Cypher

Paul Warren, paul.warren@open.ac.uk; *Paul Mulholland*, paul.mulholland@open.ac.uk

N.B. this handout is to be read prior to undertaking the study, and to be retained for reference at any time during the study. The final page contains a summary of all you need to know.

1 Introduction

The purpose of this study is to understand the difficulties people experience using a graph database querying language known as Cypher[1]. No prior knowledge or experience is required for the study; all the necessary features are explained in this handout. The object is to understand how people manipulate the language constructs mentally, so please do not use pen and paper.

The structure of this handout is as follows. Section 2 describes the basics of Cypher graph databases. Section 3 describes how to use Cypher to create a graph database. Section 4 then provides an overview Cypher's querying facilities. Sections 5 and 6 describe some specific features of Cypher which are used in this study. Section 7 gives an overview of how the study is organised. Finally, there is a one-page summary which provides all the necessary information in compact form.

Note that this handout describes only a simplified subset of Cypher required to take part in the study. The full language contains many other features.

2 Graph databases in Cypher

A graph database, as created by Cypher, consists of nodes joined by edges. The nodes can represent, e.g. people, towns, companies. Nodes are written as round brackets. These brackets may be empty, e.g. *()*. Nodes may have labels, which begin with a colon, e.g. *(:Dog)*; N.B. in our study nodes have at most one label. Nodes may also have properties, e.g. *({name: 'Fido'})* is a node with a property *name:* which has value *'Fido'*. As in this example, properties are written with a terminating colon and enclosed within curly brackets. Their values are either integer numbers or character strings, in quotes. A node may have more than one property. In this case, the property / value pairs are enclosed within the same pair of curly brackets, and separated by commas, e.g. *({name: 'Mary', role: 'lawyer'})*. A node may also have a label and one or more properties, e.g. *(:Dog {name: 'Fido'}).*

Nodes can be connected by edges. These are written as square brackets. They always have one, and only one, type which begins with a colon, e.g. *[:worksFor]*. They may have any number of properties, written just as for nodes, e.g. *[:worksFor {from: 2000, until: 2010}]*. Nodes always have directionality, indicated by an 'arrow' at one end, e.g. the following indicates that Mary worked for BigCo between 2000 and 2010:

({name: 'Mary'}) –[:worksFor {from: 2000, until: 2010}]–>({name: 'BigCo'})

3 Creating graph databases with Cypher

Cypher uses the CREATE command to add nodes and edges to a graph database. For example, to create a node with label *:Dog* and a property *name:* with value *'Fido'*, we can write:

[1] Cypher has been developed by Neo4j, see https://neo4j.com/developer/cypher-query-language/

CREATE (:Dog {name: 'Fido'})

We can create two nodes and an edge between them in one command, e.g.

CREATE ({name: 'Stephen'})–[:marriedTo]–>({name: 'Mary'})

This creates two nodes connected by an edge, indicating that Stephen is married to Mary. If we want to use a CREATE command to create a network of nodes, then we need to use variables to indicate when a node is reused. Consider the following CREATE command:

CREATE ({name: 'Stephen'})–[:marriedTo]–>({name: 'Mary'}),
({name: 'John')–[:brotherOf]–>({name: 'Mary'})

Note that the first line ends in a comma, indicating that the second line is a continuation of the first, i.e. part of the CREATE command. This would appear to create three nodes, indicating that Stephen is married to Mary, and John is the brother of Mary. However, this CREATE command would actually create four nodes. This is because *name:* is just a property. It has no special status and it does not indicate that the node with name *'Mary'* in the first line is the same as the node with name *'Mary'* in the second line. Thus, Stephen is married to one Mary, whilst John is the brother of a different Mary. To indicate that two nodes are the same, we use a variable. Here we use the variable *a* to indicate that the node at the end of the first line is the same as the node at the end of the second line:

CREATE ({name: 'Stephen'})–[:marriedTo]–>(a {name: 'Mary'}),
({name: 'John'}) –[:brotherOf]–>(a)

This creates three nodes: Stephen, Mary, and John. Stephen is married to Mary, and John is the brother of Mary. In the first line we associate the property *name:* with the node represented by the variable *a*, and give the property the value *'Mary'*. Then *a*, at the end of the second line, refers to the same node, which has the property *name:* with value *'Mary'*.

Note that *a* is just a variable with scope limited to that one CREATE command. It has no existence outside that command. It is not stored in the database. Its sole purpose is to enable a node to be referred to several times within one command. Purely for the purposes of this study, Cypher variables are written as single lowercase letters.

In a single CREATE command it is possible to assign any number of properties to the nodes and edges, e.g.

CREATE ({name: 'Mary'}) –[:worksFor {role: 'lawyer', from: 2000}]
->({name: 'BigCo'})

This creates:

- A node with a property *name:* with value *'Mary'*.
- A node with a property *name:* with value *'BigCo'*.
- An edge between these nodes with type *:worksFor*, a property *role:* with value '*lawyer'* and a property *:from* with value *2000*. The edge also has directionality.

For any given node or edge, a property can only take one value. If we want to indicate that Mary started working for BigCo as a lawyer in 2000 and became a manager in 2010, we cannot write:

CREATE ({name: 'Mary'}) –[:worksFor {role: 'lawyer', from: 2000, role:
'manager', from: 2010}]-> ({name: 'BigCo'})

This attempts to assign two values to *:role* and two values to *:from* for one particular edge. This makes no sense; in fact, what actually happens is that the second occurrence of each property value overwrites the first. However, it is possible to create two edges of the same type between the same two nodes. So, we can achieve the required effect by writing:

CREATE (a {name: 'Mary'}) –[:worksFor {role: 'lawyer', from: 2000}]
-> (b {name: 'BigCo'}),
(a) –[:worksFor {role: 'manager', from: 2010}]-> (b)

Note here the use of the variables *a* and *b* to indicate that the two nodes in the second part of the CREATE command are the same as the two nodes in the first part.

4 Querying with Cypher

To query with Cypher, we use the MATCH command, e.g.

MATCH (a) –[:brotherOf]-> (b) RETURN a.name, b.name

Here, *a* and *b* are variables, and the RETURN clause is used to return the values of the *name:* property of these variables. Hence, using the database from the previous section, the query would return *'John', 'Mary'*.

We can use properties in a query to identify a node. The following will identify the node with *name: 'John'*, and then use the edge *:brotherOf* to reach the node with *name: 'Mary'*:

MATCH ({name: 'John'}) –[:brotherOf]-> (a) RETURN a.name

We can also use properties in a query to identify an edge. For example, using the example at the end of the last section, if we wished to know when Mary began work as a manager (as distinct from as a lawyer) at BigCo, we could write:

MATCH ({name: 'Mary'})-[a:worksFor {role: 'manager'}]->
({name: 'BigCo'}) RETURN a.from

Given that the CREATE command at the end of the last section created two edges between Mary and BigCo, this MATCH command indicates that we are interested in the edge for which the property *:role* has value *'manager'*, rather than the value *'lawyer'*.

5 More complex queries

MATCH clauses can contain any number of edges and nodes. For example, suppose we wish to know for which company Stephen's wife works. We can write:

MATCH ({name: 'Stephen}) –[:marriedTo]-> (a) –[:worksFor]-> (b) RETURN b.name

Using the database created in Section 3, this will return *'BigCo'*. In this query we have a variable, *a*, which is not used in the RETURN clause. We can avoid using this variable, and simply write the middle node as a pair of empty round brackets:

MATCH ({name: 'Stephen}) –[:marriedTo]-> () –[:worksFor]-> (a) RETURN a.name

In this example the two edges both 'pointed' from left to right. However, it is possible to create a MATCH command with edges pointing in different directions. Suppose we wish to know who is the brother-in-law of Stephen. The following query will return *'John'*:

MATCH ({name: 'Stephen'}) –[:marriedTo]-> () <-[:brotherOf]- (a) RETURN a.name

As a more complex example of a query, considering again the database created at the end of Section 3, we may wish to know for which companies Mary works, with what roles and from when. We can write:

MATCH ({name: 'Mary'}) –[a:worksFor]-> (b) RETURN a.role, a.from, b.name

This query would return two sets of results:

'lawyer', 2000, 'BigCo'
'manager', 2010, 'BigCo'

It is possible to repeat a given edge any number of times. We do this by using the star operator (*). We can illustrate this with a new database illustrating the relationship between animal groups, with just two edges:

CREATE ({group: 'Monkey})-[:subGroupOf]->
({group: 'Primate')-[subGroupOf]->({group: 'Mammal')

This creates three nodes and defines a hierarchical relation between them. We wish to know all the subgroups and sub-subgroups of the *group: 'Mammal'*. We can write the query:

Match (a)-[SubGroupOf]-> ({group: 'Mammal'}) RETURN a.group*

The * symbol means that the edge can appear once, or be concatenated with itself an unlimited number of times, i.e.

-[SubGroupOf]->
-[SubGroupOf]->() -[SubGroupOf]->
-[SubGroupOf]->() -[SubGroupOf]->() -[SubGroupOf]->
…

Hence, the query will output:

'Primate', 'Monkey'

Here, *group: 'Primate'* is a subgroup of *group: 'Mammal'*, and *group: 'Monkey'* is a subgroup of *group: 'Primate'*. If the database contained a subgroup of *group: 'Monkey'*, that would also be returned, and so on indefinitely.

6 WHERE clause, equality checking and labels() function

Sometimes we may wish to impose additional constraints in a query. We can do this by adding a WHERE clause. For example, suppose we have a graph containing the two nodes created by the following command:

CREATE ({name: 'Joe', livesIn: 'London'}), ({name: 'Sue', livesIn: 'London'})

We wish to output the name of the person who lives in the same city as 'Joe'. We can do this with:

MATCH (a {name: 'Joe'}), (b) WHERE a.livesIn = b.livesIn RETURN b.name

This query will output *'Sue'*. Here, we have a WHERE clause which uses the equals sign to make a test for equality between two property values.

In Section 2, we made a distinction between a node label, *e.g. :Horse*, and a character string serving as a property value, e.g. *'Horse'*. We may wish to compare node labels and character strings. We can do this using the *labels()* function and the IN keyword. The argument to the

labels() function is a variable representing a node. The function returns a list containing the string representations of the labels associated with the node, without the initial colon. So, if we have a node with label *:Horse*, represented by a variable *a*, then *labels(a)* will be a list containing just *'Horse'*. We can then test whether a particular string is in that list by using the IN keyword and *labels()* function in a WHERE clause.

To illustrate the use of these features, suppose we have nodes representing individual animals, e.g.

(:Horse {name: 'Arkle'}), (:Horse {name: 'Red Rum'}), (:Zebra {name: 'Stripey'})

Here we are using labels *(:Horse, :Zebra)* to represent the type of animal. Now suppose we want to output the names of those animals with label *:Horse*; i.e. we want to output *'Arkle'* and *'Red Rum'*. We could write:

MATCH (a :Horse) RETURN a.name

However, to illustrate the use of *labels()*, we can achieve the same effect by writing:

MATCH (a) WHERE 'Horse' IN labels(a) RETURN a.name

In our illustrative example, the first alternative is the simpler. However, one of the questions in the study illustrates a more realistic usage.

7 Overview of study

The study consists of 15 questions, the first two of which are practice questions to get you used to the format. Your responses to these will not be used in the subsequent analysis. There are two types of questions: modelling questions and querying questions.

In the modelling questions you are presented with some information in English; one or more queries, in English, that one might want to ask; and some alternative models in Cypher. You are asked to indicate which of these models are correct. Here, 'correct' means that the Cypher model accurately represents the information and that Cypher queries can be constructed to represent the English queries. You are also asked to rank the models. You can use whatever criteria you wish to perform this ranking, and it is possible to give two or more models the same rank. Optionally, you can add free format textual comment for each model.

In the querying questions you are provided with a model in Cypher; a query in English; and some alternative queries in Cypher. You are asked to indicate which of the Cypher queries correctly represent the English query, when applied to the Cypher model.

For each question, there may be any number of correct responses. None may be correct; or they may all be correct; or some may be correct and some may be incorrect. For each proposed response, you will need to click to indicate 'correct' or 'incorrect'. You can change your response, by clicking on the alternative button, at any time until you click on 'Submit and continue' at the bottom right. When you do click on 'Submit and continue', you move on to the next page. It is not possible to return to a previous page. Please do not attempt to do so with the browser 'back' button. Finally, please note that this project has been reviewed by, and received a favourable opinion from, The Open University Human Research Ethics Committee, reference HREC/3568.

THE FOLLOWING PAGE SUMMARISES EVERYTHING YOU NEED TO KNOW

Summary

Cypher conventions

- Nodes are enclosed within round brackets; edges within square brackets.
- Nodes have zero, one or more labels and any number of property / value pairs.
- Edges have one type and any number of property / value pairs. They also have directionality, indicated by an 'arrow', i.e. ->.
- Node labels and edge types begin with a colon, e.g. *:Dog, :marriedTo.*
- Properties end in a colon; their values are character strings in quotes or integers. All the property value pairs associated with a node or edge are enclosed in curly brackets, e.g. *{name: 'Mary', role: 'lawyer'}*
- Cypher keywords are written in capitals, e.g. *CREATE, MATCH, RETURN*
- Purely for the purposes of this study, variables are written as single lowercase letters.

CREATE command

- To create a single node, we can write, e.g. *CREATE (:Dog {name: 'Fido'})*
  this creates a node, with label *:Dog* and a property *:name* with value *'Fido'*
- To create two nodes and an edge, with properties, e.g.
  CREATE ({name: 'Mary'})–[:worksFor {from: 2000}]–>({name: 'BigCo'})
- To create three nodes, with edges between them, e.g.
  CREATE ({name: 'Stephen'})–[:marriedTo]–>(a {name: 'Mary'}),
  ({name: 'John'}) –[:brotherOf]–>(a)
  Note the use of the variable *a* to indicate that two nodes are the same.
- For any given node or edge, a property can only take one value. However, we can have more than one edge of the same type between the same two nodes. Thus, we can write:
  CREATE (a {name: 'Mary'}) –[:worksFor {role: 'lawyer', from: 2000}]
  -> (b {name: 'BigCo'}),
  (a) –[:worksFor {role: 'manager', from: 2010}]-> (b)

Querying with the MATCH and RETURN keywords

- Queries begin with MATCH and RETURN a property value of a variable, e.g.
  MATCH ({name: 'John'}) –[:brotherOf]-> (a) RETURN a.name
- MATCH commands can contain more than one edge, e.g.
  MATCH ({name: 'Stephen}) –[:marriedTo]-> (a) –[:worksFor]-> (b) RETURN b.name
- Nodes do not need to contain a variable, e.g. the middle node above could be *()*
- Edges in a query can be of opposite directionality, e.g.
  MATCH ({name: 'Stephen'}) –[:marriedTo]-> () <-[:brotherOf]- (a) RETURN a.name
- The star operator (*) is used to indicate repetition of an edge an unlimited number of times, e.g. *-[:SubGroupOf*]->*
- Queries can also return property values of edges, e.g.
  MATCH ({name: 'Mary'}) –[a:worksFor]-> (b) RETURN a.role, a.from, b.name

WHERE, equality checking and labels() function

- The WHERE clause is used to provide extra constraints in a query, e.g.
  MATCH (a {name: 'Joe'}), (b) WHERE a.livesIn = b.livesIn RETURN b.name
  This query outputs the name of everyone who lives in the same place as Joe.
- The labels() function takes an argument which is a variable representing a node and returns a list of the node labels, converted to character strings, and without the initial colon. It can be used to compare a character string and a node label, e.g.
  MATCH (a) WHERE 'Horse' IN labels(a) RETURN a.name