CMPINF 2110

Spring 2021

Week 12

Neo4j Northwind example continued

Unique or DISTINCT categories, found with the DISTINCT keyword applied within the RETURN clause

If we just wanted the number of unique values we can apply the count() function!

neo4j$

```
//if we just wanted to count the number of unique values
MATCH (c:Category)
RETURN count(DISTINCT c);
```

Table | Text | Code

	count(DISTINCT c)
1	8

Started streaming 1 records after 5 ms and completed after 6 ms.

neo4j$

```
//but why do we get the same result for category if we just query category label?
MATCH (c:Category)
RETURN c;
```

Displaying 8 nodes, 0 relationships.

Same process to count the number of unique Products

neo4j$

```
//how many unique products are there?
MATCH (p:Product)
RETURN count(DISTINCT p);
```

Table

	count(DISTINCT p)
1	77

Text

Code

Started streaming 1 records after 5 ms and completed after 6 ms.

We get the same number of nodes whether we use distinct or not for Product

neo4j$

```
//same query result for counting Products
MATCH (p:Product)
RETURN count(p);
```

Table

	count(p)
1	77

Text

Code

Started streaming 1 records after 6 ms and completed after 7 ms.

Query the suppliers, products, and categories and the relationships between them

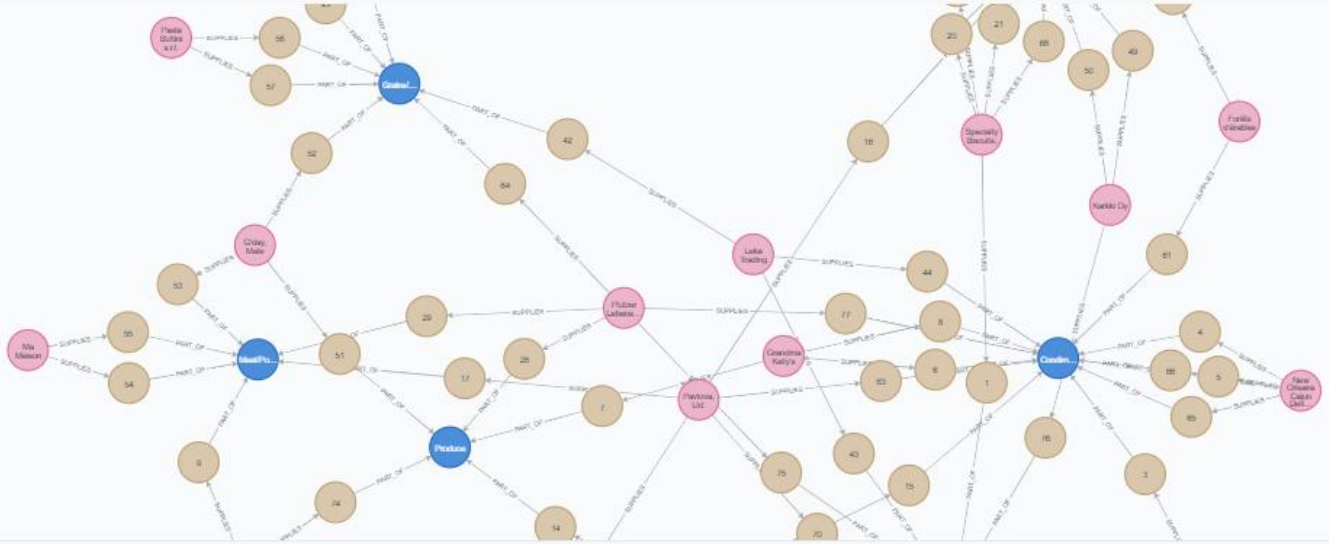
neo4j$

```
//all suppliers, products, and categories
MATCH (s:Supplier)-[r1:SUPPLIES]→(p:Product)-[r2:PART_OF]→(c:Category)
RETURN s, r1, p, r2, c;
```

Graph | Table | Text | Code

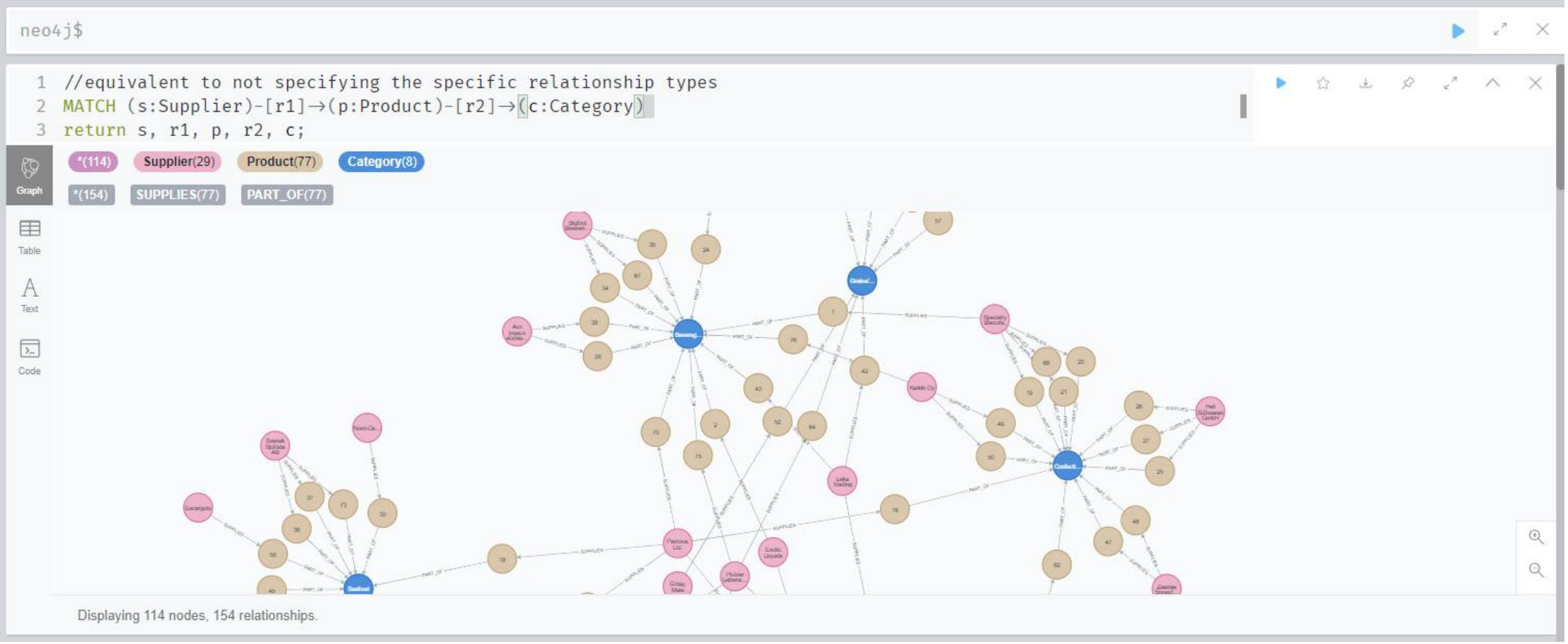
*(114) | Supplier(29) | Product(77) | Category(8)

*(154) | SUPPLIES(77) | PART_OF(77)



Displaying 114 nodes, 154 relationships.

