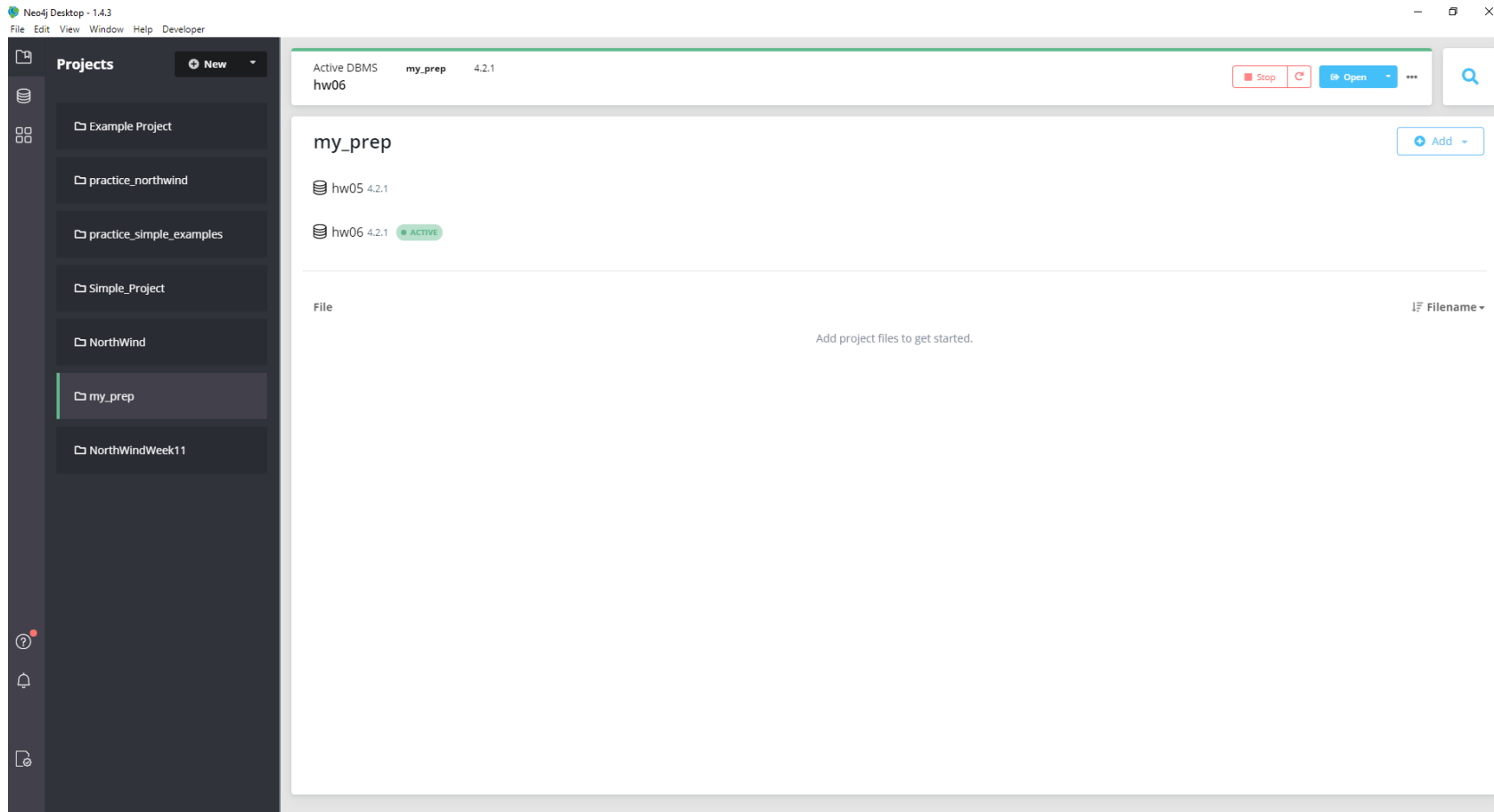


CMPINF 2110

Spring 2021

Homework 06 Solutions

Create a new local DBMS in Neo4j desktop. I chose to name mine hw06.



Add Device Label and create nodes with properties set to the columns of the devices table

The screenshot displays the Neo4j Browser interface. On the left, the 'Database Information' sidebar shows the following details:

- Use database:** neo4j - default
- Node Labels:** (4) Device
- Relationship Types:** No relationships in database
- Property Keys:** class, deviceId
- Connected as:** Username: neo4j, Roles: admin, PUBLIC, Admin: .server user list, .server user add, Disconnect: .server disconnect
- DBMS:** Version: 4.2.1, Edition: Enterprise, Name: neo4j, Databases: .dbs, Information: .sysinfo, Query List: .queries

The main query editor shows the following Cypher query:

```
1 //create devices nodes
2 LOAD CSV WITH HEADERS FROM
  'https://raw.githubusercontent.com/jyurko/CMPINF_2110_Spring_2021_data/main/hw06/devices.csv' AS row
3 MERGE (device:Device {deviceId: row.device_id})
4 ON CREATE SET device.class = row.class;
```

The execution result indicates: 'Added 4 labels, created 4 nodes, set 8 properties, completed after 676 ms.'

Below the query editor, the ':play start' button is visible. At the bottom, there are three cards for getting started, a play guide, and Cypher basics, each with a corresponding button.

Add Employee Label and create nodes with properties set to the columns of the employees table

The screenshot displays the Neo4j Browser interface. On the left, the 'Database Information' sidebar shows the database name 'neo4j - default', 'Node Labels' (Device, Employee), 'Relationship Types' (None), 'Property Keys' (class, deviceID, employeeID, name, started), and 'Connected as' (Username: neo4j, Roles: admin, PUBLIC). The main area shows two Cypher queries and their results.

Query 1: Create machine nodes

```
1 //create machine nodes
2 LOAD CSV WITH HEADERS FROM
  'https://raw.githubusercontent.com/jyurko/CMPINF_2110_Spring_2021_data/main/hw06/employees.csv' AS row
3 MERGE (employee:Employee {employeeID: row.employee_id})
4   ON CREATE SET employee.name = row.name,
5   employee.started = row.started;
```

Added 5 labels, created 5 nodes, set 15 properties, completed after 288 ms.

Query 2: Create devices nodes

```
1 //create devices nodes
2 LOAD CSV WITH HEADERS FROM
  'https://raw.githubusercontent.com/jyurko/CMPINF_2110_Spring_2021_data/main/hw06/devices.csv' AS row
3 MERGE (device:Device {deviceID: row.device_id})
4   ON CREATE SET device.class = row.class;
```

Added 4 labels, created 4 nodes, set 8 properties, completed after 676 ms.

Add Machine Label and create nodes with properties set to the columns of the machines table

The screenshot displays the Neo4j Browser interface. On the left, the 'Database Information' sidebar shows the following details:

- Use database:** neo4j - default
- Node Labels:** (13) Device, Employee, Machine
- Relationship Types:** No relationships in database
- Property Keys:** class, deviceID, employeeID, machineID, name, started, type
- Connected as:** Username: neo4j, Roles: admin, PUBLIC, Admin: server user list, server user add, server disconnect, Disconnect: server disconnect
- DBMS:** Version: 4.2.1, Edition: Enterprise, Name: neo4j, Databases: db, sysinfo, Information: sysinfo, Query List: queries

The main panel shows two executed Cypher queries:

Query 1:

```
1 //create machine nodes
2 LOAD CSV WITH HEADERS FROM
3   'https://raw.githubusercontent.com/jyurko/CMPINF_2110_Spring_2021_data/main/hw06/machines.csv' AS row
4 MERGE (machine:Machine {machineID: row.machine_id})
5   ON CREATE SET machine.name = row.name,
6   machine.type = row.type;
```

Added 4 labels, created 4 nodes, set 12 properties, completed after 225 ms.

Query 2:

```
1 //create machine nodes
2 LOAD CSV WITH HEADERS FROM
3   'https://raw.githubusercontent.com/jyurko/CMPINF_2110_Spring_2021_data/main/hw06/employees.csv' AS row
4 MERGE (employee:Employee {employeeID: row.employee_id})
5   ON CREATE SET employee.name = row.name,
6   employee.started = row.started;
```

Added 5 labels, created 5 nodes, set 15 properties, completed after 288 ms.

Add Part Label and create nodes with properties set to the columns of the parts table

The screenshot displays the Neo4j Browser interface. On the left, the 'Database Information' sidebar shows the database name 'neo4j - default', node labels 'Device' and 'Employee', and relationship types. The main panel shows two Cypher queries and their execution results.

Query 1: Create part nodes

```
1 //create part nodes
2 LOAD CSV WITH HEADERS FROM
  'https://raw.githubusercontent.com/jyurko/CMPINF_2110_Spring_2021_data/main/hw06/parts.csv' AS row
3 MERGE (part:Part {partID: row.part_id})
4   ON CREATE SET part.type = row.type;
```

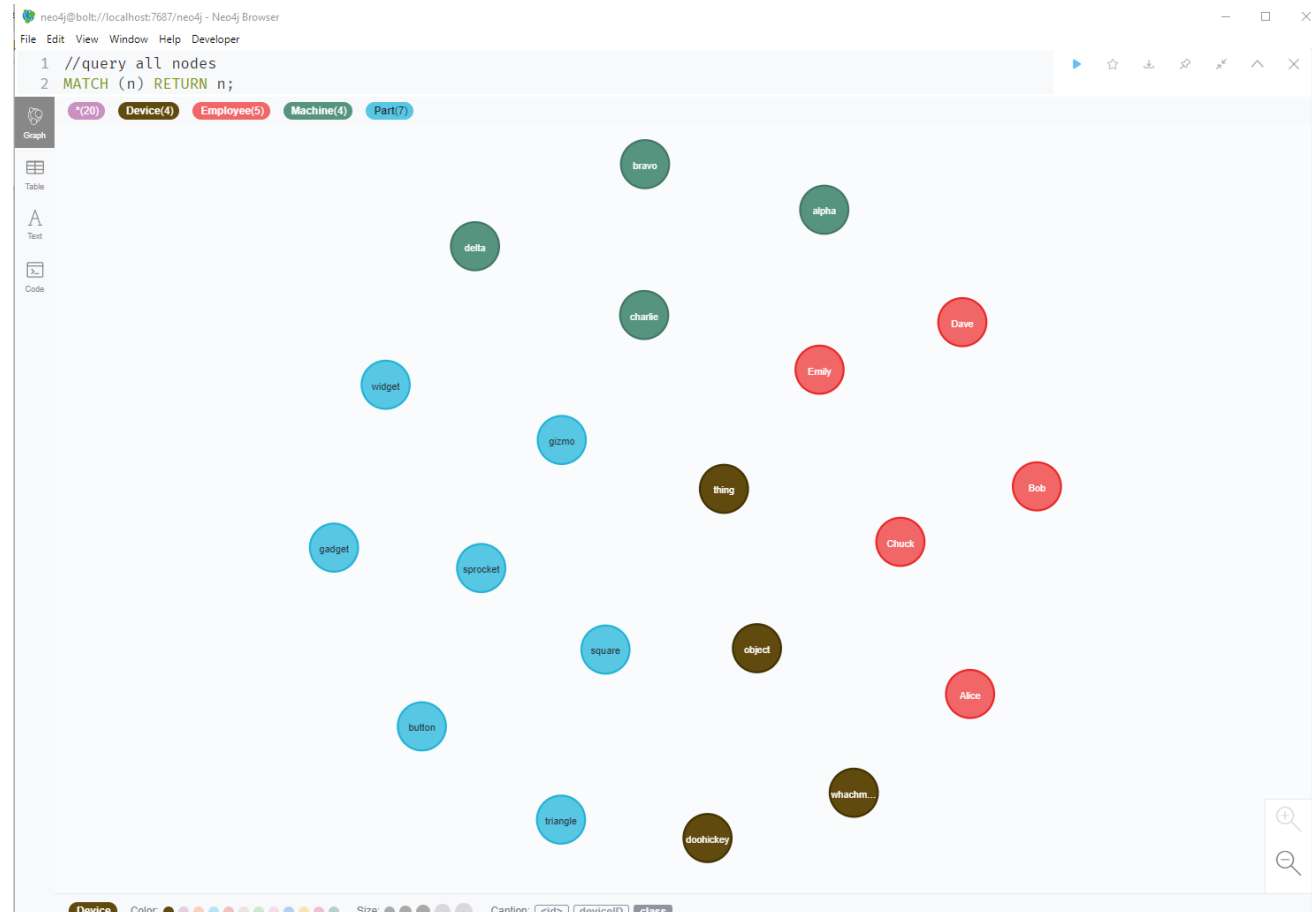
Added 7 labels, created 7 nodes, set 14 properties, completed after 187 ms.

