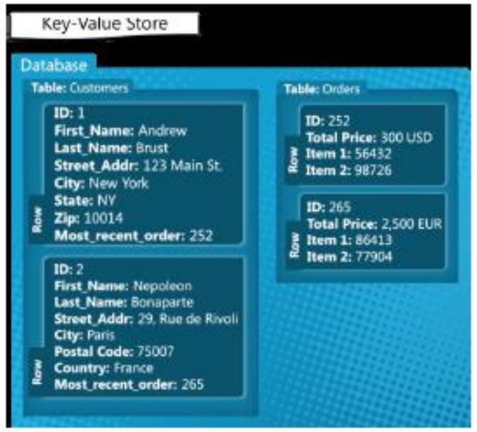
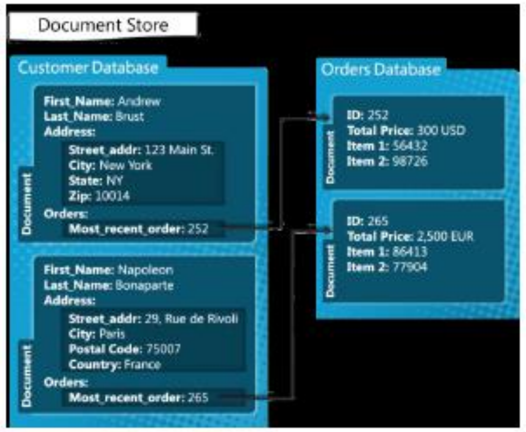
Graph Databases

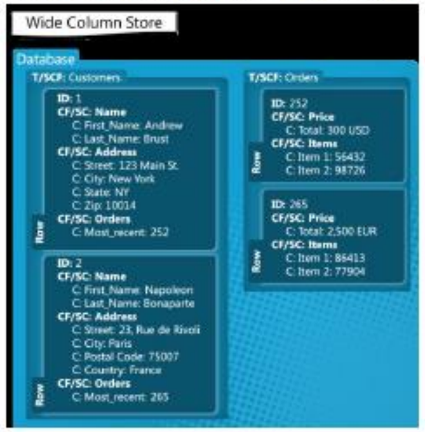
The 4 main types of NoSQL databases



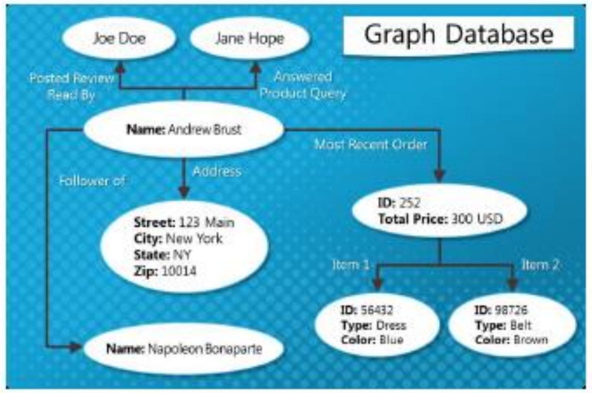
(a) Example of Key-Value Store



(b) Example of Document Store



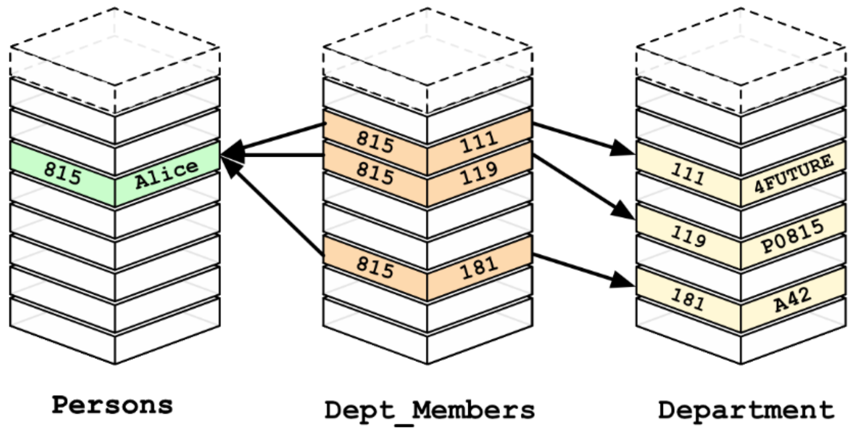
(c) Example of Wide Column Store



(d) Example of Graph Database

From Relational to Graph

- In **relational** databases, references to other rows and tables are indicated by referring to their **primary-key** attributes via **foreign-key** columns.



- If you use many-to-many relationships, you have to introduce a ***JOIN* table** that holds foreign keys of both participating tables which further increases join operation costs.

- Those costly join operations are usually addressed by denormalising data to reduce the number of joins necessary.