For this example, we could have just said ANY relationship between the LABELS



Focus on a single Supplier

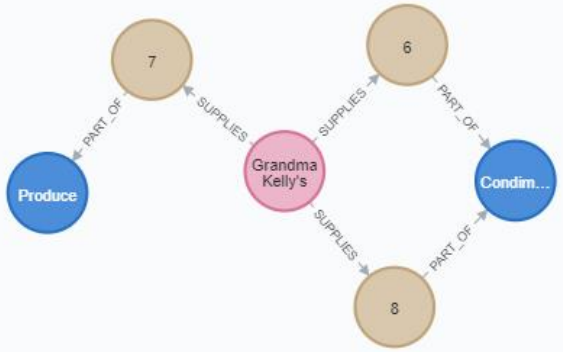
neo4j$

```
//what if we want to focus on a single supplier, supplierID=3
MATCH (s:Supplier {supplierID: '3'})-[r1]→(p:Product)-[r2]→(c:Category)
return s, r1, p, r2, c;
```

Graph | Table | Text | Code

*(6) Supplier(1) Product(3) Category(2)

*(6) SUPPLIES(3) PART_OF(3)



Displaying 6 nodes, 6 relationships.

The WITH clause and IN operator allow us to filter/subset multiple suppliers!

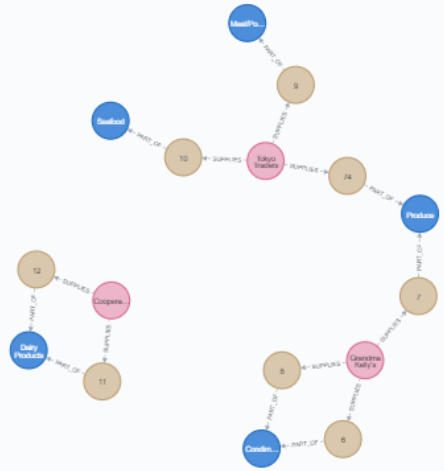
neo4j$

```
//what if we wanted to query 3 suppliers?
//can use the WITH clause and IN operator!
WITH ['3', '4', '5'] as ids
MATCH (s:Supplier)-[r1]→(p:Product)-[r2]→(c:Category)
WHERE s.supplierID IN ids
return s, r1, p, r2, c;
```

Graph | Table | Text | Code

*(16) Supplier(3) Product(8) Category(5)

*(16) SUPPLIES(8) PART_OF(8)



Displaying 16 nodes, 16 relationships.

Focus on just a single category in our 3 suppliers example

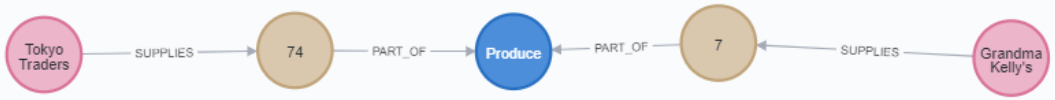
neo4j$

```
//consider 1 category within the 3 suppliers
WITH ['3', '4', '5'] as ids
MATCH (s:Supplier)-[r1]→(p:Product)-[r2]→(c:Category {categoryID: '7'})
WHERE s.supplierID IN ids
return s, r1, p, r2, c;
```

Graph | Table | Text | Code

*(5) Supplier(2) Product(2) Category(1)

*(4) SUPPLIES(2) PART_OF(2)



Displaying 5 nodes, 4 relationships.

Our query is specifying a PATTERN

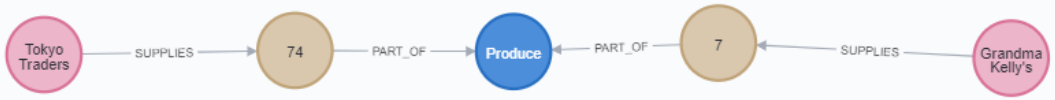
neo4j$

```
//consider 1 category within the 3 suppliers
WITH ['3', '4', '5'] as ids
MATCH (s:Supplier)-[r1]→(p:Product)-[r2]→(c:Category {categoryID: '7'})
WHERE s.supplierID IN ids
return s, r1, p, r2, c;
```

Graph | Table | Text | Code

*(5) Supplier(2) Product(2) Category(1)

*(4) SUPPLIES(2) PART_OF(2)



Displaying 5 nodes, 4 relationships.

Simpler format for the PATTERN does not specify any information about the relationship besides the direction

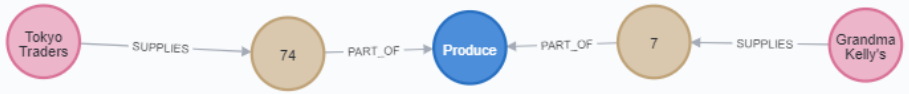
neo4j$

```
//consider 1 category within the 3 suppliers
//even simpler format for the pattern
WITH ['3', '4', '5'] as ids
MATCH (s:Supplier)⟶(p:Product)⟶(c:Category {categoryID: '7'})
WHERE s.supplierID IN ids
return s, p, c;
```

Graph | Table | Text | Code

*(5) Supplier(2) Product(2) Category(1)

*(4) SUPPLIES(2) PART_OF(2)



Displaying 5 nodes, 4 relationships.

Simplest format of the PATTERN, no direction on the relationships between NODES

neo4j$

```
//consider 1 category within the 3 suppliers
//simplest pattern between nodes, no direction on the relationships
WITH ['3', '4', '5'] as ids
MATCH (s:Supplier)--(p:Product)--(c:Category {categoryID: '7'})
WHERE s.supplierID IN ids
return s, p, c;
```

Graph | Table | Text | Code

*(5) Supplier(2) Product(2) Category(1)

*(4) SUPPLIES(2) PART_OF(2)

Displaying 5 nodes, 4 relationships.

Our PATTERN is describing a <u>PATH</u> between NODES

- `()--()--()`

- This path consists of 3 nodes in a series.

- However, we are not just looking for any NODES…we are looking for NODES associated with specific LABELS

- `(s:Supplier)--(p:Product)--(c:Category)`

What happens if we do NOT specify the label of the middle NODE in the PATH?

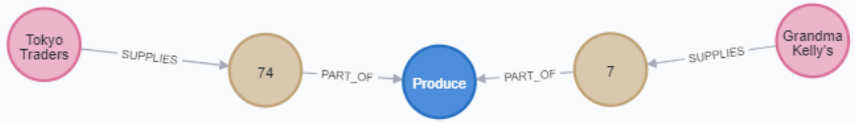
neo4j$

```
//consider 1 category within the 3 suppliers
//what happens if we do not specify the label for the middle node?
WITH ['3', '4', '5'] as ids
MATCH (s:Supplier)--(p)--(c:Category {categoryID: '7'})
WHERE s.supplierID IN ids
return s, p, c;
```

Graph | Table | Text | Code

*(5) Supplier(2) Product(2) Category(1)

*(4) SUPPLIES(2) PART_OF(2)



Displaying 5 nodes, 4 relationships.

We get the same result! For this example, we are returning a single NODE that related to Supplier and Category nodes!

