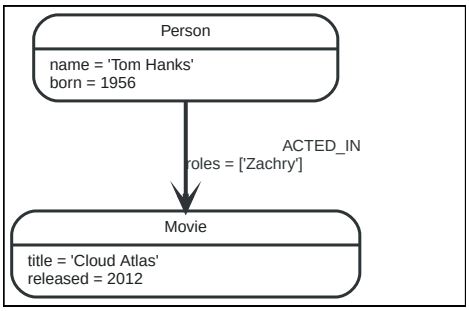
We get a result in any both cases: either the data (potentially more than one row) that was already in the graph or a single, newly created `Movie` node.

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'll 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
```

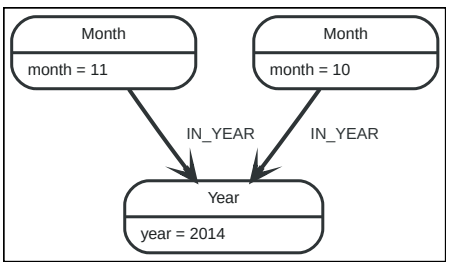


In case the direction of a relationship is arbitrary, you can leave off the arrowhead. MERGE will then check for the relationship in either direction, and create a new directed relationship if no matching relationship was found.

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

This section describes how to manipulate the output of Cypher queries in order to get the results you are looking for.

This section includes:

- Example graph
- Filtering results

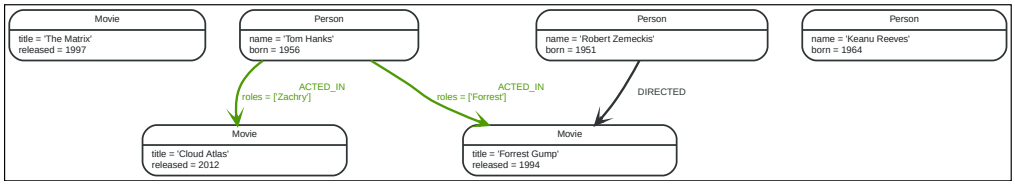
- Returning results
- Aggregating information
- Ordering and pagination
- Collecting aggregation

Example graph

First we create some data to use for our examples:

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

This is the resulting graph:



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.

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

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

1 row
```

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

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

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

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

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

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

1 row
```

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.

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

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

2 rows
```

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
```

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

3 rows
```

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

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

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

2 rows
```

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
```

```
+--------+
| people |
+--------+
| 3      |
+--------+
1 row
```

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
```

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

1 row
```

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;
```

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

1 row
```

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
```

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

2 rows
```

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

This section describes how to compose large statements using the `UNION` and `WITH` keywords.

This section includes:

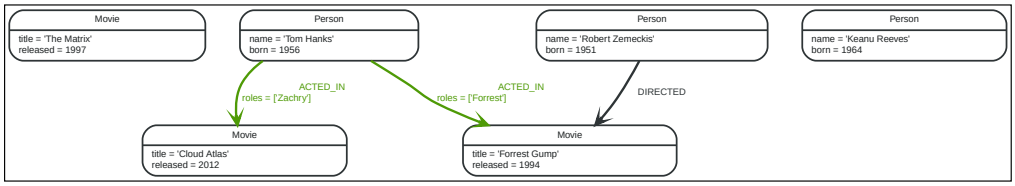
- Example graph
- UNION
- WITH

Example graph

We continue using the same example data as before:

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

This is the resulting graph:



UNION

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

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

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

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

3 rows
```

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
```

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

1 row
```

Defining a schema

This section describes how to define and use indexes and constraints.

This section includes:

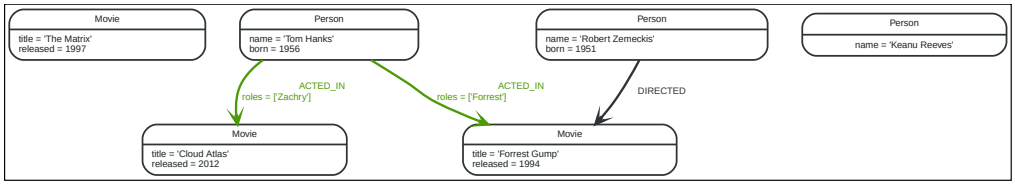
- Example graph
- Defining and using indexes
- Defining and using constraints

Example graph

First we create some data to use for our examples:

```
CREATE (matrix:Movie { title:"The Matrix",released:1997 })
CREATE (cloudAtlas:Movie { title:"Cloud Atlas",released:2012 })
CREATE (forrestGump:Movie { title:"Forrest Gump",released:1994 })
CREATE (keanu:Person { name:"Keanu Reeves"})
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:



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

In this case we want to create an index to speed up finding actors by name in the database:

```
CREATE INDEX ON :Actor(name)
```

In most cases it is not necessary to specify indexes when querying for data, as the appropriate indexes will be used automatically. For example, the following query will automatically use the index defined above:

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

A *composite index* is an index on multiple properties for all nodes that have a particular label. For example, the following statement will create a composite index on all nodes labeled with `Actor` and which have both a `name` and a `born` property. Note that since the node with the `Actor` label that has a

`name` of "Keanu Reeves" does not have the `born` property. Therefore that node will not be added to the index.

```
CREATE INDEX ON :Actor(name, born)
```

We can inspect our database to find out what indexes are defined. We do this by calling the built-in procedure `db.indexes`:

```
CALL db.indexes
YIELD description, tokenNames, properties, type;
```

```
+-----------------------------------------------------------------------------------------+
| description                   | tokenNames | properties       | type                   |
+-----------------------------------------------------------------------------------------+
| "INDEX ON :Actor(name)"       | ["Actor"]  | ["name"]         | "node_label_property" |
| "INDEX ON :Actor(name, born)" | ["Actor"]  | ["name", "born"] | "node_label_property" |
+-----------------------------------------------------------------------------------------+

2 rows
```

Learn more about indexes in Cypher Manual Indexes.