Advantage of Graph Databases

- When there are relationships that you want to analyze, Graph databases become a very nice fit because of the data structure
- Graph databases are very fast for associative data sets
  - Like social networks
- Map more directly to object oriented applications
  - Object classification and Parent->Child relationships

Neo4J

What is Neo4j

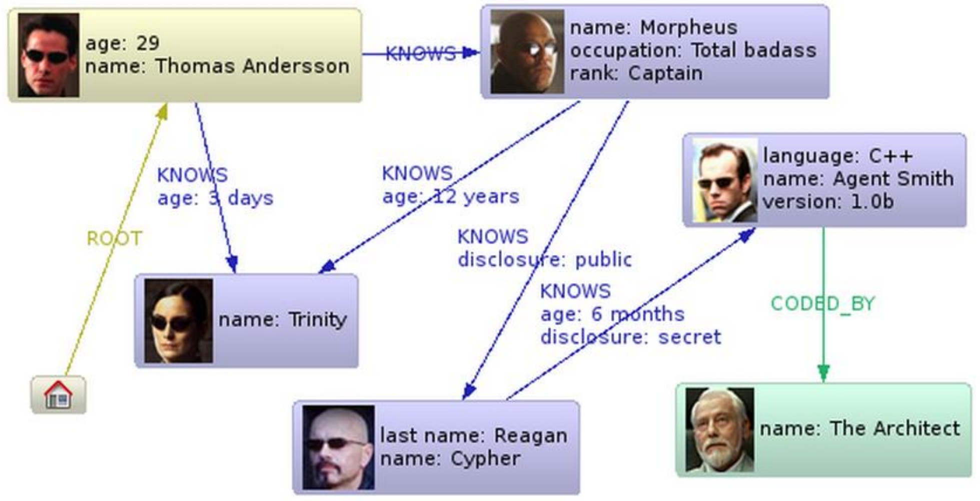
- **Introduced in 2010**
- Developed by Neo Technologies
- Most Popular Graph Database
- Implemented in Java
- Open Source

Neo4j
the world's
leading graph database

(www.neo4j.org)

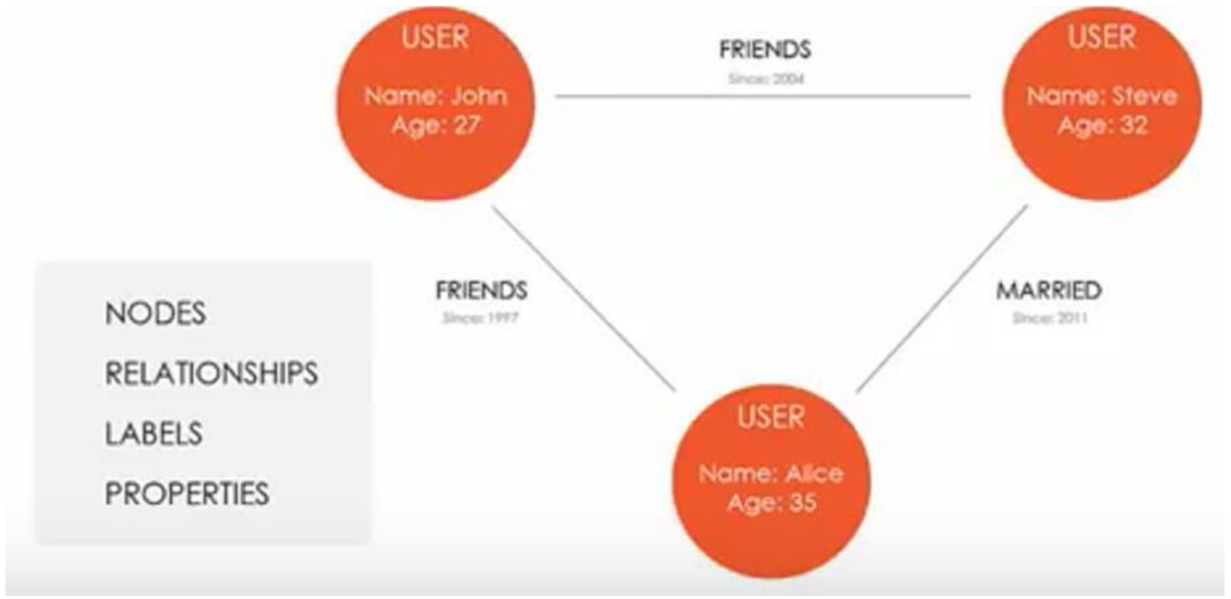
Neo4j

- Neo4j is a graph database
- Database full of linked nodes
- Stores data as nodes and relationships



neo4j.org

Neo4j


USER
Name: John
Age: 27
FRIENDS
Since: 2004
USER
Name: Steve
Age: 32
FRIENDS
Since: 1997
MARRIED
Since: 2011
USER
Name: Alice
Age: 35
NODES
RELATIONSHIPS
LABELS
PROPERTIES

Schemaless

- Unlike a traditional RDBMS Neo4j is a *schemaless* database.
- You don't need to define tables and relationships before you can start adding data.
- A node can have any properties you like, and any node can be related to any other node.

So what does a (property) graph look like?

Graph components

Node (Vertex)

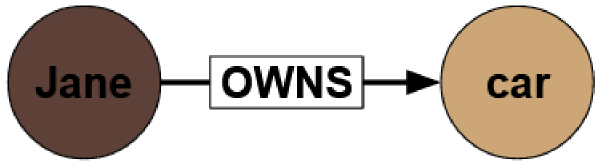
- The main data element from which graphs are constructed

Graph components

Node (Vertex)

- The main data element from which graphs are constructed

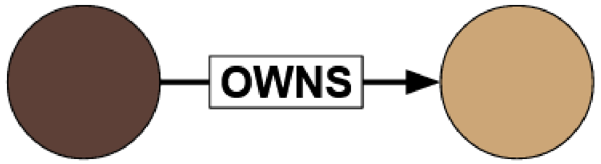
Relationship (Edge)



- A link between two nodes. Has:
  - Direction
  - Type
- *A node without relationships is permitted. A relationship without nodes is not*

Property graph database

Node (Vertex)
Relationship (Edge)



