neo4j

The #1 Platform for Connected Data

[:Get Started with]->

Cypher

Beginner's Guide

Get Started with Cypher

Note: This guide is an introductory overview of the Cypher graph query language. For the most up-to-date documentation on Cypher, please see the Neo4j Cypher Manual.

You'll probably want to follow this guide with a Neo4j instance in front of you. The best way to set up a Neo4j environment is to use Neo4j Desktop.

Download Neo4j Desktop and follow the installation instructions for your operating system. Neo4j Desktop manages installation of the Neo4j Database and provides access to many useful development tools.

Alternatively, you can use Neo4j Aura in the cloud, fully automated, managed and integrated with the Google Cloud Platform.

You can also follow this guide using the Neo4j Sandbox. Once logged in you can easily spin up a blank sandbox, and walk through the examples in this guide.

Welcome to your journey with Cypher. Please enjoy this guide.

—Michael Hunger

Chapter 4. Get started with Cypher

This guide will introduce you to Cypher, Neo4j's query language. It will help you:

- start thinking about graphs and patterns
- apply this knowledge to simple problems
- learn how to write Cypher statements

4.1. Patterns

- 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 are simply 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 (mildly) complex pattern which performs this match:

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

Pattern recognition is fundamental to the way that the brain works. Consequently, humans are very good at working with patterns. When patterns are presented visually, for example in a diagram or map, humans can use them to recognize, specify, and understand concepts. As a pattern-based language, Cypher takes advantage of this capability.

Like SQL, used in relational databases, Cypher is a textual declarative query language. It uses a form of ASCII art *(https://en.wikipedia.org/wiki/ASCII_art)* to represent graph-related patterns. SQL-like clauses and keywords, for example `MATCH`, `WHERE` and `DELETE` are used to combine these patterns and specify desired actions.

This combination tells Neo4j which patterns to match and what to do with the matching items, for example nodes, relationships, paths and lists. However, Cypher does *not* tell Neo4j *how* to find nodes, traverse relationships etc.

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

4.1.1. Node syntax

Cypher uses a pair of parentheses (usually containing a text string) to represent a node, eg: `()`, `(foo)`. This is reminiscent of a circle or a rectangle with rounded end caps. Here are some ASCII-art encodings for example Neo4j 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` label (prefixed in use with a colon) declares the node's type. This restricts the pattern, keeping it from matching (say) a structure with an `Actor` node in this position. Neo4j's node indexes also use labels: each index is specific to the combination of a label and a property.

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.

4.1.2. 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/or 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. (Note that the value of a property may be an array.)

4.1.3. 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"} )
```

Like with node labels, the relationship type `ACTED_IN` is added as a symbol, prefixed with a colon: `:ACTED_IN`. Variables (eg, `role`) can be used elsewhere in the statement to refer to the relationship. Node and relationship properties use the same notation. In this case, we used an array property for the `roles`, allowing multiple roles to be specified.

Pattern Nodes vs. Database Nodes

When a node is used in a pattern, it *describes* zero or more nodes in the database. Similarly, each pattern describes zero or more paths of nodes and relationships.

4.1.4. 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, including `nodes(path)`, `relationships(path)`, and `length(path)`.

4.1.5. 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. Neo4j then figures out how to achieve the desired goal in an efficient manner.

4.2. Patterns in practice

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

4.2.1. 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.

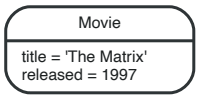
For the patterns we've looked at so far this could look like the following:

```
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:

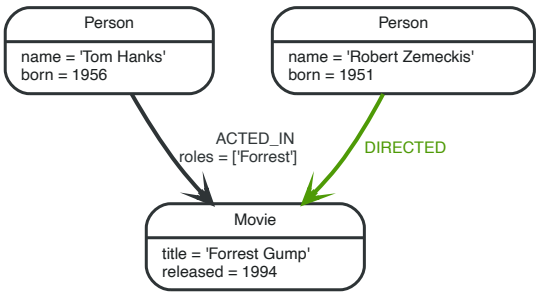
```
+--------------------------------------+
| p                                    |
+--------------------------------------+
| Node[1]{name:"Keanu Reeves",born:1964} |
+--------------------------------------+
1 row
Nodes created: 1
Properties set: 2
Labels added: 1
```

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.

