Data Engineering LTAT.02.007

Ass Prof. Riccardo Tommasini
Fabiano Spiga, Hassan Eldeeb, Mohamed Ragab

Graph Databases (Neo4j)

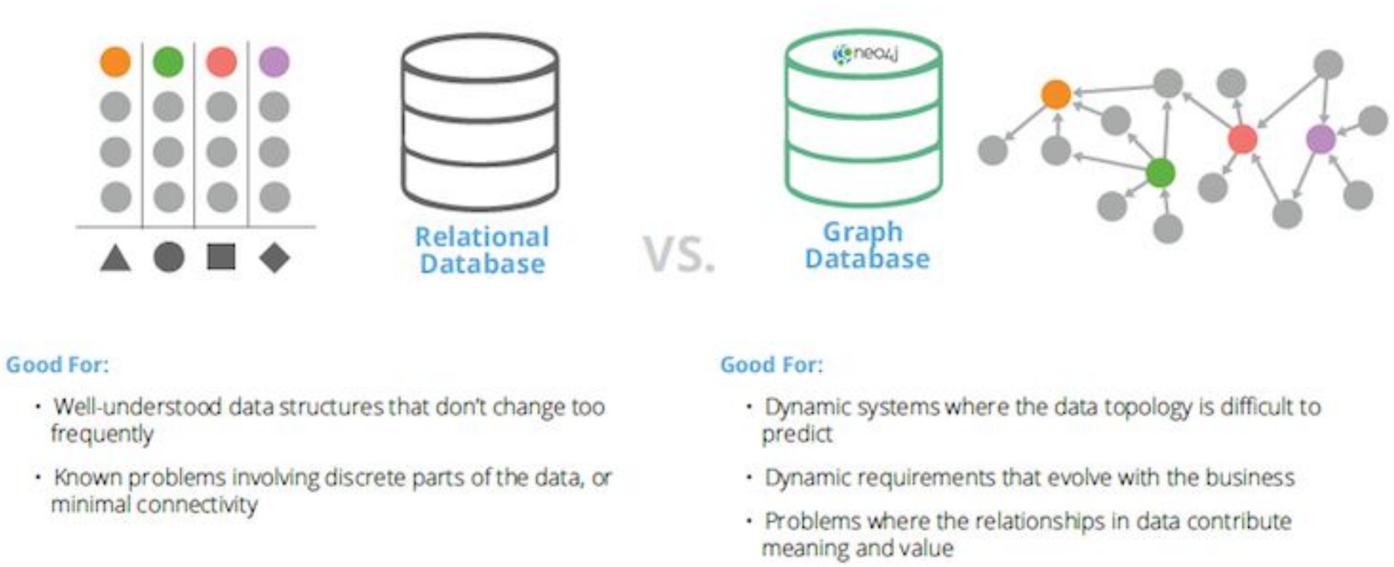
lab 05

https://courses.cs.ut.ee/2020/dataeng

Agenda

- Relational vs. graph databases
    - Which to use and when?
- From Relational to Graph
- What is Neo4j?
- The Property Graph Model (PGM)
- Cypher Query Language
- Cypher In Action
    - MovieGraph Dataset in neo4j
- Cypher Query Clauses and Examples

Relational vs. graph databases



Good For:

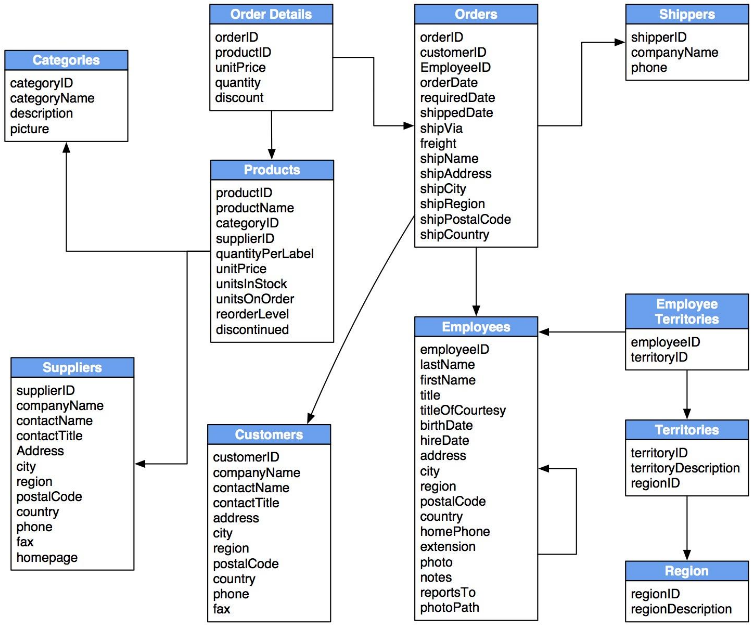
- Well-understood data structures that don't change too frequently
- Known problems involving discrete parts of the data, or minimal connectivity

Good For:

- Dynamic systems where the data topology is difficult to predict
- Dynamic requirements that evolve with the business
- Problems where the relationships in data contribute meaning and value

https://neo4j.com/news/relational-vs-graph-databases/

From Relational to Graph



Categories
categoryID
categoryName
description
picture

Order Details
orderID
productID
unitPrice
quantity
discount

Products
productID
productName
categoryID
supplierID
quantityPerLabel
unitPrice
unitsInStock
unitsOnOrder
reorderLevel
discontinued

Orders
orderID
customerID
EmployeeID
orderDate
requiredDate
shippedDate
shipVia
freight
shipName
shipAddress
shipCity
shipRegion
shipPostalCode
shipCountry

Shippers
shipperID
companyName
phone

Suppliers
supplierID
companyName
contactName
contactTitle
Address
city
region
postalCode
country
phone
fax
homepage