Query 2: Create machine nodes

```
1 //create machine nodes
2 LOAD CSV WITH HEADERS FROM
  'https://raw.githubusercontent.com/jyurko/CMPINF_2110_Spring_2021_data/main/hw06/machines.csv' AS row
3 MERGE (machine:Machine {machineID: row.machine_id})
4   ON CREATE SET machine.name = row.name,
5   machine.type = row.type;
```

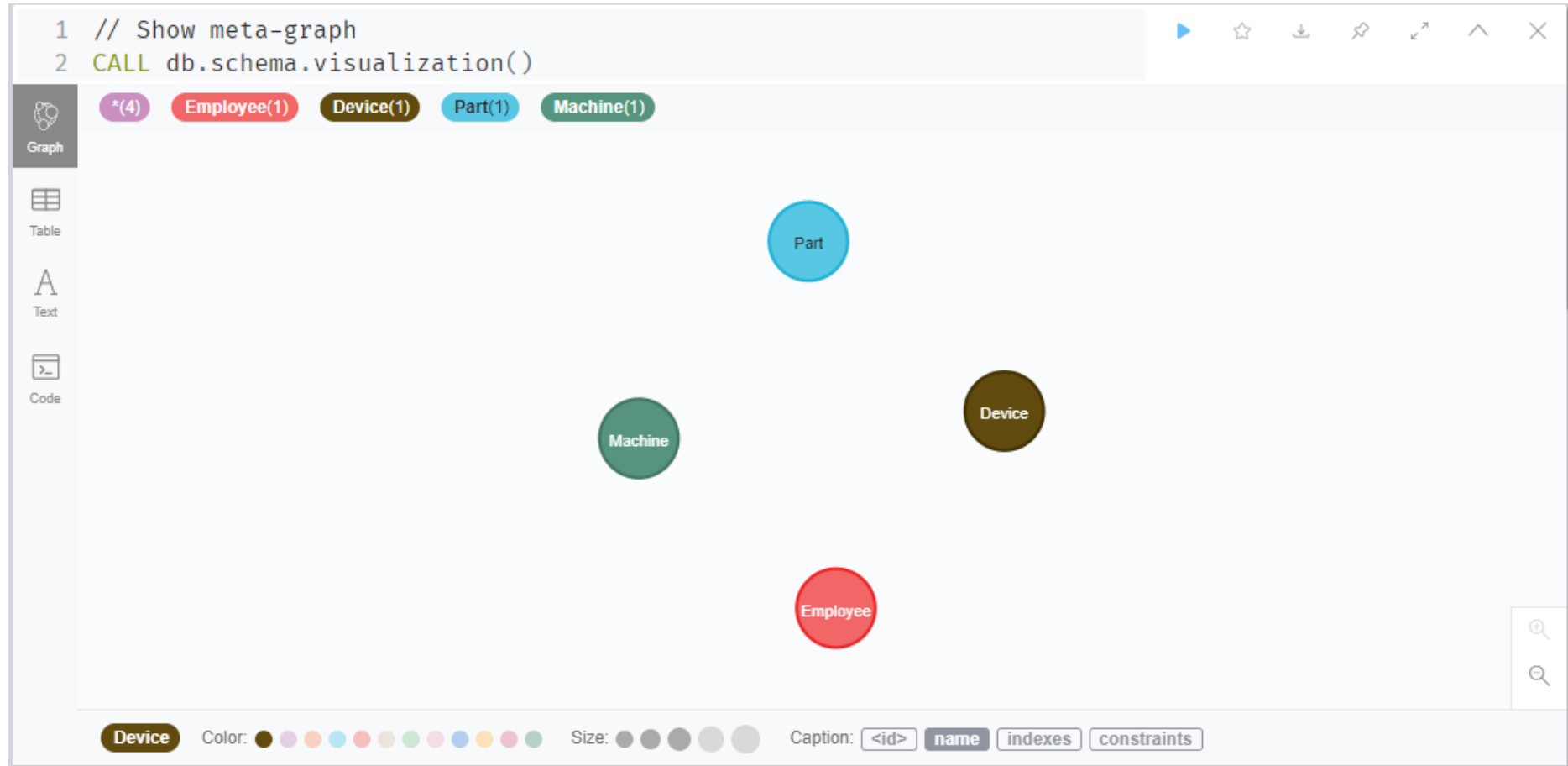
Added 4 labels, created 4 nodes, set 12 properties, completed after 225 ms.

As a check, query all nodes in the graph model



No relationships are defined as of yet!

The “meta graph” is another way to reveal no relationships are defined as of yet



Add Job label and create nodes with jobID property

The screenshot displays the Neo4j Browser interface. On the left, the 'Database Information' sidebar shows the following details:

- Use database:** neo4j - default
- Node Labels:** (69) Device, Employee, Job, Machine, Part
- Relationship Types:** No relationships in database
- Property Keys:** class, deviceID, employeeID, jobID, machineID, name, partID, started, type
- Connected as:** Username: neo4j, Roles: admin, PUBLIC, Admin: server user list, server user add, server disconnect
- DBMS:** Version: 4.2.1, Edition: Enterprise, Name: neo4j, Databases: db, sysinfo, queries

The main panel shows the execution of a Cypher query in the 'neo4j\$' console:

```
1 //create job nodes
2 LOAD CSV WITH HEADERS FROM
  'https://raw.githubusercontent.com/jyurko/CMPINF_2110_Spring_2021_data/main/hw06/machines_prints_jobs.csv' AS row
3 MERGE (job:Job {jobID: row.job_id});
```

The execution result indicates: 'Added 49 labels, created 49 nodes, set 49 properties, completed after 215 ms.'

Below the console, a 'Show meta-graph' query is executed:

```
1 // Show meta-graph
2 CALL db.schema.visualization();
```

The resulting meta-graph visualization shows five nodes: Employee (red), Device (brown), Part (blue), Machine (green), and Job (yellow). The nodes are arranged in a circular pattern, with Job at the top, Machine on the left, Device on the right, Employee at the bottom, and Part at the top-right.

Add Item label and create nodes with itemID property

The screenshot displays the Neo4j Browser interface. On the left, the 'Database Information' sidebar shows the database name 'neo4j - default' and a list of node labels: 'Device', 'Employee', 'Item', 'Job', 'Machine', and 'Part'. The 'Relationship Types' section indicates 'No relationships in database'. The 'Property Keys' section lists various properties like 'class', 'deviceID', 'employeeID', 'itemID', 'jobID', 'machineID', 'name', 'partID', 'started', and 'type'. The 'Connected as' section shows the user 'neo4j' with roles 'admin, PUBLIC'. The 'DBMS' section shows version '4.2.1' and edition 'Enterprise'.

The main panel shows the Neo4j Cypher query editor with the following code:

```
neo4j$  
1 //create item nodes  
2 LOAD CSV WITH HEADERS FROM  
   'https://raw.githubusercontent.com/jyurko/CMPINF_2110_Spring_2021_data/main/hw06/parts_in_jobs.csv' AS row  
3 MERGE (item:Item {itemID: row.item_id});
```

The execution results show: 'Added 149 labels, created 149 nodes, set 149 properties, completed after 1075 ms.'

Below the query editor, the 'Show meta-graph CALL db.schema.visualization()' command is executed, resulting in a graph visualization. The graph shows six nodes: 'Employee' (red), 'Device' (dark blue), 'Part' (light blue), 'Job' (orange), 'Machine' (green), and 'Item' (purple). The nodes are connected by lines, representing relationships between them.

Add Assembly label and create nodes with assemblyID property

The screenshot displays the Neo4j Browser interface. On the left sidebar, the 'Database Information' panel is active, showing the following details:

- Use database:** neo4j - default
- Node Labels:** Assembly (258), Employee, Item, Job, Machine, Part, Device
- Relationship Types:** No relationships in database
- Property Keys:** assemblyID, class, deviceID, employeeID, itemID, jobID, machineID, name, partID, started, type
- Connected as:** Username: neo4j, Roles: admin, PUBLIC, Admin: server user list, server user add, server disconnect, Disconnect: server disconnect
- DBMS:** Version: 4.2.1, Edition: Enterprise, Name: neo4j, Databases: dbs, Information: sysinfo, Query List: queries

The main query editor shows the following Cypher query:

```
neo4j$  
1 //create assembly nodes  
2 LOAD CSV WITH HEADERS FROM  
   'https://raw.githubusercontent.com/jyurko/CMPINF_2110_Spring_2021_data/main/hw06/employees_assembles_devices.csv' AS row  
3 MERGE (assembly:Assembly {assemblyID: row.assembly_id});
```

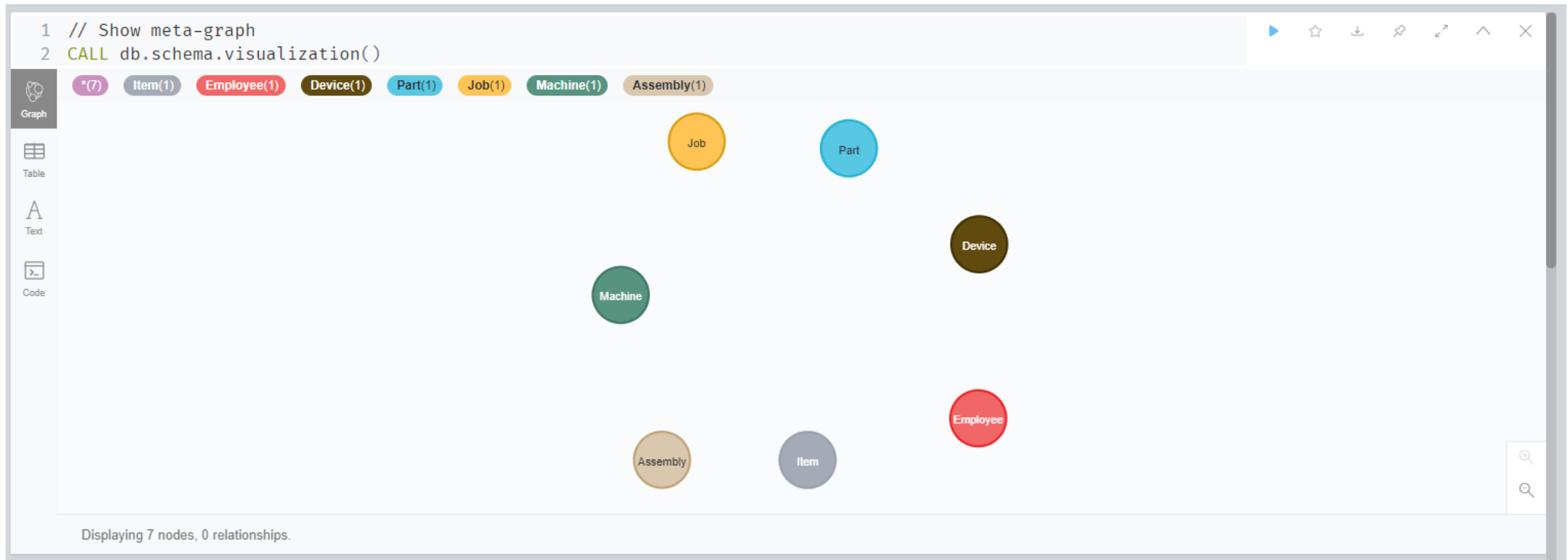
The execution result indicates: "Added 50 labels, created 50 nodes, set 50 properties, completed after 203 ms."

Below the query editor, a second command is entered:

```
neo4j$ // Show meta-graph CALL db.schema.visualization()
```

The result shows a meta-graph visualization with nodes representing the different labels: (0), Item(1), Employee(1), Device(1), Part(1), Job(1), and Machine(1). The graph shows a central 'Part' node connected to 'Job', 'Machine', and 'Employee' nodes, with 'Device' and 'Item' nodes also present.

All nodes have been created. The meta-graph shows no relationships have been defined as of yet.