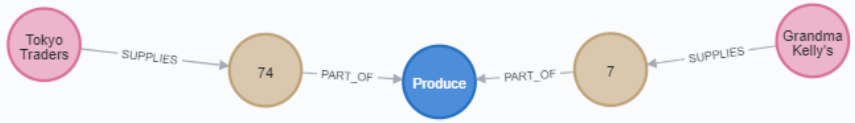
neo4j$

```
//consider 1 category within the 3 suppliers
//what happens if we do not specify the label for the middle node?
WITH ['3', '4', '5'] as ids
MATCH (s:Supplier)--(p)--(c:Category {categoryID: '7'})
WHERE s.supplierID IN ids
return s, p, c;
```

Graph | Table | Text | Code

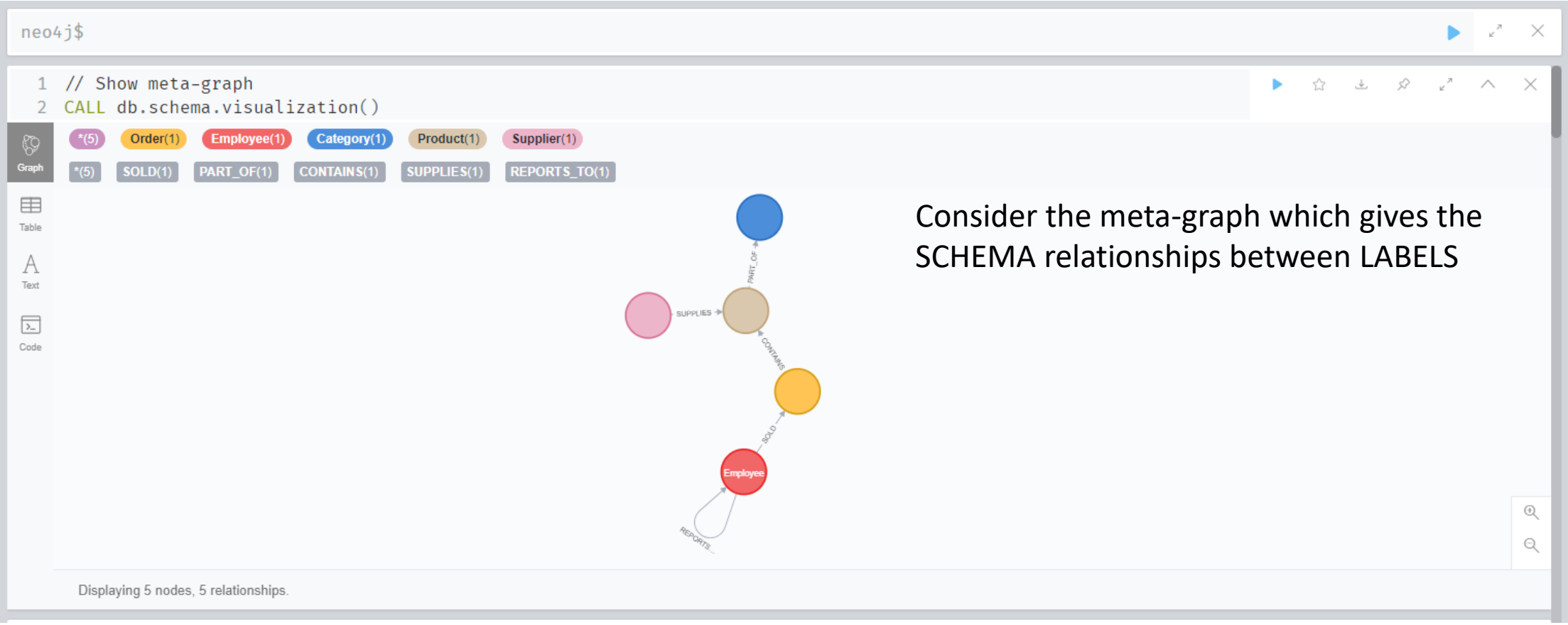
*(5) Supplier(2) Product(2) Category(1)

*(4) SUPPLIES(2) PART_OF(2)



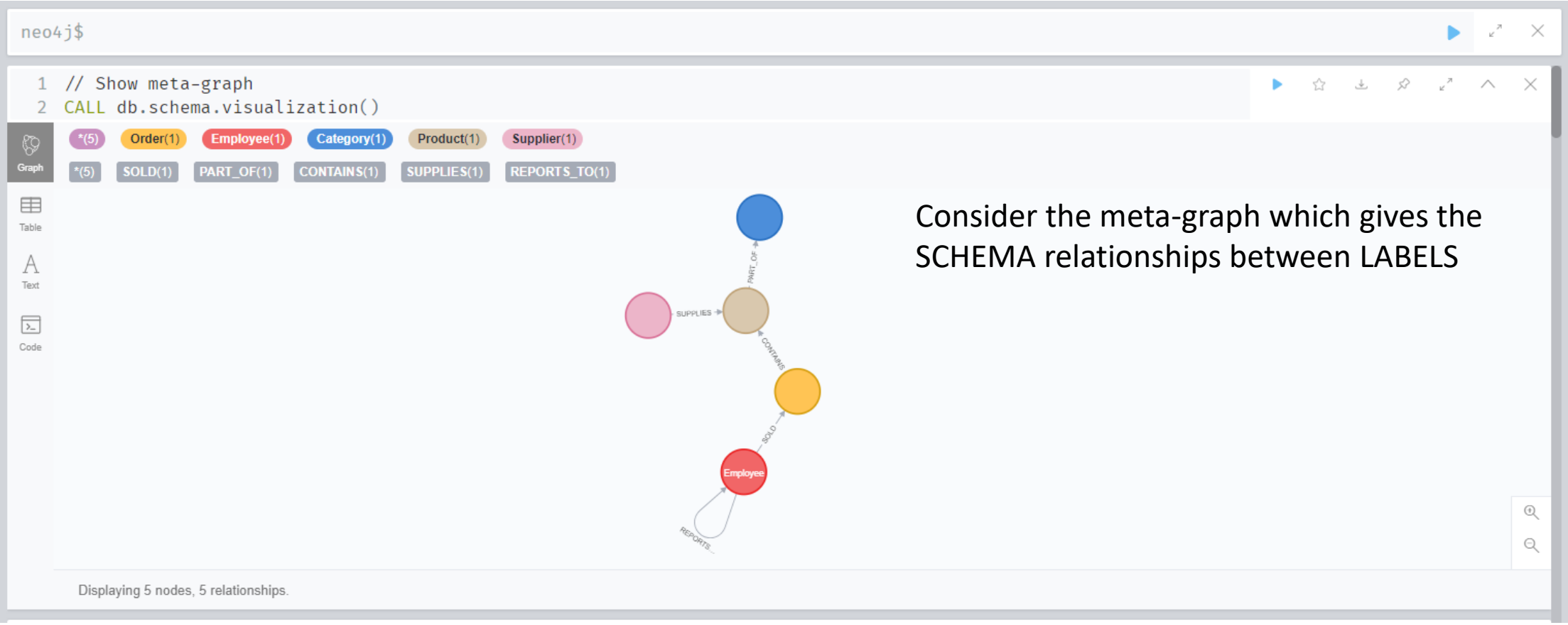
Displaying 5 nodes, 4 relationships.