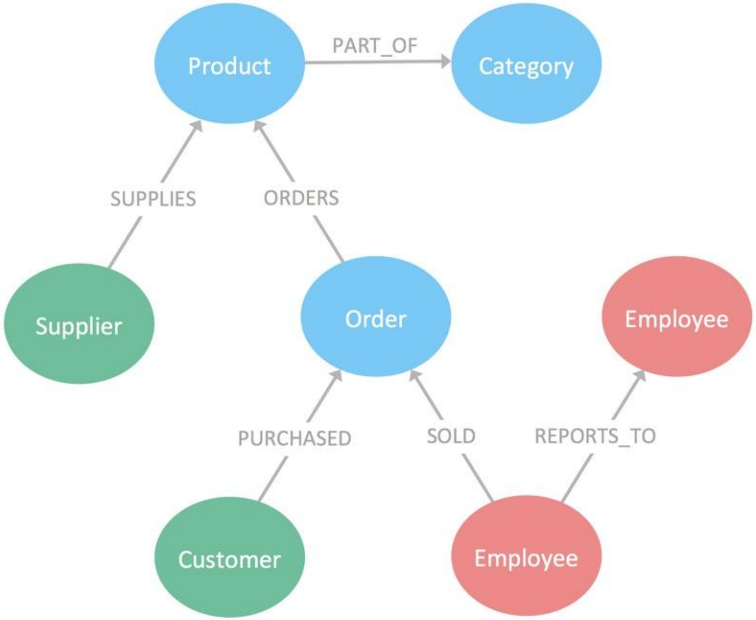
Customers
customerID
companyName
contactName
contactTitle
address
city
region
postalCode
country
phone
fax

Employees
employeeID
lastName
firstName
title
titleOfCourtesy
birthDate
hireDate
address
city
region
postalCode
country
homePhone
extension
photo
notes
reportsTo
photoPath

Employee Territories
employeeID
territoryID

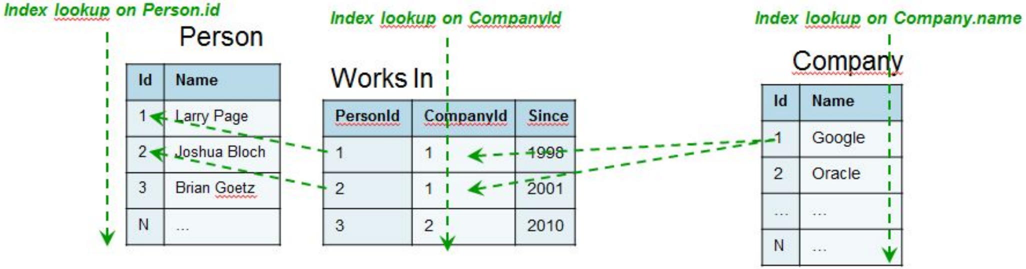
Territories
territoryID
territoryDescription
regionID

Region
regionID
regionDescription



https://neo4j.com/business-edge/relational-to-relationships/

From Relational to Graph



Index lookup on Person.id
Index lookup on CompanyId
Index lookup on Company.name

Person

Id	Name
1	Larry Page
2	Joshua Bloch
3	Brian Goetz
N	...

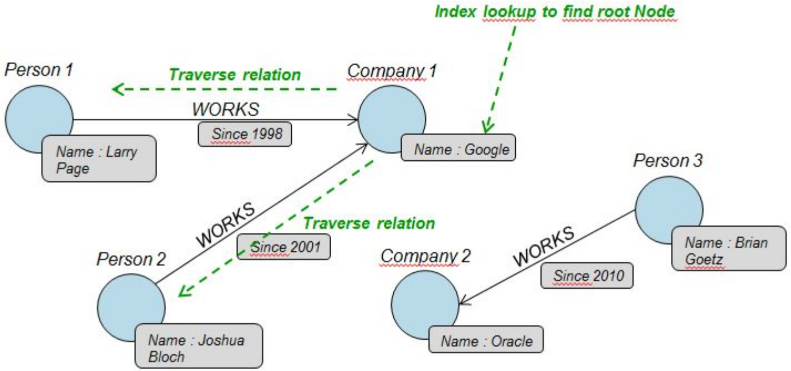
Works In

PersonId	CompanyId	Since
1	1	1998
2	1	2001
3	2	2010

Company

Id	Name
1	Google
2	Oracle
...	...
N	...

```
Select Person.Name
from Person, Company, WorksIn
where Company.name='Google'
and WorksIn.CompanyId = Company. Id
and WorksIn.PersonId = Person.Id
```



https://blog.octo.com/en/graph-databases-an-overview/

Graph Databases

neo4j

OrientDB

ArangoDB

Amazon Neptune

TITAN

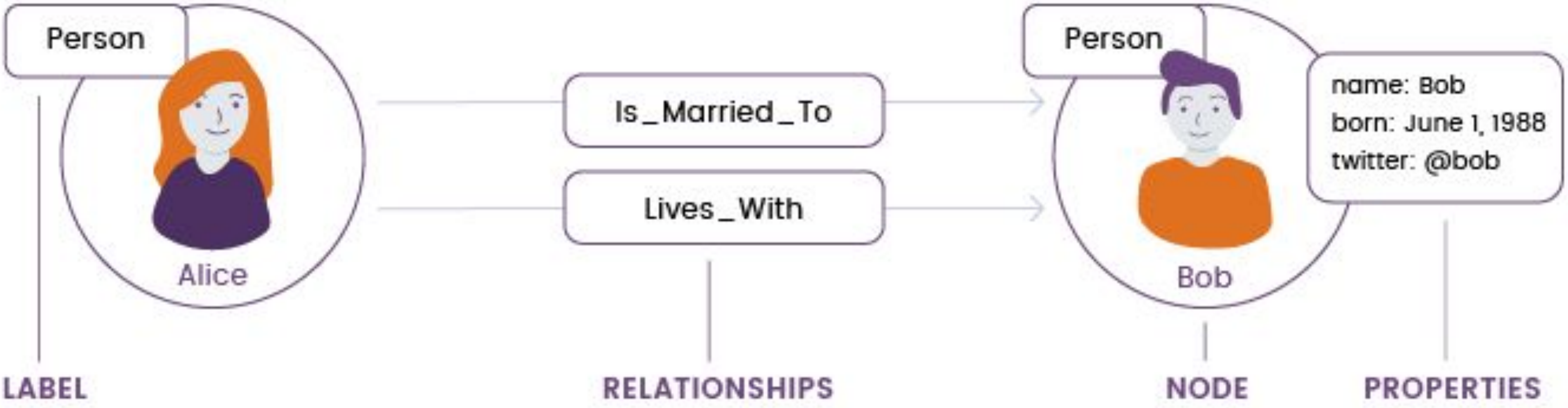
AllegroGraph
Franz Inc.

https://www.g2.com/categories/graph-databases

Neo4j

- A graph database management system developed by Neo4j, Inc.
- Supports ACID (Atomicity, Consistency, Isolation, Durability) properties.
- It implements a Property Graph Model to store the data.
- Its implementation in Java also makes it widely usable, you still can access it using other languages using available drivers.
- It uses Cypher as a declarative query language.