Create the ASSEMBLES relationship type between Employee nodes and Assembly nodes

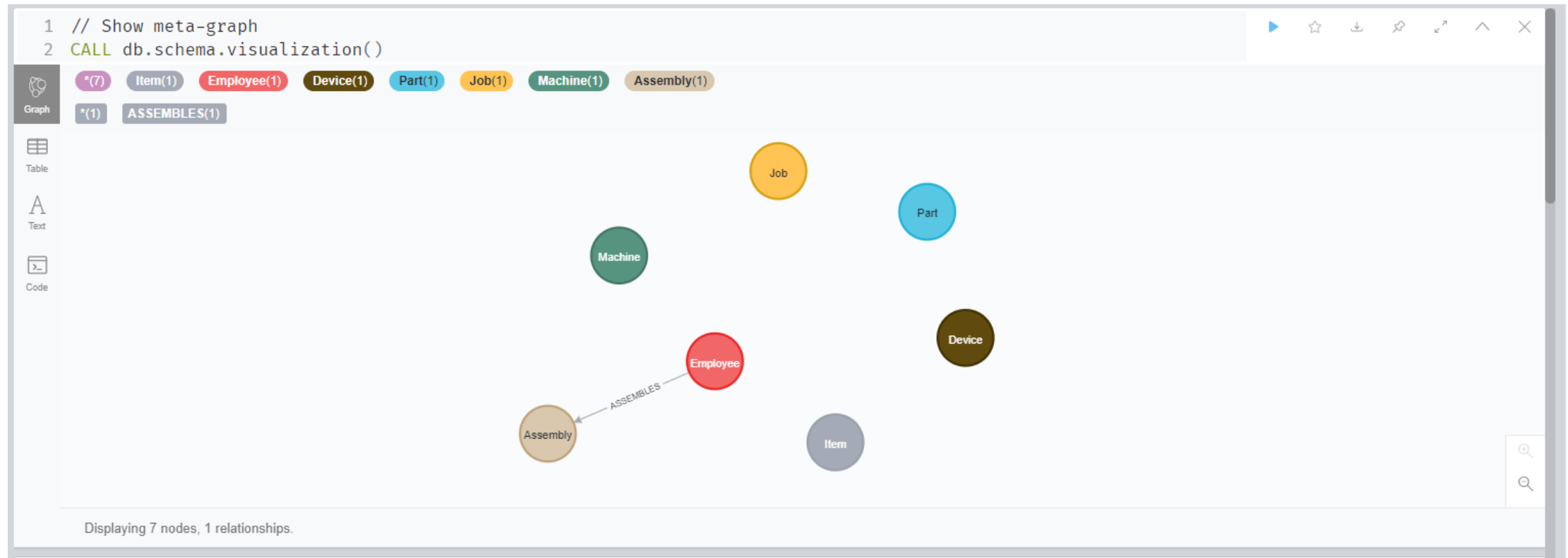
The screenshot displays the Neo4j Browser interface. On the left sidebar, the 'Database Information' panel shows the database 'neo4j - default'. Under 'Node Labels', there are 268 labels including 'Assembly', 'Device', 'Employee', 'Item', 'Job', 'Machine', and 'Part'. Under 'Relationship Types', there is 1 relationship type named 'ASSEMBLES'. The 'Property Keys' section lists various keys like 'assemblyID', 'class', 'deviceID', 'employeeID', 'itemID', 'jobID', 'machineID', 'name', 'partID', 'started', and 'type'. The 'Connected as' section shows the user 'neo4j' with roles 'admin, PUBLIC'. The 'DBMS' section shows version '4.2.1', edition 'Enterprise', name 'neo4j', and databases 'dbs', 'sysinfo', and 'queries'.

The main panel shows a Cypher query executed in the 'neo4j\$' session:

```
1 //create relationship between employee and assembly
2 LOAD CSV WITH HEADERS FROM
  'https://raw.githubusercontent.com/jyurko/CMPINF_2110_Spring_2021_data/main/hw06/employees_assemblies_devices.csv' AS row
3 MATCH (employee:Employee {employeeID: row.employee_id})
4 MATCH (assembly:Assembly {assemblyID: row.assembly_id})
5 MERGE (employee)-[:ASSEMBLES]->(assembly);
```

The query result shows 'Created 50 relationships, completed after 295 ms.' Below the query, a 'Graph' visualization shows a meta-graph with nodes for 'Job', 'Part', 'Device', 'Machine', and 'Employee'. The nodes are represented by colored circles with their respective counts in parentheses: Job(7), Item(1), Employee(1), Device(1), Part(1), Job(1), Machine(1), Assembly(1), and Employee(1).

We now have a relationship!



Create the OPERATES relationship type between Employee and Job nodes

The screenshot displays the Neo4j Browser interface with the following components:

- Database Information Panel (Left):**
 - Use database:** neo4j - default
 - Node Labels:** *268, Assembly, Device, Employee, Item, Job, Machine, Part
 - Relationship Types:** *99, ASSEMBLES, OPERATES
 - Property Keys:** assemblyID, class, deviceID, employeeID, itemID, jobID, machineID, name, partID, started, type
 - Connected as:** Username: neo4j, Roles: admin, PUBLIC, Admin: .server user list, .server user add, .server disconnect
 - DBMS:** Version: 4.2.1, Edition: Enterprise, Name: neo4j, Databases: db, Information: sysinfo, Query List: queries
- Query Editor (Top Right):**

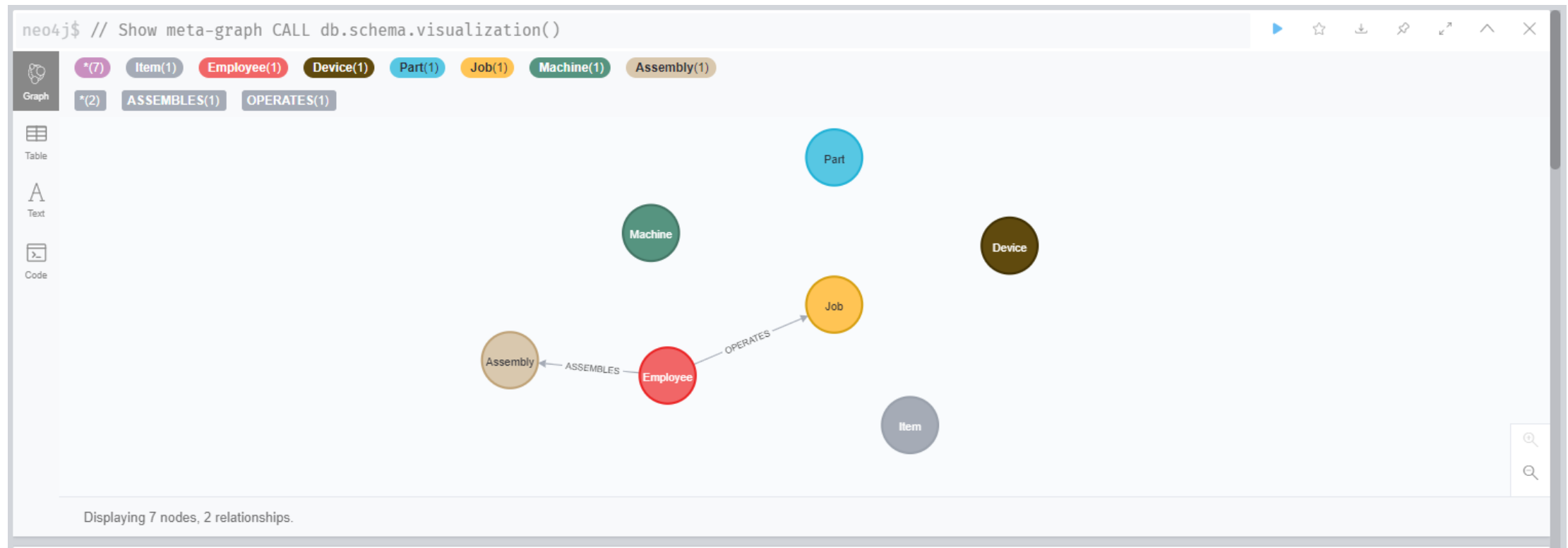
```
neo4j$  
  
1 //create relationship between employee and job  
2 LOAD CSV WITH HEADERS FROM  
   'https://raw.githubusercontent.com/jyurko/CMPINF_2110_Spring_2021_data/main/hw06/machines_prints_jobs.csv' AS  
   row  
3 MATCH (employee:Employee {employeeID: row.employee_id})  
4 MATCH (job:Job {jobID: row.job_id})  
5 MERGE (employee)-[:OPERATES]-(job);
```

Created 49 relationships, completed after 195 ms.
- Query Results (Bottom Right):**

```
1 // Show meta-graph  
2 CALL db.schema.visualization()
```

The visualization shows a meta-graph with nodes for Employee (red), Job (yellow), Machine (green), Part (blue), Device (brown), and Assembly (orange). Relationships are shown as lines connecting these nodes, including an ASSEMBLES relationship between Assembly and Employee.

Check the meta-graph to see the two relationships



Create the PRINTS relationship type between Machine and Job nodes

The screenshot displays the Neo4j Browser interface with the following components:

- Database Information Panel (Left):**
 - Use database:** neo4j - default
 - Node Labels:** (204) Assembly, Device, Employee, Item, Job, Machine, Part
 - Relationship Types:** (148) ASSEMBLES, OPERATES, PRINTS
 - Property Keys:** assemblyID, class, deviceID, employeeID, itemID, jobID, machineID, name, partID, started, type
 - Connected as:** Username: neo4j, Roles: admin, PUBLIC, Admin: server user list, server user add, server disconnect
 - DBMS:** Version: 4.2.1, Edition: Enterprise, Name: neo4j, Databases: db, sysinfo, queries
- Query Editor (Top Right):**

```
neo4j$  
  
1 //create relationship between machine and job  
2 LOAD CSV WITH HEADERS FROM  
   'https://raw.githubusercontent.com/jyurko/CMPINF_2110_Spring_2021_data/main/hw06/machines_prints_jobs.csv' AS  
   row  
3 MATCH (machine:Machine {machineID: row.machine_id})  
4 MATCH (job:Job {jobID: row.job_id})  
5 MERGE (machine)-[:PRINTS]-(job);
```

Created 49 relationships, completed after 221 ms.
- Meta-graph Visualization (Bottom Right):**

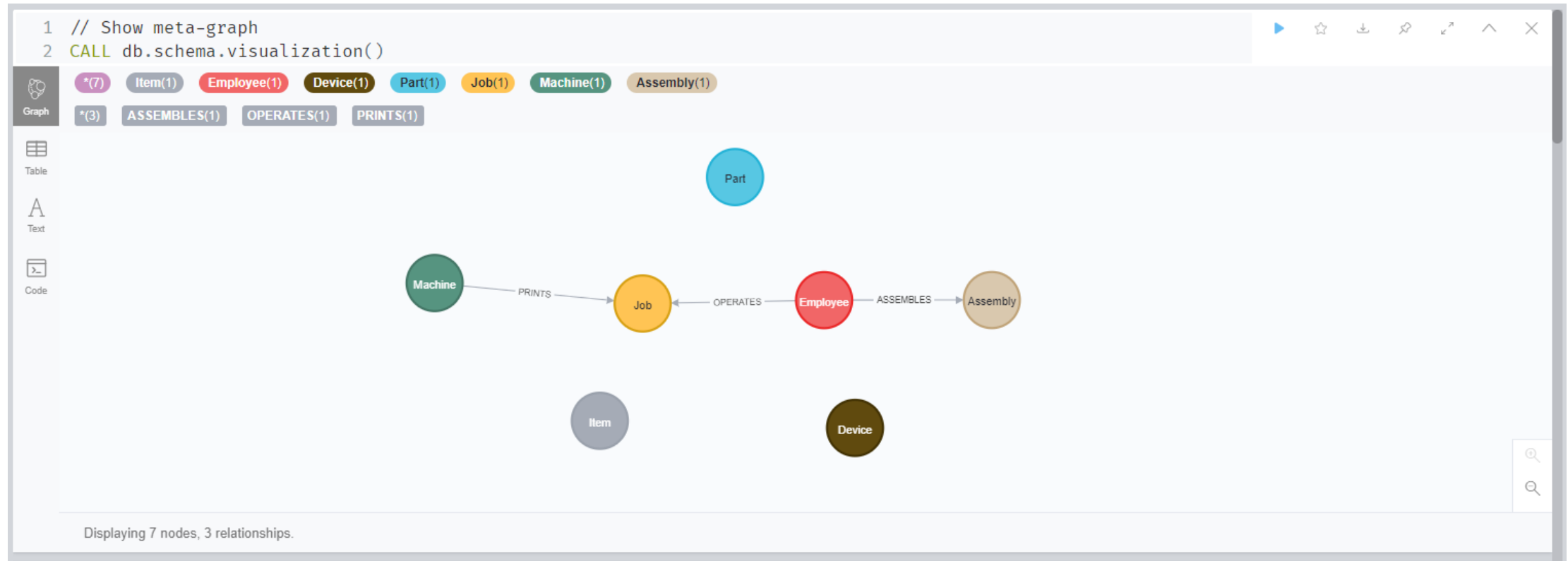
neo4j\$ // Show meta-graph CALL db.schema.visualization()

The meta-graph shows the following nodes and relationships:

 - Nodes:** Assembly (1), Employee (1), Device (1), Part (1), Job (1), Machine (1), Assembly (1)
 - Relationships:** ASSEMBLES (1), OPERATES (1)

The graph visualizes the relationships between the nodes, showing a central 'Employee' node connected to 'Assembly' and 'Job' nodes, which are further connected to 'Machine' and 'Part' nodes. The 'PRINTS' relationship type is highlighted in the meta-graph.

