Governors State University
OPUS Open Portal to University Scholarship

All Capstone Projects

Student Capstone Projects

Fall 2015

Graph Database

George Dovgin
Governors State University

GRAPH DATABASE

By

George Dovgin

B.S., Southern Illinois University, Edwardsville, 1992

GRADUATE PROJECT

Submitted in partial fulfillment of the requirements

For the Degree of Master of Science,

With a Major in Computer Science

Governors State University

University Park, IL 60484

2015

Abstract

This project will review the new technology of graph databases. Graph databases, which model data using nodes and relationships, utilize a different paradigm than the rows and columns of relational databases.

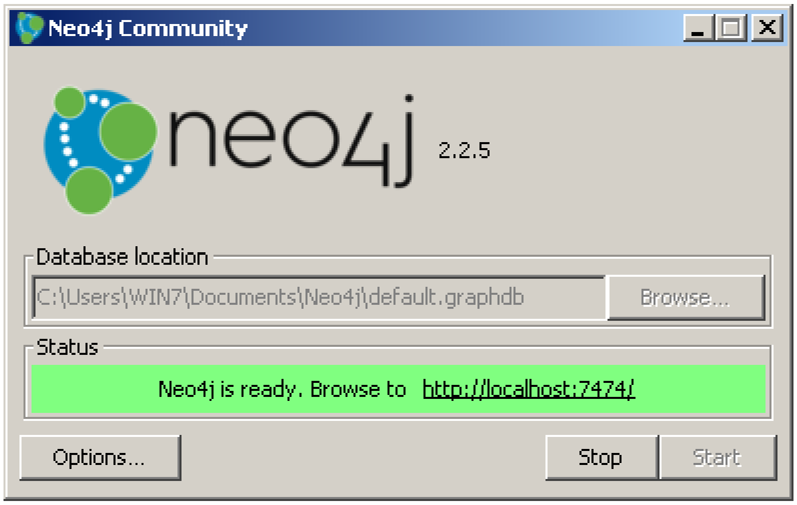
The main goals of this project are to provide the basic background information on graph database technology and then use this knowledge to convert an RDBMS into a GDBMS. The RDBMS used will be the sample Accounts Payable (AP) relational database used in the Murach SQL 2012 book. The following will be accomplished:

- Explore graph database versus relational for querying and updating the Accounts Payable database. Review Cypher (Neo4j graph query language) and run CRUD queries against the AP graph database.
- Show step by step instructions to convert the Murach SQL 2012 Accounts Payable database into the graph database.

Getting Started

Setup a graph database

The graphs database used for this project is called Neo4j.   It's java based so be sure to download the Java Development Kit (JDK). The community edition is open sourced and can be downloaded for free. There is also an advanced server edition too for commercial purposes. Just visit the website called http://www.neo4j.com and select download. After installing, run the startup application program.   Note: The real application program is a browser-based application.



Database location: C:\Users\WIN7\Documents\Neo4j\default.graphdb

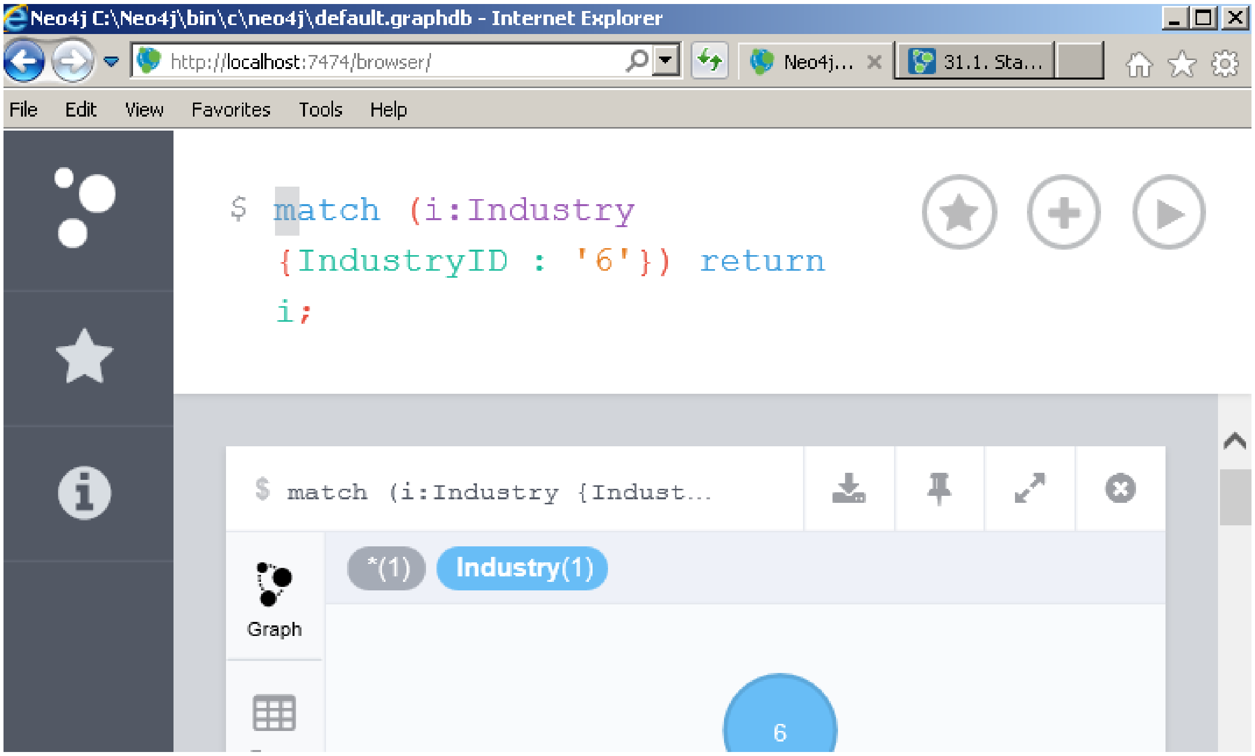
Status: Neo4j is ready. Browse to http://localhost:7474/

Figure 1 Main Window for Neo4j

The database server will be in a stopped stated because the default database has not been created. Press start and the database will be created for you in the database location. The database status will then provide a hyperlink to browser-based application. Press the hyperlink and it will launch the browser application.

