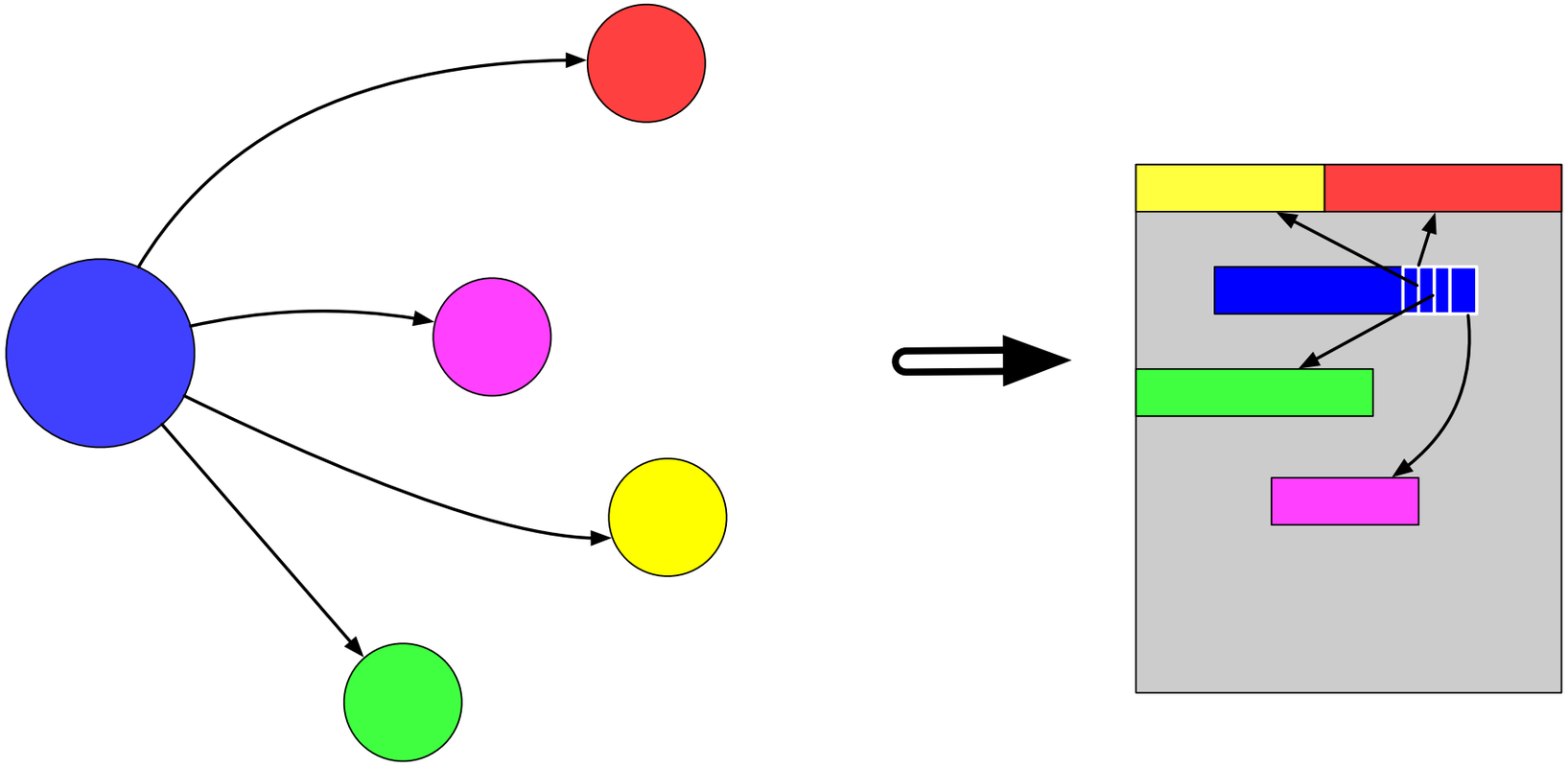
Neo4j

Neo4j

- Neo4j Graph Database
  - Store nodes together with adjacencies as pointers
  - Edge chasing is quick

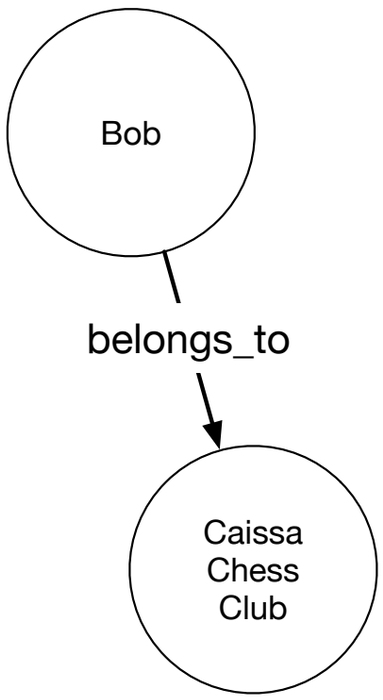


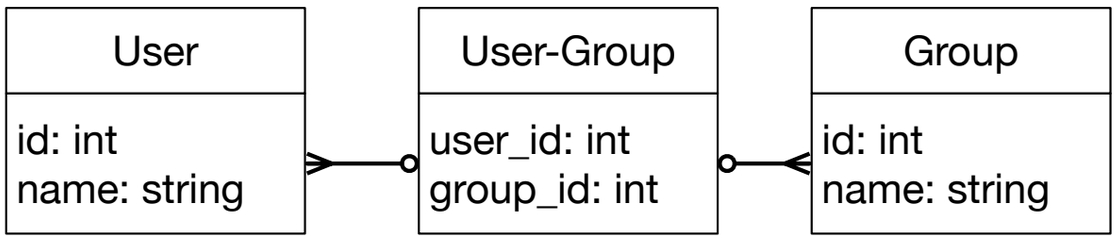
Neo4j

- Implements ACID transaction model
- Is schema-less
  - Nodes can have any property
  - Nodes can have any type of relation to another node
- Query language “Cipher” does matching

Neo4j

- Modeling with diagrams can be simpler





Neo4j

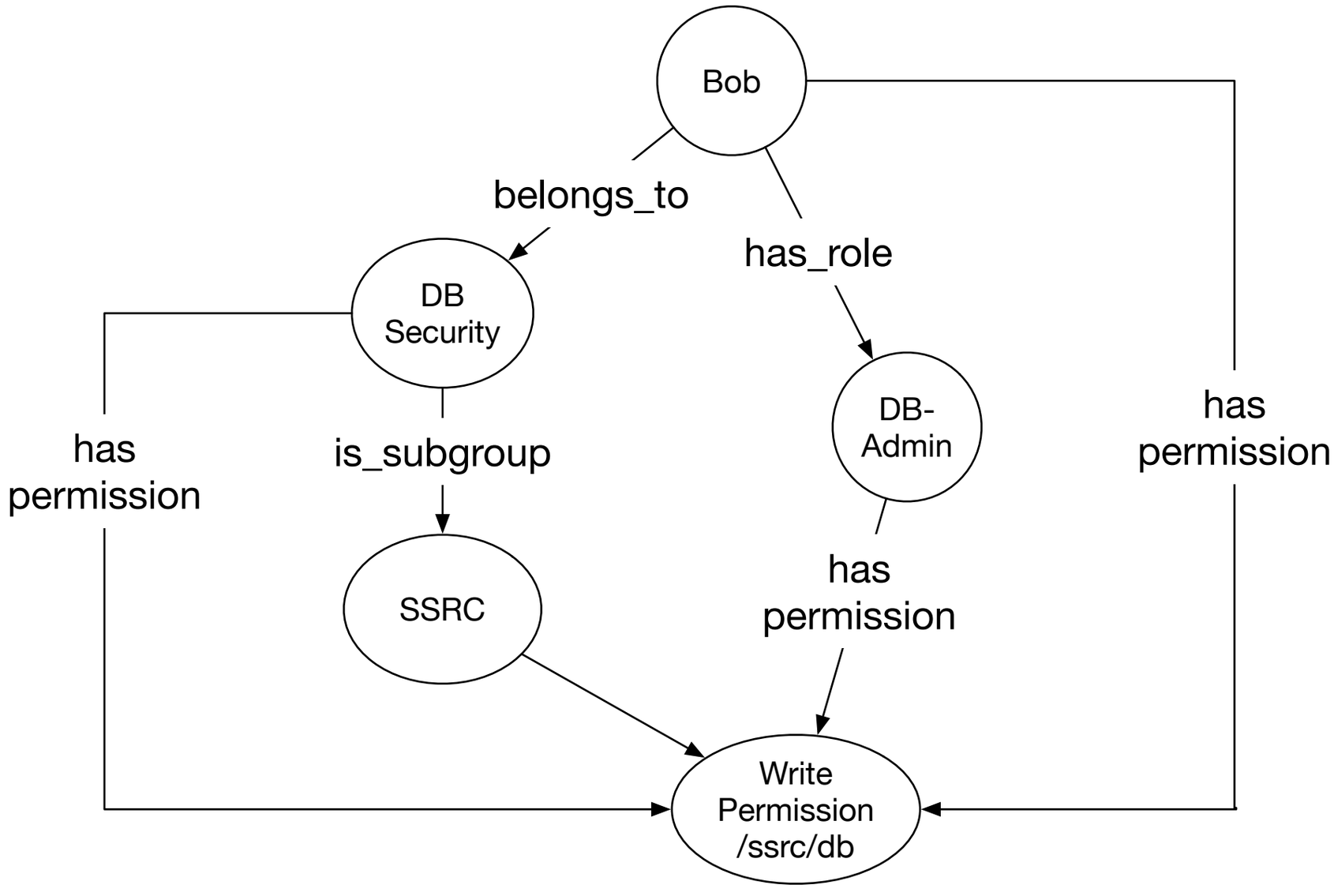
- Cypher uses ASCII-art for queries

```
MATCH (p)-[:belongs_to]->(g)
WHERE p.name = "Bob"
RETURN g.name
```

```
MATCH (p)-[:belongs_to]->(g)
WHERE g.name = "Caissa Chess Club"
RETURN p.name
```