The meta-graph shows the 3 relationships currently in the graph database



Create the IS_IN relationship type between Item and Job nodes

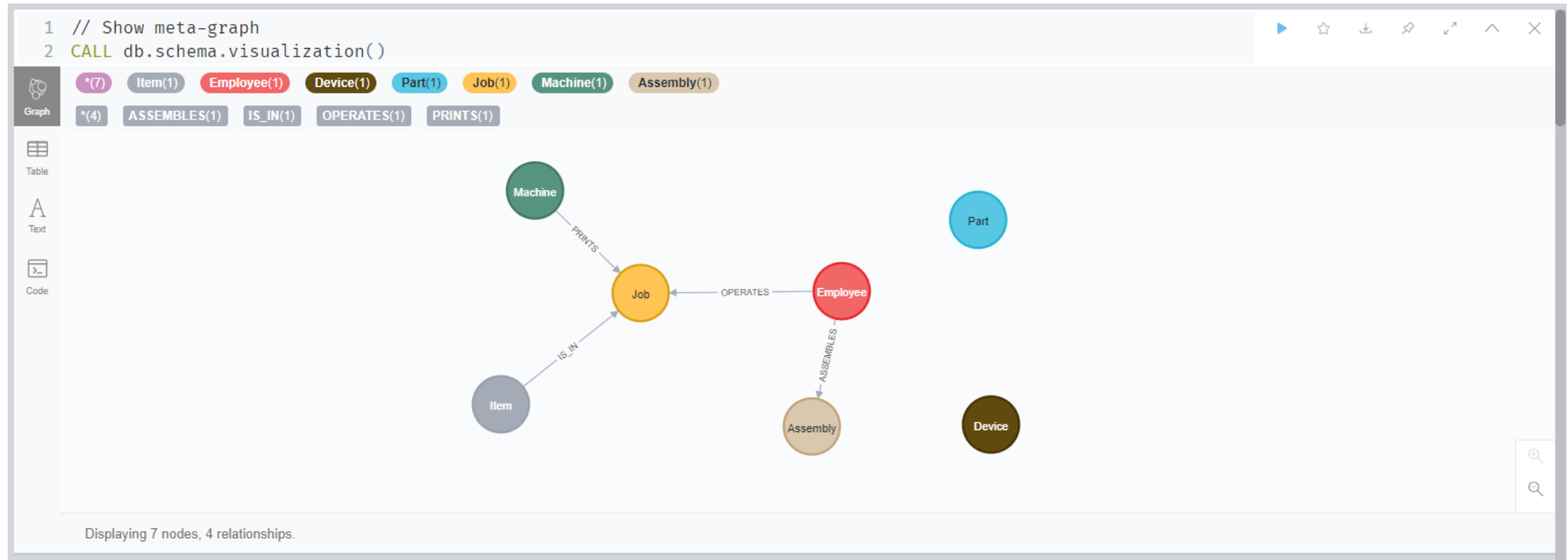
The screenshot displays the Neo4j Browser interface. On the left, the 'Database Information' sidebar shows the 'neo4j - default' database selected. Under 'Node Labels', there are 298 labels including Assembly, Device, Employee, Item, Job, Machine, and Part. Under 'Relationship Types', there are 297 types including ASSEMBLES, IS_IN, OPERATES, and PRINTS. Under 'Property Keys', there are 10 keys including assemblyID, class, deviceID, employeeID, itemID, jobID, machineID, name, partID, started, and type. The 'Connected as' section shows the user 'neo4j' with roles 'admin, PUBLIC' and various permissions. The 'DBMS' section shows version 4.2.1, edition 'Enterprise', name 'neo4j', and databases 'db, sysinfo, queries'.

The main query editor shows the following Cypher query:

```
1 //create relationship between item and job
2 LOAD CSV WITH HEADERS FROM
3   'https://raw.githubusercontent.com/jyurko/CMPINF_2110_Spring_2021_data/main/hw06/parts_in_jobs.csv' AS row
4 MATCH (item:Item {itemID: row.item_id})
5 MATCH (job:Job {jobID: row.job_id})
6 MERGE (item)-[:IS_IN]-(job);
```

The query execution results show 'Created 149 relationships, completed after 242 ms.' Below the query editor, a meta-graph visualization is shown, displaying the relationships between the nodes. The graph shows a central 'Job' node (yellow) connected to 'Machine' (green) via 'PRINTS', to 'Employee' (red) via 'OPERATES', and to 'Assembly' (brown) via 'ASSEMBLES'. The 'Employee' node is also connected to 'Assembly' via 'ASSEMBLES'. The 'Machine' node is connected to 'Job' via 'PRINTS'. The 'Job' node is connected to 'Item' (grey) via 'IS_IN'. The 'Device' (dark green) node is also present in the graph.

Meta-graph shows all the relationships associated with the Job Label



Create the IS_COMPONENT_OF relationship type between Item and Assembly nodes

The screenshot displays the Neo4j Browser interface with the following components:

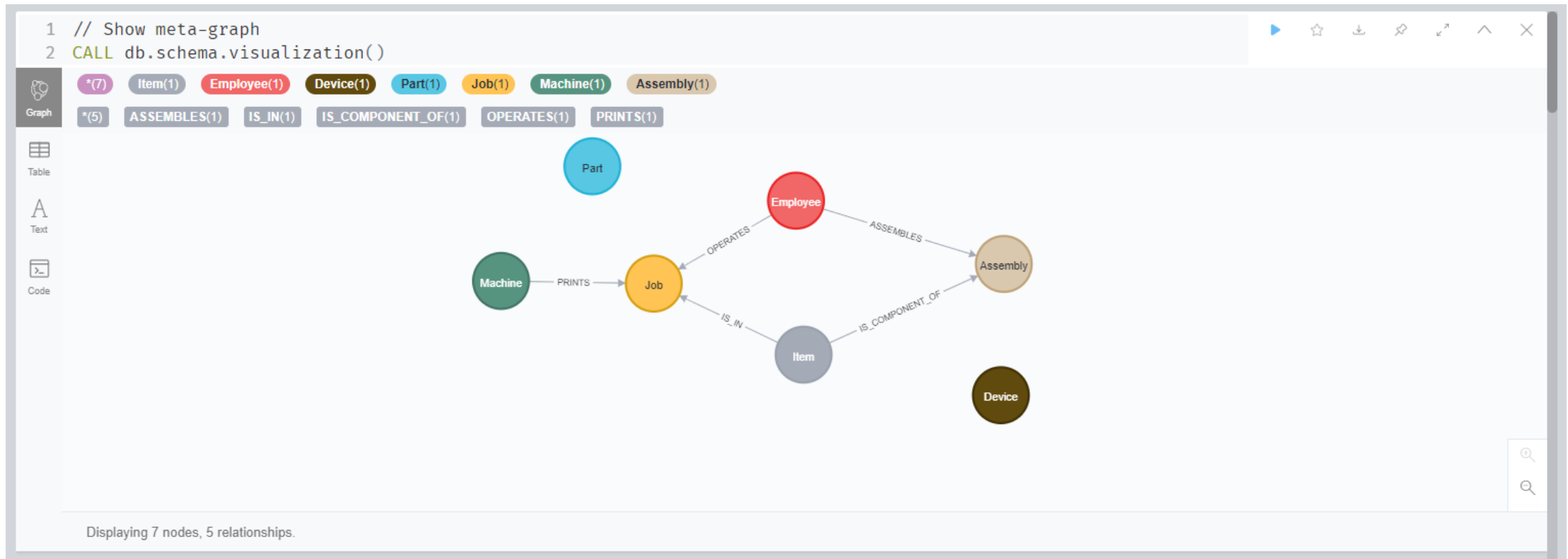
- Database Information Panel (Left):**
 - Use database:** neo4j - default
 - Node Labels:** (268) Assembly, Device, Employee, Item, Job, Machine, Part
 - Relationship Types:** (448) ASSEMBLES, IS_COMPONENT_OF, IS_IN, OPERATES, PRINTS
 - Property Keys:** assemblyID, class, deviceID, employeeID, itemID, jobID, machineID, name, partID, started, type
 - Connected as:** Username: neo4j, Roles: admin, PUBLIC, Admin: server user list, server user add, server disconnect
 - DBMS:** Version: 4.2.1, Edition: Enterprise, Name: neo4j, Databases: db
- Query Console (Top Right):**

```
neo4j$
1 //create relationship between item and assembly
2 LOAD CSV WITH HEADERS FROM
  'https://raw.githubusercontent.com/jyurko/CMPINF_2110_Spring_2021_data/main/hw06/part_componet_of_device.csv'
  AS row
3 MATCH (item:Item {itemID: row.item_id})
4 MATCH (assembly:Assembly {assemblyID: row.assembly_id})
5 MERGE (item)-[:IS_COMPONENT_OF]-(assembly);
```

Created 149 relationships, completed after 386 ms.
- Visualization (Bottom Right):**

The graph shows nodes for Machine, Job, Employee, Part, Item, Assembly, and Device. Relationships include PRINTS (Machine to Job), IS_IN (Item to Job), OPERATES (Employee to Job), and ASSEMBLES (Employee to Assembly). The IS_COMPONENT_OF relationship is highlighted in the legend.

The meta-graph shows just the Part and Device labels are not connected in the network



Create the IS_TYPE_OF relationship between the Item and Part nodes

The screenshot displays the Neo4j Browser interface. On the left, the 'Database Information' sidebar shows the database 'neo4j - default' and various node labels and relationship types. The main panel shows a Cypher query being executed:

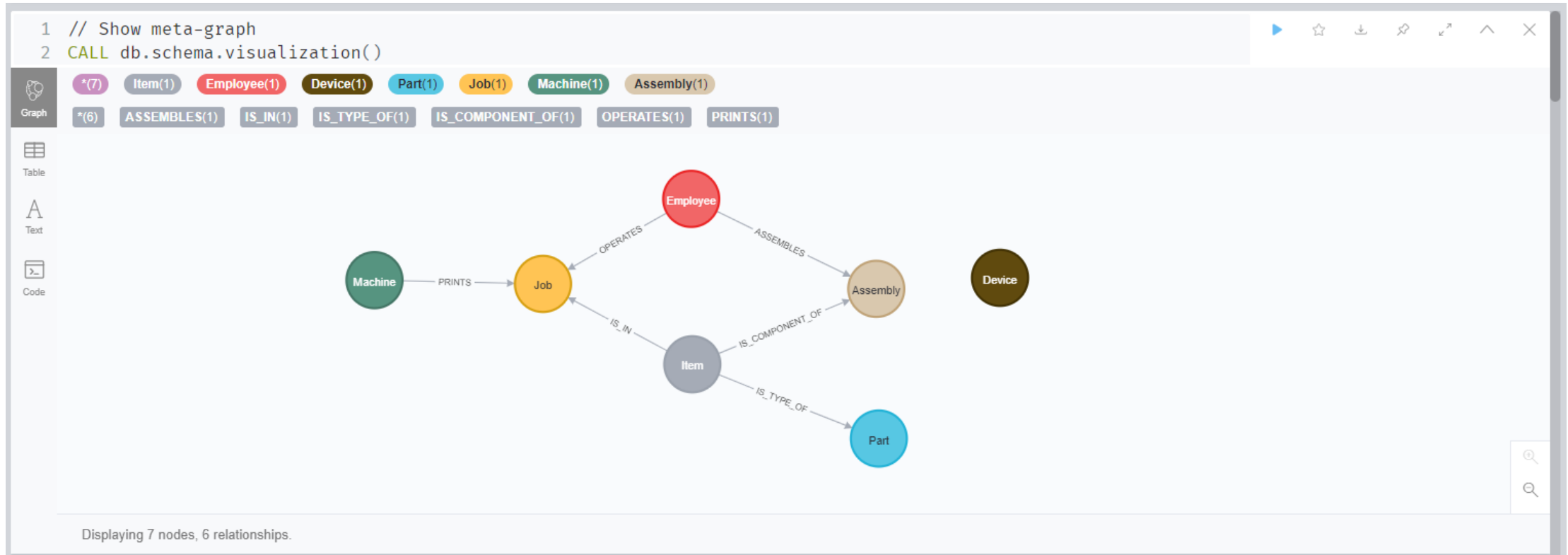
```
1 //create relationship between item and part
2 LOAD CSV WITH HEADERS FROM
3   'https://raw.githubusercontent.com/jyurko/CMPINF_2110_Spring_2021_data/main/hw06/parts_in_jobs.csv' AS row
4 MATCH (item:Item {itemID: row.item_id})
5 MATCH (part:Part {partID: row.part_id})
6 MERGE (item)-[:IS_TYPE_OF]-(part);
```

The query result shows 'Created 149 relationships, completed after 199 ms.' Below the query, a second query is executed to visualize the meta-graph:

```
1 // Show meta-graph
2 CALL db.schema.visualization();
```

The visualization shows a graph with nodes representing different entity types: Item (blue circle), Part (blue circle), Employee (red circle), Job (yellow circle), Machine (green circle), Assembly (brown circle), and Device (dark brown circle). Relationships are shown as lines connecting these nodes, with labels indicating the relationship type: ASSEMBLES, IS_IN, IS_COMPONENT_OF, OPERATES, and PRINTS. The graph illustrates the relationships between these entities, such as an Employee assembling an Assembly, a Machine printing a Job, and a Job being a component of an Assembly.