Running Neo4j in the Browser

Most work will be accomplished in the extremely functional browser application. Here is where queries are run, files are imported and most importantly the graph database nodes and relationships are displayed.

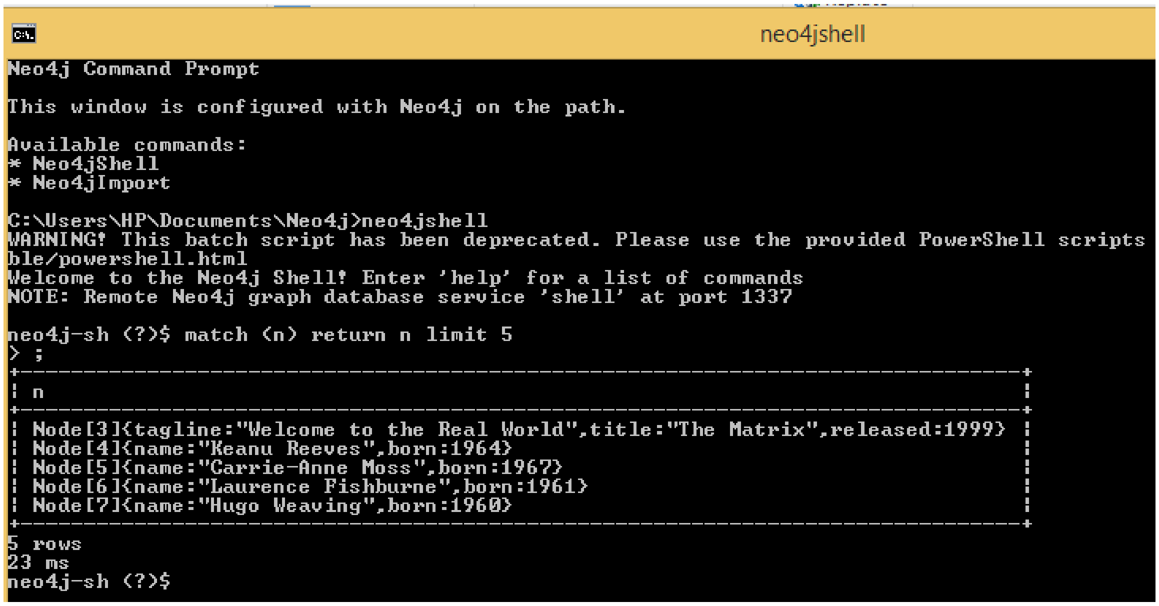


Neo4j C:\Neo4j\bin\c\neo4j\default.graphdb - Internet Explorer
http://localhost:7474/browser/
Neo4j...
31.1. Sta...
File
Edit
View
Favorites
Tools
Help
$ match (i:Industry {Indust...
*(1)
Industry(1)
Graph
6

```
$ match (i:Industry
  {IndustryID : '6'}) return
  i;
```

Running Neo4j at the Console

For some, the console is preferred. To launch the console, go the Main Window for Neo4j and press the Options button. Then press the Command Prompt button.



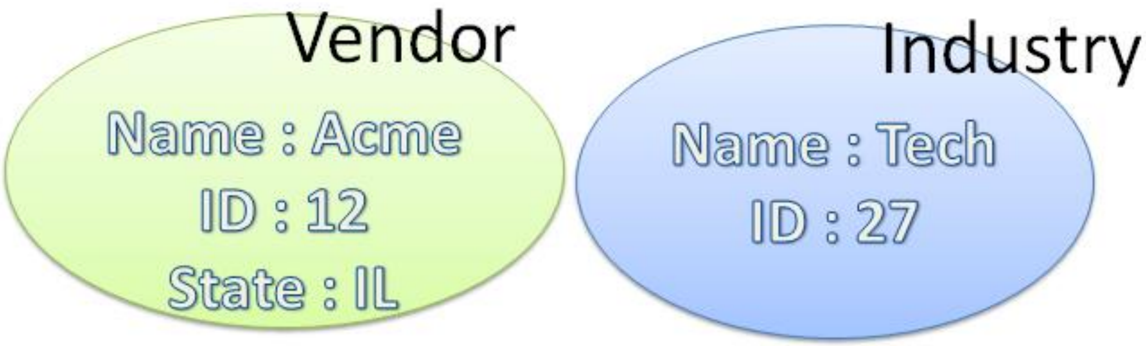
```
Neo4j Command Prompt

This window is configured with Neo4j on the path.

Available commands:
* Neo4jShell
* Neo4jImport

C:\Users\HP\Documents\Neo4j>neo4jshell
WARNING! This batch script has been deprecated. Please use the provided PowerShell scripts
ble/powershell.html
Welcome to the Neo4j Shell! Enter 'help' for a list of commands
NOTE: Remote Neo4j graph database service 'shell' at port 1337

neo4j-sh (?)$ match (n) return n limit 5
> ;
+--------------------------------------------------------------------------------+
| n                                                                              |
+--------------------------------------------------------------------------------+
| Node[3]{tagline:"Welcome to the Real World",title:"The Matrix",released:1999} |
| Node[4]{name:"Keanu Reeves",born:1964}                                         |
| Node[5]{name:"Carrie-Anne Moss",born:1967}                                     |
| Node[6]{name:"Laurence Fishburne",born:1961}                                   |
| Node[7]{name:"Hugo Weaving",born:1960}                                         |
+--------------------------------------------------------------------------------+
5 rows
23 ms
neo4j-sh (?)$
```

Graph Database Background

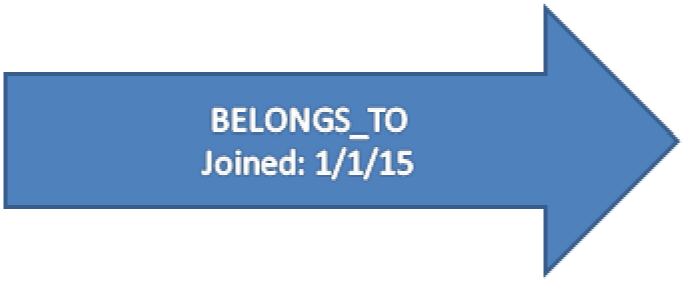
Graph nodes and relationships

Data structured in graphs rather than in tables represent a paradigm shift for connection data. Graphs use node and relationships, not rows and columns.