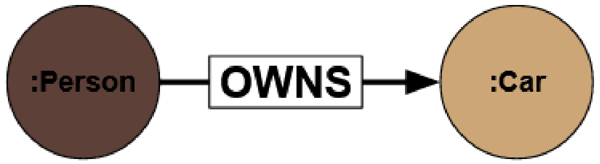
Property graph database

Node (Vertex)

Relationship (Edge)

Label

- Define node category (optional)



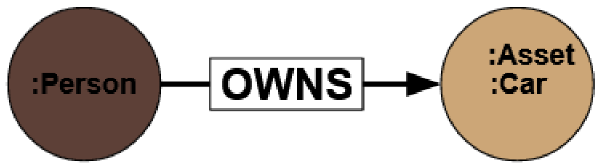
Property graph database

Node (Vertex)

Relationship (Edge)

Label

- Define node category (optional)
- Can have more than one



Property graph database

Node (Vertex)

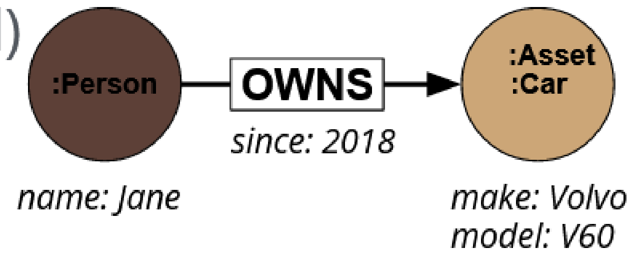
Relationship (Edge)

Label

- Define node category (optional)
- Can have more than one

Properties

- Enrich a node or relationship
- No need for nulls!



How do I query the graph?

Cypher

A pattern-matching query language made for graphs

Cypher

- Neo4j’s query language is called Cypher
- Cypher is a declarative query language for graphs that uses graph pattern-matching as a main mechanism for graph data selection
- Another popular graph query language is Gremlin which Neo4J also supports

Neo4j CQL

- Is a query language for Neo4j Graph Database.
- Is a declarative pattern-matching language.
- Follows SQL like syntax.
- Syntax is very simple and in human readable format.

Like Oracle SQL

- Neo4j CQL has commands to perform Database operations.
- Neo4j CQL supports many clauses such as WHERE, ORDER BY, etc., to write very complex queries in an easy manner.
- Neo4j CQL supports some functions such as String, Aggregation. In addition to them, it also supports some Relationship Functions.

Cypher – Graph Query Language

- Data Model is composed of nodes and relationships that connect them

- **Nodes**: Entities or objects round brackets ()
- **Relationships**: Links between nodes square brackets []

  Direction angle brackets <   >
- **Label**: Type of node or relationship colon :
- **Property**: Features of a node or relationship braces {}

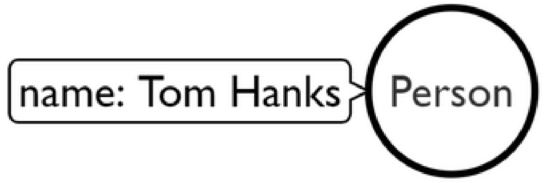
Description	Node	Relationship
Generic	()	--　　-->　　-[]-
With a reference	(n)	-[r]-
With a node label or rel type	(:Person)	-[:ACTED_IN]-
With a label/type and an inline property	(:Person {name: ‘Bob’})	-[:ACTED_IN {role: ‘Dave’}]-
With a reference, label/type and an inline property	(p:Person {name: ‘Bob’})	-[r:ACTED_IN {role: ‘Rob’}]-

Use MATCH to retrieve nodes

```
//Match all nodes
MATCH (n)
RETURN n;
```

Use MATCH to retrieve nodes

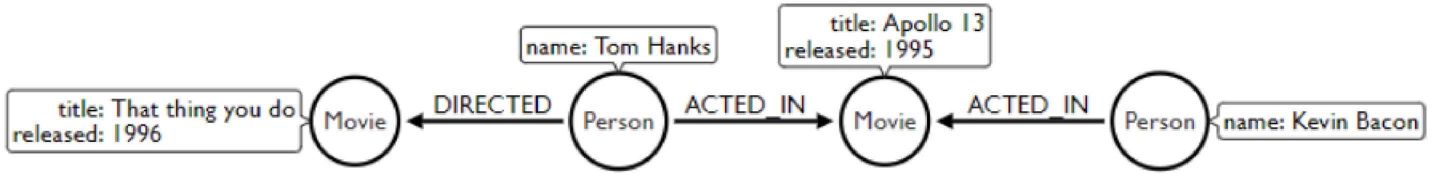
```
//Match all nodes
MATCH (n)
RETURN n;


//Match all nodes with a Person label
MATCH (n:Person)
RETURN n;


//Match all nodes with a Person label and property name is "Tom Hanks"
MATCH (n:Person {name: "Tom Hanks"})
RETURN n;
```



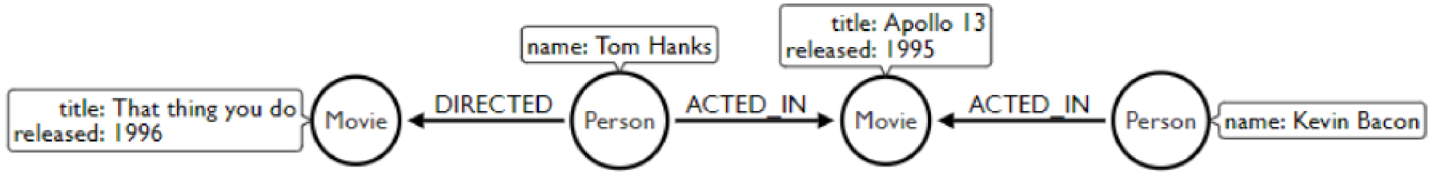
Use MATCH and properties to retrieve nodes

```
//Return nodes with label Person and name property is "Tom Hanks" - Inline
MATCH (p:Person {name: "Tom Hanks"}) //Only works with exact matches
RETURN p;

//Return nodes with label Person and name property equals "Tom Hanks"
MATCH (p:Person)
WHERE p.name = "Tom Hanks"
RETURN p;

//Return nodes with label Movie, released property is between 1991 and 1999
MATCH (m:Movie)
WHERE m.released > 1990 AND m.released < 2000
RETURN m;
```

