LEARNING MADE EASY

Neo4j Special Edition

Graph Databases for dummies

A Wiley Brand

Discover graph database fundamentals

See how graph databases help your company

Understand graph models and queries

Brought to you by:

neo4j

Dr. Jim Webber

Rik Van Bruggen

About Neo4j, Inc.

Neo4j is the leader in graph database technology. As the world's most widely deployed graph database, Neo4j helps global brands — including Comcast, eBay, NASA, UBS, and Volvo — to reveal and predict how people, processes, and systems are interrelated. With this relationships-first approach, applications built by using Neo4j graph technology tackle connected data challenges such as analytics and artificial intelligence, fraud detection, real-time recommendations, and knowledge graphs. Find out more at **neo4j.com**.

Graph Databases

Neo4j Special Edition

by Dr. Jim Webber
and Rik Van Bruggen

for dummies®
A Wiley Brand

Graph Databases For Dummies, Neo4j Special Edition

Published by
John Wiley & Sons, Inc.
111 River St.
Hoboken, NJ 07030-5774
`www.wiley.com`

For general information on our other products and services, or how to create a custom *For Dummies* book for your business or organization, please contact our Business Development Department in the U.S. at 877-409-4177, contact `info@dummies.biz`, or visit `www.wiley.com/go/custompub`. For information about licensing the *For Dummies* brand for products or services, contact `BrandedRights&Licenses@Wiley.com`.

ISBN: 978-1-119-74602-7 (pbk); ISBN: 978-1-119-74579-2 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Publisher's Acknowledgments

Some of the people who helped bring this book to market include the following:

Project Editor: Carrie Burchfield-Leighton

Sr. Managing Editor: Rev Mengle

Acquisitions Editor: Steve Hayes

Production Editor: Mohammed Zafar Ali

Business Development Representative: Molly Daugherty

Table of Contents

Introduction

Graph databases have been the fastest growing database technology for almost a decade. Some people are drawn to graph databases for technical or performance reasons; others become interested because of the intuitive data model. One thing is clear: Graphs represent a departure from the relational and NoSQL models, but this departure is inherently worthwhile.

About This Book

This book is all about getting started with graph databases. We give you the basics so you can get started quickly. We want to make this a short, easy, and enjoyable journey so you feel confident building applications using graph data. We also use the Neo4j graph database for all our examples. You can run those examples, too, after downloading the Neo4j desktop app, which you can also use to build your own graph-based applications.

This book has digestible information that's aimed at the curious technical or managerial reader who wants to get started with, or wants to help others get started with, graph database technology. We hope you feel right at home and are able to find what you're looking for as quickly as possible. We hope you finish reading this book with a basic understanding of how to apply graphs to a handful of use cases and with enthusiasm for the technology.

Icons Used in This Book

Throughout this book, we occasionally use special icons to call attention to important information. Here's what to expect:

REMEMBER

When you see this icon, we give you important points to remember when designing or querying your graphs.

TECHNICAL STUFF

The Technical Stuff icon is used when we dive deeper into technical context and internals that the avid technical reader may enjoy. If you aren't that technical reader, feel free to skip over these.

TIP

For extra help on any common gotchas, tips guide you around those little annoyances while getting started.

WARNING

Watch out for this info. Steer clear of roadblocks, errors, and other issues with the information found here.

Beyond the Book

This book may not be the only resource you need when you dig down in to the details, but we hope to start you off in the right direction. To level-up your graph skills beyond what you find in this book, we recommend these resources:

- `graphdatabases.com`: Download a free copy of *Graph Databases,* coauthored by Dr. Jim Webber, coauthor of *this* book.
- `neo4j.com/sandbox`: Experience Neo4j for yourself. Get started with built-in guides and sample datasets for popular use cases.
- `community.neo4j.com`: Visit the Neo4j community website.
- `neo4j.com/graphacademy`: Neo4j offers free online training and certification for you to consume at your own pace.
- `neo4j.com/developer`: Find developer resources.
- `neo4j.com/resources`: Visit the Neo4j resources library for developers.
- `neo4j.com/use-cases`: Learn about several different graph database use cases.
- `neo4j.com/customers`: Find out what Neo4j customers have to say about graph databases.

Also, if you read through the pages of this book, please keep in mind that we'd love to hear from you and learn about your graph project success. Tweet us `@neo4j`.

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

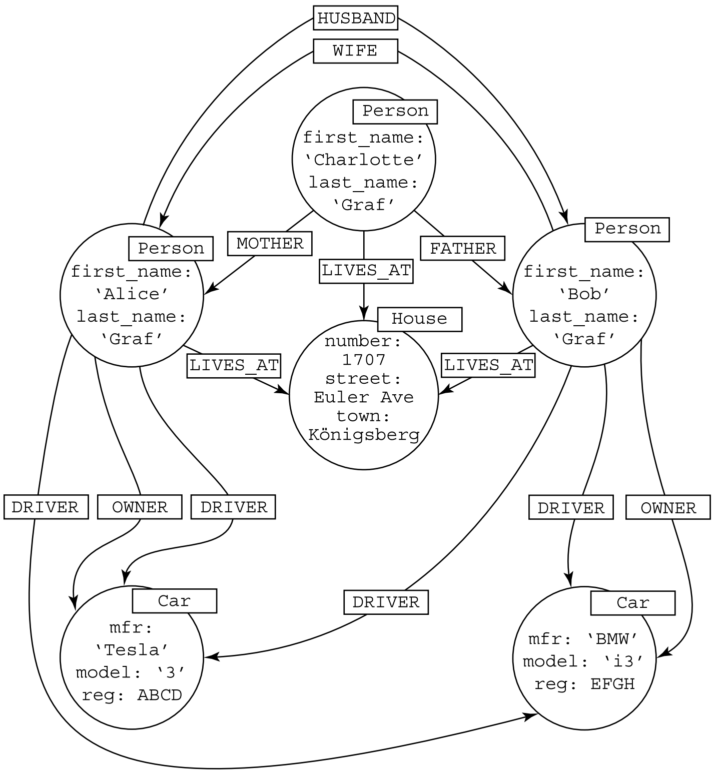
Unlike traditional databases where an up-front schema is required, we like to take the approach that data should grow organically where it can, and be constrained where it must. Constraints act as a schema for parts of the graph that require stronger governance, while other parts of the graph can change 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.



HUSBAND
WIFE
Person
first_name: 'Charlotte' last_name: 'Graf'
MOTHER
LIVES_AT
FATHER
Person
first_name: 'Alice' last_name: 'Graf'
Person
first_name: 'Bob' last_name: 'Graf'
House
number: 1707 street: Euler Ave town: Königsberg
LIVES_AT
LIVES_AT
DRIVER
OWNER
DRIVER
DRIVER
OWNER
Car
mfr: 'Tesla' model: '3' reg: ABCD
DRIVER
Car
mfr: 'BMW' model: 'i3' reg: EFGH

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.

IN THIS CHAPTER

- » **Starting at the whiteboard**
- » **Working with indirect connections**
- » **Refining the model with questions**

Chapter 2 Building Rich Graph Data Models

We cover the basics of graphs in Chapter 1. By assembling nodes, relationships, and properties, you can create graphs that are intuitive, high-fidelity representations of your domains. Graph models are easily understood by humans, and that is a big part of what makes graphs so powerful. Our brains are wired to think associatively, so we easily understand a network of concepts connected to other concepts. It's simple enough that you can even explain it to the boss!

