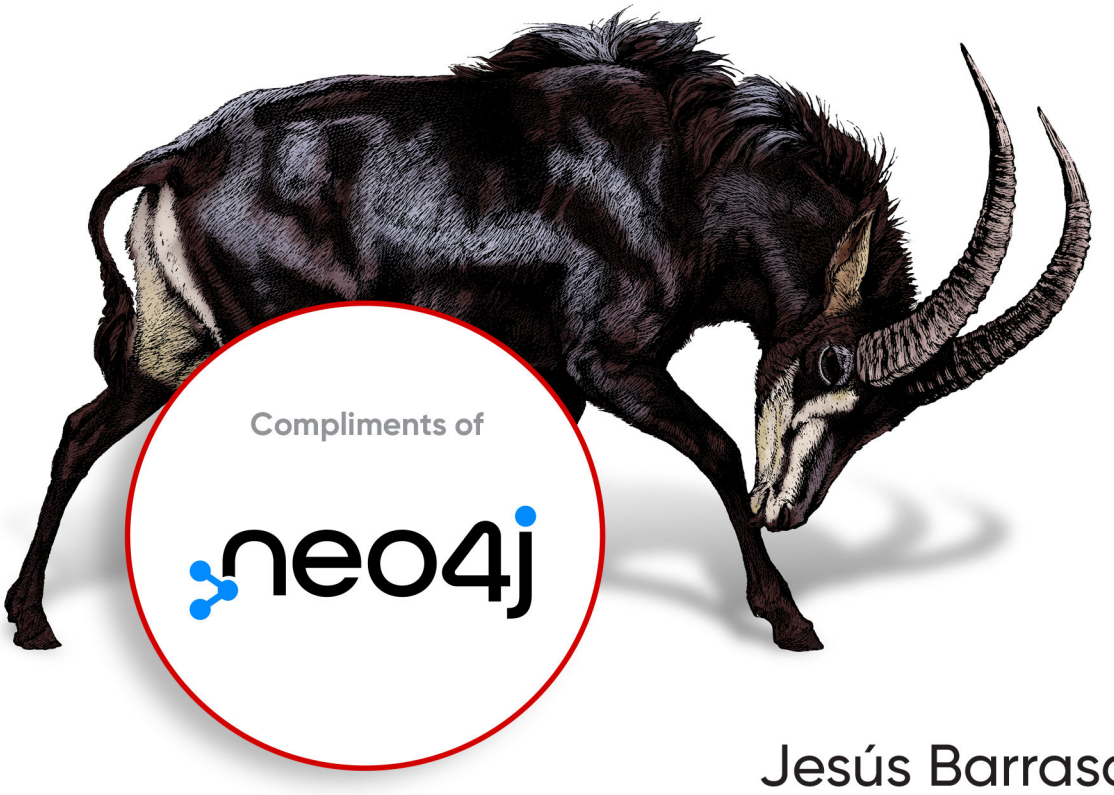
O'REILLY®

Building Knowledge Graphs

A Practitioner's Guide



Jesús Barrasa
& Jim Webber

O'REILLY®

Building Knowledge Graphs

Incredibly useful, knowledge graphs help organizations keep track of medical research, cybersecurity threat intelligence, GDPR compliance, web user engagement, and much more. They do so by storing interlinked descriptions of entities—objects, events, situations, or abstract concepts—and encoding the underlying information. How do you create a knowledge graph? And how do you move it from theory into production?

Using hands-on examples, this practical book shows data scientists and data engineers how to build their own knowledge graphs. Authors Jesús Barrasa and Jim Webber from Neo4j illustrate common patterns for building knowledge graphs that solve many of today's pressing knowledge management problems. You'll quickly discover how these graphs become increasingly useful as you add data and augment them with algorithms and machine learning.

- **Learn knowledge graph organizing principles**
- **Explore graph databases and knowledge graphs**
- **Import structured and unstructured data**
- **Build integration-and-search knowledge graphs**
- **Discover pattern detection knowledge graphs**
- **Use natural language knowledge graphs and chatbots**

Dr. Jesús Barrasa, an expert in semantic technologies and graph databases, is head of the solutions architecture team in EMEA at Neo4j and leads the development of neosemantics (a Neo4j plugin for RDF). He cowrote *Knowledge Graphs* (O'Reilly) and is cohost of the *Going Meta* live webcast.

Dr. Jim Webber is chief scientist at Neo4j, where he works on fault-tolerant graph databases. He coauthored *Graph Databases for Dummies* (Wiley) and *Graph Databases* and *Knowledge Graphs*, both for O'Reilly. He's also a visiting professor at Newcastle University, UK.

"An expert guide for engineers building or starting to work with knowledge graphs!"

—**Luanne Misquitta**
CTO, GraphAware

"This must-have book is the missing manual for working with graph data. It's a fun and incredibly useful guide. I highly recommend it."

—**Pete Aven**
Director/Solution Architect, MariaDB

"This book is outstanding for both beginners embarking on their knowledge graph journey and experienced graph enthusiasts seeking to elevate their skills and understanding."

—**Tareq Abedrabbo**
Data CTO, Mesh AI

MACHINE LEARNING

Twitter: @oreillymedia
linkedin.com/company/oreilly-media
youtube.com/oreillymedia

ISBN: 978-1-098-12711-4

Building Knowledge Graphs

A Practitioner's Guide

Jesús Barrasa and Jim Webber

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Building Knowledge Graphs
by Jesús Barrasa and Jim Webber

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*https://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Nicole Butterfield
Development Editor: Corbin Collins
Production Editor: Jonathon Owen
Copyeditor: Shannon Turlington
Proofreader: Piper Editorial Consulting, LLC

Indexer: nSight, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

June 2023: First Edition

Revision History for the First Edition
2023-06-21: First Release

This work is part of a collaboration between O'Reilly and Neo4j. See our statement of editorial independence.

978-1-098-12711-4

[LSI]

Table of Contents

Preface

Graph databases and graph data science have reached a significant level of adoption. They have been extensively used for a range of discrete use cases like logistics, recommendations, and fraud detection. But there is a bigger emerging trend to arrange data in a deliberate manner that enables insight at scale across functional silos. The technology underpinning this trend is know as a knowledge graph.

The forces behind the trend are clear: organizations are no longer suffering from data scarcity. In fact, in an era when big data seems to be a solved problem (at least from a storage point of view), many organizations are practically drowning in data. Industry anecdotes of many thousands of relational tables per day being ingested into a data lake abound, but with an abundance of data there comes the unexpected challenge of what to with it. This is where knowledge graphs help.

A *knowledge graph* is a purposeful arrangement of data such that information is put in context and insight is readily available. Individual records are placed in an associative network of relationships that provide rich semantic connectivity and context. That network of relationships—a graph—is an incredibly intuitive way of representing useful knowledge. Data that might have originally existed to serve a fraud-detection use case can be repurposed seamlessly within the knowledge graph to provide data for recommending financial products (or vice versa). And from there it is straightforward to connect other data to support other vertical use cases or horizontal analysis.

Importantly, while the term *knowledge graph* has only come to prominence in industry relatively recently, knowledge graph systems have been in existence for some time. This book tries to distill our experience of understanding knowledge graphs deployed in real systems by organizations around the world. It addresses the emerging trend of building systems on knowledge graphs as well as thinking about knowledge graphs as a general-purpose underlay for the enterprise. It also addresses the contemporary intersection of knowledge graphs and artificial intelligence (AI), where knowledge

graphs provide high-quality features for machine learning, are themselves enriched by AI, and can even tame the hallucinatory nature of large language models (LLMs).

While this is our most in-depth and unapologetically technical book on the topic, this isn't our first time writing about knowledge graphs. In fact, in the book *Knowledge Graphs: Data in Context for Responsive Businesses* (O'Reilly), we highlighted the business benefits of knowledge graph adoption aimed at an audience of CIOs and CDOs. But this book goes much deeper technically, and it contains enough implementation detail for a range of tools, patterns, and practices so that you can build your own knowledge graphs with confidence. We hope that what you learn here will propel you to your first successful knowledge graph project and beyond!

Who This Book Is For

This is a technical book, aimed at computing professionals—typically software engineers, system architects, and techincal managers—who want to understand both the potential of knowledge graphs and how to go about implementing them. While no prior experience with knowledge graphs (or graphs in the general sense) is required, readers will get the most of the book if they are modestly comfortable with database concepts like queries and have some programming experience.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-889-8969 (in the United States or Canada)
707-829-7019 (international or local)
707-829-0104 (fax)
support@oreilly.com
https://www.oreilly.com/about/contact.html

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/building-knowledge-graphs*.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*

Follow us on Twitter: *https://twitter.com/oreillymedia*

Watch us on YouTube: *https://www.youtube.com/oreillymedia*

Acknowledgments

We are grateful to all those who helped us while we were writing this book: the staff at O'Reilly, especially Corbin Collins, whose efforts kept our prose properly Americanized, and our Neo4j colleagues, especially Maya Natarajan and Deb Cameron, who worked with us on early drafts.

We are grateful to Dr. Nicola Vitucci, who provided deep technical expertise and patient guidance on our prose.

We would also like to thank Nigel Small, who provided detailed technical feedback on idiomatic Python graph data science.

Finally, we would like to thank the technical reviewers from O'Reilly, Max de Marzi and Janit Anjaria, for their enthusiastic and detailed feedback.

CHAPTER 1
Introducing Knowledge Graphs

We're overwhelmed by data. It's everywhere and being collected at a fantastic rate and stored at substantial cost. But we're not necessarily getting value from that data, though there is significant value in it, if only we could understand it.

All is not lost. Over the last decade, a new category of technology based on graphs has moved from obscurity to prominence. Graphs have come to underpin everything from consumer-facing systems like navigation and social networks to critical infrastructure like supply chains and power grids.

These important graph use cases have reached a common conclusion: applying knowledge in context is the most powerful tool that most businesses have. A set of patterns and practices called *knowledge graphs* has been emerging to help understand data in context, where the context is represented as a graph of connected data items. With knowledge graphs, there is hope that you can distill business value from data. This book is an attempt to show how it can be done.

Knowledge graphs are useful because they provide contextualized understanding of data. Context derives from the layer of metadata (graph topology and other features) that provides rules for structure and interpretation. This book shows how the connected context provided by knowledge graphs enables you to extract greater value from existing data, drive automation and process optimization, improve predictions, and support an agile response to changing business environments.

We wrote this book for technology professionals who are interested in building and operating knowledge graphs within their businesses. In a way, it's a sequel to or expansion of O'Reilly's *Knowledge Graphs: Data in Context for Responsive Businesses*, which we had the pleasure of writing in 2021. That report was intended for chief information officers (CIOs) and chief data officers (CDOs), helping them to

understand the benefits of knowledge graphs. This time we're aiming at data and software professionals who build sophisticated information systems.

The book is arranged in two parts. The first part deals with graph fundamentals, including graph databases, query languages, data wrangling, and graph data science. It teaches technical practitioners the fundamental tooling needed to understand the second part of this book, which tackles significant knowledge graph use cases and how to implement them, complete with code examples and system architectures.

For data and software professionals, this book provides an on-ramp to the world of knowledge graphs and a language for discussing their implementation with your peers and management. It also gives deep examples for how to build and use knowledge graphs—from graph basics all the way to graph machine learning.

For CIOs or CDOs, this book may still be useful since it provides a good overview of knowledge graphs and how they are delivered. You can skim the earlier chapters and code examples if that's not your thing, but you'll still be able to understand what your practitioner colleagues are doing and why.

This chapter explains the background and motivation for knowledge graphs. Here we'll introduce the notions of graphs and graph data and start to show how to build smarter systems with knowledge graphs.

What Are Graphs?

Knowledge graphs are a type of graph, so it's important to have a basic understanding of graphs. Graphs are simple structures that use nodes (or vertices) connected by relationships (or edges) to create high-fidelity models of a domain. To avoid any confusion, the graphs in this book have nothing to do with visualizing data as histograms or plotting a function, which are called *charts*, as shown in Figure 1-1.

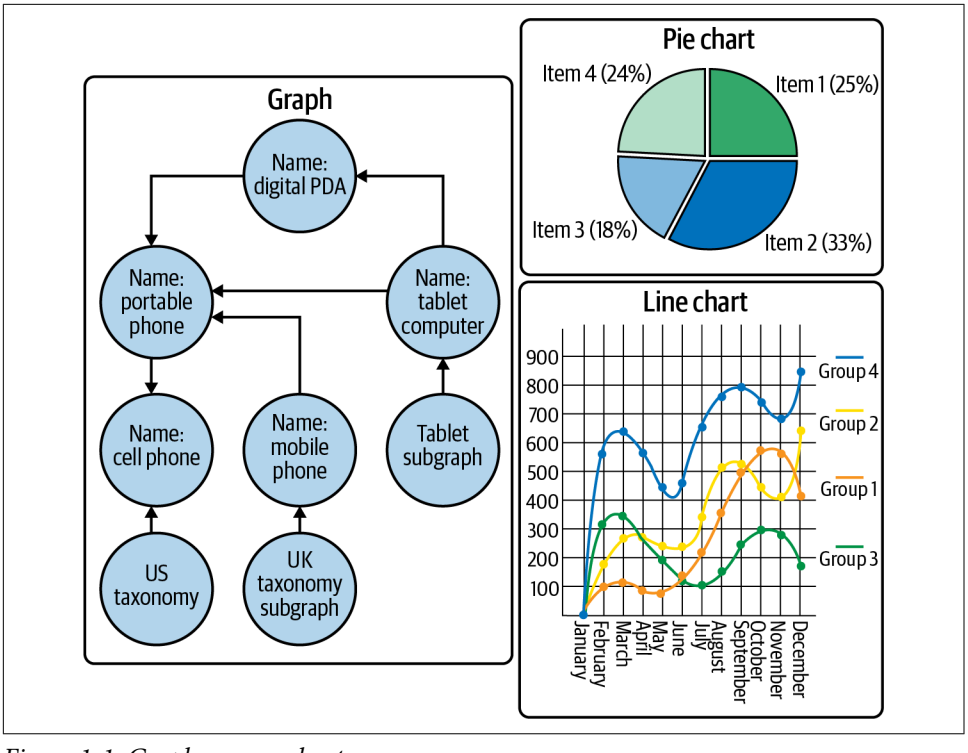


Figure 1-1. Graphs versus charts

The graphs in this book are sometimes referred to as *networks*. They are a simple but powerful way of showing how things connect.

Graphs are not new. In fact, *graph theory* was invented by Swiss mathematician Leonhard Euler in the 18th century to help compute the minimum distance that the emperor of Prussia had to walk to see the town of Königsberg (modern-day Kaliningrad) by ensuring that each of its seven bridges was crossed only once, as shown in Figure 1-2.

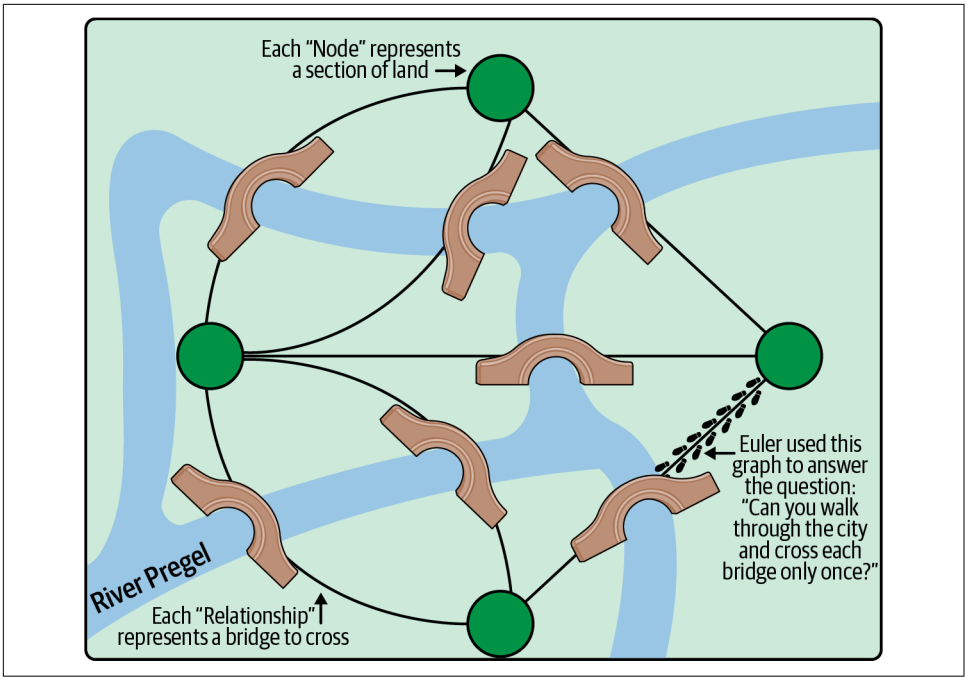


Figure 1-2. A graphical representation of Königsberg and its seven bridges across the Pregel River

Euler's insight was that the problem shown in Figure 1-2 could be reduced to a logical form, stripping out all the noise of the real world and concentrating solely on how things are connected. He was able to demonstrate that the problem didn't need to involve bridges, islands, or emperors. He proved that in fact the physical geography of Königsberg was completely *irrelevant*.

To us, Euler's approach is similar to modern software development. The inherent noise of the real world is stripped away so that a more valuable logical representation—the software—remains. For software professionals, this feels comfortingly familiar.

You can use the superimposed graph in Figure 1-2 to figure out the shortest route for walking around Königsberg without having to put on your walking boots and try it for real. In fact, Euler proved that the emperor could *not* walk the whole town crossing each bridge only once, since there would have needed to be (at least) one island (node) with an even number of connecting bridges (relationships) from which the emperor could start his walk. No such island existed in Königsberg, so no such route (path) is possible.

Building on Euler's work, mathematicians have studied various graph models, all variations on the theme of nodes connected by relationships. Some models allow relationships to be *directed*, where they have an explicit start and end node, while some have *undirected* relationships connecting nodes. Some models, like *hypergraphs*, allow relationships to connect multiple nodes.

In theory, there's no single best graph model to choose (though you can usually transform from one model to another). But there *are* better or worse models in practice, especially for building computer systems. In this book we've chosen the *labeled property graph model* as the foundation. It's a popular model that is simple for software and data professionals to understand. It is expressive enough to represent even the most complicated domains and is information rich (unlike the graphs beloved of mathematicians).

The Property Graph Model

The property graph model is the most popular model for modern graph databases. Correspondingly, it's a common basis for creating knowledge graphs. It consists of the following elements:

Nodes representing entities in the domain

- Nodes can contain zero or more *properties*, which are key-value pairs representing entity data such as price or date of birth.
- Nodes can have zero or more *labels*, which declare the node's purpose in the graph, such as representing a `Customer` or a `Product`.

Relationships representing how entities interrelate

- Relationships have a *type*, such as `BOUGHT`, `FOLLOWS`, or `LIKES`.
- Relationships have a *direction*, going *from* one node *to* another (or back to the same node).
- Relationships can contain zero or more properties, which are key-value pairs representing some characteristic of the link, such as a timestamp or distance.
- Relationships never dangle—there are always a start node and an end node (which can be the same node).

You can use these *primitives* (nodes, labels, relationships, and properties) along with rules to assemble sophisticated, high-fidelity graph data models with relative ease.

Figure 1-3 shows a small social graph, but compared to the example in Figure 1-2, this graph holds much more information.

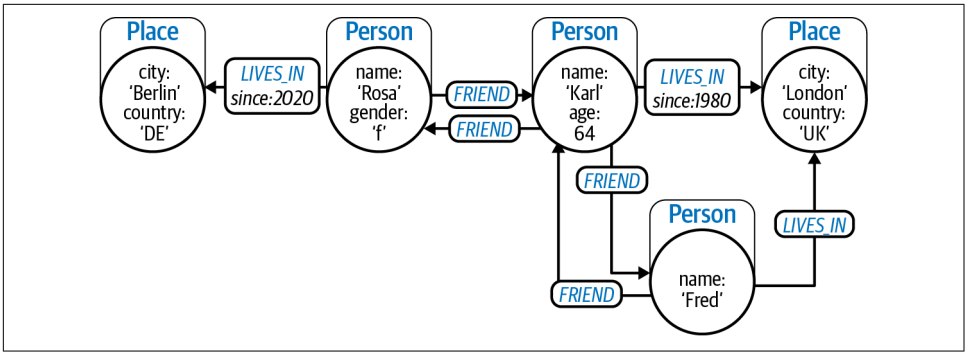


Figure 1-3. A graph representing people, their friendships, and their locations

In Figure 1-3 each node has a label that represents its role in the graph. Some nodes are labeled `Person` and some are labeled `Place`, representing people and places respectively. Stored inside those nodes are properties. For example, one node has `name:'Rosa'` and `gender:'f'` that you can interpret as being a female person called Rosa. Note that the `Karl` and `Fred` nodes have slightly different properties on them, which is a perfectly fine tool as it accommodates the messiness of the real world.

The property graph model does not enforce any schemas based on labels or relationship types. It's deliberately intended to be flexible and accommodating to help develop high-fidelity models. If you really need to ensure that nodes with certain labels have certain properties, you can apply *constraints* to the label to ensure those properties exist, are unique, and so on. You can view this not as schema or schemaless, but *schema-ish*. The idea is to use schema-like constraints just where you need them rather than eagerly locking down the whole data model. Those schema-like constraints, like everything else in a graph data model, can change over time. Real-world data is often uneven and incomplete, so your knowledge graphs should reflect this reality.

Between the nodes in Figure 1-3 you see relationships. The relationships are richer than in Figure 1-2 since they have a type, a direction, and optional properties. Relationships cannot "dangle"; they always have a start node and an end node, even if the start and end nodes are the same.

For instance, there is a `Person` node with the property `name:'Rosa'` that has an outgoing `LIVES_IN` relationship with the property `since: 2020` to the `Place` node with the property `city:'Berlin'`. You can read this as "Rosa has lived in Berlin since 2020," based on the direction of the relationships. You can also see that `Fred` is a

`FRIEND` of `Karl` and that `Karl` is a `FRIEND` of `Fred`. `Rosa` and `Karl` are also friends, but `Rosa` and `Fred` are not.

In the property graph model, there are no limits on the number of nodes or the number or type of relationships that interconnect them. Some nodes are densely connected while others are sparsely connected. All that matters is that the model matches the problem domain. Similarly, some nodes have lots of properties, while some have few or none. Some relationships have lots of properties, but many have none at all. This is all perfectly normal for knowledge graphs.

It's easy to see how the graph in Figure 1-3 can answer questions about friendships and who lives where. Extending the model to include other data like hobbies, publications, or jobs is also straightforward: just keep adding nodes and relationships to match your problem domain. Creating large, complex graphs with millions or billions of connections isn't a problem for modern graph databases and graph-processing software, so building even very large knowledge graphs is achievable.

Graph data models can comfortably represent complex networks of relationships in a way that is both human readable *and* machine friendly. Graphs might seem very technical at first, but they are created from very simple primitives, making them very accessible in practice. In fact the combination of a simple data model and the ease of algorithmic processing to discover connections, patterns, and features is what has made graphs so popular. It's a powerful combination you will also exploit in your knowledge graphs.

The Motivation for Knowledge Graphs

Interest in knowledge graphs has exploded, with a myriad of research papers, solutions, analyst reports, groups, and conferences on this topic. Knowledge graphs have become so popular partly because graph technology has accelerated in recent years but also because there is strong demand to make sense of data.

External factors have undoubtedly accelerated knowledge graphs to greater prominence. Stresses from the COVID-19 pandemic and fallout from geopolitics have strained some organizations to the point of breaking. Decision making has never needed to be more rapid. At the same time, businesses are hampered by the lack of timely and accurate insight.

Now businesses are reconfiguring their operations and processes to flex rapidly. As historical knowledge ages faster and is invalidated by market dynamics, many organizations need new ways of capturing, analyzing, and learning from data. Businesses need rapid insights and recommendations, from customer experience and patient outcomes to product innovation, fraud detection, and automation. They need contextualized data to generate knowledge.

Knowledge Graphs: A Definition

Now you know a little about graphs and the motivation for using knowledge graphs. But clearly not all graphs are knowledge graphs (despite the hype). *Knowledge graphs* are a specific type of graph with an emphasis on contextual understanding. Knowledge graphs are interlinked sets of facts that describe real-world entities, events, or things and their interrelations in a human- and machine-understandable format.

Critically, knowledge graphs must have an *organizing principle* so that a user (or a computer system) can *reason* about the underlying data. The organizing principle provides an additional layer of structure that adds context to support knowledge discovery. The organizing principle makes the data itself smarter. This idea runs contrary to the norm where intelligence resides in applications and data is dumb, just something to be mined and refined. Having smarter data both simplifies systems and encourages broad reuse.

Where's the Data?

A knowledge graph can be a self-contained unit which exists in a graph database, or it can involve several coordinated graph stores forming a federation of graphs. Or a knowledge graph can be built on top of a data lake to bring structure and knowledge to undifferentiated bulk storage. A knowledge graph could also be a logical layer providing structure and insight over multiple data sources of different kinds so that data consumers get a holistic, curated view of the data.

In principle, knowledge graphs are agnostic about the physical storage of the underlying data. They can support different architectural approaches, from virtualized ones where the knowledge graph is a smart index over externally stored data to the fully materialized ones where data is fully hosted in a graph platform, and any hybrid approach between the two.

Summary

Organizing principles, reasoning, and knowledge discovery might seem complicated at first. But in reality, you can think of knowledge graphs as a rich index over data that provides curation, much like a skilled librarian recommending books and journals to a researcher.

From here on things get a little more technical. In Chapter 2 you'll learn how we can extend the definition of an organizing principle to act as a contract between a knowledge graph and its consuming users and systems. You'll also learn about several different options for creating organizing principles.

CHAPTER 2

Organizing Principles for Building Knowledge Graphs

Graphs are common in modern computer systems. They're a pleasant and flexible data model for supporting interactive queries, real-time analytics, and data science. But what transforms a graph into a knowledge graph is the application of an *organizing principle* that helps people and software to understand it. Historically, this has loftily been called *semantics*, but you can think of it as *making the data smarter*.

Knowledge graphs are the result of decades of research in semantic computing. With modern graph technology, you can easily apply the fruit of all of that research to contemporary problems.

This chapter introduces common organizing principles for knowledge graphs. Once you've finished reading this chapter, you will be able to choose from different organizing principles that best suit the problems you want to solve.

Organizing Principles of a Knowledge Graph

The notion that knowledge graphs help make data smarter is appealing. Rather than having to repeatedly encode smart behavior into applications, it is encoded once, directly into the data. Smarter data enables knowledge reuse and reduces duplication and discrepancies.

There are several different approaches to organizing data in a graph, each with its own benefits and quirks. You're free to choose the ones that fit your problem, and you're free to combine approaches as well.

Starting with a basic (but useful) graph, we'll show how to add successive layers of organization, demonstrating how knowledge graphs can be used to solve increasingly sophisticated problems.

Plain Old Graphs

The term *graph* (as opposed to knowledge graph) is used when we're thinking about those graphs that don't have an explicit organizing principle. But as a graph enthusiast, you know that regular graphs are also very useful. The difference between a plain old graph and a knowledge graph is that the interpretation of the information in plain old graphs is encoded into the systems that use the graph, rather than being part of the data itself. In other words, the organizing principle is "hidden" in the logic of the queries and programs that consume the data in the graph.

As a familiar example, think about the sales data for an online store. Sales data is typically large and dynamic, combining customer shopping information with a product catalog, including product descriptions, categories, and manufacturers. A small fragment of a sales and product catalog graph is shown in Figure 2-1.

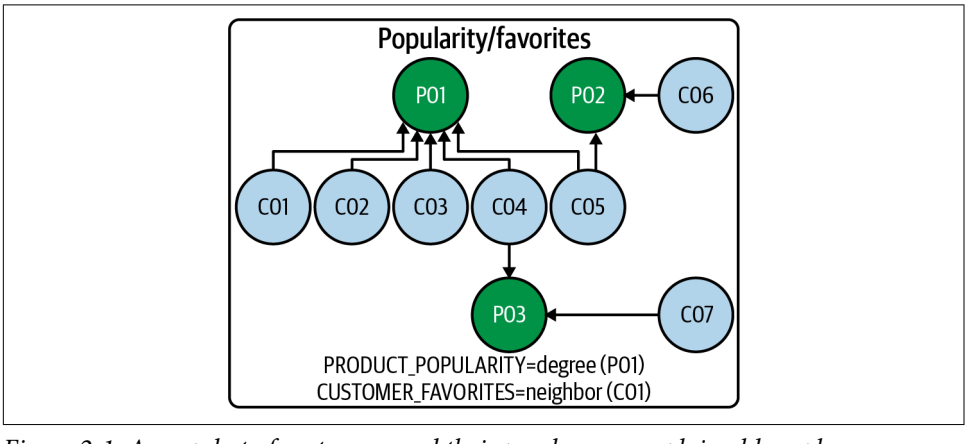


Figure 2-1. A snapshot of customers and their purchases as a plain old graph

You may not find this graph intuitive at first. But if an engineer has encoded the knowledge that the `P` nodes represent products and `C` nodes represent customers, and that connections between nodes represent purchases, a program will be able to answer straightforward questions like "What products did this customer buy?" Such a program will also be able to answer those questions in reverse, such as "Which customers bought this product?" The ability to answer questions in both directions is inherently valuable to a retailer.

