Extracted from:

Seven Databases in Seven Weeks, Second Edition

A Guide to Modern Databases and the NoSQL Movement

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Seven Databases in Seven Weeks

Second Edition

A Guide to Modern Databases and the NoSQL Movement

Luc Perkins
with Eric Redmond and Jim R. Wilson

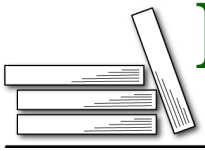
Series editor: *Bruce A. Tate*
Development editor: *Jacquelyn Carter*

Seven Databases in Seven Weeks, Second Edition

A Guide to Modern Databases and the NoSQL Movement

Luc Perkins
with Eric Redmond
and Jim R. Wilson

The Pragmatic Bookshelf
Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Series Editor: Bruce A. Tate
Copy Editor: Nancy Rapoport
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-253-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—April 2018

Big Data

Up until now, we've dealt with very small datasets, so now it's time to see what Neo4j can do with some big data. We'll explore a dataset covering information about over 12,000 movies and over 40,000 actors, 6,000 directors, and others involved in those movies.

This dataset has been made available[4] by the good folks at Neo4j, who have conveniently made the data directly digestible by Neo4j; thus, we don't need to convert it from CSV, XML, or some other format.

First, let's download the dataset as a Zip file, unzip it, and add it to the /data folder in our Neo4j directory:

```
$ cd /path/to/neo4j
$ curl -O <copy URL from footnote 4>
$ unzip cineasts_12k_movies_50k_actors_2.1.6.zip
$ mv cineasts_12k_movies_50k_actors.db data/movies.db
```

That dataset was generated to work with version 2.1.6 of Neo4j, but we're using a much later version (3.0.7). We'll need to make one small configuration change to enable it to work with our version. In the /conf folder there's a file called neo4j.conf, and inside that file there's a line that looks like this:

```
#dbms.allow_format_migrations=true
```

Delete the # at the beginning of the line. That will instruct Neo4j to automatically migrate the format to fit our version. Now, fire up the Neo4j shell, specifying our movies.db database and the config file we just modified:

```
$ bin/neo4j-shell -path data/movies.db -config conf/neo4j.conf
```

This is our first encounter with the Neo4j shell. It's somewhat like the web console we used earlier in this chapter, but with the crucial difference that it returns raw values rather than pretty charts. It is a more direct and no-frills way to interact with Neo4j and better to use once you have gotten the hang of the database. When the shell fires up, you should see a shell prompt like this:

```
neo4j-sh (?)$
```

You can enter help to get a list of available shell commands. At any time in the shell, you can either enter one of those commands *or* a Cypher query.

Our shell session is already pointing to our movie database, so let's see what nodes are there:

4. http://example-data.neo4j.org/files/cineasts_12k_movies_50k_actors_2.1.6.zip

```
neo4j> MATCH (n) RETURN n;
```

Whoa! That's a lot of nodes, 63,042 to be exact (you can obtain that result by returning `count(n)` instead of just `n`). We warned you that this is a Big Data section! Let's make some more specific queries now. First, let's see what types of relationships exist:

```
neo4j> MATCH ()-[r]-() RETURN DISTINCT type(r);
+------------+
| type(r)    |
+------------+
| "ACTS_IN"  |
| "DIRECTED" |
| "RATED"    |
| "FRIEND"   |
+------------+
4 rows
```

Here, the `()-[r]-()` expresses that we don't care what the nodes look like; we just want to know about the relationship between them, which we're storing in the variable `r`. You can also see that in Cypher you use `type(r)` instead of, say, `r.type` to get the relationship type (because types are a special attribute of relationships). As you can see, there are four types of relationships present in the database. Now, let's look at all the nodes and see both which labels are applied to them and how many nodes are characterized by that label:

```
$ MATCH (n) RETURN DISTINCT labels(n), count(n);
+-----------------------------------------+
| labels(n)                     | count(*) |
+-----------------------------------------+
| ["Person","Actor","Director"] | 846      |
| ["Person","User"]             | 45       |
| ["Person","Actor"]            | 44097    |
| ["Person","Director"]         | 5191     |
| []                            | 1        |
| ["Movie"]                     | 12862    |
+-----------------------------------------+
6 rows
```

As you can see, all nodes that aren't movies have the `Person` label (or no label); of those, all `Persons` are either `Actors`, `Directors`, `Users` (the folks who put the dataset together), or a `Director` and `Actor` (we're looking at you, Clint Eastwood). Let's perform some other queries to explore the database.