In this chapter, we outline the approach that underpins the utility of graphs. By understanding connectivity and refining the network through answering questions, you can achieve clean, understandable models. You don't produce mysterious models that only specialists can understand; instead, your models, once visualized, can readily be understood by others who aren't technical specialists. By using this simple, powerful approach, you can

- » Quickly develop your graph data models based on a natural understanding and visual representation of these relationships
- » Easily refine the model by focusing on the added-value questions to which you want answers

A Whiteboard Works Wonders!

The best way to start developing a graph data model is to ask your business domain experts to use a whiteboard to explain their problem. Chances are they'll start drawing circles and arrows explaining the flow of information, processes, and key entities and the relationships between them. Business domain experts build the model right before your eyes.

Take a look at Figure 2-1. This graph shows you an example of a motor insurance fraud detection model. Domain experts in "crash for cash," "car ditching," or "phantom vehicle" schemes find it natural to draw links between the different entities in this model.

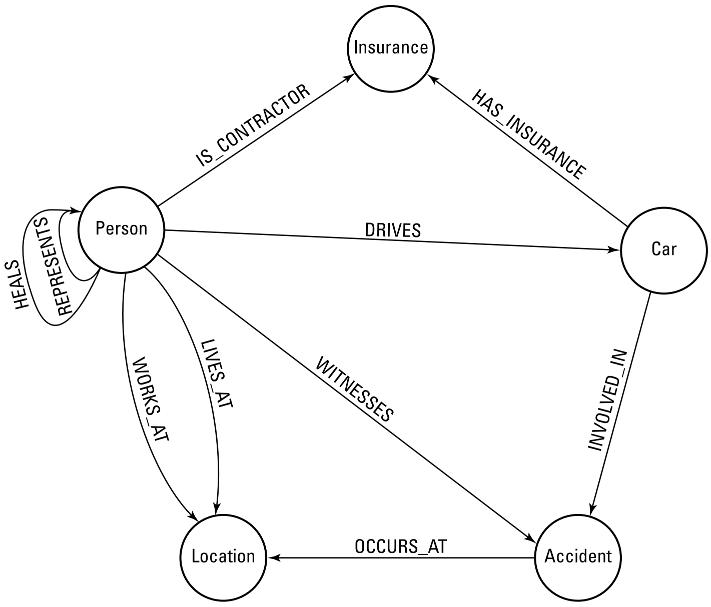


FIGURE 2-1: A graph model for car insurance fraud investigation.

In Figure 2-1, nodes are labeled with their roles and are structured with named, directed relationships. For example, a *Person LIVES_AT* a *Location*, and a *Person DRIVES* a *Car* that *HAS_INSURANCE* and was *INVOLVED_IN* an *Accident*. This is a rich model but one that's easy to understand.

TECHNICAL STUFF

Figure 2-1 is like a schema for a graph. We often talk about "schema" for a graph at the modeling level, but at the implementation level, graphs are populated by instances of data. There will be as many *Person* nodes and *Car* nodes as there are people and cars in the system. There will be as many *DRIVES* relationships as necessary to show the people who are car drivers.

People naturally think about their data in this way, so the first iteration of your graph model is rarely difficult. It requires only a whiteboard, pens, and time for discussion.

DISCOVERING INDIRECT CONNECTIONS

You may already know the direct connections among things, such as the things you buy, your friends, and the payments you make. But there's real value in going deeper in the graph, following transitive (indirect) connections — known as paths — through the graph.

Friends-of-friends are your biggest influencers; the fastest journey on the London Underground may involve line changes and may pass through many stations; an offshore account several times removed from a potential fraudster raises our suspicions. These examples are indirect connections.

Based on the model in Figure 2-1 (see the first section in this chapter, "A Whiteboard Works Wonders!"), a suspicious pattern may be where Alice drives a car that's in an accident witnessed by Bob and where Bob drives a car in an accident witnessed by Alice. This example of a fraud ring may seem trivial, but it's easily uncovered by graphs, even when they're scaled up to large criminal organizations.

Patterns are where hidden value lies, obscured from sight by other data models, but for graph models it is just a few hops away. This is why we often say that graphs are everywhere because these connections are of ever-increasing interest.

Refining the Model with Questions

After you have an initial version of your data model, you refine it by answering questions from your domain experts. Often the questions you want to answer involve many nodes and relationships where you look for patterns or traverse deeply into the graph to look for paths connecting interesting nodes.

For example, in a social recommendation system, shown in Figure 2-2, you could search for the pattern of immediate friends and friends-of-friends, then identify the products they've bought. You can omit the products you've already directly purchased, and you have a set of products ready to recommend.

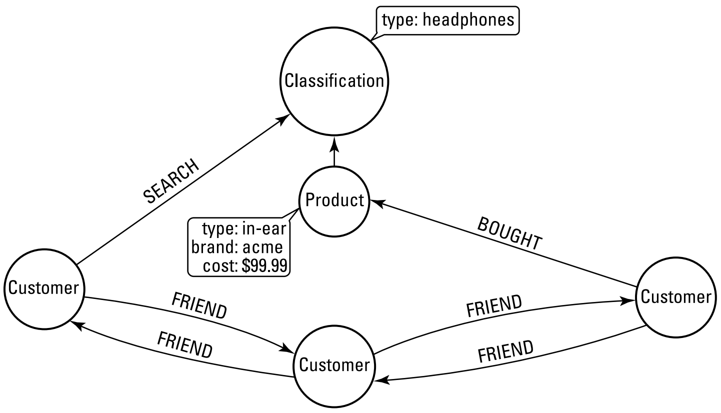


FIGURE 2-2: Following a social recommendation system that leads a user to a potential purchase.

TIP

With path searches, you may query a transport network so you can deliver products not only based on recommendations but also on delivery time, cost, and carbon footprint. In network science, designs such as Figure 2-3 are known as *layered graphs.* In this case, one layer deals with products, and another deals with logistics, which is a helpful separation of concerns for data modelers.

When you query the graph, you can choose which of these layers will be involved. For example, if you don't care about shipping logistics, you can leave out those labels and relationship types. Equally, if you don't care about product hierarchies but want to know the cheapest cost of shipping, then use only those layers. You don't pay additional penalties for storing a rich model, and when you want to combine all the layers, you can.

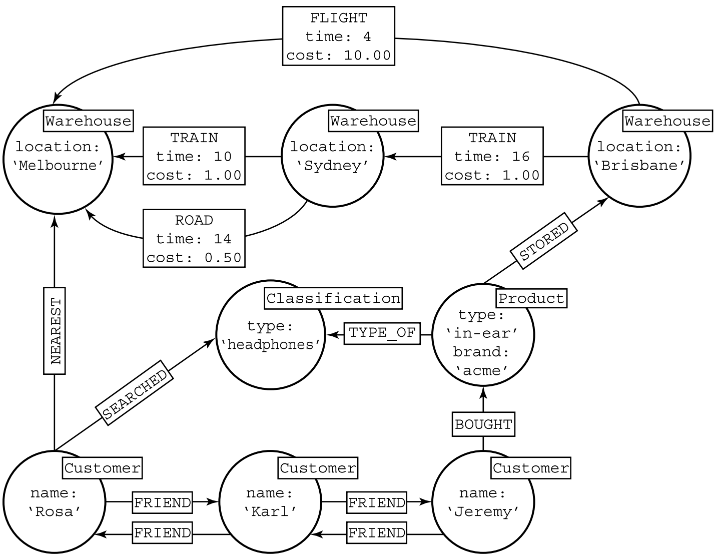


FIGURE 2-3: A simple logistics network.