Such a program would also be able to compute a product's popularity by counting the number of incoming relationships (which is called *calculating the indegree* of the node). In this way, the program could identify products that are quite popular as well as others that are less so.

No doubt there is a lot of value in the graph. But what would happen if a data scientist with no previous knowledge of the domain wanted to run basket analysis to find

out which products are purchased together? Someone would have to explain how to interpret the data in the graph because there is no organizing principle to help the data scientist to make sense of it beyond nodes connected by relationships. The knowledge of how to interpret the data in the graph is encoded into the software systems, not into the data.

There's also a genuine problem if the person who created the graph leaves the organization. The user now has to reverse engineer the code in the algorithms to interpret the graph. In mature businesses, this happens all too often.

Of course, once you understand the meaning of the data in the graph, you could continue to build new software to process knowledge out of that graph. But a better solution is to make the data in the graph smarter by applying an organizing principle. This surfaces the latent and implicit knowledge in a graph, turning it into a knowledge graph.

Reading the Manual Versus Not Reading the Manual

It might seem like a lot of unnecessary effort to make data self-explanatory. In your lives, you're surrounded by examples of things that are not self-explanatory. For example, you might get a new Ncooking gadget and have to read the instruction manual to use it.

But you also have devices where there is no need for an instruction manual. The organizing principle is so good that you can be immediately productive. For example, the touch interfaces of modern phones and tablets have such good usability that instruction manuals aren't needed. In fact, toddlers can use them.

Data can be seen in the same way. An instruction manual might work, but it's better to have the data organized in a way that makes it easy and even pleasurable to work with. Even if creating such a scheme takes effort, it will be repaid many times over by a useful production system.

Richer Graph Models

Compared to the austere lines-and-circles graphs beloved of mathematicians, there is already a richer graph model—the property graph model.

The property graph model is far more organized than the plain old graphs. It supports labeled nodes, type and direction for relationships, and properties (key-value pairs) on both nodes and relationships. Any software that understands the property graph model can process it in accordance with that simple organizing principle.

Figure 2-2 shows an enriched view of the sales and product catalog graph, including labels, properties, and named relationships.

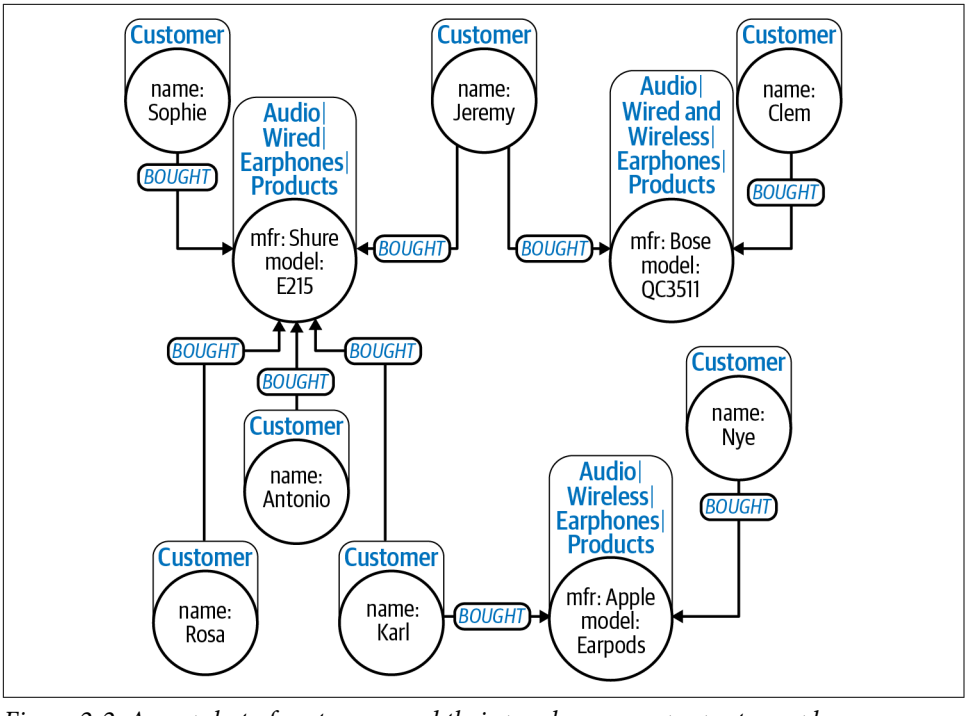


Figure 2-2. A snapshot of customers and their purchases as a property graph

A property graph gives human and software agents a set of essential clues about the information it contains: node labels, relationship types and direction, and the types and names of property data stored on them. It is even possible to get a formal description of the shape of the graph (its schema) via introspection. Effectively, this organizing principle makes a graph self-describing (to a certain extent) and is a clear first step toward making data smarter. Just by using node labels, software can extract all similar types of entities from a graph. In the case of Figure 2-2, software can easily extract all the customers, for example. A data scientist will have a much better (and more productive) time working with this information-rich graph.

Importantly, some processing can be done without knowledge of the domain, just by using the features of the property graph model. A common example is visualization. Figure 2-3 shows how nodes with the same labels will be displayed with similar visual style in two popular visualization tools, Linkurious and Neo4j Bloom. With just an understanding of the organizing principle, these tools render the data in a helpful, visual manner. Moreover, Bloom uses the metadata to provide intuitive Google-query-style exploration atop the graph.

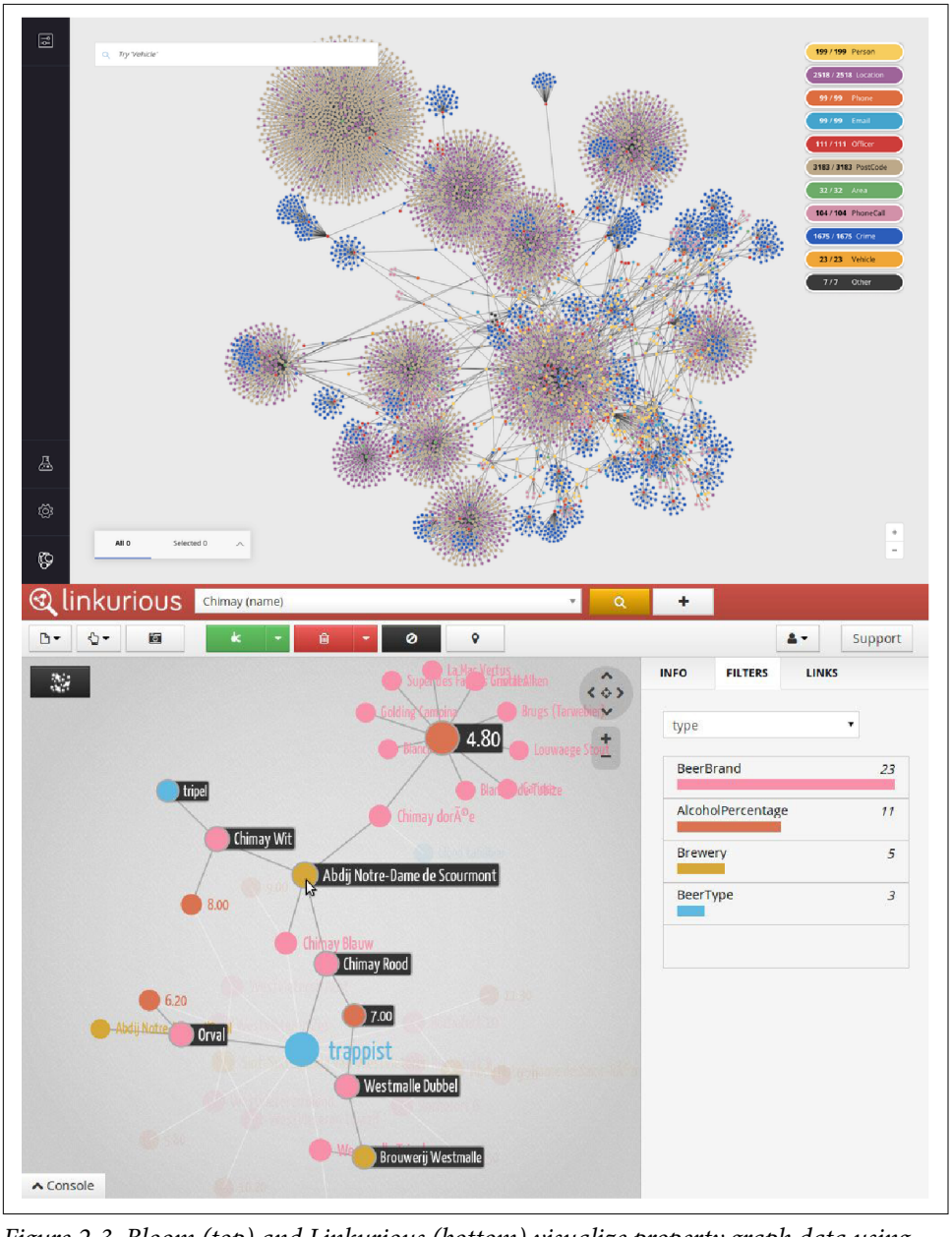


Figure 2-3. Bloom (top) and Linkurious (bottom) visualize property graph data using the model's organizing principles

To a consumer application—like visualization tools—the organizing principle is a contract. The contract tells the tool that it can expect to find labeled nodes, connected by directed and typed relationships, with properties on both if desired. As long as the graph and the tools both honor this contract, the system will work, even in the absence of detailed domain knowledge.

Organizing Principles Are Contracts

The organizing principle provides a contract between the graph and its users. But this contract doesn't only apply to humans—it also applies to software.

With the property graph model, consuming software can expect to find labeled nodes and directed, labeled relationships in the data. The property graph model is a *contract*, and software tools that uphold this contract can process, visualize, transform, and transmit the data in a manner that is consistent with its intent.

Although it's commonplace and powerful, the property graph model is a relatively low-level organizing principle. The property graph model becomes even more useful if it's combined with other higher-order organizing principles like the taxonomies and ontologies discussed later in this chapter.

In Figure 2-2 we've used the the property graph model to showcase various product features. A competent user can extract all *wireless headphones* from the data as easily as they could extract all *audio* products. With only slightly more effort, aggregate sales data per product or per product category can also be computed. But the way the data is set up does not provide much scope for reasoning: labels don't give enough information to know that, for example, one product can be substituted for another. For that you need a stronger organizing principle.

Knowledge Graphs Using Taxonomies for Hierarchy

In Figure 2-2, you saw that the associative connection between customers and the products they've bought is explicit in the graph in the form of a relationship. Creating categories of nodes using labels is also clearly useful, but the associativity *between* labels was missing.

The Labeled Property Graph Model

In the labeled property graph model, labels are akin to tags which describe the role of a node in the graph. There is no associativity between labels, so you can't for example infer that an apple is a specialization of a fruit. This absence of associativity between labels is considered to be a feature, not a bug. Type systems and polymorphism are an application-level concern, while graph databases deal with data and structure only.

Accordingly, the designers of the labeled property graph model separated data and type system. The database owns data (and its structure of course), while the applications that use the database may have their own rich type systems for their own purposes.

Keep in mind that labels are not types with inheritance characteristics. They are indicators of a node's role (or roles) in the graph. While labels might be mapped to types in an application, that's a design decision left for the implementer of a system that consumes the knowledge graph.

Labels don't tell us, for example, that one category is broader than another one or that certain products are compatible with others or are even possible substitutions based on the categories to which they belong. You cannot tell from labels alone that `giraffe` is a specialization of `mammal` or that `apple` could be a reasonable substitution for `banana` when you are interested in `fruit`. This functionality *is* present in some other graph models, notably in some implementations of the Semantic Web technology stack, but the implementation complexity is high, and reasoning over the model is often computationally slow, even intractable.

At a business level, lacking data about categories and substitutions means lost sales opportunities. It is clear that better product catalogs mean better shopping experiences for the customer and more revenue for the retailer. An expressive way of categorizing products is therefore a worthwhile investment.

As savvy buyers, you know that headphones and earphones are both part of the more general category of personal audio. You also know that there are specific styles of wireless and wired headphones and earphones, and you know that some personal audio can be both wired and wireless. To meet customer expectations, the retailer needs to be able to use this information. It can also be useful to mix product information with historic customer preferences, stock levels, profit margins, and so on to provide an even richer set of data to guide customers' buying decisions and maximize value for the seller.

In the example of a product catalog, those questions aren't limited to buying specific headphones but also include higher-order questions like "What's good equipment for listening to classical music while I'm running?" These kinds of questions cannot be answered with algorithms alone. You need smarter data. A good place to start is to enrich the way products are classified with a higher-order organizing principle so that you can better reason about products' substitutability. That is, if the retailer is out of stock of a specific item, it might still win a sale if it is able to offer similar items.

To support *x is a kind of y* reasoning, you need a more hierarchical view of your data using a structure known as a *taxonomy*. A taxonomy is a classification scheme that organizes categories in a broader-narrower or generalization-specialization hierarchy. Items that share similar qualities are grouped into the same category, and the taxonomy organizes concepts by relating categories to one another. This kind of hierarchical organization places more specific things like products (which are numerous) at the bottom of the hierarchy while more general things like brands or product families (which are less numerous) are placed toward the top of the hierarchy.

The hierarchy is constructed with `Category` nodes connected by `SUBCATEGORY_OF` relationships, though on the framework or vocabulary used, these could also be called `NARROWER_THAN` or `SUBCLASS_OF`. Subsequently, products can be connected to the appropriate part of the taxonomy to classify them as ready for sale. This is shown in Figure 2-4, where the graph supports better customer service by offering alternative products in the same category *and* in nearby categories.

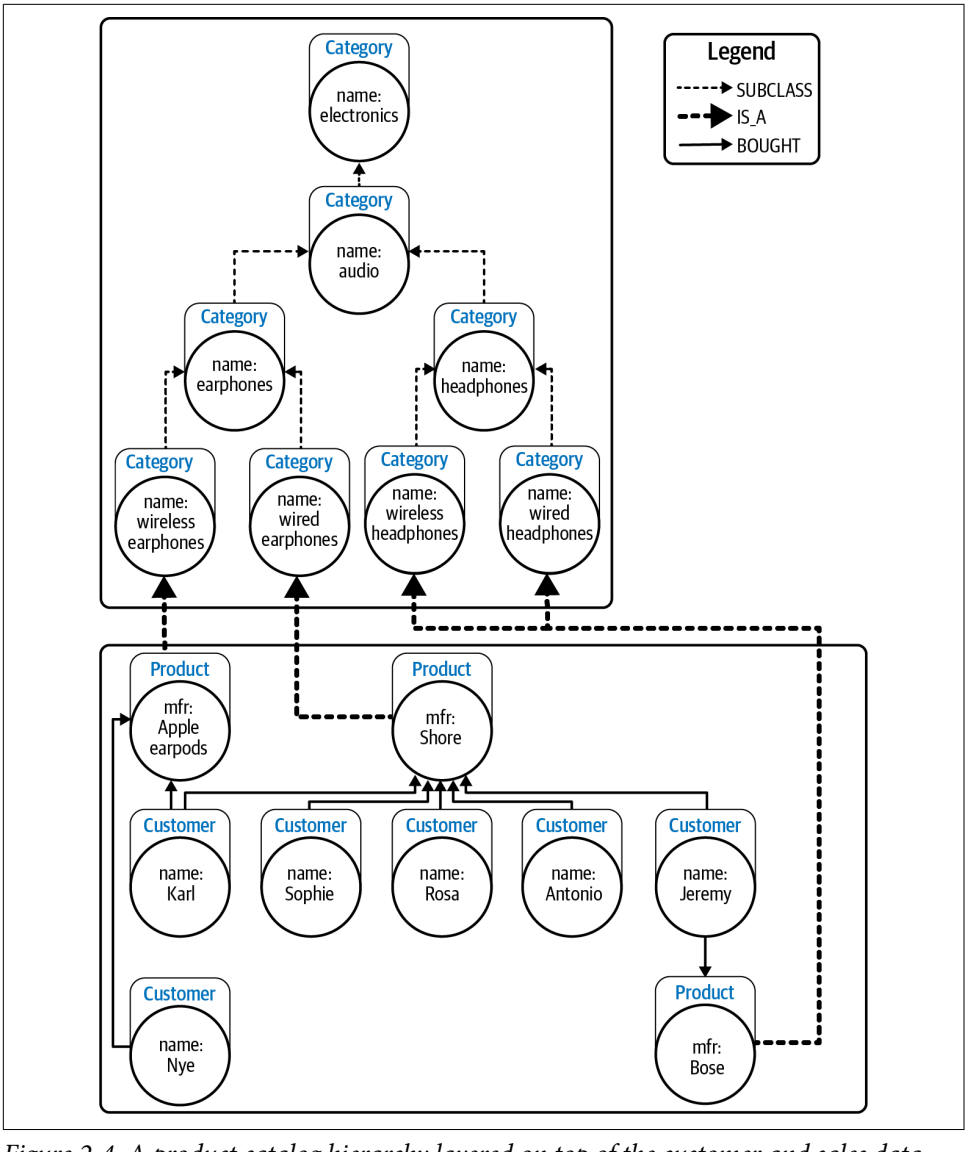


Figure 2-4. A product catalog hierarchy layered on top of the customer and sales data

The beauty of knowledge graphs is that you can choose to compose multiple organizational hierarchies simultaneously to provide even more insight. In Figure 2-5 you can see how different taxonomies can coexist in a single knowledge graph.

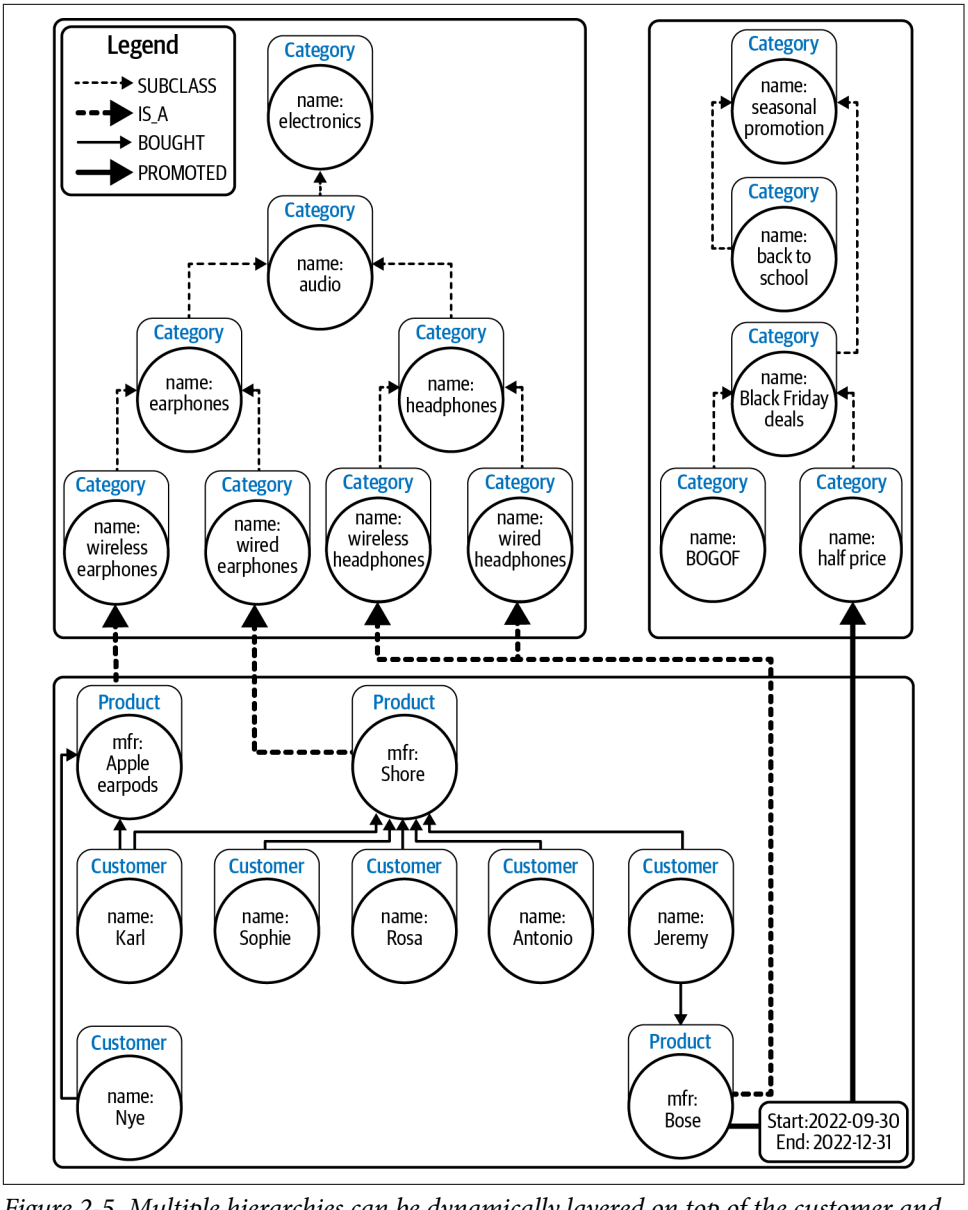


Figure 2-5. Multiple hierarchies can be dynamically layered on top of the customer and sales data

Currently, the Bose QC35II headphones are connected via a `PROMOTED` relationship to the `half price category` node, itself a subcategory of `Black Friday deals`, which is a seasonal promotion. They are also connected to both the `wired headphones` and `wireless headphones` categories and would be presented to customers browsing in those categories. At other times of the year, the same pair of headphones would be classified differently—for example, in a high-end audio category. This is possible because classification is dynamic in a knowledge graph. The new categories and their associativity as well as the linkage of products to the categories are just additional nodes and relationships. This gives you enormous flexibility since it's just data in a graph database.

In "Richer Graph Models" on page 11, you achieved a level of domain independence by using the property graph model. Now you can take things a step further by using a taxonomy. Any application that understands your taxonomy (simply the meaning of broader and narrower) can exploit the graph in a much richer way without any specific knowledge of the domain. For example, algorithms could use the taxonomy to compute semantic similarity between products by applying standard taxonomy metrics like *path similarity*, *Leacock-Chodorow*, or *Wu and Palmer*. It makes no difference whether your domain is audio equipment or fruit: you can process the smarter data in the taxonomy with these standard algorithms.

Using multiple categories makes the data more expressive and allows for sophisticated exploitation with low complexity. This modest taxonomical organization provides immediate benefits in terms of the knowledge that can be extracted.

It is pleasing and convenient that taxonomies map easily onto the labeled property graph model. But taxonomies are not your only option for organizing knowledge and making data smarter. There are still higher order organizing principles you can use.

Knowledge Graphs Using Ontologies for Multilevel Relationships

While taxonomies represent collections of topics with `SUBCATEGORY_OF` relationships between them, an *ontology* increases the level of sophistication. Like taxonomies, ontologies are classification schemes that describe the categories in a domain and the relationships between them. But ontologies are not restricted to hierarchical (broader-narrower) structures, so they can offer richer associativity.

Ontologies enable you to define more complex types of relationships between categories, such as `PART_OF`, `COMPATIBLE_WITH`, or `DEPENDS_ON`. They also allow you to define hierarchies of relationships and to characterize those relationships in more nuanced ways (transitive, symmetric, and so on). These concepts also map comfortably onto the labeled property graph model.

Following the guidance in an ontology, you can explore the categories in a domain not just vertically (hierarchically) but also horizontally, which allows you to address cross-cutting concerns. For example, you could reason that an iPhone 12 is a valid search result for a customer looking for a mobile phone because it is an iOS device. Using a regular taxonomy you can also express that iOS is a subcategory of mobile phone. But from the semantics of the `UPSELL` relationship defined in the ontology, you can further reason that an iPhone 12 Pro should be recommended to customers looking at an iPhone 12. Figure 2-6 shows how an ontology supports this upsell feature using `UPSELL` relationships to help guide a user of the product catalog toward better buying decisions.

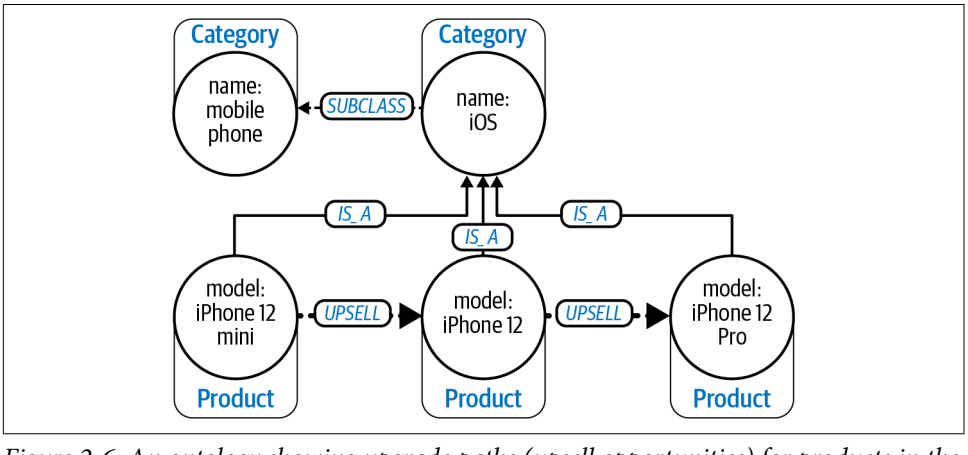


Figure 2-6. An ontology showing upgrade paths (upsell opportunities) for products in the catalog

At a still higher level, for graphs that span systems you can use an ontology to act as a semantic bridge. An ontology that defines cross-equivalence (linking between the same concepts in both systems) allows you to traverse the entirety of the business domain while mapping it back to a standard, well-understood vocabulary.

As the use cases for ontologies become more sophisticated, such as acting as contracts between systems, so does their complexity. However, ontologies can be built in a modular fashion to make them more composable. This allows you to get immediate value from them and manage their complexity. Figure 2-7 shows a sophisticated use of *layered* ontologies to bridge several underlying taxonomies/ontologies. Each layer in the graph can be queried independently, but when the layers are brought together, they provide an ability to reason *across* those domains.

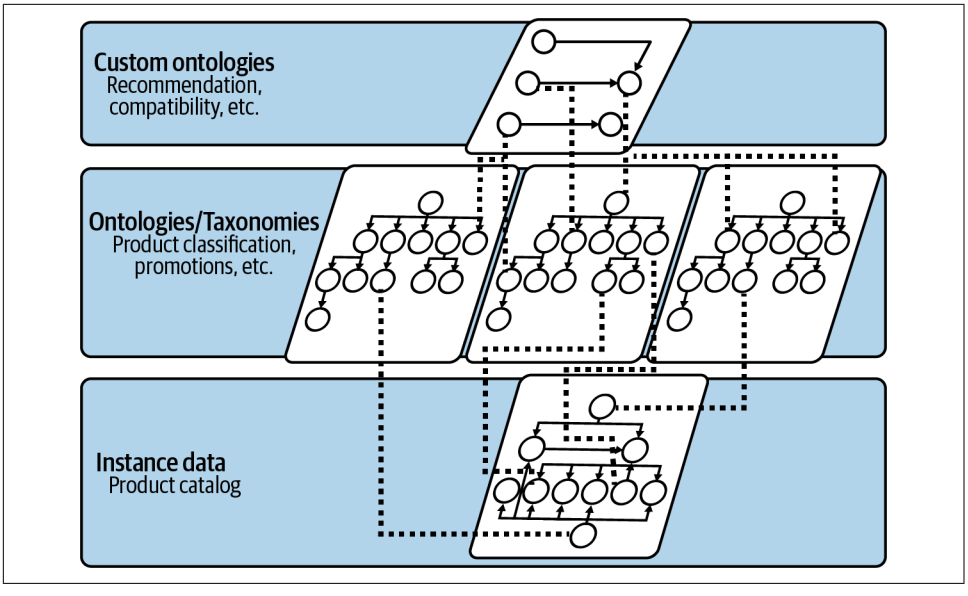


Figure 2-7. A sophisticated layering of ontologies and taxonomies over the same data set

In Figure 2-6, you were only able to reason about electronics that complement one another through a single ontology. But when you use an overarching cross-domain ontology, you are able to reason about more wide-ranging needs, including a bigger range of products, recommendations, special offers and promotions, and more. At a business level, this is a distinct shift from "tell me what gadgets complement each other" to "help me get the things I need (at the best price)."

