IN THIS CHAPTER

- Introducing graph database basics
- Following the graph learning curve

Chapter 1
Introducing Graph Databases

Since the turn of the century, an explosion of new database technologies has ended the prior dominance of relational systems. These various new kinds of databases distinguished themselves with the umbrella term NoSQL. While the terminology is debatable, NoSQL technology really is different from the relational world. Instead of storing data in rows in tables, databases store nested documents, key-value pairs, or columnar form data.

There are good reasons for the emergence of new data models. Document databases optimize for ease of storage and retrieval with a file cabinet metaphor of document-in, document-out. Column store databases optimize for scale and the ability to scan many records rapidly. In optimizing for their use cases though, the new databases opted for simplistic data models. For example, understanding how two records are related is part of the relational model via joins, but no equivalent mechanism exists in document, key-value, or column store databases.

In this chapter, you discover the fundamental building blocks of graphs and how to use them to create sophisticated, high-fidelity data models.

Exploring Graph Database Basics

A *graph database* uses highly inter-linked data structures built from nodes, relationships, and properties. In turn, these graph structures support sophisticated, semantically rich queries at scale.

Graph databases turn NoSQL thinking on its head: Relationships between data are just as important as the data itself. A graph database builds a network of interconnected entities to represent its domain. Like relational databases, you can query that model to gain insight, but unlike relational databases, the data model is intuitive. Using graph data models doesn't require a semester of classes on normalization and years of system administration experience on how to denormalize relational data for performance. Instead, with a handful of simple tools, you can build expressive and understandable data models that are highly performant. In this section, you explore the new data model basics.

Understanding who uses graph databases and why

Graph databases are general-purpose data technology. They can be used by a wide variety of domains from healthcare to finance, and energy to disaster response. The key to understanding when to use a graph database is the value of links. If your data is connected, whether it supports an online mobile app or an offline machine learning framework, then a graph is going to be a good choice.

Conversely if your data is bulk storage, blob storage, time-series, or logs, then a graph may not be the best choice because there aren't many links between the data to exploit. Graphs *are* general-purpose, but they are *not* the only useful data model. Graphs are broadly useful, and we give you a range of examples throughout this book.

Seeing the benefits of graph databases

Graphs bring several benefits across the whole life cycle of a system. For the production lifetime of a system, graphs offer superior querying of complex models, enabling business to ask pertinent questions with high performance. This alone is enough to put

graphs on your to-do list. But graphs also offer ease of development, where combining simple patterns allows you to build large sophisticated networks that represent your problem domain in high-fidelity.

Explaining Labeled Property Graphs

The most widely used model for graph databases is the *labeled property graph model.* To experts, this shorthand is useful to distinguish between this model and other more mathematically inclined models, such as hypergraphs. But if you aren't an expert, this description may need a little unpacking.

The fundamental components of the labeled property graph model are nodes and relationships (you may also know these as vertices and edges) and constraints.

REMEMBER

In the labeled property graph model, we use naming conventions to distinguish elements at a glance. When reading this chapter or others, the following helps describe the naming conventions:

- **Node labels are *PascalCase.*** Every word starts with an uppercase letter with no spaces.
- **Relationships are *SNAKE_CASE_ALL_CAPS.*** Replace all the spaces with an underlined character and convert all the letters to capitals.
- **Properties on nodes and relationships are *snake_case.*** Replace all spaces with an underlined character and lowercase all the words.

Defining nodes

A *node* typically represents some entity, such as a person, product, electrical junction, mouse click, or patient diagnosis. You can optionally add labels to a node, which indicates the node's role in the graph. For example, you could label a node representing a corporate customer as *Business* and *Customer,* while labeling a private individual as a *Person* and *Customer*. With these labels, you can easily find all customers, all individual customers, or all business customers and use them as starting points in graph queries. We cover graph queries more in Chapter 4.

You can add data properties to nodes. For example, you could add *first_name* and *last_name* properties to a node labeled *Person* or add an *invoice_address* property to a node labeled *Business*.