What if we were interested in a longer PATH?



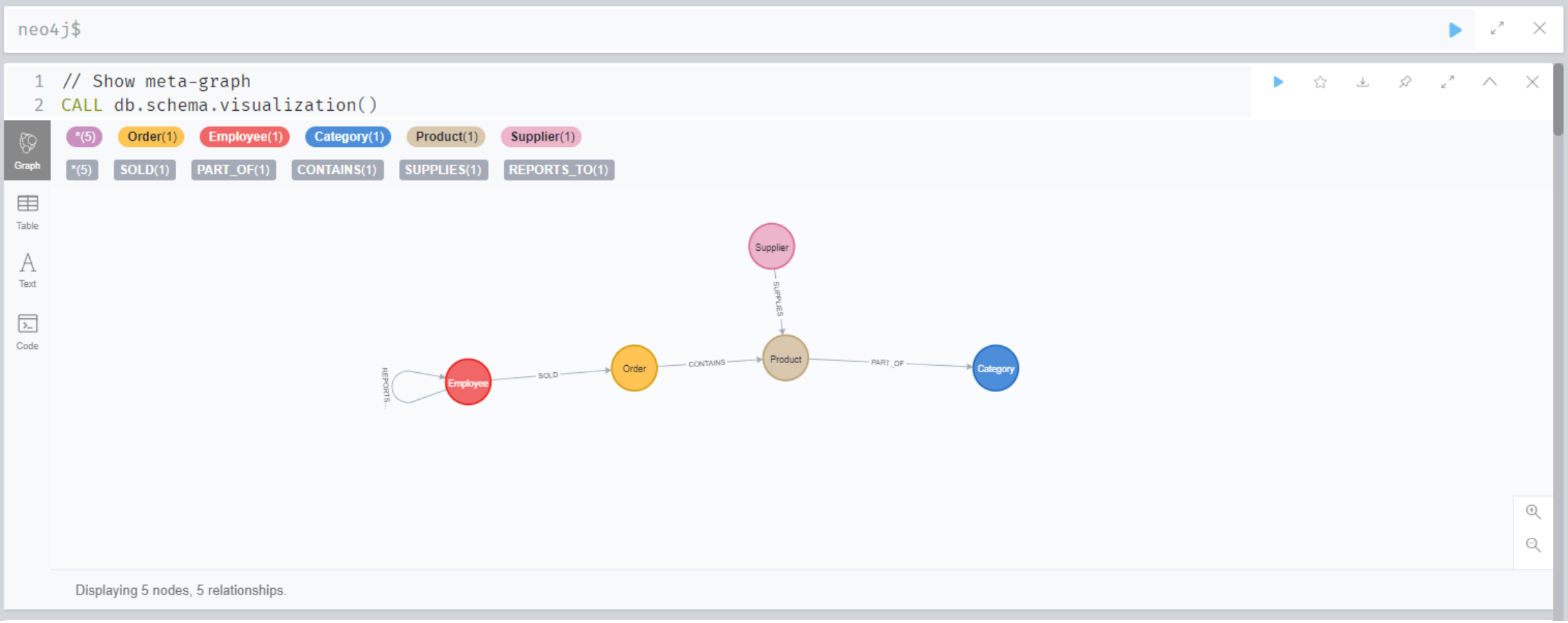
Consider the meta-graph which gives the SCHEMA relationships between LABELS

What if we were interested in a longer PATH?



Consider the meta-graph which gives the SCHEMA relationships between LABELS

For example, what if we want to work with the PATH between `(Employee)--(Order)--(Product)--(Category)`



Let’s focus just on a single employee

neo4j$

```
//employee-order-product-category path
//isolate one employee
MATCH (e:Employee {employeeID: '1'})--(o:Order)--(p:Product)--(c:Category)
RETURN e, o, p, c;
```

Graph | Table | Text | Code

*(204) | Employee(1) | Order(123) | Product(72) | Category(8)

*(540) | SOLD(123) | CONTAINS(345) | PART_OF(72)

Displaying 204 nodes, 540 relationships.

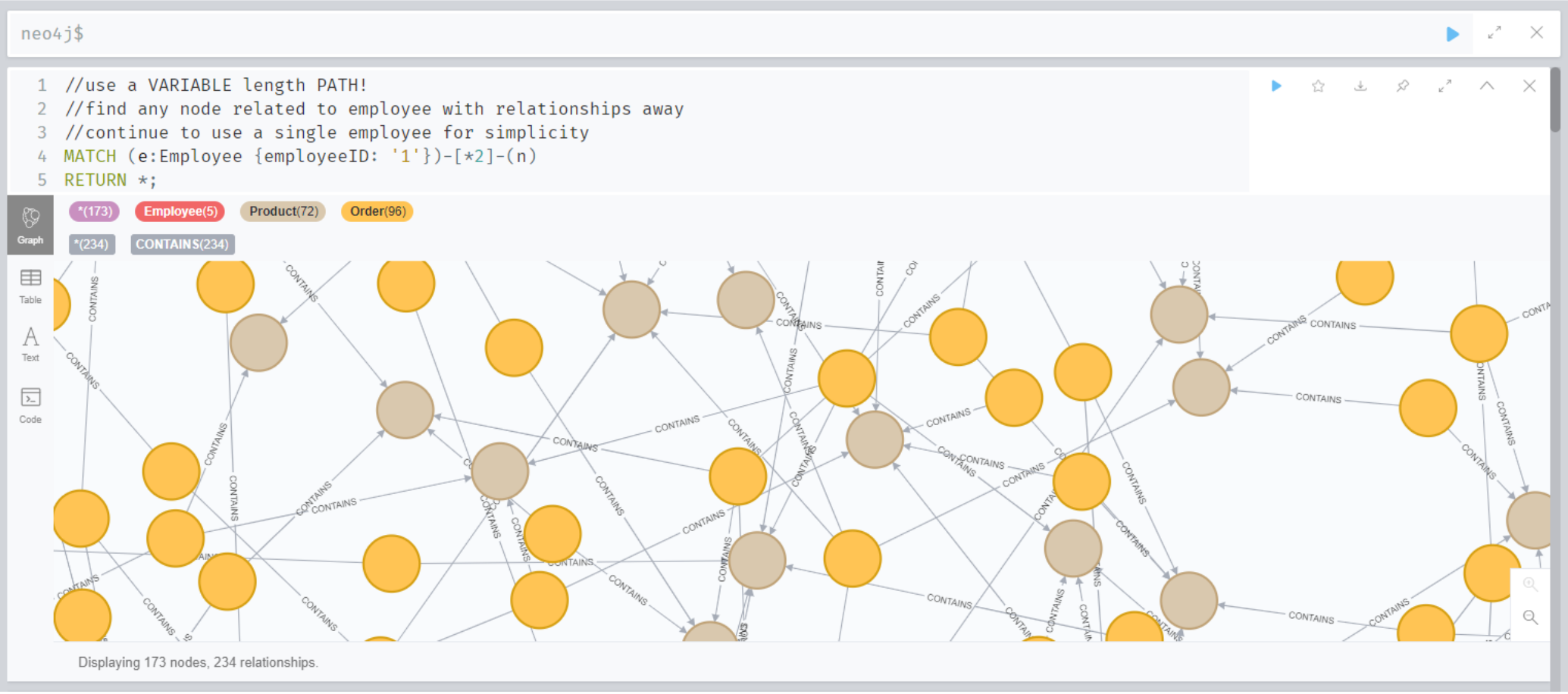
If we are just interested in the PATH between Employee and Category, do we need to know the explicit nodes between the two of interest?