Meta-graph shows only the Device label is not connected to other Labels



Create the IS_CLASS_OF relationship between the Assembly and Device nodes

The screenshot displays the Neo4j Browser interface with the following components:

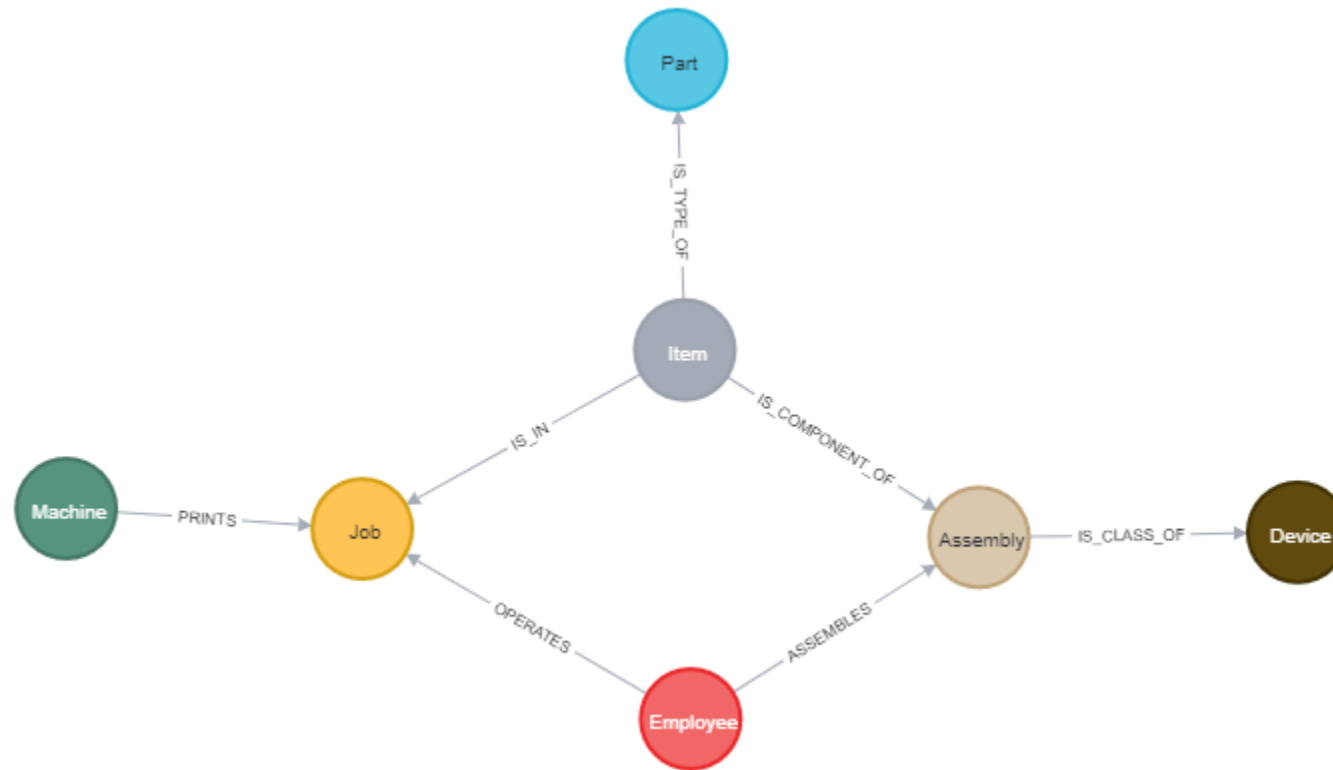
- Database Information Panel (Left):**
 - Use database:** neo4j - default
 - Node Labels:** {268} Assembly, Device, Employee, Item, Job, Machine, Part
 - Relationship Types:** {645} ASSEMBLES, IS_CLASS_OF, IS_COMPONENT_OF, IS_IN, IS_TYPE_OF, OPERATES, PRINTS
 - Property Keys:** assemblyID, class, deviceID, employeeID, itemID, jobID, machineID, name, partID, started, type
 - Connected as:** Username: neo4j, Roles: admin, PUBLIC, Admin: .server user list, .server user add, .server disconnect
 - DBMS:** Version: 4.2.1
- Code Editor (Top Right):**

```
1 //create relationship between assembly and device
2 LOAD CSV WITH HEADERS FROM
  'https://raw.githubusercontent.com/jyurko/CMPINF_2110_Spring_2021_data/main/hw06/employees_assembles_devices.csv' AS row
3 MATCH (assembly:Assembly {assemblyID: row.assembly_id})
4 MATCH (device:Device {deviceID: row.device_id})
5 MERGE (assembly)-[:IS_CLASS_OF]-(device);
```

Created 50 relationships, completed after 210 ms.
- Graph View (Bottom Right):**

Visualizes the graph structure with nodes and relationships. The nodes shown are Machine, Job, Employee, Assembly, Device, Item, and Part. The relationships shown are PRINTS, OPERATES, ASSEMBLES, IS_IN, IS_COMPONENT_OF, IS_TYPE_OF, and PRINTS.

Completed graph schema!



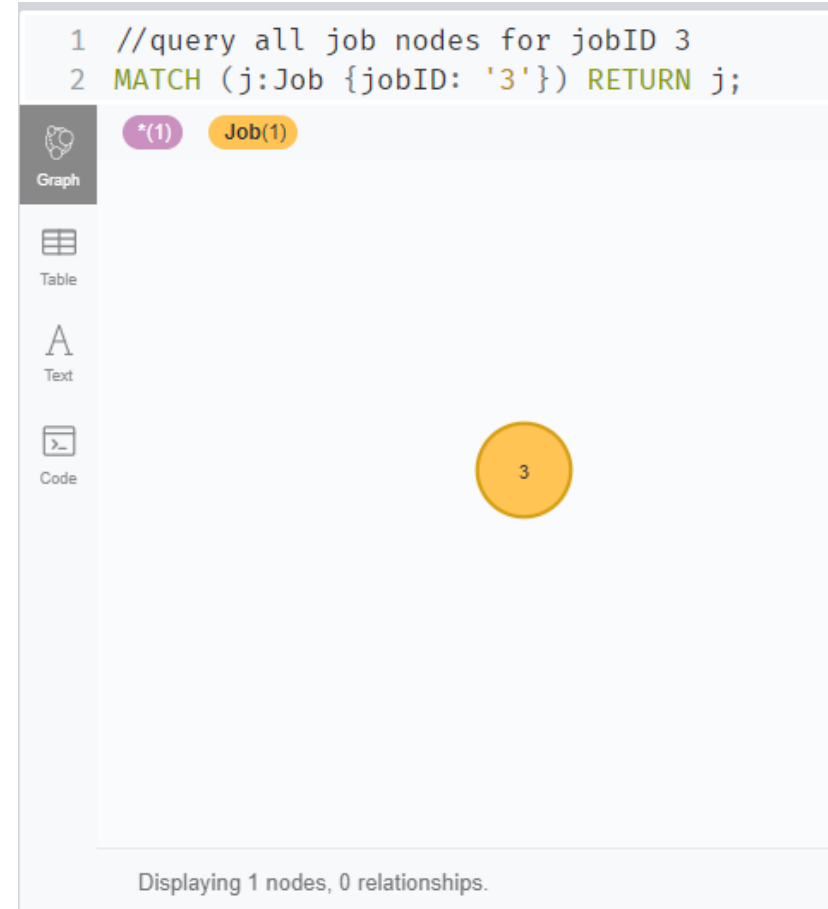
Required queries

1. Query all nodes related to the job_id = 3 Job.
2. Count all parts in job_id = 3.
3. Query all nodes related to the Employee Alice.
4. Count all devices assembled by Employee Chuck.
5. Count all parts printed by the Machine delta.

1. Query all nodes related to job_id = 3

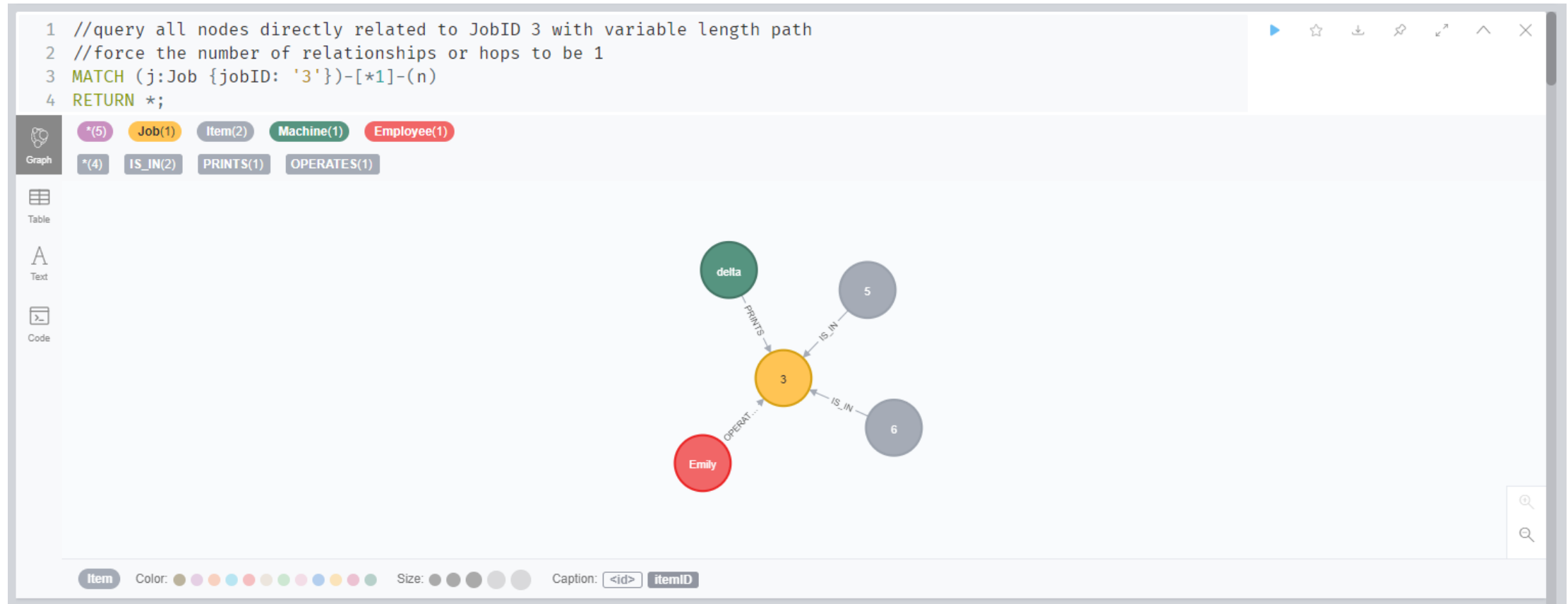
- If we only query the Job nodes associated with jobID='3', we will only get a single node!
- This is NOT what this particularly query was looking for.

```
1 //query all job nodes for jobID 3
2 MATCH (j:Job {jobID: '3'}) RETURN j;
```



The visualization shows a single yellow circular node with the number '3' inside. Above the node, there is a purple pill labeled '* (1)' and an orange pill labeled 'Job(1)'. On the left side, there is a vertical toolbar with icons for Graph, Table, Text, and Code. The 'Graph' icon is highlighted. At the bottom, a status bar reads 'Displaying 1 nodes, 0 relationships.'