Combining layers that contain entities such as customers or the classification of the product for which they're searching is obviously useful. What's less obviously useful are the intermediate entities that provide connectivity across a graph between friends and logistics hubs. This is the data you want to discover, and you use questions to refine and enrich the initial version of the graph data model. By asking, "What is the cheapest way we can ship mid-market headphones from Brisbane to Melbourne?" you drive out another part of your graph model. As you get more sophisticated, you can ask even more complex questions of your supply chain like "How much revenue do we potentially lose if our Sydney warehouse is out of action for two weeks?" all using the same question-driven method.

REMEMBER

This isn't the end of your evolution. You continue to refine your model as your business needs evolve. Refining your model is safe because the previous queries you've written act as regression tests, so your model is kept fit for the future.

IN THIS CHAPTER

- » Knowing where to start
- » Importing data into a graph database

Chapter 3 Importing Graph Data into Your Graph

After you understand how to create and refine a model, which we cover in Chapter 2, you can import data into your database, ready to serve queries.

Starting with the Model

In Chapter 2, we show you how to design a graph model and refine it as you discover other queries that are needed. Sometimes you're building a completely new system, where the data comes from your new application. But often, you're building new systems alongside existing ones, or replacing old systems that already have data, just not yet in a graph.

As an example, imagine you have a single table of product purchases (a comma-separated values, or CSV, in a file works similarly). You want to import this table structure into a graph so you can query it. Your example looks like the tabular data in Figure 3-1.

Taking into consideration your data in Figure 3-1, your target graph model will look like Figure 3-2.

Person	Product	Date
Emil Eifrem	Phone	22/05/2020
Emil Eifrem	Computer	20/05/2020
Jim Webber	Computer	08/05/2020
Jim Webber	Bike	12/05/2020
Lance Walter	Fitness equipment	05/05/2020
Lance Walter	Computer	13/05/2020
Lars Nordwall	Boat	21/05/2020
Lars Nordwall	Fitness equipment	11/05/2020
Lisa Hatheway	Phone	27/05/2020
Lisa Hatheway	Boat	10/05/2020
Rik Van Bruggen	Bike	06/05/2020
Rik Van Bruggen	Tablet	27/05/2020

FIGURE 3-1: A tabular dataset of people buying products.

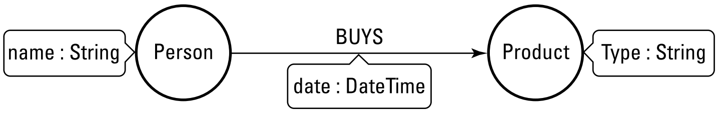


FIGURE 3-2: A graph model for people buying products.

To import your data, follow this strategy:

1. **Convert the *Person* entities in the first column into *Person* nodes.**
2. **Convert the *Product* entities in the second column to *Product* nodes.**
3. **Create a *BUYS* relationship from *Person* to *Product* for every row in the table and assign the date in the third column to the *date* property on that relationship.**

After you fill the model with your data, you get the graph, as shown in Figure 3-3.

Victory! You've successfully loaded data into a graph from a tabular data source.

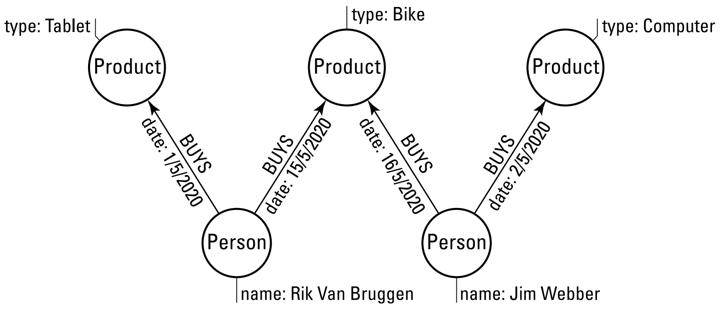


FIGURE 3-3: The final graph representation of people buying products.

WARNING

Life is never *that* simple, and there are a few gotchas to avoid. Importing data sounds easy enough, but to be successful, keep in mind a couple of details:

- **Start small; scale later:** Test your import strategy on reasonable data sizes (think thousands, not billions of entities to be imported) before you scale them. Check out the later section "Importing Made Easy" for more information on import strategies.
- **Get rid of the duplicates:** Most useful datasets have some data quality problems, but importing even the cleanest datasets may cause duplicate data in your models if you aren't careful.

Importing Made Easy

Importing data into a graph database, like any data-intensive operation, requires some wrangling: The data has to move from one (tabular) data model to a graph data model, requiring transformations in the process. In Neo4j, for example, import tooling has two variants:

- **Online CSV import:** Starts from a CSV file and imports the data in a file into a running (online) graph database
- **Offline CSV import:** Starts from a CSV file and imports a file into a graph database file structure, offline, ready for the database to be started

Online and offline CSV imports have pros and cons, which we cover in this section.

Online CSV importing

Good graph databases have a built-in ability to import data from CSV files on your filesystem or over the network. In Neo4j, the LOAD CSV command takes the data in your import file and maps it to nodes, relationships, and properties in a running database. The LOAD CSV command performs a transactional update: It will either completely commit or rollback, so it's simple to use. For example, the following snippet loads CSV data into a graph of people who bought products:

```
LOAD CSV WITH HEADERS FROM
  "personbuysproduct.csv" as csv
CREATE (p:Person {name: csv.Person})
  -[b:BUYS {date: csv.Date}]->
  (pr:Product {name: csv.Product})
RETURN "Import Successful!"
```

This import creates a number of unique, disconnected *Person BUYS Product* subgraphs, as shown in Figure 3-4. The Neo4j Browser displays light gray nodes that represent people and dark gray nodes represent the products they've bought. There's massive duplication and little connectivity — sure signs of a poor graph.

This graph isn't quite right because there are duplicated people and products due to the CREATE statement always creating new nodes and relationships regardless of the current state of the graph. Instead, you should use the MERGE command, which does a combination of two things:

- It will try to find the pattern that you're specifying (a node, a relationship, or a combination of both), like a MATCH statement (see Chapter 4 for more information).
- If MERGE finds existing matches, it leaves them alone. If no matches are found, MERGE creates new elements in the database.

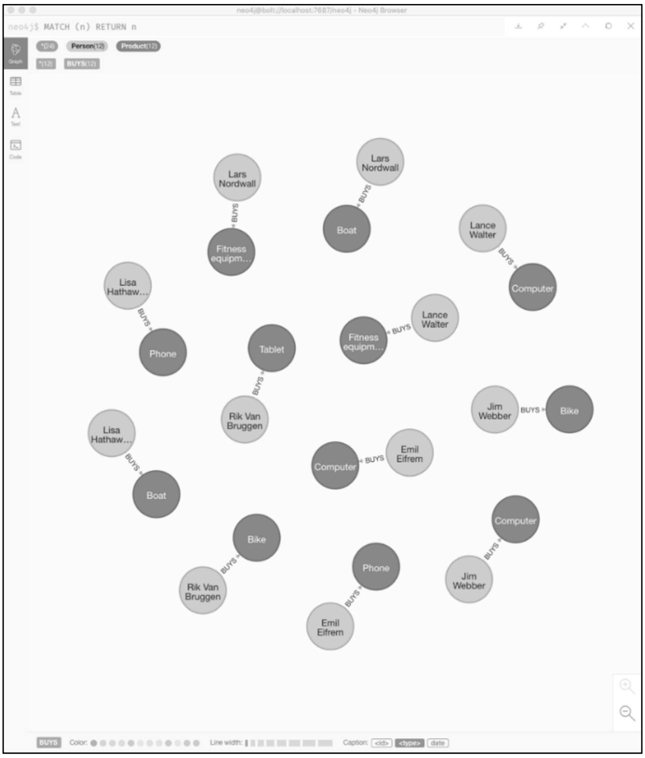


FIGURE 3-4: An online import of data.

