

# Appendix Q

## Working with Neo4j

### Preview

Even though Neo4j is not yet as widely adopted as MongoDB, it has been one of the fastest growing NoSQL databases, with thousands of adopters such as LinkedIn and Walmart. Neo4j is a graph database. Other types of NoSQL databases focus primarily on problem domains where volume and velocity are the core issues. In those situations, aggregate aware databases that can minimize interdependencies among the data can reap the benefits of scaling out to massive clusters of servers. Graph databases address a completely different type of problem. Like relational databases, graph databases still work with concepts similar to entities and relationships. However, in relational databases, the focus is primarily on the entities. In graph databases, the focus is on the relationships. This appendix shows you some step-by-step illustrations of using the Neo4j graph database.

### Data Files and Available Formats

File name	Format/Description
Ch14_FCC.txt	Text file for Neo4j examples (also used in Chapter 14)

*Data Files Available on [cengagebrain.com](http://cengagebrain.com)*



## Note

Neo4j is a product of Neo4j, Inc. There are multiple versions of Neo4j available. In this appendix, we use the Community Server v.3.2.2 edition, which is open source and available free of charge from Neo4j, Inc. New versions are released regularly. This version of Neo4j is available from the Neo4j website for Windows (64-bit and 32-bit), MacOS, and Linux.

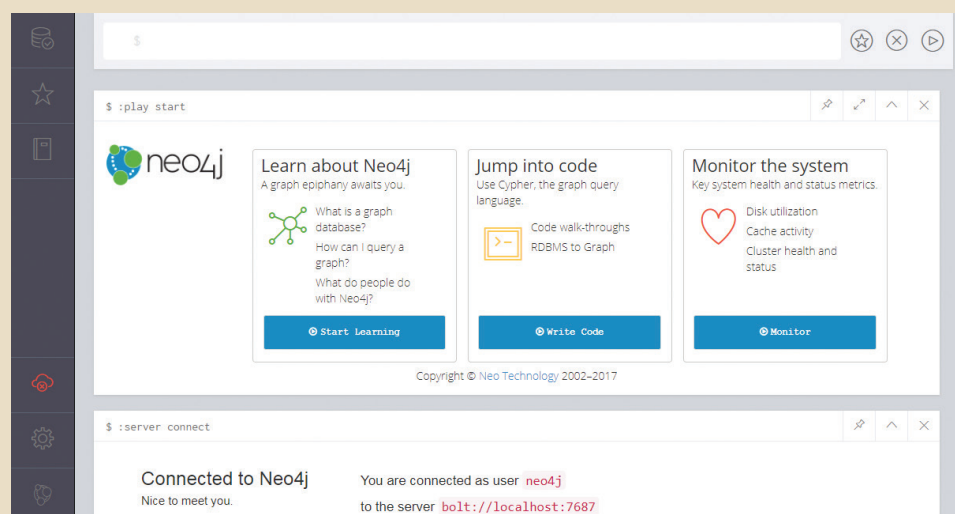
## Q-1 Working with Graph Databases Using Neo4j

Graph databases are used in environments with complex relationships among entities. Graph databases, therefore, are heavily reliant on interdependence among their data, which is why they are the least able to scale out among the NoSQL database types. Consider an example of a social network such as LinkedIn that connects people together. A person can be friends with many other people, each of whom can be friends with many people. In terms of a relational model, we could represent this as a person entity with a many-to-many unary relationship. In implementation, we would create a bridge for the relationship and end up with a two-entity solution. Imagine the person table has 10,000 people (rows) in it, and those people average 30 friends each so that the bridge table has 300,000 rows. A query to retrieve a person and the names of his or her friends would require two joins: one to link the person to their friends in the bridge and another to retrieve those friends' names from the person entity. A relational database can perform this query quickly. The problem comes when we look beyond that direct friend relationship. What if we want to know about friends of friends? Then another join connecting the bridge table to itself will have to be included. Joining a 300,000-row table to itself is not trivial (there are 90 billion rows in the Cartesian product that the DBMS engine is contending with to construct the join). The relational database can handle that volume, but it is starting to slow. Now query for friends of friends of friends. This requires joining yet another copy of the bridge table so the query, producing a Cartesian product with  $2.7 \times 10^{16}$  rows! As you can see, by the time we are working the "six degrees of separation" types of problems, relational database technology is unable to keep up. These types of highly interdependent queries about relationships that could take hours to run in a relational database are the forte of graph databases. Graph databases can complete these queries in seconds. In fact, you often encounter the phrase "minutes to milliseconds" when adopters describe their use of graph databases.

## Q-2 The Neo4j Interface

Neo4j provides multiple interface options. It was originally designed with Java programming in mind, and optimized for interaction through a Java API. Later releases have included the options for a Neo4j command shell, similar to the MongoDB shell, a REST API for website interaction, and a graphical, browser-based interface for intuitive interactive sessions. Figure Q.1 shows the browser-based interface.

FIGURE Q.1 GRAPHICAL, BROWSER-BASED NEO4J INTERFACE



We focus on the browser-based interface throughout this appendix. The browser interface has evolved over different versions and is likely to evolve further. In the current version of the web interface, the left side of the screen is a menu with options for viewing database information, documentation, and changing settings. The main section of the screen is a stream of database activity for the current session. When you first log in, this section shows links to take you to various activities. Notice that the stream is composed of several individual blocks, or frames. Each block can be closed by clicking the “X” in the upper-right corner of that block, expanded to fill the screen, and other options. The results of query execution appear in the stream. As a session goes on, the stream can become quite long and filled with the results of all previous queries. Eventually, this can begin to slow your browser performance, so clearing unneeded results by entering the `:clear` command at the editor prompt is recommended.

At the top of the main section is an editor bar with a “\$” prompt. This is the editor prompt for entering new commands. At the right end of the editor bar are options to “play” (right triangle) or execute the query that has been entered at the prompt, “clear” (x) to clear the editor bar, and “update favorite” (star) to save the query to the list of favorite queries that is accessible in the left-side menu. The editor is in single-line mode by default. When in single-line mode, pressing Enter executes the command. This can be convenient because many Neo4j commands can be rather short. If a command is too long or complex for single-line mode, the editor can be placed in multiline mode by pressing Shift+Enter. For very large commands, full-screen mode for the editor can be toggled on and off by pressing Esc. When in multiline or full-screen mode, commands can be executed by pressing Ctrl+Enter (Cmd+Enter for Mac). In all cases, the command in the editor bar can be executed by clicking the “play” button at the right end of the bar.

## Q-3 Creating Nodes in Neo4j



### Note

An instance of Neo4j can have only one active database at a time. However, the data path for the database can be changed in the configuration before starting the Neo4j server. If the data path is changed to point at an empty directory, Neo4j automatically creates all needed files in that directory on start-up. By keeping each database in a separate folder and changing the data path before starting the server, multiple databases can be maintained for practice.

Graph databases are composed of nodes and edges. Roughly speaking, nodes in a graph database correspond to entity instances in a relational database. In Neo4j, a label is the closest thing to the concept of a table from the relational model. A label is a tag that is used to associate a collection of nodes as being of the same type or belonging to the same group. Just as entity instances have values for attributes to describe the characteristics of that instance, a node has properties that describe the characteristics of that node. Unlike the relational model, graph databases are schema-less so nodes with the same label are not required to have the same set of properties. In fact, nodes can have more than one label if they logically belong to more than one group.

Consider an example of a club for food critics where members share reviews of area restaurants. Each club member would be represented as a node. Each restaurant would be represented as a node. Although both members and restaurants are nodes, the members are one kind or type of node while the restaurants are another kind or type of node. To help distinguish the types of nodes both in code and in the minds of users and programmers, you can use labels. The nodes for members might get a Member label, and nodes for restaurants get a Restaurant label. This makes it more convenient in code to distinguish between the types of nodes.

The interactive, declarative query language in Neo4j is called **Cypher**. We will be using Cypher to manipulate and traverse (query) the graph. Cypher is declarative, like SQL, even though the syntax is very different. However, being a declarative language instead of an imperative language, like MQL, Cypher is very easy to learn and a few simple commands can be used to perform basic database processing.

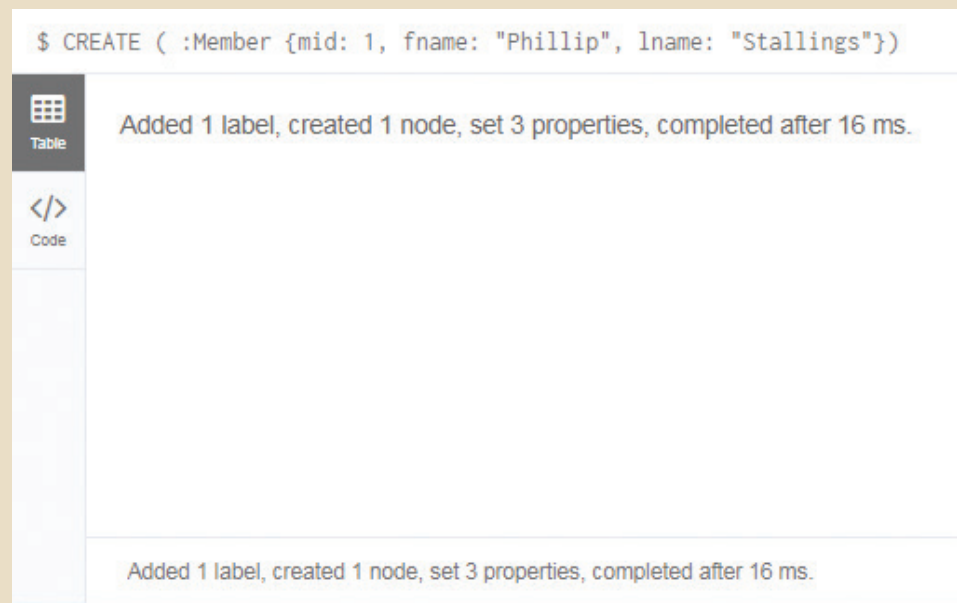
Nodes and relationships are created using a CREATE command. The following code creates a member node, as shown in Figure Q.2:

```
CREATE (:Member {mid: 1, fname: "Phillip", lname: "Stallings"})
```

### Cypher

A declarative query language used in Neo4j for querying a graph database.

FIGURE Q.2 CREATING A NODE IN NEO4J



### Note

Neo4j creates an internal ID field named `<id>` for every node and relationship; however, this field is for internal use within the database for storage algorithms. It is not intended to be, and should not be used as, a unique key.

