Extracted from:

Exploring Graphs with Elixir

Connect Data with Native Graph Libraries and Graph Databases

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Your Elixir Source

Exploring Graphs with Elixir

Connect Data with Native Graph Libraries and Graph Databases

Tony Hammond

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

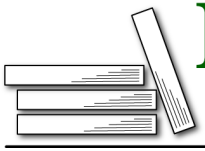
Exploring Graphs with Elixir

Connect Data with Native Graph Libraries and Graph Databases

Tony Hammond

The Pragmatic Bookshelf

Raleigh, North Carolina



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Series Editor: Bruce A. Tate
Development Editor: Jacquelyn Carter
Copy Editor: Corina Lebegioara
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

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

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

ISBN-13: 978-1-68050-840-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—November 2022

CHAPTER 5

Navigating Graphs with Neo4j

We are now going to turn our attention to property graphs. The property graph is perhaps the best-known data model for semantic graphs (graphs with an explicit information superstructure).

Property graphs—also known as labeled property graphs—are graphs in which both nodes and edges may be attributed properties and in which nodes may be labeled for grouping.

To study property graphs in a more controlled way, we'll benefit greatly by using a database to store our graphs so that we can requery them without having to rebuild them. And a true graph database—a database that deals with graphs as first-class data structures—would be even better. Unquestionably, one of the most popular graph databases is Neo4j,[1] which was one of the initial movers in this field. Neo4j has been a major player in driving forward the current interest in graph databases.

We should call out here a couple of key Neo4j technologies that we'll be using for connecting to the database and for querying over the graphs it manages:

- *Bolt*[2] is a high-performance network protocol that was introduced with the Neo4j 3.0 release in 2016 to speed up database connections. It uses binary encoding over TCP or web sockets and has built-in TLS support.
- *Cypher*[3] is the declarative graph query language developed by Neo4j and is now open-sourced to the `openCypher`[4] project. (See the Cypher Refcard[5] for a handy quick reference.)

1. https://neo4j.com/
2. https://boltprotocol.org/
3. https://neo4j.com/docs/cypher-manual/current/
4. http://www.opencypher.org/
5. https://neo4j.com/docs/cypher-refcard/current/

We're going to use the bolt_sips[6] package from Florin Pătraşcu[7] that implements a Neo4j driver for Elixir wrapped around the Bolt protocol. (The package integrates and continues work from boltex,[8] an independent implementation of the Bolt protocol in Elixir by Michael Schaefermeyer.[9])

But before we get to that, let's first review the property graph model. We'll then create a new PropertyGraph project and look at querying with Cypher, and we'll also try out the Bolt driver. And then we'll implement a graph service for the project using our common graph services API. Lastly, we'll see how to switch graph service contexts easily.

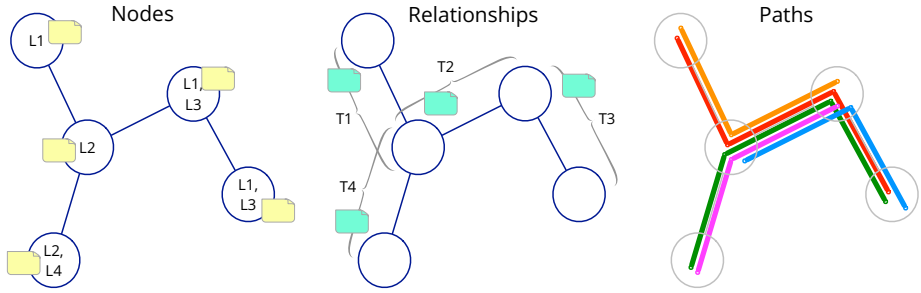
Property Graph Model

The distinguishing feature of a property graph is that graph vertices and edges may be decorated with attributes (or properties). In Neo4j parlance, we talk about *nodes* (for vertices) and *relationships* (for edges).

We'll discuss property graphs here from a Neo4j perspective.

The graph elements we'll talk about are nodes, relationships, and paths, along with their associated IDs, properties, labels, and types. See Graph Database Concepts[10] in the Neo4j documentation for more details.

The following diagram here shows some of these graph constructs:



Nodes are shown with optional labels L1, L2, and so on, while the required single-value relationship types are shown as T1, T2, and so on. Property maps

6. https://hex.pm/packages/bolt_sips
7. https://hex.pm/users/florin
8. https://hex.pm/packages/boltex
9. https://about.me/mschae
10. https://neo4j.com/docs/getting-started/current/graphdb-concepts/

are shown with yellow document icons for nodes and green document icons for relationships. Some various paths are shown between node pairs.

Nodes

Nodes in Neo4j are graph vertices and are allocated a system ID. They may have zero or more user-defined labels associated with them. Labels are used for grouping nodes into sets.

Nodes may additionally have a map of property names and property values associated with them.

Relationships

Relationships in Neo4j are graph edges that relate two nodes and are allocated a system ID. They take a single user-defined relationship type.

Relationships may additionally have a map of property names and property values associated with them.