Extending the MATCH



Extending the MATCH



```
//Find all the movies Tom Hanks acted in
MATCH (:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m:Movie)
RETURN m.title;

//Find all the movies Tom Hanks directed and order by latest movie
MATCH (:Person {name:"Tom Hanks"})-[:DIRECTED]->(m:Movie)
RETURN m.title, m.released ORDER BY m.released DESC;

//Find all of the co-actors Tom Hanks have worked with
MATCH (:Person {name:"Tom Hanks"})-->(:Movie)<-[:ACTED_IN]-(coActor:Person)
RETURN coActor.name;
```

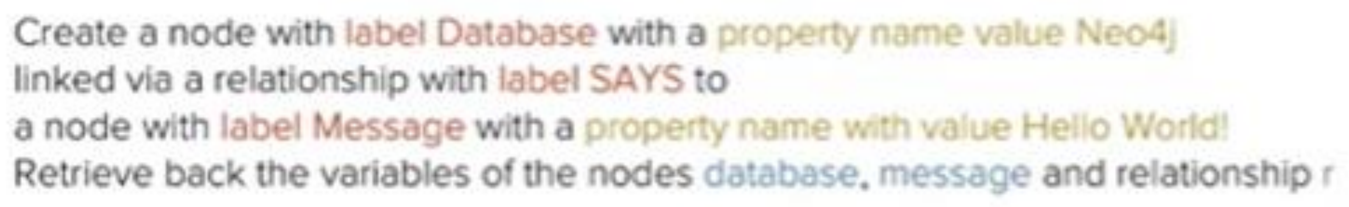
CREATE

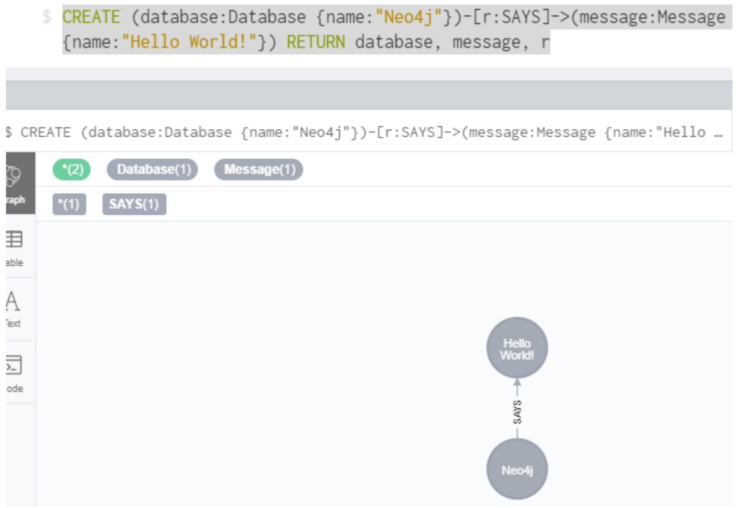
```
//Create a person node called "Tom Hanks"
CREATE (p:Person {name:"Tom Hanks"});


//Create an ACTED_IN relationship between "Tom Hanks" and "Apollo 13"
MATCH (p:Person {name:"Tom Hanks"}), (m:Movie {title:"Apollo 13"})
CREATE (p)-[:ACTED_IN]->(m);


//Create the pattern of "Tom Hanks" ACTED_IN "Apollo 13"
//This will create the entire pattern, nodes and all!
CREATE (:Person {name:"Tom Hanks")-[:ACTED_IN]->(:Movie {title:"Apollo 13});
```

Hello World Example

CREATE (database:Database {name:"Neo4j"})-[r:SAYS]->(message:Message {name:"Hello World!"})

RETURN database, message, r


Create a node with label Database with a property name value Neo4j
linked via a relationship with label SAYS to
a node with label Message with a property name with value Hello World!
Retrieve back the variables of the nodes database, message and relationship r


```
$ CREATE (database:Database {name:"Neo4j"})-[r:SAYS]->(message:Message
{name:"Hello World!"}) RETURN database, message, r
```

```
$ CREATE (database:Database {name:"Neo4j"})-[r:SAYS]->(message:Message {name:"Hello …
```

Cypher Query, Basic Syntax

MATCH | CREATE	Nodes and relationships of interest
WHERE	Additional filters (labels, properties)
RETURN | DELETE	Elements to retrieve or delete
//	Comments

Match

- **Get all nodes**
- By just specifying a pattern with a single node and no labels, all nodes in the graph will be returned.
- MATCH (n) RETURN n

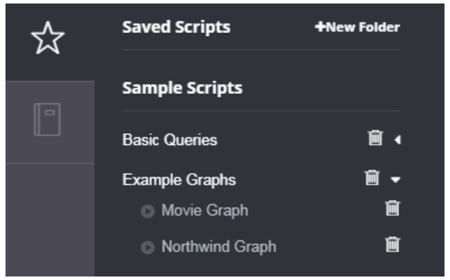
- Returns all nodes in the database

Match

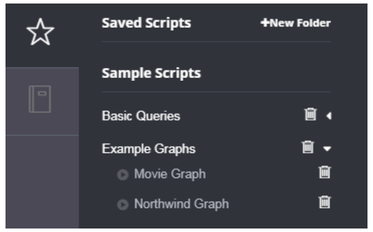
Get all nodes with a label

- Getting all nodes with a label on them is done with a single node pattern where the node has a label on it.

```
MATCH (movie:Movie)
RETURN movie.title
```



Match



- **Related nodes**

The symbol -- means related to, without regard to type or direction of the relationship.