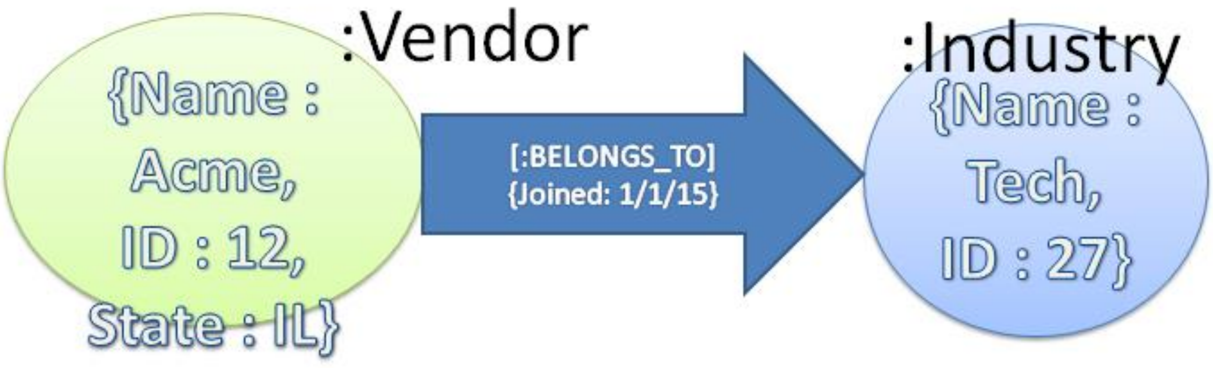
Nodes are like rows in a table. A node has a type (like a table name in relational) and some properties (key/value pairs), like name, id, and state. See the figure below for a graphical representation of nodes.



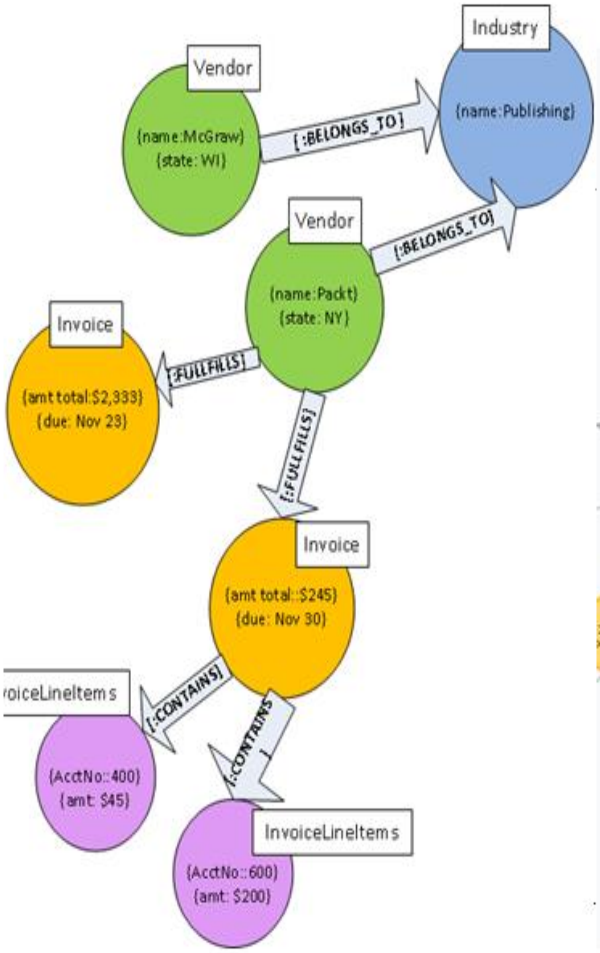
Relationships are used to connect nodes together. Just like nodes, relationships can have properties (key/value pairs) too. This is in contrast to RDBMS that use foreign keys and join table to connect data. In the figure below, this relationship has one property called Joined with a value of 1/1/15.

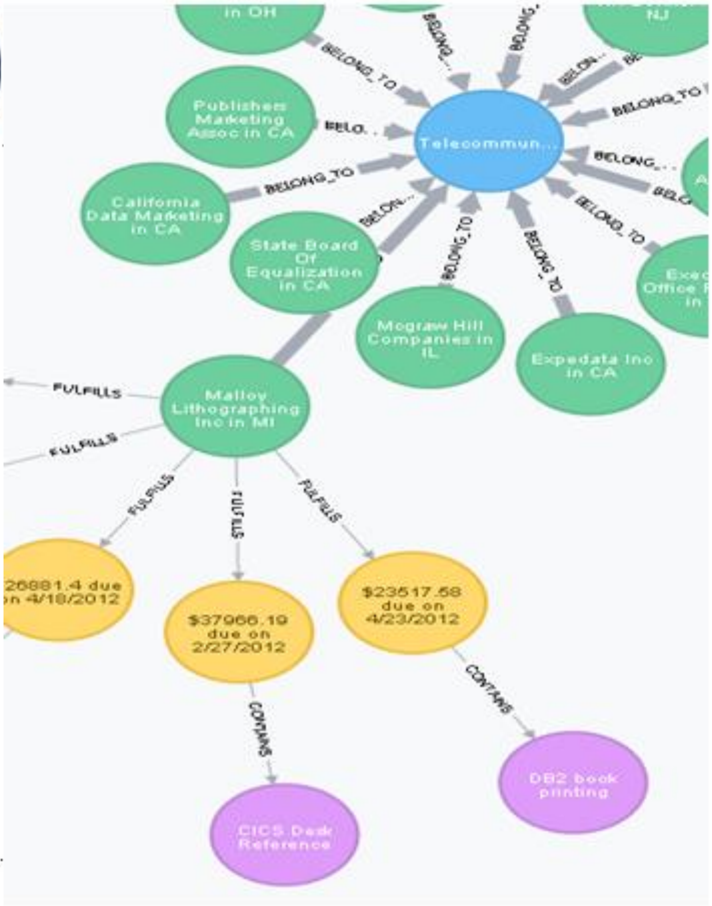


Putting the two key components together, we can say that node (a) Vendor "belongs to" node (b) Industries. See figure below.