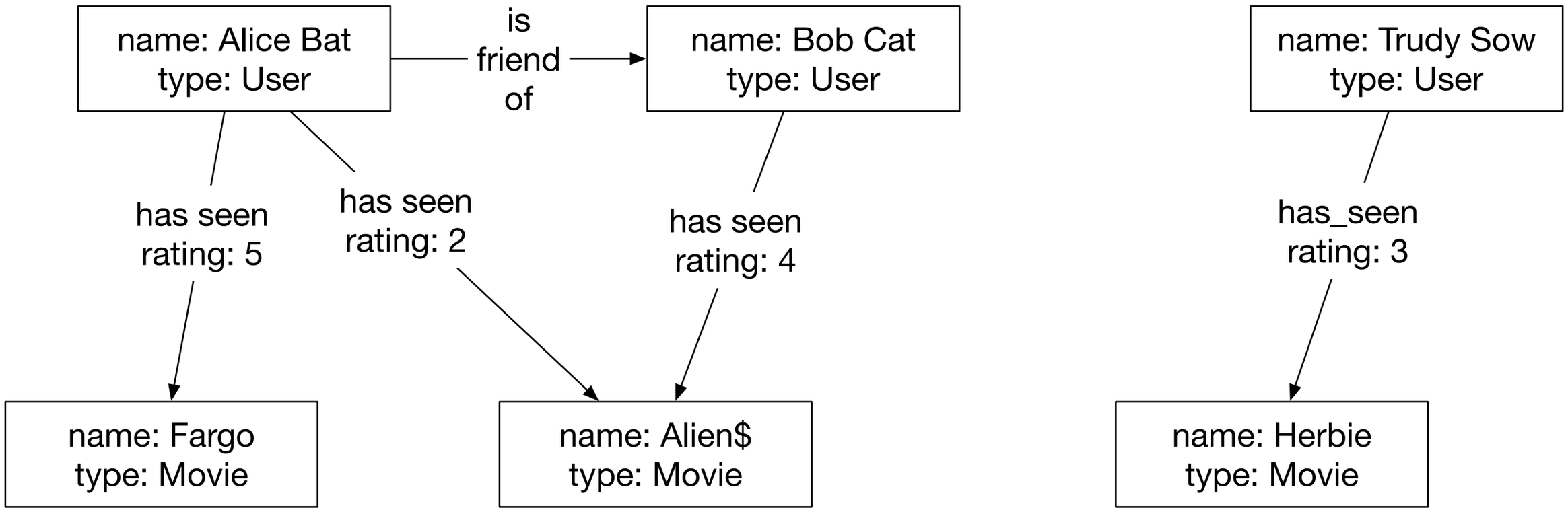
Neo4j

- More complicated relations are harder to represent in relational tables



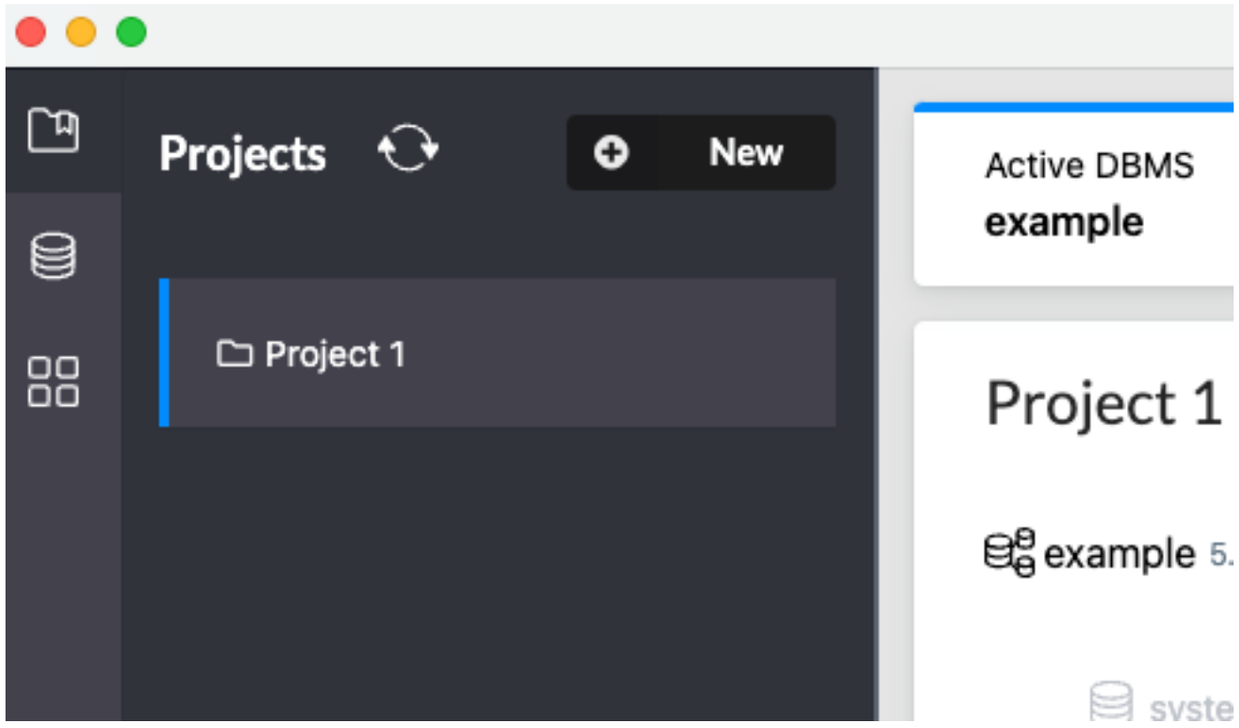
Neo4j

- Nodes and relationship can have many properties



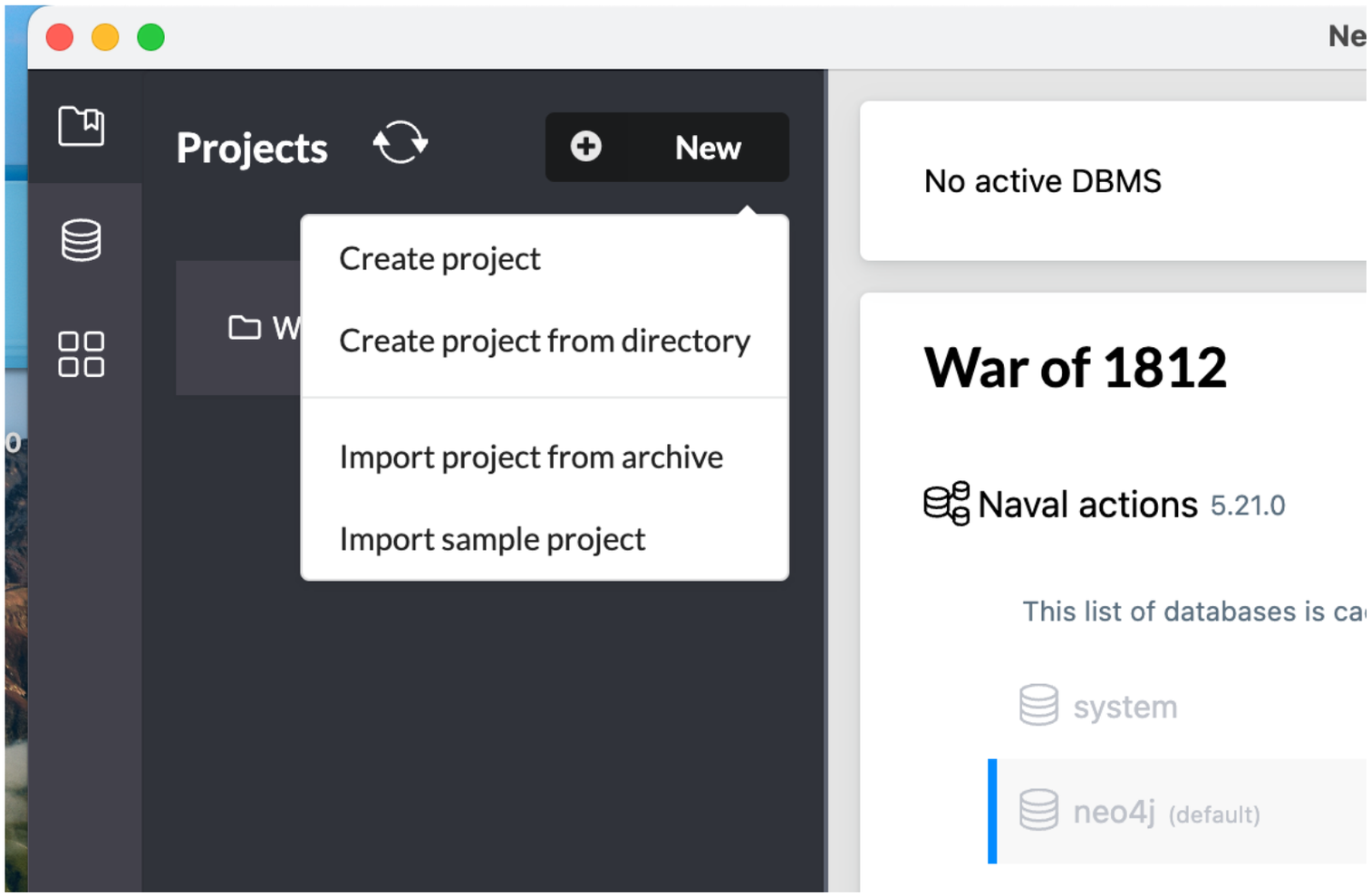
Neo4j

- Create a project using Neo4j Desktop
    - Make sure to **stop** a currently running DBMS
        - Use the **stop** button
- On the left sidebar, click new
- This creates a project named "Project"



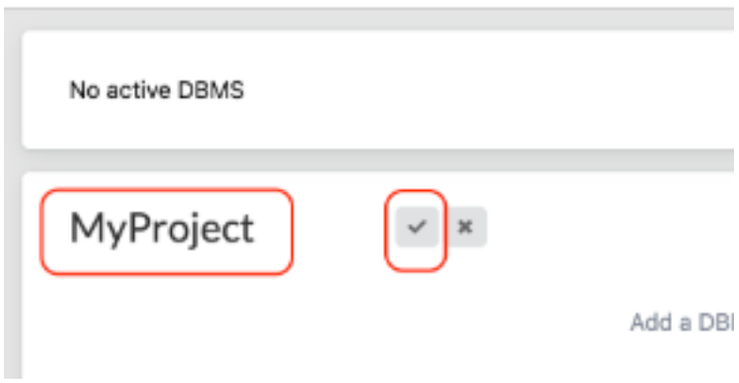
Neo4j

- Click on Create project



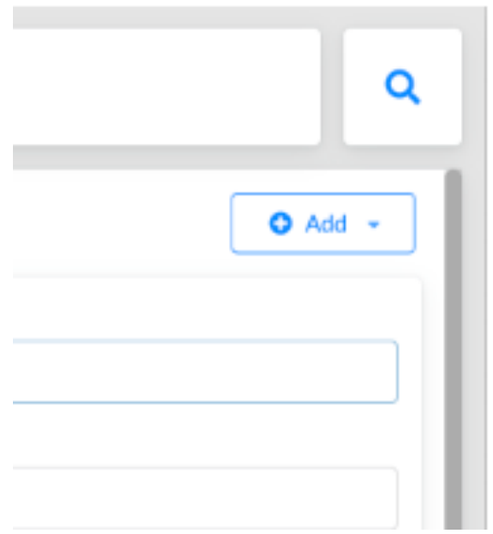
Neo4j

- Next to the name, you can edit
  - Hover your mouse so that you can see the button