By modifying the import statement to use MERGE, you get the following:

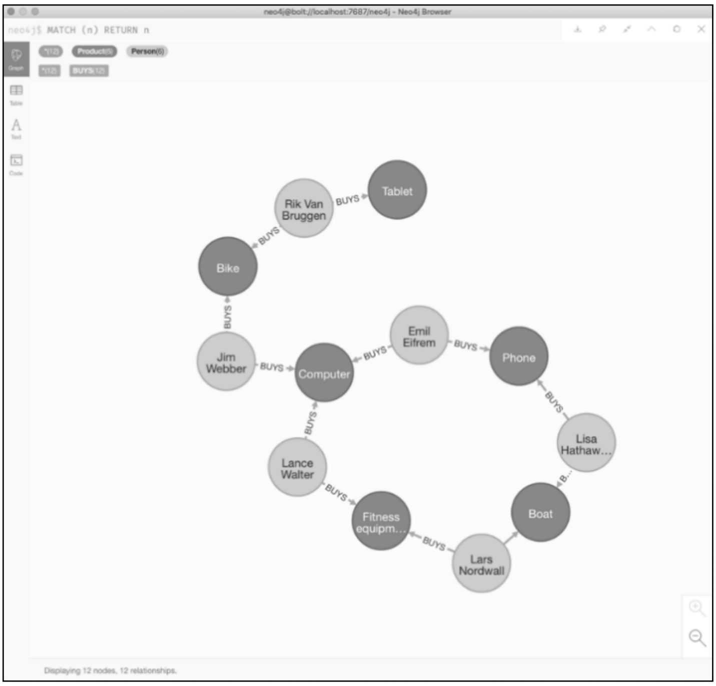
```
LOAD CSV WITH HEADERS FROM
  "personbuysproduct.csv" AS csv
MERGE (p:Person {name: csv.Person})
MERGE (pr:Product {name: csv.Product})
CREATE (p)-[b:BUYS {date: csv.Date}]->(pr)
RETURN "Import Successful!"
```

In this case, you have to split the query into three key parts:

- MERGE: The first MERGE either matches an existing *Person* node, or it fails to match an existing *Person* node and creates one.

- **»** MERGE: The second MERGE either matches an existing *Product* node, or it fails to match an existing *Product* node and creates one.
- **»** CREATE: This part creates a relationship between the person and product. Duplicates are allowed here because a person can buy multiple products.

Figure 3-5 shows you the correct graph. The light gray *Person* nodes connect to dark gray *Product* nodes in a connected, de-duplicated graph as intended.


neo4j@bolt://localhost:7687/neo4j - Neo4j Browser

```
neo4j$ MATCH (n) RETURN n
```

Displaying 12 nodes, 12 relationships.

FIGURE 3-5: Correctly importing data without duplicates.

TIP

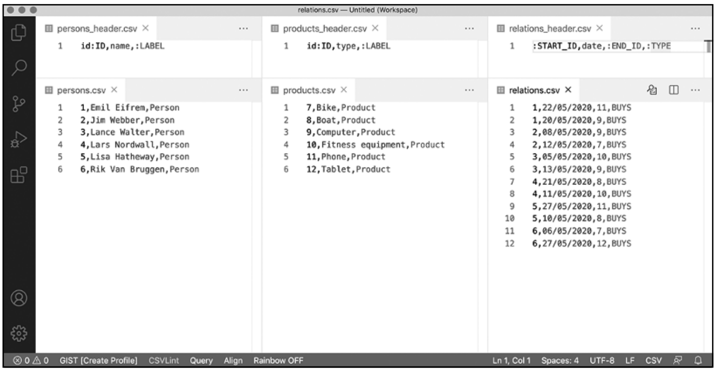
MERGE is a tricky concept. Naively trying *MERGE (:Person)-[:BUYS]->(:Product)* will create a lot of new people and products. The reason is that MERGE matches the *whole* pattern, and if it can't *match* the whole pattern, it *creates* the whole pattern (we cover more about patterns in Chapter 4). For this reason, we recommend having several *MERGE* statements with smaller patterns — for example, just a single node, instead of a single *MERGE* with a large pattern.

Offline CSV importing

LOAD CSV (see the preceding section) is fine for small- to medium-sized data import jobs. The transactional behavior means you can use it alongside your regular queries. But when your import job becomes too large, switching to a non-transactional, offline import method may be the way forward. For large import jobs, such as commissioning a database, using an offline import tool is faster. Offline CSV import is fast, but to get the best out of it, you need to wrangle data before using it. Create separate CSV files for

- Every node type that you want to import
- Every relationship type that you want to import

For each import file, you must also create a header file that describes the structure of the corresponding node/relationship file. For example, that means you need six CSV files to import the people-buying-products data, as shown in Figure 3-6.



persons_header.csv
```
id:ID,name,:LABEL
```

products_header.csv
```
id:ID,type,:LABEL
```

relations_header.csv
```
:START_ID,date,:END_ID,:TYPE
```

persons.csv
```
1,Emil Eifrem,Person
2,Jim Webber,Person
3,Lance Walter,Person
4,Lars Nordwall,Person
5,Lisa Hatheway,Person
6,Rik Van Bruggen,Person
```

products.csv
```
7,Bike,Product
8,Boat,Product
9,Computer,Product
10,Fitness equipment,Product
11,Phone,Product
12,Tablet,Product
```

relations.csv
```
1,22/05/2020,11,BUYS
1,20/05/2020,9,BUYS
2,08/05/2020,9,BUYS
2,12/05/2020,7,BUYS
3,05/05/2020,10,BUYS
3,13/05/2020,9,BUYS
4,21/05/2020,8,BUYS
4,11/05/2020,10,BUYS
5,27/05/2020,11,BUYS
5,10/05/2020,8,BUYS
6,06/05/2020,7,BUYS
6,27/05/2020,12,BUYS
```

⊗ 0 ⚠ 0 GIST [Create Profile] CSVLint Query Align Rainbow OFF
Ln 1, Col 1 Spaces: 4 UTF-8 LF CSV

FIGURE 3-6: Six files prepared for offline import.

In this Neo4j example, you invoke the offline CSV importer by using the Neo4j admin tool (neo4j-admin) to perform the import:

```
bin/neo4j-admin import
--nodes import/persons_header.csv,import/persons.
  csv
--nodes import/products_header.csv,import/
  products.csv
```

```
--relationships import/relations_header.
  csv,import/relations.csv
--database offlineimport
```

This import command works in an all-or-nothing fashion: It either completely succeeds or completely fails. It will also completely replace the structure and data of any existing database that you may already have in place. You'll find that this command is also fast — billions of records per hour.

IN THIS CHAPTER

» **Learning the Cypher query language**

» **Building more sophisticated patterns**

» **Managing graph updates**

» **Pruning the search space with predicates**

» **Supporting aggregate functions**

» **Returning insight to the user**

Chapter 4 Querying Your Graph

In this chapter, we show you how to query your graphs using the Cypher query language.