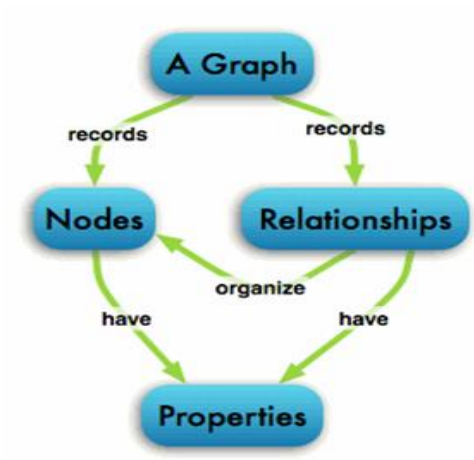
One advantage when drawing nodes and relationships into more complex data models is that the actual graph database almost exactly follows the whiteboard model. In the example below for the AP database, the left side is a drawing done in Microsoft Visio whereas the right side is an actual graph database in Neo4j. See figure below.



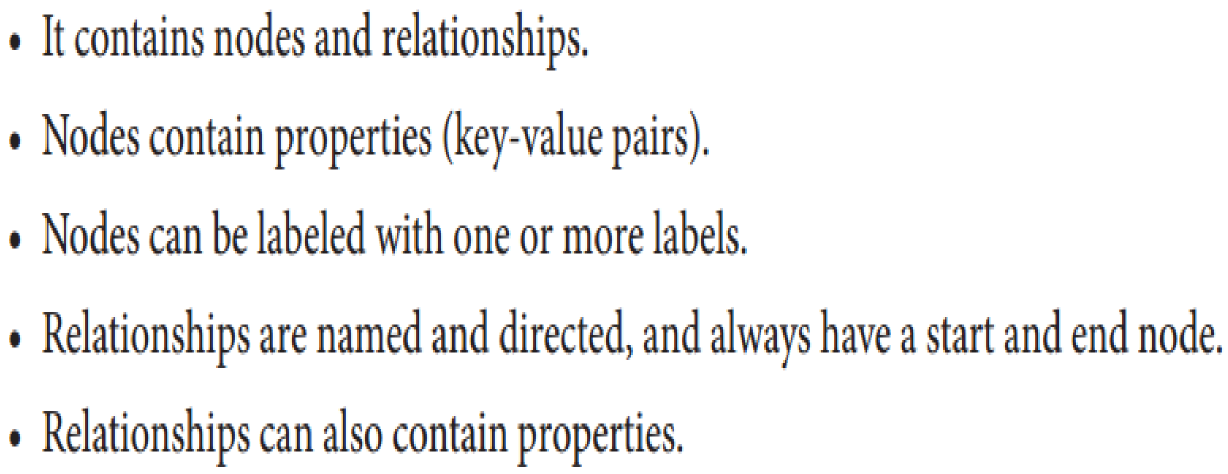


As you can see the white board drawing is almost identical to the actual working graph database!

Lastly to provide insights into the powerful expressiveness of graph databases, the figure below is graph of a graph database structure. A graph records nodes and relationships. Nodes and relationships have properties and relationships organize nodes.



To wrap up the rules for nodes and relationships, Ian Robinson and James Webber in Graph Databases, 2nd Ed summarize a graph database succinctly:

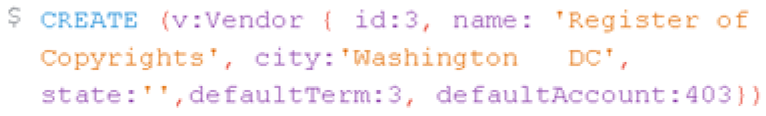


- It contains nodes and relationships.
- Nodes contain properties (key-value pairs).
- Nodes can be labeled with one or more labels.
- Relationships are named and directed, and always have a start and end node.
- Relationships can also contain properties.

CRUD – Create, Retrieve, Update, Delete

Create

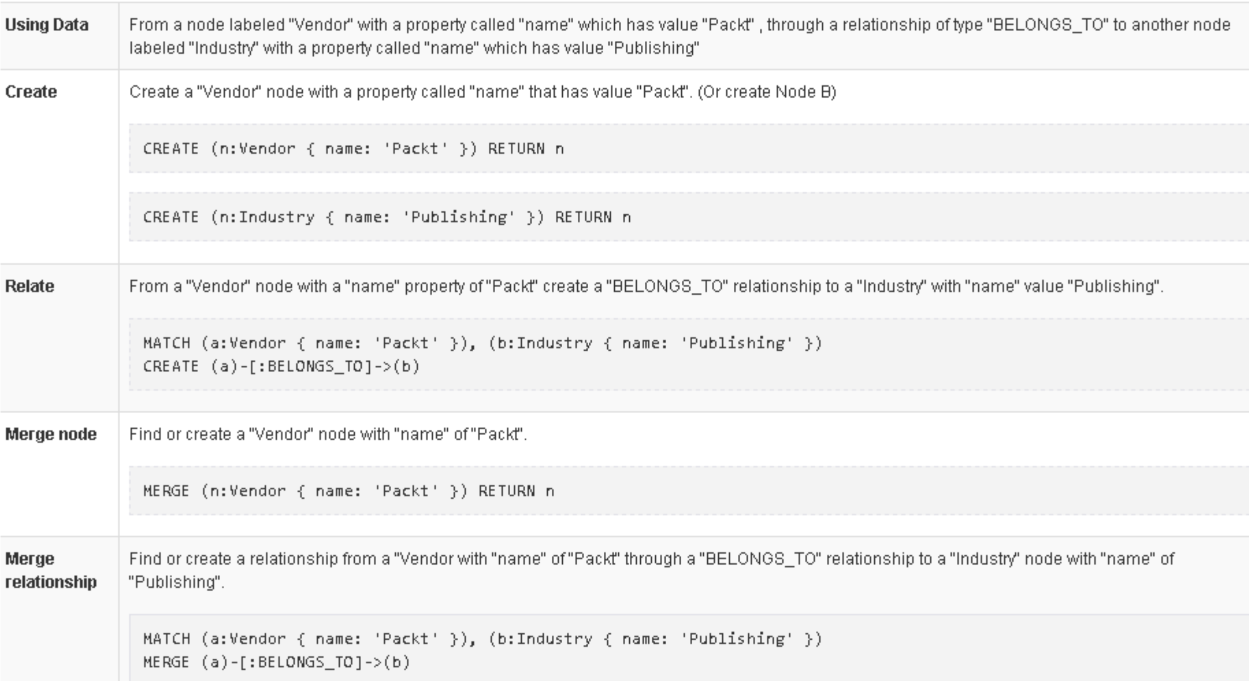
Since we now know how graphs are structured, let's use Cypher (the DML SQL equivalent language) for creating a node, in this case Vendor. Use the keyword "create" followed by an open parenthesis, then the properties (key/value pairs) surround by curly braces, then a close parenthesis. The open/close parenthesis encapsulate a node.



```
$ CREATE (v:Vendor { id:3, name: 'Register of
  Copyrights', city:'Washington   DC',
  state:'',defaultTerm:3, defaultAccount:403})
```

There is another way to create a node by use of "merge" which combines a match and create, i.e., don't create if a match is found.