- Change the name and then click the check mark



Neo4j

- You need to create a database in the project
  - Click the add button on the right
  - You can select the Neo4j version
    - Which might trigger an update



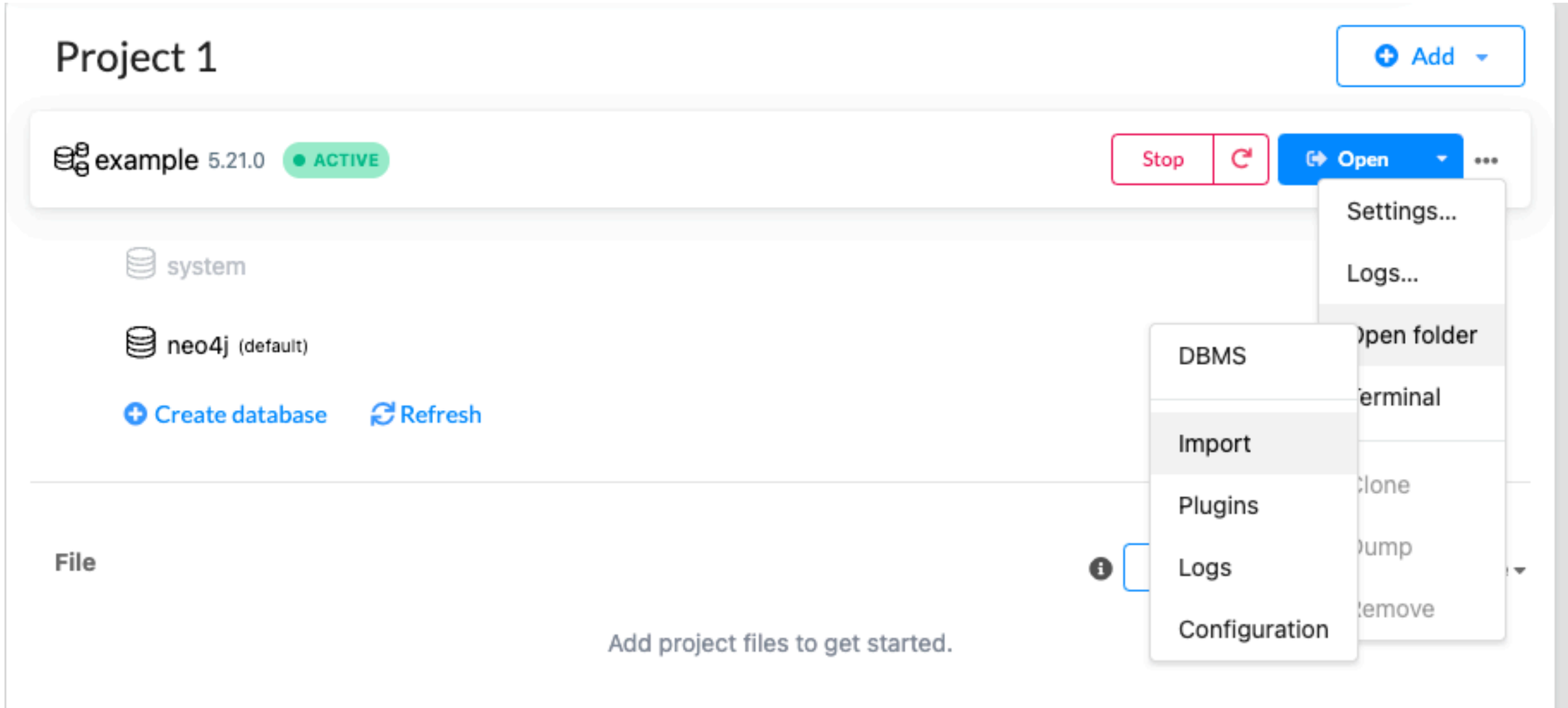
- You need to select a password with >= 8 characters
- Hit the create button
- You can start and stop a database by hovering to the right of the database name

Neo4j

- Importing data with csv
  - Download from
    - https://s3.amazonaws.com/dev.assets.neo4j.com/wp-content/uploads/desktop-csv-import.zip
- OR
  - Go to the tutorial
    - "how to import csv file in neo4j desktop"