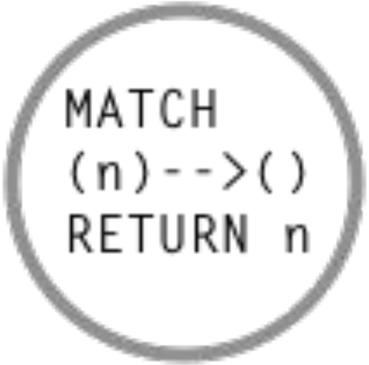
Property Graph Model



https://www.slideshare.net/lvca/how-graph-databases-started-the-multi-model-revolution

What is Cypher?

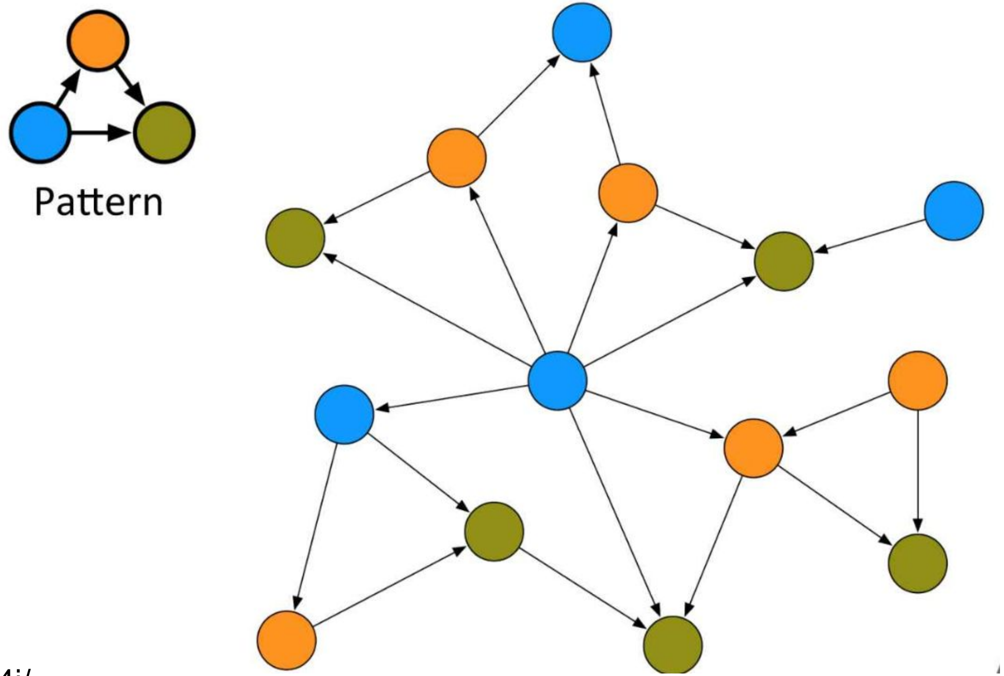
- Graph Query Language for Neo4j, and other GDBs.
- Aims to make graph querying simple.
- Cypher is Declarative QL
    - specify starting point specify desired outcome
    - algorithm adaptable based on query
- Based on graph Pattern matching (ASCII-art patterns).
- Cypher is Familiar for SQL users.



```
MATCH
(n)-->()
RETURN n
```