4.2.2. 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.

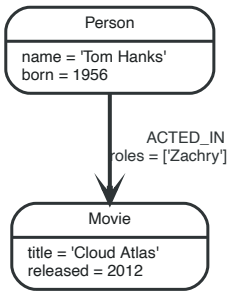
Now we know enough to connect new nodes to existing ones and can combine `MATCH` and `CREATE` to attach structures to the graph.

4.2.3. 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`.

4.2.4. 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
```

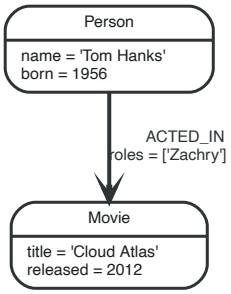
We get a result in any both cases: either the data (potentially more than one row) that was already in the graph or a single, newly created `Movie` node.

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
```

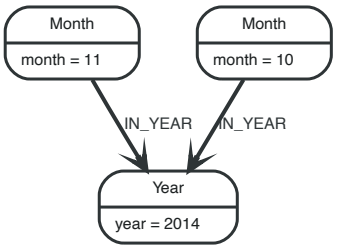


In case the direction of a relationship is arbitrary, you can leave off the arrowhead. `MERGE` will then check for the relationship in either direction, and create a new directed relationship if no matching relationship was found.

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.

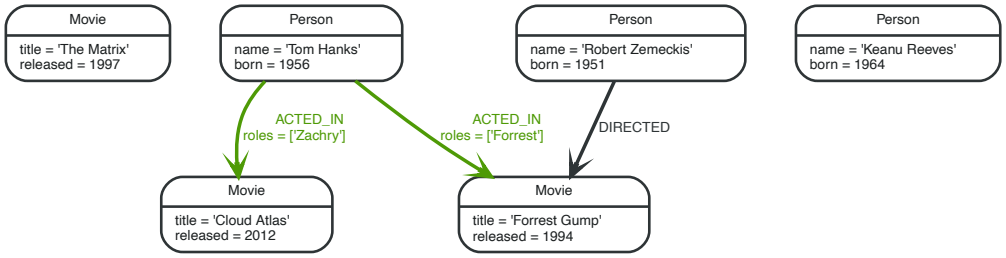
4.3. Getting correct results

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

Let's first get some data in to retrieve results from:

```
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 data we will start out with:



4.3.1. Filtering results

So far we've matched patterns in the graph and always returned all results we found. Quite often there are conditions in play for what we want to see. Similar to in *SQL* those filter conditions are expressed in a `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                                         |
+-------------------------------------------+
| Node[0]{title:"The Matrix",released:1997} |
+-------------------------------------------+
1 row
```

For equality on one or more properties, a more compact syntax can be used as well:

```
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 below 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                                                             |
+-----------------------------------------------------------------------------------------------------------------------------------+
|Node[5]{name:"Tom Hanks",born:1956} | :ACTED_IN[1]{roles:["Zachry"]} | Node[1]{title:"Cloud Atlas",released:2012}   |
+-----------------------------------------------------------------------------------------------------------------------------------+
1 row
```

One aspect that might be a little surprising is that you can even use patterns 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 additional patterns as well (or `NOT`).

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

```
+-----------------------------------------------------------------------------------+
| p                                  | m                                            |
+-----------------------------------------------------------------------------------+
| Node[5]{name:"Tom Hanks",born:1956} | Node[1]{title:"Cloud Atlas",released:2012}  |
| Node[5]{name:"Tom Hanks",born:1956} | Node[2]{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 also more advanced ways of filtering like list-predicates which we will look at later on.

4.3.2. Returning results

So far we've returned only nodes, relationships, or paths directly via their variables. But the `RETURN` clause can actually return any number of expressions. But what are actually expressions in Cypher?

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

Predicates that you'd use in `WHERE` count as boolean expressions.

Of course simpler 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`. You can later on refer to that column using its alias.