Ontologies *make knowledge actionable*, enabling human or software agents to carry out sophisticated tasks. For example, if an online retailer ties its product hierarchy into stock control data through an ontological layer, it has a way of offering good alternatives to the user when the current item is out of stock or even to recommend products with better margins. All of this value comes at the modest cost of capturing how the business works in a (machine-readable) ontology.

Which Is the Best Organizing Principle for Your Knowledge Graph?

The type of organizing principle for a knowledge graph should always be driven by its intended use. There is little value in building rich and expressive features if there is no associated process or agent (human or software) that makes use of them. It is a very common mistake to aim for an overly ambitious metamodel up front. It will be

an expensive effort in both time and resources at a point where the value of the work is not yet well understood. It risks the details of the organizing principle being out of date by the time it is complete.

Don't Boil the Ocean

From time to time, you see efforts to try to fully understand and capture the whole language of the business. There's no doubt that such work is painstaking and time-consuming.

What's less certain is its value. Occasionally, you really do need to completely capture the language of a domain in detail to release value, but usually that's not the case.

It might provide comfort to some to create an all-encompassing organizing principle up front, but that will only capture the system as it exists at the moment the work was done. Usually, it's better to treat it as malleable (and versionable).

Not only does this incremental approach allow you to gain value earlier, but it also sets up practices that assume you will learn and adapt as the business domain evolves.

This lesson has been learned well in enterprise software development. Instead of building monolithic software applications that deliver every feature on day one, today Agile methods are often used to deliver software that is sufficient to solve a problem right away, with the understanding that improved software that solves more of the problem space will follow.

In general, when creating organizing principles for knowledge graphs you should embrace *just-enough semantics*. Introduce semantic metadata for the use cases you currently have and add more as your use cases demand. You should not build a complex ontology when a simpler taxonomy or even just a property graph is enough to deliver the current capabilities required. Trying to build the more complicated system right away is often overengineering.

Building just what you need for now plays well with iterative systems delivery practices too. Iterative construction of knowledge graphs helps to avoid the common trap of ontological perfectionism. This approach ensures knowledge graphs that are fit for purpose for the long haul, delivers value early, and reduces overall risk.

But the question remains whether you should use an existing ontology off the shelf or invest the time to create your own. In the next section, you'll learn about the benefits and trade-offs of both approaches.

Organizing Principles: Standards Versus Create Your Own

There are several widely used ontologies that serve particular domains. Here are a few examples:

- SNOMED CT for clinical documentation and reporting
- Library of Congress Classification (LCC), widely used in academia
- Financial Industry Business Ontology (FIBO) for finance and business
- Schema.org, a collaborative effort to create a shared ontology for the web
- Dublin Core Metadata Initiative (DCMI), which has many schemas used for describing web resources and more

If you work in a domain that uses a standard ontology, it's a good idea to consider adopting that model wholesale rather than writing one of your own. Adopting an existing ontology fosters interoperability since the consumers of your knowledge graph are also able to adopt standards for interoperability. Interoperability is increasingly important when the communicating parties are separate entities, such as participants in a supply chain. In some cases, it may even be mandatory to use a specific standard for things like regulatory reporting.

Using a standard ontology is a form of knowledge reuse. You are treading a path that others have trodden before and can benefit from their efforts. But if there is no standard for your domain, if existing standards are only a partial fit, or if you really do need fine-grained control over the organizing principle, then you will need to write your own from scratch or use a public standard as a starting point for evolution.

Creating Your Own Organizing Principle

If you are faced with the task of creating an organizing principle, there are a few approaches from which to choose. One is to use natural language like English to describe the semantics of your organizing principle. This approach has a low barrier to entry but has other disadvantages. Since the semantics are not machine readable, programmers must turn the specification into code, which tends to introduce discrepancies.

Another way is to formally define the organizing principle using one of the standard languages available. The most widely used standard languages are RDF Schema and Web Ontology Language (called OWL) for ontologies and Simple Knowledge Organization System (SKOS) for taxonomical classification schemes. Each of these languages allows for different expressivity levels: from the basic definition of categories and relationships to taxonomies and more sophisticated constructs like complex classes.

An advantage of using a standard ontology language to describe your organizing principle is that you get support from software. Writing ontologies by hand can be difficult and prone to error, but (visual) ontology editors can help avoid certain types of errors. Still, becoming fluent in one or more ontological description languages has a cost that must be weighed against its benefits.

It's not only during the creation of the organizing principle that you can benefit from standards-aware software, though. Once the organizing principle has been applied to your knowledge graph, software based on standards can run automated tasks on your data. For example, suppose you use OWL as the framework for the organizing principle of a knowledge graph. In that case, a program capable of running OWL-based reasoning would be able to—within computational limits—derive new facts from your data *without it having to understand the domain of that data*.

Automated reasoning is very appealing, but it is important to keep an objective view before jumping in. What kinds of things can you express with these standards? What kinds of inferencing can you run on them? And the most important question: *how do they align with your business needs?*

We've found that organizations that are successful with knowledge graphs tend to leverage standards to a certain extent while ensuring they can easily layer on additional meaning as needed to move fast and keep up with business changes. It seems that a mix of standards and adaptive customization is most aligned with the reality of modern business.

Essential Characteristics of a Knowledge Graph

It's not true that a specific graph technology is required for knowledge graphs, despite the historical association with the Semantic Web and RDF triplestores. Our preference is the property graph model. Property graphs offer a sensible way to develop and maintain knowledge graphs, and they have excellent tool support and a large, active community of practice.

Good knowledge graphs are flexible and easy to maintain. Avoid cumbersome technologies that bog you down. While you're working hard to keep them up to date, the business will have moved on.

A good knowledge graph must be performant. Software teams and users both work around pain points, and slow systems are painful. A fast, up-to-date knowledge graph empowers users, while a slow, hard-to-change one will fall into disuse.

Knowledge graphs are not static. People and systems *using* a knowledge graph *enrich* it. Do everything you can to encourage use, and your knowledge graph will flourish as interactions enrich it in a virtuous cycle.

Data Exchange and RDF

Resource Description Framework (RDF) is often presented as a key differentiator for knowledge graph implementations and has been conflated with the concept of knowledge graphs. However, RDF is a model for data exchange, not a guide to storage or querying. It describes how data is serialized as triples of subject, predicate, and object (roughly equivalent to start node, relationship, and end node).

The techniques described in this chapter are completely independent from implementation decisions. You certainly don't need an RDF triplestore to build knowledge graphs, while the market direction is strongly towards labeled property graphs.

In practice, RDF is a reasonable choice as a format for sharing data, but your data model shouldn't be dictated by your wire format.

Summary

This chapter has discussed how to organize data to extract knowledge from it. You now have a good definition of a knowledge graph and the organizing principles that allow you to derive meaning from it. You've seen several different organizing principles, from labeled property graphs to sophisticated ontologies, and you've given some thought to the trade-offs of each.

Soon, we'll turn to the applications of knowledge graphs to illustrate how businesses generate value from knowledge extracted from their smarter data. But to get there, you'll need to learn a little about graph technology, from simple things like loading data into a graph database all the way through to graph algorithms and machine learning. But first, in Chapter 3 we're going to learn the basics of graph databases and graph queries.

CHAPTER 3
Graph Databases

In our book *Knowledge Graphs: Data in Context for Responsive Businesses* (O'Reilly), we said that knowledge graphs predate modern graph data technology. In fact, the concept of knowledge graphs appeared alongside the Semantic Web, before the term *graph database* had been coined.

More recently, graph databases and graph processing have become a significant trend in contemporary business systems. We've observed a correlation between the renewed interest in knowledge graphs and the widespread emergence of graph databases and graph data science tools. In a sense, modern graph technology has put knowledge graphs within practical reach of many organizations, rather than being mostly confined to academia.

This chapter will introduce you to graph databases at a basic practitioner level. It shows you how to use a graph database and, in particular, how to use a query language to store and query knowledge graphs. It also shows you how graph databases work and why they are significantly more performant for knowledge graph workloads than other data technologies.

The most popular graph database in terms of its maturity and reach is Neo4j, and all of the technical examples in this and subsequent chapters are based on the Neo4j graph database and associated tooling.[1] Neo4j is freely available from *https://neo4j.com*. The easiest way to run Neo4j on your computer is via the Neo4j desktop app. If you'd prefer not to install Neo4j, then consider Neo4j Sandbox and Neo4j Aura cloud services. Neo4j Sandbox is a free learning tool to help you get started with graphs. Neo4j AuraDB is a graph database as a service. It has a free tier, which is intended for prototyping systems that will (eventually) move into production.

This is an important chapter, but it is not intended to be comprehensive. Instead, it gives you the sufficient exposure to be able to work through the solutions contained in the rest of this book. If you're interested and looking for more in-depth information, there are many good books on the topic such as *Graph Databases* by Ian Robinson, Jim Webber, and Emil Eifrem (O'Reilly).

The Cypher Query Language

For most knowledge graph developers (and even some users), the majority of interactions with the database will be via the Cypher query language. Cypher is a declarative, pattern-matching query language, originally created by Neo4j but now implemented by several other systems and currently under standardization by the International Organization for Standardization (ISO) as GQL (Graph Query Language, or "SQL for Graphs" informally) at the time of writing.

Cypher's primary design goal was to be humane and expressive. Its creators saw that people are quite comfortable drawing diagrams with labeled arrows and boxes in drawing tools like Microsoft Visio. Even more importantly, people are comfortable *reading* those diagrams. Having learned from hard experience with database query languages like SQL and SPARQL, the creators of Cypher reasoned that a good graph query language should be predominantly *visual*, like a drawing tool. Correspondingly, a visual notation for graphs is at the heart of Cypher.

1 The authors work at Neo4j at the time of writing.

Creating Data in a Knowledge Graph

Don't worry; being visual doesn't mean you have to be good at graphics! Being visual means that the structure of queries visually resembles the knowledge graph structure in a way that is intuitive. This is best demonstrated with an example. You might remember the small graph in Figure 3-1, which you first saw in Chapter 1. It shows a group of friends and friends of friends along with some data about where they live.

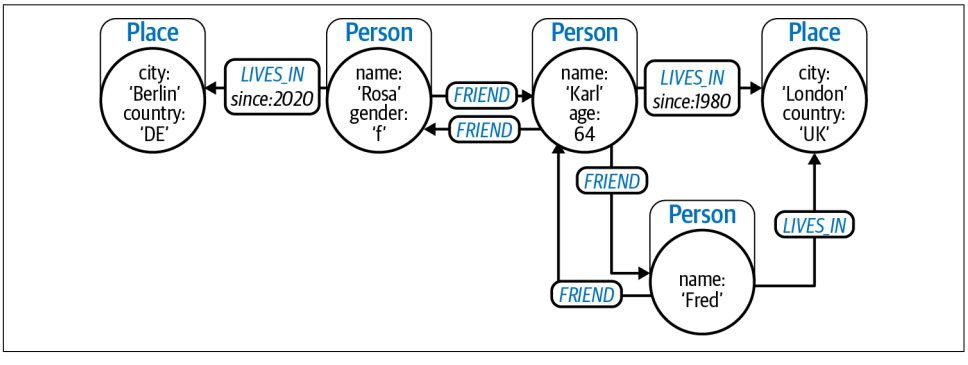


Figure 3-1. A graph representing people, their friendships, and their locations

Looking at the node representing Rosa in Figure 3-1, you can represent her and her connections as ASCII art with no loss of fidelity. As an example, you can capture the semantics of "Rosa lives in Berlin" with the pattern `(:Person {name:'Rosa'})-[:LIVES_IN]->(:Place {city:'Berlin', country:'DE'})`, where:

- The items in `()` parentheses represent nodes.
- The `:Person` and `:Place` text represents labels on those nodes.
- The `-[:LIVES_IN]->` arrow syntax is a relationship named `LIVES_IN` between people and places.
- The `{name:'Rosa'}` and `{city:'Berlin', country:'DE'}` are key-value properties that can be stored on nodes (and relationships).

Now, if you prepend `CREATE` to that pattern, you'll have a valid Cypher query as shown in Example 3-1.

Example 3-1. Using Cypher's `CREATE` keyword to insert a subgraph

```
CREATE (:Person {name:'Rosa'})-[:LIVES_IN {since:2020}]->
       (:Place {city:'Berlin', country:'DE'})
```

In Example 3-1 the graph structure is very clear. Nodes are represented by parentheses `()`, which look like circles in ASCII art. Next, there are labels for the nodes like `:Place` and `:Person`, which indicate the role those nodes play in the graph (places and people in this case). Nodes can have zero or more of these labels.

Labels group nodes together by role. When a node is created with a label, it is added to the index for that label. Most of the time, you won't notice these indexes, but the database query planner uses them to make your queries faster. Node labels add fidelity to your models and increase performance. There's a good maxim here: if you know something about your domain, put it in your knowledge graph.

When storing data, the direction of a relationship is mandatory since it creates a correct, high-fidelity model, disambiguating that Rosa lives in Berlin, rather than Berlin living in Rosa. To store data in Neo4j, you can simply type it into the console as shown in Figure 3-2; after all, *what you draw is what you store*.

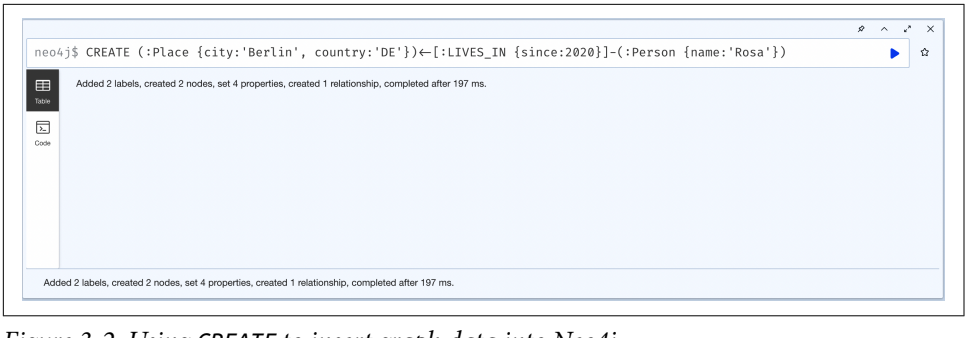


Figure 3-2. Using `CREATE` to insert graph data into Neo4j

Once you've loaded some data into the database, you can query it using Cypher's `MATCH` keyword. The purpose of `MATCH` is to compare user-defined patterns against the underlying knowledge graph. You can see this in Figure 3-3, where the pattern `MATCH (n) RETURN n` asks the database to find any nodes, represented by parentheses `(n)`, and bind those matches the variable `n` to be returned to the caller.

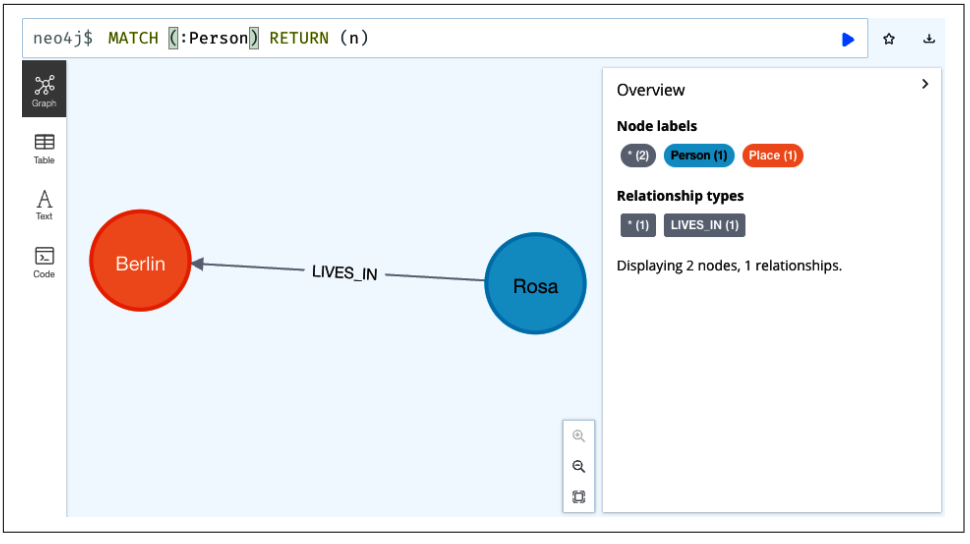


Figure 3-3. Using `MATCH` to find all items in the database

Importantly, you can traverse a relationship in either direction *at the same constant cost*. This is a fundamental tenet of the labeled property graph model. In practice it means that the latency of a query is proportional to how much of a knowledge graph you search. Latency is not simply a function of the size of the graph.

The default view in Figure 3-3 also expands any relationships between the nodes matched, so you can treat this query as "show me the whole graph." For large graphs, this is impractical and usually undesirable, so often you'll append `LIMIT` plus a positive integer value to limit the number of records returned, just like in SQL.

Avoiding Duplicates When Enriching a Knowledge Graph

If you want to grow your graph, you need to create more nodes and relationships. Naively, you might think that just by using `CREATE` you can build a large graph, but `CREATE` *always* creates new data. If you run the code in Figure 3-2 again and refresh the graph display, you will see more than one Rosa node and more than one Berlin node, as shown in Figure 3-4.

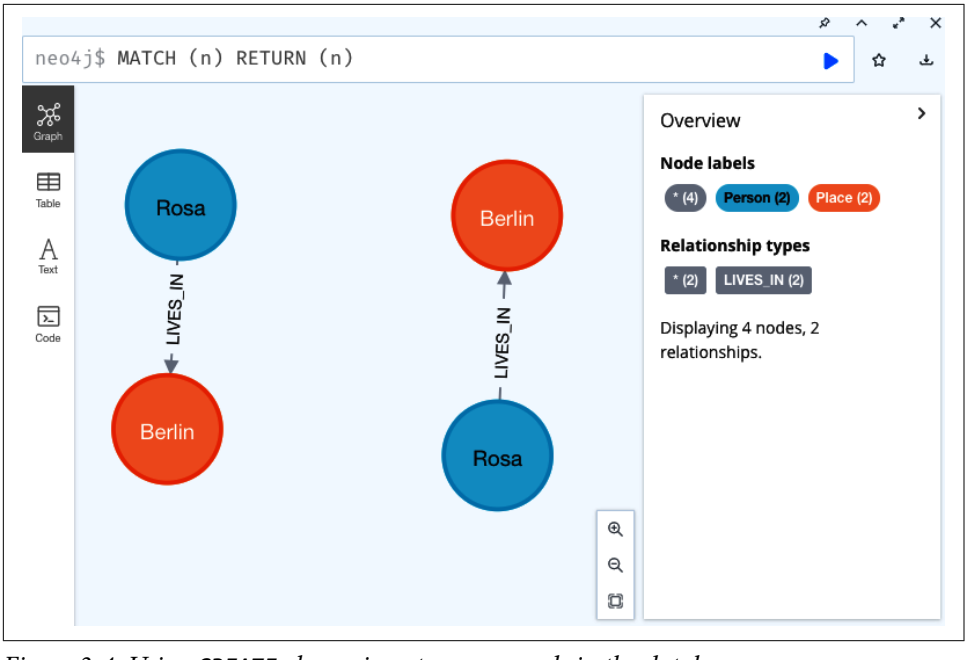


Figure 3-4. Using `CREATE` always inserts new records in the database

Sometimes you do want to `CREATE` new records, but often you don't want duplicates in your data. Take a step back, clean up the database, and start again. To remove records from the database, Cypher supports `DELETE`, which removes the matched records or cleanly aborts if that would leave any relationships dangling. It also has a specialized `DETACH DELETE`, which removes a matched node *and* its associated relationships, shown in Example 3-2.

Example 3-2. Using Cypher `DELETE` and `DETACH DELETE`

```
MATCH (n) DELETE n // Deletes nodes
                   // Cleanly aborts if it leaves relationships dangling.

MATCH ()-[r:LIVES_IN]->()   // Deletes LIVES_IN relationships between any nodes
DELETE r

MATCH (n) DETACH DELETE n // Deletes all nodes and any relationships attached,
                          // effectively deleting whole graph.
```

Reset the database by issuing the command `MATCH (n) DETACH DELETE n`. This time around, since you know that `CREATE` will always insert new records, you'll use Cypher's `MERGE` keyword, which inserts records *only if* the entirety of the supplied pattern does not already exist.

First, create a graph where Karl lives in London: **`MERGE (:Person {name:'Karl', age:64})-[:LIVES_IN {since:1980}]→(:Place {city:'London', country:'UK'})`**. Since there are no records in the database, this `MERGE` acts like a `CREATE`, and as you would expect, two nodes and a single relationship are persisted in the knowledge graph, as shown in Figure 3-5.

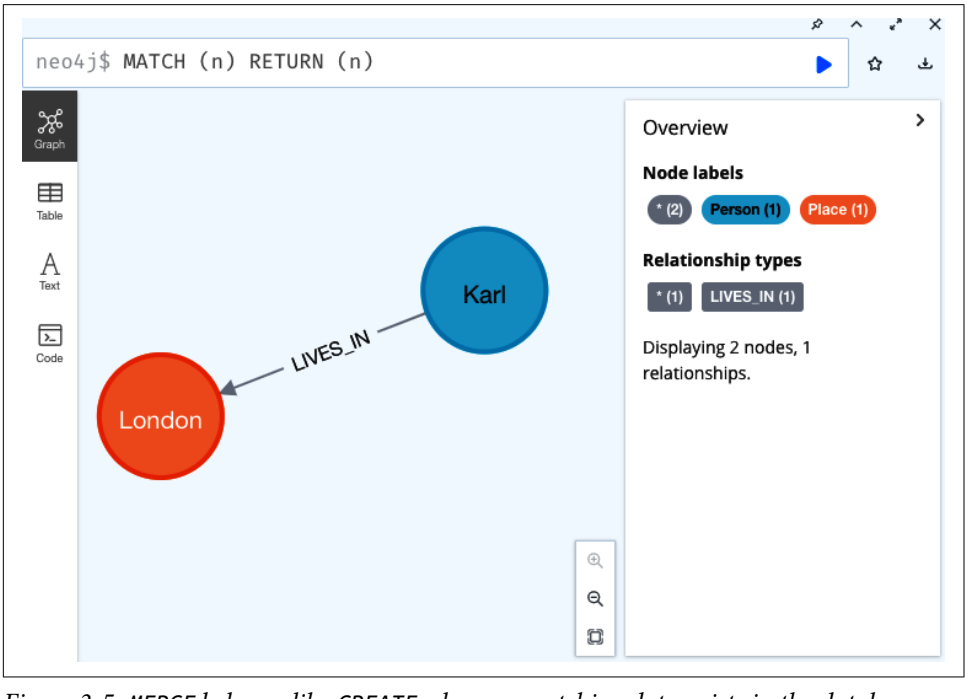


Figure 3-5. `MERGE` behaves like `CREATE` when no matching data exists in the database

Now you can see what happens when you `MERGE` Fred, who also happens to live in London. If you type **`MERGE (:Person {name:'Fred'})-[:LIVES_IN]→(:Place {city:'London', country:'UK'})`**, you might expect the database to create a new node to represent Fred and connect it via a new `LIVES_IN` relationship to the existing node representing London. But, surprisingly, that is not what happens.

`MERGE` is subtle. Its semantics are a mix of `MATCH` and `CREATE` insofar as it will *either* match *whole* patterns or create new records that match the pattern *in its entirety*. It will *never* partially `MATCH` and partially `CREATE` a pattern. As such, in Figure 3-6 what happens is that `MERGE` cannot fully match the pattern `(:Person {name:'Fred'})-[:LIVES_IN]->(:Place {city:'London', country:'UK'})`, so it creates that pattern instead, resulting in a unwanted duplicate `London` node (and no connection between `Fred` and `Karl`).

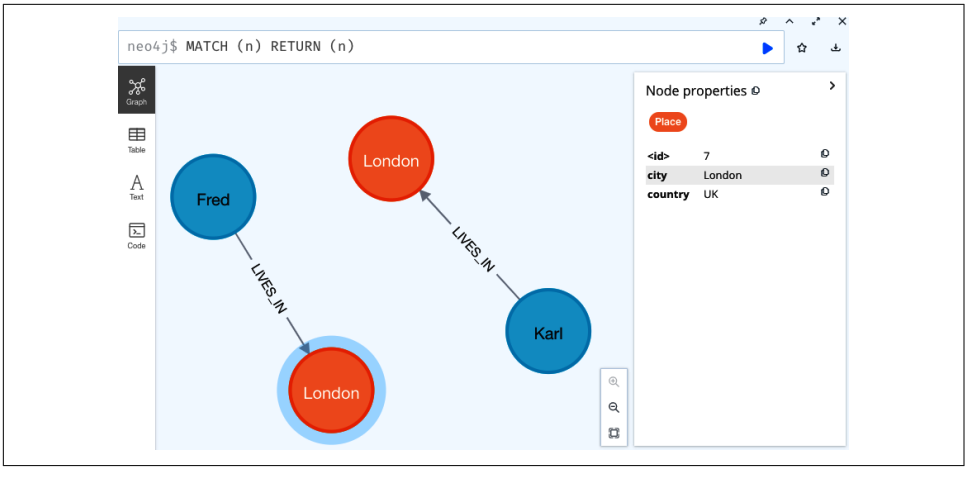


Figure 3-6. `MERGE` can't wholly `MATCH` the pattern on existing data, so it acts like `CREATE`

You're not too far into creating your data model, so the quickest thing is to type **`MATCH (n) DETACH DELETE n`**, which clears out the whole graph. This time around, think more carefully about the data model. Clearly, there should be only one London, UK, and many people can live there. Also clearly, other places called London (e.g., London, Ontario, in Canada) exist and should be allowed. You can constrain the data model to support the existence of a single London, UK, node after which any updates that try to create additional identical nodes will be (cleanly and safely) rejected.

Create a constraint using `CREATE CONSTRAINT no_duplicate_cities FOR (p:Place) REQUIRE (p.country, p.city) IS NODE KEY`. This statement declares that `Place` nodes need a unique composite key composed from a combination of `city` and `country` properties. Composite constraints are available in Neo4j Enterprise Edition, which is available for free for non-production use as part of Neo4j Desktop. In turn, this ensures the coexistence of London, UK, and London, Ontario, in the database but prevents duplicates of them (and any other city and country combination).

Now that you have a constraint in place, you can go back to thinking about how you'd like to connect both `Karl` and `Fred` to the node representing London, UK. To do so, you have to decompose this into three distinct steps:

1. Create or find a node representing London.
2. Create or find a node representing Karl and connect it to the node representing London.
3. Create or find a node representing Fred and connect it to the node representing London.

Recall that `MERGE` has the semantics that it exclusively creates or matches patterns in the graph. The `MERGE` statements in Example 3-3 are executed as part of a single transaction and so will be applied (or rejected) atomically. Moreover, because of the uniqueness constraint, the database won't accept duplicate `Place` nodes and will abort any transactions that try to create them.

Example 3-3. Multiline `MERGE` query

```
MERGE (london:Place {city:'London', country:'UK'})  // Creates or matches a node to
                                                    // represent London, UK
                                                    // Binds it to variable "london"

MERGE (fred:Person {name:'Fred'})   // Creates or matches a node to represent Fred
                                    // Binds it to variable "fred"

MERGE (fred)-[:LIVES_IN]->(london)  // Create or match a LIVES_IN relationship
                                    // between the fred and london nodes

MERGE (karl:Person {name:'Karl'})   // Creates or matches a node to represent Karl
                                    // Binds it to variable "karl"

MERGE (karl)-[:LIVES_IN]->(london)  // Create or match a LIVES_IN relationship
                                    // between the karl and london nodes
```

If you run the code in Example 3-3 on a clean database, you can then run **`MATCH (n) RETURN (n)`** to see all of its contents. What you get is a graph like Figure 3-7.

