COLUMNS

Practical Perl Tools

It's a Relationship Thing

DAVID N. BLANK-EDELMAN

David has over 30 years of experience in the systems administration/DevOps/SRE field in large multiplatform environments and is the author of the O'Reilly Otter book (new book on SRE forthcoming!). He is one of the co-founders of the now global set of SREcon conferences. David is honored to serve on the USENIX Board of Directors where he helps to organize and engineer conferences like LISA and SREcon. dnb@usenix.org

Faithful readers of this column will know I have a declared affection for graphs. In the past, we've looked at ways to represent graphs in Perl and ways to draw them. But strangely enough, we've never looked at one of the more interesting uses for them these days: as a data representation for a database, that is, graph databases. This column aims to right that wrong. Rather than take on the entirety of the graph database space in this column, we'll use Neo4j, one of the more popular ones, as a springboard for how they work and how we can interact with them via Perl.

The Things and the Other Things (the Basics)

Graph databases are all about dealing with pieces of data and the relationships between those pieces of data. But wait (you cry out in alarm), "Aren't all databases about this? Cough, cough, relational databases, cough, cough…"

(super oversimplification alert!) Relational databases store information in a number of tables containing rows of data (records). The row is broken up into columns (fields). We perform operations that attempt to find matches between the contents of a column in one table and a column in another table. When we get a match, we treat the corresponding rows as related. The usual example of this is a table of people and a table of addresses. If both tables share a "Person ID" column, we can pick an ID from the people table, match it to the address table, and determine that person's address or addresses. Let's say we want to also store orders this person has placed. Add another table. And stored payment info for this person? Add another table. Membership in a special discount group? Add another table. Cumbersome, but pretty straightforward.

To make this example more interesting, let's say we want to introduce a new concept of a family and keep track of the relationship between the people in that people table. Can do, but it starts to get more and more hairy the more complicated these relationships get. Anyone who has had to work with gnarly JOIN statements knows exactly what I am talking about. And the point where you have to redo your entire data model based on new requirements for data representation? No fun at all.

Graph databases (Neo4j in particular, to keep this concrete) go about this in a different way. In Neo4j, you have nodes. These are the things you are storing. A person, an address, an order, a piece of payment info, a discount group, that sort of thing. Each node has a label (typically used to identify the "kind" of node—examples would be *person* or *address*) and a set of properties stored in the node (examples: first/last name, street name, item number, credit card number, discount percentage).

Now let's add the relations part that will construct the graph. We can connect two nodes via a unidirectional (more on this later) relationship. For example: CHILD, MARRIED, LIVES_AT, ORDERED, JOINED. Relationships are first-class things unto themselves. They have labels like the ones I just used as examples (CHILD, MARRIED, etc.) and properties too (e.g., MARRIED could have a property of wedding_date).

If I wanted to write some of these nodes and relationships down, you could imagine I might write them something like this:

```
(adam) -[:MARRIED]-> (steve)
(steve) -[CHILD]-> (jaime)
(adam) -[CHILD]-> (jaime)
(jaime) -[:LIVES_AT]-> (2560 Ninth Street)
(jaime) -[:ORDERED]-> (order 2560)
(jaime) -[:ORDERED]-> (order 2561)
(jaime) -[:ORDERED]-> (order 2562)
```

Adam (well, the node representing Adam) has a married relationship to Steve (his person node). They have a kid named Jaime. She lives on Ninth Street and placed three different orders.

But You Promised Perl Code

For some reason I always feel compelled to give fair warning when I will be taking an approach that leads with a language that is not Perl, so here's your warning: Neo4j has a built-in graph query language called Cypher. Yes, I know, not a particularly encouraging name for a new thing to learn, but I didn't name it. It is possible to perform Neo4j actions from Perl without knowing any Cypher, but you won't get very far that way. This means we are going to dive into some basic Cypher first before getting to the Perl code. Sorry not sorry?

And being the meany I am, we already did it. The previous section had examples of nodes and relationships using Cypher conventions. Nodes are placed in parentheses. Relationships are specified in square brackets connected to nodes via arrows showing the direction of that relationship.

Quick aside about the directional nature of relationships: in Neo4j, you can only create relationships that go a single way from one node to another. But there is no restriction at query time around direction. You can easily (and with no performance hit) query either for relationships that exist (i.e., "Who is Steve married to?") and for nodes in a relationship that goes in a specific direction (i.e., "Who is Adam's kid?").

Let's see some more Cypher statements so you can get a sense of how data is inserted and queried in a Neo4j graph database. For these examples, we're going to use the example movie database that ships with the community (free) version of Neo4j (you can access it from the web console with ":play movies"). Adam, Steve, and Jaime are a lovely family, but let's play around with a larger data set.