De-Cyphering Graphs

You may be used to writing SQL for relational databases. The equivalent of SQL for graph databases is Cypher, a de-facto standard originally developed by Neo4j, Inc., and a driver behind the International Organization for Standardization (ISO) Graph Query Language (GQL) project.

Cypher is a friendly declarative, pattern-matching language. Users draw pictures of what they want to find by using ASCII art. At the heart of Cypher is the *MATCH* clause. With *MATCH*, you draw ASCII art pictures of patterns that you want the database to find in your graph — for example, if you have a social network with *Person* nodes connected by directed *FRIEND_OF* relationships. You can find your direct friends in a social network by running the following query:

```
MATCH (:Person {name:'Alice'})
        -[:FRIEND_OF]->(p:Person)
RETURN p
```

Let us break down this query:

- **MATCH:** Tells the database that it has to look for a pattern
- **(:Person {name:'Alice'}):** Any node with (at least) the *Person* label and a property with key *name* and value *Alice*

  This part of the query binds the rest of the query to a starting point (or points, if there's more than one Alice) in the graph.
- `-[:FRIEND_OF ]->:` Any outgoing relationship typed *FRIEND_OF*, where the > specifies the direction
- **(p:Person)**. Any node with (at least) the *Person* label, bound to the variable *p,* which can be referenced at any later point in the query
- **RETURN p:** Returns any matching *Person* nodes to which *p* was bound

The query tells the database to first find all *Person* nodes with a *name:Alice* property, and from there, find all the outgoing *FRIEND_OF* relationships that terminate at another *Person* node. Figure 4-1 gives you the result — Alice's friends, those who are one "hop" away from her: Rosa, Antonio (who doesn't consider Alice a friend), and Bob.

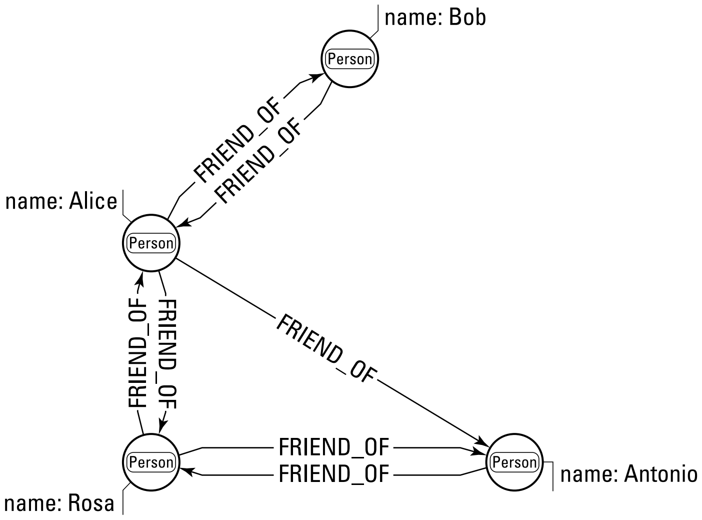


FIGURE 4-1: Finding Alice's direct (depth-one) friends.

You can extend this query to one and two hops away from Alice to include her friends and friends-of-friends:

```
MATCH (a:Person {name:'Alice'})
        -[:FRIEND_OF*1..2]->(p:Person)
WHERE a <> p
RETURN p
```

For finding friends-of-friends, there are three modest changes:

- **`-[:FRIEND_OF*1..2]->` :** Any outgoing relationship typed *FRIEND_OF* at exactly 1 or 2 hops from the originating node
- **`(a:Person {name:'Alice'})`:** Binds Alice nodes to the variable *a* so they can be referred to later in the query
- **`WHERE a <> p`:** Excludes Alice from the results because Alice is a friend of Rosa, who is a friend of Alice, which means Alice is a friend-of-friend to herself

Figure 4-2 shows you the results. This extended query shows Alice's friends of friends that extend to Karl. You can also restrict the query to only friends-of-friends by changing the relationship depth to exactly two: `-[:FRIEND_OF*2]->`.

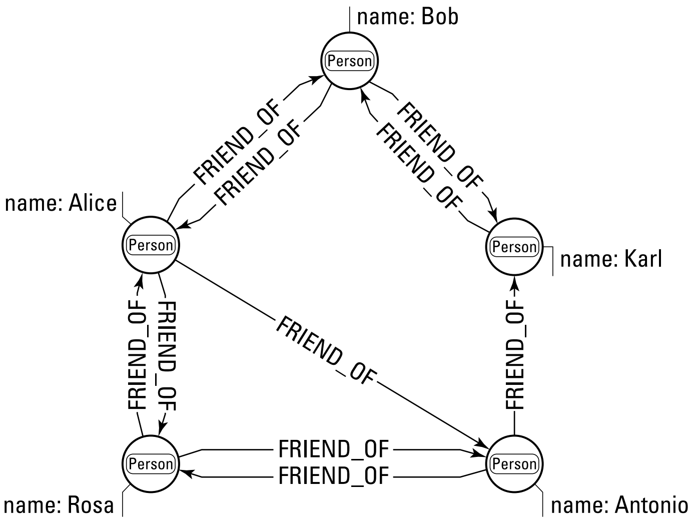


FIGURE 4-2: An extended query to show Alice's friends of friends.

Building Bigger Patterns

After you know the *MATCH* syntax (see the preceding section), you can start to build much more sophisticated patterns. For instance, in a social recommendation system, you could look for products that your network of friends have bought through running this query:

```
MATCH (a:Person {name:'Alice'})
        -[:FRIEND_OF*1..2]->(p:Person)
        -[:BOUGHT]->(prod:Product)
WHERE a <> p
RETURN prod
```

To turn the social network into a simple recommendation graph, you make a small number of simple changes:

- **`-[:FRIEND_OF*1..2]->` :** Any outgoing relationship typed *FRIEND_OF* at depth 1 to 2 from the originating node, meaning direct friends and their direct friends only
- **`(:Person)-[:BOUGHT]->(prod:Product)`:** Matches any outgoing *BOUGHT* relationships from the friend or friend of friend that terminate in a *Product* node; also binds that node to the variable *prod* for later use in the query.
- **`RETURN prod`:** Returns any matched *Product* nodes bound to variable *prod* at the end of the query

TECHNICAL STUFF

Graph databases use indexes to find starting points for your pattern matching queries. In Neo4j, any node labels are automatically indexed, but you may also choose to add other indexes on important properties in your graph. Lacking an index on a frequently searched property is a common reason why queries run slowly.

After you understand basic patterns, you can adapt them in many ways. For example, Cypher has an *OPTIONAL MATCH* clause that matches patterns against your graph, just like a *MATCH* does (see "De-Cyphering Graphs" earlier in this chapter). The difference is that if no matches are found, *OPTIONAL MATCH* uses an empty space for missing parts of the pattern. Be aware that *OPTIONAL MATCH* often takes more time to process than *MATCH* because it's less strict, so use it sparingly.

The more specific you can make a pattern the better. If you know a relationship's type and direction, include them in your query pattern. If you know a specific node label, include that too. In fact, the more specific you are with your queries, the less work the database will have to perform to retrieve the answer because it will need to explore fewer items, shrinking the search space so queries run faster. In building bigger patterns, you can also

- Loosen the *MATCH* pattern by loosening the constraints.
- Change the direction of relationships or omit direction altogether (which matches either direction).
- Specify several relationship types or none at all (which matches any), and do the same with node labels.
- Specify a depth on relationships, or use the * wildcard to allow any depth depending on your needs.