The neo4j console provides a template to help you create nodes and relationships. An example below is provided that will create two nodes, a Vendor called Packt and an Industry of Publishing and then create a relationship of Belongs_To between these nodes. See figure below.



Using Data	From a node labeled "Vendor" with a property called "name" which has value "Packt" , through a relationship of type "BELONGS_TO" to another node labeled "Industry" with a property called "name" which has value "Publishing"

Create	Create a "Vendor" node with a property called "name" that has value "Packt". (Or create Node B)
	`CREATE (n:Vendor { name: 'Packt' }) RETURN n`
	`CREATE (n:Industry { name: 'Publishing' }) RETURN n`

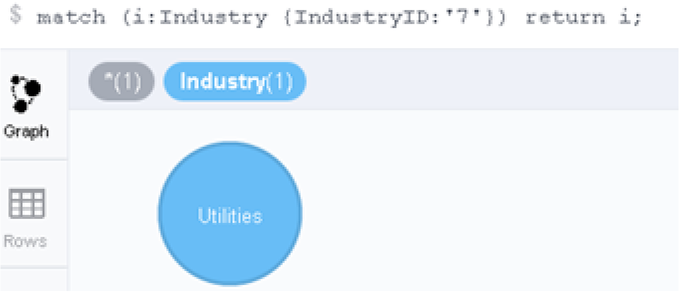
Relate	From a "Vendor" node with a "name" property of "Packt" create a "BELONGS_TO" relationship to a "Industry" with "name" value "Publishing".
	`MATCH (a:Vendor { name: 'Packt' }), (b:Industry { name: 'Publishing' })`
	`CREATE (a)-[:BELONGS_TO]->(b)`

Merge node	Find or create a "Vendor" node with "name" of "Packt". `MERGE (n:Vendor { name: 'Packt' }) RETURN n`

Merge relationship	Find or create a relationship from a "Vendor with "name" of "Packt" through a "BELONGS_TO" relationship to a "Industry" node with "name" of "Publishing".
	`MATCH (a:Vendor { name: 'Packt' }), (b:Industry { name: 'Publishing' })`
	`MERGE (a)-[:BELONGS_TO]->(b)`

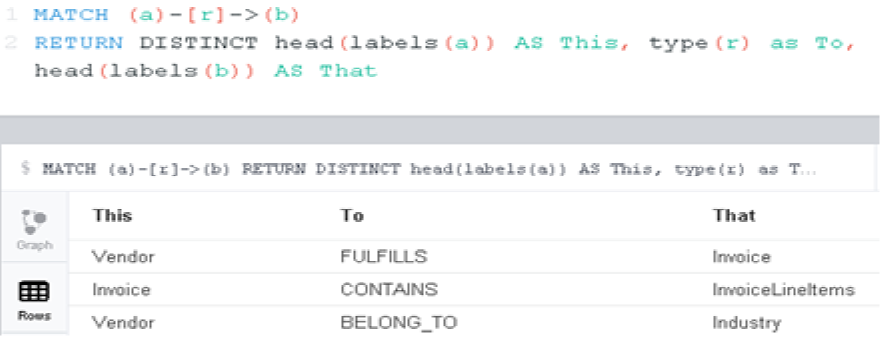
Retrieve

Retrieval as mentioned above use the match keyword, match needs a label (which is like a table name in SQL) and specific properties to search on. The identifier used with the label is then used to store the return value.



```
$ match (i:Industry {IndustryID:'7'}) return i;
```

A powerful feature of Cypher is an easy way to retrieve all the relationships between all the nodes. The way to specify a match on (a)-[r]->(b) where any node is related to any other node.   See figure below.

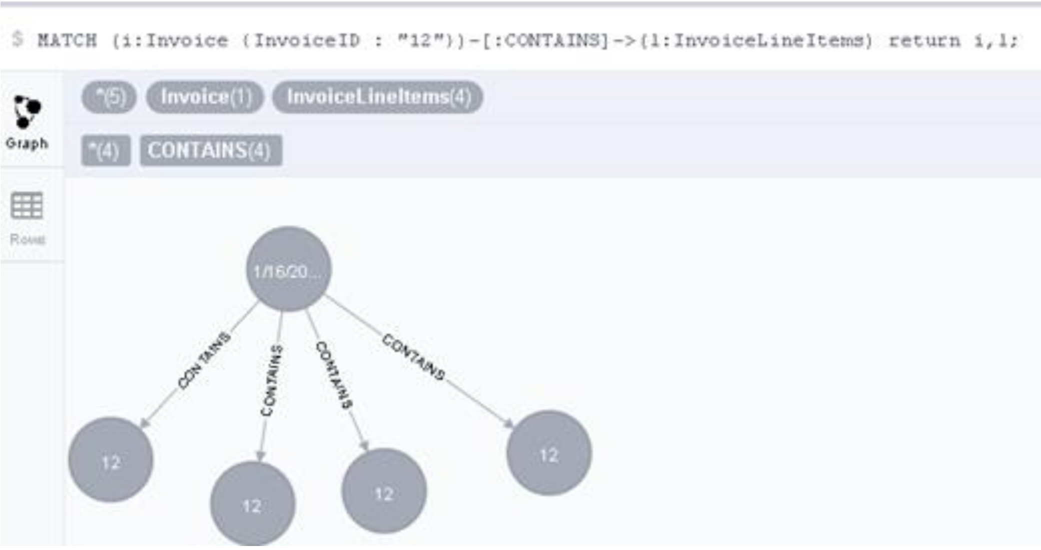


```
MATCH (a)-[r]->(b)
RETURN DISTINCT head(labels(a)) AS This, type(r) as To,
head(labels(b)) AS That
```

```
$ MATCH (a)-[r]->(b) RETURN DISTINCT head(labels(a)) AS This, type(r) as T...
```