Pattern Matching

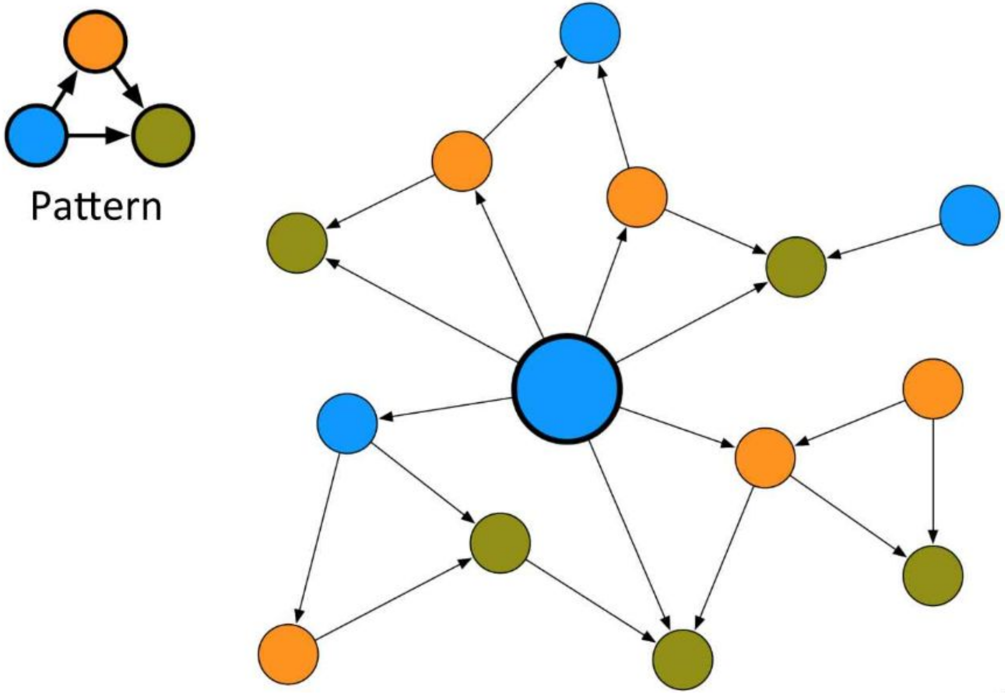
- Queries in form of graph pattern



https://www.infoq.com/presentations/Complex-Data-graph-Neo4j/

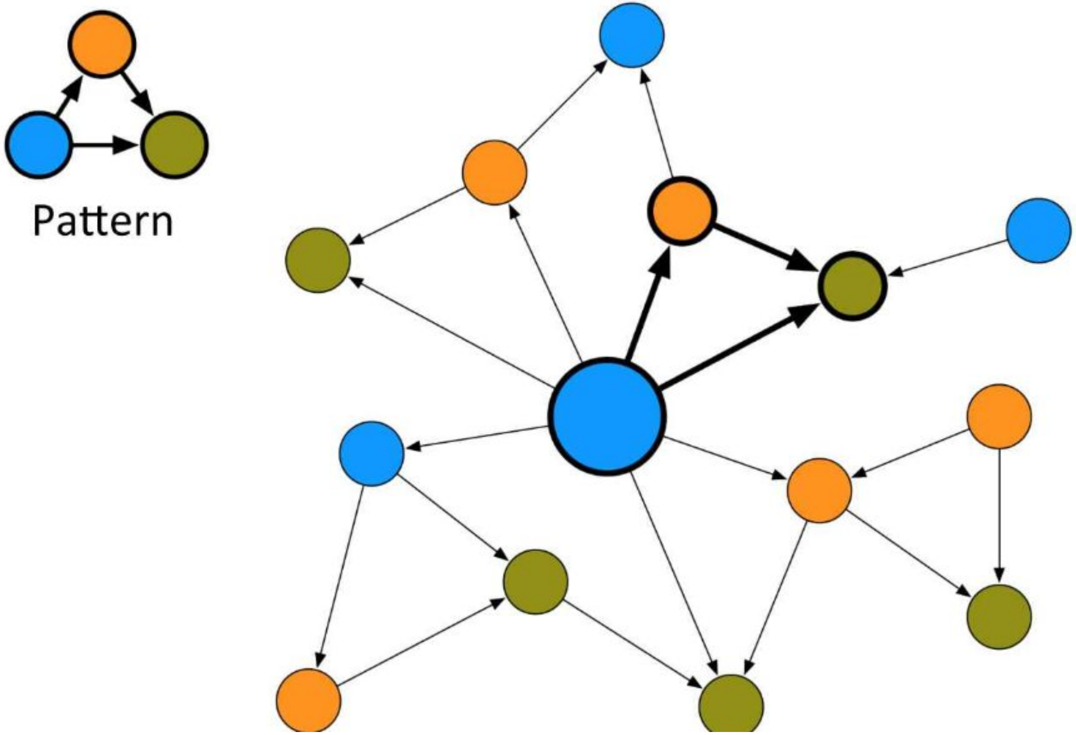
Pattern Matching

- Select at least Start Node



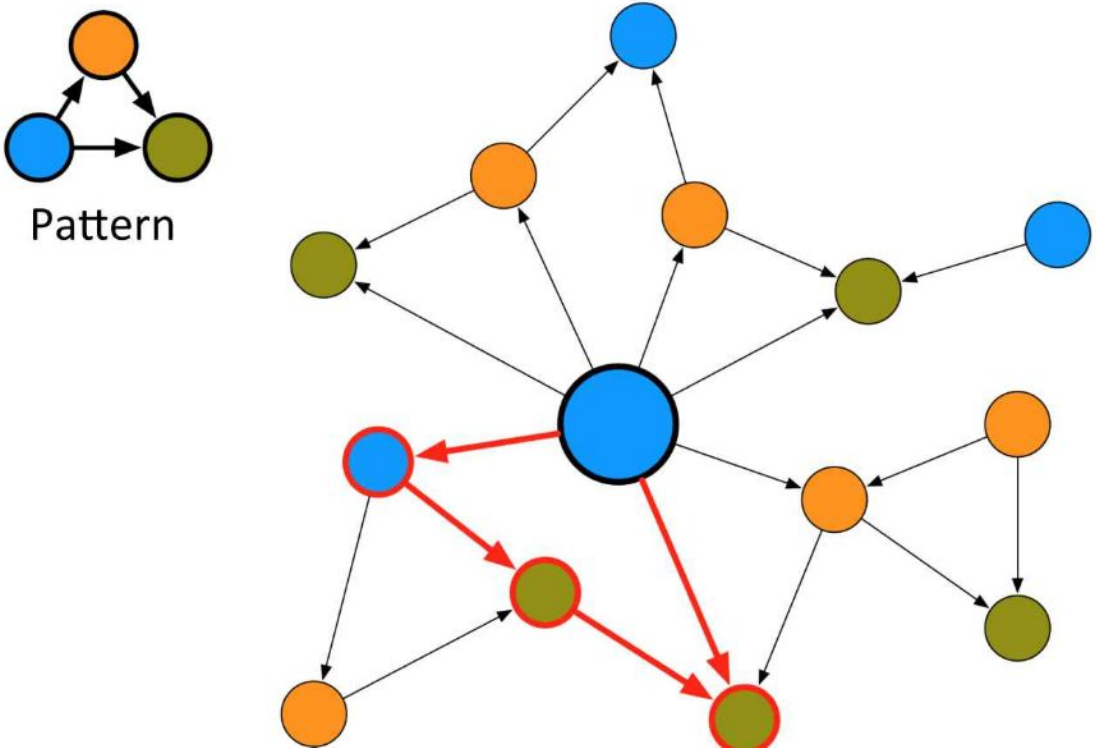
Pattern Matching

- Our first Match



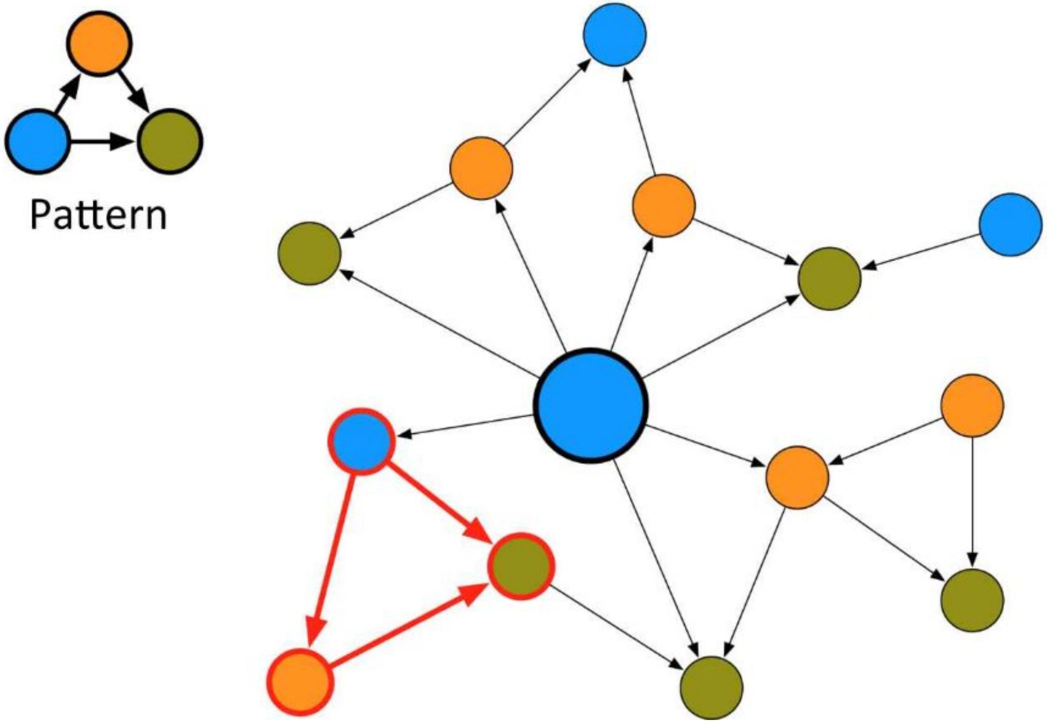
Pattern Matching

- This is not match!



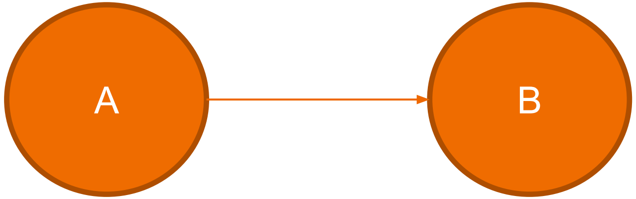
Pattern Matching

- This is also non-Match.
- Why?!



ASCII-art patterns

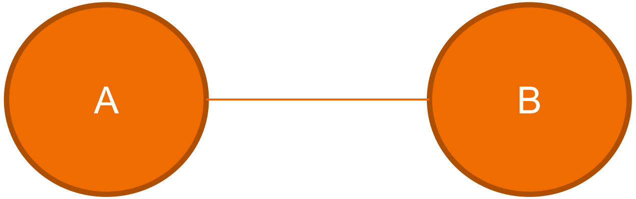
Directed relationship



(A) -->(B)

ASCII-art patterns

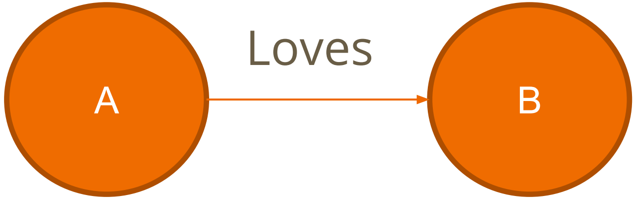