- Instead of specifying the exact number of nodes in the PATH, let's consider VARIABLE LENGTH PATHS!

- The syntax for variable length paths is DIFFERENT than defined length paths.

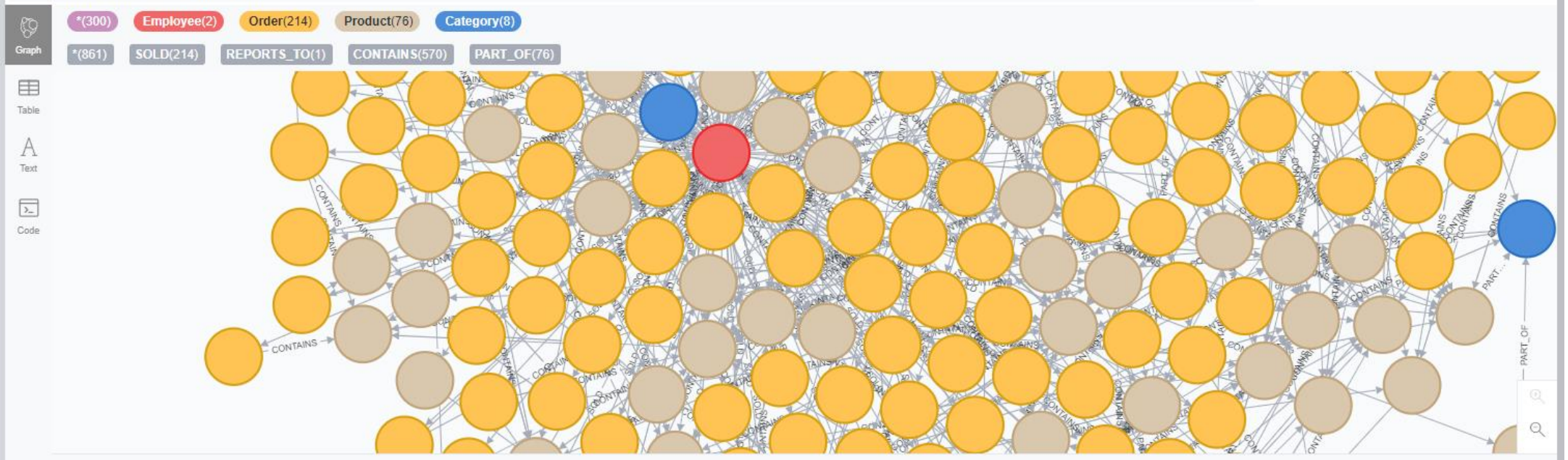
- We must use single dashes instead of two dashes and we must use the asterisk to specify the number of relationships or HOPS between nodes

Query all nodes 2 relationships (hops) away from Employee nodes



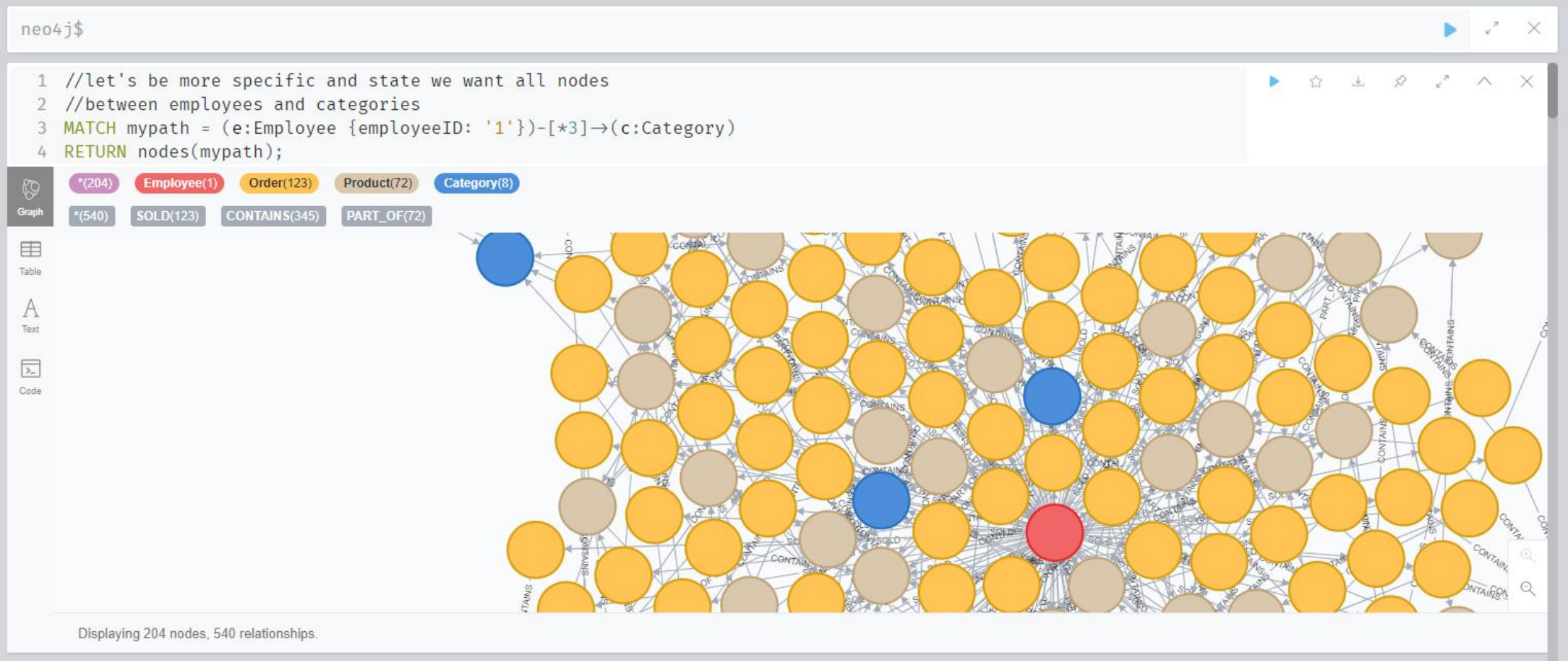
Assign the PATH to a variable and RETURN all nodes in the path with 3 relationships away from Employee nodes

```
//use a VARIABLE length PATH!
//find any node related to employee with 3 relationships away
//continue to use a single employee for simplicity
//assign the PATH to a variable and return NODES in the path
MATCH mypath = (e:Employee {employeeID: '1'})-[*3]→(n)
RETURN nodes(mypath);
```