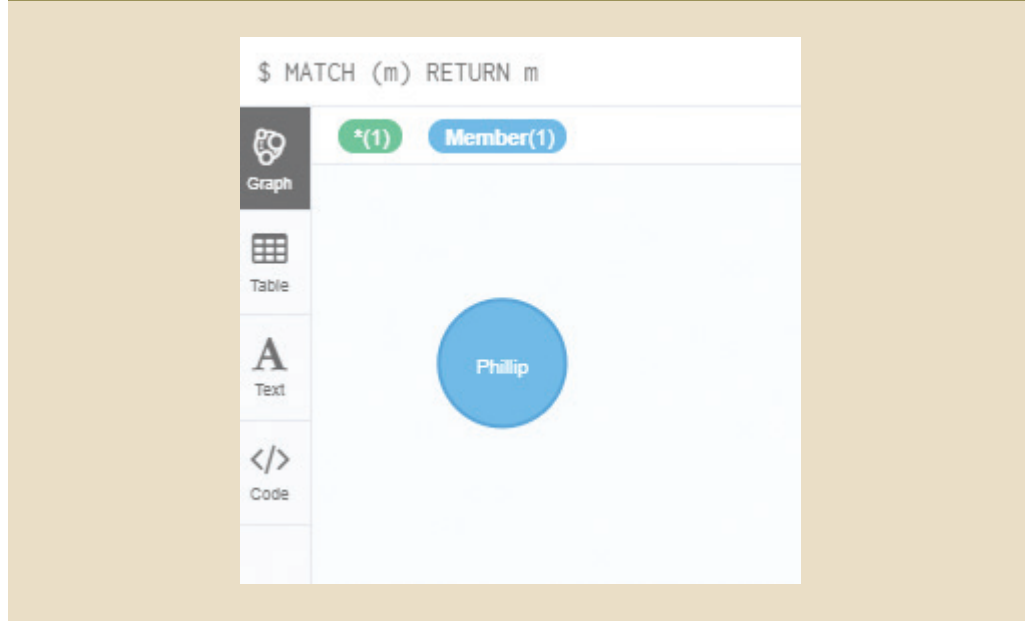
The previous command creates a node with the `Member` label. That node was given the properties `mid` with the value 1, `fname` with the value “Phillip”, and the property `lname` with the value “Stallings”. The `mid` property is being used as a member ID field to identify the members. If there is not already a label named `Member`, it is created at the same time the node is, as shown in the output.

Retrieving nodes and relationships will be covered in detail a little later, but for now, the following command can be used to retrieve all nodes from the database, as shown in Figure Q.3:

```
MATCH (m)
```

```
RETURN (m)
```

FIGURE Q.3 VIEWING THE NEW MEMBER NODE



**MATCH** is the Cypher command to find data in the database. In this case, “m” is being used as a variable. Just as with most variables, we can name the variable anything we want, but we try to be descriptive. The current database contains only a single Member node so “m” was chosen to represent Member. Without any qualifiers to restrict the results, the MATCH command matches every node by default. In the previous command, then, all nodes are found and assigned to the variable *m*. **RETURN** is the Cypher command to return the values in a variable. In this case, we are returning every value in the variable *m*, which contains every node in the database due to the unrestricted MATCH command.

Executing this command retrieves the node we created and displays it in a frame in the stream of the main window. Options for viewing the data being returned are located on the left side of the frame. If the data contains nodes, as with the previous command, there is an option to view the result as a graph. Alternatively, you can choose to view it in a tabular format that displays the result in a JSON format, in a text format that mirrors the output you would see in the Neo4j shell, or the coded format that would be returned if the command had been issued through one of the programming APIs. In the graph view of the data, the layout can be manipulated by dragging nodes, and the display options such as node size, color, and caption can be customized.

The following command can be used to create a restaurant node with the label Restaurant, and properties for restaurant id (rid), name, and city:

```
CREATE (:Restaurant {rid: 1, name: "Joe's Eatery", city: "Browningville"})
```

#### match

A keyword in the Cypher language used by Neo4j for pattern matching.

#### return

A keyword in a Cypher language used by Neo4j to specify the nodes and properties returned by a traversal

## Q-4 Updating Nodes in Neo4j

Like most processing in Cypher, updating a node in Neo4j involves the use of the MATCH command. Before a change can be made to a node, the node must be found in the database, which is the purpose of MATCH. Once MATCH locates the node, the SET command can be used to update or add properties to the node. For example, if we want

to change the city of our restaurant node from “Browningville” to “Brownville” and add property named state with the value “OH”, the following command could be used:

```
MATCH(r {rid: 1})
```

```
SET r.city = "Brownville", r.state = "OH"
```

Neo4j responds that the properties have been set as shown in Figure Q.4. The MATCH command finds all nodes that have `rid = 1` and assigns them all to the variable `r` (there was only one such node in our database). The SET command changes the values in properties. In the previous command, SET changes the city property of the node contained in variable `r` to have the value “Brownville”, and it changes the state property of the node contained in variable `r` to have the value “OH”. If the property does not exist, SET will add that property to the node. You can retrieve all of the nodes in the database again as shown previously. By hovering the cursor over the Joe’s Eatery node, you can see the changes to the properties, or you can click on the table or text views on the left of the frame to see the properties, as shown in Figure Q.5.

FIGURE Q.4 UPDATING A RESTAURANT NODE

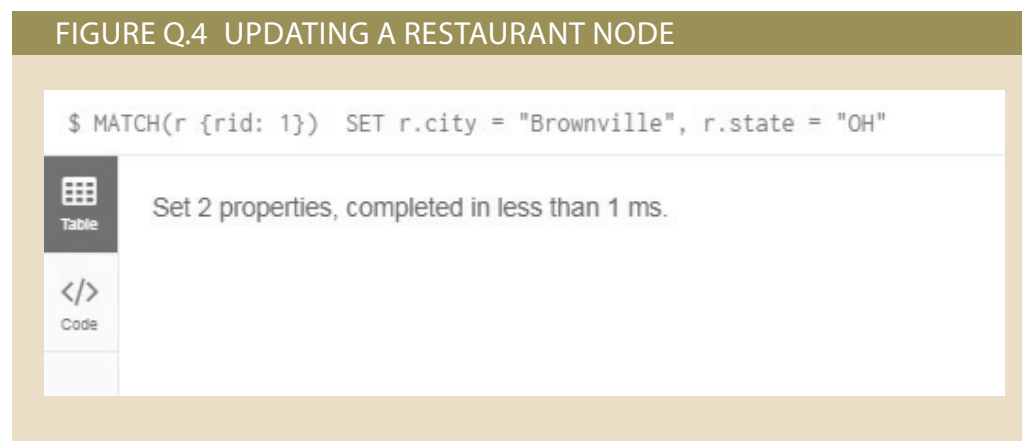




FIGURE Q.5 VIEWING THE RESTAURANT NODE PROPERTIES

The screenshot shows a Neo4j Cypher query interface. At the top, the query is `$ MATCH (m) RETURN m`. On the left, there is a sidebar with icons for Graph, Table (selected), Text, and Code. The main area displays the results of the query, which are two JSON objects representing restaurant nodes. The first object has properties `"fname": "Phillip", "lname": "Stallings", "mid": 1`. The second object has properties `"city": "Brownville", "name": "Joe's Eatery", "state": "OH", "rid": 1`. At the bottom, a status message reads: "Started streaming 2 records after 1 ms and completed after 1 ms."

```
$ MATCH (m) RETURN m
```

m

```
{
  "fname": "Phillip",
  "lname": "Stallings",
  "mid": 1
}
```

```
{
  "city": "Brownville",
  "name": "Joe's Eatery",
  "state": "OH",
  "rid": 1
}
```

Started streaming 2 records after 1 ms and completed after 1 ms.

Removing a property is done in a similar fashion, except with the REMOVE command instead of the SET command. If we wish to remove the city property from Joe's Eatery, the following command could be used (see Figure Q.6):

```
MATCH (r {rid: 1})
REMOVE r.city
```



FIGURE Q.6 RESTAURANT WITH CITY PROPERTY REMOVED



## Q-5 Retrieving Nodes Using MATCH

As we have already seen, MATCH is used to find nodes. More accurately, MATCH is used for pattern matching. Retrievals in Neo4j are all about pattern matching. Recall the graph view of the nodes that were created above. The nodes are represented as circles. MATCH uses the symbols on the keyboard to “draw” patterns. Nodes are inside parentheses () because they look like a circle. Properties are listed inside the parentheses because they are the properties of the node. MATCH places the nodes that match the pattern specified in the variable specified. Multiple patterns can be specified with MATCH. For example, we could use the following command to find both our Member node and Restaurant node, with the result shown in Figure Q.7:

```
MATCH (m {mid: 1}), (r {rid: 1})
```

```
RETURN m, r
```

FIGURE Q.7 RETRIEVING NODES WITH TWO PATTERNS



The first pattern finds all nodes with the property `mid` with the value 1 and assigns them to the variable `m`. The second pattern finds all nodes with the property `rid` with the value 1 and assigns them to the variable `r`. `RETURN` then displays all the nodes in both variables. Combining patterns with a comma causes a logical *and* between the patterns. In this case, the query returned both nodes because the set of nodes in the two patterns did not have to overlap. Consider the following command, which will not return any nodes:

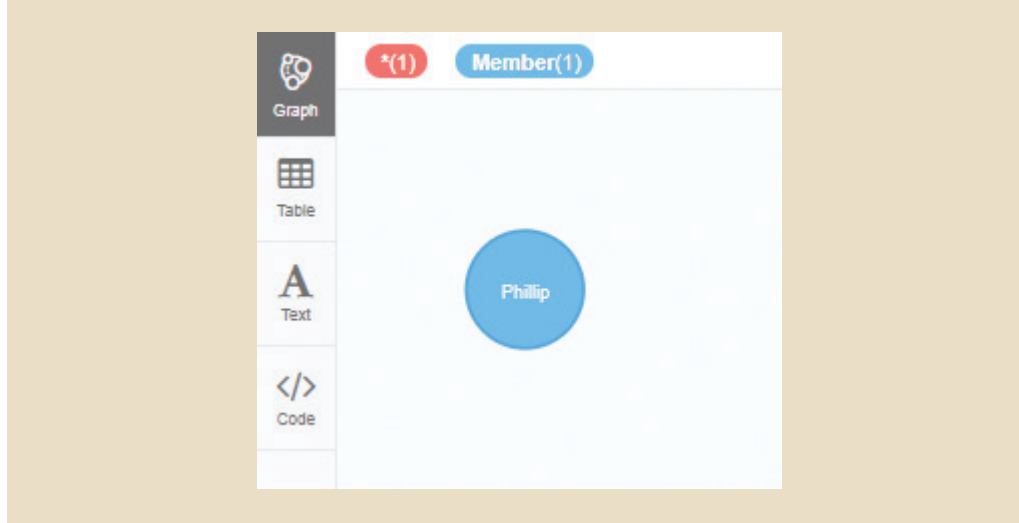
```
MATCH (x {mid: 1}), (x {rid: 1})
RETURN x
```

The previous command returns no nodes because the same variable `x` was used in both patterns. This means that to be returned, a node would have to match both patterns. On the other hand, the following command will return a node:

```
MATCH (x {fname: "Phillip"}), (x {lname: "Stallings"})
RETURN x
```

Figure Q.8 shows that this command will return a node because our Member node matches both patterns. Notice again that the name of the variable does not matter. In this case, the variable `x` is not very descriptive. The important thing to remember is that whatever the variable name is, it must be used consistently throughout the command whenever you want to refer to the same set.