Let's see everyone who's both an actor and a director, and then get the count of people who share that distinction:

```
> MATCH (p:Actor:Director) RETURN p.name;
> MATCH (p:Actor:Director) RETURN count(p.name);
```

Now let's see who directed the immortal *Top Gun*:

```
> MATCH (d:Director)-[:DIRECTED]-(m:Movie {title: "Top Gun"}) RETURN d.name;
```

Let's see how many movies the legendary Meryl Streep has acted in:

```
> MATCH (a:Actor {name: "Meryl Streep"})-[:ACTS_IN]-(m:Movie)
  RETURN count(m);
```

Finally, let's get a list of actors who have appeared in over fifty movies:

```
> MATCH (a: Actor)-[:ACTS_IN]->(m:Movie)
  WITH a, count(m) AS movie_count
  WHERE movie_count > 50
  RETURN a.name; # only 6 actors!
```

Now that we've played with this specific dataset a little bit, let's solve a more challenging algorithmic problem that uses Neo4j more like the high-powered graph database that it really is. What's the most common algorithmic problem in showbiz? You may have guessed already...

Six Degrees of...

...you guessed it: Kevin Bacon. We're going to solve the six degrees of Kevin Bacon problem here so that you can memorize some of the key results and be a big hit at your next dinner party. More specifically, we want to know how many actors are within six degrees of Mr. Bacon, what percentage of the actors in the database have that distinction, what the shortest "path" from an actor to Kevin Bacon happens to be, and so on. You'll find some similar Neo4j exercises online but this one utilizes a very large dataset that can generate more true-to-life results.

What you may find in this exercise is that Cypher has *a lot* already baked into the language. To get the results we're after, we won't need to write a sophisticated algorithm on the client side or traverse a node tree or anything like that. We just need to learn a little bit more about how Cypher works.

In the last section, you saw that you can make very specific queries about nodes and relationships. Let's find out which `Movies` nodes Kevin Bacon has the relationship `ACTED_IN` with (let's see the number of movies first and then list the titles):

```
> MATCH (Actor {name:"Kevin Bacon"})-[:ACTS_IN]-(m:Movie) RETURN count(m);
> MATCH (Actor {name:"Kevin Bacon"})-[:ACTS_IN]-(m:Movie) RETURN m.title;
```

Only thirty movies in our database! Amazing that such a not-exceedingly prolific actor is so well connected in Hollywood. But remember, the magic of Kevin Bacon is not the *number* of movies he's been in; it's the *variety* of actors

he's shared the screen with. Let's find out how many actors share this distinction (this query will make more sense later):

```
> MATCH (Actor {name: "Kevin Bacon"})-[:ACTS_IN]->(Movie)
    <-[:ACTS_IN]-(other:Actor)
  RETURN count(DISTINCT other);
+----------+
| count(a) |
+----------+
| 304      |
+----------+
```

Still not a *huge* number, but remember this is only one degree of Kevin Bacon. Here, we can see that you can actually *reverse* the direction of the relationship arrow in a query, which is quite useful in more complex queries like this.

Now let's find out how many actors are *two* degrees from Kevin Bacon. From the previous expression, it's not entirely clear *how* to write that query because we'd need to make that relationship chain much more complex, resulting in a mess of arrows and brackets. Fortunately, Cypher provides us with some syntactical sugar for this using star notation (*).

```
> MATCH (Actor {name: "Kevin Bacon"})-[:ACTS_IN*1..2]-(other:Actor)
  RETURN count(DISTINCT other);
+----------+
| count(a) |
+----------+
| 304      |
+----------+
```

Two things to be aware of. First, there's no `Movie` label anywhere here. That's because `Actor`s can only have `ACTS_IN` relationships with `Movie`s, so we can safely leave that part out. Second, note that that's the same result as before (a count of 313), so something is not quite right. It turns out that this Cypher star notation is a bit tricky because each "jump" between nodes counts. So you'll need to think of this query in terms of a four-jump chain (actor-movie-actor-movie-actor) and rewrite the query:

```
> MATCH (Actor {name: "Kevin Bacon"})-[:ACTS_IN*1..4]-(other:Actor)
  RETURN count(DISTINCT other);
+-----------------------+
| count(DISTINCT other) |
+-----------------------+
| 9096                  |
+-----------------------+
```

Be Wary of Repetition

You may have noticed the DISTINCT expression in many of these Cypher queries. This is *extremely* important in Cypher queries because it enables you to exclude redundant results. Running the previous query without using DISTINCT results in a count of 313, which suggests that there are a few actors who are within two degrees of Kevin Bacon more than once. Quite the distinction (no pun intended)!

For some datasets, these discrepancies may be much larger, skewing the results beyond recognition and usefulness. When dealing with graphs, this is something that can really bite you, so if your results ever seem off, checking for redundancy is a good place to start.

If you use 5 instead of 4 in that query, you'll get the same result and for the same reason. You need to make an actor-to-movie jump to include more actors in the result set.

As you can see, the quotient between 2 degrees and 1 degree is about 79, so the web of relationships fans out very quickly. Calculating 3 degrees yields 31,323. Counting 4 degrees takes *minutes* and might even time out on your machine, so we don't recommend running that query unless you need a (long) hot chocolate break.