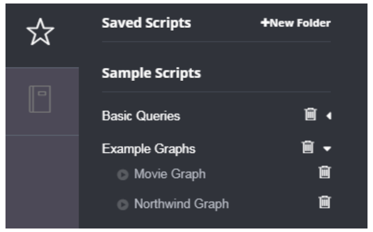
MATCH (Person { name: 'Tom Hanks' })--(Movie)

RETURN Movie.title

Returns all Tom Hanks movies

```
CREATE (TheMatrix:Movie {title:'The Matrix', released:1999, tagline:'Welcome to the Real World'})
CREATE (Keanu:Person {name:'Keanu Reeves', born:1964})
CREATE (Carrie:Person {name:'Carrie-Anne Moss', born:1967})
CREATE (Laurence:Person {name:'Laurence Fishburne', born:1961})
CREATE (Hugo:Person {name:'Hugo Weaving', born:1960})
CREATE (LillyW:Person {name:'Lilly Wachowski', born:1967})
CREATE (LanaW:Person {name:'Lana Wachowski', born:1965})
CREATE (JoelS:Person {name:'Joel Silver', born:1952})
CREATE
(Keanu)-[:ACTED_IN {roles:['Neo']}]->(TheMatrix),
(Carrie)-[:ACTED_IN {roles:['Trinity']}]->(TheMatrix),
(Laurence)-[:ACTED_IN {roles:['Morpheus']}]->(TheMatrix),
(Hugo)-[:ACTED_IN {roles:['Agent Smith']}]->(TheMatrix),
(LillyW)-[:DIRECTED]->(TheMatrix),
(LanaW)-[:DIRECTED]->(TheMatrix),
(JoelS)-[:PRODUCED]->(TheMatrix)
```

Match - Relationships



- When the direction of a relationship is of interest, it is shown by using --->or <-- like this:

```
MATCH (:Person { name: 'Kevin Bacon' }) -->(movie)
RETURN movie.title
```

Returns any nodes connected with the Person ‘Kevin Bacon' by an outgoing relationship.

```
MATCH (:Person { name: 'Kevin Bacon' })-[r:]->(movie)
RETURN type(r)
```

Returns the type of each outgoing relationship from ‘Kevin Bacon'

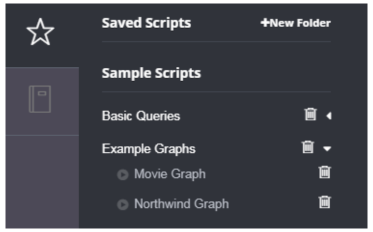
Match - Relationships

- Match on relationship type

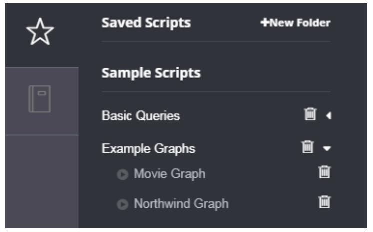
When you know the relationship type you want to match on, you can specify it by using a colon together with the relationship type.

```
MATCH (m:Movie { title: ' The Matrix' })<-[:ACTED_IN]-(Person)
RETURN Person.name
```

- Returns all Persons that ACTED_IN The Matrix '.



Match - relationships



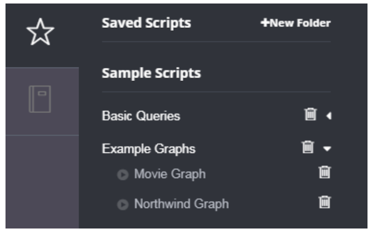
- To match on one of multiple types, you can specify this by chaining them together with the pipe symbol |.

MATCH (m { title: ‘The Matrix' })<-[:ACTED_IN|:DIRECTED]-(person)

RETURN person.name

Returns nodes with an ACTED_IN or DIRECTED relationship to ‘The Matrix'.

Match



- **Find all movies**

```
MATCH (n:Movie) RETURN n
```

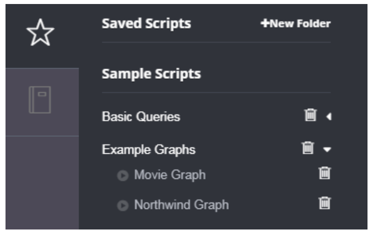
- **Find The Matrix movie**

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

- **Find all movies with Matrix in their title**

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

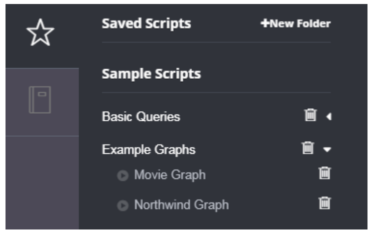
Match



- **Count all movies with Matrix in their title**

```
MATCH (m:Movie)
WHERE m.title =~ '.*Matrix.*'
RETURN count(m);
```

Update



- **Update Hugo Weaving’s year of birth to 1961**

MATCH (p:Person{name:'Hugo Weaving'}) SET p.born = 1961 return p

- **Update Hugo Weaving’s to have a phone number 123456**

MATCH (p:Person{name:'Hugo Weaving'}) SET p.phone = 123456 return p

Delete

The DELETE clause is used to delete nodes, relationships or paths.