FIGURE Q.8 MATCHING TWO PATTERNS WITH ONE NODE



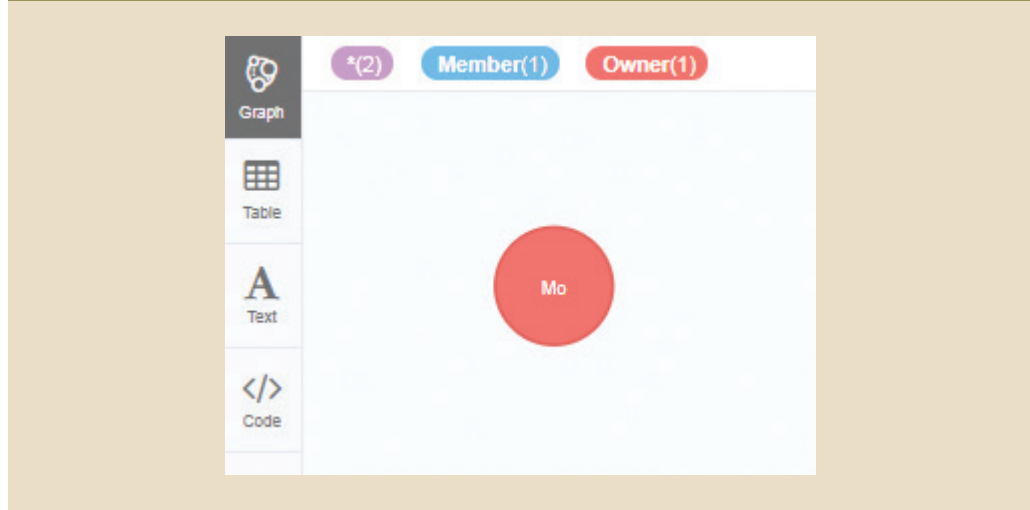
In addition to matching patterns based on properties, it is also possible to use labels in the pattern. The following command will return our only Member node:

```
MATCH (phil :Member)
RETURN phil
```

Notice that the variable name used is “phil”. You will see in practice that when programmers know that the pattern is designed to match only one node, the variable name often reflects the name of the node. This becomes more useful in improving the readability of the query when the retrievals become more complex, as we will see later. In the previous command, the label Member is used to limit the pattern to only nodes with that label. As mentioned earlier, it is possible for a node to have more than one label. For example, a club member may also be the owner of a restaurant. It may be valuable to use an Owner label to simplify working with the group of nodes that represent restaurant owners. The following command creates a new node that is labeled as both a member and an owner, and returns that node all in one command, as shown in Figure Q.9:

```
CREATE (new :Member:Owner {mid: 2, fname: “Mo”, lname: “Saleem”})
RETURN new
```

FIGURE Q.9 CREATING A NODE WITH TWO LABELS



The newly created node has both the Member label and the Owner label. The following command will return all nodes with the Member label:

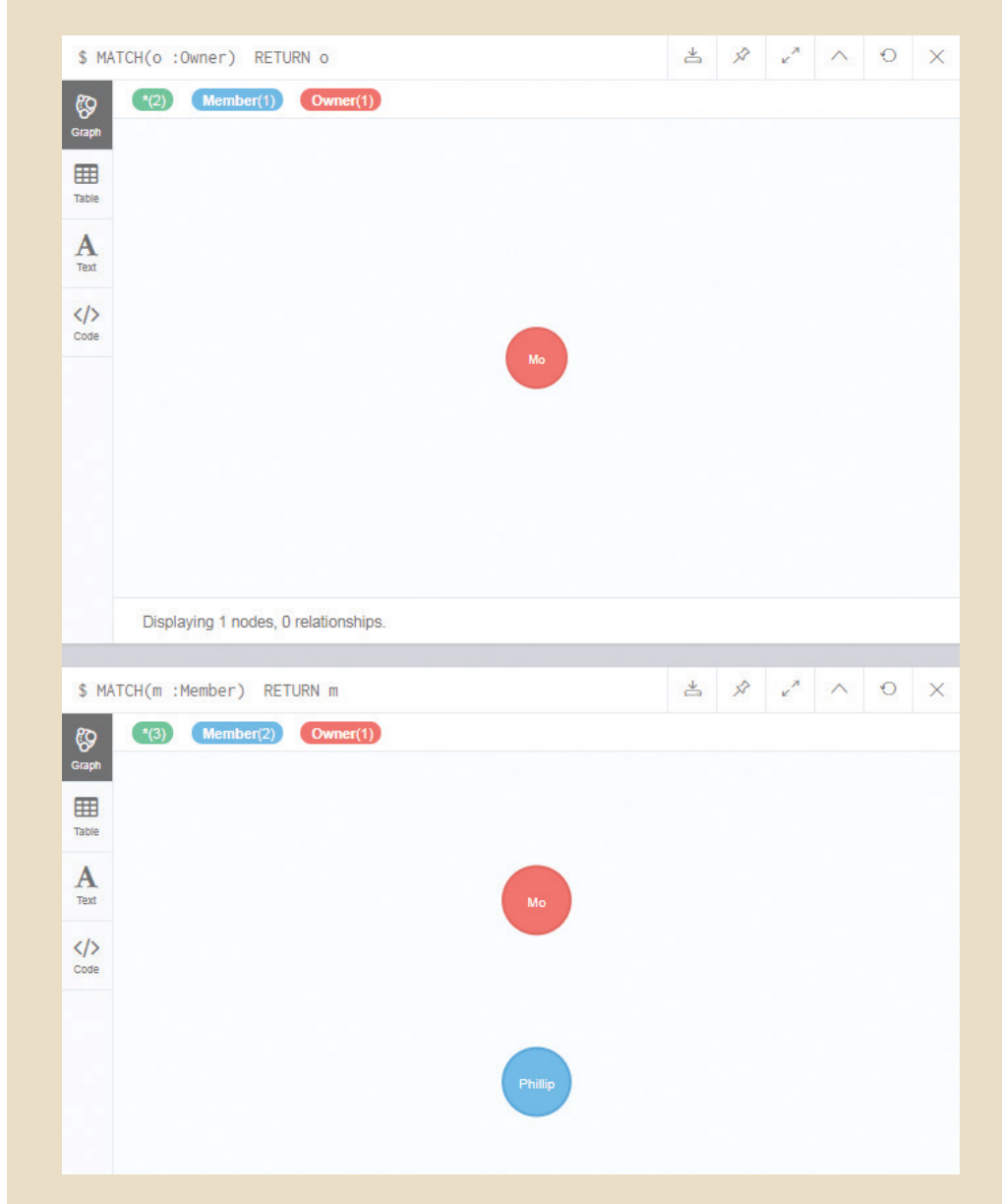
```
MATCH(m :Member)
RETURN m
```

The next command will return all nodes with the Owner label:

```
MATCH(o :Owner)
RETURN o
```

The results of both commands are shown in Figure Q.10. Notice that the Mo Saleem node was returned both times.

FIGURE Q.10 RETRIEVING THE SAME NODE WITH DIFFERENT LABELS

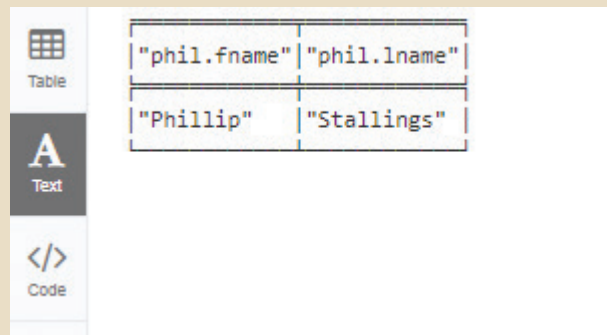


### Q-5a RETURN Nodes and Properties

The RETURN keyword in Cypher is used to specify the data that is to be returned by the query by listing the variables that contain the data to be returned. As you have already seen, listing a variable that contains a node will return the entire node. We can view the node graphically with the graph view of the Neo4j interface, or we can view tabular or text versions of the data. However, it is not required to retrieve the entire node. Just as when using the SET command to update a property of a node, dot notation can be used to specify a property of a node to be returned. For example, the following command returns only the first and last name properties of the Phillip Stallings member node, as shown in Figure Q.11:

```
MATCH (phil :Member {mid: 1})
RETURN phil.fname, phil.lname
```

FIGURE Q.11 RETRIEVING SPECIFIED PROPERTIES



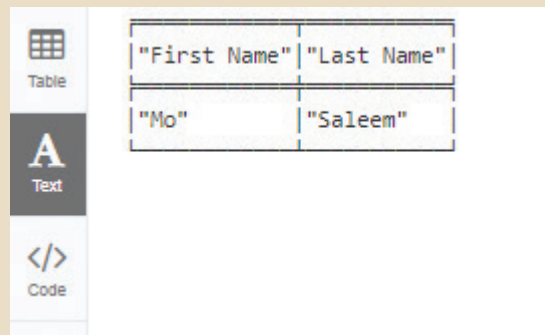
"phil.fname"	"phil.lname"
"Phillip"	"Stallings"

Notice that the graph view is not available in the interface because the entire node was not returned.