First off, let's start by populating the database. Here are some of the lines from the script that creates a movie node and a number of people nodes:

```
CREATE (TheMatrix:Movie {title:'The Matrix', released:1999,
tagline:'Welcome to the Real World'})
CREATE (Keanu:Person {name:'Keanu Reeves', born:1964})
CREATE (Carrie:Person {name:'Carrie-Anne Moss', born:1967})
CREATE (Laurence:Person {name:'Laurence Fishburne',
born:1961})
CREATE (Hugo:Person {name:'Hugo Weaving', born:1960})
CREATE (LillyW:Person {name:'Lilly Wachowski', born:1967})
CREATE (LanaW:Person {name:'Lana Wachowski', born:1965})
CREATE (JoelS:Person {name:'Joel Silver', born:1952})
```

To break this apart, let's pick on Keanu. We create a Keanu node with the label "Person". That node has two properties (his name and birthdate). Now let's add some relationships:

```
CREATE
  (Keanu)-[:ACTED_IN {roles:['Neo']}]->(TheMatrix),
  (Carrie)-[:ACTED_IN {roles:['Trinity']}]->(TheMatrix),
  (Laurence)-[:ACTED_IN {roles:['Morpheus']}]->(TheMatrix),
  (Hugo)-[:ACTED_IN {roles:['Agent Smith']}]->(TheMatrix),
  (LillyW)-[:DIRECTED]->(TheMatrix),
  (LanaW)-[:DIRECTED]->(TheMatrix),
  (JoelS)-[:PRODUCED]->(TheMatrix)
```

A little more dense, but if you can handle Perl data structures, surely a little punctuation won't throw you. Let's start from the bottom because those statements are simpler. The Wachowski sisters directed *The Matrix*, so we created DIRECTED relationships from them to their movie. Similarly, Joel Silver produced the movie, so there is a PRODUCED relationship put into place. Back to Keanu: Keanu acted in the film, so there is an ACTED_IN relationship specified. The part of that line which may look peculiar is this part:

```
{roles:['Neo']}
```

Earlier I mentioned that relationships are first-class citizens. They also have properties (just like nodes do). This is just attaching a property to that relationship. The property holds a list (actors can play more than one part in a movie), hence the square brackets in the property definition.

Just to make sure this is crystal clear (and because I find it so cool), not only are we specifying a relationship between a person node and a movie node (actor acted in movie), we also get to store information about that relationship (which roles or anything else we want) in that relationship definition itself. Relationships matter.

Once we load it into the database, we can ask the web interface to show us an interactive diagram of the database using the Cypher statement "MATCH (a) RETURN (a)" (match every node and returns them). By default the web interface only shows 300 nodes at a time, so the pretty picture in Figure 1 is only showing 300 nodes.

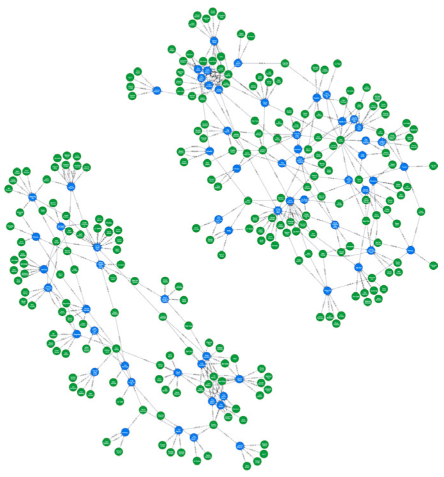

Figure 1: Three hundred of the nodes and relationships in our movie database

Now let's do some querying. We can find Keanu like this:

```
MATCH (k {name: "Keanu Reeves"}) RETURN (k)
```

Here we've said, "Find the node with that property, assign it to k, and return k." But this query doesn't really do the right thing. If there was a movie node (the blockbuster documentary on the life of Keanu), that would also get returned. Better would be to include the label in the query:

```
MATCH (k:Person name: "Keanu Reeves") RETURN (k)
```

If I run this from the command-line shell that ships with Neo4j (cypher-shell), I get the result I would expect:

```
Neo4j> MATCH (k:Person {name: "Keanu Reeves"}) RETURN (k);
+------------------------------------------------------+
| k                                                    |
+------------------------------------------------------+
| (:Person {born: 1964, name: "Keanu Reeves"})         |
+------------------------------------------------------+
```

Now let's find his movies. To do this, we'll have to construct a query that includes a relationship:

```
MATCH (k:Person {name: "Keanu Reeves"})-[:ACTED_IN]->(kmovies)
    RETURN k.name,kmovies.title;
```

Breaking this down: find a Person node with the property "name" set to "Keanu Reeves," then look for all nodes related to that node by an ACTED_IN relationship. Return this, showing only the name property for the k nodes and the title property for the kmovies nodes. The result:

```
+------------------------------------------------+
| k.name         | kmovies.title                 |
+------------------------------------------------+
| "Keanu Reeves" | "Something's Gotta Give"      |
| "Keanu Reeves" | "Johnny Mnemonic"             |
| "Keanu Reeves" | "The Replacements"            |
| "Keanu Reeves" | "The Matrix Revolutions"      |
| "Keanu Reeves" | "The Devil's Advocate"        |
| "Keanu Reeves" | "The Matrix Reloaded"         |
| "Keanu Reeves" | "The Matrix"                  |
+------------------------------------------------+
```

One last query, so let's make it a fun one. What if we want to find all of the actors Keanu acted with in the database? Let's break the question down first, then see it in Cypher form:

1. What movies has he been in? Just did that, check.
2. What actors have acted in those same movies? That can be stated as "Given a movie, what actors have an ACTED_IN relationship to that movie?"