These processes are all quite flexible and natural:

- `()` matches any node
- `--` matches any relationship
- `(:Person)-->()` matches any relationship that's outgoing from a *Person* node to any other node
- `(:Person)-[:A|B]->()` matches either *A* or *B* relationship types outgoing from any *Person* node to any other node

Updating the Graph

In Chapter 3, you create a graph from scratch using *CREATE* and *MERGE*. To update the graph, you can also use

- *SET:* Allows you to change or add a property. For example, the following query sets any *Person* nodes in the graph with the property key *name* and value *Rosa* to create or update a property with key *surname* to value *Luxemburg*:

```
MATCH (p:Person {name:'Rosa'})
SET p.surname='Luxemburg'
```

» *REMOVE:* Does the opposite of *SET*:

```
MATCH (p:Person {name:'Rosa'})
REMOVE p.surname
```

» *DELETE:* Removes detached nodes and relationships along with their properties. For example, the following query removes a *FRIEND_OF* relationship from *Rosa* to *Karl:*

```
MATCH (:Person {name:'Rosa'})
   -[f:FRIEND_OF]->(:Person {name:'Karl'})
DELETE f
```

WARNING

» *DELETE* is safe by default. If you try to delete a node that's still attached to relationships, it will fail. There's also *DETACH DELETE,* which removes the entire matched pattern and any dangling relationships (so use it carefully). The query looks like this:

```
MATCH (k:Person {name:'Karl'})
DETACH DELETE k
```

Filtering with Predicates

Predicates are functions that return true or false results for a set of non-empty input. They're used to focus a query as part of a *WHERE* clause. For example, you may want to send an invitation to a party, leaving out friends who are too young, because your party is at a bar. To perform this query, you use a predicate on the *age* property of *Person* nodes:

```
WHERE a <> b AND b.age > 20
```

TIP

If you know any values for properties, make sure to include them as predicates. The database query planner can use them to reduce the search space, which makes your queries run faster.

Alternately, you may want to invite only married members of your social network by specifying a pattern for married folks. To do this, you use the *EXISTS* function to specify that nodes bound to the variable *b* should have a *MARRIED_TO* relationship in any direction — so no < or > in the pattern — to any other node with label *Person.* That query looks like this:

```
WHERE a <> b AND
   EXISTS ( (b)-[:MARRIED_TO]-(:Person) )
```

You can rewrite this query for all the unmarried folks, too, by replacing *EXISTS* with its logical opposite, *NOT*:

```
WHERE a <> b AND
   NOT ( (b)-[:MARRIED_TO]-(:Person) )
```

A twist on this query would be where you'd match only if all the friends in your network were able to attend:

```
WHERE a <> b AND
   NONE (x IN nodes(b) WHERE x.age < 21)
```

The *NONE* predicate function in the above query ensures that no matches at all will be returned if any of the social network members are minors (under 21 years of age), so either everyone gets invited, or no one does.

Aggregating Data

After you've received matches back from a graph query, you may want to further process that data. For example, you want to know the number of friends in your social network rather than the individuals, or the price range of a number of products that you're being recommended. To do this, the graph database supports aggregate functions such as *count*, which in the following query returns the number of people in Alice's social network:

```
MATCH (a:Person {name:'Alice'})
       -[:FRIEND_OF _OF*1..2]->(p:Person)
WHERE a <> p
RETURN count(p)
```

If you want to know the average age of Alice's social network, swap *count* for *avg* on any *age* properties in *Person* nodes:

```
MATCH (a:Person {name:'Alice'})
       -[:FRIEND_OF _OF*1..2]->(p:Person)
WHERE a <> p
RETURN avg(p.age)
```

In addition, the graph database supports many other common aggregate functions like *MIN*, *MAX*, and some statistical functions as well as lists aggregates, such as *reduce*:

```
RETURN reduce(totalAge = 0,
  n IN nodes(p)|totalAge + n.age)
  AS totalAges
```

The reduce function allows you to add the ages of people in Alice's social network, or you can more generally add a property value over a list.

Returning Insight

After you're done with matching, predicates, and aggregate functions, the final step is to return answers to the user. If you've been reading this book to this point, you may recall that the social network query example ends with *RETURN p* to return all matching *Person* nodes. You can easily send those matches back in ascending order by using *ORDER BY*:

```
RETURN p
ORDER BY p.age ASC
```

WARNING

ORDER BY is computationally heavy because it must sort all matched records on the server before returning results to the user. Use it sparingly.

Matches can also be ordered in descending order by adding *DESC*. If you want to limit the number of results returned, then use *LIMIT*.

```
RETURN p
ORDER BY p.age DESC
LIMIT 10
```

IN THIS CHAPTER

- » Connecting to the database
- » Using Neo4j clusters for availability
- » Monitoring systems and performing backups
- » Coexisting with other systems

Chapter 5 Using Graphs in Production with Neo4j

In this chapter, we show you how to move your graph into a production setting by using the Neo4j graph database.

Connecting to the Database

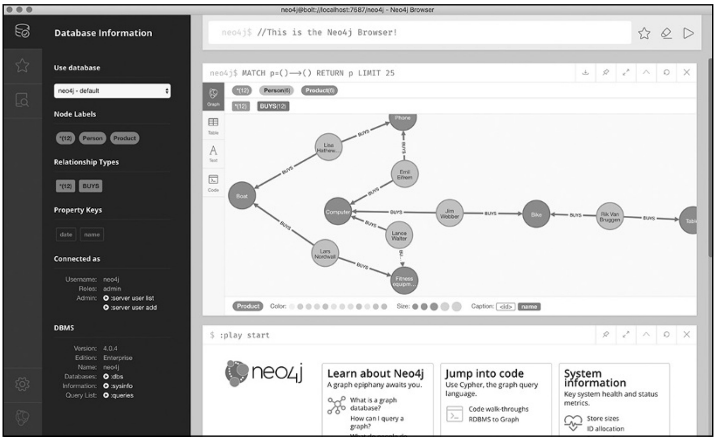
Broadly speaking, you run queries on your graphs in two ways:

- » When you're prototyping in development or running ad-hoc queries on a stable system
- » From your application, where you have a known set of queries that you want to send to the database

In each case in this section, we use Neo4j as the example to help bridge the gap.

Neo4j Browser

The first piece of software that you come across when using Neo4j is the Neo4j Browser. It's a graphical user interface (GUI) tool that allows you to rapidly prototype Cypher queries and display the resulting graphs. (We cover Cypher queries in more detail in Chapter 4.) Neo4j Browser, shown in Figure 5-1, comes bundled with Neo4j.



```
neo4j$ //This is the Neo4j Browser!
```

```
neo4j$ MATCH p=()-->() RETURN p LIMIT 25
```

```
$ :play start
```

Username: neo4j
Roles: admin
Admin: :server user list, :server user add

Version: 4.0.4
Edition: Enterprise
Name: neo4j
Databases: :dbs
Information: :sysinfo
Query List: :queries

FIGURE 5-1: The Neo4j Browser.

With the Neo4j Browser, you can try out various data models and prototype Cypher queries. You simply write your Cypher query in the box at the top of the browser and execute the query, which visualizes your results as a graph. You repeat this until you're happy with the results.

TIP

If you're doing data science or performing a lot of ad-hoc queries, try the Neo4j Bloom visualization tool for natural-language graph searches and the Neo4j Graph Data Science library for high performance graph analytics algorithms.

After you're happy with your query, the next step is to embed it into your application.