When returning individual properties, it is possible to provide aliases as is done in SQL with column aliases. In Cypher, the AS keyword is required before the alias. If an alias contains a space or a reserved keyword, then it must be enclosed in backticks, as shown in Figure Q.12 for the following code:

```
MATCH (mem :Member {mid: 2})
RETURN mem.fname AS 'First Name', mem.lname AS 'Last Name'
```

FIGURE Q.12 PROPERTIES WITH ALIASES



"First Name"	"Last Name"
"Mo"	"Saleem"



### Note

The following sections on Neo4j assume that you have loaded the data from the Ch14\_FCC.txt file that is available online and used with Chapter 14, Big Data and NoSQL. This file contains a single, massive command that will create 78 additional members, 43 owners, 67 restaurants, and 8 cuisines. Providing the code as a single command is necessary if you are using the browser interface. Because it is designed for interactive use, it does not support script files with multiple commands. The command includes many statements that may seem unfamiliar to you, but by the time you finish this chapter, all of the commands should make sense.

## Q-5b Retrieving Nodes with WHERE

The data model used for the following sections is from the Food Critics Club (FCC). The database tracks data on members, restaurants, and owners like the ones created in commands in the previous sections. It also records data on the different cuisines, or styles of food. Members may have left reviews of one or more restaurants. With each review, the member can rate the restaurant on a scale of 1 to 5 for each of the following categories: taste, service, atmosphere, and value. Each member is assigned a unique member id (mid). This is recorded along with the member's first and last name, email address, and a FCC user name. Optionally, the member's year of birth and state of residency may also be recorded. Restaurants have a restaurant id number (rid) and name. Typically, the restaurant's address and phone number are also recorded. Most restaurants have a price rating on a scale of 1 to 5, where 1 is the cheapest and 5 is the most expensive.

Thus far, we have specified criteria for the nodes to retrieve either by label or by property inside the node pattern. The major limitation with using properties inside the node for retrievals is that only equalities can be included. Further, multiple properties and values are always treated with a logical *and*. This is similar to the way MongoDB interpreted a key:value pair in a criterion as an equality. Neo4j and Cypher do the same thing.

Alternatively, we can add a WHERE clause following the MATCH pattern. The WHERE clause allows both *and* and *or* logical connectors, and it allows criteria to include inequalities. Unlike MongoDB, Neo4j uses mathematical operators for inequalities instead of functions. Therefore, the WHERE clause in Cypher is similar to the WHERE clause in SQL. The following command retrieves all members that were born after 1987.

```
MATCH (m :Member)
WHERE m.birth > 1987
RETURN m
```

Using the WHERE clause allows both logical *and* and *or* operations. As with SQL, when combining *and* and *or* in the same WHERE clause, it is recommended to use parentheses to ensure the correct order of operation for the logical connectors. The following command finds all members that live in the states of TN or KY and were born before 1984:

```
MATCH (m :Member)
WHERE (m.state = "TN" or m.state = "KY") and m.birth < 1984
RETURN m
```

## Q-5c Common String Functions in Neo4j

The Cypher language in Neo4j provides many of the same character string functions available in SQL that were discussed in Chapter 7, Introduction to Structured Query Language (SQL). Table Q.1 lists some of the most common string functions in Cypher. By default, queries in Neo4j are case sensitive. For example, it is common practice to start labels with a capital letter, thus we have :Member, :Restaurant, :Owner, and :Cuisine labels. If we issue a query that includes :member or :MEMBER as a label, it will not match any nodes because the label is case sensitive. Similarly, if we had queried for the STATE or State property of a member, no nodes would be found because the property has been entered in all lowercase letters in our database.

### Online Content



The Ch14\_FCC.txt file used in the following sections is available at [www.cengagebrain.com](http://www.cengagebrain.com). The contents of the file should be copied and pasted into the Neo4j editor bar and executed using the play button in the interface.



TABLE Q.1

## COMMON CYPHER STRING FUNCTIONS

FUNCTION	DESCRIPTION
lower(<text>)	Converts text to lowercase letters
left(<text>, <number>)	Retrieves the specified number of letters from the left side of the text
right(<text>, <number>)	Retrieves the specified number of letters from the right side of the text
substring(<text>, <start>, <number>)	Retrieves the specified number of letters from the string starting with the specified character position
trim(<text>)	Removes all whitespace from both sides of the text
upper(<text>)	Converts text to uppercase letter

As the database programmer, it is your responsibility to ensure consistency in how labels and properties are capitalized to avoid problems later. Your organization probably has defined standards for these issues. However, we cannot always control the way that users capitalize data values that are being entered into the system. Therefore, string functions like upper() and lower() are available. The following command retrieves all restaurants located in a city named Brentwood, regardless of how the value is capitalized in the database. The results are shown in Figure Q.13.

```
MATCH (res :Restaurant)
WHERE lower(res.city) = "brentwood"
RETURN res
```

FIGURE Q.13 RESTAURANTS IN BRENTWOOD



res
<pre>{   "zip": "3833",   "phone": "(404) 555-1616",   "city": "Brentwood",   "street": "1412 Frolick Wind Road",   "price": 3,   "name": "El Toros",   "state": "GA",   "rid": 4639 }</pre>
<pre>{   "zip": "3833",   "phone": "(615) 555-1606",   "city": "Brentwood",   "street": "1412 Frolick Wind Road",   "price": 1,   "name": "The Fridge",   "state": "TN",   "rid": 4651 }</pre>

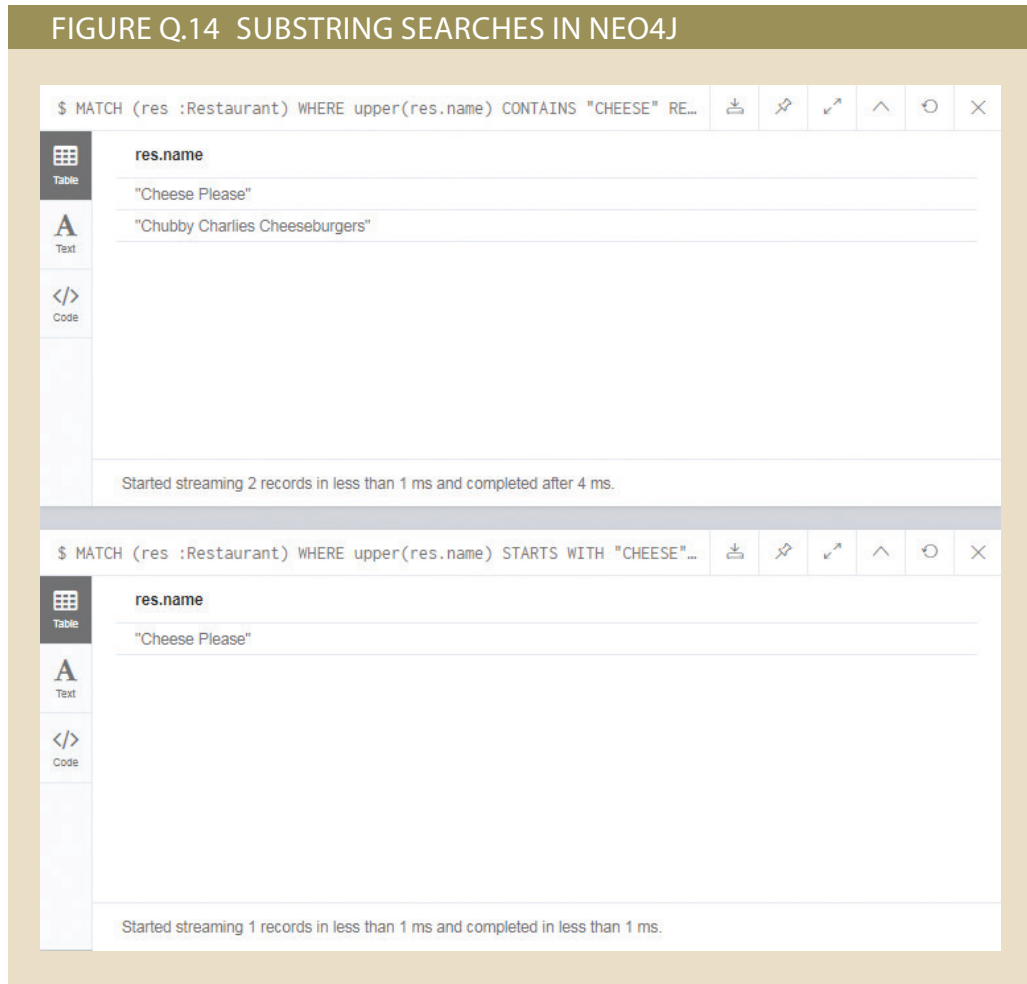
Instead of the LIKE operator with wildcards that are used in SQL, Cypher provides three separate operators with similar capabilities for substring searches. Recall that a substring search looks for a smaller piece of text inside a larger piece of text. The *starts with*, *contains*, and *ends with* keywords can be used to create criteria that start with, contain, or end with a specified character value. The following command uses the *starts with* operator to find the restaurant names that start with the word “Cheese”:

```
MATCH (res :Restaurant)
WHERE upper(res.name) STARTS WITH "CHEESE"
RETURN res.name
```

The next command finds restaurant names that contain the word “Cheese” anywhere within the name. The results of this command and the previous command can both be seen in Figure Q.14.

```
MATCH (res :Restaurant)
WHERE upper(res.name) CONTAINS "CHEESE"
RETURN res.name
```

FIGURE Q.14 SUBSTRING SEARCHES IN NEO4J



Interestingly, many of the string functions used in Cypher can also be used in the RETURN command to change the way that properties are displayed. For example, the substring function can be used in both the WHERE and RETURN clauses, as shown in the following command:

```
MATCH (res :Restaurant)
WHERE substring(res.phone, 1, 3) = "615"
RETURN res.name, substring(res.phone, 5) as phone
```

The substring function returns a part of a text value. Notice that the substring function accepts three parameter values. The first parameter specifies the text field to take the partial value from, the second parameter is which character position to start with, and the third parameter is how many letters of text to return. If the third parameter is omitted,

substring will return the rest of the string from the starting position. For the second parameter, the starting position, counting the characters begins with 0. So in the phone number (615) 555-1212, the left parenthesis is in position 0 and the area code begins in position 1. It can seem a little confusing at first because the first character is in position 0, the second character is in position 1, the third character is in position 2, and so, but with a little practice, you will get accustomed to it.