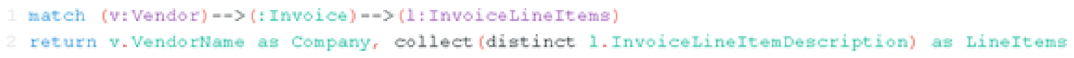
This	To	That
Vendor	FULFILLS	Invoice
Invoice	CONTAINS	InvoiceLineItems
Vendor	BELONG_TO	Industry

You can also retrieve a graph structure by using similar syntax above but in this case return the identifiers used to store the labels of connecting nodes:

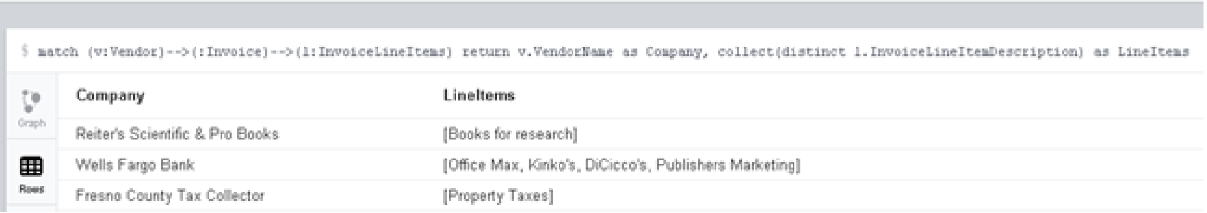


```
$ MATCH (i:Invoice {InvoiceID : "12"})-[:CONTAINS]->(l:InvoiceLineItems) return i,l;
```

One final example of retrieval is a multi-join that can return all line items for a vendor. It will match on "vendor->Invoice->InvoiceLineItems"



```
match (v:Vendor)-->(:Invoice)-->(l:InvoiceLineItems)
return v.VendorName as Company, collect(distinct l.InvoiceLineItemDescription) as LineItems
```

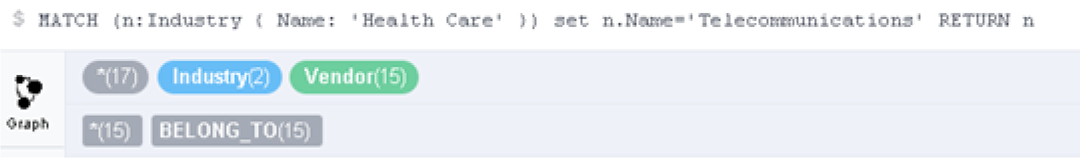


```
$ match (v:Vendor)-->(:Invoice)-->(l:InvoiceLineItems) return v.VendorName as Company, collect(distinct l.InvoiceLineItemDescription) as LineItems
```

Company	LineItems
Reiter's Scientific & Pro Books	[Books for research]
Wells Fargo Bank	[Office Max, Kinko's, DiCicco's, Publishers Marketing]
Fresno County Tax Collector	[Property Taxes]

Update

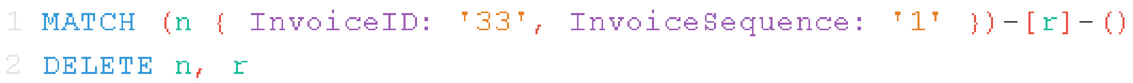
Updates in Cypher follow the same pattern as create and retrieve, namely find a match first. In the update, simply set the property associated with the identify used for a particular label.



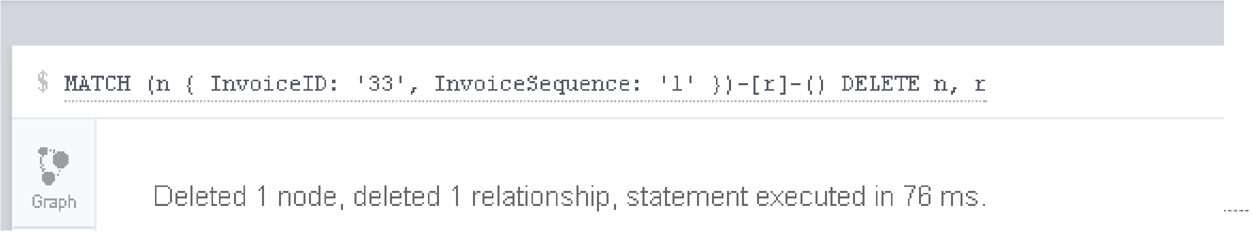
```
$ MATCH (n:Industry { Name: 'Health Care' }) set n.Name='Telecommunications' RETURN n
```

Delete

When deleting a node, make sure you also remove the connecting relationships, otherwise the remove will fail. In the case below, LineItem with InvoiceID 33, Sequence 1 should be deleted. However, since the InvoiceLineItem is connected via a relationship to an Invoice, that relationship must be deleted too. So in the Cypher statement below, both the node and relationship are deleted.



```
MATCH (n { InvoiceID: '33', InvoiceSequence: '1' })-[r]-()
DELETE n, r
```



```
$ MATCH (n { InvoiceID: '33', InvoiceSequence: '1' })-[r]-() DELETE n, r
```

Deleted 1 node, deleted 1 relationship, statement executed in 76 ms.

Relational Database

In contrast to an Entity Relationship Diagram (ERD) for a database, there is no schema for a graph database. A graph database is “schema-less” but it is still possible to abstract the node and relationship details into a diagram. Also, when exporting data from Relational it is not necessary to export the database schema.

Creating a Graph Database

Constructing the Graph Database

From SQL Server, export each database table (with Headers included) as a CSV file. Then execute the LOAD statement in Neo4j.   For this project there are four tables thus there are four CSV files. Each CSV file will be loaded into Neo4j. Note: Neo4j provides an example of importing the NorthWind database. This helpful step-by-step example can be found in the information page of the browser application.