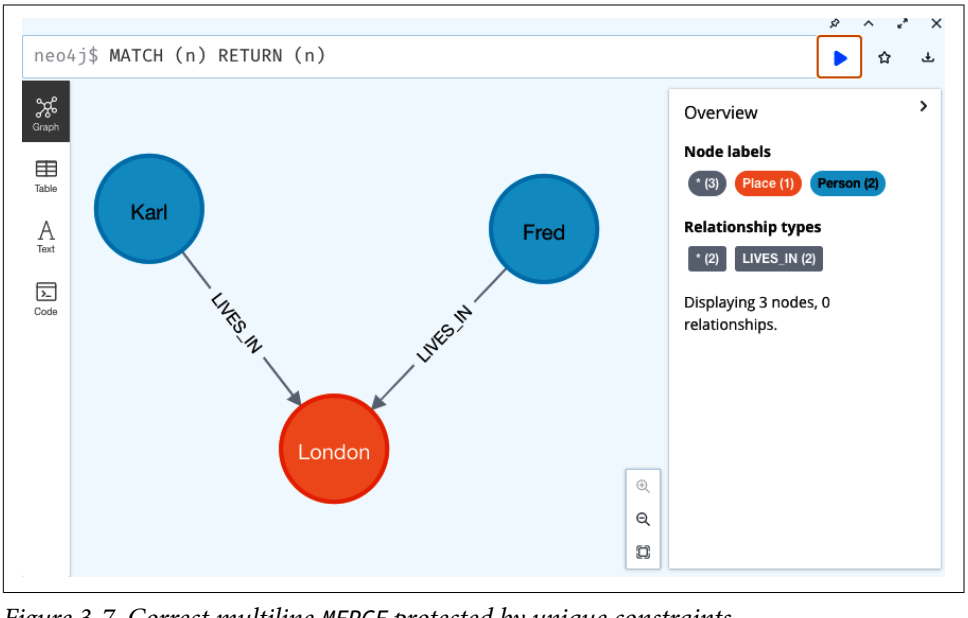


Figure 3-7. Correct multiline `MERGE` protected by unique constraints

As *developers* of knowledge graphs, you have to understand Cypher at a reasonable level. Sometimes queries are typed in the command line, usually for prototyping or ad hoc queries that are outside the normal application workload. Most queries are hosted inside apps that interact with the knowledge graph. Queries are parameterized and sent by the app across the network to be evaluated by the database, just like any relational or NoSQL database. The deployment architecture for graph databases is the same as any other database. It's the data model that sets graph databases apart from other technology.

By repeatedly following the same pattern using `MERGE`, you can build large knowledge graphs that capture your domain in high fidelity, as shown in Figure 3-8.

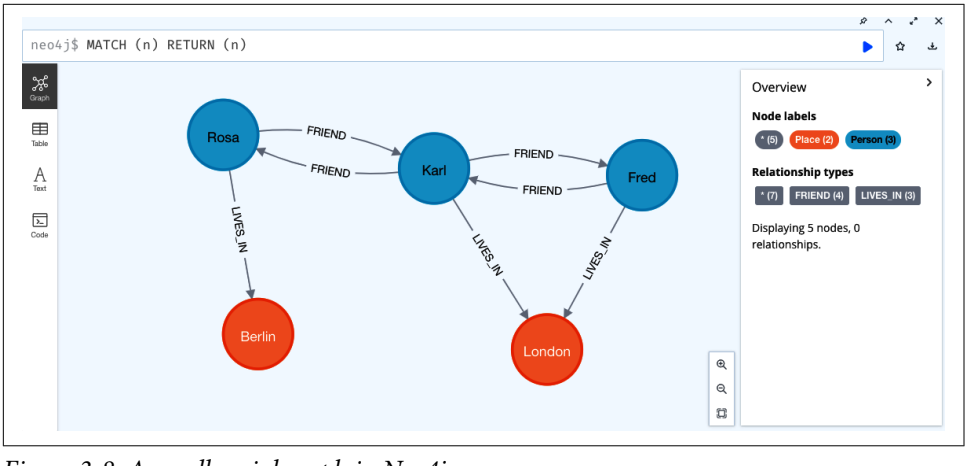


Figure 3-8. A small social graph in Neo4j

Of course, since you are interacting with a database, you can update and enrich the data in your knowledge graph after it has been created. For example, you can add a property that sets Rosa's date of birth:

```
MATCH (p:Person)
WHERE p.name = 'Rosa'
SET p.dob = 19841203
```

Equally, you can remove properties (without deleting the associated node or relationship):

```
MATCH (p:Person)
WHERE p.name = 'Rosa'
REMOVE p.dob
```

`REMOVE` can also be used to strip a label from a node:

```
MATCH (p:Person)
WHERE p.name = 'Rosa'
REMOVE p:Person
```

At this point, you've learned to store data in your knowledge graphs along with some rudimentary querying. It's time to expand your skills with queries that are bound to known nodes in the graph.

Graph Local Queries

Now that you can store data in your knowledge graph like in Figure 3-8, you can turn your attention to querying it. Queries are written using Cypher and follow very similar patterns to the way you stored nodes, relationships, and properties.

The heart of a Cypher query is the `MATCH` clause, as you have already seen with the pattern `MATCH (n) RETURN (n)`. `MATCH` is used to ask the database to find records in the knowledge graph which match a particular pattern. Using `MATCH`, you can specify parts of the graph that you know, binding the pattern to parts of the graph that exist, and leave other parts unbound for the database to find. As a simple example, you could ask the question "Who lives in Berlin?" as shown in Example 3-4. This is called a *graph local query* because it is bound to a specific part of the graph—in this case, the node representing Berlin.[2]

Example 3-4. Who lives in Berlin?

```
MATCH (p:Person)-[:LIVES_IN]->(:Place {city:'Berlin', country:'DE'})
RETURN (p)
```

The query in Example 3-4 is quite readable even if you're new to Cypher. It starts with `MATCH`, which tells the database that you want to find patterns. The pattern of interest has some parts that you know exist in the knowledge graph and leaves some parts for the query to find. Specifically, the pattern `(:Place {city:'Berlin', country:'DE'})` will match the node representing Berlin, and `-[:LIVES_IN]->` will match incoming `LIVES_IN` relationships to that Berlin node. This is a loose pattern since no properties are specified, but it will still match correctly in your knowledge graph as it stands. The `(p:Person)` part of the pattern will match any `Person` nodes in the graph and bind those matches to variable `p` for later use. When taken together, the pattern asks the database to find any matching patterns where any `Person` node has an outgoing `LIVES_IN` relationship to the specific `Place` node representing Berlin.

2 Graph local queries may process many or few records; the local element simply states that some parts of the pattern are bound to known nodes or relationships, rather than iterating the whole graph.

Finally, the query returns any `Person` nodes that have been matched as part of this pattern by `RETURN (p)`, which results in just the node representing Rosa, as shown in Figure 3-9.

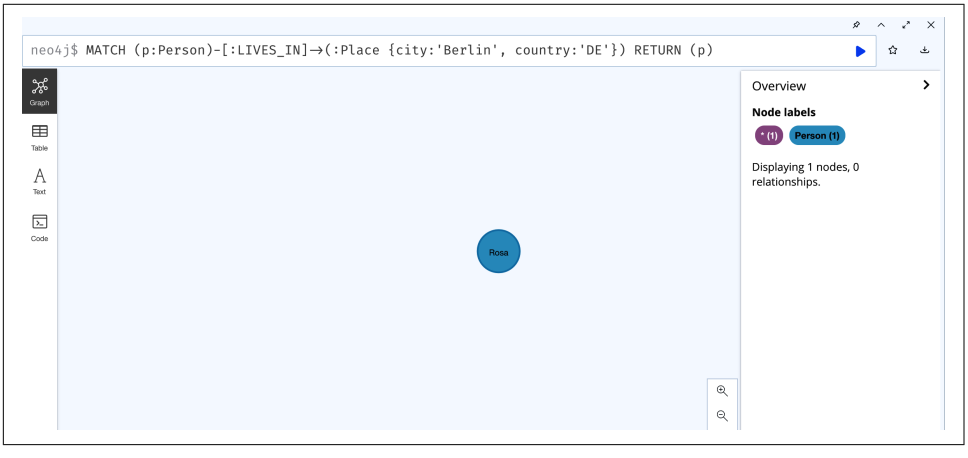


Figure 3-9. Results showing that only Rosa lives in Berlin

This query provides a small amount of useful information from the social graph. But that query is quite *shallow* since it only considers paths of length 1, starting from the node representing Berlin and matching only its immediate neighbors.

More often than not, you'll want to explore deep into a knowledge graph to find insight. Fortunately, this is simple to do in Cypher and computationally cheap to execute in Neo4j. To demonstrate, you can use the social network to find some friends of friends with whom Rosa might want to interact.

In Example 3-5 you start with the node representing Rosa and then look for any `Person` nodes that are connected to her via two outgoing `FRIEND` relationships. You can use the variable-length path syntax `*2..2` in this example to specify path lengths from 2 to 2 (that is, exactly length 2). Less compactly, you could have written the pattern in full to the same effect: `(:Person)-[FRIEND]->(:Person)-[FRIEND]->(:Person)`. These are equivalent, though the shorter version is preferred since it's more readable.

Example 3-5. Naive friends of friends

```
MATCH (:Person {name:'Rosa'})-[:FRIEND*2..2]->(fof:Person)
RETURN (fof)
```

If you run the query from Example 3-5, you get the slightly puzzling results shown in Figure 3-10.

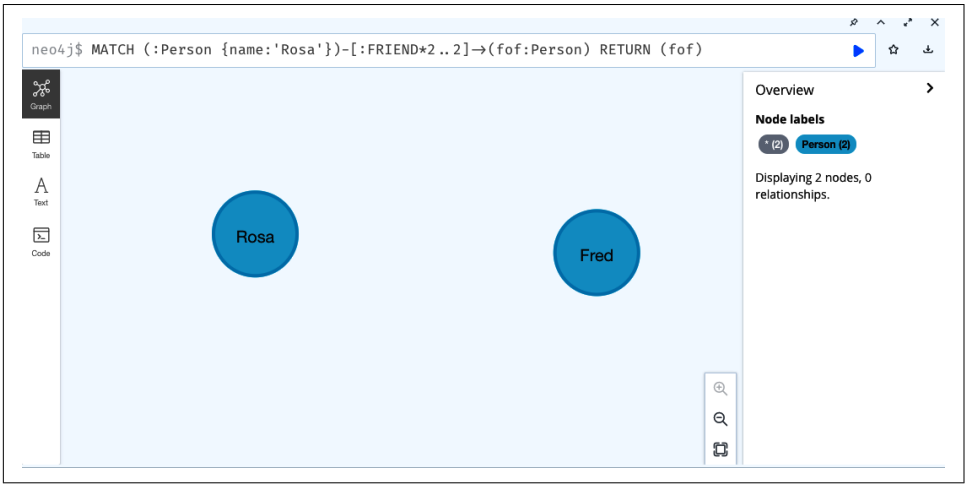


Figure 3-10. Rosa shouldn't be considered her own friend of a friend

In Figure 3-10, Rosa appears as a friend of a friend to *herself*. That doesn't feel right, and you can find the reason why if you look at the graph in Figure 3-8. Since Rosa is a friend of Karl, and Karl is a friend of Rosa, there is a depth-two path that matches your `-[:FRIEND*2..2]->` pattern, from Rosa to Karl and back to Rosa! To avoid including Rosa, you augment your `MATCH` clause with a `WHERE` predicate to constrain the search pattern, as shown in Example 3-6.

Example 3-6. Correctly finding friends of friends

```
MATCH (rosa:Person {name:'Rosa'})-[:FRIEND*2..2]->(fof:Person)
WHERE rosa <> fof
RETURN (fof)
```

The `WHERE rosa <> fof` predicate enriches the pattern. Now it only matches when the node representing Rosa is not the same as the node matched, avoiding the Rosa-Karl-Rosa problem you saw earlier.

There are more predicates you can apply using `WHERE`, including Boolean operations, string matching, path patterns, list operations, property checks, and more, such as:

- `WHERE n.name STARTS WITH 'Ka'`
- `WHERE n.name CONTAINS 'os'`
- `WHERE NOT n.name ENDS WITH 'y'`
- `WHERE NOT (p)-[:KNOWS]->(:Person {name:'Karl'})`
- `WHERE n.name IN ['Rosa', 'Karl'] AND (p)-[LIVES_IN]->(:Place {city:'Berlin'})`

You'll see more of these patterns throughout the book.

After adding the predicate into the query, you find that Rosa's only friend of a friend is Fred (via her friend Karl), shown in Figure 3-11.

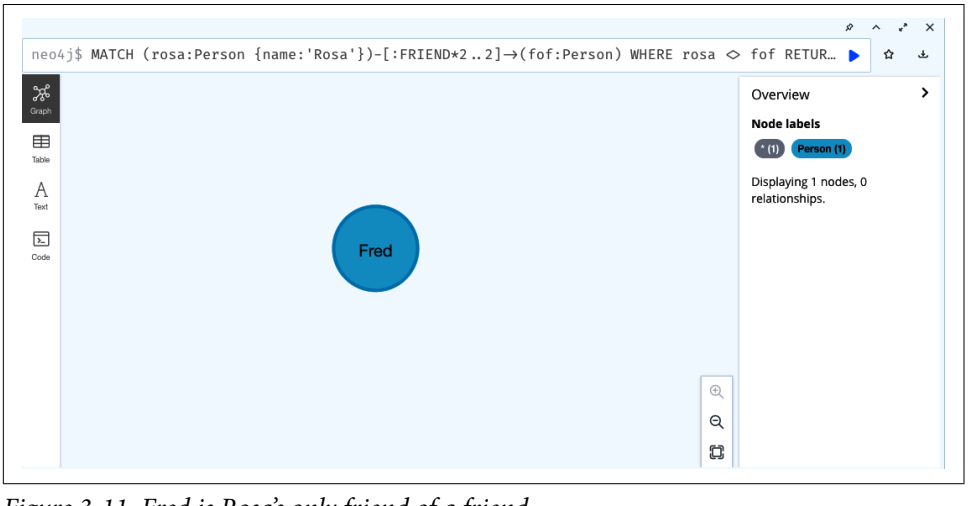


Figure 3-11. Fred is Rosa's only friend of a friend

Following the same query structure, you could just as easily ask, "Who has friends or friends of friends of someone who lives in Berlin?" where you mix relationship types in patterns and increase the depth of the query. This is normal when querying knowledge graphs and easy to express, as shown in Example 3-7.

Example 3-7. Correctly finding friends and friends of friends of someone who lives in Berlin

```
MATCH (:Place {city:'Berlin'})<-[:LIVES_IN]-(p:Person)<-[:FRIEND*1..2]-(f:Person)
WHERE f <> p
RETURN f
```

In this example, you can see that the friends and friends-of-friends don't have to live in Berlin themselves. But it would be a simple addition of another `WHERE` predicate to change the pattern to enforce that, such as:

```
MATCH (:Place {city:'Berlin'})<-[:LIVES_IN]-(p:Person)
        <-[:FRIEND*1..2]-(f:Person)
WHERE f <> p AND (f)-[:LIVES_IN]->(:Place {city:'Berlin'})
RETURN f
```

In Example 3-7 a single `LIVES_IN` relationship must be matched from the node representing Berlin to a person, meaning a person who lives in Berlin. Then there is a variable-length path `<-[:FRIEND*1..2]-` which matches a friend (at depth 1) or a friend of a friend (at depth 2). As previously shown in Example 3-6, you have to ensure that the starting person isn't counted as their own friend of a friend, and for that you need to add `WHERE f <> p`.

Expert relational database users might be concerned about a performance hit from this uninhibited increase in depth but shouldn't be. Unlike similar queries in a relational databases which use recursive joins, the latency of a query in Neo4j is proportional to how many nodes and relationships are visited, rather than proportional to the size of the database. There are no join-bomb problems to be concerned about, and since the cost of traversing a single relationship is very low (on the order of millionths of a second), a great deal of graph search work can be performed very quickly.

Now you have an answer to who has friends or friends of friends who live in Berlin: it's Karl (Rosa's friend) and Fred (Karl's friend), as shown in Figure 3-12.

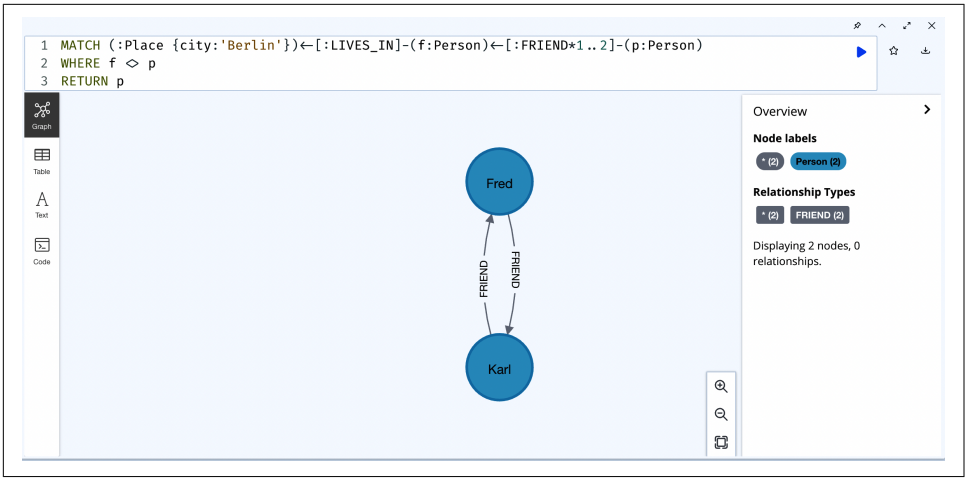


Figure 3-12. Karl and Fred have friends or friends of friends who live in Berlin

Graph Global Queries

So far, you've anchored your queries to specific nodes, such as asking explicitly about Rosa or Berlin. These are called called graph local queries and are quite common in practice.

But what if you want to query the whole graph, as is often the case with knowledge graphs? These queries are called *graph global*. You might want to ask the simple question, "Which are the most popular cities to live in?" In this case, your query pattern isn't bound to any specific node but instead must consider all cities and their populations. In your query, you don't specify any particular node as a starting point because you want to process large parts of the knowledge graph (or even all of it).

The Cypher query and results from the very small social network are shown in Figure 3-13. You can see that London is the most popular city in the social network with two people living there, followed by Berlin with one person. Tabular results are used rather than graph visualization since aggregate numerical data is the goal.

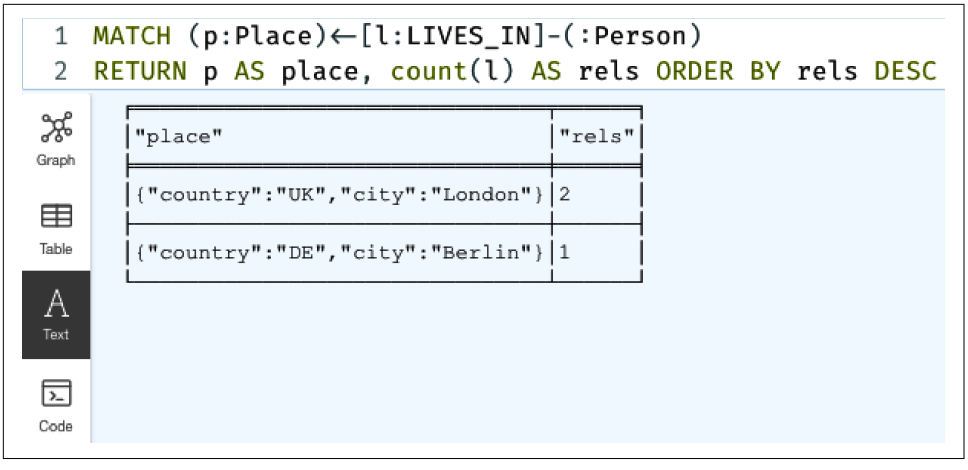


Figure 3-13. London is the most popular city in your social network, followed by Berlin

The graph global query in Figure 3-13 is quite readable and in fact is very similar to the graph local queries that you saw earlier. At the start of the query, there is `MATCH (p:Place)<-[l:LIVES_IN]-(:Person)`, which is a familiar pattern by now. You ask the database to match any pattern that has a `Place` node with an incoming `LIVES_IN` relationship from any `Person` node. Whenever the pattern is matched, you can access variables bound to the matched nodes. `Place` nodes are bound to the variable `p` and `LIVES_IN` relationships bound to the variable `l` where they can be accessed elsewhere in the query.

You don't strictly need the `Person` label in this query since you know that only `Place` and `Person` nodes are joined by `LIVES_IN` relationships. But when querying graphs, if you know something about your graph, *explicitly put it in your query*. For example, adding the `Person` label rather than using an anonymous node `()` will help the query planner create better query plans, which improves performance and reduces latency. But *never* leave out the relationship type since that will explode the search space for a query and consequently greatly increase latency.

Following on from the `MATCH` clause in Figure 3-13, you see `RETURN p AS place, count(l) AS rels ORDER BY rels DESC`. This takes more unpacking but will be familiar to anyone with experience with SQL. First, `RETURN p AS place` means that any pattern matches bound to `p` (`Place` nodes) will be returned, but instead of being called `p` in the results (too pithy), results will be called `place`. Along with `place`, the query returns the number of `LIVES_IN` relationships attached to a `Place` node and with `count(l) AS rels` where `l` is matched from the pattern (specifically `LIVES_IN`

relationships) and given the friendly name `rels`. Since you want popular cities, you'll need to impose an ordering on results using `ORDER BY rels DESC`, which says to order the results on the value of the variable `rels` and return them in descending order (highest first).

Since Figure 3-13 is a very small social network, you don't need to filter the results. With just two rows of data returned, all that was needed was to make the names a little more friendly and sort the results. But for production databases, full results could be much larger and may not even be helpful (too much data overwhelms the user). Fortunately, Cypher has aggregation functions `avg` (average), `max` (maximum), `min` (minimum), `sum`, and so forth, along with clauses like `SKIP` (skip some results) and `LIMIT` (limit the number of results returned). Using these, you can craft graph global queries that run over very large knowledge graphs and still return compact, pertinent aggregate information to the user.

Calling Functions and Procedures

Like Cypher, Neo4j supports operations on knowledge graphs by calling functions and procedures which implement specific tasks. Invoking them is easy: you use the Cypher `CALL` syntax followed by the (namespaced) procedure name and any arguments in parentheses.

For instance, an often-used procedure is to examine the schema of the graph. Figure 3-14 shows how to visualize the schema of the social network knowledge graph that you first saw in Figure 3-8.

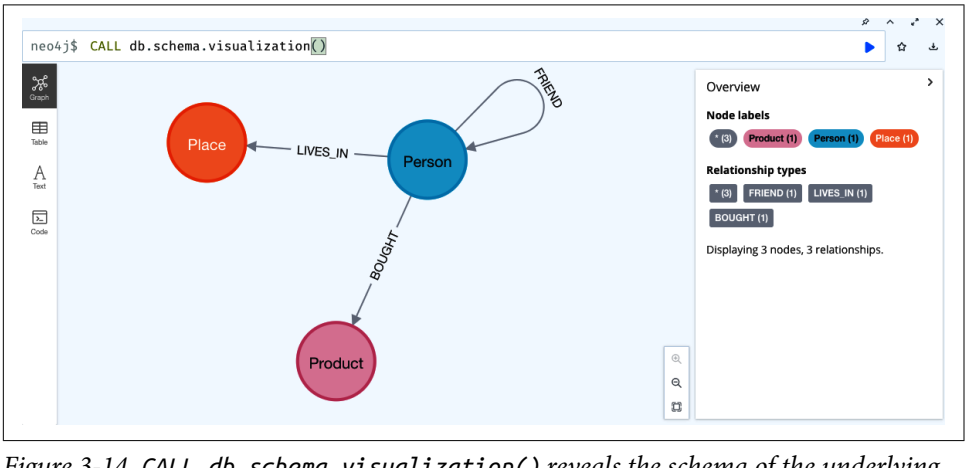


Figure 3-14. `CALL db.schema.visualization()` reveals the schema of the underlying knowledge graph

Another commonly used set of procedure calls comes from the APOC library. APOC is a popular library that ships with Neo4j and contains a plethora of useful functionality and shortcuts for operating on knowledge graphs.

APOC is full of useful procedures and functions. It's a good idea to get to know them to avoid reinventing them yourself.

Example 3-8 shows how the `apoc.atomic.concat` procedure atomically joins property strings together (with no partial results visible).

Example 3-8. Concatenating property strings with APOC

```
MATCH (p:Person {name:'Fred'})
// Fred becomes Freddy
CALL apoc.atomic.concat(p,"name",'dy')
YIELD oldValue, newValue
RETURN oldValue, newValue;
```

Invoking functions is similar to invoking procedures, but you don't need the `CALL` syntax. Instead, simply insert the function where you'd normally insert a value, as shown in Example 3-9.

Example 3-9. APOC function for date/time conversion

```
RETURN apoc.date.convert(datetime().epochSeconds, "seconds", "days") as outputInDays;
```

Using procedures and functions from libraries can remove drudgery and reduce the possibility of errors when using a knowledge graph. Before embarking on something potentially complicated in Cypher, it's worth checking to see if there's a procedure that can help.

Supporting Tools for Writing Knowledge Graph Queries

There are two more tools you should know to become competent at Cypher. The `EXPLAIN` and `PROFILE` keywords help developers understand the performance of their queries, which is especially important as your knowledge graphs become larger.

`EXPLAIN` provides a visualization of a query plan, guiding the developer toward reducing the amount of data ("DB hits") flowing through the query. The visualization that `EXPLAIN` creates, shown in Figure 3-15, is particularly useful for understanding the behavior of larger queries before actually running them.

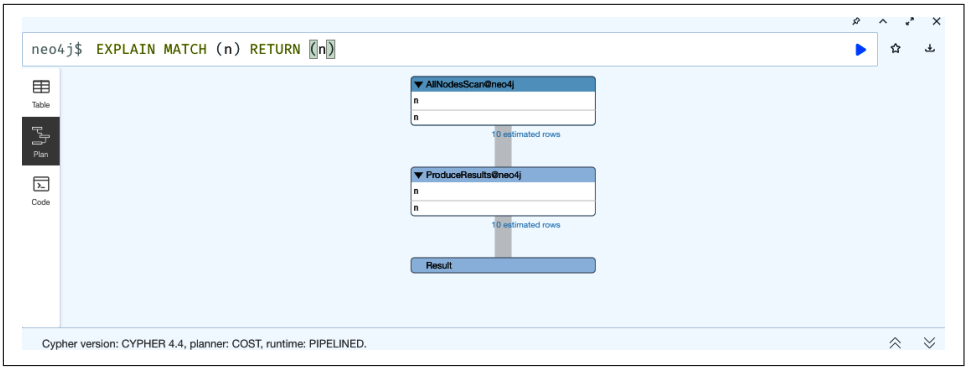


Figure 3-15. `EXPLAIN` shows how the query planner thinks the query will execute without actually executing it

`PROFILE` goes one step further and instruments a running query to see the actual behavior at runtime. Using `PROFILE`, developers can see how the query behaves when executed on the real knowledge graph and tune it accordingly, as shown in Figure 3-16.

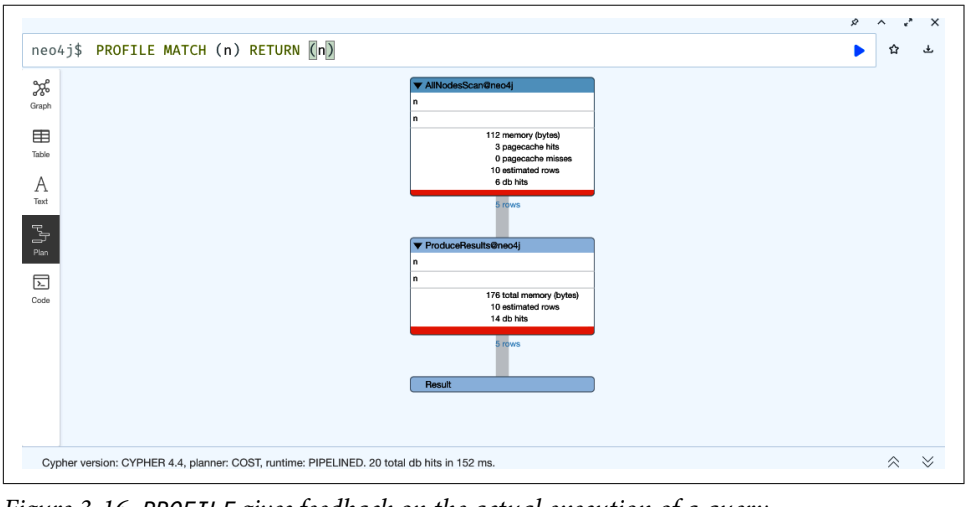


Figure 3-16. `PROFILE` gives feedback on the actual execution of a query

Careful use of `EXPLAIN` and `PROFILE` can significantly boost query performance. For example, if excessive amounts of data (or DB hits) flow from operator to operator, look for opportunities to refactor the query to reduce cardinality earlier. At a macro level, where the visualization shows expensive tasks like scanning, it can indicate that a useful index is missing, trading brute-force scans for index lookups. The orange box at the bottom of each operator shows the time cost of execution. Always seek to minimize that, even at the expense of DB hits.