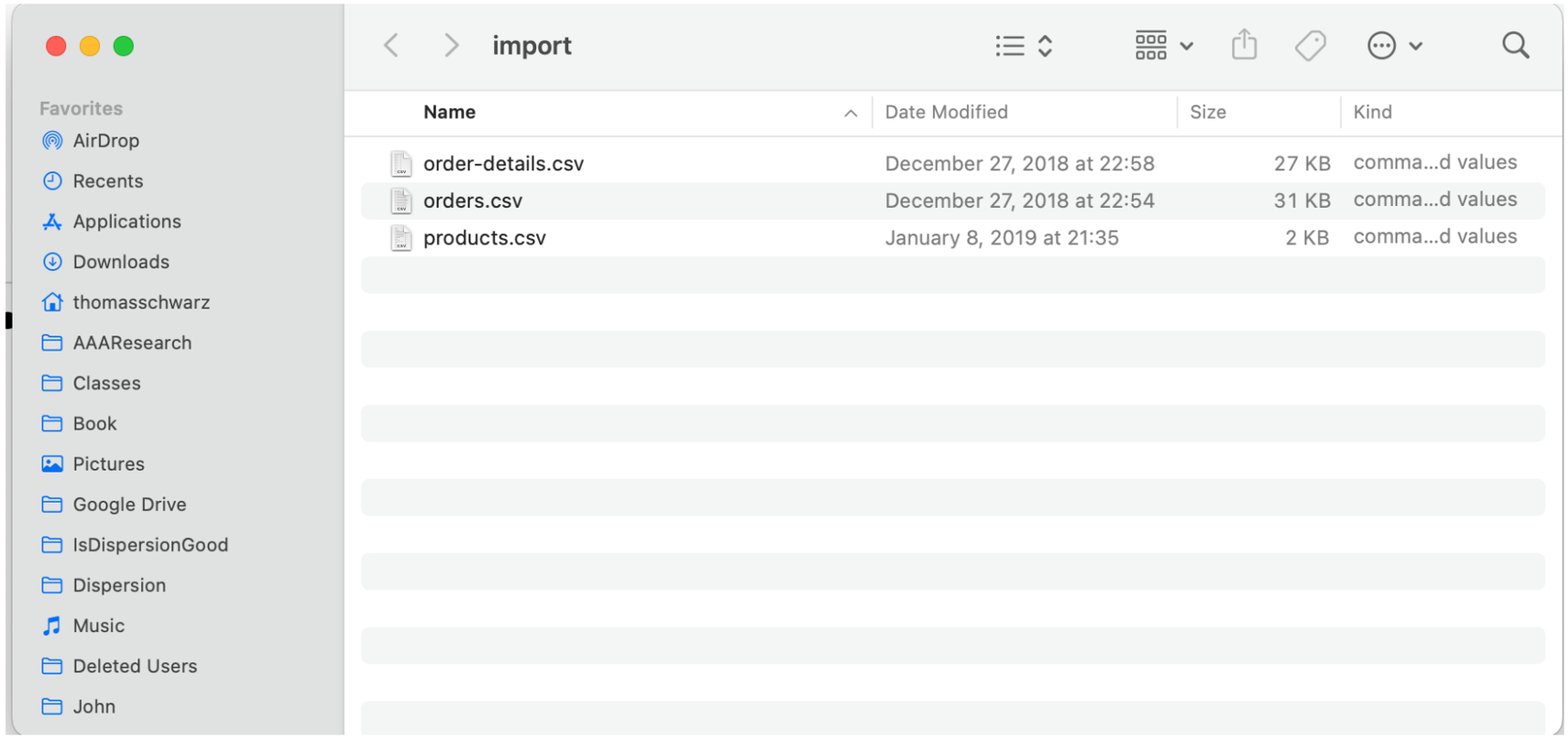
Neo4j

- You need to place the downloaded csv files in an import directory
- To the right of the DBMS, there are three dots for options



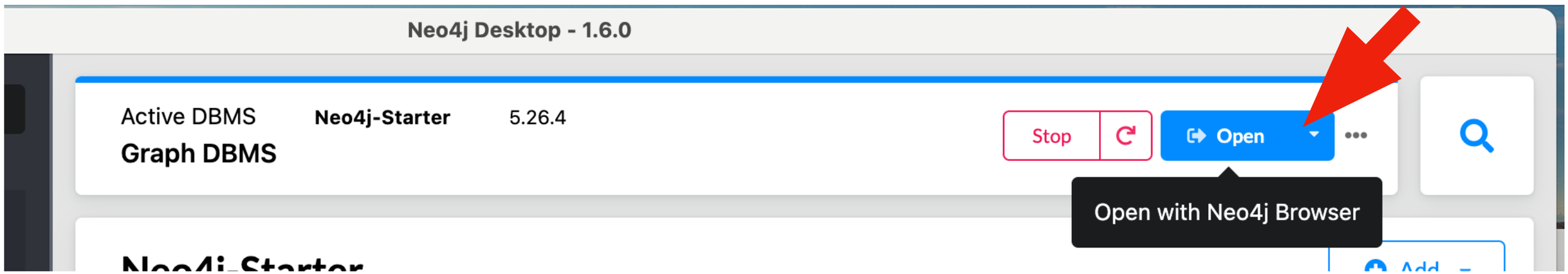
Neo4j

- Click on the "import" menu item
  - This should open up a directory viewer
  - Copy or move your csv files there



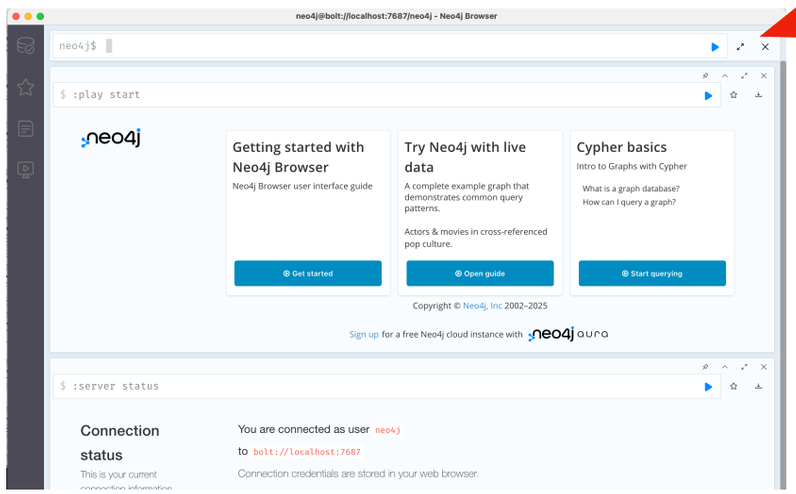
Neo4j

- Start the Neo4j Browser
  -



Neo4j

- At the very top of the **browser**, there is a window for commands



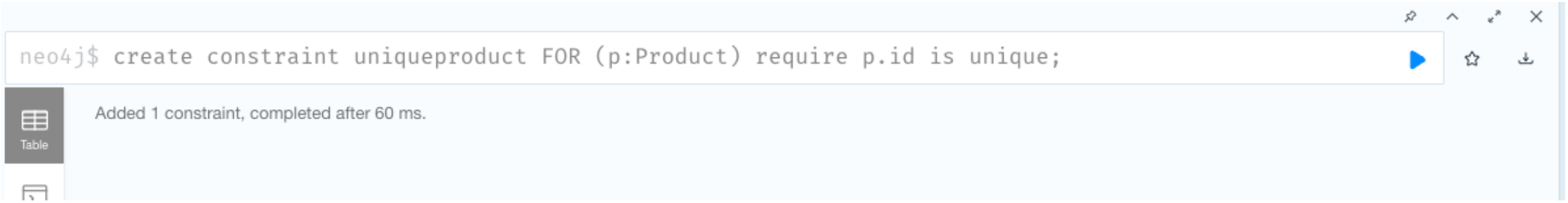
- Inside this window on the right is the "execute" / "play" button
- When it opens, most of the real-estate is used up by helpful links

Neo4j

- We can look at csv-file contents with LOAD CSV
  - This will **not** import data.
  - In the window on top, type in (or copy from the tutorial)
    - `LOAD CSV FROM 'file:///products.csv' AS row RETURN row LIMIT 10;`
  - Notice the three forward slashes
  - This will display the first 10 rows
    - On the left, you can select the visualization: Table, Text, Code

Constraints

- You can use constraints to filter out bad data
- Create a constraint "uniqueproduct"



- `create constraint uniqueproduct FOR (p:Product) require p.id is unique;`
- Do the same for orders:
  - `create constraint uniqueorder FOR (o:Order) require o.id is unique;`

Conversions

- When importing data, we need to convert strings to other data-types
  - `toInteger`
  - `toFloat`
  - `datetime` or `date`
  - `toString`

Conversions

- You can try out the result of conversions on csv data:
  - ```
    LOAD CSV FROM 'file:///products.csv' AS row
    WITH toInteger(row[0]) AS productId,
    row[1] AS productName,
    toFloat(row[2]) AS unitCost
    RETURN productId, productName, unitCost
    LIMIT 3;
    ```
    ```

Conversions

- For orders, some string surgery is needed:
  - The csv file has data times with a space
  - Neo4j needs a T instead:
    - 1996-07-04 00:00:00.000 —> 1996-07-04 T00:00:00.000

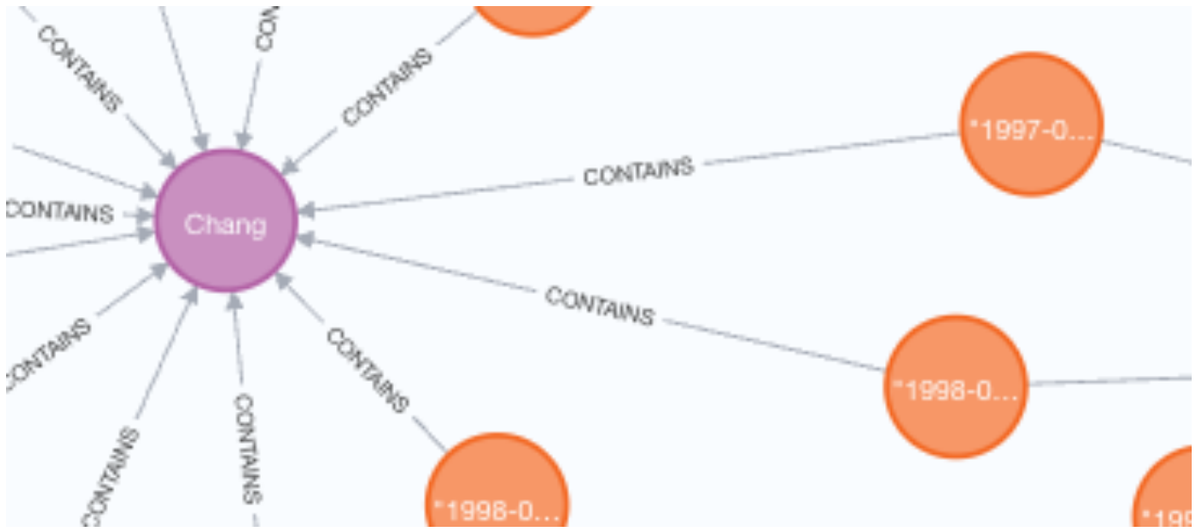
```
LOAD CSV WITH HEADERS FROM 'file:///orders.csv'
AS row
WITH toInteger(row.orderID) AS orderId,
datetime(replace(row.orderDate,' ','T')) AS orderDate,
row.shipCountry AS country
RETURN orderId, orderDate, country
LIMIT 5;
```

Conversions