- **Delete single node**

```
MATCH (n:Person { name: 'UNKNOWN' }) DELETE n
```

- **Delete all nodes and relationships**

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

Delete ctd.

- **Delete a node with all its relationships**

When you want to delete a node and any relationship going to or from it, use DETACH DELETE.

```
MATCH (n { name: 'Andy' })
DETACH DELETE n
```

Delete ctd.

- **Delete relationships only**

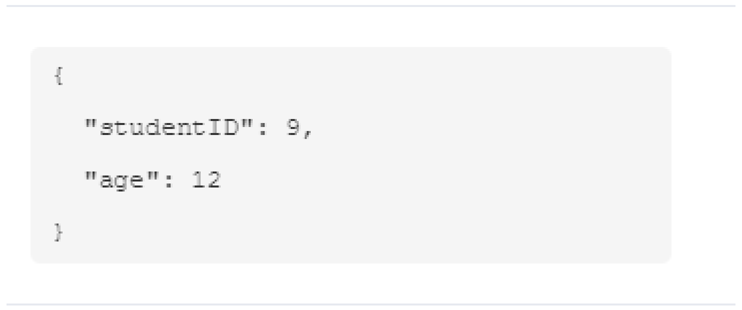
It is also possible to delete relationships only, leaving the node(s) otherwise unaffected.

```
MATCH (n { name: 'Andy' })-[r:KNOWS]->()
DELETE r
```

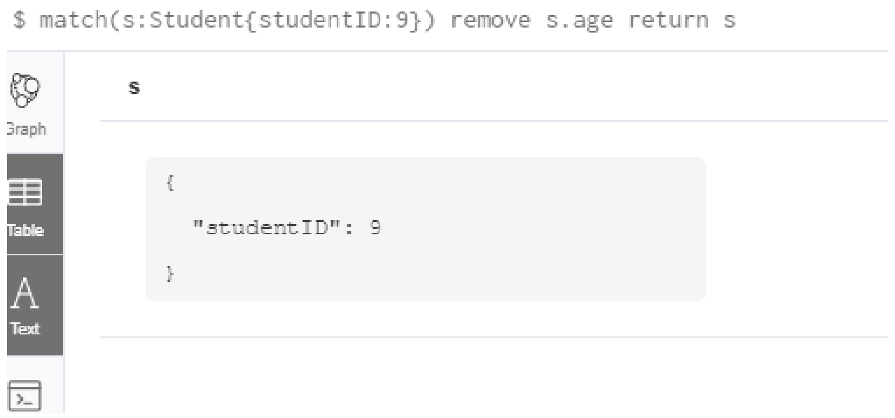
This deletes all outgoing KNOWS relationships from the node with the name 'Andy'.

Delete/Remove a property

- Remove is used to remove a property value from a node or a relationship



```
{
  "studentID": 9,
  "age": 12
}
```

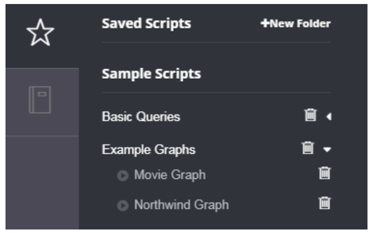


```
$ match(s:Student{studentID:9}) remove s.age return s
```

```
s

{
  "studentID": 9
}
```

- The node is returned and no age property exists

Max / Min



- **Find the max year**

```
match(p:Person)return  max(p.born)
```

- **List the name of the youngest person**

```
 match(p:Person)with p order by p.born desc  where p.born is not null
return p.name, p.born limit 1
```

With Clause: This clause is used to continue the query with the variable p

Average and Count

- **What is the average review rating?**

```
MATCH p = ()-[r:REVIEWED]->()
RETURN avg(r.rating);
```

REVIEWED is a relationship type

Rating is a Property of Reviewed

- **Count the number of movies**

```
MATCH (n:Movie) RETURN count(n)
```

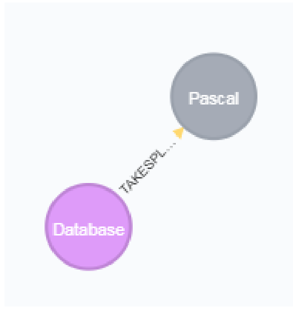
Merge

- MERGE either matches existing nodes and binds them, or it creates new data and binds that.
- It’s like a combination of MATCH and CREATE

Example for College Graph

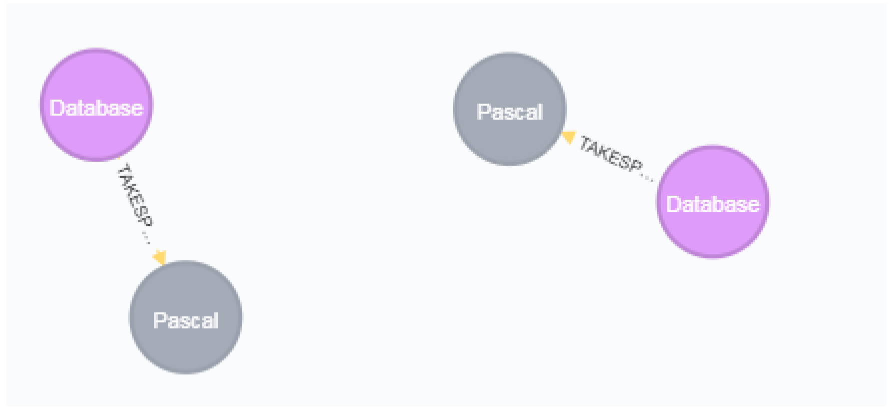
```
merge(c1:Course{courseNr:"1",courseName:"Database"}) - [:TAKESPLACEIN] ->
(r:room{roomName:"Pascal"}) return c1
```

- This will create the relationship to state that databases takes place in Pascal room. It will not create any new room or course nodes



Merge ctd.