A variable length path makes it easy to query any node label within 1 relationship or hop away from the Job node with jobID = '3'



However, we do not need a variable length path for this particular query. We can simply query ANY node Label related in any way to the Job node with jobID='3'

```
1 //do not really need to the variable length path for this particular
2 //query though. can just match any relationship for any node.
3 MATCH (j:Job {jobID: '3'})--(n)
4 RETURN *;
```

Graph

***(5)** Job(1) Item(2) Machine(1) Employee(1)

***(4)** IS_IN(2) PRINTS(1) OPERATES(1)

Table

Text

Code

```
graph LR
    delta((delta)) -- PRINTS --> 3((3))
    Emily((Emily)) -- OPERATES --> 3
    3 -- IS_IN --> 5((5))
    3 -- IS_IN --> 6((6))
```

Displaying 5 nodes, 4 relationships.

Table view gives the “JSON-like” representation of the query, which is saved as CSV file

```
1 //do not really need to the variable length path for this particular
2 //query though. can just match any relationship for any node.
3 MATCH (j:Job {jobID: '3'})--(n)
4 RETURN *;
```

Graph

	j	n
1	<pre>{ "identity": 22, "labels": ["Job"], "properties": { "jobID": "3" } }</pre>	<pre>{ "identity": 74, "labels": ["Item"], "properties": { "itemID": "6" } }</pre>
2	<pre>{ "identity": 22, "labels": ["Job"] }</pre>	<pre>{ "identity": 73, "labels": ["Team"] }</pre>

Started streaming 4 records after 5 ms and completed after 5 ms.

all_nodes_related_to_jobID_3 - Excel

File Home Insert Page Layout Formulas Data Review View Tell me what you want to do... Yurko, Joseph Share

Clipboard Font Alignment Number Styles Cells Editing

A2 : {"jobID":3}

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1		n												
2	["jobID":3]	["itemID":6]												
3	["jobID":3]	["itemID":5]												
4	["jobID":3]	["name":"delta","machineID":4,"type":"printer"]												
5	["jobID":3]	["name":"Emily","employeeID":5,"started":2011]												
6														
7														
8														
9														
10														
11														
12														
13														
14														

all_nodes_related_to_jobID_3

Ready

Let's check our graph query with Pandas

We first need to JOIN all necessary DataFrames together

```
In [48]: q01 = df_pij.merge( df_mpj, on='job_id', how='left' ).\
merge( df_machines.rename(columns={'name': 'machine_name', 'type': 'machine_type'}), on='machine_id', how='left').\
merge( df_employees.rename(columns={'name': 'employee_name'}), on='employee_id', how='left').\
merge( df_parts.loc[:, ['part_id', 'type']], on='part_id', how='left')
```

```
In [49]: q01
```

```
Out[49]:
```

	item_id	part_id	job_id	machine_id	employee_id	machine_name	machine_type	employee_name	started	type
0	1	4	1	2	2	bravo	printer	Bob	2018	sprocket
1	2	4	1	2	2	bravo	printer	Bob	2018	sprocket
2	3	1	2	4	4	delta	printer	Dave	2019	widget
3	4	3	2	4	4	delta	printer	Dave	2019	gadget
4	5	2	3	4	5	delta	printer	Emily	2011	gizmo
...
144	145	5	48	2	5	bravo	printer	Emily	2011	button
145	146	3	48	2	5	bravo	printer	Emily	2011	gadget
146	147	5	49	3	2	charlie	printer	Bob	2018	button
147	148	5	49	3	2	charlie	printer	Bob	2018	button
148	149	3	49	3	2	charlie	printer	Bob	2018	gadget

149 rows × 10 columns

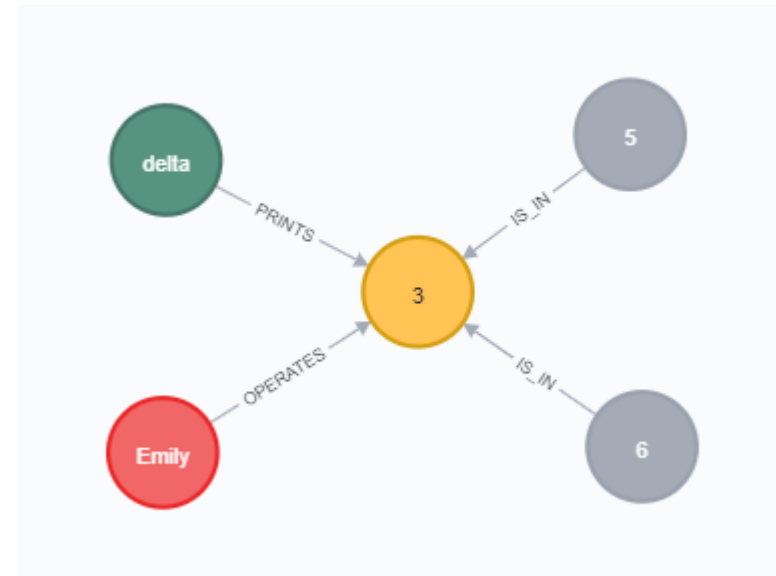
Then, filter or subset to the specific Job of interest

```
In [51]: q01.loc[ q01.job_id == 3, ['job_id', 'employee_name', 'machine_name', 'item_id']]
```

```
Out[51]:
```

	job_id	employee_name	machine_name	item_id
4	3	Emily	delta	5
5	3	Emily	delta	6

We get the SAME information
as returned by our GRAPH!



2. Count all parts in job_id = 3.

- The previous query revealed that there are 2 items printed in job_id = 3.
- We can count the number of distinct items using the DISTINCT keyword and count() functions in Cypher.

Query Job and Item nodes related in any way, return the number of unique items per job

```
1 //count all parts for job_id=3
2 //consider the number of unique Items
3 //identify the item node in the query rather than a general node
4 MATCH (j:Job {jobID: '3'})--(item:Item)
5 RETURN j.jobID, count(DISTINCT item.itemID)
```

	j.jobID	count(DISTINCT item.itemID)
1	"3"	2

Table

A
Text

Code

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

Alternatively, we could count the unique TYPES of parts for `job_id = 3`.

- Requires including the relationship between the Item node and Part node.
- But allows us to count the number of unique TYPES of parts for a given job.

We get the same result as the previous query which counted the unique number of Items

```
1 //count the unique part types for job_id=3
2 MATCH(j:Job {jobID: '3'})--(item:Item)--(part:Part)
3 RETURN j.jobID, count(DISTINCT part.type);
```

	j.jobID	count(DISTINCT part.type)
1	"3"	2

Started streaming 1 records after 7 ms and completed after 9 ms.

Confirm this is indeed the case with Pandas

As we see below, the two items are two different types of parts!

```
1 //count the unique part types for job_id=3
2 MATCH(j:Job {jobID: '3'})--(item:Item)--(part:Part)
3 RETURN j.jobID, count(DISTINCT part.type);
```

	j.jobID	count(DISTINCT part.type)
1	"3"	2

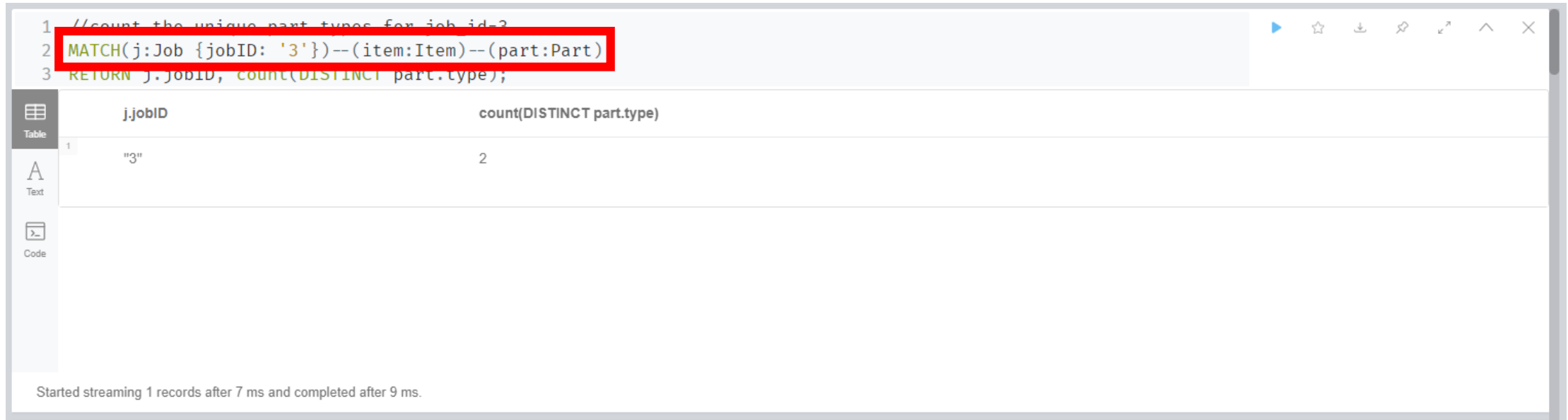
```
In [52]: q01.loc[ q01.job_id == 3, ['job_id', 'employee_name', 'machine_name', 'item_id', 'type']]
```

```
Out[52]:
```

	job_id	employee_name	machine_name	item_id	type
4	3	Emily	delta	5	gizmo
5	3	Emily	delta	6	sprocket

If we do not care about the directionality of the relationship, we do not need to specify it

No arrows are included in the pattern!



The screenshot shows a query editor with three lines of Cypher code. The second line, `MATCH(j:Job {jobID: '3'})--(item:Item)--(part:Part)`, is highlighted with a red box. Below the editor is a table with two columns: `j.jobID` and `count(DISTINCT part.type)`. The table contains one row with the values `"3"` and `2`. A status bar at the bottom indicates that streaming started after 7 ms and completed after 9 ms.

```
1 //count the unique part types for job id=3
2 MATCH(j:Job {jobID: '3'})--(item:Item)--(part:Part)
3 RETURN j.jobID, count(DISTINCT part.type);
```

j.jobID	count(DISTINCT part.type)
"3"	2

Started streaming 1 records after 7 ms and completed after 9 ms.

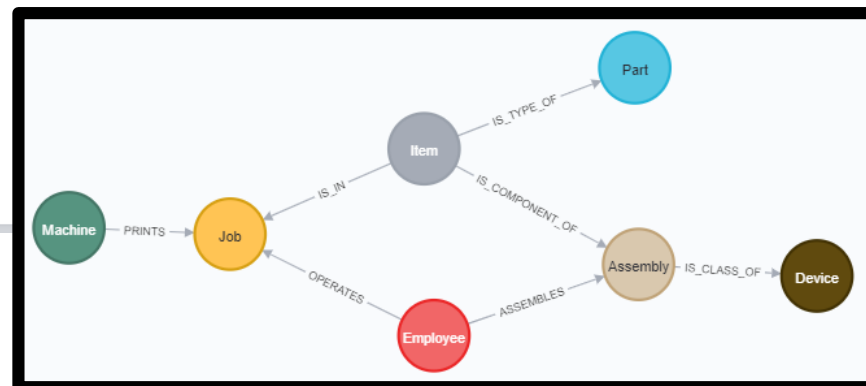
If we want a specific directional relationship, we must be sure that is the direction embedded in the graph

For example, if we give the WRONG directional relationship no results will be returned!

```
1 //count the unique part types for job_id=3
2 //what happens if we include the wrong directions?
3 MATCH(j:Job {jobID: '3'})→(item:Item)→(part:Part)
4 RETURN j.jobID, count(DISTINCT part.type);
```

(no changes, no records)

The query shown above is incorrect because the directionality is Item → Job NOT Job → Item!
As shown by the meta-graph below.



Completed after 7 ms.