Undirected relationship



```
(A)--(B)
```

ASCII-art patterns

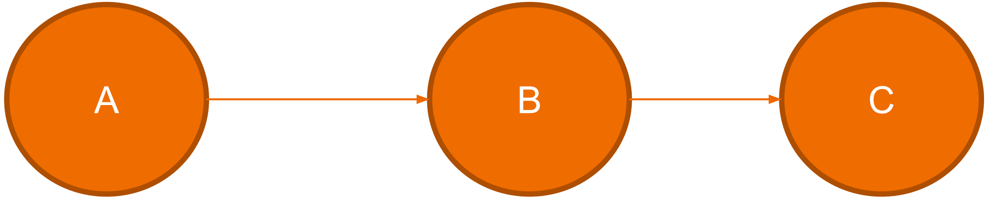
Specified relationship



(A) –[:Loves]->(B)

ASCII-art patterns

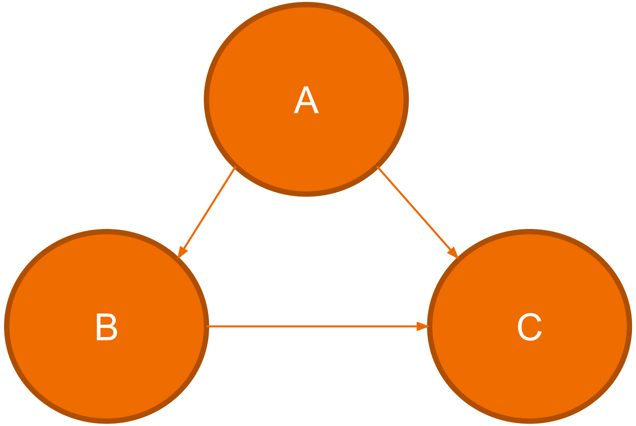
Joined Paths



(A) -->(B) --> (c)

ASCII-art patterns

multiple Paths



```
>>   (A) -->(B) --> (c), (A) --> (c)
>>   (A) -->(B) --> (c) <-- (A)
```

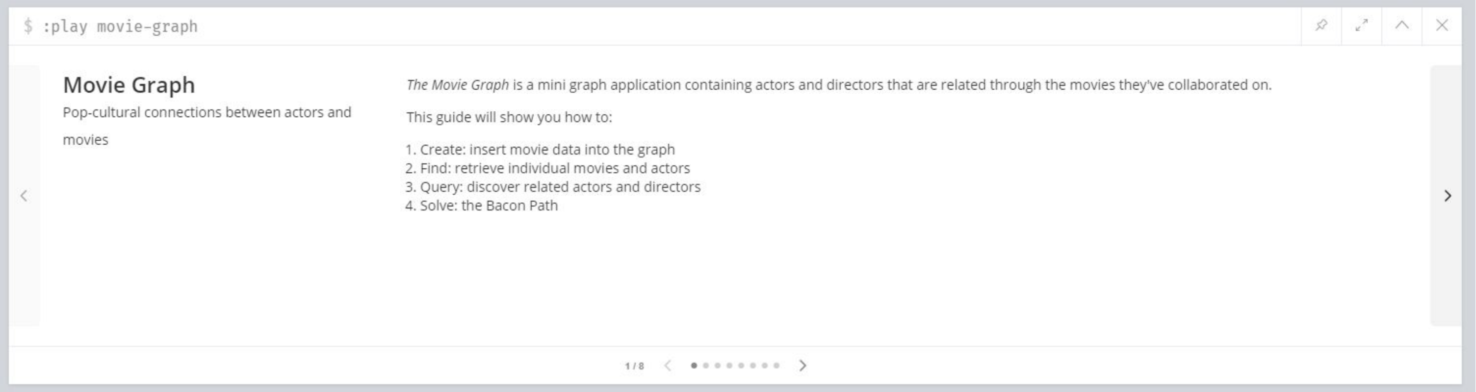
Cypher In Action

https://s3.amazonaws.com/images.seroundtable.com/google-hands-dirty-1436184854.jpg

Movie Dataset Example

- Excerpt of actors, directors, producers,, etc and related movies.
- Available on your installed Neo4j versions.