Creating Nodes

Color Code

Manual way to create a node:

Command: CREATE (vendor3:Vendor { **id**:3, **name**: 'Register of Copyrights', **city**:'Washington DC', **state**:'',**defaultTerm**:3, **defaultAccount**:403})

Response: Added 1 label, created 1 node, set 6 properties

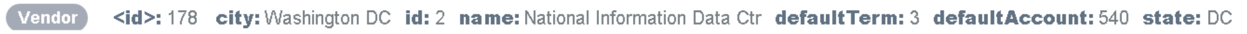
Query: `match (v:Vendor { id : 1 }) return v`

Response:

Set Query: `match (v:Vendor { id : 2 }) set v.state='DC' return v`

Response:

. When clicking on node, Vendor information confirms state change:

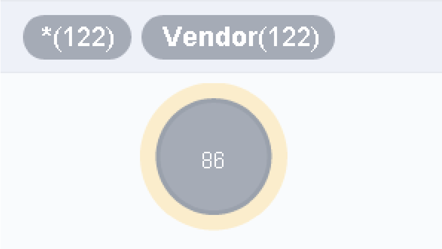


Vendor <id>: 178 city: Washington DC id: 2 name: National Information Data Ctr defaultTerm: 3 defaultAccount: 540 state: DC

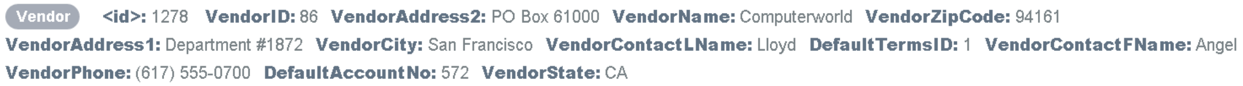
CSV alternative:

```
load csv with headers from "file:///C:/AP/Vendors.CSV" as row create (n:Vendor) set n = row
```

Added 122 labels, created 122 nodes, set 1464 properties,

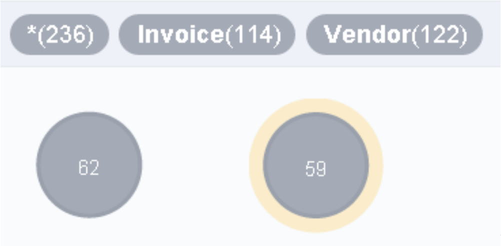


Note that vendors have no relationships yet.

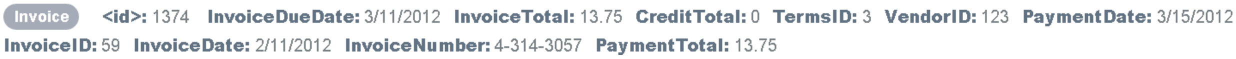


Vendor <id>: 1278 VendorID: 86 VendorAddress2: PO Box 61000 VendorName: Computerworld VendorZipCode: 94161 VendorAddress1: Department #1872 VendorCity: San Francisco VendorContactLName: Lloyd DefaultTermsID: 1 VendorContactFName: Angel VendorPhone: (617) 555-0700 DefaultAccountNo: 572 VendorState: CA

```
$load csv with headers from "file:///C:/AP/Invoices.CSV" as row create (n:Invoice) set n = row
```



Note that Invoices display InvoiceID, similar to Vendors displaying VendorID. We'll see later on the node's display is configurable and in fact with graph style sheets can have multiple properties displayed for a single node.



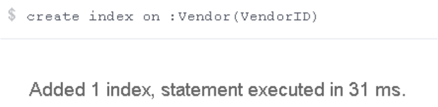
Invoice <id>: 1374 InvoiceDueDate: 3/11/2012 InvoiceTotal: 13.75 CreditTotal: 0 TermsID: 3 VendorID: 123 PaymentDate: 3/15/2012 InvoiceID: 59 InvoiceDate: 2/11/2012 InvoiceNumber: 4-314-3057 PaymentTotal: 13.75

Create Indexes

Now we will create indexes on the node Labels. This will allow faster retrieval when filtering by Lables.

```
create index on :Invoice(InvoiceID)
```

Added 1 index, statement executed in 67 ms.



```
$ create index on :Vendor(VendorID)
```

Added 1 index, statement executed in 31 ms.

Creating Relationships

The next step will create a relationship between Invoice and Vendor. It will do so without the usage of foreign keys, but instead with a create relationship statement, that is invoked after the same IDs from Invoices and Vendors are matched together.

```
match (i:Invoice),(v:Vendor) where i.VendorID = v.VendorID create (v)-[:FULFILLS]->(i)
```

Response: Created 114 relationships, statement executed in 136 ms.

Next the Invoice Line Items will be added.

```
$load csv with headers from "file:///C:/AP/InvoiceLineItems.CSV" as row create (n:InvoiceLineItems)
set n = row
```

Response: Added 118 labels, created 118 nodes, set 590 properties

Again the relationships will be created to "join" each Invoice with its associated Line Items.

```
match (i:Invoice),(l:InvoiceLineItems) where i.InvoiceID = l.InvoiceID create (i)-[:CONTAINS]->(l)
```

Response: Created 118 relationships.

Now to add complexity to the graph database, I created an Industry table that stored 10 unique industries. To tie a Vendor to an Industry this requires a many-to-many relationship from Vendors to Industries. To model this in SQL, a juncture table for VendorIndustry would need to be created expressly to store the foreign keys of Vendors and Industries. However in Neo4j, fortunately direct relationships are created between Vendors and Industries.

```
load csv with headers from "file:///C:/AP/Industry.CSV" as row create (n:Industry) set n = row
```

Response: Added 10 labels, created 10 nodes, set 20 properties.

Check an example industry: `match (i:Industry {Name: "Technology"}) return i`

Response:

Now to link Vendors to Industries, a match of both types is followed by equivalence checks on respective ID values, then followed by a create relationship command of Vendors belong to Industries.

load csv with headers from "file:///C:/AP/VendorIndustry.CSV" as row

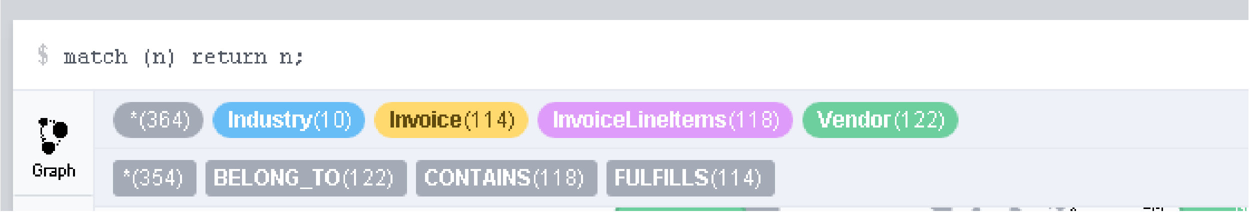
match (v:Vendor), (i:Industry)

```
where v.VendorID = row.VendorID AND i.IndustryID = row.Industry.ID

create (v)-[vendors:BELONG_TO]->(i)

set vendors = row
```

Response: Set 244 properties, created 122 relationships.