Thus far, we've only really been counting nodes that share some trait, though our ability to describe those traits has been enhanced. We're still not equipped to answer any of our initial questions, such as how many degrees lie between Kevin Bacon and other actors, what percentage of actors lie within N degrees, and so on.

To get traction into those questions, we need to begin querying for path data, as we did in the REST exercises. Once again, Cypher comes through in the clutch for us with its shortestPath function, which enables you to easily calculate the distance between two nodes. You can specify the relationship type you're interested in specifically or just use * if the relationship type doesn't matter.

We can use the shortestPath function to find the number of degrees separating Kevin Bacon and another dashing actor, Sean Penn, using this query:

```
> MATCH (
    bacon:Actor {name: "Kevin Bacon"}),
    (penn:Actor {name: "Sean Penn"}
  ),
  p=shortestPath((bacon)-[:ACTS_IN*]-(penn))
  RETURN length(p);
+---------------+
| length(p) / 2 |
```

```
+---------------+
| 2             |
+---------------+
```

But wait a second. According to IMDB, Messieurs Bacon and Penn starred together in *Mystic River*. So why does it take 2 degrees to connect these two when it should be one? Well, it turns out that *Mystic River* isn't in our database.

```
> MATCH (m:Movie {name: "Mystic River"})
  RETURN count(DISTINCT m);
+-------------------+
| count(DISTINCT m) |
+-------------------+
| 0                 |
+-------------------+
```

Looks like our database is lacking some crucial bits of cinema. So maybe don't use these results to show off at your next dinner party just yet; you might want to find a more complete database for that. But for now, you know how to calculate shortest paths, so that's a good start. Try finding the number of degrees between some of your favorite actors.

Another thing we're interested in beyond the shortest path between any two actors is the percentage of actors that lie within N degrees. We can do that by using a generic `other` node with the `Actor` label and counting the *number* of shortest paths that are found within a number of degrees. We'll start with 2 degrees and divide the result by the total number of actors, making sure to specify that we don't include the original Kevin Bacon node in the shortest path calculation (or we'll get a nasty and long-winded error message).

```
> MATCH p=shortestPath(
    (bacon:Actor {name: "Kevin Bacon"})-[:ACTS_IN*1..2]-(other:Actor)
  )
  WHERE bacon <> other
  RETURN count(p);
+----------+
| count(p) |
+----------+
| 304      |
+----------+
```

Just as expected, the same result as before. What happened there is that Neo4j traversed every relationship within 1 degree of Kevin Bacon and found the number that had shortest paths. So in this case, `p` returns a list of many shortest paths between Kevin Bacon and many actors rather than just a single shortest path to one actor. Now let's divide by the total number of actors (44,943) and add an extra decimal place to make sure we get a float:

```
> MATCH p=shortestPath(
    (bacon:Actor {name: "Kevin Bacon"})-[:ACTS_IN*1..2]-(other:Actor)
  )
  WHERE bacon <> other
  RETURN count(p) / 44943.0;
+----------------------+
| count(p) / 44943.0   |
+----------------------+
| 0.006764123445252876 |
+----------------------+
```

That's a pretty small percentage of actors. But now re-run that query using 4 instead of 2 (to symbolize 2 degrees rather than 1):

```
> MATCH p=shortestPath(
    (bacon:Actor {name: "Kevin Bacon"})-[:ACTS_IN*1..4]-(other:Actor)
  )
  WHERE bacon <> other
  RETURN count(p) / 44943.0;
+--------------------+
| count(p) / 44943.0 |
+--------------------+
| 0.2023674432058385 |
+--------------------+
```

Already up to 20 percent within just 1 extra degree. Running the same query gets you almost 70 percent for 3 degrees, a little over 90 percent for 4 degrees, 93 percent for 5, and about 93.4 percent for a full 6 degrees. So how many actors have *no* relationship with Kevin Bacon whatsoever in our database? We can find that out by not specifying an N for degrees and just using any degree:

```
> MATCH p=shortestPath(
    (bacon:Actor {name: "Kevin Bacon"})-[:ACTS_IN*]-(other:Actor)
  )
  WHERE bacon <> other
  RETURN count(p) / 44943.0;
+--------------------+
| count(p) / 44943.0 |
+--------------------+
| 0.9354960728033287 |
+--------------------+
```

Just a little bit higher than the percentage of actors within 6 degrees, so if you're related to Kevin Bacon *at all* in our database, then you're almost certainly within 6 degrees.

Day 2 Wrap-Up

On Day 2, we broadened our ability to interact with Neo4j by taking a look at the REST interface. You saw how, using the Cypher plugin, you can execute

Cypher code on the server and have the REST interface return results. We played around with a larger dataset and finally finished up with a handful of algorithms for diving into that data.