Neo4j Internals

It might seem slightly strange to read about database internals in a book aimed at an important use case.[3] You *definitely* don't need to be a database engineer to get significant value from knowledge graphs. But this is a good guiding principle:

> You don't have to be an engineer to be a racing driver, but you do have to have mechanical sympathy.
>
> —Jackie Stewart, three-time Formula 1 racing champion

With a little mechanical sympathy, you can design knowledge graphs that will be robust and performant, delighting the users of your systems. Knowing a little about how the database works under the covers will help you to work with the grain, not against it. At the highest level, a graph database performs two functions on behalf of its users:

- Queries graph data performantly
- Stores graph data safely

You will see how each of these is implemented (at a high level) in Neo4j so that you can utilize the machinery well and create responsive knowledge graph systems.

For any production system, you need high availability, monitoring, security, online updates, and many other capabilities. It's safe to say that these exist (albeit at varying levels of maturity) across many of the graph databases available today both in the cloud and on premises.

Query Processing

To query knowledge graphs performantly, a graph database must make traversals—the act of moving from one node across a relationship to another node—fast (for low latency) and cheap (for high concurrent throughput). Accordingly, the way Neo4j stores its data (both on disk and when it resides in RAM) is optimized for graph traversals. Neo4j stores the structure of a graph (nodes and relationships) separately from the property data. The graph structure is stored as fixed-length records: one store for nodes and a similar one for relationships. Multiplying the ID of a record by its size in bytes gives you its offset in the corresponding store file. This pattern is known as *index-free adjacency* and has been written about in "The Graph Traversal Pattern" by Marko A. Rodriguez and Peter Neubauer. Index-free adjacency via pointer-chasing is very efficient on modern computers.

3 If you're certain that you don't want to learn how graph databases work, you can skip ahead to Chapter 4.

It's noteworthy that there are typically no indexes involved in traversing a graph (unless the query optimizer spots a shortcut that involves indexes). The relationships on a node act as a local "index" that guides the traversal to where it can (legally) go next.

Index-free adjacency is such a powerful property because it gives you `O(1)` or "constant time" traversals. This means that graph query latency is proportional to the amount of graph traversed, rather than being proportional to the size of the graph. It has the effect that even complex queries tend to run quickly and at low computational cost.

Property storage in Neo4j is different, designed for data flexibility. Since properties can take many forms in knowledge graphs, like strings, numbers, lists, and more, the storage format needs to be accommodating. In Neo4j, properties are maintained in lists of descriptive property records where the head of the list is referenced from the associated node or relationship. Unlike node or relationship stores, property stores are `O(N)` or "linear time" for reads, meaning that access to a property is proportional to the number of properties stored for that node or relationship. For writes, the property stores remain `O(1)` since the new property is added to the head of the relationship chain.

One thing that knowledge graph developers should be aware of is the cumulative cost of property access. Though each access itself might be cheap in elapsed time (especially if the data is currently cached in RAM), accessing many properties during traversals adds up. In the case of larger traversals, frequent access to properties can increase latency greatly, up to double according to some practitioners.

Where possible, queries should traverse the graph structure first and then retrieve property data once target nodes and relationships have been found. This ensures that most work is done quickly and cheaply with `O(1)` operations, with fewer more expensive `O(N)` operations.

Sometimes you can't avoid lots of property lookups in your knowledge graph queries, because properties influence how the graph is traversed. While the database can't do anything about the theoretical cost of those accesses, it can in practice greatly reduce their impact by ensuring that the working set is in the faster parts of the memory hierarchy. Modern computers, of course, have a well-understood memory hierarchy: from disks (including SSDs) being the slowest, through RAM, to CPU caches being the fastest.

Use Modern Hardware

Don't use spinning disks. The seek time for a fast spinning disk is 5 ms, which is a very long time for computers. During that time, the database could have completed many thousands of traversals instead of waiting for I/O. For any serious project, it's worth investing in good disks and plenty of RAM to host the data for the best possible performance.

It's the database management system's job to ensure that the records needed next by a query are in memory (most of the time) rather than having to go to disk to retrieve them. It uses locality-aware caching strategies to try to ensure that hot data is in a fast part of the memory hierarchy. If this is maintained, the physical cost of accessing any record, including properties, is significantly reduced. To maximize the chances of this working sustainably, a healthy ratio of RAM to data size should be used. For the most demanding use cases, the whole knowledge graph should fit in RAM. With this in mind, sometimes the cheapest and quickest remedy for slow workloads is simply to add more RAM!

ACID Transactions

To store data safely, Neo4j—like many other databases—has a *transactional* write channel that provides fault-tolerant writes. This is based on a classic "write-ahead log," whereby intended updates to the database are first persisted in an ordered fashion on disk before being applied to the knowledge graph. This gives the database the ability to tolerate power cycling. When power is restored, it can recover any transactions that have been cleanly persisted in the log and safely discard those whose data was only partially stored.

Neo4j maintains ACID (atomicity, consistency, isolation, durability) transactions across servers as well as in the single-server case. To do so, it uses an algorithm called Raft that keeps individual transaction logs "tied" together (hence the name).

Atop Raft, Neo4j provides a causal barrier, which means that users of the database always see at least their own writes, even with servers spread out over a wide-area network. This causal feature is opt-in and transparent but makes working with the database cluster as simple as reasoning about a variable in a programming language.

Making a database fault tolerant is not an easy thing to do, and there are many edge cases. But from the point of view of a knowledge graph user, your data is kept safe

from corruption and highly available in the presence of software and hardware faults. A trustworthy, fault-tolerant graph database is the right underlay for knowledge graphs at enterprise scale (and beyond). Having an "operationally boring" graph database leaves you to focus on unlocking the value from your knowledge graphs.

Summary

In this chapter, you've seen how Cypher queries are a pleasant pattern-matching syntax based on ASCII art. The underlying graph database provides a high-performance platform for storing and querying knowledge graph data.

Using what you've learned in this chapter, you can query knowledge graphs performantly and update them safely. But it's often the case that at least some data for the knowledge graph already exists in other systems. The next chapter discusses methods for importing data into a knowledge graph to use it as a foundation for further enrichment and gaining valuable insight.

CHAPTER 4

Loading Knowledge Graph Data

In Chapter 3 you became acquainted with graph databases and query languages, and even learned a little about database internals. You also saw how to query and update knowledge graphs. But there's a special case of updating that warrants more depth: bulk imports, which are useful across the whole lifecycle of a knowledge graph.

For each of the techniques in this chapter, it makes sense to start small. Take a small, representative slice of your data and import it into the database. Check that the model works as intended before running the full import job. A quick check can save you minutes or hours of frustration.

In this chapter, you'll learn about three ways of bulk loading data into a knowledge graph. You'll learn how to incrementally load bulk data into a live knowledge graph and how to bootstrap a knowledge graph with huge amounts of data. But first you'll learn how to use a graphical tool to define the structure of a knowledge graph and map data onto that structure for bulk imports.

Loading Data with the Neo4j Data Importer

You should start with the easiest of the three popular tools for Neo4j bulk data loading. The Neo4j Data Importer is a visual tool that allows you to draw your domain model as a graph and then overlay data onto that graph, with comma-separated values (CSV) files that contain the data for your nodes and relationships.

Data Importer greatly simplifies bootstrapping your knowledge graph, especially for beginners. However, it does still require that you have your data in CSV files ready to use.

To keep the examples short, they'll use the social network data that you saw in Chapters 1 and 3. Bear in mind that this simple example works but real imports tend to use much larger files!

First, draw the social network. The diagram in Figure 4-1 shows the domain model, consisting of `Person` and `Place` nodes and `FRIEND` and `LIVES_IN` relationships.

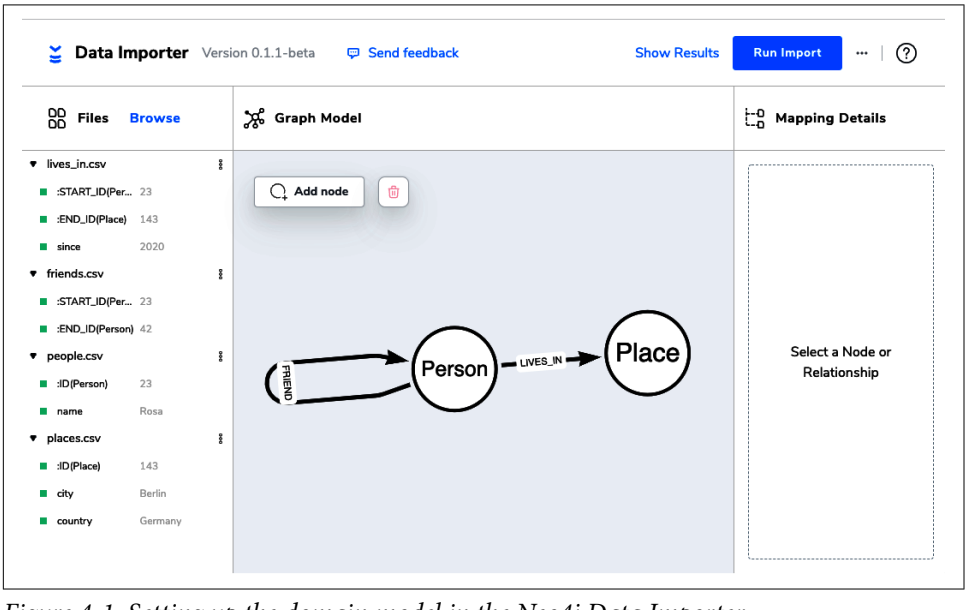


Figure 4-1. Setting up the domain model in the Neo4j Data Importer

First, add your CSV files like those shown in Examples 4-1, 4-2, 4-3, and 4-4 by browsing for them in the left panel of the tool. Draw the nodes and relationships that constitute the domain model using the simple drawing tools, and annotate each of them using the mapping details panel on the right.

Example 4-1. `people.csv` data for `Person` nodes in the Neo4j Data Importer

```
:ID(Person),name
23,Rosa
42,Karl
55,Fred
```

In Example 4-1, the ID of the node and its label are specified as `:ID(Person)` with a single property `name`. In the subsequent rows, you can see data that matches that pattern, like `23,Rosa`. Note that the `ID` column is used later to link the graph together; it's not (necessarily) part of the domain model and can be expunged from the model at a later point.

Example 4-2. `friends.csv` for `FRIEND` relationships between `Person` nodes in the Neo4j Data Importer

```
:START_ID(Person),:END_ID(Person)
23,42
42,23
42,55
55,42
```

In Example 4-2, the start and end IDs of the `Person` nodes are specified with `:START_ID(Person),:END_ID(Person)`. You have a relationship that goes from one `Person` node represented as an ID to another. In each row you can see the corresponding IDs of the start and end nodes of a `FRIEND` relationship, which matches the data in Example 4-1.

Example 4-3 follows the same pattern as Example 4-1, but for `Place` nodes rather than `Person` nodes. The header specifies that there will be an `ID` for `Place` nodes followed by `city` and `country` properties to be written onto each of those nodes.

Example 4-3. `places.csv` data for `Place` nodes in the Neo4j Data Importer

```
:ID(Place),city,country
143,Berlin,Germany
244,London,UK
```

Finally, Example 4-4 declares the `LIVES_IN` relationships between `Person` and `Place` nodes. The header specifies the ID of `Person` nodes as the start of the relationship, the ID of `Place` nodes as the end of the relationship, and an optional `since` property on the relationship. The CSV data then follows that pattern.

Example 4-4. `lives_in.csv` for `LIVES_IN` relationships between `Person` and `Place` nodes in the Neo4j Data Importer

```
:START_ID(Person),:END_ID(Place),since
23,143,2020
55,244
42,244,1980
```

Given these CSV files, you can now run the Neo4j Data Importer. It executes against a live database and so requires an endpoint and credentials in order to work. Once the Data Importer has successfully executed, you will receive an import report like the one shown in Figure 4-2.

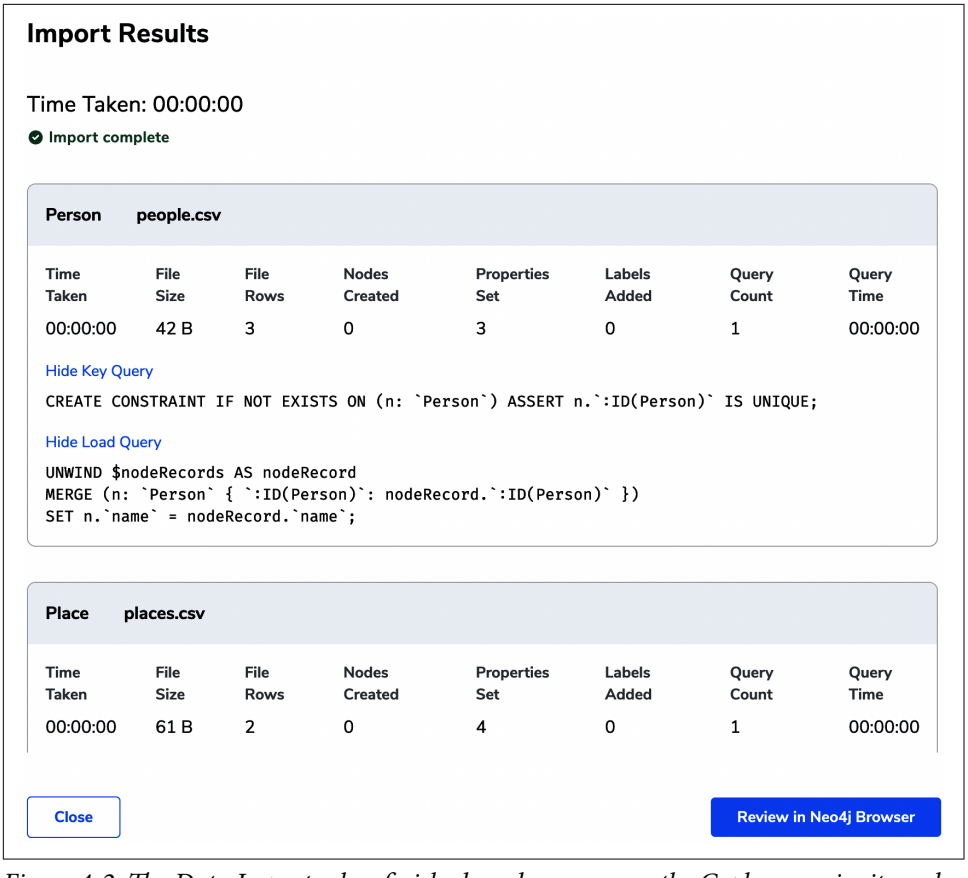


Figure 4-2. The Data Importer has finished, and you can see the Cypher queries it used to load data into Neo4j

In Figure 4-2, you can see some of the Cypher scripts that have executed against the database along with statistics on the import. A handful of common Cypher idioms stand out from the report: `CREATE CONSTRAINT`, `UNWIND`, and `MERGE`. Here you should use `UNWIND` to feed rows of CSV data into a `MERGE` operation that creates a `Person` node with an `ID` property. It then writes a `since` property on that node using `SET`. If there is no `since` property for any row, that's not a problem because `SET` tolerates the absence of data for those relationships.

`UNWIND` is a Cypher clause that takes a list of values and transforms them into rows that are fed into subsequent statements. It's a very common way to handle input data, and you'll see it frequently in this chapter. Its usual form is

```
UNWIND [1, 2, 3, null] AS x
RETURN x
```

where the values `1`, `2`, `3`, and `null` will be returned as individual rows to the caller.

`UNWIND` is again used to feed relationship data from Example 4-4 into the graph, as shown in Figure 4-3.

LIVES_IN **lives_in.csv**

Time Taken	File Size	File Rows	Relationships Created	Properties Set	Query Count	Query Time
00:00:00	70 B	3	0	2	1	00:00:00

Hide Load Query

```
UNWIND $relRecords AS relRecord
MATCH (source: `Person` { `:ID(Person)`: relRecord.`:START_ID(Person)` })
MATCH (target: `Place` { `:ID(Place)`: relRecord.`:END_ID(Place)` })
MERGE (source)-[r: `LIVES_IN`]→(target)
SET r.`since` = toInteger(relRecord.`since`);
```

Figure 4-3. Mixing `MERGE` and `MATCH` with data fed by `UNWIND`

Also in Figure 4-3, you can see how a combination of `MERGE` and `MATCH` is used. Remember that in Chapter 3 you used `MERGE` when it wasn't clear whether or not a node or relationship existed, since `MERGE` acts like `MATCH` when a record exists and like `CREATE` when it does not. Here you don't `MERGE` for `Person` and `Place` nodes because they are guaranteed to exist in the input CSV file. Instead, you only need to `MERGE` the `LIVES_IN` relationship between those existing nodes, where `MERGE` will ensure there are no duplicate relationships.

Of course, you could write the equivalent Cypher by hand and run it as a set of scripts, and it would have the same effect (as per the following section, "Online Bulk Data Loading with LOAD CSV"). You could also take this Cypher and treat it as an automatically generated source, which could be modified and hosted in source control so that it can be evolved as the knowledge graph grows over time.

The Neo4j Data Importer gives you a head start not only with the Cypher code for ingestion but also by visualizing, verifying, and debugging the model and the data that will flow into it *before* you start the import. As such, it's valuable to experienced Neo4j developers as well as folks who are new to graph databases.

Online Bulk Data Loading with LOAD CSV

In Chapter 3 you saw how to create (and update) records in the database using the `CREATE` and `MERGE` Cypher commands. In "Loading Data with the Neo4j Data Importer" on page 51, you learned how a visual tool generates Cypher code that then loads CSV data into a live database.

But there is another way of scripting CSV imports: using Cypher's `LOAD CSV` command. `LOAD CSV` allows you to import CSV data into an online graph from a variety of places. You can load CSV data via web addresses (like Amazon S3 buckets or Google Sheets) and filesystems, and `LOAD CSV` imports happen while the database is live and serving other queries. Usefully, CSV files can also be compressed, which helps when transferring large amounts of data.

`LOAD CSV` is most simply explained by example. First load all the `Place` nodes from the social network using `LOAD CSV`. The CSV file used is in Example 4-5.

Example 4-5. CSV data for `Place` nodes

```
city,country
Berlin,Germany
London,UK
```

Since the place data is uniform, you can use a straightforward `LOAD CSV` pattern to insert the nodes into the knowledge graph, as shown in Example 4-6.

Example 4-6. Simple `LOAD CSV` for `Place` nodes

```
LOAD CSV WITH HEADERS FROM 'places.csv' AS line
MERGE (:Place {country: line.country, city: line.city})
```

While Example 4-6 isn't a complete solution to loading the knowledge graph (since it only loads places), it exemplifies the basic pattern for using `LOAD CSV`. The first line breaks down as follows:

- `LOAD CSV` is the Cypher command you want to execute.
- `WITH HEADERS` tells the command to expect header definitions in the first line of the CSV file. While headers are optional, it's usually a good idea to use them. Headers allow you to refer to columns with a friendly name (rather than just a numerical index), as shown in the the `MERGE` statement in Example 4-6.
- `FROM 'places.csv'` tells the command where to find the CSV data. Note that Neo4j security permissions may have to be altered if you want to import from nondefault locations in the filesystem or from remote servers.

- `AS line` binds each row in the CSV file to a variable in Cypher called `line` that can be accessed subsequently in the query.

The data about people is much less regular. The only thing known for all people is their name. For some people, you might also know their gender or age, as shown in Example 4-7.

Example 4-7. CSV data for `Person` nodes

```
name,gender,age
Rosa,f,
Karl,,64
Fred,,
```

Having irregular data (just like in the real world) means you need to take a slightly more sophisticated `LOAD CSV` approach. You'll need to change the script to be like Example 4-8. Here you can use `SET` to avoid writing any null properties into the node, which would cause the operation to fail. If you try to use the simpler approach from Example 4-6, the query will fail, complaining about `null` property values.

Example 4-8. More sophisticated `LOAD CSV` for irregular `Person` nodes

```
LOAD CSV WITH HEADERS FROM 'people.csv' AS line
MERGE (p:Person {name: line.name})
SET p.age = line.age
SET p.gender = line.gender
```

Finally, you can add the `LIVES_IN` and `FRIEND` relationships to link the new nodes into a knowledge graph (or link them to existing nodes if you have a knowledge graph already). In Example 4-9, the `FRIEND` relationships are encoded in CSV such that, for example, there is a `FRIEND` relationship from `Rosa` to `Karl` and another from `Karl` to `Fred`.

Example 4-9. `FRIEND` relationships in CSV format

```
from,to
Rosa,Karl
Karl,Rosa
Karl,Fred
Fred,Karl
```

This is a small example, so the names of the people are sufficiently unique to act as IDs. For a real import, this is unlikely to hold, and introducing unique numeric IDs for nodes is sensible. In practical terms, this would mean adding an incrementing integer ID for each row in the people and places CSV files.

You can use that data to create `FRIEND` relationships (if they don't already exist), as shown in Example 4-10. Note that because you know the `Person` nodes have already been populated in Example 4-9, you can just `MATCH` them and fill in any relationships between them using `MERGE` in Example 4-10.

Example 4-10. Loading `FRIEND` relationships

```
LOAD CSV WITH HEADERS FROM "friend_rels.csv" AS line
MATCH (p1:Person {name:line.from}), (p2:Person {name:line.to})
MERGE (p1)-[:FRIEND]->(p2)
```

The `LIVES_IN` relationships are a little trickier since they optionally have `since` properties on them. In Example 4-11, the line `Fred,London,,` has a trailing double comma, which indicates no value in the `since` field. Care is needed when handling this situation.

Example 4-11. `LIVES_IN` relationships in CSV format

```
from,to,since
Rosa,Berlin,2020
Fred,London,,
Karl,London,1980
```

To load the data from Example 4-11, you use the Cypher code in Example 4-12. Note that you are not using `MERGE (person)-[:LIVES_IN {since:line.since}]->(place)` to set the `since` property value because it would fail for those rows where the property is omitted. Once again, you should use `SET` *after* the relationship record has been created to add the property. This is perfectly fine because the statement is executed transactionally, so the relationship and its property will be written atomically despite being separated syntactically.

Example 4-12. Loading `FRIEND` relationships with property data

```
LOAD CSV WITH HEADERS FROM "friend_rels.csv" AS line
MATCH (person:Person {name:line.from}), (place:Place {city:line.to})
MERGE (person)-[r:LIVES_IN]->(place)
SET r.since=line.since
```

For very large data imports, say one million records and up, it's often sensible to break the operation into smaller batches. This ensures that the database isn't overwhelmed with large inserts and keeps the other queries on the knowledge graph running smoothly. Prior to Neo4j 4.4, you would use the `apoc.periodic.iterate` function from the APOC library for this purpose. From Neo4j 4.4, similar functionality is available directly in Cypher using `CALL {...} IN TRANSACTIONS OF ... ROWS`. Using this method you can change the loading of `Person` nodes into smaller batches, as shown in Example 4-13. In a real system, the number of rows and number of batches would both, of course, be much higher.

Example 4-13. Loading `People` nodes in batches (prefix with `:auto` if running from the Neo4j Browser)

```
LOAD CSV WITH HEADERS FROM 'people.csv' AS line
CALL {
  WITH line
  MERGE (p:Person {name: line.name})
  SET p.age = line.age
  SET p.gender = line.gender
} IN TRANSACTIONS OF 1 ROWS
```

One of the nice things about using `LOAD CSV` is that it's regular Cypher. So all the things you've learned about Cypher can be put to use, including `EXPLAIN` and `PROFILE` to analyze, debug, and tune bulk ingestion. For example, the simple query plan for loading `FRIEND` relations is shown in Figure 4-4.

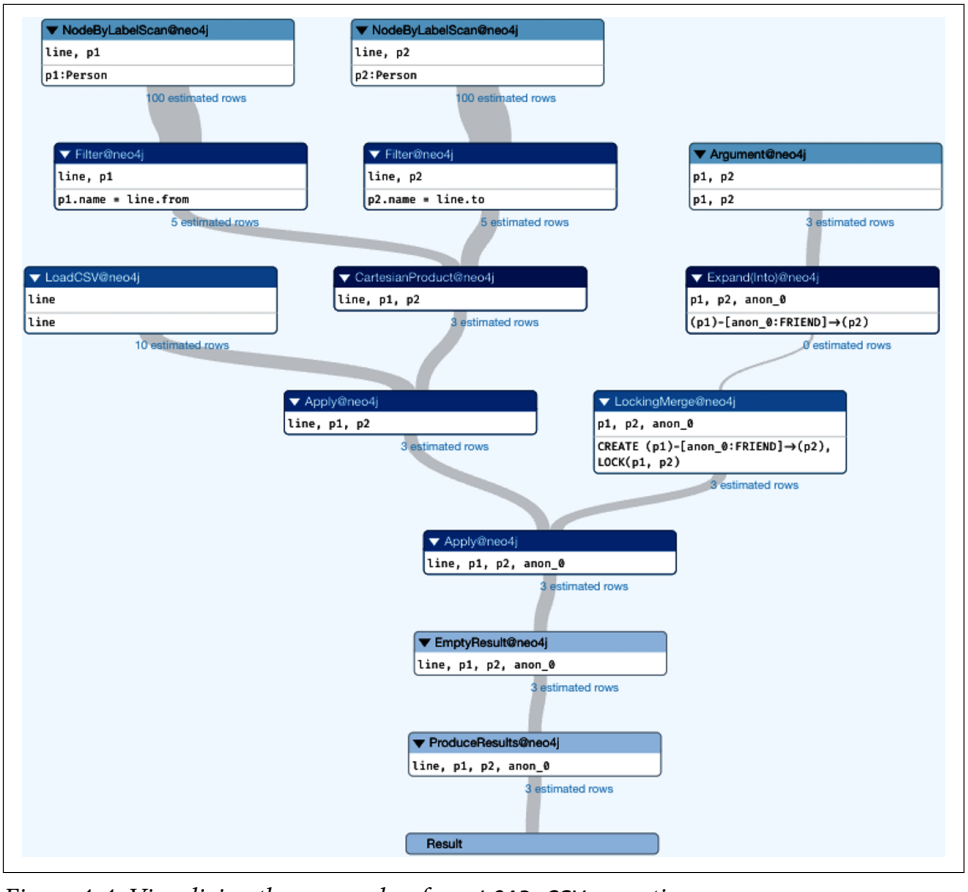


Figure 4-4. Visualizing the query plan for a `LOAD CSV` operation

Figure 4-4 looks good overall. Even the Cartesian product operator (which the Neo4j Browser will warn you about) is not a problem because it's fed by a filter that reduces its inputs to just two matching nodes and the corresponding CSV line. Of course, that might not always be the case, and using `EXPLAIN` or `PROFILE` on a small amount of representative input data can help solve any performance problems before they start.

In addition to Cartesian products, look out for any `eager` operators in the query plan or profile. Eager operators pull in all data immediately and often create a choke point. Mark Needham has an excellent blog post on strategies for removing eager operators.

`LOAD CSV` is fast and can be used at any time during the lifecycle of your knowledge graph. The database remains online during updates so that it can process other queries. It can, however, take some tuning to achieve good throughput without

dominating regular database operations. But `LOAD CSV` is not the last option for bulk inserts—you have one more to add to your toolkit.

Initial Bulk Load

Often there's an initial import that bootstraps the knowledge graph with potentially large amounts of data. It's feasible to use the Neo4j Data Importer or `LOAD CSV` to do this, but there is a faster way, albeit one that is somewhat low level.

The Neo4j command-line tool `neo4j-admin` actually has an offline importer built in. The `neo4j-admin import` command builds a new Neo4j database from a set of CSV files. (For more depth, there's a full tutorial on `neo4j-admin import`.)

The `neo4j-admin` tool ingests data very rapidly, with sustained performance of around one million records per second, and has optimizations for high-throughput devices like SSDs and storage area networks (SANs). It's a high-performance method for building your bulk imports to your knowledge graph—with the caveat that the database must be offline each time it is run.[1]

The caveat with `neo4j-admin import` is that it's an offline importer, which is why it's so fast. The database will be ready for serving knowledge graph queries as soon as the import finishes but is unavailable during building.

To use `neo4j-admin import`, you first have to assemble your data into CSV files and place them on a filesystem that the importer can access. The filesystem doesn't necessarily have to be local—it can be a network mount. It *does* have to be a filesystem since `neo4j-admin import` will not read data from, for example, S3 buckets or other data stores that are not filesystems. Since data volumes may be large, the tool also supports compressed (gzipped) CSV files.