Explaining relationships

To link nodes together, you use relationships. *Relationships* are singly-typed, directed, and can optionally have properties attached to them. The type of a relationship provides a predicate (for example, *MANAGES*) while the direction of the relationship shows the subject and object (for example, Rosa manages Karl, not the other way around).

Any number of relationships of any type, in any direction can be attached to a node. Some nodes are sparsely connected, some densely. This distribution is quite normal, and the model allows for infinite variation.

Enforcing constraints

After you have the basic structures in place, you may want to structure how the graph evolves. By declaring *constraints*, you can ask the database to enforce that certain properties must be present for certain node labels or relationship types — for example, that *first_name* and *last_name* must be present on nodes with *Person* labels or a *power_rating* must be present on *POWER_LINE* relationships. You can also ask the database to ensure that fields are unique when adding a Social Security Number (*SSN*) to *Person* nodes, for example.

TIP

Unlike traditional databases where an up-front schema is required, we like to take the approach that data should grow organically where it can, and be constrained where it must. Constraints act as a schema for parts of the graph that require stronger governance, while other parts of the graph can change in a less constrained way. We call it *less-schema* rather than *schema-less.* This approach gives both flexibility and good governance.

If your query violates a constraint, it will be rolled back, keeping the data consistent.

Building a Sample Graph

In the preceding section, "Explaining labeled property graphs," we laid out the basic parts of graphs for you. In this section, you can put those tools and knowledge to work and build a simple graph. The example we provide is of an atomic family, which consists of two parents and their offspring.

Figure 1-1 shows you three nodes labeled *Person*. Inside the nodes, you see *first_name* and *last_name* properties for Alice, Bob, and Charlotte.

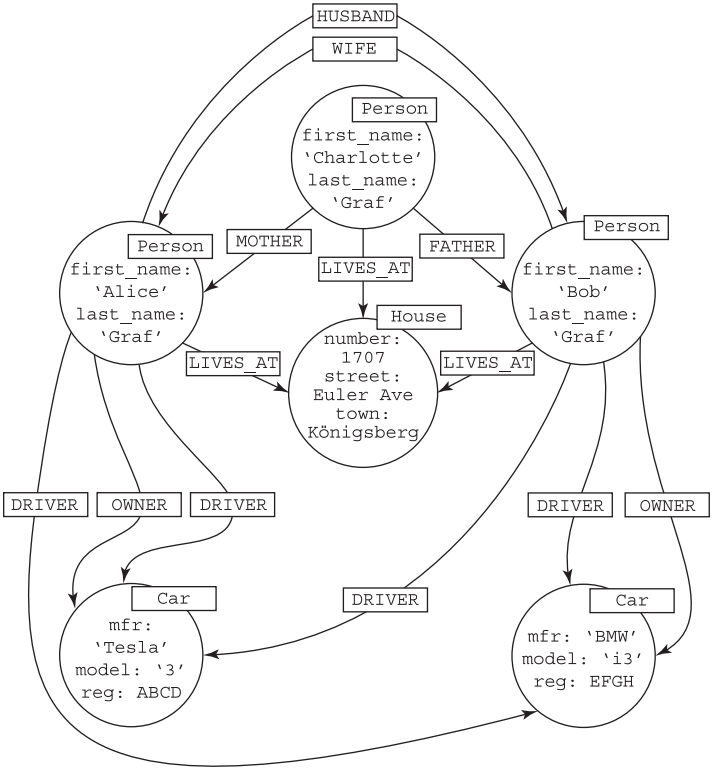

FIGURE 1-1: A graph showing a family with its home and vehicles.

You may infer some relationship between the three people in Figure 1-1 by their common last names, but the relationships between them make it explicit. Charlotte has two outgoing relationships: *MOTHER* joins her to Alice, and *FATHER* joins her to Bob. Those relationships are read as Charlotte's *MOTHER* is Alice, and Charlotte's *FATHER* is Bob.

TIP

If you read from a node along an outgoing relationship to another node, you get a sensible sentence. A good spot-check to see if the model is sound is that the nodes and relationships make logical sense. If you had made errors in this model (for example, *Car DRIVES Person)*, you'd know that you had some work to do.