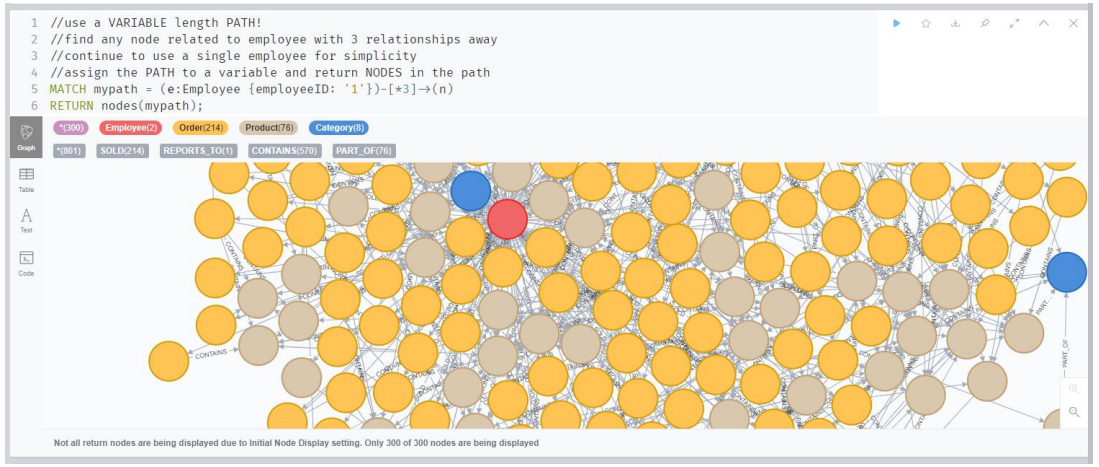


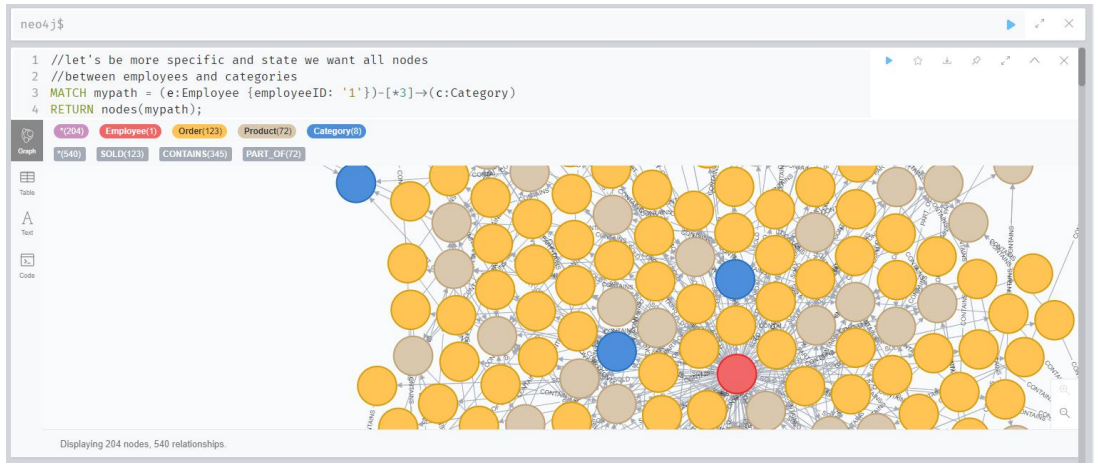
Not all return nodes are being displayed due to Initial Node Display setting. Only 300 of 300 nodes are being displayed

We can specify the start and end node Labels. This allows us to focus on a PATH of interest!



Number of returned NODES are different!
When we do NOT specify the end node LABEL we will return ANY node 3 relationships away!

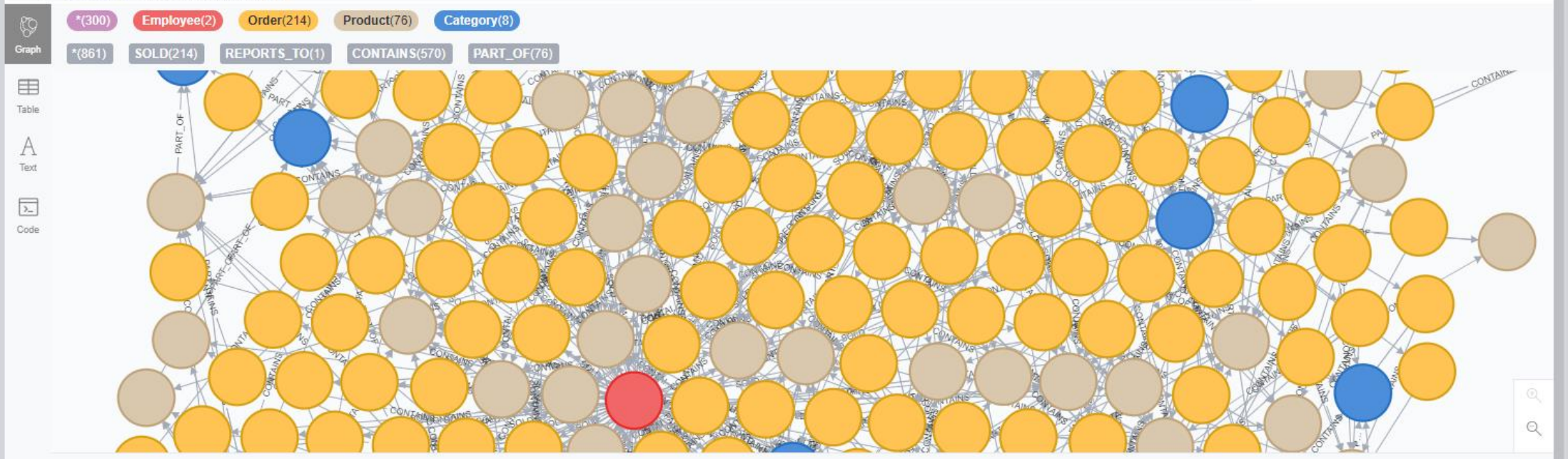




Can specify the variable length path several ways, for example all paths with 3 OR FEWER relationships

neo4j$

```
//can also return any node 3 or fewer relationships
MATCH mypath = (e:Employee {employeeID: '1'})-[*..3]→(n)
RETURN nodes(mypath);
```



Not all return nodes are being displayed due to Initial Node Display setting. Only 300 of 300 nodes are being displayed

Let’s go back to being explicit in querying the Employee to Order to Product to Category Path

- How many unique Categories are associated with Employee 1?

neo4j$

```
//how many unique categories are associated with employee 1?
//or what categories has employee 1 sold?
MATCH (e:Employee {employeeID: '1'})--(o:Order)--(p:Product)--(c:Category)
RETURN DISTINCT c;
```

Graph

Table

Text

Code

*(8) Category(8)

Displaying 8 nodes, 0 relationships.

How many categories were sold by each Employee?

```
//how many categories are sold per employee?
MATCH (e:Employee)--(o:Order)--(p:Product)--(c:Category)
RETURN e.employeeID, e.title, count(DISTINCT c);
```

Table | Text | Code

	e.employeeID	e.title	count(DISTINCT c)
1	"1"	"Sales Representative"	8
2	"2"	"Vice President, Sales"	8
3	"3"	"Sales Representative"	8
4	"4"	"Sales Representative"	8
5	"5"	"Sales Manager"	8
6	"6"	"Sales Representative"	8
7	[illegible]	[illegible]	[illegible]

Started streaming 9 records after 6 ms and completed after 75 ms.