```
LOAD CSV WITH HEADERS FROM 'file:///order-details.csv'
AS row
WITH toInteger(row.productID) AS productId,
toInteger(row.orderID) AS orderId,
toInteger(row.quantity) AS quantityOrdered
RETURN productId, orderId, quantityOrdered
LIMIT 8;
```

Database Design

- Datamodel:
  - Orders (orange) and products (violet) are nodes
  - Order-details gives the relationship



Creating Data

- Now we are ready to create the elements in the data-base
- We use Merge because a constraint violation will not cause the rest of the operation to abort

```
LOAD CSV FROM 'file:///products.csv' AS row
WITH toInteger(row[0]) AS productId, row[1] AS
productName, toFloat(row[2]) AS unitCost
MERGE (p:Product {productId: productId})
SET p.productName = productName,
p.unitCost = unitCost
RETURN count(p);
```

Creating Data

```
LOAD CSV WITH HEADERS FROM 'file:///orders.csv' AS row
WITH toInteger(row.orderID) AS orderId,
datetime(replace(row.orderDate,' ','T')) AS orderDate,
row.shipCountry AS country
MERGE (o:Order {orderId: orderId})
SET o.orderDateTime = orderDate,
o.shipCountry = country
RETURN count(o);
```

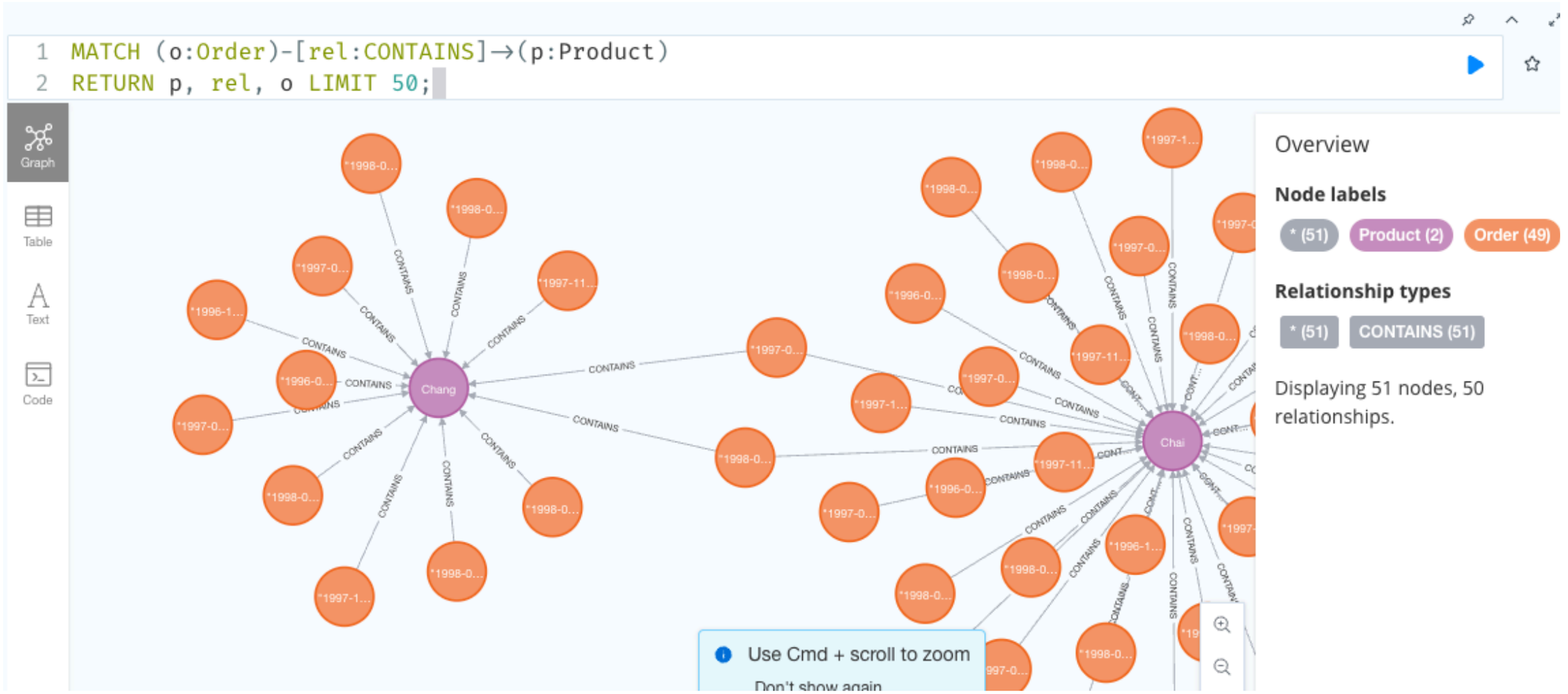
Creating Data