Here's the Cypher version where our query asks both questions at once:

```
MATCH (k:Person {name:"Keanu Reeves"})-[:ACTED_IN]->
       (movie)
          <-[:ACTED_IN]-(actor) RETURN actor.name
```

I've broken it up on several lines to make it easier to read, but you would likely use it as a single line. We find the movie nodes representing the movies Keanu has acted in and then look for other actors who also have acted in each movie (have an ACTED_IN relationship to it). The results:

```
+-------------------------+
| actor.name              |
+-------------------------+
| "Diane Keaton"          |
| "Jack Nicholson"        |
| "Takeshi Kitano"        |
| "Dina Meyer"            |
| "Ice-T"                 |
| "Brooke Langton"        |
| "Gene Hackman"          |
| "Orlando Jones"         |
| "Laurence Fishburne"    |
| "Hugo Weaving"          |
| "Carrie-Anne Moss"      |
| "Charlize Theron"       |
| "Al Pacino"             |
| "Laurence Fishburne"    |
| "Carrie-Anne Moss"      |
| "Hugo Weaving"          |
| "Emil Eifrem"           |
| "Laurence Fishburne"  |
| "Carrie-Anne Moss"    |
| "Hugo Weaving"        |
+-----------------------------+
```

Chances are we only want to see each actor's name once, but I wanted to drive home the notion that we're walking around a graph to return results. To produce a list that has each actor listed only once, we would write:

```
RETURN DISTINCT actor.name;
```

There's a ton more things we can do using Cypher (for example, it becomes easy to do a "six degrees of Kevin Bacon" query), but I'd recommend you look at the Neo4j doc and demo packages for that information. Let's actually see some Perl.

DBI-ish...

We've explored DBI ("the standard database interface model for Perl") in the past, so hopefully this section elicits feelings of "Oh good, it is that easy." Before we actually look at REST::Neo4p, the Perl module we are going to use, I should mention that there does exist a DBD::Neo4p. This means you could use *exactly* the DBI syntax if you really wanted to. DBI is definitely the way to go when you are dealing with relational databases and you want to make sure that your code has some level of portability between database engines. I suppose it is plausible that you might be switching from a standard relational database, and so you will be treating the graph database like any other database engine initially, but this feels like a bit of a stretch for me. There might be another scenario I'm not thinking of, but, in the meantime, let's dive into REST::Neo4p. Even if it isn't DBI per se, it is definitely modeled on it.

Just as you would do with any DBI code, the first step is to connect to the database:

```
use REST::Neo4p;
REST::Neo4p->connect( 'http://127.0.0.1:7474',
                      'neo4j', 'password' );
```

From here we can go two ways with the module:

1. We could use method calls to operate on the database.
2. We could tee up and then execute Cypher commands in a very similar fashion to the way we might use SQL in a standard DBI program.

Let's see both approaches.

The first method would look like this:

```
my $movie = REST::Neo4p::Node->new(
    {
       title     => 'The Room',
       released  => '2003',
       tagline   => 'Experience this quirky new black comedy,
it's a riot!'
    }
)->set_labels('Movie');
my $person = REST::Neo4p::Node->new(
    {
       name    => 'Tommy Wiseau',
       born    => '1955',
    }
)->set_labels('Person');
$person->relate_to( $movie, 'DIRECTED');
```

We create nodes and their properties and relationships (with no properties—we would just need to add in a hash reference with the info, similar to the way it is done for node creation).

For the second approach, using Cypher from Perl, it should be relatively straightforward to people used to working with SQL from Perl:

```
my $cypher = REST::Neo4p::Query->new(
    'MATCH (theroom {title: "The Room"})<-[:DIRECTED]-(director)
        RETURN director');
$cypher->execute;
while (my $result = $cypher->fetch) {
        print $result->[0]->get_property('name'), "\n";
}
```

In this code we prepare and execute a Cypher query that

- starts by looking for the movie node with a property matching the title we are looking for,
- then looks for nodes that have a relationship of director to that movie and returns the nodes it finds.

The rest of the Perl code just iterates over the returned nodes and prints out the key property from them (the name of the director).

And that's the basics of working with Neo4j from Perl. Take care, and I'll see you next time.