The correct directional query has the arrows pointing away from the Item node

```
1 //count the unique part types for job_id=3
2 //what happens if we include the correct direction?
3 MATCH(j:Job {jobID: '3'})←(item:Item)→(part:Part)
4 RETURN j.jobID, count(DISTINCT part.type);
```

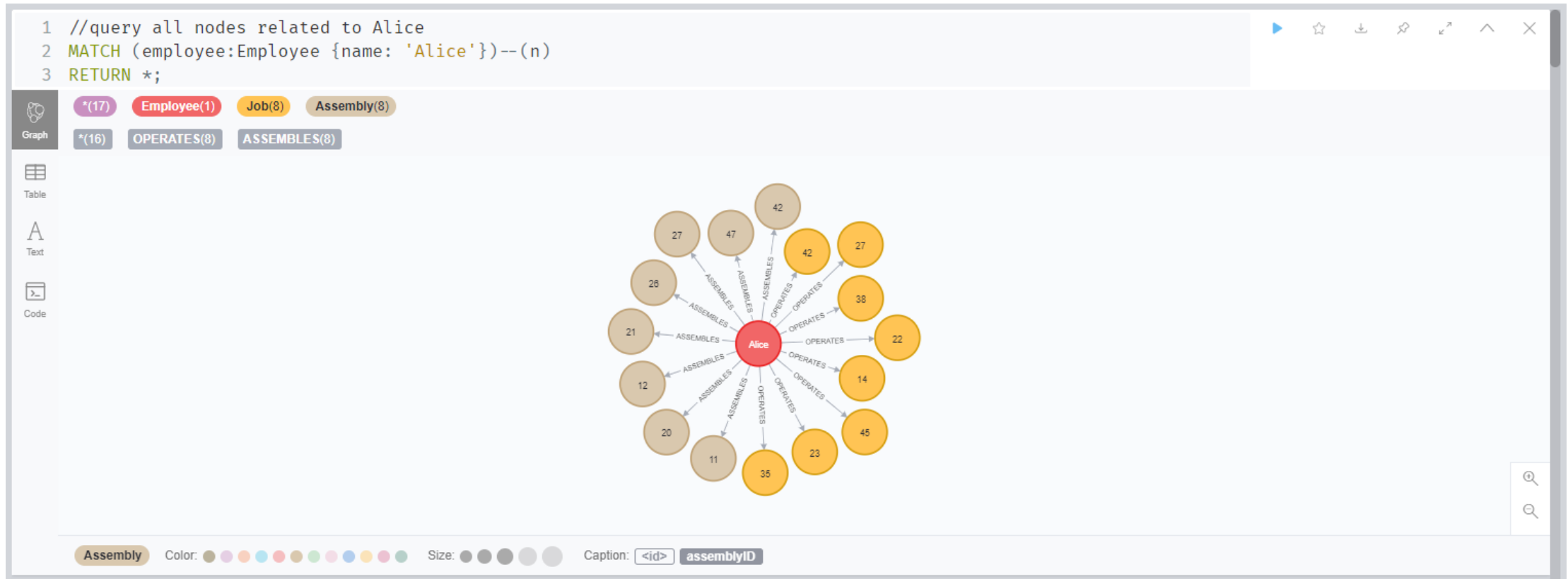
	j.jobID	count(DISTINCT part.type)
1	"3"	2

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

3. Query all nodes related to the Employee Alice.

- This query can be structured multiple ways.

One way is to specify any node of any relationship type with the Employee node



There are 8 Jobs and 8 Assemblies connected with Alice.

To check that this is correct, let's use Pandas to count the number of Jobs which Alice was the machine operator.

```
In [59]: q03_a = df_mpj.merge( df_employees, on='employee_id', how='left' )
```

```
In [60]: q03_a.loc[ q03_a.name == 'Alice', :]
```

Out[60]:

	job_id	machine_id	employee_id	name	started
13	14	4	1	Alice	2017
21	22	3	1	Alice	2017
22	23	3	1	Alice	2017
26	27	3	1	Alice	2017
34	35	1	1	Alice	2017
37	38	2	1	Alice	2017
41	42	2	1	Alice	2017
44	45	3	1	Alice	2017

```
In [62]: q03_a.loc[ q03_a.name == 'Alice', :].shape[0]
```

Out[62]: 8

Yes! There are 8 jobs!

```
In [63]: q03_b = df_ead.merge( df_employees, on='employee_id', how='left' )
```

```
In [64]: q03_b.loc[ q03_b.name == 'Alice', :]
```

Out[64]:

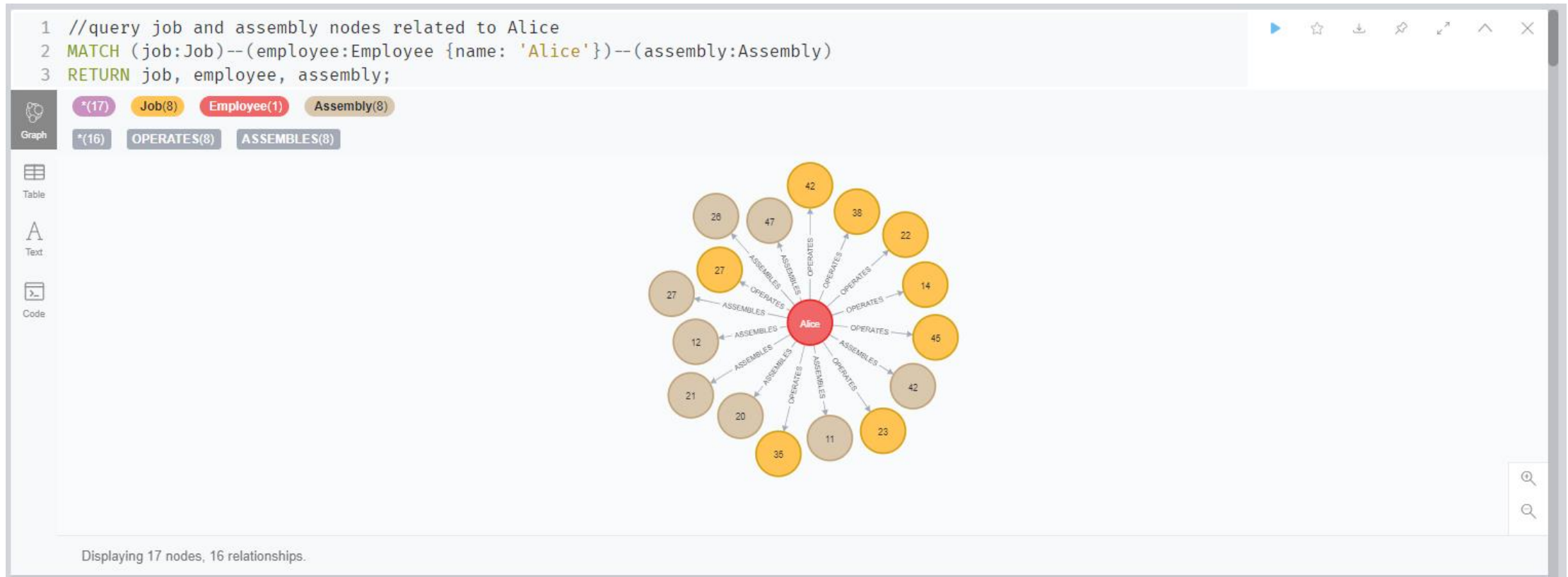
	assembly_id	device_id	employee_id	name	started
10	11	2	1	Alice	2017
11	12	2	1	Alice	2017
19	20	2	1	Alice	2017
20	21	4	1	Alice	2017
25	26	1	1	Alice	2017
26	27	1	1	Alice	2017
41	42	2	1	Alice	2017
46	47	1	1	Alice	2017

```
In [65]: q03_b.loc[ q03_b.name == 'Alice', :].shape[0]
```

Out[65]: 8

Yes! There are 8 assemblies!

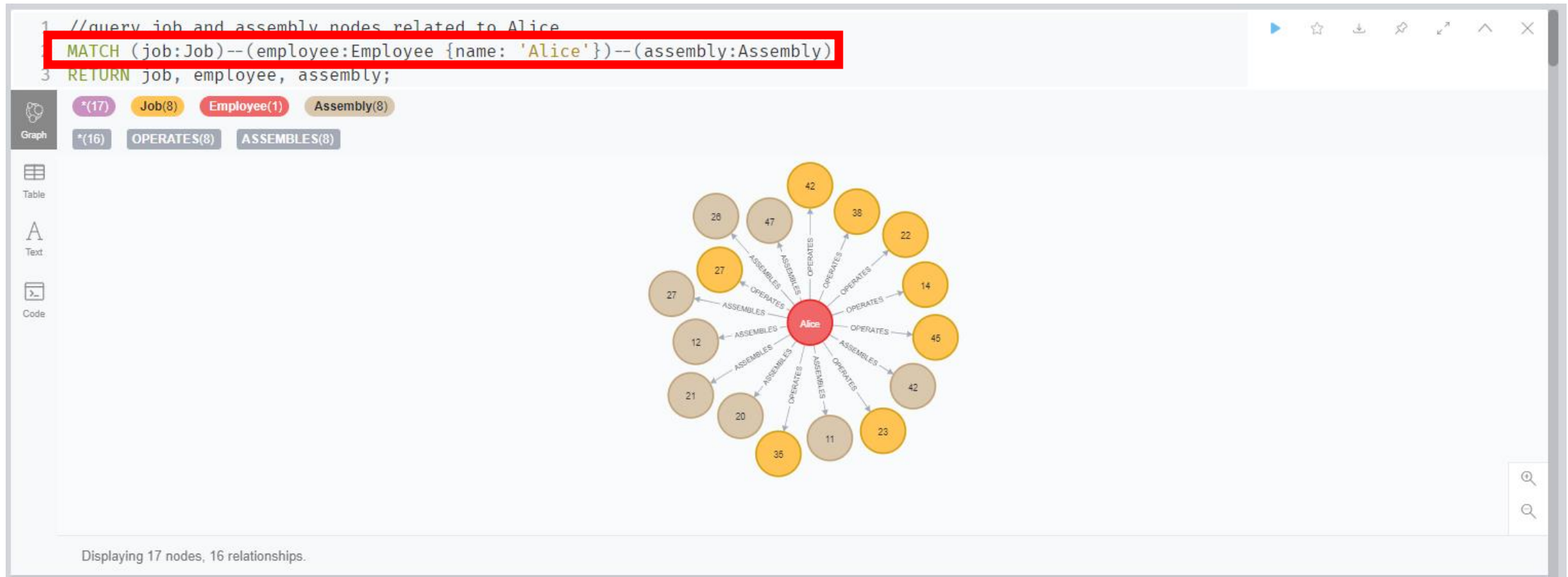
Alternatively, we can perform the same query by providing the specific node Labels we are interested in.



The number of nodes and relationships returned are the same as the previous query. Though the graphical ordering of the nodes is slightly different.

Alternatively, we can perform the same query by providing the specific node Labels we are interested in.

Easier to structure this query pattern with the Employee node in the middle.

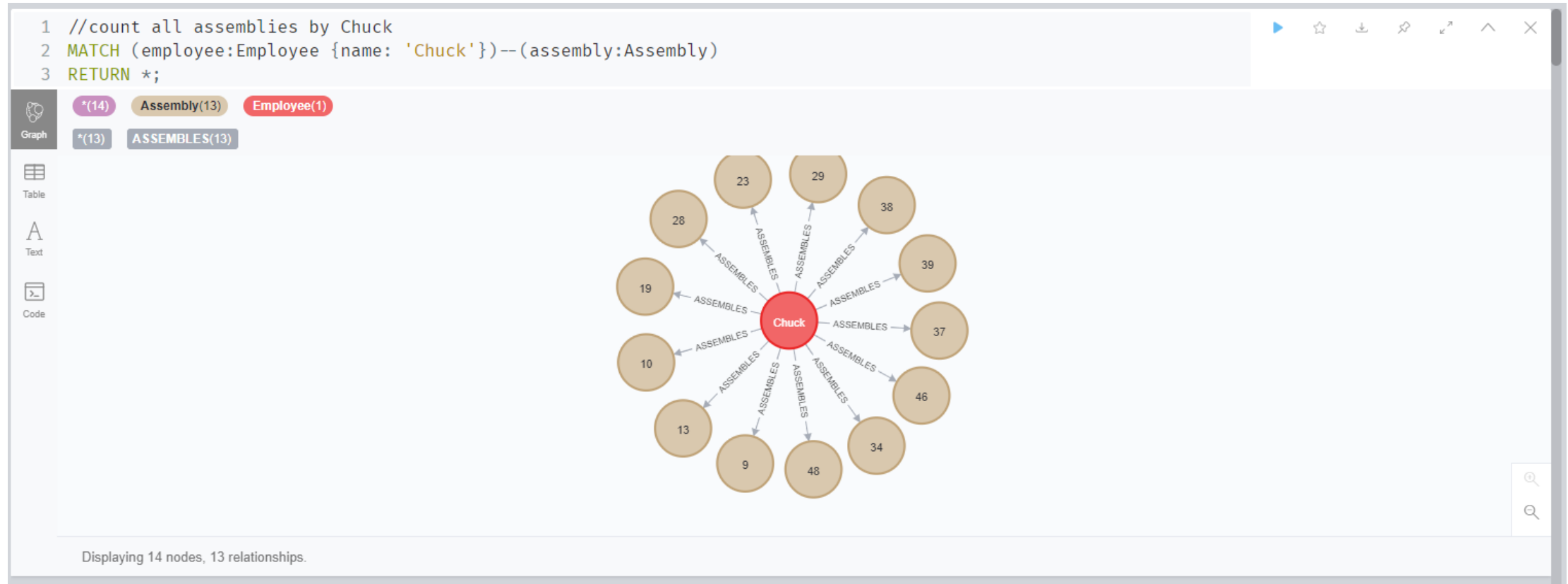


The number of nodes and relationships returned are the same as the previous query. Though the graphical ordering of the nodes is slightly different.

