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*

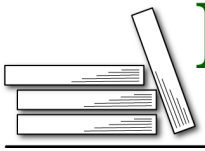
Exploring Graphs with Elixir

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 1

Engaging with Graphs

Graphs and graph databases are everywhere. This is no big surprise given that graphs can address the two main concerns we have in dealing with the huge volumes of data all around us—organization and scale. Differently from the more usual "buckets of data" or relational data approaches, graphs can bring both order and growth to data as data goes large. And they can do this both organically and holistically. This is what makes graphs such a fascinating field to work with.

As an organizational pattern, graphs operate at all levels from the smallest static structures, such as chemical compounds—think of a water molecule—to large-scale dynamic structures, such as web-based social networks—think Facebook or Twitter.

Graphs are especially useful in dealing with messy and irregular datasets and hard-to-fit data. They cope particularly well with sparse datasets. Unlike the relational model, with fixed tables optimized for transactional database requests, graphs tend to turn things on their head. Instead of dealing with objects as sets of relations and then attempting joins over these sets, it is the relationships between objects that become the chief organizing principle. It's all about the connections rather than the records. Schemas take a backseat—still incredibly useful but not overly restrictive. We have a much more fluid way to relate our data items.

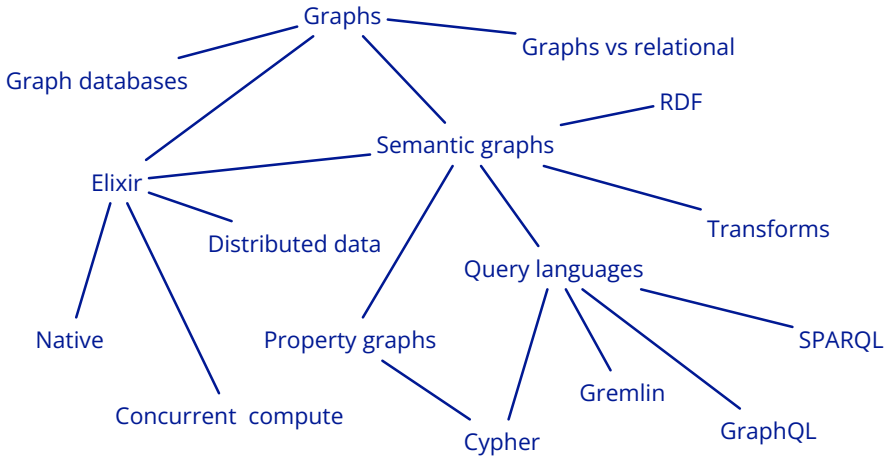
With graphs, we are typically working with an open-world assumption and thus with partial knowledge. We can't conclude anything definite from missing data. Any missing data may arrive at any future time. This is in contrast to more familiar data models which commonly use a closed-world assumption where everything is known ahead of time and locked down. Those data models are predictable and provide solid guarantees about data integrity. The down-side is that they are regimented.

More simply said, graphs are great at gluing pieces of data together.

Although graph data structures can connect data items, in practice graphs themselves tend to be disconnected from each other both physically in separate graph databases and conceptually in terms of data models. To move data between graphs, it helps to understand their respective data models and how we can transform data from one graph model to another.

In later chapters, we'll work with the different graph models and also look at graph transformations. The graph-to-graph problem is almost as challenging as the structured-data-to-graph (table or document to graph) problem.

So what's this book about then? Maybe this concept map (and yes, it's a graph) can assist in indicating some of the things we'll explore.



We'll deal with graphs as structures for organizing data at large. We'll see how we can use Elixir to process both database graphs and distributed graphs. We'll focus on the so-called "semantic" graphs—that is, graphs with an information-bearing capacity. We'll need to consider different graph models, what they provide, and how they can be related, and we'll need to work with different query languages. We'll cover all of these in this book.

First, let's look into what graphs actually are and also some common paradigms for graph models. In later chapters, we'll work with different graph packages in Elixir, but until then we can try our hand at building a graph with a library that ships with Elixir. To compare the different graph packages with their respective graph models, we'll need a reference graph model. We'll define one here so we can see how these graph models variously deal with this reference model.