In the WHERE clause of the previous command, the substring looks in the restaurant phone number property, then, starting with the character in position 1, retrieves three characters (in this case, the second, third, and fourth characters from the phone property). In the RETURN clause, substring uses the restaurant's phone number again, and then returns all of the characters starting with the sixth character, as shown in Figure Q.15.

FIGURE Q.15 USING SUBSTRINGS IN WHERE AND RETURN



"res.name"	"phone"
"Beijing Table"	" 555-1571"
"Shanghai Home"	" 555-6521"
"Authentica Rustica"	" 555-6113"
"Free Range Salads"	" 555-2276"
"Elvenskirts"	" 555-2512"
"Thick Cuts Steakhouse"	" 555-2933"
"The Sicilian"	" 555-2751"
"NOLOs"	" 555-1708"
"La Grande"	" 555-8889"
"Racy Burgers"	" 555-1137"
"Ravmonds"	" 555-2481"

## Q-5d Retrieving Data on Relationships in Neo4j

At the heart of the need for graph databases is the ability to work with highly interconnected data. Therefore, data retrievals involving relationships are the most common type of queries. Fortunately, retrieving data on relationships is very similar to retrieving data on nodes, so the skills you have been developing in the previous sections will work very well with more complex queries too.

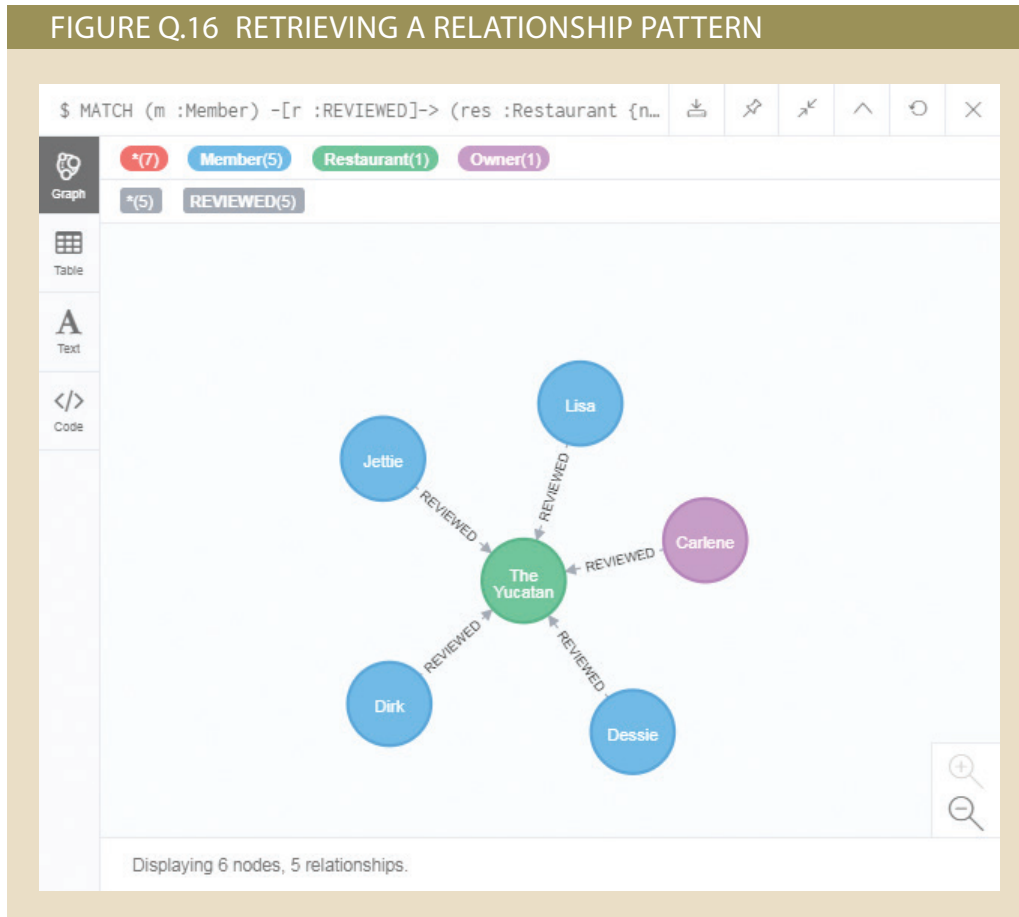
**Retrieving Data on Immediate Relationships** Recall that the MATCH command is used for pattern matching. In the patterns, nodes are represented with parentheses and relationships are represented with arrows. A relationship can be in only one direction, or it can be bidirectional. A relationship in one direction is specified in a pattern as `-->` or as `<--` depending on the direction.

Relationships can have labels, just like nodes can have labels. Relationships can have properties, just like nodes. In a Cypher query, a relationship can be assigned to a variable so that data about it can be returned, just like a node. The techniques for specifying labels and properties and assigning to variables are the same for relationships as nodes. In the FCC database, there is a directional “review” relationship between members and restaurants. The relationship is directional because members review restaurants, restaurants do not review members. The pattern would be `(Member) --> (Restaurant)`. Just as data about a node can be placed inside the node parentheses, data about the relationship can be placed inside the arrow. Data inside a relationship pattern is placed inside square brackets. The following command retrieves data that matches the pattern of a member reviewing a restaurant named The Yucatan.

```
MATCH (m :Member) -[r :REVIEWED]-> (res :Restaurant {name: "The Yucatan"})
RETURN m, r, res
```

If you view the result in graph view, as shown in Figure Q.16, you will see all of the members who have reviewed the specified restaurant. Changing to the table view, you can see the properties of the nodes as well as the properties of the relationships.

FIGURE Q.16 RETRIEVING A RELATIONSHIP PATTERN



Relationship properties can be included in the pattern of the query or in a WHERE clause. The following command retrieves every member who has reviewed the restaurant “Tofu for You” and rated the restaurant a “4” on taste.

```
MATCH (m :Member) -[r :REVIEWED {taste: 4}]-> (res :Restaurant {name: "Tofu
for You"})
```

```
RETURN m, r, res
```

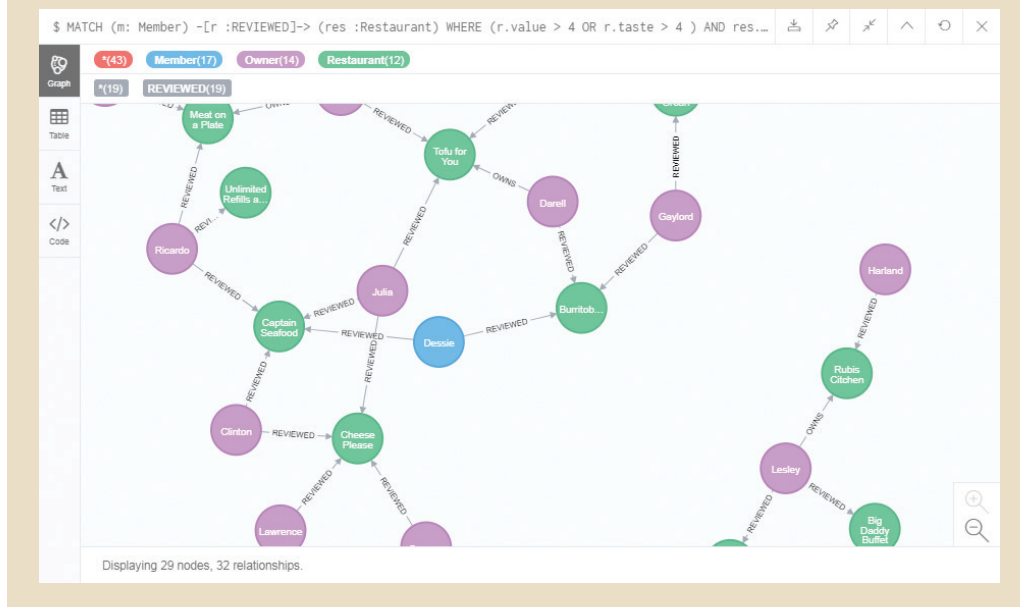
Again, to use inequalities or logical *or* connectors, the WHERE command can be used, as shown in the following command, with the results shown in Figure Q.17:

```
MATCH (m :Member) -[r :REVIEWED]-> (res :Restaurant)
```

```
WHERE (r.value > 4 OR r.taste > 4 ) AND res.state = "KY"
```

```
RETURN m, r, res
```

FIGURE Q.17 RETRIEVING A RELATIONSHIP PATTERN USING WHERE



The order in which the pattern is written does not affect the result. For example, the following command produces exactly the same result as the previous command:

```
MATCH (res :Restaurant) <-[r :REVIEWED]- (m :Member)
WHERE (r.value > 4 OR r.taste > 4) AND res.state = "KY"
RETURN m, r, res
```

When the direction of the relationship is not important, the relationship pattern omits the arrowhead, which matches on a relationship in either direction. Using a label for the relationship is optional as well. For example, if we want to see everyone associated with The Sicilian restaurant, the following command can be used:

```
MATCH (m :Member) -- (res :Restaurant {name: "The Sicilian"})
RETURN m, res
```

Notice that because a variable was not assigned to the relationship, the relationship could not be included in the RETURN. Therefore, even though the relationships show in the graph view, looking at the table view shows that none of the relationship data is returned.