At a minimum, the importer expects the CSV files to be presented as separate files for nodes and relationships. However, it's good practice to have separate CSV files for different kinds of nodes and relationships, for example, splitting out `Person` and `Place` nodes and `FRIEND` and `LIVES_IN` relationships.

Very large files should be split into multiple smaller CSV files with the CSV header itself in a separate file. This makes any text editing of large files much less painful since smaller files are more convenient for text editing.

1 Prior to Neo4j 5, `neo4j-admin import` could only be used to create an initial knowledge graph for the system. That constraint has been subsequently relaxed to allow multiple offline imports throughout the system lifecycle.

You don't have to make sure the CSV files are perfect, just in good enough condition. The importer will tolerate different separator characters, ignore additional columns, skip duplicates and bad relationships, and even trim strings for you. However, you'll still have to ensure there are no special characters, missing quotes, unseen byte-order marks, and the like that can affect the importer.

Start by getting your CSV files ready. Example 4-14 is a typical self-contained CSV file. It has a header row that defines the names of the columns and then rows of data. The header `:ID(Person)` says the column represents the ID of a node, and the node has the `Person` label. As with the Neo4j Data Importer, the ID isn't necessarily part of the domain model but is used by the tool to link nodes together. The `name` header declares that the second column is a property called `name` that will be written into the current node.

Example 4-14. `people.csv` contains unique IDs for `Person` nodes and a `name` property for each.

```
:ID(Person),name
23,Rosa
42,Karl
55,Fred
```

While Example 4-14 is self-contained, for larger imports it's more convenient to separate things out. For example, for `FRIEND` relationships between `Person` nodes, you can split the CSV files into a header file that describes the data and several (typically large) data files, as shown in Examples 4-15, 4-16, and 4-17. This separation makes data wrangling much easier since changes to the header don't involve opening one really large file in a text editor!

Example 4-15. `friends_header.csv` contains the signature for relationship data for the `(:Person)-[:FRIEND]→(:Person)` pattern

```
:START_ID(Person),:END_ID(Person)
```

Example 4-16. `friends1.csv` contains the first set of relationship data for `(:Person)-[:FRIEND]->(:Person)` patterns

```
23,42
42,23
```

Example 4-17. `friends2.csv` contains the second set of relationship data for the `(:Person)-[:FRIEND]→(:Person)` pattern

```
42,55
55,42
```

The importer treats Examples 4-15, 4-16, and 4-17 logically as a single unit, no matter how many files there are ultimately. You can use the same separation of concerns for nodes as well as relationships. Example 4-18 declares a CSV header for `Place` nodes which have a `Place` label and `ID`, `city`, and `country` properties. Examples 4-19 and 4-20 contain the data formatted according to the header.

Example 4-18. `places_header.csv` contains the signature for `Place` nodes

```
:ID(Place), city, country
```

Example 4-19. `places1.csv` contains data for `Place` nodes

```
143,Berlin,Germany
```

Example 4-20. `places2.csv` contains data for `Place` nodes

```
244,London,UK
```

To connect people and places, you need to have something like Example 4-21, which contains relationships starting with `Person` nodes and ending with `Place` nodes. Some, but not all, of those relationships also have `since` properties on them representing the date when the person began living in the place. The `since` property will, if present, come from the third column of the CSV file.

Example 4-21. `people_places.csv` contains data for `LIVES_IN` relationships and an optional `since` property

```
:START_ID(Person),:END_ID(Place),since
23,143,2020
55,244
42,244,1980
```

Assuming the CSV data is sufficiently correct, with no superfluous characters or dangling relationships, you can run the importer over the data to bootstrap the knowledge graph. Example 4-22 shows a command-line example that will insert data from the CSV files into a knowledge graph. Note that it's a multiline command, so be careful to include those \ characters at the end of each line.

Example 4-22. Running `neo4j-admin import`

```
bin/neo4j-admin import --nodes=Person=import/people.csv \
--relationships=FRIEND=import/friends_header.csv,import/friends1.csv,\
    import/friends2.csv \
--nodes=Place=import/places_header.csv,import/places1.csv,import/places2.csv \
--relationships=LIVES_IN=import/people_places.csv
```

The `neo4j-admin import` tool is very fast and suitable for very large datasets (containing many billions of records), with approximately a million records per second of sustained throughput. Bear in mind that large imports need large amounts of RAM and might still take hours to execute. The tool is not resumable, so be careful not to type `Ctrl-c` in the terminal window while it's running; otherwise, you'll have to start the import all over again, which is painful for long-running tasks.

Summary

At this point, you should be comfortable with the concepts of online and offline bulk imports. You've seen how each of these techniques operates and have considered their pros and cons. You should also be able to make sensible choices around which of these methods would be appropriate for the knowledge graph that you're building, including the possibility of using several tools throughout the knowledge graph's lifecycle.

From here you're going to move further up the stack and explore how data can be streamed into the knowledge graph to keep it continuously up to date and trigger further processing.

CHAPTER 5

Integrating Knowledge Graphs with Information Systems

Knowledge graphs are useful systems in their own right. But the real power of knowledge graphs is when they are integrated with other systems. When integrated, a knowledge graph will enrich other systems and in turn be enriched by them in a virtuous cycle.

There are several ways to integrate knowledge graphs into your data fabric, so this chapter will guide you through some popular choices. These range from client-side database drivers, (custom) functions and procedures, and APIs to streaming middleware and finally ETL (extract, transform, load) tools. For each of these, you'll be provided with a technical architecture sufficient to give direction and further pointers so that you can tackle implementation confidently.

Towards a Data Fabric

Data fabrics are a general-purpose, organization-wide data access layer that offers a connected view of the data in the underlying systems. A data fabric abstracts away details of how data is accessed in third-party systems and allows your users to access that data transparently via the knowledge graph. When supporting a knowledge graph, the data fabric provides an enterprise-wide index over systems so that relevant data can be consumed as a dependable layer of *master data* with *golden records* for important entities like customers, products, and events.

While the bytes-over-the-wire part of the data fabric is achieved with traditional integration methods, the informational model coalesces around a knowledge graph at the heart of the fabric. The knowledge graph's job is to provide a sophisticated index

to curate data across silos irrespective of how the data is physically stored in those silos as relational tables or NoSQL keys and values, as shown in Figure 5-1.

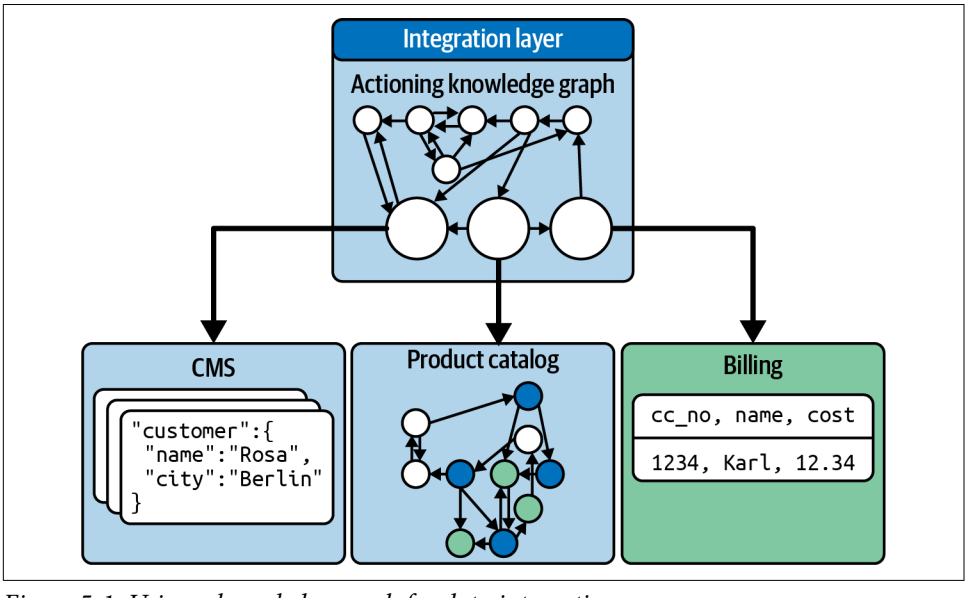


Figure 5-1. Using a knowledge graph for data integration

The knowledge graph in Figure 5-1 provides a curated index across multiple underlying data systems (which are typically unaware of the data fabric). It acts as an entry point so that when a client application or user requests "Give me the customer John Smith from Seattle," it can follow the relationships in the graph to leaf nodes, which redirect to records in other systems.

In Figure 5-1, customer records are kept in a document database, the product catalog is stored in another graph database, and billing information is stored in a relational database. All of these are accessed transparently through an integration layer, using techniques as part of the data fabric. The knowledge graph retrieves data from the underlying systems and combines it into a holistic set of data for the client, without the client having any knowledge that multiple backend systems were used.

Data architects often turn to graphs because they are flexible enough to accommodate multiple heterogeneous representations of the same entities as described by each of the source systems. With a graph, it is possible to associate underlying records incrementally as data is discovered. There is no need for big, up-front design, which serves only to hamper business agility. This is important because data fabric integration is not a one-off effort, and a graph data model remains flexible over the lifetime of the data domains.

To improve the quality and correctness of the data and increase interoperability across the enterprise, you can overlay an organizing principle, such as an ontology, taxonomy, or enterprise canonical model, atop the integrated data and present it to consumers as a knowledge graph. The organizing principle makes it possible to validate data across systems and check for inconsistencies or violations of the semantics of the data.

Curating your data with a knowledge graph supported by a data fabric has significant benefits. Characteristics like node degrees, neighborhood information, and centrality metrics can be used to build sophisticated and effective data integrations. For example, you can define a matching rule as follows: two nodes represent the same product and therefore can be deduplicated, if their names have a strong string similarity (say over 95%) and they have identical clustering coefficients (see Chapter 6). The result is that data owners get significantly improved matching rates when aligning multiple data sources using a knowledge graph.

But to achieve these goals, you need to deploy a data fabric to integrate your knowledge graph with underlying systems. It's time to examine some implementation options.

The Database Driver

The most commonly used way of integrating with a knowledge graph is through a *driver*. A driver is a piece of client-side middleware that sits underneath your application code and makes network calls to the knowledge graph, integrating the two parts of the system.

In fact, you've already seen drivers in action in this book. Recall that in Chapters 3 and 4 you saw multiple examples of interacting with the database using the Cypher query language via the Neo4j Browser tool. Neo4j Browser provides a REPL (read-evaluate-print loop) console through which you can enter Cypher queries for evaluation by the database.

It might look as if the Browser is directly connected to the database, but that's not the case. There's a connection string which the Browser uses to establish an instance of a *Neo4j driver* through which it communicates over the network (even if it's the local network interface) to the Neo4j server that hosts the knowledge graph. Neo4j drivers implement a client-server architecture between a knowledge graph and its consuming systems (including human users), as shown in Figure 5-2.

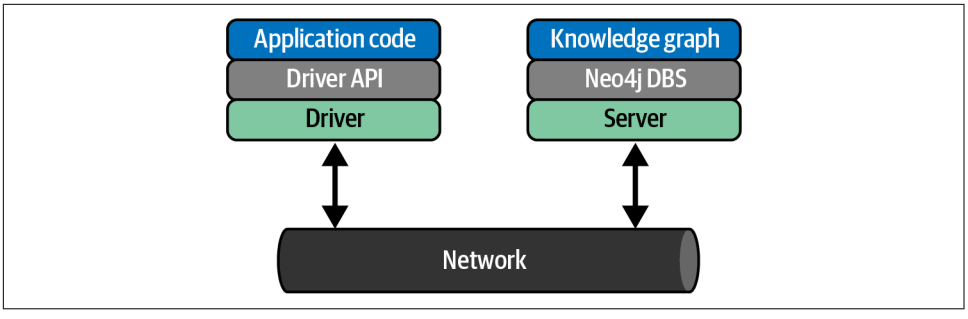


Figure 5-2. A high-level view of the client-server architecture connecting the knowledge graph to other systems via the network

In the general case, a Neo4j driver is a piece of client-side middleware that allows your application or microservice code to connect to the Neo4j server which hosts your knowledge graph. In the case of the Neo4j Browser, the driver happens to be written in JavaScript. But there are numerous other drivers from which to choose based on your language and platform preferences.

Neo4j Drivers

Neo4j provides several language drivers for common programming languages and frameworks:

- Java
- Spring Framework and optional Object-Graph Mapper (Java)
- .NET
- JavaScript
- Python
- Go

In addition, there are several well-maintained drivers written by the Neo4j community:

- Ruby
- PHP
- Erlang/Elixir
- Perl
- C/C++
- Clojure
- Haskell
- R
- Py2Neo Object-Graph Mapper (Python)

You choose the driver for your preferred language (or closest, such as the Java driver for other Java Virtual Machine languages). Each driver provides the same functionality: connecting your application code to Neo4j (including clusters of Neo4j servers and the AuraDB cloud service) via the Bolt protocol or HTTP.

The best way to understand how to use Neo4j drivers to interact with your knowledge graph is by example. Example 5-1 shows a typical example using the Neo4j Java driver (noting that other languages follow a similar pattern).

Example 5-1. Using the Neo4j Java driver

```
import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.driver.Result;
import org.neo4j.driver.Session;

import static org.neo4j.driver.Values.parameters;

public class JavaDriverExample implements AutoCloseable
{
    // Driver instances are quite expensive.
    // You typically want just one of these for your whole application lifetime.
    private final Driver driver;

    public JavaDriverExample(String uri, String user, String password )
    {
        // 1. Driver objects are expensive
        driver = GraphDatabase.driver( uri, AuthTokens.basic( user, password ) );
    }

    @Override
    public void close() {
        driver.close();
    }

    public Result findFriends(final String name)
    {
        // 2. Sessions are cheap. Use them whenever you need them.
        try ( Session session = driver.session() )
        {
            // 3. Telling the driver whether the query updates the graph (or not)
            // helps to make good routing decisions in a cluster.
            // Session#writeTransaction is also available.
            return session.readTransaction(tx ->
            {
                Result result = tx.run("MATCH (a:Person)-[:FRIEND]->(b:Person) " +
                                       "WHERE a.name = $name " + // 4. Parameters are good
                                       "RETURN b.name",
                        parameters("name", name));
                return result;
            });
        }
    }

    public static void main( String... args ) throws Exception
```

```
    {
        try ( JavaDriverExample example =
                      new JavaDriverExample( "bolt://localhost:7687",
                        "neo4j",
                        "password" ) )
        {
            example.findFriends( "Rosa" );
        }
    }
}
```

There are four important things in Example 5-1 to keep in mind when you're using any of the Neo4j drivers:

1. *Drivers objects are expensive.* You should only have one of these for the lifetime of the application connecting to the knowledge graph. It's expensive because it deals with setting up (and tearing down) network connections, including authentication. You certainly don't want to be paying that price on a per-query basis.
2. *Session objects are cheap.* They're also quite sophisticated, including various aysnc APIs to help smooth processing on behalf of clients. They also support causal boundaries (also known as *bookmarks*) so that clients always see *at least* their own writes even on a globally distributed cluster.
3. *Tell the receiving server whether a query is read-only or read/write.* This helps a cluster make better routing decisions about where queries should be processed to maintain high throughput.
4. *Parameterize all queries.* Parameterized queries don't need to be reparsed or have new query plans generated by the database. They're faster than just using literal strings across the network. Instead of concatenating strings like `MATCH (:Person {name:'Karl'})`, use `MATCH (n:Person {name: $name})` and send parameters separately. Parameterizing queries also helps protect against injection attacks.

From Neo4j 5 onward, Cypher will try to automatically parameterize queries, even if they are not constructed that way. It's still good practice to identify all parameters to your query, in case the parser cannot.

The short example in Example 5-1 shows how easy it is to build applications or microservices to interact with the knowledge graph over the network. But you can also use logic that is deployed on the database server itself.

Graph Federation with Composite Databases

Neo4j Composite Databases enable you to bring together multiple graph data sources and provide a single unified point of access to all of them. This technique, called *federation*, does not physically combine the databases but creates a virtual composite. The consumer application is unaware that data is drawn from multiple sources.

Figure 5-3 shows three independent graph databases, one with a product catalog and two with sales information. The two sales databases use the same graph data model, but one of them contains orders from Europe, Middle East, and Africa (EMEA) customers, and the other contains orders from Asia-Pacific (APAC) customers.

Example 5-2. A logical database called `globalsales` integrates multiple physical databases

```
# Run in Cypher shell or one line at a time in Neo4j browser
CREATE DATABASE db1;

CREATE DATABASE db2;

CREATE COMPOSITE DATABASE globalsales;

CREATE ALIAS emeasales
  FOR DATABASE db1;

CREATE ALIAS apacsales
  FOR DATABASE db2;
```

Neo4j Composite Databases allow you to retrieve data from all federated databases with a single Cypher query. The query in Example 5-3 returns all customers who placed more than 100,000 orders.

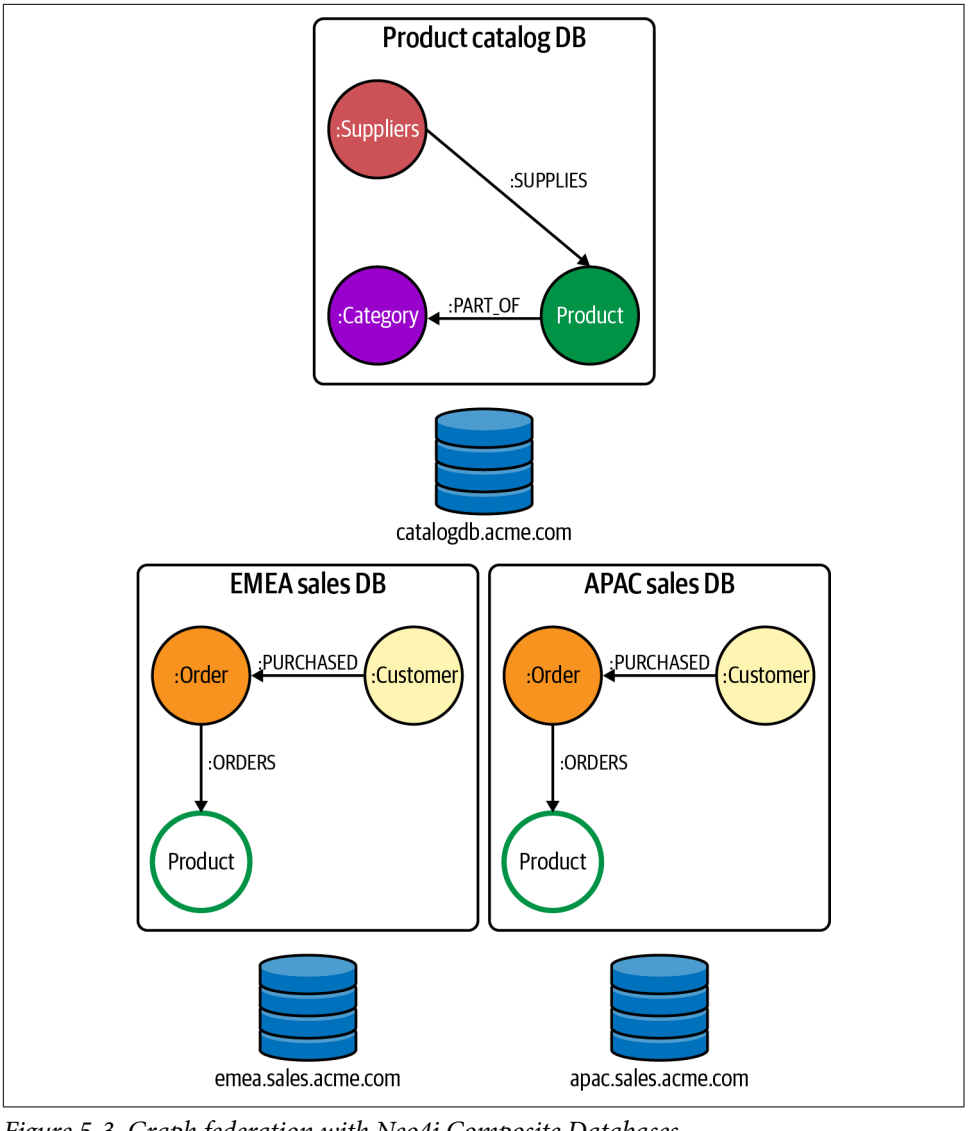


Figure 5-3. Graph federation with Neo4j Composite Databases

To create a composite database using Neo4j Composite Databases, add code like Example 5-2 to the *neo4j.conf* file. This example creates a composite database called `globalsales`.

Example 5-3. A single Cypher query retrieves all customers who placed more than 100,000 orders

```
UNWIND ['globalsales.apac', 'globalsales.emea'] AS g
CALL {
  USE graph.byName(g)
  MATCH (c:Customer)-[:PURCHASED]->()-[o:ORDERS]->(:Product)
  WITH c, sum(o.quantity * o.unitPrice) AS totalOrdered
  WHERE totalOrdered > 100000
  RETURN c.customerID AS name, c.country AS country, totalOrdered
}
RETURN name, country, totalOrdered
```

The query in Example 5-3 federates EMEA and APAC sales from separate underlying graphs to provide an aggregated view to the consuming application. You can easily extend that to include the product catalog. In Example 5-4, the query returns a list of global customers who have purchased products of a given category, such as `Beverages`.

Example 5-4. A query that returns a list of global customers who purchased beverages from multiple underlying graphs

```
CALL {
  USE globalsales.catalog
  MATCH (p:Product{discontinued:false})-[:PART_OF]->(:Category{categoryName:'Beverages'})
    RETURN COLLECT(p.productID) AS pids
}
WITH pids, [g IN fabricnw.graphIds()] AS gids
UNWIND gids AS gid
CALL {
    USE globalsales.graph(gid)
        MATCH (c:Customer)-[:PURCHASED]->()-[o:ORDERS]->(p:Product)
    WHERE p.productID
    RETURN distinct c.customerID AS name, c.country country, p.productName AS product
}
RETURN name, country, product
```

Aside from the graph itself, the most compelling aspect of Example 5-4 is that the implementation complexity is abstracted away from the user. The fact that (at least) two underlying graph databases are composed to a single virtual database is handled by Neo4j Fabric, leaving you to focus on your domain.

Server-Side Procedures

Client-server interactions using Cypher and Neo4j drivers are commonplace. But sometimes your needs aren't wholly met by that paradigm and you need specific logic to run close to the knowledge graph on the server (like a stored procedure in SQL databases).

Neo4j has an excellent library of functions and procedures that come with the database (e.g., for managing users and roles, visualizing the inferred knowledge graph schema, or doing data science operations, as you'll see in Chapter 6). APOC, as you saw in Chapter 4, has lots of different functions and procedures you can call on to help with common and not-so-common cases.

Since knowledge graphs are better when they are integrated with other systems, why not use use APOC to make calls into a SQL database to enrich the knowledge graph data at query time? Using the pattern shown in Example 5-5, you can extract data from a SQL database as part of a Cypher query and have that data mixed in with the knowledge graph data. No copying required.

Example 5-5. Calling to a SQL database from a procedure

```
CALL apoc.load.driver("com.mysql.jdbc.Driver");

WITH "select firstname, lastname from employees
      where firstname like ? and lastname like ?" AS sql

CALL apoc.load.jdbcParams("northwind", sql, ['F%', '%w'])
YIELD row

MATCH (row)-[:WORKS_FOR*1..3]->(boss:Person)
RETURN (boss)
```

The Cypher code in Example 5-5 loads a SQL database driver into Neo4j and then uses it to access a SQL database called `northwind`. The code runs a SQL query against `northwind`, which retrieves the first and last names of employees in the database and returns them as nodes in Cypher. From here, you can mix those nodes into regular Cypher queries on the graph. In this case, the query tries to find who the person from the SQL database works for in the knowledge graph, up to three levels up.

Using procedures, you can enrich the knowledge graph with data from other databases at query time without having to move or copy that data from its source system. No need to rip and replace, but you do need to take the latency (and reliability) of the remote call into consideration since it will add to the latency of your knowledge graph query.

Example 5-6 shows an interoperability procedure from the APOC library to extract some data from MongoDB (a store which itself has no knowledge graph capabilities). The intent is to enrich your knowledge graph with data from the other system. The `CALL` deals with connecting to the remote database, and returns data from a MongoDB query in a format suitable for further Cypher processing. Specifically you take the `id` of a document in MongoDB and `MATCH` it to a node in the knowledge

graph. With that data, you then search at depth 1 or 2 out from the matching `experiment` node in the knowledge graph to try to find any environmental contaminants that may be present. The query then returns those contaminants to the caller.

Example 5-6. Using APOC to access data in a different database

```
CALL apoc.mongodb.find('mongodb://mongo:neo4j@mongo:27017', 'results',
    '2022-02-22-wetlab', {`status`: "failed"}, null, null)
YIELD failed_experiments
MATCH (:Experiment { id, failed_experiments.id})-[*1..2]->(environmental_factors)
RETURN environmental_factors.contaminants AS contaminants
```

Now that you have seen the principle established, it's not hard to imagine how you might also bring in data from other sources, not just other databases. APOC has a simple solution to load JSON data from a file or web page (e.g., a file stored in Amazon S3), which you can see in Example 5-7.

Example 5-7. Calling a JSON Web API from a procedure

```
WITH "https://example.org/karl.json" AS url
CALL apoc.load.json(url) YIELD value
UNWIND value.products AS product

WITH product

MATCH (k:Person {name:'Karl'})
MERGE (k)-[:BOUGHT]->(:Product {name:product.name,
                                price:product.price,
                                description:product.description})
```

Like the equivalent SQL example in Example 5-5, Example 5-7 shows how you can consume JSON-formatted data easily in the context of a query running on a knowledge graph. In this specific example, you use that JSON data to enrich the underlying graph via `MERGE` with people who have bought products.

Data Virtualization with Neo4j APOC

Sometimes you won't have the luxury of combining underlying graph databases into a single virtual knowledge graph. If your underlying data is in different nongraph sources, like time-series data, clickstream data, or log data, you're going to have to use a different approach.

Fortunately, the APOC library supports the definition of a catalog of *virtual resources*. A virtual resource is an external data source that Neo4j can use to query and retrieve data on demand, presenting it as virtual nodes enriching the data stored in the graph, as shown in Figure 5-4.

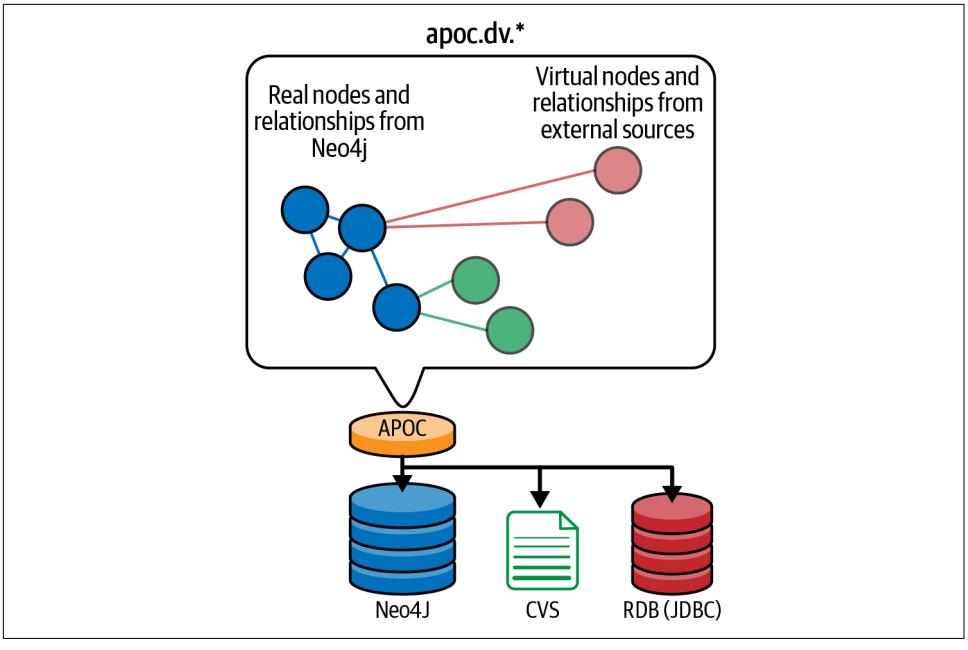


Figure 5-4. Virtual nodes and relationships mapping to external data sources

Figure 5-4 shows the architecture for the system. The library call sits as an intermediary between the end user and the databases they want to use, which include a Neo4j graph, a CSV file, and a relational database. As queries are executed on the graph database, the APOC library retrieves data from the CSV file and relational database and presents it as virtual nodes and relationships to the query engine where they are mixed in with the native graph data from Neo4j.