How would we have calculated this in Pandas?

```
//how many categories are sold per employee?
MATCH (e:Employee)--(o:Order)--(p:Product)--(c:Category)
RETURN e.employeeID, e.title, count(DISTINCT c);
```

Table | Text | Code

	e.employeeID	e.title	count(DISTINCT c)
1	"1"	"Sales Representative"	8
2	"2"	"Vice President, Sales"	8
3	"3"	"Sales Representative"	8
4	"4"	"Sales Representative"	8
5	"5"	"Sales Manager"	8
6	"6"	"Sales Representative"	8
7	[illegible]	[illegible]	[illegible]

Started streaming 9 records after 6 ms and completed after 75 ms.

`orders` DataFrame gives us the `EmployeeID` for each `OrderID` + `ProductID` combination, while the `products` DataFrame gives us the `CategoryID` for each `ProductID`

```
In [62]: orders.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2155 entries, 0 to 2154
Data columns (total 19 columns):
OrderID           2155 non-null int64
CustomerID        2155 non-null object
EmployeeID        2155 non-null int64
OrderDate         2155 non-null object
RequiredDate      2155 non-null object
ShippedDate       2082 non-null object
ShipVia           2155 non-null int64
Freight           2155 non-null float64
ShipName          2155 non-null object
ShipAddress       2155 non-null object
ShipCity          2155 non-null object
ShipRegion        856 non-null object
ShipPostalCode    2100 non-null object
ShipCountry       2155 non-null object
OrderID.1         2155 non-null int64
ProductID         2155 non-null int64
UnitPrice         2155 non-null float64
Quantity          2155 non-null int64
Discount          2155 non-null float64
dtypes: float64(3), int64(6), object(10)
memory usage: 320.0+ KB
```

```
In [63]: products.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 77 entries, 0 to 76
Data columns (total 10 columns):
ProductID          77 non-null int64
ProductName        77 non-null object
SupplierID         77 non-null int64
CategoryID         77 non-null int64
QuantityPerUnit    77 non-null object
UnitPrice          77 non-null float64
UnitsInStock       77 non-null int64
UnitsOnOrder       77 non-null int64
ReorderLevel       77 non-null int64
Discontinued       77 non-null int64
dtypes: float64(1), int64(7), object(2)
memory usage: 6.1+ KB
```

How can we know which Employee sold which Category?

```
In [62]: orders.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2155 entries, 0 to 2154
Data columns (total 19 columns):
OrderID           2155 non-null int64
CustomerID        2155 non-null object
EmployeeID        2155 non-null int64
OrderDate         2155 non-null object
RequiredDate      2155 non-null object
ShippedDate       2082 non-null object
ShipVia           2155 non-null int64
Freight           2155 non-null float64
ShipName          2155 non-null object
ShipAddress       2155 non-null object
ShipCity          2155 non-null object
ShipRegion        856 non-null object
ShipPostalCode    2100 non-null object
ShipCountry       2155 non-null object
OrderID.1         2155 non-null int64
ProductID         2155 non-null int64
UnitPrice         2155 non-null float64
Quantity          2155 non-null int64
Discount          2155 non-null float64
dtypes: float64(3), int64(6), object(10)
memory usage: 320.0+ KB
```

```
In [63]: products.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 77 entries, 0 to 76
Data columns (total 10 columns):
ProductID          77 non-null int64
ProductName        77 non-null object
SupplierID         77 non-null int64
CategoryID         77 non-null int64
QuantityPerUnit    77 non-null object
UnitPrice          77 non-null float64
UnitsInStock       77 non-null int64
UnitsOnOrder       77 non-null int64
ReorderLevel       77 non-null int64
Discontinued       77 non-null int64
dtypes: float64(1), int64(7), object(2)
memory usage: 6.1+ KB
```

JOINS!

```
In [62]: orders.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2155 entries, 0 to 2154
Data columns (total 19 columns):
OrderID           2155 non-null int64
CustomerID        2155 non-null object
EmployeeID        2155 non-null int64
OrderDate         2155 non-null object
RequiredDate      2155 non-null object
ShippedDate       2082 non-null object
ShipVia           2155 non-null int64
Freight           2155 non-null float64
ShipName          2155 non-null object
ShipAddress       2155 non-null object
ShipCity          2155 non-null object
ShipRegion        856 non-null object
ShipPostalCode    2100 non-null object
ShipCountry       2155 non-null object
OrderID.1         2155 non-null int64
ProductID         2155 non-null int64
UnitPrice         2155 non-null float64
Quantity          2155 non-null int64
Discount          2155 non-null float64
dtypes: float64(3), int64(6), object(10)
memory usage: 320.0+ KB
```

```
In [63]: products.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 77 entries, 0 to 76
Data columns (total 10 columns):
ProductID          77 non-null int64
ProductName        77 non-null object
SupplierID         77 non-null int64
CategoryID         77 non-null int64
QuantityPerUnit    77 non-null object
UnitPrice          77 non-null float64
UnitsInStock       77 non-null int64
UnitsOnOrder       77 non-null int64
ReorderLevel       77 non-null int64
Discontinued       77 non-null int64
dtypes: float64(1), int64(7), object(2)
memory usage: 6.1+ KB
```

Join the `orders` and `products` DataFrames together

In [64]:
```
orders.loc[:, ['OrderID', 'EmployeeID', 'ProductID', 'Quantity']].copy().\
merge(products.loc[:, ['ProductID', 'CategoryID']].copy(), on='ProductID', how='left')
```

Out[64]:

	OrderID	EmployeeID	ProductID	Quantity	CategoryID
0	10248	5	11	12	4
1	10248	5	42	10	5
2	10248	5	72	5	4
3	10249	6	14	9	7
4	10249	6	51	40	7
...	...	...	...	...	...
2150	11077	1	64	2	5
2151	11077	1	66	1	2
2152	11077	1	73	2	8
2153	11077	1	75	4	1
2154	11077	1	77	2	2

2155 rows × 5 columns

Perform the groupby and aggregation operations

In [65]:
```
orders.loc[:, ['OrderID', 'EmployeeID', 'ProductID', 'Quantity']].copy().\
merge(products.loc[:, ['ProductID', 'CategoryID']].copy(), on='ProductID', how='left').\
groupby(['EmployeeID']).\
aggregate(num_rows = ('Quantity', 'size'),
          num_categories = ('CategoryID', 'nunique')).\
reset_index()
```

Out[65]:

	EmployeeID	num_rows	num_categories
0	1	345	8
1	2	241	8
2	3	321	8
3	4	420	8
4	5	117	8
5	6	168	8
6	7	176	8
7	8	260	8
8	9	107	8