Thus far, we have limited our queries to a single relationship; however, a query in Neo4j can involve multiple relationships using more than two nodes. A simple example would be to find the name of every owner of an Italian restaurant, as shown here:

```
MATCH (c :Cuisine {name: "Italian"}) <-- (res :Restaurant) <-[:OWNS]- (m :Member)
RETURN m.fname, m.lname
```

Recall from the previous discussion of using multiple patterns with the MATCH command, that if the same variable is used in multiple nodes, Cypher interprets this as requiring the same node in both patterns. The same is true when there are multiple references to a node in a relationship pattern. To include patterns that include multiple instances of a node label, but allow the nodes to be different, be certain to assign the nodes to

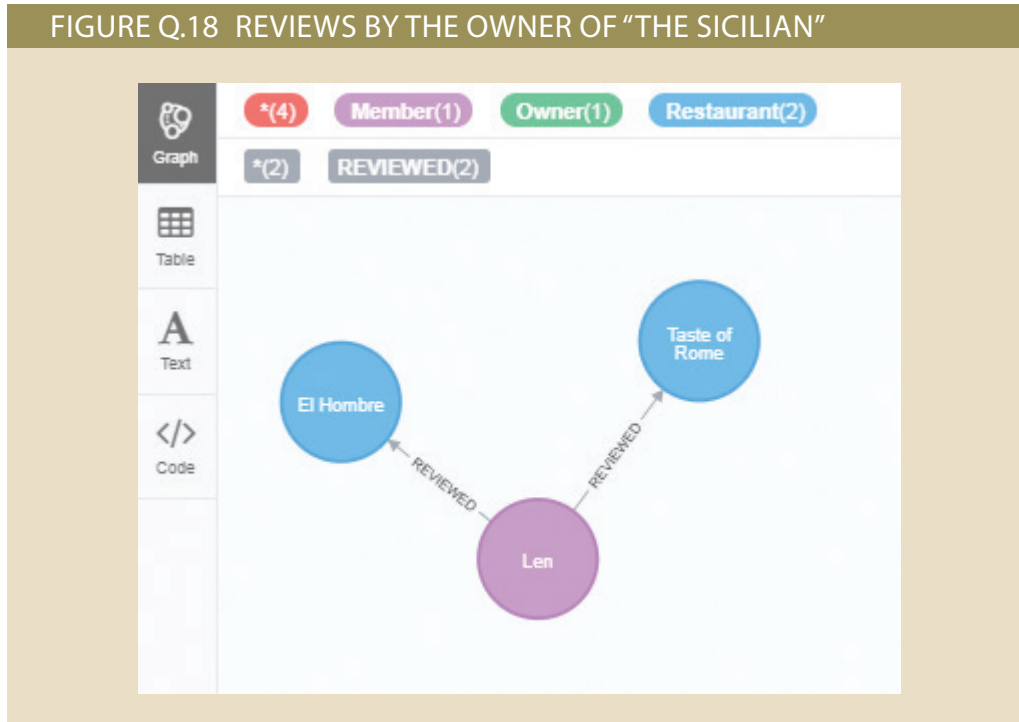


different variables. For example, the following command can retrieve all reviews completed by the owner of “The Sicilian” restaurant by using different variables for the two restaurant nodes. The results are shown in Figure Q.18.

```
MATCH (sicilian :Restaurant {name: "The Sicilian"}) <-[:OWNS]- (m :Member) -[r :REVIEWED]-> (res :Restaurant)
```

```
RETURN m, r, res
```

FIGURE Q.18 REVIEWS BY THE OWNER OF “THE SICILIAN”



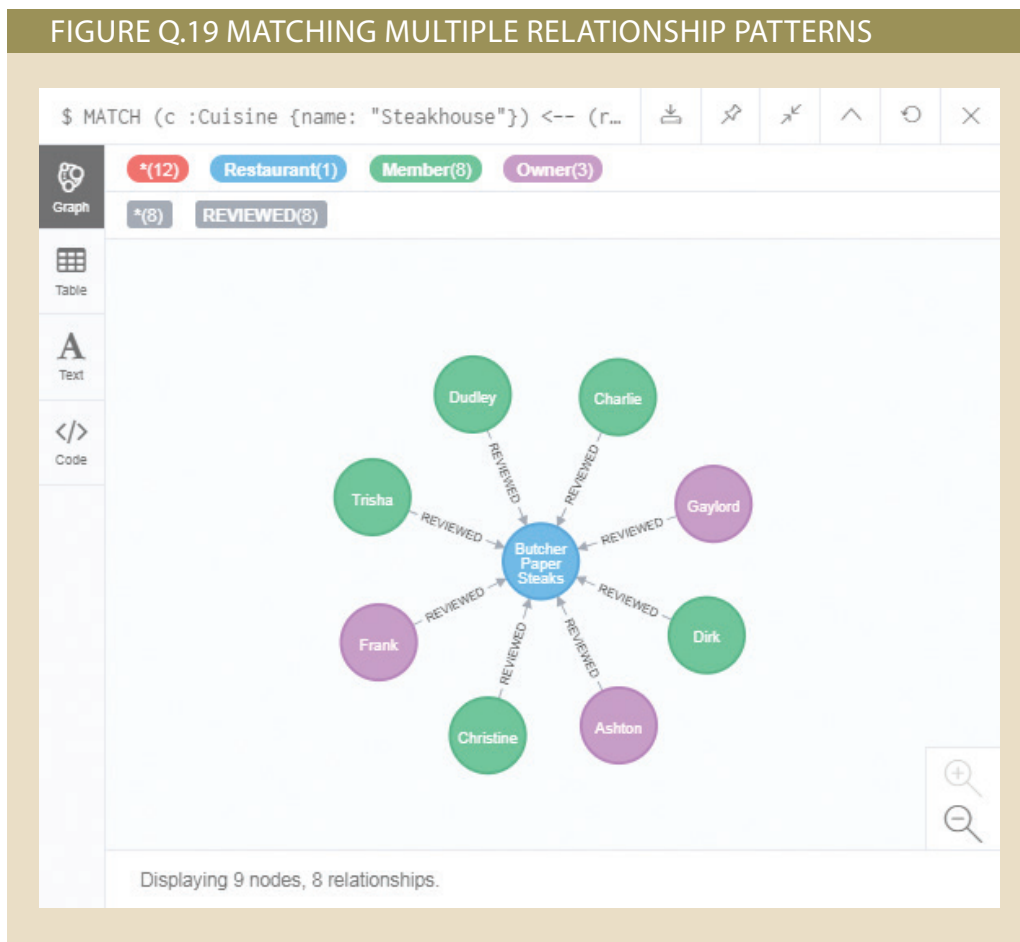
Using multiple patterns is also possible with relationships. In fact, if a query pattern requires more than two relationships connecting to the same node, then multiple patterns will be required. This requirement is logical if you consider that when written as a command, one relationship can be written on the left side of a node and another relationship can be written on the right side of a node, but there is no way to write a relationship above or below a node in command format. Therefore, multiple patterns are used. For example, imagine that member “Dirk Gray” is thought to be influential in the Food Critics Club, so we want to see everyone that has reviewed a steakhouse restaurant that was also reviewed by Dirk Gray. This will require a restaurant to have three relationships. It must have been reviewed by Dirk. The cuisine must be steakhouse. It must be reviewed by another member. The following command uses multiple patterns to provide the requested data, with the results shown in Figure Q.19.

```
MATCH (c :Cuisine {name: "Steakhouse"}) <-- (res :Restaurant) <-[r :REVIEWED]- (m :Member),
```

```
(res) <-[dr :REVIEWED]- (dirk :Member {fname: "Dirk", lname: "Gray"})
```

```
RETURN res, r, m, dirk, dr
```

FIGURE Q.19 MATCHING MULTIPLE RELATIONSHIP PATTERNS

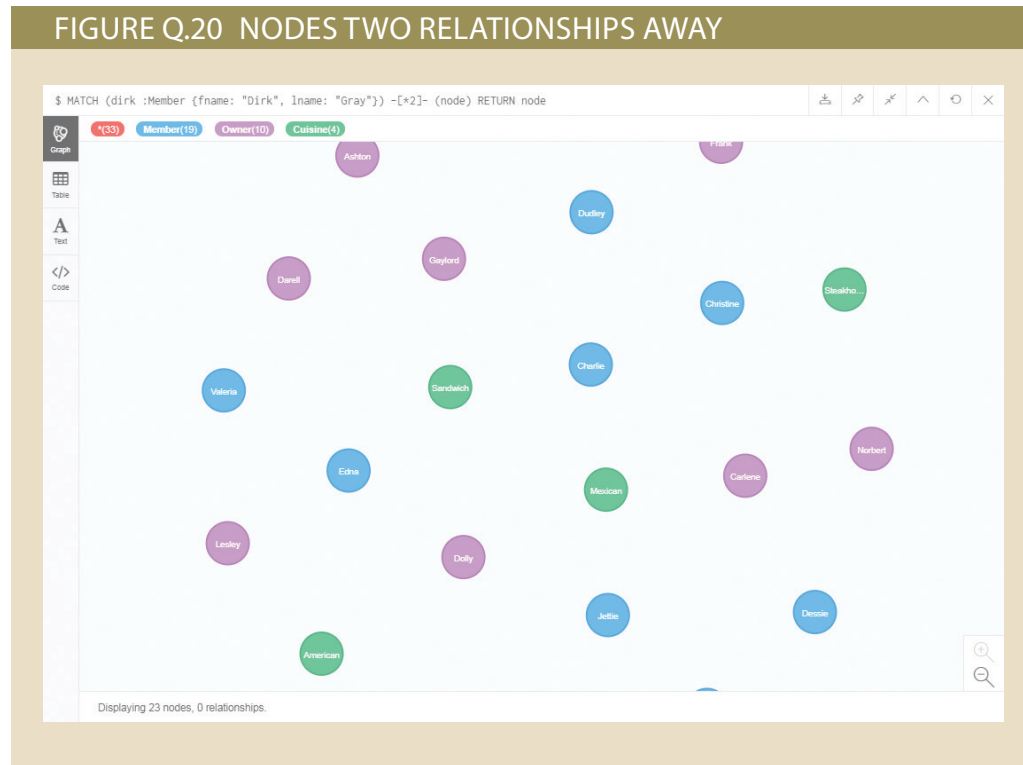


The previous command uses different variables for the two member nodes because we want the member in the first pattern to be any member, but we want the member in the second pattern to be one specific member who is different from the member in the first pattern. Similarly, we used different variables for the two reviewed relationships because we know that the reviews made by the other members are not the same as the reviews made by Dirk. However, we use the same variable for both restaurant nodes because we want the restaurants found in the first pattern to match the restaurants found in the second pattern.

**Retrieving Data on Distant Relationships** The previous section looked at immediate relationships, or relationships of one node being directly related to another node. The power of graph databases comes into play, however, when we begin to examine more distant relationships. As stated earlier, problems like six degrees of separation are what set graph databases apart in performance. Graph databases are designed for these types of problems, so Neo4j and the Cypher language make creating these types of queries very easy. If you understand retrieving data on immediate relationships, it is a simple modification to examine distant relationships. The only change is to specify a distance in the relationship of the pattern. The distance can be a specific number or it can be a range. Distances are indicated with an asterisk (\*) in the relationship. For example, the following command returns all nodes that are two relationships away from Dirk Gray,

with the results shown in Figure Q.20. The relationship pattern ignores direction and contains no label so that the query returns every node that is connected through any type of relationship.

```
MATCH (dirk :Member {fname: "Dirk", lname: "Gray"}) -[*2]- (node)
RETURN node
```



Notice that the 23 nodes returned by the previous command include members and cuisines, but not restaurants. Each restaurant that Dirk reviewed is one relationship away. Other members who own or reviewed those same restaurants would be two relationships away. The cuisines associated with those restaurants are two relationships away. The command, however, specified to only show the nodes that are exactly two relationships away from Dirk, so the intervening restaurants that are one relationship away are not included in the output. Moving from a distance of two to a distance of three, four, or more is just a matter of changing the distance specified inside the relationship. This is how Neo4j easily deals with the types of “friends of friends of friends of friends” problems faced by social media sites that relational databases cannot handle well, as discussed earlier in this chapter.



### Note

At first glance, Neo4j appears to provide conflicting information in Figure Q.20. At the top of the figure, the output shows `*(33)` indicating 33 nodes. At the bottom, it states that it is displaying 23 nodes. The discrepancy is because all owners are also members. Therefore, the 10 owner nodes are being counted twice because they are counted as members and then as owners.

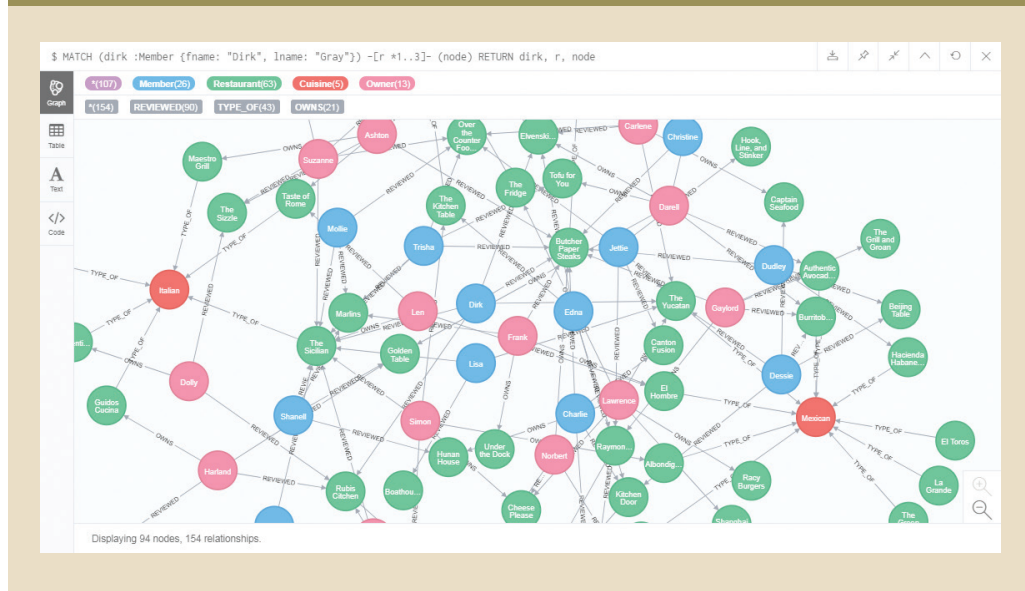
Instead of providing a specific distance, a distance range can be supplied that allows the intervening nodes and relationships to be included in the output. The following command modifies the previous command to display the distance as a range from one to three relationships away from Dirk.

```
MATCH (dirk :Member {fname: "Dirk", lname: "Gray"}) -[r *1..3]- (node)
```

```
RETURN dirk, r, node
```

Notice that when specifying a range for the distance, the beginning and ending values for the range are separated with two dots (.). As shown in Figure Q.21, as the distance grows the number of nodes and relationships returned increases significantly.

FIGURE Q.21 NODES WITHIN A DISTANCE RANGE



As noted before, the last two commands are retrieving nodes based on any type of relationship. If desired, the type of relationship can be restricted. For example, if we want to return only the members who are up to four relationships away from Dirk, but who are connected only through restaurant reviews, the following command can be used:

```
MATCH (dirk :Member {fname: "Dirk", lname: "Gray"}) -[r :REVIEWED *1..4]- (m :Member)
```

```
RETURN dirk, r, m
```

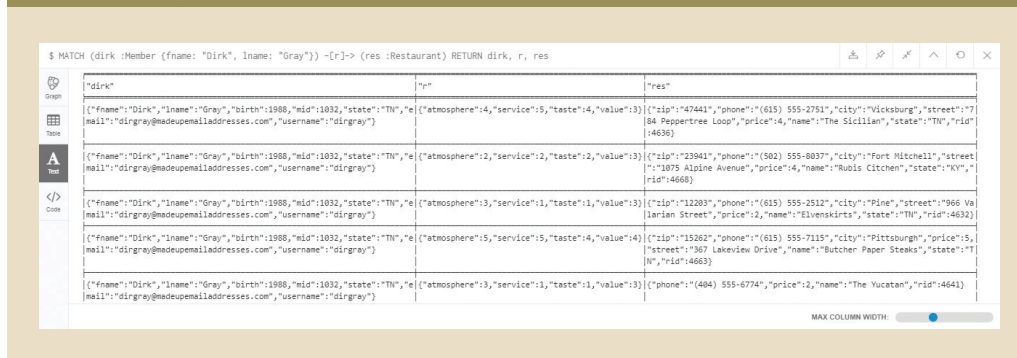
**Retrieving Paths in Neo4j** One way to conceptualize relationships is to think of them as paths. For example, there is a path from Dirk to the restaurant “Rubis Citchen” to the member Lesley Haywood. Like finding distant relationships, graph databases excel at finding paths because this is a primary task for graph databases. Finding paths is also simple to program in Neo4j because the syntax is very similar to finding distant or immediate relationships. We are still focused on pattern matching when querying Neo4j, even when we are looking for a path. That is because all relationship patterns are paths! When we look for immediate relationships, we are looking for a path with a length of one relationship. When we look for a distant relationship with a distance in the range of

\*1..3, we are looking for a path with a path length of three. When we look for a distant relationship with a specific distance of \*4, we are looking for the endpoint node for a path with a path length of four.

The difference between retrieving a path and other retrievals that we have done so far is primarily in the way the data is returned. Let's consider two similar commands. First, the following command retrieves data about every restaurant reviewed by member Dirk Gray, with the text view of the data shown in Figure Q.22.

```
MATCH (dirk :Member {fname: "Dirk", lname: "Gray"}) -[r]-> (res :Restaurant)
RETURN dirk, r, res
```

FIGURE Q.22 TEXT VIEW OF RESTAURANTS REVIEWED BY DIRK GRAY



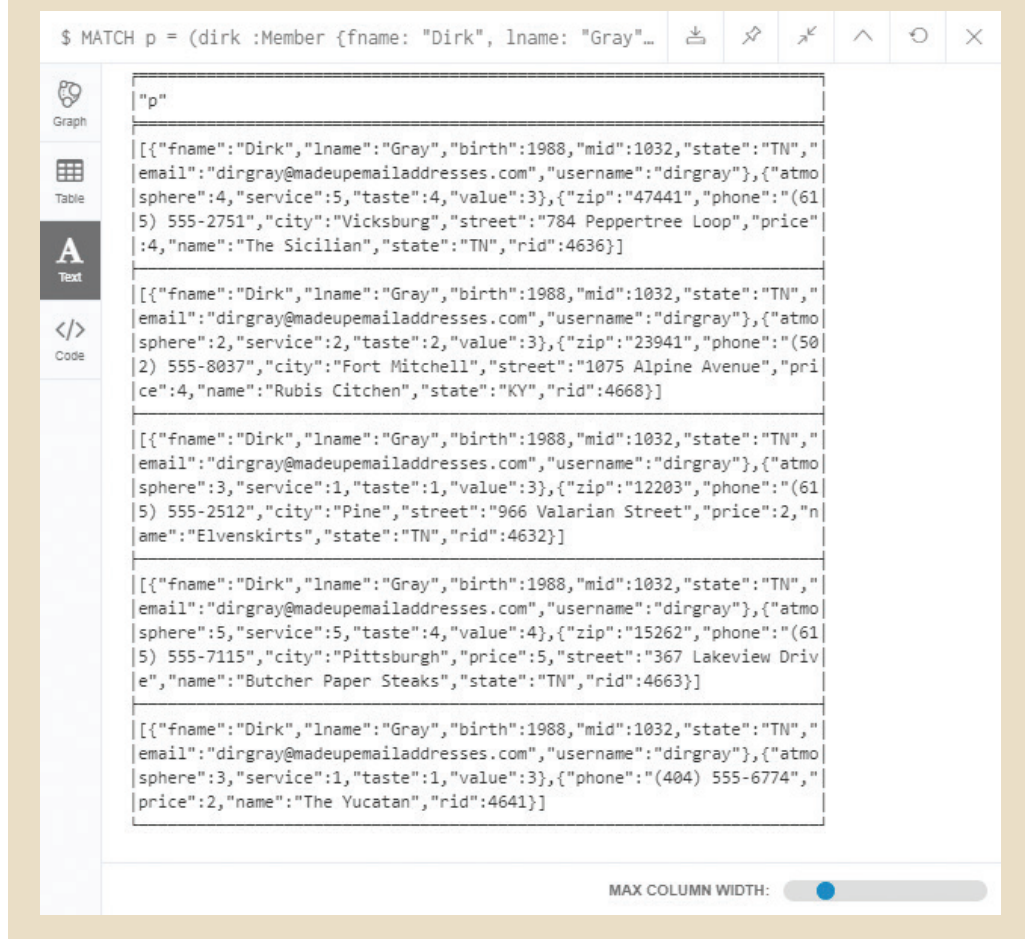
"dirk"	"r"	"res"
[{"fname": "Dirk", "lname": "Gray", "birth": 1988, "mid": 1032, "state": "TN", "email": "dirgray@madeupemailaddresses.com", "username": "dirgray"}]	[{"atmosphere": 4, "service": 5, "taste": 4, "value": 3}]	[{"zip": 47441, "phone": "(615) 555-2751", "city": "Vicksburg", "street": "7784 Peppertree Loop", "price": 4, "name": "The Stillen", "state": "TN", "rid": 4636}]
[{"fname": "Dirk", "lname": "Gray", "birth": 1988, "mid": 1032, "state": "TN", "email": "dirgray@madeupemailaddresses.com", "username": "dirgray"}]	[{"atmosphere": 2, "service": 2, "taste": 2, "value": 3}]	[{"zip": 23043, "phone": "(502) 555-8837", "city": "Fort Mitchell", "street": "13075 Alpine Avenue", "price": 4, "name": "Rudis Citchen", "state": "KY", "rid": 4668}]
[{"fname": "Dirk", "lname": "Gray", "birth": 1988, "mid": 1032, "state": "TN", "email": "dirgray@madeupemailaddresses.com", "username": "dirgray"}]	[{"atmosphere": 3, "service": 1, "taste": 1, "value": 3}]	[{"zip": 12203, "phone": "(615) 555-2512", "city": "Pine", "street": "966 W Larian Street", "price": 2, "name": "Elvenshirts", "state": "TN", "rid": 4632}]
[{"fname": "Dirk", "lname": "Gray", "birth": 1988, "mid": 1032, "state": "TN", "email": "dirgray@madeupemailaddresses.com", "username": "dirgray"}]	[{"atmosphere": 5, "service": 5, "taste": 4, "value": 4}]	[{"zip": 15262, "phone": "(615) 555-7115", "city": "Pittsbourg", "price": 5, "street": "1307 Lakeview Drive", "name": "Butcher Paper Steaks", "state": "TN", "rid": 4663}]
[{"fname": "Dirk", "lname": "Gray", "birth": 1988, "mid": 1032, "state": "TN", "email": "dirgray@madeupemailaddresses.com", "username": "dirgray"}]	[{"atmosphere": 3, "service": 1, "taste": 1, "value": 3}]	[{"phone": "(404) 555-6774", "price": 2, "name": "The Yucatan", "rid": 4641}]

Second, the following command retrieves data about the path for every restaurant reviewed by member Dirk Gray and assigns that path to the variable *p*. The text view of the data returned is shown in Figure Q.23.

```
MATCH p = (dirk :Member {fname: "Dirk", lname: "Gray"}) -[r]-> (res :Restaurant)
RETURN p
```



FIGURE Q.23 TEXT VIEW OF THE PATH OF REVIEWS BY DIRK GRAY



If you only view the graph view of the two commands, the results look identical. However, when looking at the table or text view, the difference in the data returned is unmistakable. The data in Figure Q.22 shows each variable, the node for Dirk, the relationship, and the node for the restaurant, being returned as separate values in separate objects. The data in Figure Q.23 shows the path being returned as a single value containing an array of objects. When querying interactively, this may seem like a small detail, but when embedded in a program, the difference is quite significant.

In addition to the changes in the way paths can be handled programmatically in applications, placing the path inside an array simplifies the ability to compare the lengths of paths by comparing the sizes of the arrays. This means that it is easy for Neo4j to complete tasks like finding the shortest path between two nodes because the DBMS merely has to compare the sizes of the arrays. This is done using a [shortestPath\(\)](#) function in Cypher along with an unspecified relationship distance. The `shortestPath()` function takes a relationship pattern as a parameter. To create a relationship of unspecified length, use the asterisk to indicate a distant relationship, but do not provide a distance length or range. For example, the following command finds the shortest path, based on restaurant reviews, between members Dirk Gray and Isiah Horn, with results shown in Figure Q.24. The results indicate that the shortest path based only on restaurant reviews has a path length of six.

### **shortestPath()**

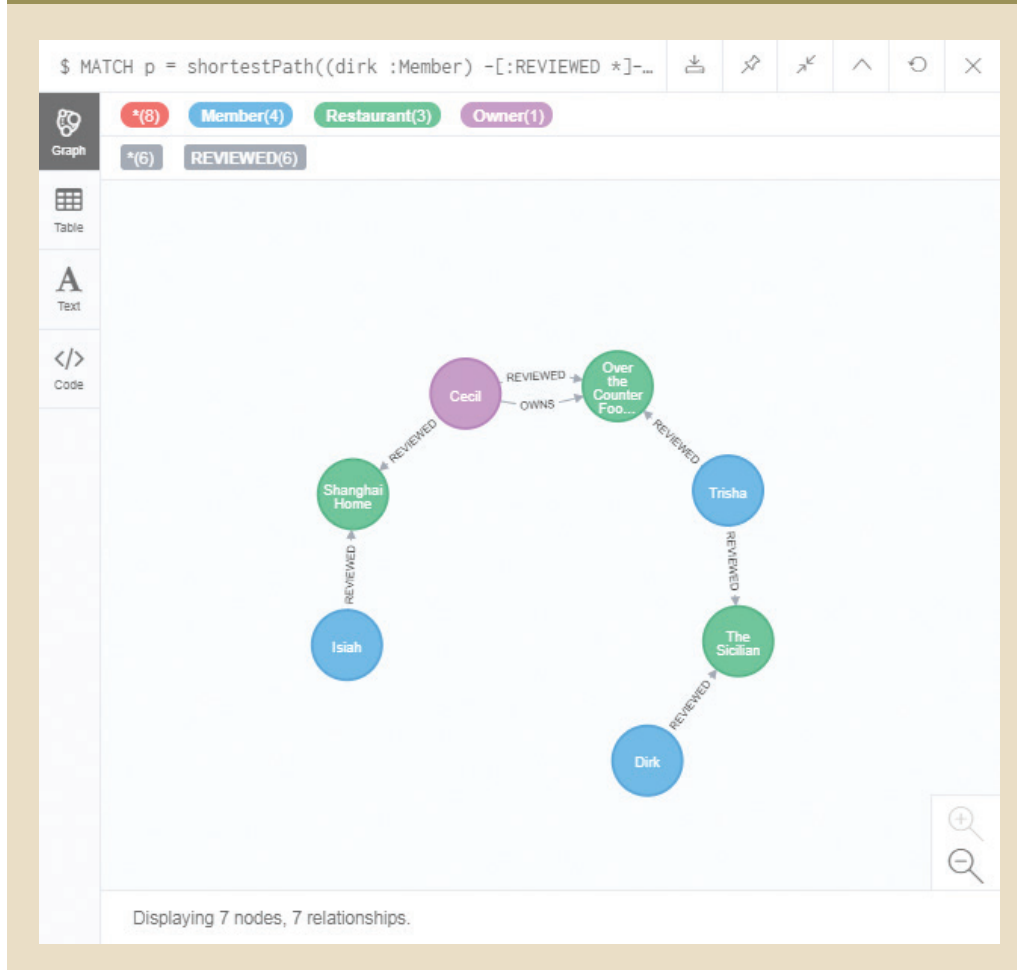
A function in the Cypher language used by Neo4j to find the path with the shortest distance between two given nodes.

```

MATCH p = shortestPath((dirk :Member) -[:REVIEWED *]- (isiah :Member))
WHERE dirk.fname = "Dirk" AND dirk.lname = "Gray" AND isiah.fname = "Isiah"
AND isiah.lname = "Horn"
RETURN p

```

FIGURE Q.24 SHORTEST PATH BETWEEN TWO SPECIFIED MEMBERS



**Creating Relationships in Neo4j** Once you are comfortable with matching patterns, creating new relationships is straightforward. The simplest way to create a relationship between two nodes is to match the two nodes, and then specify the new relationship. For example, to create a restaurant review between Dirk Gray and The Sicilian, match the member node for Dirk, match the restaurant node for The Sicilian, and then issue a CREATE command that includes the pattern for the relationship desired:

```

MATCH (dirk :Member {fname: "Dirk", lname: "Gray"}), (sicilian :Restaurant {name: "The Sicilian"})
CREATE (dirk) -[:REVIEWED {taste: 4, service: 5, atmosphere: 4, value:3}]-> (sicilian)

```

This appendix has introduced you to working with Neo4j as an example of a graph database. Graph databases are an important and growing segment of NoSQL database options that organizations are using to improve their ability to work with highly interdependent



data. You have been introduced to creating a graph database through the creation of nodes and relationships. You have seen that nodes are often indicated as being similar by sharing the same labels, but that even nodes with the same label can vary in their properties. Allowing the nodes to vary in properties makes Neo4j a schema-less database. In addition to properties of the nodes, relationships can also have properties. The graph can be traversed, or queried, using pattern matching. Patterns can specify nodes, relationships, and properties to identify paths through the graph. The ability to quickly match complex patterns allows graph databases to excel at queries that focus on the relationships among the data.

## Summary

- Neo4j is a graph database that stores data as nodes and relationships, both of which can contain properties to describe them.
- Neo4j databases are queried using Cypher, a declarative language that shares many commonalities with SQL, but is still significantly different in many ways.
- Data retrieval is done primarily through the MATCH command to perform pattern matching.

## Key Terms

Cypher, Q-4  
match, Q-6

return, Q-6  
shortestPath(), Q-28

## Review Questions

1. Explain the difference between using the same variable name and different variable names when matching multiple patterns in Neo4j.
2. What is the difference between using WHERE and embedding properties in a node when creating a pattern in Neo4j?

## Problems

For the following problems, use the Food Critics Club (FCC) graph database that was created and used earlier in the text for use with Neo4j.

1. Create a node that meets the following requirements. Use existing labels and property names as appropriate.

The node will be a member, and should be labeled as such, with member id 5000.

The member's name is "Abraham Greenberg".

Abraham was born in 1978, and lives in the state of "OH".

Abraham's email address is `agreen@nomail.com`, and his username is `agberg`.

2. Create a restaurant node with restaurant id is 10000, the name "Hungry Much", and located in Cobb Place, KY.
3. Update the "Hungry Much" restaurant created above to add the phone number "(931) 555-8888", and a price rating of 2.
4. Create a REVIEWED relationship between the member created above and the restaurant created above. The review should rate the restaurant as a 5 on taste, service, atmosphere, and value.
5. Create a REVIEWED relationship between member Frank Norwood and the restaurant created above. The review should rate the restaurant as a 4 on taste, service, and value, and rate the restaurant as a 2 on atmosphere.
6. Write a query to display member Frank Norwood and every restaurant that he has rated as a 4 or above on value.
7. Write a query to display cuisine, restaurant, and owner for every "American" or "Steakhouse" cuisine restaurant.
8. Write a query to return the shortest path based only on reviews between members Abraham Greenberg and Herb Christopher.

### Online Content



The `Ch14_FCC.txt` file to create the graph database used in these questions was also used in Chapter 14. The file is available at [www.cengagebrain.com](http://www.cengagebrain.com). The contents of the file should be copied and pasted into the Neo4j editor bar and executed using the play button in the interface.