Note that relationships in Neo4j always have a direction that is defined at create time but may be omitted at query time.

Paths

Paths in Neo4j are sequences of relationships that join sequences of nodes and are used to answer traversal questions. The sequence of relationships in the path is always distinct, whereas the sequence of nodes may or may not be distinct.

One common traversal question is: “What is the shortest path between two given nodes?” This also brings up the notion of path length. Cypher includes many handy functions to answer such questions.

Creating the PropertyGraph Project

OK, enough of the theory. Let’s try querying some property graphs for real. We’re going to need a couple of things: a database and a database driver.

PropertyGraph Project/Database Setup

See Appendix 1, Project Setups, on page ?, for details on retrieving a working project with code and data.

And see Appendix 2, Database Setups, on page ?, for help on setting up a local copy of Neo4j.

For the database driver, we'll use the bolt_sips package. We'll want to create a new project under our umbrella app. Let's call this project PropertyGraph. (See the bolt_sips project for detailed installation instructions.[11])

Follow the usual drill for creating the new project, PropertyGraph. Go to the ExGraphsBook home project (see ExGraphsBook Umbrella, on page ?), cd down into the apps directory, and open up the new PropertyGraph project:

```
$ mix new property_graph --sup
```

This will generate an app with a supervision tree and an application callback. We'll use the PropertyGraph.Application module to set up the supervision tree.

You should now have an apps directory that looks like this:

```
.
├── apps
│   ├── graph_commons
│   ├── native_graph
➤  │   └── property_graph
```

Now cd into the property_graph directory:

```
.
├── README.md
├── lib
➤  │   ├── property_graph
│   │   └── application.ex
│   └── property_graph.ex
├── mix.exs
└── test
    └── ...
```

Note that the --sup flag has generated an extra directory property_graph under lib with an application.ex file.

We can declare a dependency on bolt_sips by adding the :bolt_sips dependency to the mix.exs file:

apps/property_graph/mix.exs

```
defp deps do
  [
    # graph_commons
    {:graph_commons, in_umbrella: true},

    # property graphs
    {:bolt_sips, "~> 2.0"}
  ]
end
```

11. https://github.com/florinpatrascu/bolt_sips

As usual, use Mix to add in the dependency:

```
$ mix deps.get; mix deps.compile
```

We'll need to specify our connection details:

```
config :bolt_sips, Bolt,
  url: "bolt://localhost:7687",
  basic_auth: [username: "neo4j", password: "neo4jtest"]
```

Add these lines (with details updated as required) to the umbrella config.exs file in the main project directory or to an environment-specific import (for example, dev.exs). Note that the url: option uses an explicit bolt: URI scheme.

We'll also need to start up our PropertyGraph.Application module:

apps/property_graph/mix.exs

```
def application do
  [
    extra_applications: [:logger],
➤   mod: {PropertyGraph.Application, []}
  ]
end
```

The :mod option specifies the application callback module, followed by any arguments to be passed on application start. The application callback module is any module that implements the Application behaviour.

We update the start/2 function in lib/property_graph/application.ex as:

apps/property_graph/lib/property_graph/application.ex

```
defmodule PropertyGraph.Application do
  use Application

  def start(_type, _args) do
    children = [
➤     {Bolt.Sips, Application.get_env(:bolt_sips, Bolt)}
    ]

➤   opts = [strategy: :one_for_one, name: PropertyGraph.Service]
    Supervisor.start_link(children, opts)
  end

end
```

The application will now be started automatically and can be tested by calling the info/0 function in Bolt.Sips:

```
iex> Bolt.Sips.info()
%{
  default: %{
    connections: %{direct: %{"localhost:7687" => 0}, routing_query: nil},
    user_options: [
```

```
      socket: Bolt.Sips.Socket,
      basic_auth: [username: "neo4j", password: "neo4jtest"],
      port: 7687,
      routing_context: %{},
      schema: "bolt",
      hostname: "localhost",
      pool_size: 15,
      max_overflow: 0,
      timeout: 15000,
      ssl: false,
      with_etls: false,
      retry_linear_backoff: [delay: 150, factor: 2, tries: 3],
      prefix: :default,
      url: "bolt://neo4j:neo4jtest@localhost:7687"
    ]
  }
}
```

Let's get a database connection:

```
iex> Bolt.Sips.conn()
#PID<0.352.0>
```

In direct mode, which is our current configuration, all the operations—read/write and delete—are sent to the database using a common connection (from a connection pool). The `conn/0` function returns the process ID for this pool connection.

Finally, let's wire our graph storage into the `PropertyGraph` module with these `use/2` macros:

apps/property_graph/lib/property_graph.ex

```
use GraphCommons.Graph, graph_type: :property, graph_module: __MODULE__
use GraphCommons.Query, query_type: :property, query_module: __MODULE__
```

Well, that about covers the setup. But before we get into any real querying, which we'll cover in Chapter 6, Querying Neo4j with Cypher, on page ?, we'll spend the rest of this chapter looking at how to create queries and send them to the database.