- In your neo4j browser run:
  :play movie-graph



```
$ :play movie-graph
```

Movie Graph

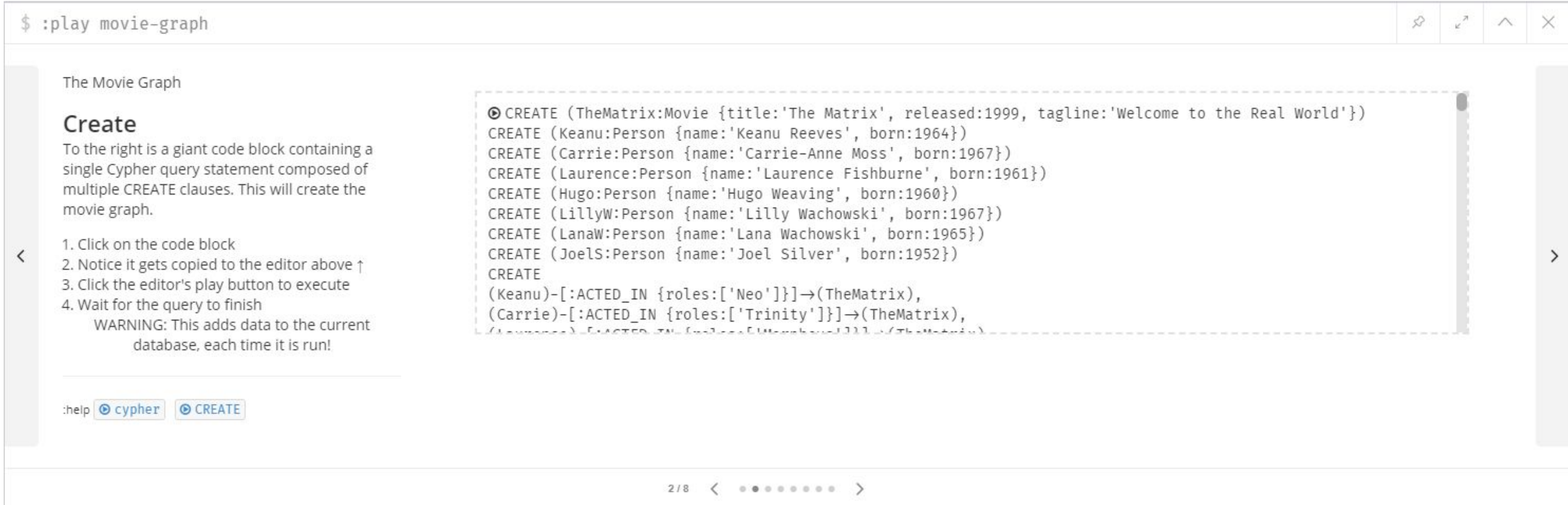
Pop-cultural connections between actors and movies

The Movie Graph is a mini graph application containing actors and directors that are related through the movies they've collaborated on.

This guide will show you how to:

1. Create: insert movie data into the graph
2. Find: retrieve individual movies and actors
3. Query: discover related actors and directors
4. Solve: the Bacon Path

Loading the movie dataset



```
$ :play movie-graph
```

The Movie Graph

Create

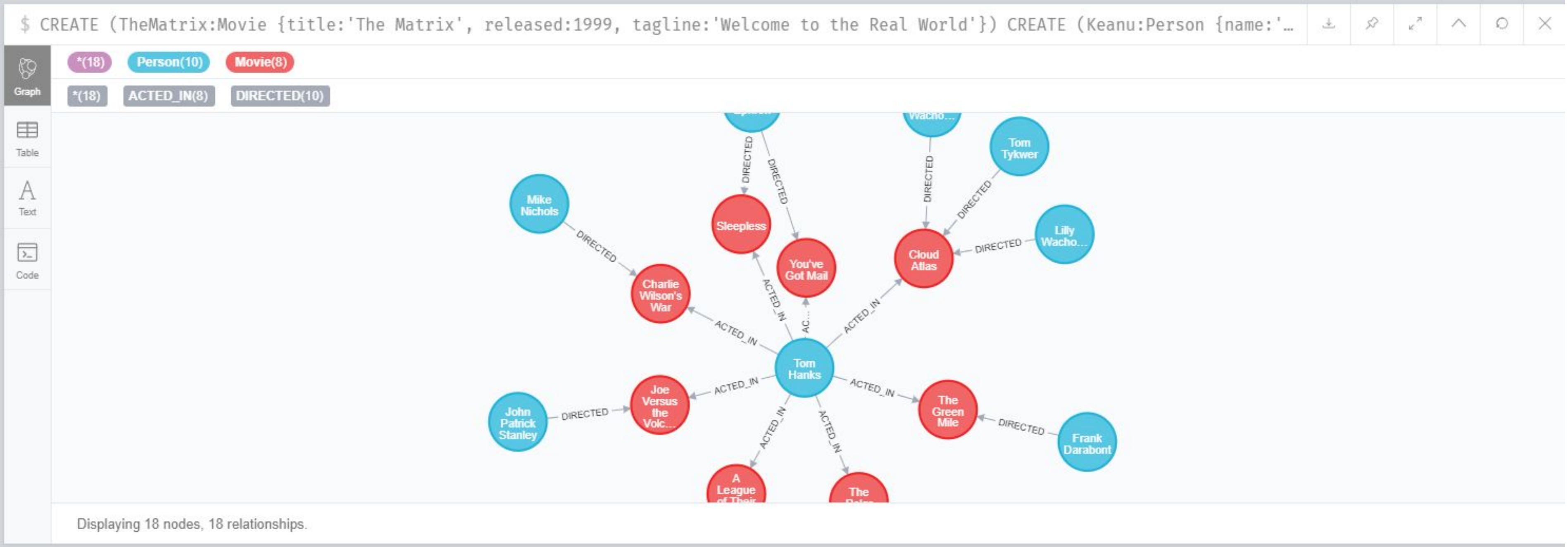
To the right is a giant code block containing a single Cypher query statement composed of multiple CREATE clauses. This will create the movie graph.