Neo4j drivers

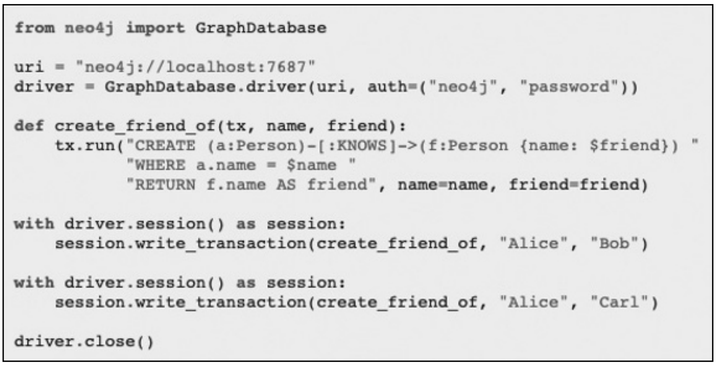
Most of the time, databases aren't exposed directly to end-users — they're encapsulated behind applications or services.

Like most databases, Neo4j is a client-server system. A client application or service sends messages containing Cypher queries to the database server that evaluates those queries and responds with a stream of results.

TECHNICAL STUFF

You can run Neo4j embedded in your own application process, if you're running on the JVM to avoid a client-server setup. This isn't common though. It tends to be used mostly by embedded systems.

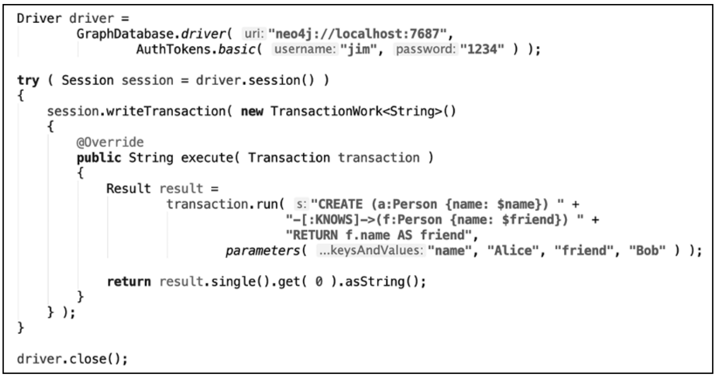
Neo4j is mostly written in Java and Scala running on the Java Virtual Machine (JVM), but the drivers present an interface to the database that looks how you may expect in your preferred programming language. For example, you can query your graph easily from Python, as shown in Figure 5-2.



```
from neo4j import GraphDatabase

uri = "neo4j://localhost:7687"
driver = GraphDatabase.driver(uri, auth=("neo4j", "password"))

def create_friend_of(tx, name, friend):
    tx.run("CREATE (a:Person)-[:KNOWS]->(f:Person {name: $friend}) "
           "WHERE a.name = $name "
           "RETURN f.name AS friend", name=name, friend=friend)

with driver.session() as session:
    session.write_transaction(create_friend_of, "Alice", "Bob")

with driver.session() as session:
    session.write_transaction(create_friend_of, "Alice", "Carl")

driver.close()
```

FIGURE 5-2: Writing a social graph via the Python driver.

In the process shown in Figure 5-2, you import the Neo4j components from the GraphDatabase Python package and initialize the driver with its connection string and security parameters. Then, you create a session to the database through which you send (write) transactions that invoke the *create_friend_of* method to create Cypher that creates nodes and relationships in the database. The same operation in Java code is shown in Figure 5-3.



```
Driver driver =
        GraphDatabase.driver( uri: "neo4j://localhost:7687",
                AuthTokens.basic( username: "jim", password: "1234" ) );

try ( Session session = driver.session() )
{
    session.writeTransaction( new TransactionWork<String>()
    {
        @Override
        public String execute( Transaction transaction )
        {
            Result result =
                    transaction.run( s: "CREATE (a:Person {name: $name}) " +
                                    "-[:KNOWS]->(f:Person {name: $friend}) " +
                                    "RETURN f.name AS friend",
                            parameters( ...keysAndValues: "name", "Alice", "friend", "Bob" ) );

            return result.single().get( 0 ).asString();
        }
    } );
}

driver.close();
```

FIGURE 5-3: Writing a social graph with the Java driver.

Neo4j provides drivers for .NET, Java, JavaScript, Go, and Python — each of which gives you easy access to the database using Cypher. In addition, the Neo4j open source community has written drivers for Ruby, PHP, R, Erlang / Elixir, C/C++, Clojure, Perl, Haskell, and others.

TECHNICAL STUFF

There are integrations of these drivers with popular programming frameworks, too. For example, there's tight integration with the Spring framework, including object-graph mapping (OGM) to make development simpler.

Keeping Data Safe with Neo4j Servers

A single Neo4j server is fine for development or ad-hoc queries, but it's not robust enough for production systems that require continuous availability or to sustain high workloads. To handle this, you run a cluster of Neo4j servers. Neo4j clustering has two roles for machines in the cluster: Core Servers and Read Replicas.

Core Servers

Core Servers process transactions and are responsible for keeping your data safe. Neo4j Core Server clusters are simple to understand: If a majority of servers are working, the cluster is working; otherwise, it becomes read-only for safety.

Setting up a cluster works on physical servers, cloud instances, and in containers. You choose how many servers to deploy based on your workload and redundancy requirements. For example, a high-workload, highly resilient cluster may need nine servers across three data centers, so it can lose up to four servers and continue processing (albeit with reduced capacity). A more modest workload and redundancy requirement might be just three servers, where you can lose one and continue processing.

After you have a cluster set up, the drivers connect to it in the usual way. There's nothing special to set up: Just point your driver at a cluster member, and the cluster topology is automatically discovered, and queries are load-balanced transparently.

Neo4j clusters are easy to use from a development point of view. You don't have to worry about the timing of server replication because Neo4j ensures that the results of your writes are always visible when you read, even if you write and read from servers on opposite sides of the world.

Read Replicas

Read Replicas are responsible for scaling out to process more queries. You can add Read Replicas to the cluster to scale out operations like analytics or reporting workloads. These read-only servers aren't involved in cluster management; instead, they act like a continuously refreshed copy of your graph.

Monitoring Systems

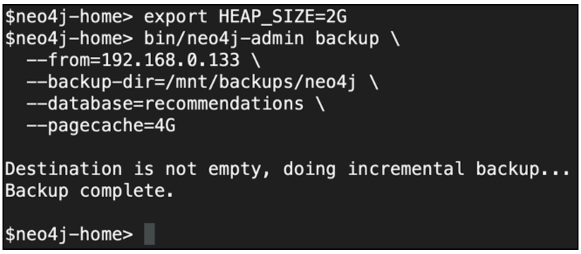
Neo4j clusters support a variety of monitoring capabilities:

- A range of cluster metrics, such as threads, requests, memory, and so on
- Individual database metrics, such as transactions and checkpoints, that are exposed through the Graphite protocol, a Prometheus endpoint, CSV files, and JMX MBeans
- User-facing logs for general communications, security, and queries
- Query management — including termination of errant queries
- Client connection management

Performing Regular Backups

Even with a highly available cluster, disasters can happen. No amount of clever Java code can prevent a database from becoming unavailable when its network connection is physically cut. To keep data safe, perform regular backups — frequency determined by your disaster response policy — and potentially invest in a disaster recovery site.