- If you used the CREATE instead of MERGE then you would have a second/ duplicate relationship created
- Merge will avoid this problem

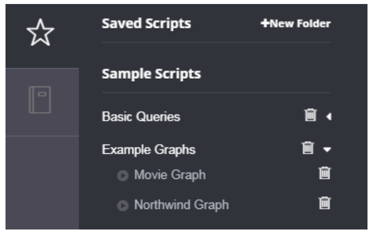


1. **Matching:**
   - If the specified pattern already exists in the graph, the **MERG**
     will find and return that existing pattern.

   No changes are made.

2. **Creating:**
   - If the pattern does not exist, the **MERGE** statement will create
   the specified pattern in the graph.

   - It creates a new node with the given label and properties.



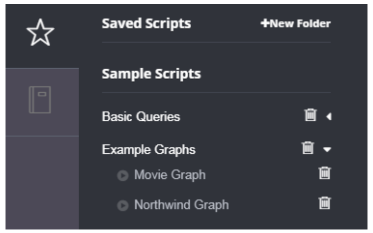
Pattern Matching

- Pattern matching is the key aspect of Cypher

```
match (user)-[:HAS_SEEN]->(movie)
return movie;
```

- When describing a relationship, the relationship type is specified after a colon (:)within the square brackets. The type should appear exactly as it was defined when the relationship was created (syntax is case-sensitive). This simple query describes a single HAS_SEEN relationship using the [:HAS_SEEN] syntax

Pattern Matching ctd.



- Using node & relationship identifiers

Both nodes and relationships in Cypher queries can be associated with identifiers, which are used to reference the same graph entity later in the same query. In the following example, the node named movie has been referenced in the return clause.

Relationships can be named as well, using slightly different syntax:

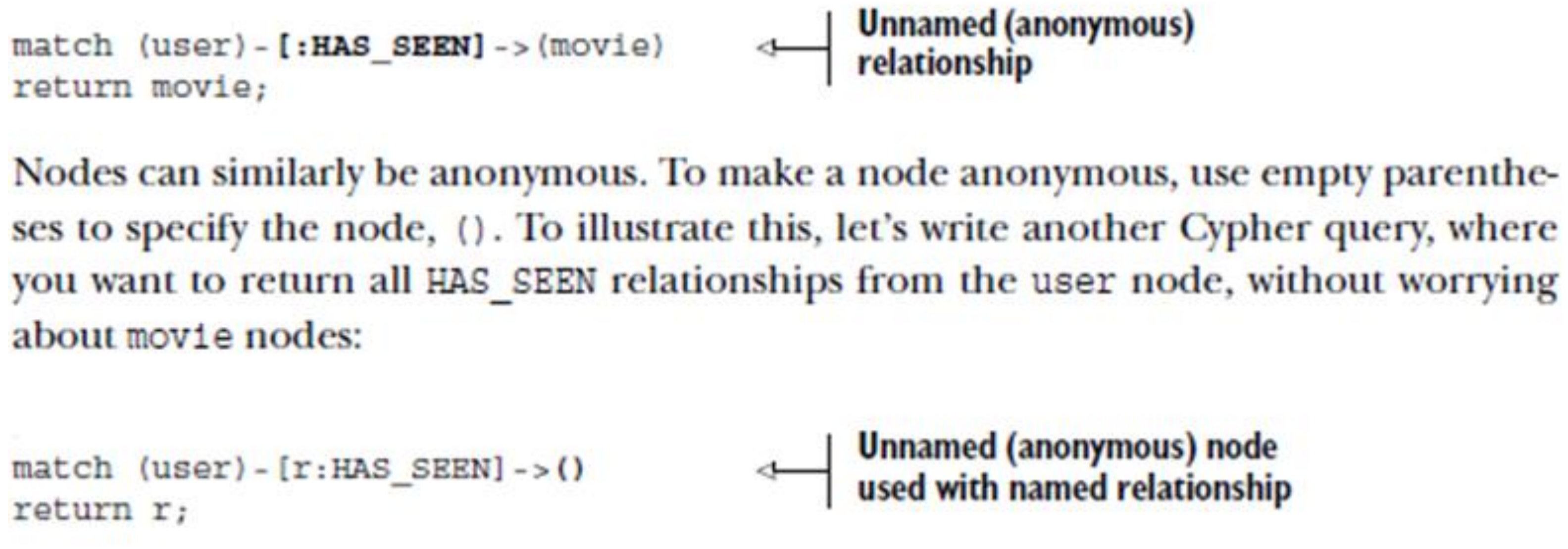
```
match (user)-[r:HAS_SEEN]->(movie)
return movie;
```

Names relationships by specifying the name before the colon and relationship type

Pattern Matching ctd.

- Naming nodes and relationships can be very useful when creating complex patterns, because you can reference them later in other parts of a Cypher query. But if you don’t need to reference the graph entity later, avoiding identifiers will make your query easier to read and understand.