1. Click on the code block
2. Notice it gets copied to the editor above ↑
3. Click the editor's play button to execute
4. Wait for the query to finish

WARNING: This adds data to the current database, each time it is run!

```
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),
```

Results of data loading

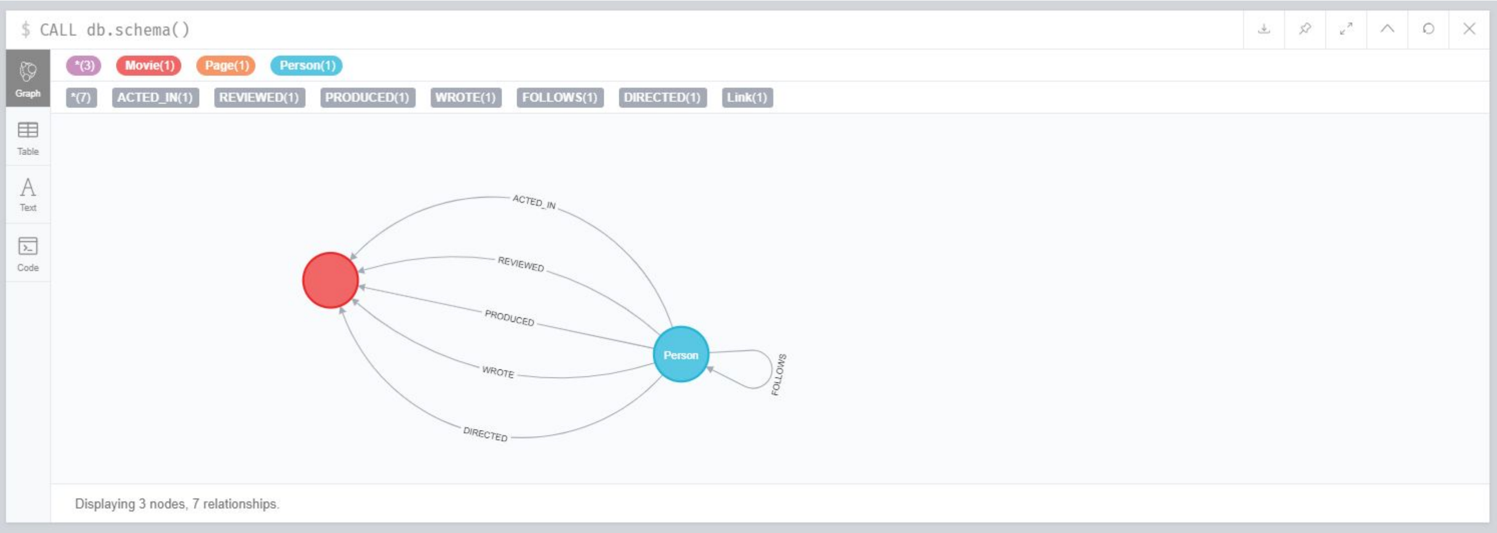


```
$ CREATE (TheMatrix:Movie {title:'The Matrix', released:1999, tagline:'Welcome to the Real World'}) CREATE (Keanu:Person {name:'…
```

Displaying 18 nodes, 18 relationships.

Show the DB Schema

- From the menu Favorites>Common Procedures choose> show meta-graph.
- Or directly run the query: CALL db.schema()



```
$ CALL db.schema()
```

Displaying 3 nodes, 7 relationships.

CREATE Nodes

- This clause is used to create nodes, relationships, and properties.
- **Creating a Single node**
  - CREATE (node_name)
- **Creating Multiple Nodes**
  - CREATE (node1) , (node2)
- **Creating a Node with a Label**
  - CREATE (node:label)
  - CREATE (Messi:player)
- **Creating a Node with Multiple Labels**
  - CREATE (node:label_1:label_2:. . . . label_n)
  - CREATE (Messi:person:player)
- **Create Node with Properties**
  - CREATE (node:label { key1: value, key2: value, ... })
  - CREATE (Messi:player{name: "Lionel Messi", YOB: 1987, POB: "Rosario"})

CREATE Relationships

- **Creating Relationships**
  - CREATE (node1)-[:RelationshipType]->(node2)
  - CREATE (Messi:player{name: "Lionel Messi", YOB: 1987, POB: "Rosario"})
  - CREATE (Arg:Country {name: "Argentina"})
  - CREATE (Messi)-[r:BORN_IN]->(Arg)
- **Creating a Relationship with Label and Properties**
  - CREATE (node1)-[label:Rel_Type {key1:value1, key2:value2, . . . n}]-> (node2)
- **Creating a Complete Path**
  - CREATE p = (Node1 {properties})-[:Relationship_Type]-> (Node2 {properties})[:Relationship_Type]->(Node3 {properties}) RETURN p

MATCH

- The MATCH clause introduces the pattern we are looking for.
- Node patterns can contain labels and properties.

```
MATCH (e:Employee)  RETURN e.name, e.surname
```

- Finding nodes by relationships

```
MATCH (e:Employee)-->(cc:CostCenter) RETURN *
```

- * Returns all named nodes, relationships and identifiers

MERGE

- MERGE command is a **combination** of CREATE command and MATCH command.
- It searches for a given pattern in the graph. If it **exists**, then it returns the results.
- If it does **NOT exist** in the graph, then it creates a new node/relationship and returns the results.
- Syntax: MERGE (node: label {properties . . . . })
  - MERGE (node:label) RETURN node
- OnCreate and OnMatch
  - MERGE (node:label {properties . . . .})
  - ON CREATE SET property.isCreated ="true"
  - ON MATCH SET property.isFound ="true"

Filtering properties

- we can get all the employees that have a relation with a specific cost center, for example CC1.
- Properties are expressed within curly brackets.

```
MATCH (e:Employee) --> (:CostCenter {code: 'CC1' })
RETURN e
```

- Here, as you can notice, we omitted the cc variable (we don’t need it!).