You can see in Figure 1-1 where each family member lives by following the outgoing *LIVES_AT* relationship from each. Follow those lines, and you see they all live at the same address. But what's really useful about graphs is that you can ask the reverse question: Who lives at this address? And you can expect the answer in the same amount of time, which is faster when compared to other kinds of databases.

TECHNICAL STUFF

Following lines between circles doesn't seem sophisticated at first glance. But it's an example of how Neo4j, a graph database, works. Given a starting point, the database engine chases pointers around the graph until it finds the answer to your queries. Pointer chasing is a cheap and fast way of navigating data because it avoids heavyweight joins and slow index lookups that are common in relational systems.

Pointer chasing even has its own special jargon: *index-free adjacency.* Informally, it means that it's possible to traverse from a node to any of its neighbors at a low, constant cost, and from there to any of its neighbors, and so on, all at a low, constant cost per hop. This means that query time is proportional to how much of the graph the query traverses. *Query latency* — how long a query takes to run — is decoupled from the overall size of the data.

In Figure 1-1, you also see that Alice and Bob each *OWN* cars and each is the *DRIVER* of both their own and each-other's vehicles. So if Charlotte needs a ride, she can ask either Mom or Dad and be taken in either car.

You can solve several other queries with the graph in Figure 1-1. These queries include knowing who's legally allowed to drive a car, knowing where a car normally resides, and so on. You can

scale this model up to a street, town, city, or country, as well. Then add in schools, hospitals, businesses, and more to produce a much bigger and richer graph, all by repeating the same simple idioms.

Climbing the Graph Learning Curve

In a graph database, nodes can be connected by any number and type of relationship in any direction. You can use as many or as few as needed to model the domain accurately. There is no normalized form to which you must adhere: If many paths between two nodes exist, that's quite normal, just like in real life. Many folks, including us authors, have initially found this hard coming from a relational background. If this model seems too loose right now, don't despair. We help you with some modeling patterns in Chapter 2.

TIP

In a graph database, each node represents a single entity and each relationship joins two specific nodes. That means if you have a lot of products to store in the database, there will be a lot of product nodes, and if you have a lot of customers for those products, there will be a lot of relationships linking them together.

Initially, the instance-oriented view of data in graph databases seems messy. After all, a relational database collects all similar data items into their own tables and permits joins between those tables. This seems to keep complexity down, in principle. But graph databases also have abstractions that can help minimize complexity.

For example, labels are similar to tables or views, grouping together similar entities. Nodes are like rows where individual properties are grouped together. Relationships dictate which joins are legal — not at the table level like in the relational model, but at a finer granularity. So you can say that *Product* nodes are linked to *Customer* nodes via *BOUGHT* and *LIKED* relationships.

TIP

Entity-relationship diagrams from the relational world often make good design diagrams for labeled nodes and their connections in a graph model. If you can draw an Entity Relationship Diagram (ERD) to model a relational database, you can create a graph data model.

In practice, graphs are simpler than relational models. Over time, thinking in graphs becomes quite natural. We found that overwhelmingly the hardest part is letting go of relational modeling and trusting that a network of nodes and relationships can be even better.

Too good to be true? We don't think so. Head to Chapter 2 to find out how to build graph models.

GOING ALL-IN ON GRAPHS

Graphs are simple to build and highly expressive, so we think you should be using them everywhere. Well, perhaps eventually, but in today's environment, there are places where other databases are a better choice. That might seem strange coming from graph aficionados, but we think graphs follow the 80-20 rule. They're great for 80 percent of tasks because they're a general-purpose database, and they're not directly helpful for 20 percent of the tasks that have specialized needs.

But sometimes graphs can be helpful for that 20 percent, too. As an example, imagine you have a bulk storage system. It may be a data lake or perhaps an object store like Amazon's S3. These storage systems work for storing large amounts of items, but they're not great systems for reasoning about data. The data model simply doesn't care about connections; it cares about volume.

In this case, graph databases can be used as the index over the bulk store. The graph can be used to link together related items to provide curated views of the underlying items. You don't have any more of those intensive batch processing jobs needed just to find linkage between records; just search paths in the graph in real time, and then go down to bulk storage to pick out only those records you need. Adding graphs to bulk storage systems adds value.