4. Count all devices assembled by Employee Chuck

- We can count the number of assemblies by querying all Assemblies related to the Employee node with name='Chuck'.
- If we want to also count the unique number of device classes, we will need to relate the Assembly node to the Device node.
- The question itself was open ended, you could have answered either way!

Querying all Assembly nodes related to Chuck returns 13 Assembly nodes



Perform the grouping and aggregation operation with the count() function

```
1 //count all assemblies by Chuck
2 MATCH (employee:Employee {name: 'Chuck'})--(assembly:Assembly)
3 RETURN employee.name, count(assembly.assemblyID);
```

	employee.name	count(assembly.assemblyID)
1	"Chuck"	13

Started streaming 1 records after 6 ms and completed after 8 ms.

We get the same count we could visually confirm from the previous query.

Confirm by counting the rows in Pandas

```
1 //count all assemblies by Chuck
2 MATCH (employee:Employee {name: 'Chuck'})--(assembly:Assembly)
3 RETURN employee.name, count(assembly.assemblyID);
```

	employee.name	count(assembly.assemblyID)
1	"Chuck"	13

```
In [67]: q03_b.loc[ q03_b.name == 'Chuck', :].shape[0]
Out[67]: 13
```

Started streaming 1 records after 6 ms and completed after 8 ms.

Or by grouping and aggregating

```
1 //count all assemblies by Chuck
2 MATCH (employee:Employee {name: 'Chuck'})--(assembly:Assembly)
3 RETURN employee.name, count(assembly.assemblyID);
```

employee.name	count(assembly.assemblyID)
"Chuck"	13

Started streaming 1 records after 6 ms and completed after

In [70]:

```
q03_b.groupby(['name']).\
  aggregate(num_assemblies = ('assembly_id', 'nunique')).\
  reset_index()
```

Out[70]:

	name	num_assemblies
0	Alice	8
1	Bob	10
2	Chuck	13
3	Dave	8
4	Emily	11

To count the number of unique Device classes assembled by Chuck:

- We need to include the relationship between the Assembly node and the Device node.
- Apply the count() function to the DISTINCT device.class property.

Notice that the pattern includes 3 nodes and 2 relationships now

```
1 //count all assemblies and unique devices by Chuck
2 MATCH (employee:Employee {name: 'Chuck'})--(assembly:Assembly)--(device:Device)
3 RETURN employee.name, count(assembly.assemblyID), count(DISTINCT device.class);
```

	employee.name	count(assembly.assemblyID)	count(DISTINCT device.class)
1	"Chuck"	13	4

Started streaming 1 records after 12 ms and completed after 14 ms.

Confirm by grouping and aggregating in Pandas

```
1 //count all assemblies and unique devices by Chuck
2 MATCH (employee:Employee {name: 'Chuck'})--(assembly:Assembly)--(device:Device)
3 RETURN employee.name, count(assembly.assemblyID), count(DISTINCT device.class);
```

	employee.name	count(assembly.assemblyID)	count(DISTINCT device.class)
1	"Chuck"	13	4

Started streaming 1 records after 12 ms and completed after 14 ms.

In [71]:

```
q03_b.groupby(['name']).\
  aggregate(num_assemblies = ('assembly_id', 'nunique'),
            num_device_classes = ('device_id', 'nunique')).\
  reset_index()
```

Out[71]:

	name	num_assemblies	num_device_classes
0	Alice	8	3
1	Bob	10	4
2	Chuck	13	4
3	Dave	8	4
4	Emily	11	4

Although not required, we could count the number of Assemblies and unique Device classes for every employee in Cypher

```
1 //count all assemblies and unique devices by each employee
2 //not required to do in the question
3 MATCH (employee:Employee)--(assembly:Assembly)--(device:Device)
4 RETURN employee.name, count(assembly.assemblyID), count(DISTINCT device.class)
5 ORDER BY employee.name;
```

	employee.name	count(assembly.assemblyID)	count(DISTINCT device.class)
1	"Alice"	8	3
2	"Bob"	10	4
3	"Chuck"	13	4
4	"Dave"	8	4
5	"Emily"	11	4

Started streaming 5 records after 5 ms and completed after 32 ms.

The Cypher query gives the same results as the Pandas merge(), groupby(), and aggregate() series of statements. Analogous to joining, grouping, and summarizing in MySQL.

```
1 //count all assemblies and unique devices by each employee
2 //not required to do in the question
3 MATCH (employee:Employee)--(assembly:Assembly)--(device:Device)
4 RETURN employee.name, count(assembly.assemblyID), count(DISTINCT device.class)
5 ORDER BY employee.name;
```

	employee.name	count(assembly.assemblyID)	count(DISTINCT device.class)
1	"Alice"	8	3
2	"Bob"	10	4
3	"Chuck"	13	4
4	"Dave"	8	4
5	"Emily"	11	4

Started streaming 5 records after 5 ms and completed after 32 ms.

```
In [72]: df_ead.merge( df_employees, on='employee_id', how='left' ).\
groupby(['name']).\
aggregate(num_assemblies = ('assembly_id', 'nunique'),
          num_device_classes = ('device_id', 'nunique')).\
reset_index()
```

Out[72]:

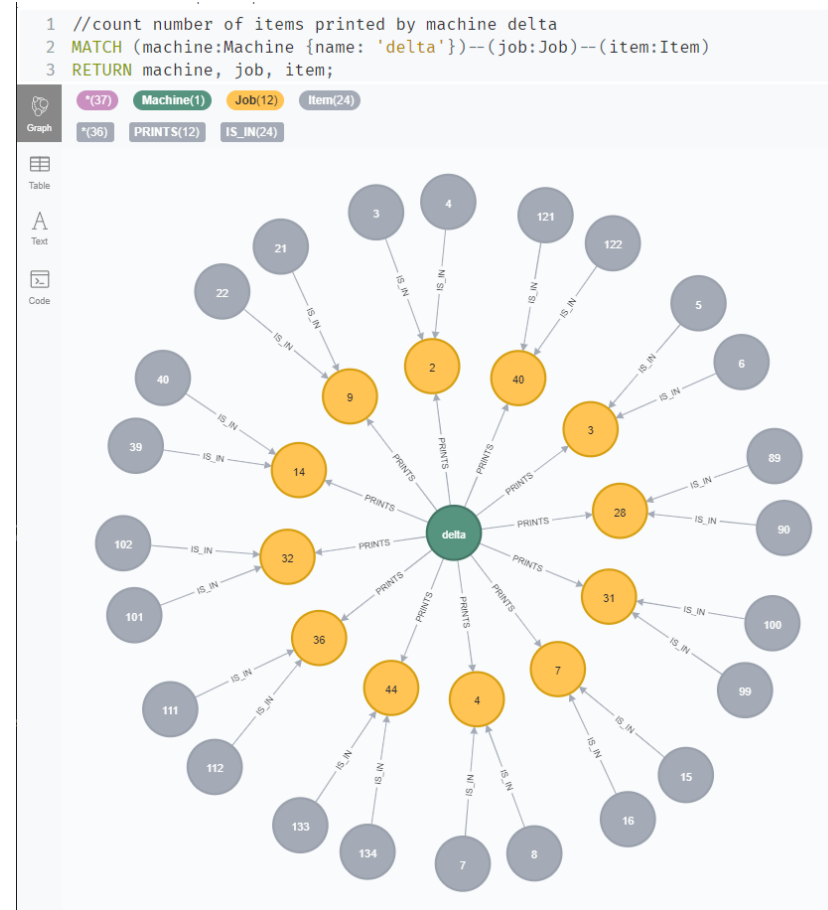
	name	num_assemblies	num_device_classes
0	Alice	8	3
1	Bob	10	4
2	Chuck	13	4
3	Dave	8	4
4	Emily	11	4

5. Count all parts printed by the Machine delta.

- Question is open ended, so could count the unique number of Items printed OR the unique number of Part types.

Although not required, the network for all Jobs printed by Machine delta and all items in those Jobs

The graph tells us
there are 12 Jobs
and 24 Items.



Query the number of Items by grouping and summarizing via the count() function

```
1 //count number of items printed by machine delta
2 MATCH (machine:Machine {name: 'delta'})--(job:Job)--(item:Item)
3 RETURN machine.name, count(DISTINCT job.jobID), count(DISTINCT item.itemID);
```

	machine.name	count(DISTINCT job.jobID)	count(DISTINCT item.itemID)
1	"delta"	12	24

Started streaming 1 records after 7 ms and completed after 9 ms.

The number of unique Jobs is included, but was not required for the question.

If we also want to query the number of unique Part types printed by Machine delta, we need to include the Item to Part relationship and apply the count() function appropriately

```
1 //count number of items and unique part types printed by machine delta
2 MATCH (machine:Machine {name: 'delta'})--(job:Job)--(item:Item)--(part:Part)
3 RETURN machine.name, count(DISTINCT job.jobID), count(DISTINCT item.itemID), count(DISTINCT part.type);
```

	machine.name	count(DISTINCT job.jobID)	count(DISTINCT item.itemID)	count(DISTINCT part.type)
1	"delta"	12	24	7

Started streaming 1 records after 9 ms and completed after 10 ms.

Although not required, it's straight forward to repeat the query for all machines not just Machine delta

```
1 //count number of items and unique part types printed for all machines
2 MATCH (machine:Machine)--(job:Job)--(item:Item)--(part:Part)
3 RETURN machine.name, count(DISTINCT job.jobID), count(DISTINCT item.itemID), count(DISTINCT part.type)
4 ORDER BY machine.name;
```

	machine.name	count(DISTINCT job.jobID)	count(DISTINCT item.itemID)	count(DISTINCT part.type)
1	"alpha"	12	48	7
2	"bravo"	11	22	5
3	"charlie"	14	55	7
4	"delta"	12	24	7

Started streaming 4 records after 8 ms and completed after 15 ms.

Check the results with Pandas

```
1 //count number of items and unique part types printed for all machines
2 MATCH (machine:Machine)--(job:Job)--(item:Item)--(part:Part)
3 RETURN machine.name, count(DISTINCT job.jobID), count(DISTINCT item.itemID), count(DISTINCT part.type)
4 ORDER BY machine.name;
```

	machine.name	count(DISTINCT job.jobID)	count(DISTINCT item.itemID)	count(DISTINCT part.type)
1	"alpha"	12	48	7
2	"bravo"	11	22	5
3	"charlie"	14	55	7
4	"delta"	12	24	7

Started streaming 4 records after 8 ms and co

```
In [80]: df_pij.merge( df_mpj, on='job_id', how='left').\
merge( df_machines.loc[:, ['machine_id', 'name']], on='machine_id', how='left').\
merge( df_parts, on='part_id', how='left').\
groupby(['name']).\
aggregate(num_jobs = ('job_id', 'nunique'),
          num_items = ('item_id', 'nunique'),
          num_part_types = ('type', 'nunique')).\
reset_index()
```

Out[80]:

	name	num_jobs	num_items	num_part_types
0	alpha	12	48	7
1	bravo	11	22	5
2	charlie	14	55	7
3	delta	12	24	7