Final Result

When all nodes and relationships are imported, the result is



```
$ match (n) return n;
```

Figure 2 Actual Graph Database of Accounts Payable

Reference Table – Industry

IndustryID	Name
0	Oil and Gas
1	Basic Materials
2	Industrials
3	Consumer Goods
4	Health Care
5	Consumer Services
6	Telecommunications
7	Utilities
8	Financials
9	Technology

Reference Table – VendorIndustry

VendorID	IndustryID	Name
1	5	US Postal Service
2	9	National Information Data Ctr
3	4	Register of Copyrights
4	5	Jobtrak
5	3	Newbrige Book Clubs
6	5	California Chamber Of Commerce
7	5	Towne Advertiser's Mailing Svcs

8	0	BFI Industries	
9	7	Pacific Gas & Electric	
10	5	Robbins Mobile Lock And Key	
11	7	Bill Marvin Electric Inc	
12	7	City Of Fresno	
13	8	Golden Eagle Insurance Co	

This table is only needed for the Import and is not modeled as a separate join table (as it is in relational).

Data Table – Vendors

vendorid	vendorname	city	state	zip
1	US Postal Service	Madison	WI	53707
2	National Information Data Ctr	Washington	DC	20090
3	Register of Copyrights	Washington	DC	20559
4	Jobtrak	Los Angeles	CA	90025
5	Newbrige Book Clubs	Washington	NJ	7882

Parent Data Table – Invoices

InvoiceID	VendorID	InvoiceNumber	InvoiceDate	InvoiceTotal	PaymentTotal	CreditTotal	TermsID	InvoiceDueDate	PaymentDate
1	122	989319-457	12/8/2011 00:00	3813.33	3813.33	0	3	1/8/2012 00:00	1/7/2012 00:00
2	123	263253241	12/10/2011 00:00	40.2	40.2	0	3	1/10/2012 00:00	1/14/2012 00:00
3	123	963253234	12/13/2011 00:00	138.75	138.75	0	3	1/13/2012 00:00	1/9/2012 00:00
4	123	2-000-2993	12/16/2011 00:00	144.7	144.7	0	3	1/16/2012 00:00	1/12/2012 00:00
5	123	963253251	12/16/2011 00:00	15.5	15.5	0	3	1/16/2012 00:00	1/11/2012 00:00
6	123	963253261	12/16/2011 00:00	42.75	42.75	0	3	1/16/2012 00:00	1/21/2012 00:00
7	123	963253237	12/21/2011 00:00	172.5	172.5	0	3	1/21/2012 00:00	1/22/2012 00:00
8	89	125520-1	12/24/2011 00:00	95	95	0	1	1/4/2012 00:00	1/1/2012 00:00
9	121	97/488	12/24/2011 00:00	601.95	601.95	0	3	1/24/2012 00:00	1/21/2012 00:00

Child Data Table – Invoice Line Items

InvoiceID	InvoiceSequence	AccountNo	InvoiceLineItemAmount	InvoiceLineItemDescription
1	1	553	3813.33	Freight
2	1	553	40.2	Freight
3	1	553	138.75	Freight

4	1	553	144.7	Int'l shipment
5	1	553	15.5	Freight
6	1	553	42.75	Freight
7	1	553	172.5	Freight

Appendix: Helpful Cypher commands

Purpose	Command	Note
Get all nodes / relationships	start n=node(*) return n	
Delete all nodes / relationships	MATCH (n) OPTIONAL MATCH (n)-[r]-() DELETE n,r;	New command in Version 2.3 is: MATCH n DETACH DELETE n

Appendix: Graph Style Sheet

This allows great customization of the nodes and relationships.

```
node {
  diameter: 75px;
  color: #A5ABB6;
  border-color: #9AA1AC;
  border-width: 2px;
  text-color-internal: #FFFFFF;
  font-size: 10px;
}
relationship {
  color: #A5ABB6;
  shaft-width: 1px;
  font-size: 8px;
  padding: 3px;
  text-color-external: #000000;
  text-color-internal: #FFFFFF;

  caption: '<type>';

}

node.Vendor {

  color: #6DCE9E;

  border-color: #60B58B;

  text-color-internal: #FFFFFF;

  caption: '{VendorName} in {VendorState}';

}


node.Invoice {

  color: #FFD86E;

  border-color: #EDBA39;

  text-color-internal: #604A0E;

  caption: '${InvoiceTotal} due on {InvoiceDueDate}';

}

node.InvoiceLineItems {

  color: #DE9BF9;

  border-color: #BF85D6;

  text-color-internal: #FFFFFF;

  caption: '{InvoiceLineItemDescription}';

}

node.Industry {

  color: #68BDF6;

  border-color: #5CA8DB;

  text-color-internal: #FFFFFF;

  caption: '{Name}';
```

```
}


relationship.BELONG_TO {

  shaft-width: 8px;

}
```

Appendix – Helpful tips for using Neo4j browser

http://neo4j.com/blog/neo4j-2-0-0-m06-introducing-neo4js-browser/

Just type in a single-line query and hit `<enter>`. You'll get a result frame in the stream, showing either a table of property data or a graph visualization of nodes and relationships.

Type another query, get another frame. Tap the up arrow to retrieve a previous entry, edit, and then run it again.

For larger queries, hit `<shift-enter>` to switch into multi-line editor mode. Now you'll need to use `<ctrl-enter>` to run, and `<ctrl>` up or down arrow to navigate history (the modifier also works in single-line mode).

Finally, and thankfully, you can save scripts. Hit the star button to save the current editor content, which will be available in the sidebar. By convention, the first line can be a comment which will be used as the name of the query. You can even drag-and-drop in scripts, for sharing queries or small-scale data import.

References

Importing data into Neo4j: http://neo4j.com/developer/guide-importing-data-and-etl/

Neo4j Cypher Refcard 2.3.1 : http://neo4j.com/docs/stable/cypher-refcard/

Robinson, Ian, and James Webber. *Graph Databases: New Opportunities for Connected Data*. Second ed.