Join with the `employees` DataFrame to bring in the `Title` for each Employee

In [66]:
```
orders.loc[:, ['OrderID', 'EmployeeID', 'ProductID', 'Quantity']].copy().\
merge(products.loc[:, ['ProductID', 'CategoryID']].copy(), on='ProductID', how='left').\
groupby(['EmployeeID']).\
aggregate(num_rows = ('Quantity', 'size'),
          num_categories = ('CategoryID', 'nunique')).\
reset_index().\
merge(employees.loc[:, ['EmployeeID', 'Title']].copy(), on='EmployeeID', how='left')
```

Out[66]:

	EmployeeID	num_rows	num_categories	Title
0	1	345	8	Sales Representative
1	2	241	8	Vice President, Sales
2	3	321	8	Sales Representative
3	4	420	8	Sales Representative
4	5	117	8	Sales Manager
5	6	168	8	Sales Representative
6	7	176	8	Sales Representative
7	8	260	8	Inside Sales Coordinator
8	9	107	8	Sales Representative

The graph statement to do all of those actions is just 2 lines of code!!!

```
//how many categories are sold per employee?
MATCH (e:Employee)--(o:Order)--(p:Product)--(c:Category)
RETURN e.employeeID, e.title, count(DISTINCT c);
```

Table | Text | Code

	e.employeeID	e.title	count(DISTINCT c)
1	"1"	"Sales Representative"	8
2	"2"	"Vice President, Sales"	8
3	"3"	"Sales Representative"	8
4	"4"	"Sales Representative"	8
5	"5"	"Sales Manager"	8
6	"6"	"Sales Representative"	8
7	[illegible]	[illegible]	[illegible]

Started streaming 9 records after 6 ms and completed after 75 ms.

What if we wanted to know the number of unique Products sold per Employee? In Pandas we just need to add another aggregation function.

In [67]:
```
orders.loc[:, ['OrderID', 'EmployeeID', 'ProductID', 'Quantity']].copy().\
merge(products.loc[:, ['ProductID', 'CategoryID']].copy(), on='ProductID', how='left').\
groupby(['EmployeeID']).\
aggregate(num_rows = ('Quantity', 'size'),
          num_categories = ('CategoryID', 'nunique'),
          num_products = ('ProductID', 'nunique')).\
reset_index().\
merge(employees.loc[:, ['EmployeeID', 'Title']].copy(), on='EmployeeID', how='left')
```

Out[67]:

	EmployeeID	num_rows	num_categories	num_products	Title
0	1	345	8	72	Sales Representative
1	2	241	8	68	Vice President, Sales
2	3	321	8	74	Sales Representative
3	4	420	8	75	Sales Representative
4	5	117	8	52	Sales Manager
5	6	168	8	57	Sales Representative
6	7	176	8	67	Sales Representative
7	8	260	8	70	Inside Sales Coordinator
8	9	107	8	53	Sales Representative

How would we write this query in Cypher?

```
//how many unique Product types were sold per employee?
MATCH (e:Employee)--(o:Order)--(p:Product)--(c:Category)
RETURN e.employeeID, e.title, count(DISTINCT c), count(DISTINCT p);
```

Table | Text | Code

	e.employeeID	e.title	count(DISTINCT c)	count(DISTINCT p)
1	"1"	"Sales Representative"	8	72
2	"2"	"Vice President, Sales"	8	68
3	"3"	"Sales Representative"	8	74
4	"4"	"Sales Representative"	8	75
5	"5"	"Sales Manager"	8	52
6	"6"	"Sales Representative"	8	57
7	[illegible]	[illegible]	[illegible]	[illegible]

Started streaming 9 records after 7 ms and completed after 15 ms.

What happens if we remove the Employee properties from the RETURN call?

```
//how many unique Product types were sold per employee?
//remove employee properties from RETURN
MATCH (e:Employee)--(o:Order)--(p:Product)--(c:Category)
RETURN count(DISTINCT c), count(DISTINCT p);
```

Table | Text | Code

	count(DISTINCT c)	count(DISTINCT p)
1	8	77

Started streaming 1 records after 5 ms and completed after 13 ms.

The grouping per employee goes away!!!

So CYPHER implicitly groups!

What if we want to know the total quantity sold per Product per Employee?

In [68]:
```
orders.groupby(['EmployeeID', 'ProductID']).\
aggregate(num_rows = ('Quantity', 'size'),
          total_quantity = ('Quantity', 'sum')).\
reset_index()
```

Out[68]:

	EmployeeID	ProductID	num_rows	total_quantity
0	1	1	2	80
1	1	2	8	231
2	1	3	3	68
3	1	4	2	6
4	1	5	1	65
...	...	...	...	...
583	9	72	2	8
584	9	74	1	36
585	9	75	4	110
586	9	76	3	73
587	9	77	2	53

588 rows × 4 columns

How do we do this in Cypher? How did we “store” the quantity property?

```
//quantity is a property of a relationship!
MATCH (o:Order)-[r:CONTAINS]-(p:Product)
RETURN o.orderID, r.quantity, p.productID
LIMIT 5;
```

Table | Text | Code

	o.orderID	r.quantity	p.productID
1	"10847"	80.0	"1"
2	"10918"	60.0	"1"
3	"10576"	10.0	"1"
4	"10935"	21.0	"1"
5	"11031"	45.0	"1"

Started streaming 5 records after 8 ms and completed after 10 ms.

Quantity is a property of the CONTAINS relationship!

Do we get the same result that we had from Pandas?

```
//total quantity of a product per employee
MATCH (e:Employee)--(o:Order)-[r:CONTAINS]-(p:Product)--(c:Category)
RETURN e.employeeID, p.productID, sum(r.quantity)
ORDER BY e.employeeID, p.productID;
```

Table | Text | Code

	e.employeeID	p.productID	sum(r.quantity)
1	"1"	"1"	80.0
2	"1"	"10"	265.0
3	"1"	"11"	85.0
4	"1"	"12"	68.0
5	"1"	"13"	80.0
6	"1"	"14"	68.0
7	[illegible]	[illegible]	[illegible]

Started streaming 588 records in less than 1 ms and completed after 18 ms.

If we remove the employee property from the RETURN clause we sum the quantity per Product!

```
//remove employee from the RETURN clause
//change the data type of productID
MATCH (e:Employee)--(o:Order)-[r:CONTAINS]-(p:Product)--(c:Category)
RETURN p.productID, sum(r.quantity)
ORDER BY toFloat(p.productID);
```

Sort in the correct numeric order for the ProductID!

Table | Text | Code

	p.productID	sum(r.quantity)
1	"1"	828.0
2	"2"	1057.0
3	"3"	328.0
4	"4"	453.0
5	"5"	298.0
6	"6"	301.0
7	[illegible]	[illegible]

Started streaming 77 records after 6 ms and completed after 17 ms.

Check with Pandas, do we get the same thing?

In [69]:
```
orders.groupby(['ProductID']).\
aggregate(num_rows = ('Quantity', 'size'),
          total_quantity = ('Quantity', 'sum')).\
reset_index()
```

Out[69]:

	ProductID	num_rows	total_quantity
0	1	38	828
1	2	44	1057
2	3	12	328
3	4	20	453
4	5	10	298
...	...	...	...
72	73	14	293
73	74	13	297
74	75	46	1155
75	76	39	981
76	77	38	791

77 rows × 3 columns