Bear in mind that accessing data from CSV files or other databases will usually have higher latency than accessing the local knowledge graph. Figure 5-4 and similar are useful insofar as they hide the difference between graph and nongraph data and allow seamless enrichment of your knowledge graph without having to take copies of other data. But you can't hide latency, so be judicious how you use it.

As an example, using the approach in Figure 5-4, you can build a digital twin of a telecom network, as shown in Example 5-8. The graph contains a representation of the network, with all of the physical, logical, and virtualized devices and the connections between them. The Cypher fragment in Example 5-8 shows how the digital twin's graph is built.

Example 5-8. Building a digital twin of a telecom network

```
// network topology creation
// adding network devices
MERGE (nsm:NetworkDevice { name: "Nogent sur Marne", nd_id: "N08",
  pos: point({latitude:2.46995, longitude:48.834374}), gtype: 1, code: "nog-1"  });
MERGE (ms:NetworkDevice { name: "Montrouge Sud", nd_id: "N11",
  pos: point({latitude:2.316613, longitude:48.805473}), gtype: 1, code: "mon-3"});
...
// adding links betweeen network devices
MATCH (nsm:NetworkDevice { code: "nog-1" })
MATCH (ms:NetworkDevice { code: "mon-3" })
MERGE (nsm)-[l:LINK]->(ms)
SET l= { linkId: "s1_385/ol", bundleId: "s1_128/ob",
  linkType: "OTN Line", capacity: 1920 };
```

If you run the code in Example 5-8, it creates the top-level graph structure for the digital twin. Figure 5-5 shows a visualization of its initial state.

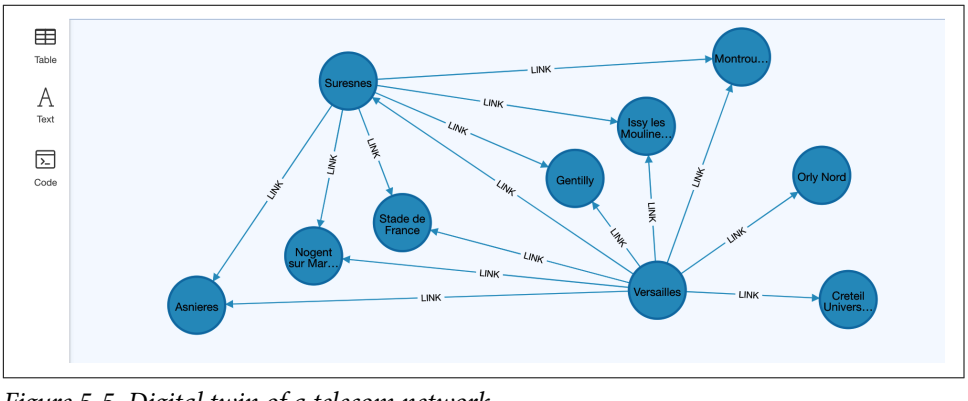


Figure 5-5. Digital twin of a telecom network

Though the knowledge graph in Figure 5-5 is functionally incomplete, it is already of some use. For example, the Cypher query shown in Example 5-9 retrieves the two endpoints of a failing network link.

Example 5-9. Retrieving the endpoints for a failing link

```
MATCH  (aend:NetworkDevice)-[l:LINK]->(zend:NetworkDevice)
WHERE l.linkId = "s1_385/ol"
RETURN aend.code, zend.code
```

For a simple root cause analysis, you correlate the failure of a link with the recent performance metrics of the two endpoints. But this is a static view of the system. To make a functionally complete system, you need to add real-time performance metrics for each network device to enrich the model. But those performance metrics aren't

in the graph, and probably shouldn't be since they are rapidly changing counters on hardware deployed into the physical telecommunications network.

In this case, you use a procedure from APOC to combine data from an external system (a SQL-queriable database) with the knowledge graph's topological data. The SQL query in Example 5-10 returns the performance metrics for the last 24 hours. It queries the `device_perf` table in the `metrics_collector` database, which is indexed by `device_id` and by `timestamp`.

Example 5-10. A SQL query that returns performance metrics for the last 24 hours

```
SELECT latency, packetloss, packetdupl, jitter, throughput
FROM device_perf
    WHERE device_id = $device_id
          AND timestamp > now() - interval '1 day' ;
```

You can create a virtualized resource to handle this integration between the knowledge graph and the other database. To do so, invoke the the `apoc.dv.catalog.add` stored procedure, then specify the SQL query to be executed. Creating the virtualized resource is shown in Example 5-11.

Example 5-11. Creating a virtualized resource using APOC

```
CALL apoc.dv.catalog.add("metrics-by-device-id", {
  type: "JDBC",
  url: "jdbc:postgresql://localhost/metrics_collector?user=bob&password=bobby",
  labels: ["Town","PopulatedPlace"],
  query: "SELECT latency, packetloss, packetdupl, jitter, throughput
            FROM device_perf
            where device_id = $device_id
                  and timestamp > now() - interval '1 day' ;",
  desc: "french towns by department number"
})
```

Now that you have created a virtualized resource, you can use it. Example 5-12 shows how to use the virtualized resource created in Example 5-11 to retrieve data from the metrics database on demand.

Example 5-12. Querying the virtualized resource using APOC

```
MATCH (aend:NetworkDevice)-[l:LINK]->(zend:NetworkDevice)
WHERE l.linkId = "s1_385/ol"
WITH aend, zend
CALL apoc.dv.query("metrics-by-device-id",{device_id: aend.nd_id}) YIELD node
RETURN aend.code, properties(node), zend.code
```

The query results combine data from the graph (topology information) with data retrieved on demand from the performance metrics database. The knowledge graph acts as a logical integration layer (a data fabric) offering users and consumer apps a unified view of the network and real-time metrics in the form of a digital twin. While you'll be aware of the technical implementation, users just see a single holistic knowledge graph with transparently up-to-date data.

Custom Functions and Procedures

The existing library of Cypher procedures is extensive, both the built-in Neo4j procedures and the APOC library. But sometimes you need to operate on your knowledge graph in a highly domain-specialized fashion, which isn't covered by Cypher or other procedure libraries.

Fortunately, Cypher is extensible. It allows both user-defined functions and user-defined procedures, so you can build functions and procedures to suit your own exacting needs.

User-defined functions and procedures in Neo4j do not carry the same stigma as stored procedures in SQL. You write JVM (Java, Scala, etc.) code, which can be tested independently of the database. You can adopt the same version control and build practices that you use for your solution as a whole so that the custom code is robust.

In Figure 5-6 you can see how user-defined functions and procedures (including the built-in ones) fit into (a simplified version of) the Neo4j architecture. Those user-defined functions and procedures are written in Java or another JVM language and invoked from Cypher.

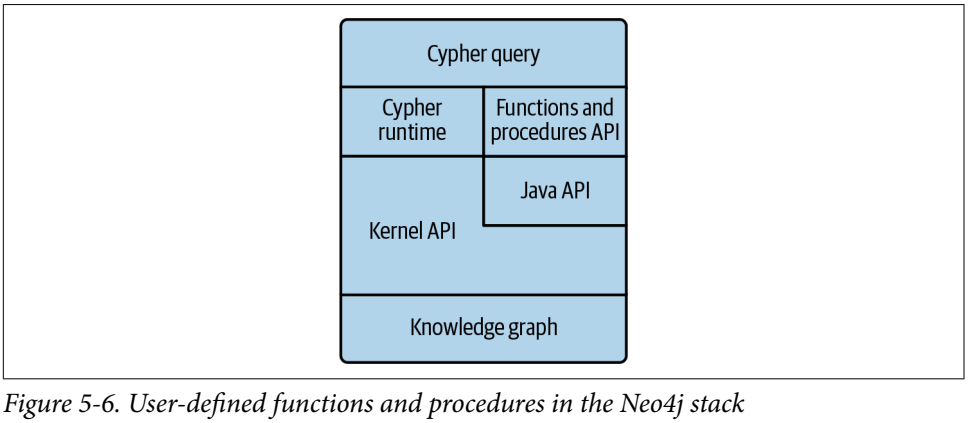


Figure 5-6. User-defined functions and procedures in the Neo4j stack

Unlike plain Cypher where the Cypher query is executed against the kernel (low-level) API from the Cypher runtime, user-defined functions and procedures are defined using Neo4j's Java API (a friendlier level of abstraction). By decorating your custom functions and procedures with the correct annotations and implementing the correct interfaces implied, you can create a procedure or function with arbitrary functionality.

For example, you might want a procedure that inspects, computes over, or even updates the knowledge graph stored in Neo4j. To demonstrate how straightforward it is to build such a procedure, Example 5-13 shows a simple algorithm for counting the degree (number of attached relationships) of all nodes in the knowledge graph.

Example 5-13. A custom procedure for creating a histogram of node degree

```
public class GetNodeDegrees {

    @Context
    public GraphDatabaseService graphDatabaseService;

    @Procedure(value = "getNodeDegrees")
    @Description("Get degrees of all nodes in the knowledge graph")
    public Stream<NodeDegree> getNodeDegrees() {

        Map<Long, Long> nodeDegrees = new TreeMap<>(Long::compare);

        graphDatabaseService.getAllNodes().forEach(n -> {
            final long nodeDegree = n.getDegree();
            long numberOfNodeWithCurrentDegree =
                                nodeDegrees.getOrDefault(nodeDegree, 0L);
            nodeDegrees.put(nodeDegree, numberOfNodeWithCurrentDegree + 1);
        });

        final Stream.Builder<NodeDegree> streamBuilder = Stream.builder();
        for (Map.Entry<Long, Long> entry : nodeDegrees.entrySet()) {
            streamBuilder.add(new NodeDegree(entry.getKey(), entry.getValue()));
        }

        return streamBuilder.build();
    }

    public class NodeDegree {
        public Long degree;
        public Long numberOfNodes;

        public NodeDegree(Long key, Long value) {
            degree = key;
            numberOfNodes = value;
        }
    }
}
```

Most of the code in Example 5-13 is logic that counts relationships incident on nodes. The more interesting parts are where the procedure binds to the underlying knowledge graph. For instance, you get access to the underlying database that hosts the knowledge graph by using the `@Context` annotation and `public GraphDatabaseService graphDatabaseService` field, which ensures that a reference to the database management system API is injected into your procedure before your code runs. You declare the entry point to your procedure with the `@Procedure` annotation with its friendly name `getNodeDegrees` and provide a friendly description with `@Description`. Finally, remember that you must return a `Stream` of a record type, like `NodeDegree` in this case.

The procedure in Example 5-13 is global, as it touches all nodes (and relationships) in the knowledge graph. If you want to write a procedure that starts at a specific node, then you don't need to inject `GraphDatabaseService` but can instead have parameters for specific nodes or relationships on the procedure method, such as `public Stream<RelationshipTypes> getRelationshipTypes(@Name("currentNode") Node node) {...}`.

Back in Example 5-7 you read in data file from a web- or filesystem-hosted JSON file and enriched the knowledge graph. That works fine for simple JSON retrieval services, but most APIs are more complicated. However, using the same pattern from Example 5-13, you can write a custom procedure that accesses service APIs too. Remember that a custom procedure can be any code you like, provided it conforms to the Neo4j procedure interface.

Data Fabric with Specialized Data Virtualization Platforms

Specialized platforms like Dremio, Denodo, Data Virtuality, and others provide the capabilities to deploy broader data fabrics built on data virtualization. Such solutions support connectors for multiple data sources, advanced query delegation, caching, and a GUI for designing logical views. When deployed alongside such specialized data fabric technology, the knowledge graph will act as one master source of data being integrated in the fabric for golden record, deduplicaton, and so forth.

Complementary Tools and Techniques

While the Neo4j drivers, functions, and procedures are popular ways of connecting to a knowledge graph, other integration tools might be used, depending on the surrounding systems. For example, you might have to provide an API to the knowledge graph so that it can be consumed from a wide range of user-facing applications. Or

you might need to run ETL tasks to take data from upstream systems and use it to enrich the knowledge graph. You may want to move data between the knowledge graph and your analytics systems or have real-time data streamed into or out of the knowledge graph. Fortunately, there is a rich set of tools available for each of these requirements. The following sections provide an overview of them, so you can implement the right ones for your system with confidence.

GraphQL

GraphQL is an API toolkit that originated from Facebook. Its original purpose was to provide an API to the underlying social graph stored in TAO (Facebook's social graph service). Today, GraphQL is a framework for API construction (that defines types, schemas, messages, and so forth) and a runtime environment (GraphQL servers) for executing API calls. While it has "graph" in its name and is certainly useful for exposing knowledge graphs to consumers, it is a general-purpose toolkit that can be happily used for nongraph systems too. The basic system architecture is a simple client-server setup, as shown in Figure 5-7.

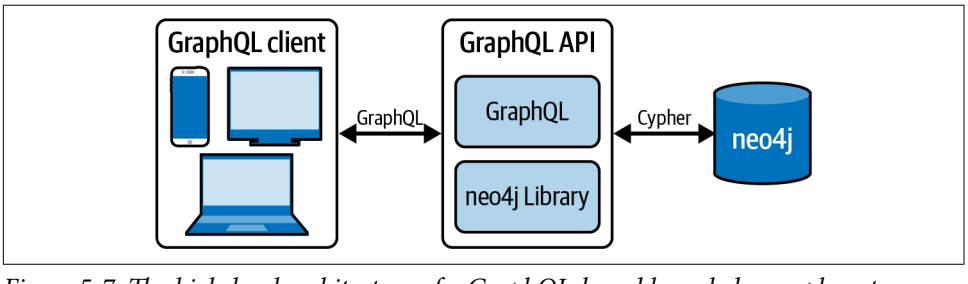


Figure 5-7. The high-level architecture of a GraphQL-based knowledge graph system

Using GraphQL to access a knowledge graph stored in a native graph database offers several advantages. These include a consistent graph data model and increased developer productivity, particularly when GraphQL is widely used across the enterprise.

Despite its name, GraphQL doesn't come with any built-in affordances for using the topologies in knowledge graphs. However, in Neo4j's GraphQL implementation, you can use the `@relationship` directive to describe relationships between nodes. For instance, based on the people and places examples in Chapter 4, you can create a GraphQL type system for an API that will allow users to query for who lives where. You can see this in Example 5-14.

Example 5-14. Using `@relationship` to create a GraphQL type system that includes links between records

```
type Place {
    city: String!
```

```
    country: String
    people: [Person!]! @relationship(type: "LIVES_IN", direction: IN)
  }

  type Person {
    name: String!
    home: Place! @relationship(type: "LIVES_IN", direction: OUT)
  }
```

Using `@relationship` in Example 5-14 has enriched an otherwise simple schema of documents with the ability to store data as knowledge graphs. Along with other data like `city` and `country`, a `Place` can have a non-nullable array of non-nullable `Person` records connected via `LIVES_IN` relationship records. Conversely, a `Person` has a `name` and a non-nullable `LIVES_IN` relationship to a `Place`.

Intuitively, you can see how Example 5-14 cleanly maps onto an underlying knowledge graph. But importantly, it provides strict guidance to the caller on the nature of the data that can be retrieved through the API.

To query or update the underlying graph, you can now send GraphQL from your GraphQL client to the server. In Example 5-15 you can see the GraphQL payload (query) sent from client to server.

Example 5-15. Updating a knowledge graph via GraphQL

```
mutation {
  createPlaces(
    input: {
      city: "Sydney"
      country: "Australia"
      people: {
        create: [
          { node: { name: "Skippy" } }
          { node: { name: "Cate Blanchet" } }
        ]
      }
    }
  ) {
    places {
      city
      country
      people {
        name
      }
    }
  }
}
```

GraphQL returns the response shown in Example 5-16.

Example 5-16. Response from a knowledge graph update via GraphQL

```
{
  "data": {
    "createPlaces": {
      "places": [
        {
          "city": "Sydney",
          "country": "Australia",
          "people": [
            {
              "name": "Skippy"
            },
            {
              "name": "Cate Blanchet"
            }
          ]
        }
      ]
    }
  }
}
```

As an API description/construction framework, GraphQL works very well with knowledge graphs, particularly where the knowledge graphs sit near the user in a system architecture. For an in-depth book on integrating knowledge graphs with GraphQL, see *Full Stack GraphQL Applications With React, Node.js, and Neo4j* by William Lyon (Manning).

Kafka Connect Plug-In

Apache Kafka is a popular enterprise middleware platform, acting as the glue between systems. At its simplest, it is a reliable publish/subscribe system for moving messages between systems at high volume. At its most sophisticated, it is itself a database and query engine for processing records as they flow through the business.

Integrating knowledge graphs with Kafka is straightforward. Using Kafka Connect Neo4j Connector, your knowledge graph can serve as a source of data, such as change data (or change data capture), or as a sink to ingest data from Kafka events into your knowledge graph.

At a systems level, the architecture is straightforward. Figure 5-8 shows how a query is executed periodically over the knowledge graph to publish data from your knowledge graph into Kafka for consumption by downstream systems.

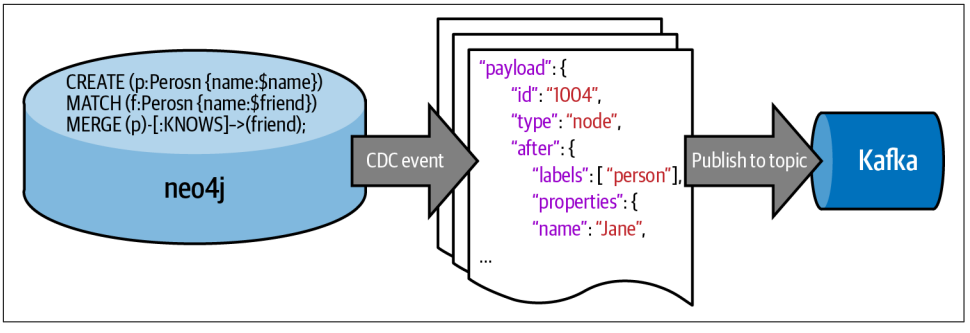


Figure 5-8. Periodically executing a Cypher query over a knowledge graph to publish data to Kafka

Configuring the Neo4j Streams plug-in to publish updates from a knowledge graph is also straightforward. In Example 5-17 a knowledge graph is set up to publish the details of any new arrivals to Berlin once every 5 seconds (5,000 ms as it is written in the configuration). Correspondingly, in the model you must change the original `since` property on `LIVES_IN` relationships from Chapter 3 to a more granular timestamp that meets *both* the needs of the model *and* the publishing mechanism. Then the Cypher query will be run periodically to find the patterns like `(Person)-[:LIVES_IN {since:'2015-06-24T12:50:35.556+0100'}]->(:Place {city:'Berlin'})` from which you can easily return the person's name and the timestamp when they moved. The full query is shown along with the other necessary configuration in Example 5-17.

Example 5-17. Configuration for a query to periodically execute over a knowledge graph, publishing results to Kafka

```
{
  "name": "Neo4jSourceConnectorAVRO",
  "config": {
    "topic": "new-arrivals-in-Berlin",

    ...

    "neo4j.streaming.poll.interval.msecs": 5000,
    "neo4j.streaming.property": "timestamp",
    "neo4j.streaming.from": "NOW",
    "neo4j.enforce.schema": true,
    "neo4j.source.query": "MATCH (p:Person)-[li:LIVES_IN]->
                           (:Place {city:'Berlin', country:'DE'})
                          WHERE li.timestamp > $lastCheck
                          RETURN p.name AS name,
                                 li.timestamp AS since"
  }
}
```

Both architecturally and practically, it's just as straightforward to ingest data into your knowledge graph from Kafka. Figure 5-9 shows the general approach.

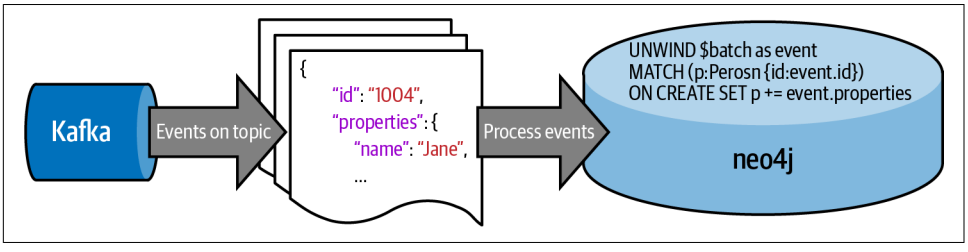


Figure 5-9. Receiving messages from Kafka and using Cypher `MERGE` to insert data into the knowledge graph

At a more detailed level, you use Cypher to unpack Kafka messages into the knowledge graph. In Example 5-18 you can see an example of the kind of patterns into which data from Kafka messages can be decomposed. You subscribe to a particular topic (e.g., `new-arrivals-in-Berlin`), which gives you access to data in messages published to it. You access data in the messages through the `event` variable and use that data to enrich your knowledge graph using Cypher features like `MERGE`.

Example 5-18. Configuration to ingest data from a Kafka topic into your knowledge graph

```
{
  "name": "Neo4jSinkConnector",
  "config": {
    "topics": "new-arrivals-in-Berlin",

    ...

    "neo4j.topic.cypher.new-arrivals-in-Berlin":
        "MERGE (person:Person {name: event.name})
        MERGE (place:Place {city: event.city, country: event.country})
        MERGE (person)-[LIVES_IN {since:'2021-06-12T10:31:11.553+0100'}]->(place)"
  }
}
```

There are of course other operational details, such as how to set up Kafka and install the Streams plug-in into Neo4j, which you will need to solve to use this technique. The Neo4j Kafka documentation can help.

Neo4j Spark Connector

Apache Spark is a (distributed) data processing framework for working with large data sets. It's a common enterprise tool that integrates well with databases and filesystems as data sources and sinks for (parallel) processing. Its developer-friendly API allows for complex data-processing operations to express succinctly as a sequence of stages whose execution can be farmed out to a compute cluster without requiring specialist distributed processing skills.

Unsurprisingly, there are connectors that allow Spark to treat your knowledge graph as both a source and a sink for data. Using the connectors, Spark developers are able to read in data from a knowledge graph for use in their processing pipelines as well as write back to the knowledge graph any results from those computations.

Installing the Neo4j Connector for Apache Spark connector is simply a matter of copying the appropriate *.jar* file into your Spark environment. Once the connector is installed, you can perform operations on the underlying graph. The simplest operations are reading and writing to the graph by label and predicate, as shown in Example 5-19. Leaving out the predicate `where` statement is fine too, but more data will be exchanged between Spark and Neo4j.

Example 5-19. Reading into Spark from Neo4j by label and predicate

```
import org.apache.spark.sql.{SaveMode, SparkSession}

val spark = SparkSession.builder().getOrCreate()

val df = spark.read.format("org.neo4j.spark.DataSource")
  .option("url", "bolt://localhost:7687")
  .option("authentication.basic.username", "neo4j")
  .option("authentication.basic.password", "neo4j")
  .option("labels", ":Person")
  .load()

df.where("name = 'John Doe'").where("age = 32").show()
```

It's also possible for Spark to perform a Cypher read query over your knowledge graph. You can see this in Example 5-20 where one `option` is a `query` whose value is the Cypher query to be executed. Note that it is good practice to return just data values from the query rather than full nodes and relationships to minimize the work involved in data transfer.

Example 5-20. Reading into Spark from Neo4j by Cypher query

```
import org.apache.spark.sql.{SaveMode, SparkSession}
```

```
val spark = SparkSession.builder().getOrCreate()

spark.read.format("org.neo4j.spark.DataSource")
  .option("url", "bolt://localhost:7687")
  .option("query", "MATCH (n:Person)-[FOLLOWS]->
                    (:Person name:'emileifrem')
                    WITH n LIMIT 20
                    RETURN id(n) AS id, n.name AS name")
  .load()
  .show()
```

For writes there is a similar idiom. You can insert nodes by label into the knowledge graph, as shown in Example 5-21.

Example 5-21. Writing from Spark into Neo4j by node label

```
import org.apache.spark.sql.{SaveMode, SparkSession}
import scala.util.Random

val spark = SparkSession.builder().getOrCreate()
import spark.implicits._

case class Person(name: String, surname: String, age: Int)

val ds = (
    // Populate the data source here
).toDS()

ds.write
  .format("org.neo4j.spark.DataSource")
  .mode(SaveMode.ErrorIfExists)
  .option("url", "bolt://localhost:7687")
  .option("labels", ":Person")
  .save()
```

You can insert into Neo4j using Cypher's patterns too, of course, which is shown in Example 5-22.

Example 5-22. Writing from Spark into Neo4j by Cypher query

```
import org.apache.spark.sql.{SaveMode, SparkSession}

val spark = SparkSession.builder().getOrCreate()

// Populate data frame in Spark
val df = ...

// Save contents to knowledge graph
df.write
  .format("org.neo4j.spark.DataSource")
```

```
  .option("url", "bolt://second.host.com:7687")
  .option("relationship", "FOLLOWS")
  .option("relationship.source.labels", ":Person")
  .option("relationship.source.save.mode", "Overwrite")
  .option("relationship.source.node.keys", "source.name:name")
  .option("relationship.target.labels", ":Person")
  .option("relationship.target.save.mode", "Overwrite")
  .option("relationship.target.node.keys", "target.name:name")
  .save()
```

In turn, Example 5-22 maps to Cypher, as shown in Example 5-23.

Example 5-23. Cypher code produced and executed on Neo4j as a result of a write from Spark

```
UNWIND $events AS event
MERGE (source:Person {name: event.source.name})
SET source = event.source
MERGE (target:Product {name: event.target.name})
SET target = event.target
CREATE (source)-[rel:FOLLOWS]->(target)
SET rel += event.rel
```

When writing into your knowledge graph from Spark, you must also provide some guidance on whether writes should use Cypher's `CREATE`, which allows duplicate patterns, or `MERGE`, which does not. In your code, setting `SaveMode.ErrorIfExists` will build a `CREATE` query, whereas `SaveMode.Overwrite` builds a `MERGE` query.

While Spark is powerful, its data model is quite different from that of a knowledge graph. When you choose to use Spark to analyze or enrich your knowledge graph, you'll have to manage the mapping between its tablular model and your knowledge graph. Think of the Spark connector as plumbing that enables the user to specify transformations between the two worlds, much like a traditional ETL tool, albeit at a lower, textual level of abstraction.

Apache Hop for ETL

If the Spark connector in "Neo4j Spark Connector" on page 87 can be thought of as a special-purpose ETL tool, then you should also be aware of general-purpose ETL tools in the context of knowledge graphs. Modern ETL tools come in a variety of configurations, from utilities you install on your computer to cloud-hosted services. In any case, the value of an ETL tool is that it enables connectivity with a wide variety of systems (not limited to knowledge graphs or even databases) and functionality to allow you to map data elements between systems conveniently.

As authors, we are agnostic about which ETL tool you choose. In this section, Apache Hop has been chosen because it's open source and has good connectivity for knowledge graphs hosted in Neo4j. Our recommendation would be to use the ETL tool that has been chosen for your enterprise. In the absence of an enterprise standard, then open source or cloud tools (e.g., AWS Glue, Cloud ETL, etc.) are reasonable choices.

Most ETL tools provide low-code/no-code environments to help you to orchestrate data flows to and from your knowledge graphs (and other systems). For example, in Figure 5-10 you can see how the ETL tool has been configured to:

- Check for system availability
- Set up the knowledge graph indexes and constraints
- Cleanse import data from Wikipedia
- Run an import job into the knowledge graph

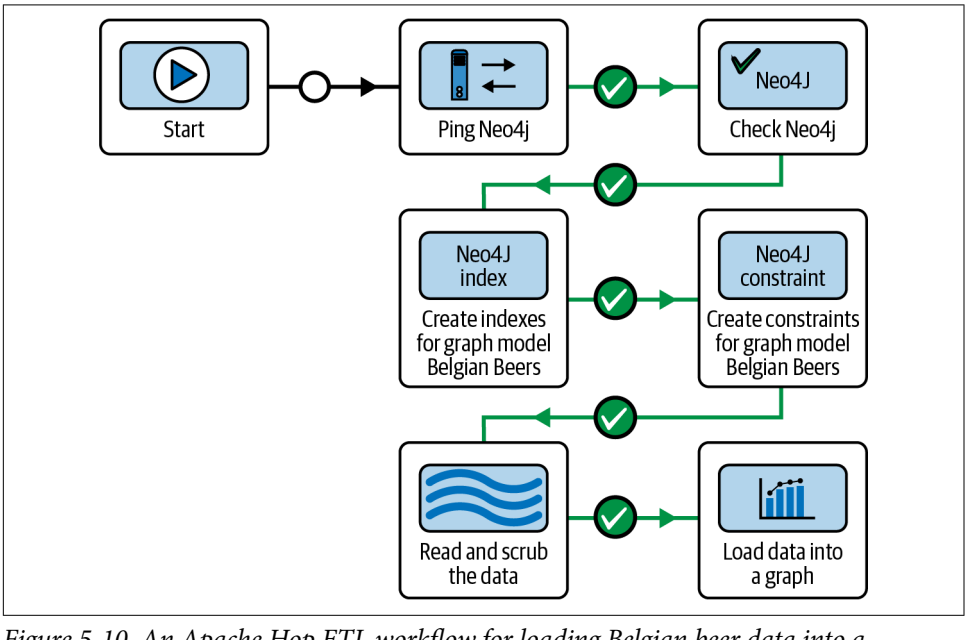


Figure 5-10. An Apache Hop ETL workflow for loading Belgian beer data into a knowledge graph (image based on Apache Hop documentation)

Within the workflow, you configure mappings from source data systems onto your knowledge graph (or vice versa). Again, this is typically low-code/no-code, as shown in Figure 5-11.