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.

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".

To create a constraint that makes sure that our database will never contain more than one node with the label `Movie` and the property `title`, we use the IS UNIQUE syntax:

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

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.

We can inspect our database to find out what constraints are defined. We do this by calling the built-in procedure `db.constraints`:

```
CALL db.constraints
```

```
+-------------------------------------------------------------+
| description                                                 |
+-------------------------------------------------------------+
| "CONSTRAINT ON ( movie:Movie ) ASSERT movie.title IS UNIQUE" |
+-------------------------------------------------------------+

1 row
```

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 will demonstrate how to import data from CSV files using `LOAD CSV`.

For a full description of `LOAD CSV` , see Cypher Manual `LOAD CSV`.

With `LOAD CSV` we can conveniently import data into Neo4j and have access to Cypher to perform actions on the data as desired.

In this example, we are given the following CSV files:

- *persons.csv*, a list of persons:

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

- *movies.csv*, a list of movies:

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

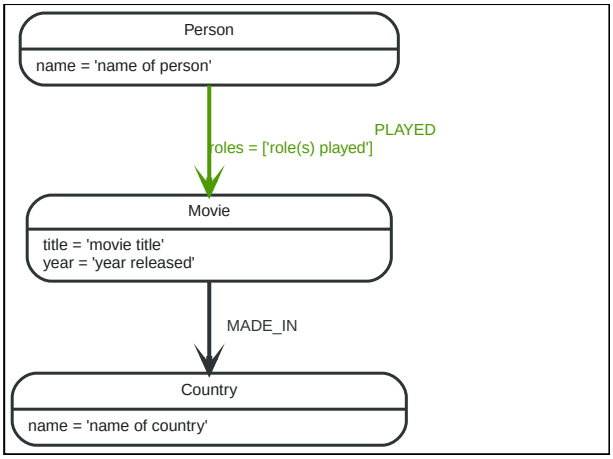
- *roles.csv*, a list of which role was played by some of these persons in each movie:

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

By inspecting the files we can see that:

- Each person has a unique id and a name.
- Each movie has a unique id, a title, a country where it was made, and a year when it was released.
- Using the *roles* file we can deduct which person has acted in which movie, and what role(s) they played.

We can come up with the following simple data model:



Before starting our imports, we will prepare our database by creating indexes and constraints. Since we expect the `id` property on `Person` and `Movie` to be unique in each set, we will create a unique constraint. This protects us from invalid data since constraint creation will fail if there are multiple nodes with the same id property.

Creating a unique constraint also implicitly creates a unique index. The `id` property is a temporary property used to look up the appropriate nodes for a relationship when importing the third file. By indexing the `id` property, node lookup (e.g. by `MATCH`) will be much faster.

```
CREATE CONSTRAINT ON (person:Person) ASSERT person.id IS UNIQUE
```

```
CREATE CONSTRAINT ON (movie:Movie) ASSERT movie.id IS UNIQUE
```

Additionally, we create an index on the `name` property on `Country` nodes to ensure fast lookups:

```
CREATE INDEX ON :Country(name)
```

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

In this example, the CSV files are stored in the default import directory on the database server, and we can access them using a `file:///` URL. Other locations are configurable, and additionally, `LOAD CSV` supports accessing CSV files via `HTTPS`, `HTTP`, and `FTP`. For complete instructions, see Cypher Manual → `LOAD CSV`.

Using the following Cypher queries, we will create a node for each person, a node for each movie and a relationship between the two with a property denoting the role. We are also keeping track of the country in which each movie was made.

Let's start with importing the *persons.csv* file. Here is the Cypher used to do the import:

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

Now, let's import the movies. This time, we are also creating a relationship to the country in which the movie was made. We are using `MERGE` to create nodes that represent countries. Using `MERGE` avoids creating duplicate country nodes in the case where multiple movies have been made in the same country.

```
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)-[:MADE_IN]->(country)
```

Finally, we will create relationships between the persons and the movies; one actor can participate in many movies, and one movie has many actors in it. Now importing the relationships is a matter of finding the nodes and then creating relationships between them.

```
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)-[:PLAYED {role: csvLine.role}]->(movie)
```

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 so needs to be periodically committed. For more information, see cypher-manual.pdf.

Finally, since the `id` property was only necessary to import the relationships, we can drop the constraints and the `id` property from all movie and person nodes.

```
DROP CONSTRAINT ON (person:Person) ASSERT person.id IS UNIQUE
```

```
DROP CONSTRAINT ON (movie:Movie) ASSERT movie.id IS UNIQUE
```

```
MATCH (n)
WHERE n:Person OR n:Person
REMOVE n.id
```

This is the resulting graph: