Neo4J: Graph Database

Giuseppe Burtini, Graeme Douglas and Yipin Guo

February 24, 2013

Basics

- **Neo4J** is a data storage and query system designed for storing graphs.
- Data as a series of relationships, modelled as a directed graph.
- Recall, a graph is a pair of sets: G(V, E) – vertices and edges
- Neo4J defines a query language called "Cypher" which allows "iterative" or "search" style queries.
- Every vertex **and** every edge can store data.

Data Representation

- Graphs are a pure generalization of the traditional relational model – anything that can be represented in the relational model can be represented in a graph.
  - Think of "relationships" as "foreign keys".
- Node data is stored as JSON "documents."
- Vertices can store "relationship" types.
- Queries are "graph traversals" – i.e., walk along the graph checking some condition, deciding where to go next.

So, what can you do with it?

- Graph databases are great for storing data that is intrinsically graphical.
- For example, human relationships, preference lists, networks.
- Of course, the graph representation is a generalization of the relational model, so you can actually represent anything you would have used an RDBMS for.
- The "iterative" query model makes recursive queries much easier in Cypher than SQL.
- For example, hierarchies or other complex tree relationships.

Introduction

Cypher is a declarative language similar to SQL, comprised of distinct clauses.

In SQL, consider clauses like "SELECT something," "FROM somewhere" and "WHERE some condition" as the clauses.

In Cypher, the important clauses are "START", "MATCH", "WHERE" and "RETURN".

A full query example.

```
1 START movie=node:node_auto_index(m_id="603")
2 RETURN movie;
```

This will grab all items with movie ID 603 as the starting point, and then immediately return them[1].

- START designates which nodes to start from (you can start from multiple nodes!)
- RETURN designates what to return – in this case, the node itself.

[1] Presumably there's only 1, as this is an ID.

A more complicated query.

```
1 START person=node:node_auto_index(name="Graeme Douglas")
2 MATCH person--friend
3 RETURN friend.name;
```

This query will grab all Graeme's friends (where "friend" is defined in either direction)

- START finds the node with name Graeme Douglas
- MATCH designates what things we want to find (do the traversal, test matches)
  - -- means "has a relationship in either direction" (i.e., at least one of them considers the other a friend)
- RETURN says to return all the friends names.

Essential clauses.

- **START** defines starting points in the graph to search from. These are retrieved by referencing element IDs or via index lookups.
- **MATCH** the graph pattern to match. Bound to the elements defined in START and RETURN
- **WHERE** predicate used to restrict results – think a selection node from relational algebra
- **RETURN** description of the data to return.

START clause.

Start clauses determine which nodes to start traversing from. This is not just a performance consideration: it can change the results, as graphs are not necessarily connected.

- `START n = node(*)` means start from every node (and call every node "n" in the other clauses)
- Note that you assign variables in the START clause which are referenced in other clauses.
- `START n = node(1,2,3)` start from nodes 1, 2 and 3.
- `START a=node(1), b=node(2)` start from nodes 1 and 2 simultaneously.
- Don't worry too much if start points are confusing to you (yet!). It'll all make sense soon.

MATCH clause.

- A match clause takes one or more patterns ("a path") which indicates how to find what you would like to match.
- Using a name in a match clause makes that name available elsewhere in the query
  - For example, in a WHERE or RETURN clause
- An empty set of brackets () can be used in place of a name if you only care about the relationship.
- Relationships can be indicated with:
  - `a--b`: a relationship in any direction
  - `a-->b`: a relationship from the left (a) to the right (b)
  - `a-[likes]-b`: a relationship of type "likes" in any direction.
  - `a-->()<--b`: all node pairs which have an outgoing relationship to any third common node

A note on depth control.

By default, when creating a path in the match clause, you're talking about one level deep.

- A path can have variable depth by simply placing an asterisk "*" at the end of the square brackets in a relationship
  - `a-[?*]->b` describes any two nodes where there is some path from a to b, at any depth
- A minimum/maximum depth can also be set by following the asterisk with a range "min..max"
- `a-[*2]->b` describes all nodes a and b where the depth of the path is at least 2 relationships.
- `a-[*2..5]->b` describes all nodes a and b where the depth of the path is at least 2 relationships and no more than 5 relationships.
- `a-[*2..2]->b` describes all nodes a and b where the depth of the path is exactly 2 relationships

WHERE clause.

This is just like a SQL where clause, it reduces the result set to those that match some predicate

- `follower.name =~ 'S.*'` – match all names starting in S and followed by any number of any character
- `(n.age < 30 and n.name = "Tobias")`
- Supports more/less everything you'd expect: or, and, not, $<$, $>$, $=$
- Regular expressions via $=\sim$ syntax.

RETURN clause.

This is just a list of the things you actually want to return – this is important, because in the match clause, you've defined variables, some of which may be important, others not.

- RETURN mystart, other.something – returns the whole mystart node, and the "something" key from the other node.
- Can return multiple things. Can return all things (nodes, relationship and path) matched in a query with RETURN *
- Return values can be nodes or keys within nodes.
- Can even return relationships via a special syntax in the match section: `a-[r:likes]-b`, RETURN r

The declarative model I.

Both SQL and Cypher are declarative languages. More/less, this means that you "declare" **WHAT** you want instead of **HOW** you want to get it.

- The "how" is left to the backend by translating your query in to an execution plan.
- In the Cypher case, the declarative style is more Prolog than SQL.
  - Specifically, you define variable names (arbitrary) and Cypher finds the "solutions" that makes the constraints hold.

The declarative model II.

Consider again our "friends" query:

```
1 START person=node:node_auto_index(name="Graeme Douglas")
2 MATCH person--friend
3 RETURN friend.name;
```

Line by line, we have:

- 1. Find the node(s) with name "Graeme Douglas" in our index and assign it to `person`. `person` will then be available to the rest of the query.

The declarative model II.

Consider again our "friends" query:

```
1 START person=node:node_auto_index(name="Graeme Douglas")
2 MATCH person--friend
3 RETURN friend.name;
```

Line by line, we have:

- 1. Find the node(s) with name "Graeme Douglas" in our index and assign it to `person`. `person` will then be available to the rest of the query.
- 2. Create the identifier `friend` and traverse the graph structure looking for anything that is connected to person, assign that to `friend`

The declarative model II.

Consider again our "friends" query:

```
1 START person=node:node_auto_index(name="Graeme Douglas")
2 MATCH person--friend
3 RETURN friend.name;
```

Line by line, we have:

- 1. Find the node(s) with name "Graeme Douglas" in our index and assign it to `person`. `person` will then be available to the rest of the query.
- 2. Create the identifier `friend` and traverse the graph structure looking for anything that is connected to person, assign that to `friend`
- 3. Return a set of tuples, each containing the name of people who met the criteria.

Other useful clauses.

- **CREATE** defines data, relationships and properties to create

Other useful clauses.

- **CREATE** defines data, relationships and properties to create
- **DELETE** removes records from the graph (nodes, relationships, etc)

Other useful clauses.

- **CREATE** defines data, relationships and properties to create
- **DELETE** removes records from the graph (nodes, relationships, etc)
- **ORDER BY** determines the order of the result set.

Other useful clauses.

- **CREATE** defines data, relationships and properties to create
- **DELETE** removes records from the graph (nodes, relationships, etc)
- **ORDER BY** determines the order of the result set.
- **LIMIT** limits the size of the result set.

Other useful clauses.

- **CREATE** defines data, relationships and properties to create
- **DELETE** removes records from the graph (nodes, relationships, etc)
- **ORDER BY** determines the order of the result set.
- **LIMIT** limits the size of the result set.
- **FOREACH** applies an updating action to be performed once per element in some list (SET).

Other useful clauses.

- **CREATE** defines data, relationships and properties to create
- **DELETE** removes records from the graph (nodes, relationships, etc)
- **ORDER BY** determines the order of the result set.
- **LIMIT** limits the size of the result set.
- **FOREACH** applies an updating action to be performed once per element in some list (SET).
- **SET** allows values to be set to properties.

Resources

- The Cypher documentation is awesome: `http://docs.neo4j.org/chunked/milestone/cypher-query-lang.html`
- Cypher quick reference: `http://neo4j.org/resources/cypher`
- An argument for graph databases: `http://highscalability.com/neo4j-graph-database-kicks-buttox`
- What is a graph DB? `http://docs.neo4j.org/chunked/milestone/what-is-a-graphdb.html`
- Top 10 ways to get to know Neo4J: `http://blog.neo4j.org/2010/02/top-10-ways-to-get-to-know-neo4j.html`