```
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                                                    |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------+
| Node[3]{name:"Keanu Reeves",born:1964}    | "Keanu Reeves"    | "KEANU REEVES"    | "n/a"    | {name -> "Keanu Reeves", label -> "Person"}    |
| Node[4]{name:"Robert Zemeckis",born:1951} | "Robert Zemeckis" | "ROBERT ZEMECKIS" | "n/a"    | {name -> "Robert Zemeckis", label -> "Person"} |
| Node[5]{name:"Tom Hanks",born:1956}       | "Tom Hanks"       | "TOM HANKS"       | "n/a"    | {name -> "Tom Hanks", label -> "Person"}       |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------+
3 rows
```

If you're interested in unique results you can use the `DISTINCT` keyword after `RETURN` to indicate that.

4.3.3. Aggregating information

In many cases you want to aggregate or group the data that you encounter while traversing patterns in your graph. In Cypher aggregation happens in the `RETURN` clause while computing your 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
```

Please note that `NULL` values are skipped during aggregation. For aggregating only unique values use `DISTINCT`, like in `count(DISTINCT role)`.

Aggregation in Cypher just works. You specify which result columns you want to aggregate and *Cypher will use all non-aggregated columns as grouping keys*.

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

To find out how often an actor and director worked together, you'd run this statement:

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

```
+-------------------------------------------------------------------------------------------------+
| actor                               | director                                  | collaborations |
+-------------------------------------------------------------------------------------------------+
| Node[5]{name:"Tom Hanks",born:1956} | Node[4]{name:"Robert Zemeckis",born:1951} | 1              |
+-------------------------------------------------------------------------------------------------+
1 row
```

Frequently you want to sort and paginate after aggregating a count(x)

4.3.4. Ordering and pagination

Ordering works like in other query languages, with an `ORDER BY expression [ASC|DESC]` clause. The expression can be any expression discussed before as long as it is computable from the returned information.

So for instance if you return `person.name` you can still `ORDER BY person.age` as both are accessible from the `person` reference. You cannot order by things that you can't infer from the information you return. This is especially important with aggregation and `DISTINCT` return values as both remove the visibility of data that is aggregated.

Pagination is a straightforward use of `SKIP &offset LIMIT &count`.

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

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

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

```
+-------------------------------------------------+
| a                                 | appearances |
+-------------------------------------------------+
| Node[5]{name:"Tom Hanks",born:1956} | 2           |
+-------------------------------------------------+
1 row
```

4.3.5. Collecting aggregation

The most helpful aggregation function is `collect()`, which, appropriately collects all aggregated values into a list. This comes very handy in many situations as 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 per each child-row, or even running `n+1` statements to retrieve the parent and its children individually.

To retrieve the cast of each movie in our database this statement could be used:

```
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.

4.4. Composing large statements

- UNION
- WITH

Let's first get some data in to retrieve results from:

```
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)
```

4.4.1. UNION

A Cypher statement is usually quite compact. Expressing references between nodes as visual patterns makes them easy to understand.

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

For instance if you want to list both actors and directors without using the alternative relationship-type syntax `()-[:ACTED_IN|:DIRECTED]->()` you can do this:

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

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

4.4.2. WITH

In Cypher it's possible to chain fragments of statements together, much like you would do within a data-flow pipeline. Each fragment works on the output from the previous one and its results can feed into the next one.

You use the `WITH` clause to combine the individual parts and declare which data flows from one to the other. `WITH` is very much like `RETURN` with the difference that it doesn't finish a query but prepares the input for the next part. You can use the same expressions, aggregations, ordering and pagination as in the `RETURN` clause.

The only difference is that you *must* alias all columns as they would otherwise not be accessible. Only columns that you declare in your `WITH` clause is available in subsequent query parts.

See below for an example where 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
```

If you want to filter by an aggregated value in SQL or similar languages you would have to use `HAVING`. That's a single purpose clause for filtering aggregated information. In Cypher, `WHERE` can be used in both cases.