Matching by the Relationship

- Relationship expressions must be specified in square brackets.
- To filter the employees who belong to a specific cost center, we have to specify the relationship type:

```
MATCH (n) -[:BELONGS_TO]-> (:CostCenter { code: 'CC1' } )
RETURN n
```

OPTIONAL MATCH

- Search for the pattern described in it, while using nulls for missing parts of the pattern.
- OPTIONAL MATCH is similar to the match clause, the only difference being it returns null as a result of the missing parts of the pattern.

```
MATCH (a:Movie { title: 'Cloud Atlas' })
OPTIONAL MATCH (a)-->(x)
RETURN x
```

- Returns **null**, since the node has no outgoing relationships.

WHERE

- Use a predicate to filter. Note that WHERE is always part of a MATCH, OPTIONAL MATCH or WITH clause.
- Putting it after a different clause in a query will alter what it does.

```
WHERE n.property <> $value
```

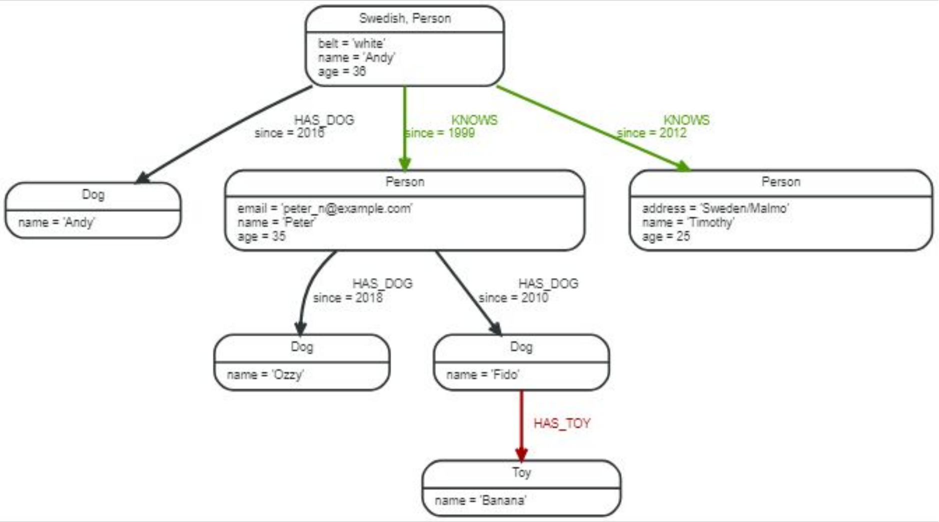
```
MATCH (e:Employee) --> (cc:CostCenter)
WHERE cc.code='CC1`
RETURN e
```

WHERE

- Filter on node label

```
Match(n)
WHERE n:Swedish
RETURN n.name, n.age
```

- Filter on a relationship property



```
MATCH (n:Person)-[k:KNOWS]->(f)
WHERE k.since < 2000
RETURN f.name, f.age, f.email
```

DISTINCT (Unique results)

- DISTINCT retrieves only unique rows depending on the columns that have been selected to output.

```
  RETURN distinct expr
```

- Example:

```
MATCH (a { name: 'A' })-->(b)
RETURN DISTINCT b
```

- The node named "B" is returned by the query, but only once.

Paging results – LIMIT and SKIP

- Paging is necessary to avoid a lot of data from being loaded all together in a single query.

```
MATCH (b:Book)
WHERE b.title =~ '(?i).*drama.*'
RETURN b
LIMIT 20

MATCH (b:Book)
WHERE b.title =~ '(?i).*drama.*`
RETURN b
SKIP 20
LIMIT 20
```

Sorting (Order By)

- Sorting with Cypher is exactly the same as sorting with SQL using (ORDER BY).

```
MATCH (b:Book) RETURN b.title
ORDER BY b.title
LIMIT 5
```

- A descending sort: By default sort is ascending, you can reverse the order using DESC clause.

```
MATCH (b:Book)
RETURN b.title
ORDER BY b.title DESC
```

COUNT

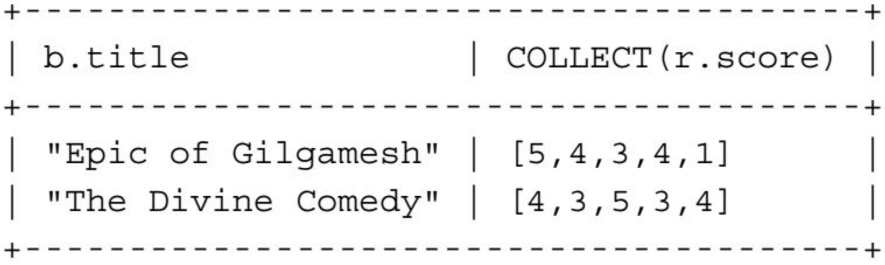
- count(*), counts the number of matching rows.
- count(variable), counts the number of non-null values.
- count(DISTINCT variable), count can take the DISTINCT operator, which removes duplicates from the values.

```
MATCH (b) <-[r:Vote]- (u:User)
RETURN COUNT(*) as votes
```

Collecting values in an array

- You can collect all the values in an array so that you can easily process them with your preferred algorithm.
- For example, the following query returns all the score votes received for books:

```
MATCH (b:Book) <-[r:Votes]- (:User)
RETURN b.title, COLLECT(r.score)
```



```
+--------------------------------------+
| b.title              | COLLECT(r.score) |
+--------------------------------------+
| "Epic of Gilgamesh"  | [5,4,3,4,1]      |
| "The Divine Comedy"  | [4,3,5,3,4]      |
+--------------------------------------+
```

WITH (Separating query parts)

- The WITH syntax is similar to RETURN.
- It separates query parts explicitly, allowing you to declare which variables to carry over to the next part.
- One common usage of WITH is to **limit** the number of entries that are then **passed on** to other MATCH clauses.

```
MATCH (user)-[:FRIEND]-(friend)
WHERE user.name = $name
WITH user, count(friend) AS friends
WHERE friends > 10
RETURN user
```

Cypher Docs. & Cheat Sheet

- https://neo4j.com/docs/cypher-manual/3.5/introduction/
- https://neo4j.com/docs/cypher-refcard/current/
- https://neo4j.com/docs/cypher-manual/current/

Funny Pictures on www.LeFunny.net

Now, It’s time to say …

THANK YOU

SEE YOU NEXT TIME!