Pattern Matching ctd.



```
match (user)-[:HAS_SEEN]->(movie)
return movie;
```

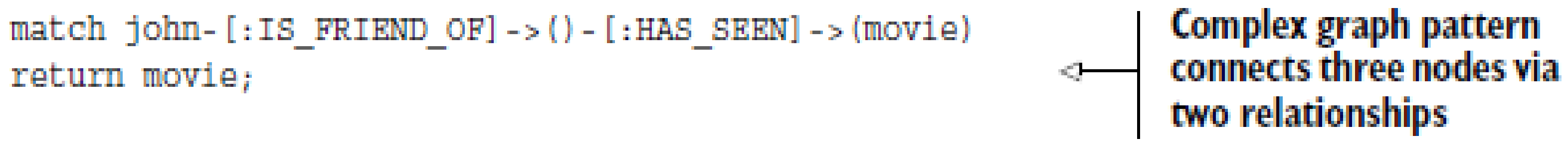
Unnamed (anonymous) relationship

Nodes can similarly be anonymous. To make a node anonymous, use empty parentheses to specify the node, `()`. To illustrate this, let's write another Cypher query, where you want to return all `HAS_SEEN` relationships from the `user` node, without worrying about `movie` nodes:

```
match (user)-[r:HAS_SEEN]->()
return r;
```

Unnamed (anonymous) node used with named relationship

Complex Pattern Matching



```
match john-[:IS_FRIEND_OF]->()-[:HAS_SEEN]->(movie)
return movie;
```

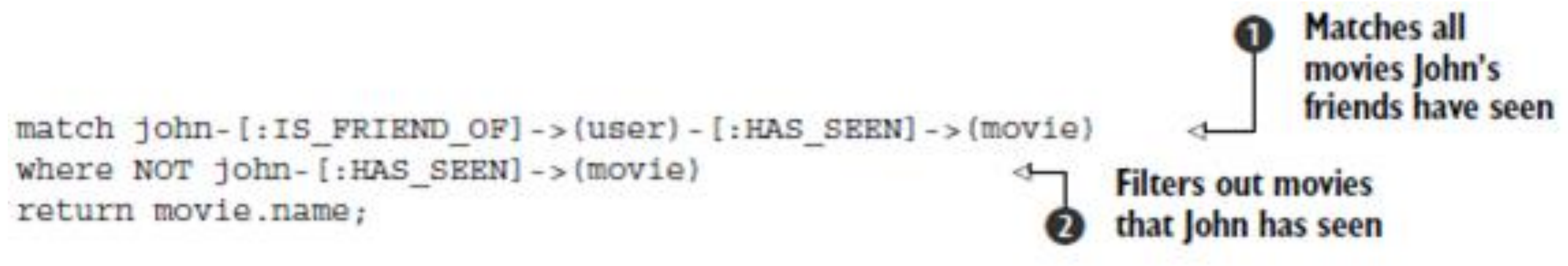
Complex graph pattern connects three nodes via two relationships

you match the outgoing IS_FRIEND_OF relationship to another node (which is left anonymous), which in turn has a HAS_SEEN relationship to another named node, movie.
You don’ t need to name any relationships or intermediate nodes because you’ re not interested in them; all you want to return are nodes representing movies that John’ s friends have seen.

Running this query will return all movie nodes that any of John’ s friends have seen,

Complex Pattern Matching ctd.

- This kind of pattern matching can be used for recommender system.
- E.g. if you want to recommend movies to John based on the movies his friend have seen, then you should skip the movies John has already seen

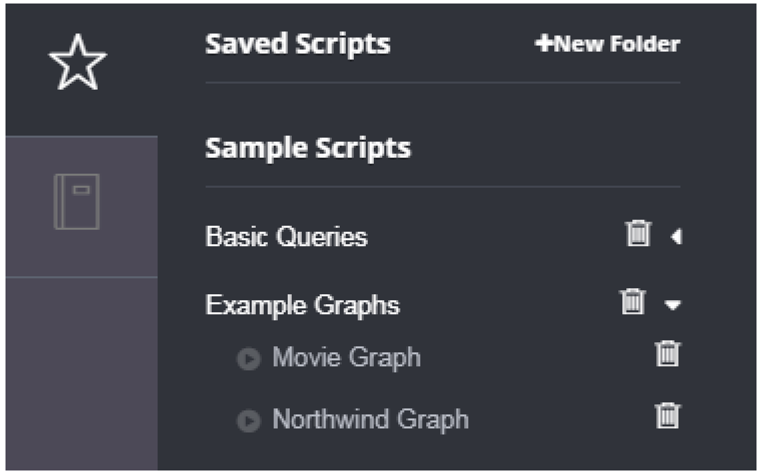


```
match john-[:IS_FRIEND_OF]->(user)-[:HAS_SEEN]->(movie)
where NOT john-[:HAS_SEEN]->(movie)
return movie.name;
```

1 Matches all movies John's friends have seen
2 Filters out movies that John has seen

Queries for Movies Database

- Run the following queries using the example graph Movie Graph



Actors who played in some movie

```
MATCH (m:Movie {title: 'The Matrix'})<-[:ACTED_IN]-(p:Person)
RETURN p.name, p.born
```

Find the most prolific actors

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

Find actors who have been in less than 3 movies

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WITH p, count(m) AS movie_count
WHERE movie_count < 3
RETURN p, movie_count
ORDER BY movie_count DESC LIMIT 5;
```

Find the actors with 5+ movies, and the movies in which they acted

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie) WITH p,
collect(m.title) AS movies
WHERE size(movies) >= 5
RETURN p, movies
ORDER BY size(movies) DESC LIMIT 10;
```

(The collect() function aggregates values into a list)