4.5. Constraints and indexes

Labels are a convenient way to group nodes together. They are used to restrict queries, define constraints and create indexes.

4.5.1. Using constraints

You can specify unique constraints that guarantee uniqueness of a certain property on nodes with a specific label. These constraints are also used by the `MERGE` clause to make certain that a node only exists once.

The following is an example of how to use labels and add constraints and indexes to them. Let's start out by adding a constraint. In this case we decide that every `Movie` node should have a unique `title`.

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

Note that adding the unique constraint will implicitly add an index on that property, so we won't have to do that separately. If we drop a constraint but still want an index on that property, we will have to create the index explicitly.

Constraints can be added after a label is already in use, but that requires that the existing data complies with the constraints.

4.5.2. Using indexes

The main reason for using indexes in a graph database is to find the starting point in the graph as fast as possible. After that seek you rely on in-graph structures and the first class citizenship of relationships in the graph database to achieve high performance. Thus graph queries themselves do not need indexes to run fast.

Indexes can be added at any time. Note that it will take some time for an index to come online when there's existing data.

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

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

Now, let's add some data.

```
CREATE (actor:Actor { name:"Tom Hanks" }),(movie:Movie { title:'Sleepless IN Seattle' }),
  (actor)-[:ACTED_IN]->(movie);
```

Normally you don't specify indexes when querying for data. They will be used automatically. This means we that can simply look up the Tom Hanks node, and the index will kick in behind the scenes to boost performance.

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

4.6. Importing CSV files with Cypher

This tutorial will show you how to import data from CSV files using `LOAD CSV`.

In this example, we're given three CSV files: a list of persons, a list of movies, and a list of which role was played by some of these persons in each movie.

CSV files can be stored on the database server and are then accessible using a `file://` URL. Alternatively, `LOAD CSV` also supports accessing CSV files via `HTTPS`, `HTTP`, and `FTP`. `LOAD CSV` will follow `HTTP` redirects but for security reasons it will not follow redirects that changes the protocol, for example if the redirect is going from `HTTPS` to `HTTP`.

For more details, see LOAD CSV.

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

Let's start with importing the persons:

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

The CSV file we're using looks like this:

persons.csv

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

Now, let's import the movies. This time, we're also creating a relationship to the country in which the movie was made. If you are storing your data in a SQL database, this is the one-to-many relationship type.

We're 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.

When using `MERGE` or `MATCH` with `LOAD CSV` we need to make sure we have an index (see Indexes) or a unique constraint (see Constraints) on the property we're merging. This will ensure the query executes in a performant way.

Before running our query to connect movies and countries we'll create an index for the name property on the `Country` label to ensure the query runs as fast as it can:

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

```
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)
```

movies.csv

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

Lastly, we create the relationships between the persons and the movies. Since the relationship is a many to many relationship, one actor can participate in many movies, and one movie has many actors in it. We have this data in a separate file.

We'll index the `id` property on `Person` and `Movie` nodes. 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. Since we expect the ids to be unique in each set, we'll 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 creates a unique index (which is faster than a regular index).

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

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

Now importing the relationships is a matter of finding the nodes and then creating relationships between them.

For this query we'll use `USING PERIODIC COMMIT` (see `PERIODIC COMMIT` query hint) which is helpful for queries that operate on large CSV files. This hint tells Neo4j that the query might build up inordinate amounts of transaction state, and so needs to be periodically committed. In this case we also set the limit to `500` rows per commit.

```
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)
```

roles.csv

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

Finally, as 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:Movie
REMOVE n.id
```

Want to dive deeper into Cypher?

- Download the complete *The Neo4j Cypher Manual* for detailed guide.
- Check out *The Neo4j Cypher Refcard* for easy Cypher code references.