By using the neo4j-admin tool, shown in Figure 5-4, you can take full or incremental backups of your Neo4j server (including a server that's a member of a cluster) while that server is still running regular queries.



```
$neo4j-home> export HEAP_SIZE=2G
$neo4j-home> bin/neo4j-admin backup \
  --from=192.168.0.133 \
  --backup-dir=/mnt/backups/neo4j \
  --database=recommendations \
  --pagecache=4G

Destination is not empty, doing incremental backup...
Backup complete.

$neo4j-home>
```

FIGURE 5-4: A *neo4j-admin* backup example.

TIP

We recommend running backup from another machine so the process doesn't contend for resources with the running database.

Integrating with Other Systems

Graph databases in production have to co-exist with other systems in production as sources and destinations for data flows that power modern enterprises. At a technical level, many patterns for data integration are available, but we show you two in this section that we see often.

ETL tools

Data integration is a key topic for many enterprises. To address the challenge, an entire category of tools exists that specialize in ETL:

- **E**xtracting data from existing formats and systems
- **T**ransforming data from an existing format to a new format
- **L**oading the newly formatted data into a target system

REMEMBER

Most ETL tools are able to connect to Neo4j as both a source and a target system. The idea behind ETL tools is that they can — in batch or real time — take data from one system and load it into another, performing any computations or transformations needed along the way. By using these tools, it's possible to take, for example, data from a web server log, transform and load it into a graph for behavioral analysis, and then extract some of that data for loading into a data lake.

Streaming

Many organizations have adopted streaming platforms as a means of integrating systems. With streaming integration, your graph can be updated in response to a variety of events received from around your enterprise. It can also trigger events for downstream systems to handle. From a systems architecture perspective, graph becomes a standard event-processing module, fitting cleanly into a modern architecture.

Apache Kafka is a popular choice for asynchronous systems integration, and it integrates well with Neo4j. Neo4j Streams and Kafka Connect are the two libraries that support streaming integration. You'd use Neo4j Streams if you're more comfortable operating Neo4j, and you'd choose Kafka Connect if you're more comfortable administrating Kafka.

Neo4j Streams integrates Neo4j with Apache Kafka event streams to serve as a source of data — for instance, change data capture (CDC) — or to ingest any kind of Kafka event into your graph. Kafka Connect does the same but is limited to treating Neo4j as a store for data at the moment.

IN THIS CHAPTER

» **Picking the right tool for the job**

» **Starting small and scaling up**

» **Answering hard questions about data modeling**

» **Connecting with the graph community**

Chapter 6 Ten Tips for Creating Successful Graphs

You've arrived at the famous *For Dummies* Part of Tens chapter. In this chapter, we give you our top ten tips for creating successful graph applications.

Use the Right Tool for the Right Job

Stepping outside of your comfort zone isn't always easy. You already know how relational databases work, and you can create amazing, brilliant workarounds for their limitations. But one day, you realize that problems that are really hard for relational databases are actually simple if you only change the data model. So, make sure to use the right tool for the right job — if you're querying paths, trees, or networks, then a graph database will make your life easier. Look for tips on finding the right tools in Chapter 1.

Make Connections

Most problems are conveniently and intuitively modeled as graphs, but until you're into graphs, you don't necessarily see it. We call this the "graph problem-problem." You can't easily spot graph problems until you're an expert, but you can't become an expert without spotting graph problems. As a short-cut to get you started, graphs work best where you have a lot of connections and where you can derive useful insights from these connections.

Take Advantage of Speed

You can functionally solve most problems in a number of alternative ways. Have you ever heard the saying, "If all you have is a hammer, everything looks like a nail"? Oh, and there's its corollary: "If all you have is a hammer, everything looks like your thumb." You can find a host of tools that allow you to process graphs, but the question is whether you can do this at speed.

Speed changes everything: It allows you to iterate and experiment more quickly, it allows you to do things in real time that you thought were only possible to do in overnight batches, and it allows you to save hardware, software, and operational costs because you can do more with less.

TIP

Make sure to use tools that can process vast numbers of connections at high speed. If you're only looking at a few dozens of nodes and relations, then fine, but as soon as you need to treat thousands/millions/billions of entities and their connections, be sure to use a graph database.

Use Graphs for Obvious Use Cases

Value in connections is abundant. Graphs are everywhere, but leveraging them efficiently often starts with obvious use cases that have been tried and tested. When you or your organization thinks about anything like social networks, knowledge graphs, real-time fraud detection, hyper-personal recommendation engines, or master data management, consider graphs. They're excellently suited for the job — and there are plenty of proof points from existing systems to back up that claim.

Begin with Modeling

Good graph-based systems begin with modeling. Your tried and true modeling techniques have served you well throughout your career, and we know they're hard to let go. But graph modeling is different, and perhaps a bit daunting at first, yet it has more degrees of freedom. If you push through the initial hurdles, you'll find it will feel natural and liberating in no time.

Start Small, Scale Next

After you have a graph data model, think about how you're going to write data into it. You have several different choices, but evaluate those options, and choose the most appropriate one for your task. We recommend starting small by using tools that offer fast feedback on your model and scaling later by using tools that support large data ingestion.

Model for Questions

Think about the questions that you want to ask of the graph data — that process is a prerequisite for good modeling and querying. Look for questions that you would struggle to answer in traditional data models. A lot of joins, recursion, and unknown-depth pathfinding are examples of graph queries that can get you to value quickly. Chapter 2 gives you tips on modeling, and Chapters 3 and 4 tell you how to get started modeling.

Focus on Value

IT systems provide value when they enter production. Putting graphs into production is delivering value into the hands of business users. Graph databases provide many techniques for ensuring performance, efficiency, and availability but need to be appropriately operationalized. Chapter 5 gives you helpful advice on how to get your graph-based system into production.

Explore Hidden Insights

Graphs are most valuable when you use them to find patterns and make predictions about future dynamics of the network. Graph data science is a sensible next step: Graph algorithms that tell you about the similarity, connectivity, and importance of different graph elements can give you useful predictions about the future state of your model, while graph data science can expose these hidden insights.

Connect with the Graph Community

You aren't alone in wanting to use graphs. Organizations, big and small, are adopting graph technology, and this community is rapidly growing. Connect with your graph community and get involved. You can learn from others and realize the value much more quickly.

TIP

Visit the Neo4j community website at `community.neo4j.com` to start making connections.

(graphs)-[:ARE]->(everywhere)

The Native Graph Database for Today's Connected Applications

> Neo4j is a native graph database, purpose-built to leverage data relationships and enable richer, more intelligent applications. Powered by a native graph storage and processing engine, Neo4j delivers an intuitive, flexible and secure database for unique, actionable insights.

> Experience Neo4j in a click.
> No download required.

> **Try Neo4j Today**
> r.neo4j.com/sandbox.

Get rapidly up to speed with graph databases

Graph Databases For Dummies, Neo4j Special Edition, is a good place to start your journey with graph databases. This book assumes no previous experience with graph databases and walks you through modeling, querying, and importing graph data, all the way through to your first production system. For enthusiasts, this book may not be the only book you ever need on graph databases, but it's a great place to start.

Inside...

- Get introduced to graph databases
- Build rich graph data models
- Import graph data into your graph
- See how to query your graph
- Use graphs in production
- Get tips for creating successful graphs

neo4j

Dr. Jim Webber is Chief Scientist at Neo4j, where his research work focuses on transaction processing and fault-tolerance for graph databases. **Rik Van Bruggen** is an avid writer, blogger, and podcaster in the Neo4j community and passionate about bringing the value of graphs to Neo4j customers.

ISBN: 978-1-119-74602-7
Not For Resale

9 781119 746027

for dummies®
A Wiley Brand