```
LOAD CSV WITH HEADERS FROM 'file:///order-
details.csv' AS row

WITH row
MATCH (p:Product {productId:
toInteger(row.productID)})
MATCH (o:Order {orderId: toInteger(row.orderID)})
MERGE (o)-[rel:CONTAINS {quantityOrdered:
toInteger(row.quantity)}]->(p)
```

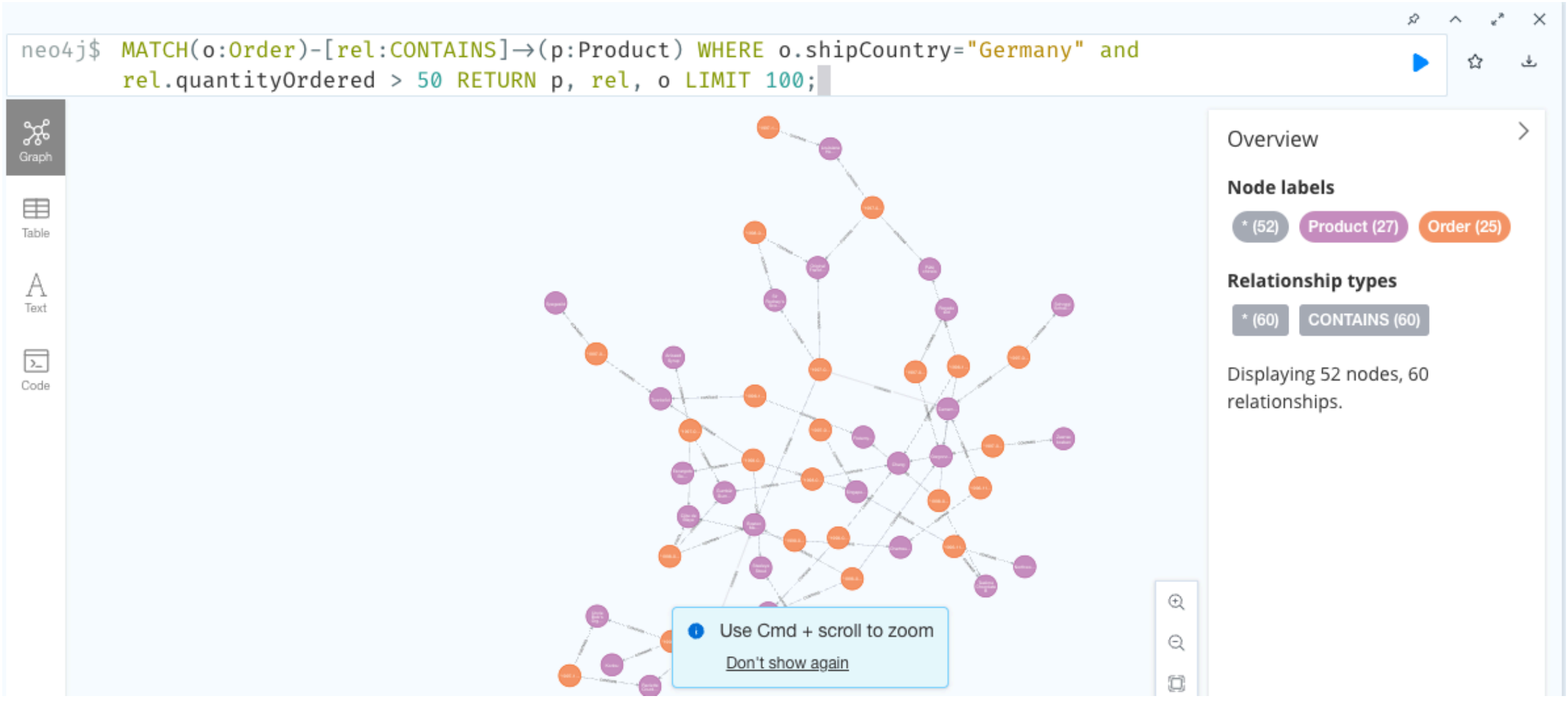
Checking

- Create a generic match

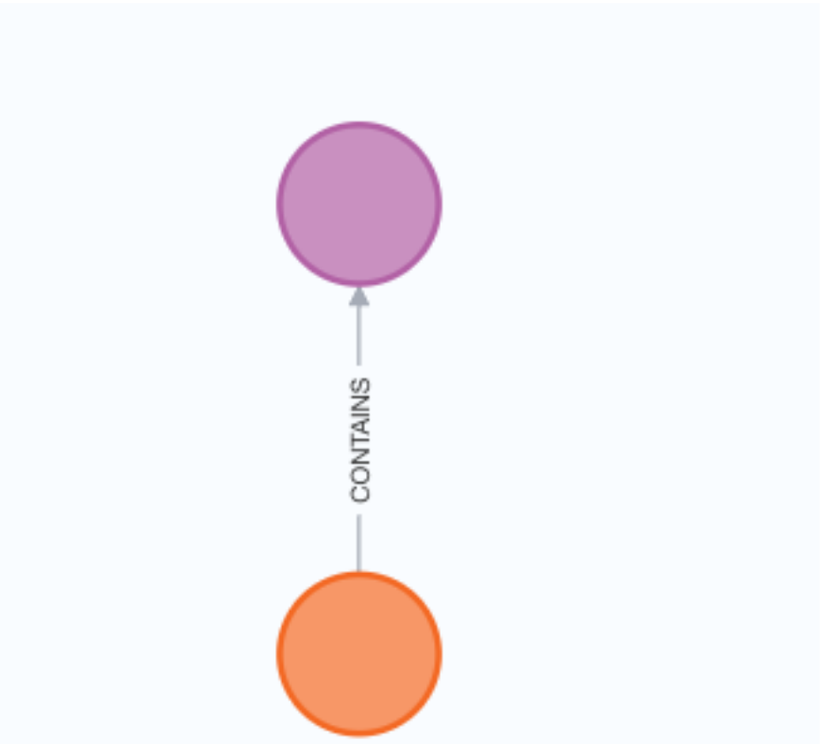


Checking

- Create a specific match



Checking



- You can look at our primitive model with
  - `call db.schema.visualization`

nodes	relationships
[(:Order {name: "Order",indexes: [],constraints: ["Constraint(id=6, name='uniqueorder', type='UNIQUENESS', schema=(:Order {id}), ownedIndex=5)"]}), (:Product {name: "Product",indexes: [],constraints: ["Constraint(id=4, name='uniqueproduct', type='UNIQUENESS', schema=(:Product {id}), ownedIndex=3)"]})]	[[:CONTAINS {name: "CONTAINS"}]]

Cypher

- Selection and Projection:
  - Match
  -
```
MATCH (p:Product)
RETURN p.productName, p.unitCost
ORDER BY p.unitCost DESC
LIMIT 10;
```

Cypher

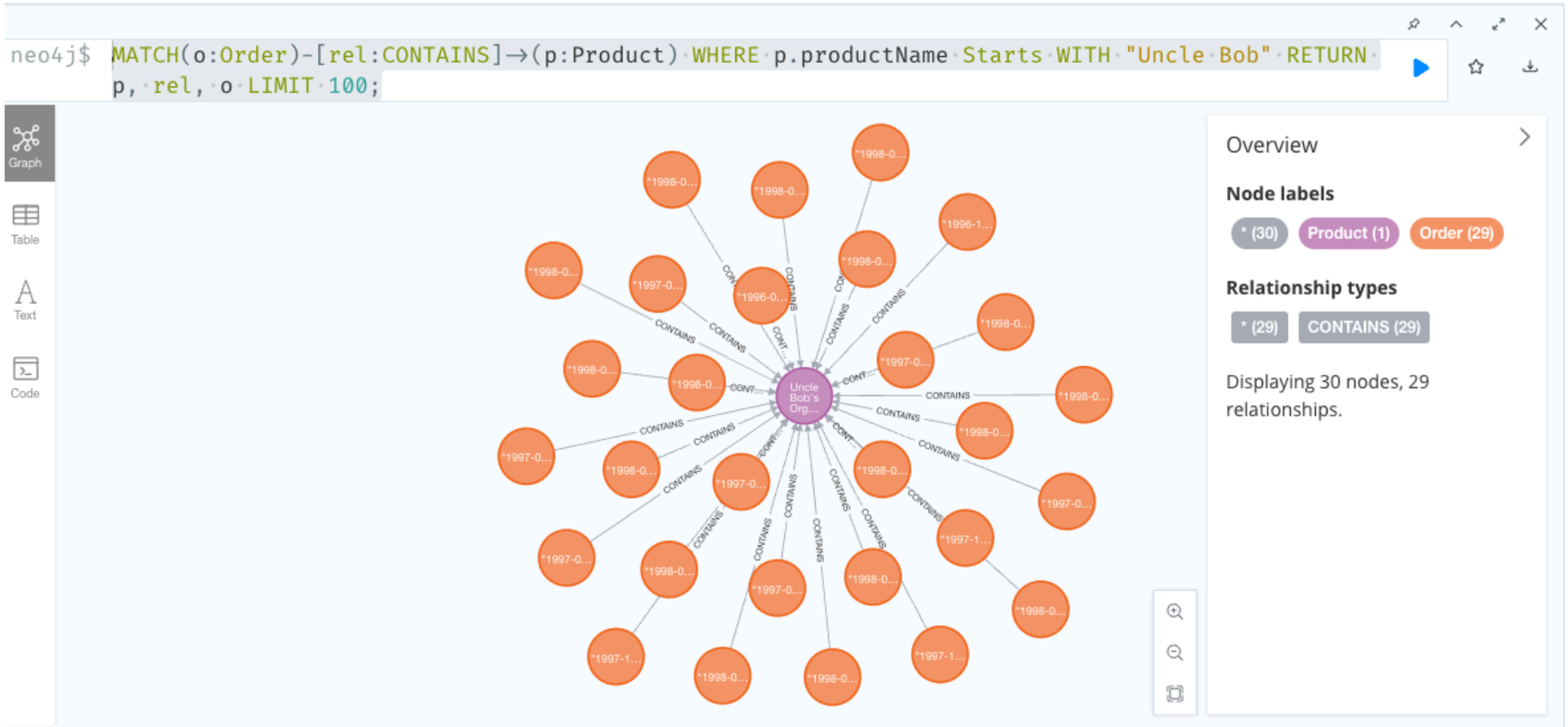
- Use a WHERE clause for properties

```
MATCH(o:Order)-[rel:CONTAINS]->(p:Product)
WHERE o.shipCountry="Germany" AND
rel.quantityOrdered > 50
RETURN p, rel, o
LIMIT 100;
```

Cypher

- Can use String functions:

```
MATCH(o:Order)-[rel:CONTAINS]->(p:Product) WHERE
p.productName Starts WITH "Uncle Bob"
RETURN p, rel, o
LIMIT 100;
```



Cypher

- Creating Nodes
  - `Create (myProduct:Product{productId: 543210, productName: "California Raisins", unitCost: 2.35})`
  - `Match (p:Product) WHERE p.productName STARTS WITH "California" RETURN p;`

Cypher

- ```
  CREATE(myneworder:Order
  {orderDateTime:
  datetime("2024-07-31T12:13:00.000"),
  orderID: 555550,
  shipCountry: "India"} )
  ```
  ```

Cypher

- Relationships:
  - Need to find the nodes first with Match
  - Then create a relationship between nodes

```
MATCH (rosie:Product{productName: "California
Raisins"}), (oscar:Order{orderId:555550}) CREATE
(oscar)-[:CONTAINS{quantityOrdered:20}]->(rosie);
```

Cypher

- Check existence

```
Match (o:Order)-[r:CONTAINS]->(p:Product)
WHERE p.productName STARTS WITH "California"
RETURN o, p, r;
```

Cypher

- Queries:
  - Nodes have round brackets ( )
  - Relationships have [ ]

```
Match ()-[r:CONTAINS]->()
WHERE r.quantityOrdered > 100
RETURN r;
```

Cypher

- 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 that additionally allows you to specify what happens if the data was matched or created."