First Contact

Before going further, let's establish some of the terms we'll be using in this book. First of all, when we talk about graphs, we obviously mean networks—not charts. Traditionally the term "graph" has referred to a diagram or plot of one quantity against another. This is the more widely understood sense of the term. But that's not our concern here. We'll use the term "graph" in its other sense of a data structure used to model relationships between things. That is, we have one set of things and another set of relationships between those things. Those two sets together constitute a graph. In its most general sense, a graph is a data model for relating a collection of things.

Network vs. Graph

We mentioned the word "network" before. This is the term used by network science as opposed to "graph," which is the preferred term used in graph theory, a branch of mathematics. A network is an engineering implementation of a graph. There are other differences too. A network is typically a dynamic system concerned with flows through a structure. A graph, on the other hand, is typically understood holistically as a static construct.

Vertex/Edge—What?

As noted, there are two components in a graph: things and relationships. In practice, you'll find many different terms for these graph building blocks:

vertex/edge
: terms used by graph theory, a formal theory in math

node/link
: terms used in network science theory

node/relationship
: terms used by the Neo4j graph database

node/arc
: terms used by the RDF graph data model

dot/line
: terms sometimes used in graph diagrams

object/arrow
: terms used by category theory, a foundational theory in math (a category is a graph with additional structure)

But it doesn't matter which terms we use. It's probably best to keep the *vertex/edge* pairing when dealing with graph theory topics and use the *node/link* pairing when talking about networks. In this book, however, we're going to use the terms *node* and *edge*, although we'll sometimes also use the term *vertex* for *node*.

Graph Models

Of course, there is more to all this than just nodes and edges. There are code libraries for modeling graphs and running graph algorithms. There are graph databases that implement particular graph models. It's important to establish now that there are two main graph models which are supported by graph databases: the property graph model, sometimes referred to as the labeled property graph, and the RDF model. The following table shows a direct feature comparison between these two graph models:

	Property Graph	RDF
sponsor	industry	W3C
standards	no	yes
field of origin	database	documents (web)
published	2007?	1999
strength	graph exploration	data integration
query language	Cypher, Gremlin	SPARQL
names	system	global (IRI)
annotations		
nodes	attributes	edges (with string nodes)
edges	attributes	—

This is obviously an oversimplification of the current position. For example, although property graphs are not standards-based, there is work ongoing to surface aspects of the model within various standards bodies. At the same time, there is the new development of RDF* (with SPARQL*), which seeks to close the gap between property graphs and RDF graphs by addressing the edge annotation problem.

But this is all getting ahead of ourselves. We'll look more closely at the different graph models as we explore the actual graph packages. And we haven't even seen a graph yet. So let's remedy that now.

Coding a Hello World Graph

As Elixir programmers, it's only natural to wonder what kind of support Elixir has for working with graph technologies. Quite a bit as it turns out. A growing number of Elixir graph packages have been in active development for some time now, and they are addressing all of the major graph types.

In this book, we're going to develop a project that will allow us to explore some of the main graph databases, graph query languages, and graph models. We'll also look at interchanging between graph types and transforming data from one to another.

But first, let's try something out of the box—no setup required.

Elixir comes with built-in graph support. We can use the Erlang library :digraph that ships with the Elixir distribution. Let's use this to string a couple of words together. Just fire up Elixir's interactive shell IEx from the command line:

```
$ iex
Erlang/OTP 24 [erts-12.3.1] ...

Interactive Elixir (1.13.4) - press Ctrl+C to exit (type h() ENTER for help)
iex>
```

Now that we're in IEx, import the :digraph library:

```
iex> import :digraph
:digraph

iex> g = new
{:digraph, #Reference<0.148244651.2766536707.134565>, ..., true}

iex> v1 = add_vertex(g, "Hello")
"Hello"

iex> v2 = add_vertex(g, "World")
"World"

iex> get_path(g, v1, v2)
false
```

So, here we created a couple of vertices and tried to get the path between them and, of course, we failed because we haven't yet defined any edges in our graph. Let's fix that and try again:

```
iex> add_edge(g, v1, v2)
[:"$e" | 0]

iex> get_path(g, v1, v2)
["Hello", "World"]
```

Ah, that's better. Our very first graph—a "Hello World" graph.