Aggregates

- Aggregates aggregate values:
  - SUM
  - MAX
  - COUNT
  - MIN

Aggregates

- Finding the product with highest quantity ordered
  - Two stages:
    - Find maximum

```
MATCH (:Order)-[r:CONTAINS]->(p:Product) WITH
max(r.quantityOrdered) as maximum RETURN maximum
```

    - Find products

Aggregates

- Need to use COLLECT in order to find all products where the maximum is reached

```
MATCH (o:Order)-[r:CONTAINS]->(p:Product)
WITH max(r.quantityOrdered) AS mm
MATCH (o1:Order)-[r:CONTAINS]->(p1:Product)
WHERE r.quantityOrdered = mm
RETURN COLLECT(p1.productName)
```

Aggregates

- Find the quantities of orders for products

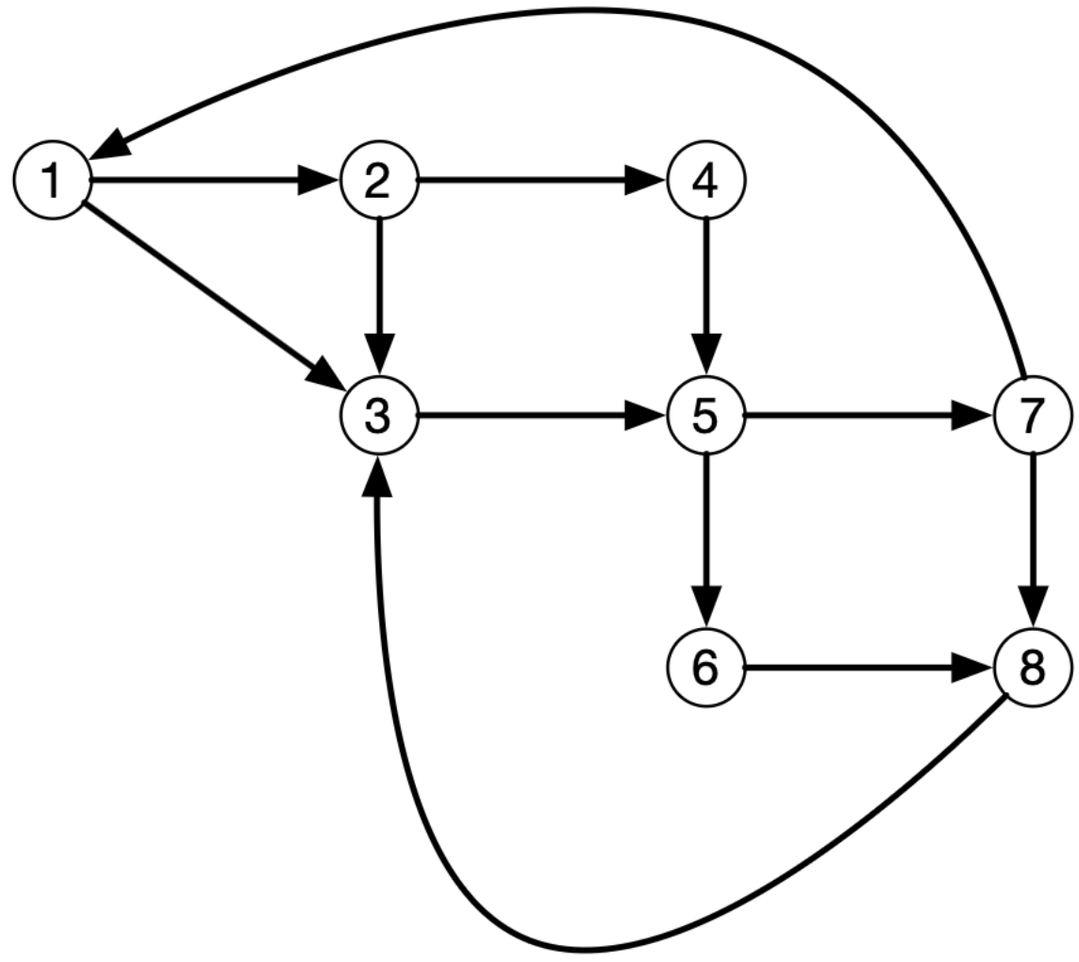
```
MATCH (o:Order)-[r:CONTAINS]->(p:Product)
RETURN p.productName, SUM(r.quantityOrdered)
```

Graph Distance

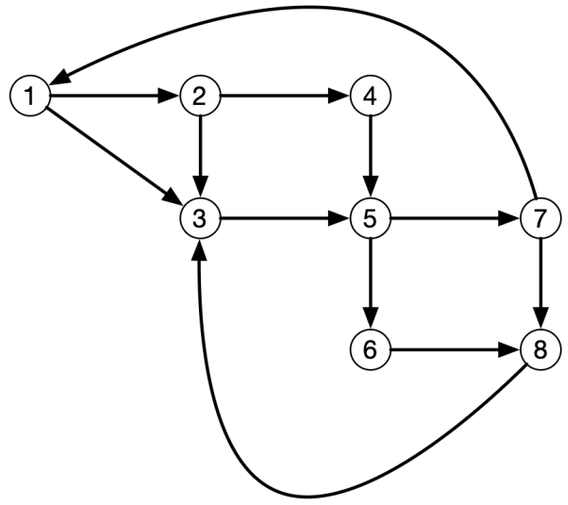
- Create nodes:

```
CREATE (n: Label {myID:1})
```

- Create node with id 5 twice



Graph Distanc



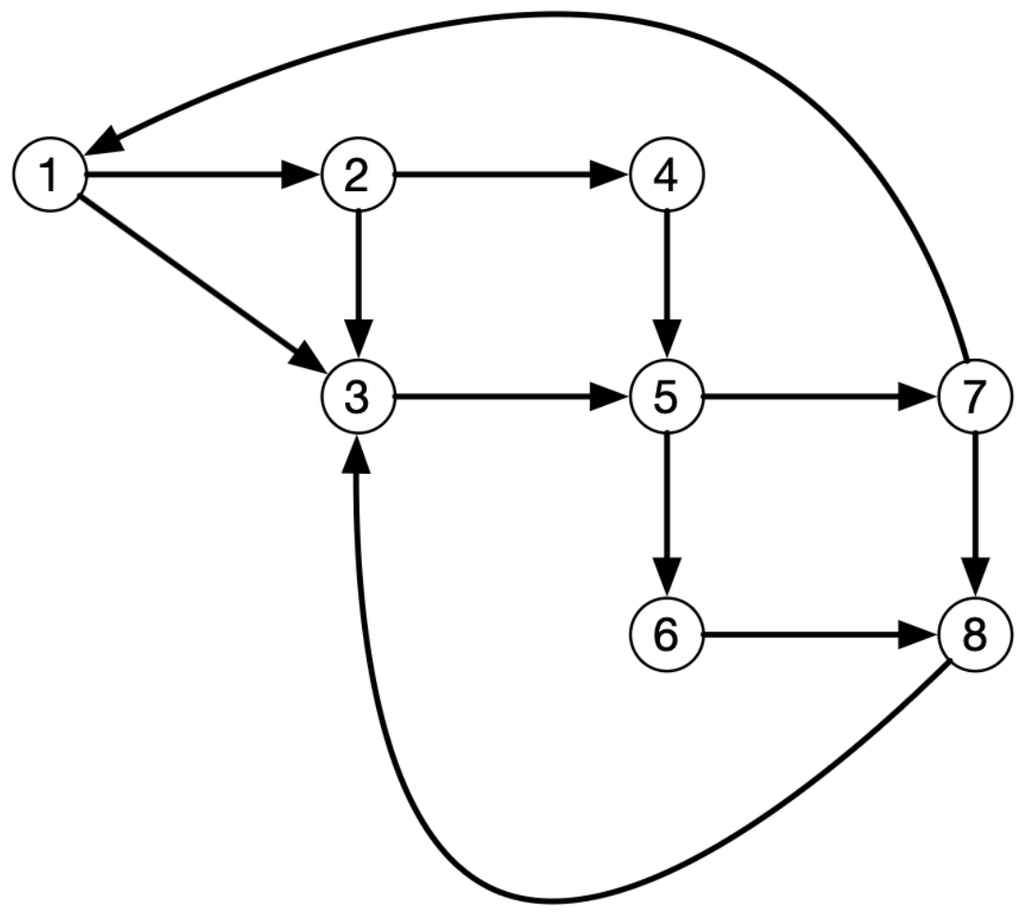
- Can use a loop with FOREACH

```
WITH [2,3,4,5,6,7,8] AS identifiers
FOREACH ( value IN identifiers | CREATE (:Label {myID : value}) );
```

Graph Distance

- Create edges: First Match left, then right node:

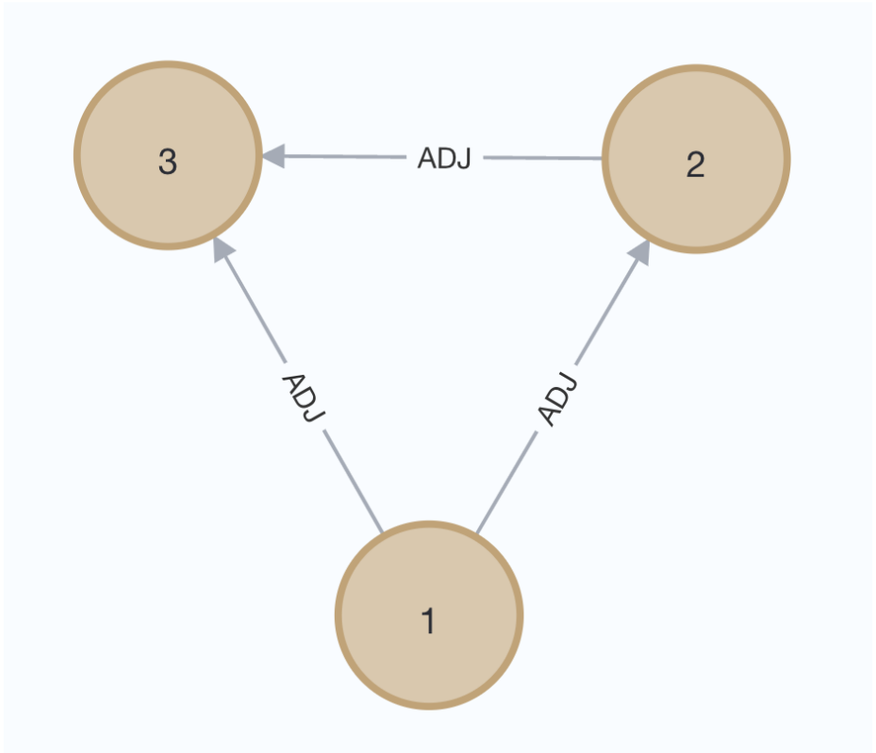
```
MATCH (a {myID:1}) MATCH (b {myID:2})
CREATE (a)-[:ADJ]->(b)
```



Graph Distance

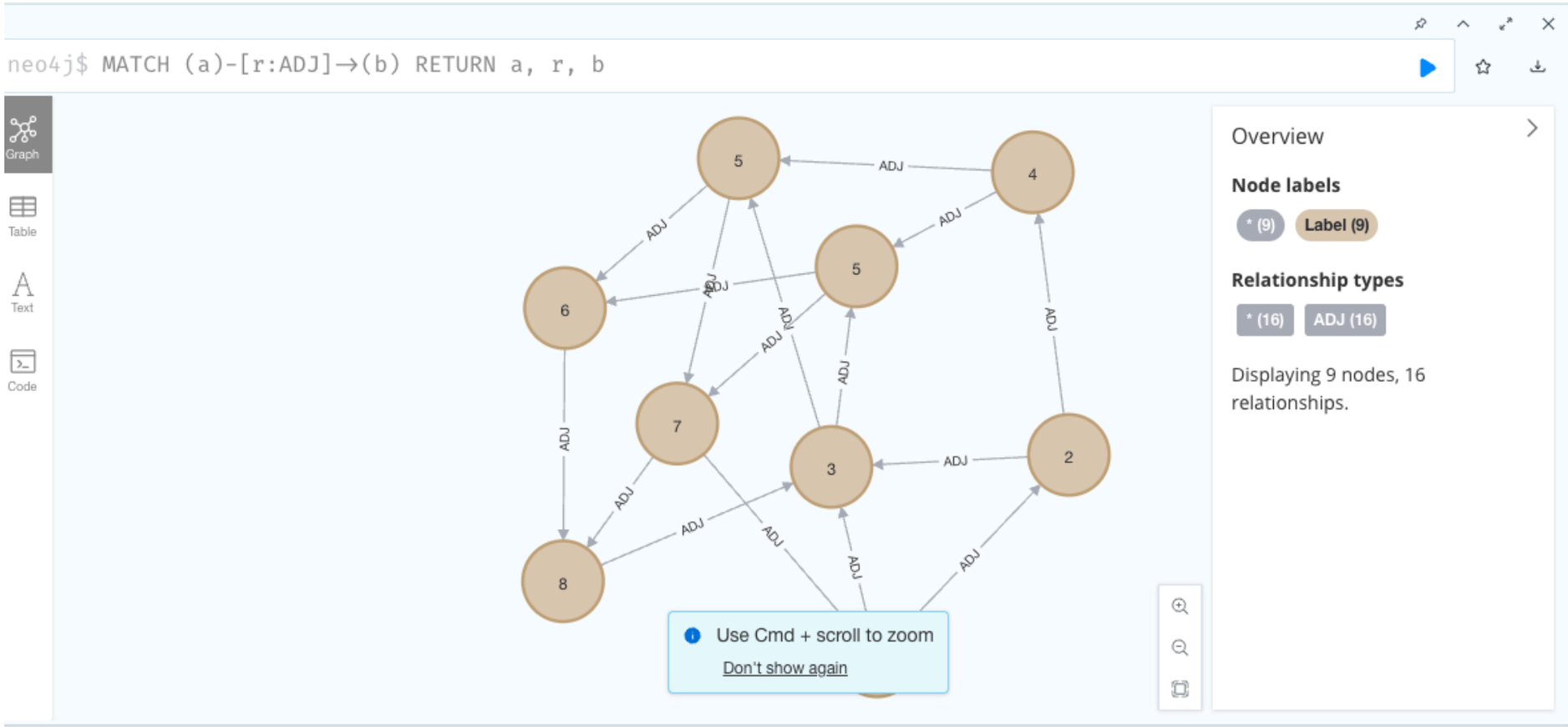
- To check your progress:

```
Match (a)-[r]->(b)
Return a, r, b;
```



Graph Distance

- Can only deletes nodes without relationship
  - We need to delete the nodes with myID 5



Graph Distance

- Delete with delete

```
MATCH (n:Label {myID: 3})-[r:ADJ]->(m: Label{myID: 5})
DELETE r;
```

- After deleting all relationships:

```
Match(n: Label {myID: 5})
DELETE n
```

Graph Distance

- Using FOREACH is tricky
  - Need to use Merge instead of Match
- We use lists for tuples:

```
WITH [[2,4], [3,5], [4,5], [5,7]] as adjacencies
FOREACH (
  pair IN adjacencies |
  MERGE (a {myID: pair[0]})
  MERGE (b {myID:pair[1]})
  CREATE (a)-[:ADJ]->(b)
);
```

Graph Distance

- Reachable in two hops from 1

```
Match ({myID:1})-[:ADJ]->()-[:ADJ]->(non :Label)
RETURN non.myID
```

Graph Distance

- Reachable in two hops from 1, but not in one
  - Use * to indicate number of hops
  - Use - to indicate bi-direction

```
MATCH (one:Label {myID:1})-[:ADJ*2]->(non)
WHERE NOT (one)-[:ADJ]-(non)
RETURN non.myID
```

Graph Distance

- Better use collect

```
MATCH (one:Label {myID:1})-[:ADJ*]->(non) RETURN
Collect(non.myID)
```

- And even better: distinct

```
MATCH (one:Label {myID:1})-[:ADJ*]->(non)
RETURN Collect(distinct non.myID)
```

Group Quiz

- Create a list of all node labels in the graph.

Solution

```
MATCH (node) RETURN COLLECT(DISTINCT node.myID);
```

Netflix Database

- Download and unzip the netflix dataset from Kaggle: netflix_titles.csv
- Create a new Neo4j project
- Place the csv file into the import folder

Netflix Database

- Now we relearn how to import data
  - We can use the csv headers to access values

```
LOAD CSV WITH HEADERS
FROM 'file:///netflix_titles.csv' AS line
CREATE (
    :Movie {
         id: line.show_id,
         title: line.title,
         releaseYear: line.release_year
    }
    )
```

Netflix Database

- Comma separated lists

```
WITH "United States, India, France"
   AS countries_as_string
WITH split(countries_as_string, ",") AS
   countries_as_list
UNWIND countries_as_list AS country_name
RETURN trim(country_name)
```

- UNWIND: Do something for every item in a list

Netflix Database

- We need to parse the director's list when importing

```
LOAD CSV WITH HEADERS FROM
    'file:///netflix_titles.csv' AS line
WITH split(line.director, ",") AS directors_list
UNWIND directors_list AS director_name
MERGE (:Person {name: trim(director_name)});
```

- MERGE: create when the node does not exist

Netflix Database

- MERGE can be problematic because we match exactly
- We can specify what we want to do depending on whether we are modifying or creating a new node
- ```
  MERGE (p:Person {name: "Bob"})
  ON CREATE SET p.surname = "Cat"
  ON MATCH SET p.birthDate = "1969-01-09"
  ```
  ```

Netflix Database

- We delete everything and put everything into a single statement

```
LOAD CSV WITH HEADERS FROM
   'file:///netflix titles.csv' AS line
CREATE (m:Movie
    {id: line.show_id,
     title: line.title,
     releaseYear: line.release_year})
WITH m, split(line.director, ",") as directors_list
UNWIND directors_list AS director_name
MERGE (p:Person {name: director_name})
MERGE (p)-[:Directed]-> (m)
```