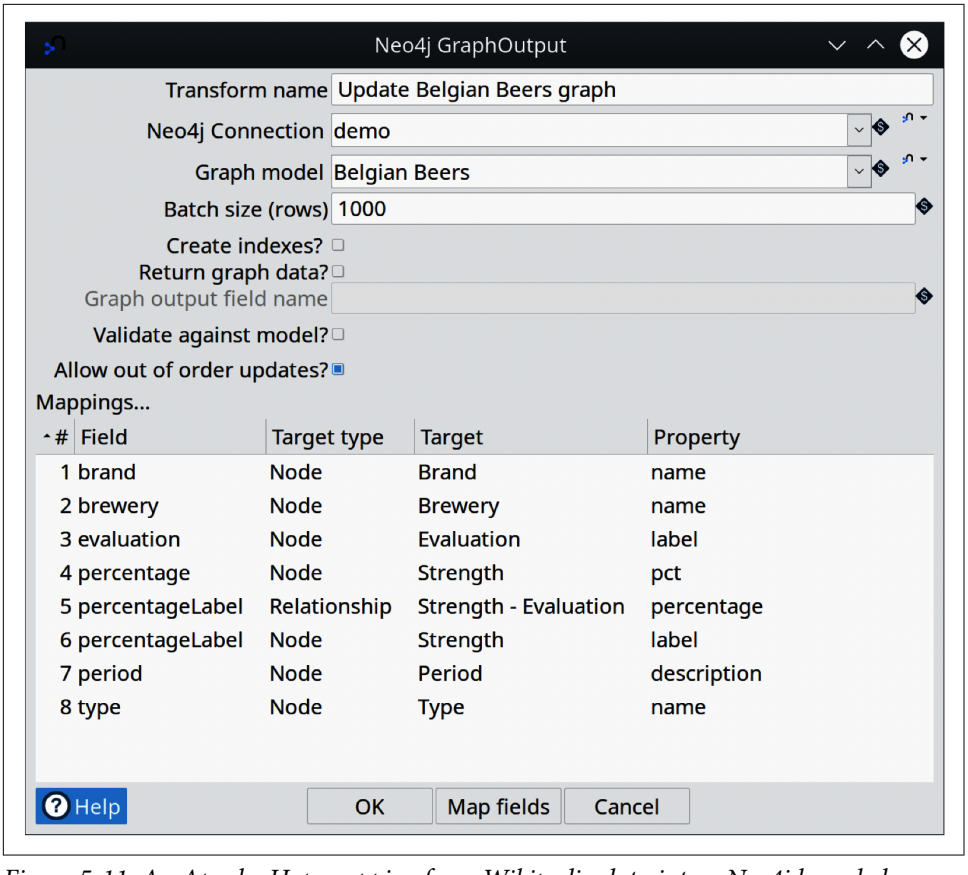


Figure 5-11. An Apache Hop mapping from Wikipedia data into a Neo4j knowledge graph (image based on Apache Hop documentation)

The mappings in Figure 5-11 are quite intuitive, if you understand brewing! Brands, breweries, and so on become nodes in the knowledge graph while the percentage label becomes a relationship between strength and evaluation nodes. While you do require some domain knowledge for this task (and we authors feel sampling Belgian beers would be a fun project to learn), you don't require much technical knowledge. It's a matter of point and click.

For more sophisticated mappings, you can also specify Cypher queries to run for knowledge graph ingress and egress. In this case, you provide parameterized Cypher to Hop and then use the UI tool to map data onto those parameters. At runtime, Hop binds values to those parameters and runs the Cypher queries to load data into or retrieve data from the knowledge graph. Finally, once the workflow has finished, the result is a knowledge graph about Belgian beers—both useful and delicious!

Summary

At this point on your journey into knowledge graphs, you've tackled some significant technical topics on graph database fundamentals, data loading, and data fabric. It takes real effort to understand these tools and how they overlap, so congratulations on getting this far!

From here on, you are going to continue your journey out of the tools and up the stack towards the user. In Chapter 6, you'll take your first steps into graph algorithms to process knowledge graphs to gain actionable insight.

CHAPTER 6

Enriching Knowledge Graphs with Data Science

This chapter will introduce you to *graph data science*. The aim of graph data science is to gain insight into your knowledge graph using graph algorithms. Towards that goal, you'll learn about the common types of graph algorithms and the insights they unearth as well as how Neo4j Graph Data Science provides a simple platform for experimentating with, sharing, and productizing graph algorithms. You'll also learn how to execute graph algorithms on real knowledge graphs and experience how the system does much of the heavy lifting for you.

Like Chapter 3, this chapter is not intended to be comprehensive but rather to give you the foundation to be able to work through the remainder of the book. For those looking for more depth, there are many good books dedicated to specific deep technical topics, like *Graph Data Science for Dummies* by Dr. Alicia Frame and Zach Blumenfeld (Wiley).

Why Graph Algorithms?

Graph algorithms yield some insight about a knowledge graph's structure. That insight could be influential people in a social graph, critical junctions in a rail network, cells of fraudsters, or a common pathogen in a disease pathway. While the design and implementation of graph algorithms is a specialist subject, using them is not. You simply need to understand the purpose of the algorithms and the syntax to execute them over your knowledge graph.

As in Chapter 3 Neo4j has been chosen for graph algorithm examples. It has the most popular and extensive graph algorithms toolkit and is easily accessible to nonspecialists through the Cypher query language.

Neo4j Graph Data Science is easily installed via the Neo4j desktop app. Neo4j Sandbox is a learning environment for Neo4j that provides access to Neo4j Graph Data Science. Neo4j AuraDS is graph data science as a service, a companion to the AuraDB graph database service.

You can use any of these to follow the examples in this chapter and undertake your own graph data science work.

Some readers might remember graph algorithms from college, perhaps even with fondness. For others, the notion of graph algorithms may be new. This chapter will start you off gently, with an overview of the different kinds of data processing that you can do on graphs, followed by a deeper dive into how to use some common graph algorithms. You'll learn how to experiment with those algorithms in a graph data science motion, and how to incorporate them into production systems. By the end of this chapter you'll feel as comfortable running graph algorithms over large knowledge graphs as you already are running Cypher queries.

Different Classes of Graph Algorithms

There are many graph algorithms from which to choose, each of which performs some computation over a given graph to generate insight from the underlying topology and data. Learning which algorithm to choose for a given situation takes modest effort. The first step is to sort the algorithms into categories to provide a high-level view of what kinds of insights can be computed. From there, you can move to the more detailed step of choosing a specific algorithm.

For the purposes of computing over knowledge graphs, there are three broad categories of algorithms:

Statistical
: Provides metrics about the graph, such as the number of nodes and relationships, degree distribution of relationships, types of node labels, and so on. These will provide context for interpretation of your results.

Analytical
: Surfaces significant patterns or latent knowledge over the whole knowledge graph or significant subcomponents.

Machine learning (ML)
: Uses the results from graph algorithms as features to train machine learning models or uses machine learning to evolve the knowledge graph itself.

This chapter is concerned with using statistical and analytical methods to uncover insight from your knowledge graph, leaving machine learning to its own full chapter in Chapter 7. Within statistical and analytic algorithms, there are five categories that are common to knowledge graph use cases:

Network propagation
: Understanding how signals propagate through a knowledge graph requires deep path computations. The resulting pathways can identify the spread of disease in a community or supply-chain weaknesses. Correspondingly, the results can be used to optimize for containment or adding redundancy to critical paths.

Influence
: In current times of near-ubiquitous social media, you are accustomed to the idea of "influencers," or people whose opinions spread rapidly through the social graph. In graph data science, the goal is more general. Influential nodes act as bridges (and bottlenecks) between subgraphs and are ideally positioned to spread information (or disruption) around the network quickly since they are, on average, close to all the other nodes. A measure of a node's influence is known as its *centrality*.

Community detection
: In the human world, communities are the ties that bind us together. That's true in the world of graph algorithms too, but to find tightly knit communities, graph algorithms tend to partition groups by looking for weak links to remove. Being able to detect communities in a knowledge graph tells you about related works that you might want to read, fraudsters cooperating to commit financial crimes, or communities where pathogens (including disinformation) will spread quickly.

Similarity
: When you visualize a knowledge graph, you can often easily spot similar patterns being repeated in the graph. For example, you can see customers with a particular purchase history who would be well served with a particular product recommendation or road intersections associated with abnormal levels of vehicle collisions. But when the graph is large, manual inspection falls short. Similarity algorithms search for known relationships/hierarchies between nodes or common properties on them.

Link prediction
: Using the existing topology of the knowledge graph with some heuristics (for example, creating triangles), link prediction algorithms can enrich the knowledge

graph by computing missing relationships. This is a common use case in social networks where possible friends/followers/contacts can be computed.

Within each of these broad categories, there are usually several different algorithms from which to choose. Each algorithm behaves differently and will typically give different results for the same input. For example, the community detection algorithms Weakly Connected Components (WCC) and Louvain give different results for the same input graph.

The question naturally arises: which should you choose? Often the answer lies in reading the documentation and choosing the algorithm that makes the most sense for your context. Sometimes the answer is to experiment with algorithms and use the one that provides the best answers for your context. But first you need to understand how graph data science functions so that you can experiment with graph algorithms.

Graph Data Science Operations

Neo4j Graph Data Science is a graph computation framework that is conveniently integrated with the Neo4j graph database. This means you can project data from your knowledge graph stored in Neo4j, then compute an analysis using a graph algorithm (using multiple CPUs). You can subsequently inspect the results from your computation, share it with your coworkers, or even write the result back to the underlying knowledge graph to enrich it. The tight integration between Neo4j Graph Data Science and the graph database removes many operational burdens, allows you to focus on running algorithms, and minimizes data wrangling.

It's common in industry for some classes of algorithms to run on graphics processing units (GPUs) to take advantage of the benefits of GPU architecture: high parallelism and high bandwidth between processing elements. However, GPUs are best suited when an algorithm can be expressed as a linear algebra problem, which is by no means universal (or trivial). Currently, Neo4j Graph Data Science is highly optimized for CPU and RAM. It provides best-in-class performance for the widest range of algorithms without the inconvenience of specialized hardware, and it typically outperforms GPU-based solutions (since not all graph algorithms can be expressed as linear algebra problems, which are efficiently executable on GPUs).

Neo4j Graph Data Science offers a broad range of high-performance, parallelized algorithms. You choose an algorithm based on the kind of insight you need and run algorithms with Cypher procedure calls, just like in "Calling Functions and Procedures" on page 44. The way Neo4j Graph Data Science works in combination with your knowledge graph is shown in Figure 6-1.

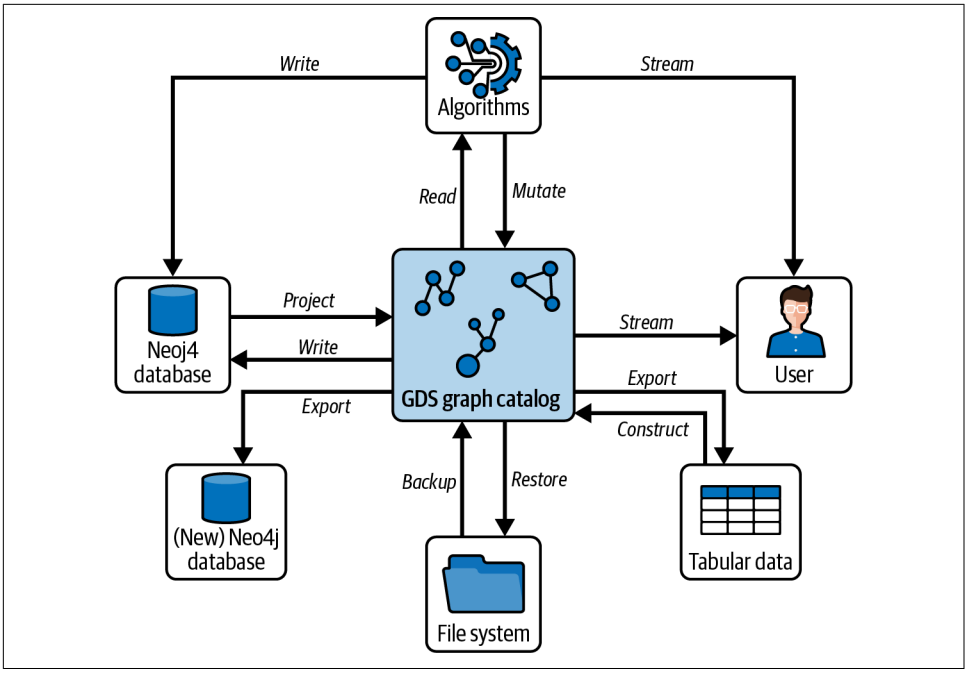


Figure 6-1. The components involved in running a graph algorithm over a knowledge graph in Neo4j

Figure 6-1 shows the four distinct phases of executing a graph algorithm over your knowledge graph:

Read projected graph
: Choose the parts of the graph that you are interested in processing and create a projection. This can be a subgraph, specific labels and relationship types, patterns from a Cypher query, or even the whole knowledge graph.

Load projected graph
: Export the graph projection into a compact, in-memory format ready for parallel processing.

Execute algorithm
: Run the algorithm with parameters you choose.

Store results
: Write the results back into the knowledge graph (e.g., enriching nodes) or compute results that can be sent to downstream systems or the calling user.

Scaling Graph Algorithms

The graph projection phase loads (part of) your knowledge graph into main memory, in a compressed format. Being in memory ensures high performance because of *locality benefits* whereby the neighbors of any given node are also in memory and are therefore fast to access during processing. This means that for very large graphs, you may have to scale up your RAM. For parallel algorithms, you can scale up CPU cores too.

The opposite approach, scale-out, is initially appealing. Adding more CPU cores and RAM by adding servers is a classic approach pioneered by Apache Hadoop. But now you have nonuniform architecture where any node's neighbors are likely to be on remote servers, accessible at a far greater time cost than local memory. In practice, distributed graph processing is inefficient since the locality benefits of being in a single large memory space are absent. Instead of a simple pointer dereferencing, you now have network operations.

While there is active research work on distributed graph processing and the underlay required to make it efficient (e.g., software-defined high-performance networks), at time of this writing, the scale-up approach used by Neo4j Graph Data Science remains the fastest and most practical method. Furthermore, its simplicity means it's often cheaper than running large clusters of compute servers. In the longer term, as research matures into technology, it's possible this situation *might* change, and when it does, as a Graph Data Science user you should be insulated from those implementation details while reaping any accrued performance benefits.

Example 6-1 shows an example of enriching a knowledge graph by computing over it. In this case, it shows how to project a social network of `Person` nodes and `FRIEND` relationships from a graph of people and places, then executes a betweenness centrality algorithm over the resulting graph projection to calculate the most influential person.

Example 6-1. Running Graph Data Science from Cypher

```
CALL gds.graph.project.cypher(
  'gds-example-graph',
  'MATCH (p:Person)
     RETURN id(p) AS id',
  'MATCH  (p1:Person)-[:FRIEND]->(p2:Person)
    RETURN id(p1) AS source, id(p2) AS target, "FRIEND" AS type');

  CALL gds.betweenness.write('gds-example-graph',{
  writeProperty:'betweennessCentrality'});
```

The query consists of four key elements:

Name the graph projection
: Here it is `gds-example-graph`, and that projection will be stored under that name in the graph catalog for access by subsequent computations.

Node query
: Select the nodes that will be included in the projection. Neo4j Graph Data Science has multiple ways of doing this, including simply by node label. The most flexible choice is a Cypher projection that allows you to specify Cypher code to select nodes of interest. The node IDs have to be returned as a column called `id` as part of the contract for projections. In Example 6-1, all `Person` nodes are selected.

Relationship query
: Select the relationships that will be included in the projection. Since this is a Cypher projection, you must supply a Cypher query that identifies the relationships of interest. Importantly, these relationships must connect to nodes from the node projection (or it will fail). The start, end, and type of each relationship must be returned as `source`, `target`, and `type` columns as per the contract for relationship projections.

Execute the algorithm
: The line `gds.betweenness.write` means that the betweenness centrality algorithm will be called and that it will write back its results into the graph, as opposed to streaming them back to the caller (`gds.betweenness.stream`) or enriching only the projection (`gds.betweenness.mutate`). Its parameters are the projection on which to operate (`gds-example-graph`) and the name of the node property into which results will be written (`betweennessCentrality`). Note that once a projection is created, you can run the algorithm as many times as you need against that projection without having to re-create it.

Betweenness centrality computes the number of paths that pass through a given node to establish the node's importance in the graph. If you run the algorithm on the small social network from Figure 3-8, then the knowledge graph will be enriched. This is shown in Figure 6-2.

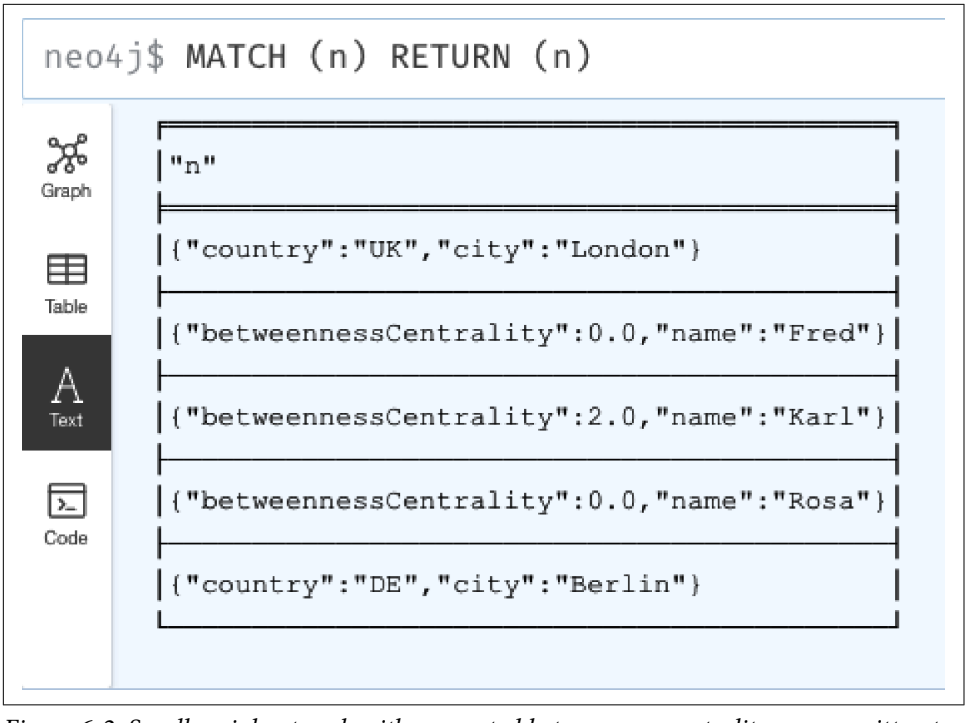


Figure 6-2. Small social network with computed betweenness centrality scores written to the `Person` nodes

In Figure 6-2, it's clear that Karl has the highest betweenness centrality. Since Karl is the connecting node between `Rosa` and `Fred`, without him the whole of the social network would collapse. This is very apparent in Figure 6-3.

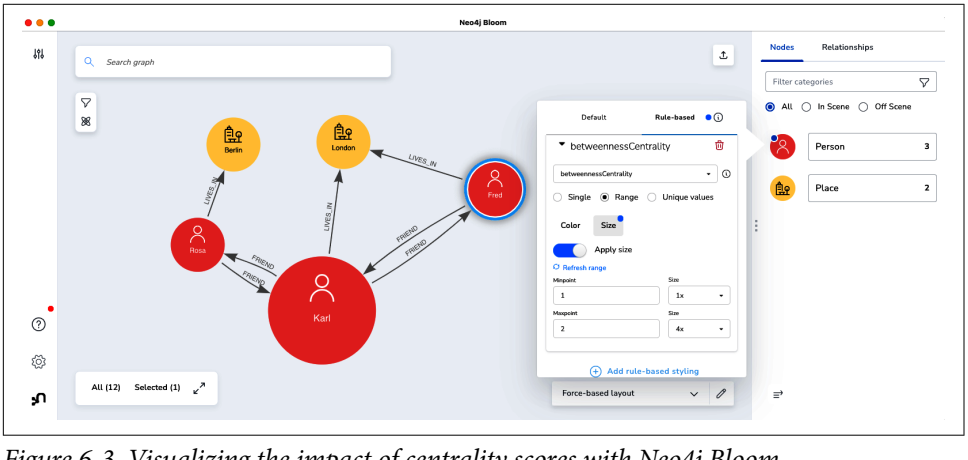


Figure 6-3. Visualizing the impact of centrality scores with Neo4j Bloom

In Neo4j's Bloom visualization tool shown in Figure 6-3, you can configure the size (or color) of a node to be based on a property. This means, for example, that you could make nodes bigger based on their betweenness centrality score so that the higher the centrality, the bigger the node is represented. It makes visualizations much more information rich and valuable for the end user.

Betweenness centrality isn't the only thing you can use to enrich your knowledge graphs. You can choose from 65+ algorithms, including pathfinding, PageRank, Louvain, and so forth. Each of them is executed just like Example 6-1, so experimenting to find the right insight for your domain is operationally straightforward at least.

On a small knowledge graph, the betweenness centrality computation finishes instantaneously. For modest graphs of tens to hundreds of thousands, you probably won't notice the execution time either. But for very large knowledge graphs on a scale of millions or billions, the computation might take several seconds or even minutes (depending on the algorithm, projection, and hardware).

In practice, this isn't a problem since you tend to enrich knowledge graphs periodically. In those situations where you want fast feedback, such as during experimentation, then choose smaller projections or select servers with more RAM and CPU.

Experimenting with Graph Data Science

Part of any data scientist's job is to experiment. If you're going to learn graph data science, you'll also need to learn good experimental habits, and that requires the ability to iterate quickly over hypotheses. Neo4j Graph Data Science contains the tools you need to competently experiment with algorithms over knowledge graphs.

However, the architecture presented in Figure 6-1 shows a very database-centric view of knowledge graph processing. With all those components, it's possible to become lost in complexity when all you're really trying to do is experiment by running algorithms over your knowledge graph. Fortunately, there is Python tooling that can abstract some of the details of databases and allow you to work at a much higher level of abstraction. After all, viewing the world as Python objects is much more idiomatic for data scientists.

Your graph data science toolkit comprises Neo4j and Graph Data Science (of course), the Neo4j Graph Data Science Python driver, and Jupyter Notebook or something similar. Since you've gotten this far in the book, it's likely you have already installed Neo4j and Graph Data Science.

The next steps are to install the Pythonic API for Graph Data Science and, if necessary, Jupyter Notebook or something similar. Assuming you have Python 3 installed, then use `pip` to install the other dependencies. `pip install graphdatascience`

installs the Pythonic API for Graph Data Science, and `pip install notebook` installs Jupyter Notebook support.

Now that you have your tooling, you can start experimenting. The following examples are all based on the rail network in Great Britain. It's large, but not too large, highly interconnected, and very much a real-world use case, making it ideal for the budding graph data scientist to explore.

You can download the source data for this experiment from *https://github.com/jbarrasa/gc-2022*.

To begin, load the data that represents the railway stations and the tracks between them. That data is stored in CSV files, one for stations (*nr-stations-all.csv*) and one for tracks linking them (*nr-station-links.csv*). The code for how to load that data using Python is shown in Example 6-2.

Example 6-2. Loading CSV data into Neo4j in Python

```
from graphdatascience import GraphDataScience

# Connect to the database
host = "bolt://127.0.0.1:7687"
user = "neo4j"
password= "yolo"

gds = GraphDataScience(host, auth=(user, password), database="neo4j")

# Load stations as nodes
gds.run_cypher(
    """
    LOAD CSV WITH HEADERS FROM "nr-stations-all.csv" AS station
    CREATE (:Station {name: station.name, crs: station.crs})
    """
)

# Load tracks bewteen stations as relationships
gds.run_cypher(
    """
    LOAD CSV WITH HEADERS FROM "nr-station-links.csv" AS track
    MATCH (from:Station {crs: track.from})
    MATCH (to:Station {crs: track.to})
    MERGE (from)-[:TRACK {distance: round( toFloat(track.distance), 2 )}]->(to)
    """
)

gds.close()
```

The Python code in Example 6-2 does three things:

Creates a connection to the database
: Creating an instance of the `GraphDataScience` class provides authenticated access to the database management system that will host the underlying knowledge graph. Pay attention to the network endpoint and username/password combinations.

Loads nodes into the graph to represent stations
: Use `gds.run_cypher`, which allows the data scientist to run arbitrary Cypher queries. In this case, it uses a `LOAD CSV` command just like in Chapter 4. The command takes data from a CSV file (ignoring the headers) and uses Cypher `CREATE` to populate the knowledge graph with `Station` nodes with their common name (`name`) and their official unique code (`crs`).

Links nodes with relationships to represent tracks
: Again, `gds.run_cypher` is used to execute a `LOAD CSV` job. Here the database looks up the `from` and `to` nodes for a piece of track using the `crs` unique codes for the stations. It then uses `MERGE` to create a single relationship of type `TRACK` between those stations while adding a `distance` property to those relationships. The `distance` property is stored as a floating point value rounded to two decimal places.

Nowhere in Example 6-2 did you have to deal with drivers, sessions, or other technical database things. You just used a Python object in your code.

Once the data is loaded into the graph, you could simply poke around the data using Cypher queries using the Python API and `run_cypher`. That's handy if you're a keen Python programmer, but it's nothing radically different from Chapter 3. However, the Python API also allows you to create projections and execute algorithms over them. It's time to run those algorithms.

One of the most common queries for travel networks is to find the shortest (or cheapest) path between two stations. It's straightforward to do this from Python with Graph Data Science. First, you need to create a projection from the railways knowledge graph, creating a domain-specific in-memory graph suitable for algorithmic computation. In this case, the projected graph is topologically similar to the original knowledge graph, but the nodes contain less data, just common station names, and distances between them are stored as a relationship property. If you later find you need more data in a projection, simply create that new richer projection where it will be added to the graph catalog alongside your existing projections. Example 6-3 shows how to use the Python API to create a Cypher projection (the other projection methods are also supported from Python).

Example 6-3. Using the Python API for Graph Data Science to create a graph projection from the railway knowledge graph

```
from graphdatascience import GraphDataScience

host = "bolt://127.0.0.1:7687"
user = "neo4j"
password= "yolo"

gds = GraphDataScience(host, auth=(user, password), database="neo4j")

gds.graph.project.cypher(
    graph_name='trains',
    node_spec='MATCH (s:Station) RETURN id(s) AS id',
    relationship_spec=
    """
    MATCH (s1:Station)-[t:TRACK]->(s2:Station)
    RETURN id(s1) AS source, id(s2) AS target, t.distance AS distance
    """
)

gds.close()
```

In Example 6-3 a `GraphDataScience` object is instantiated to obtain an authenticated connection to the database hosting the knowledge graph. Then, `gds.graph.project.cypher` is used to create a projection called `trains` that will be stored in the graph catalog for later use. The content of the `trains` projection is defined by `node_spec` and `relationship_spec` properties. The `node_spec` parameter is a `MATCH` clause that returns the node IDs of all stations. The `relationship_spec` parameter is a `MATCH` clause that returns the node IDs of `source` and `target` stations, along with a `distance` property for the `TRACK` relationships that will connect stations in the projection. Once executed, the projection is stored in the graph catalog ready for experimentation.

The Python code for running a shortest-path algorithm is shown in Example 6-4. It follows the same pattern as the other examples in terms of setting up a connection to the database. It then sets up parameters for the experiment by using `find_node_id` and a Cypher expression to find the IDs of the nodes representing Birmingham New Street and Edinburgh stations, which are stored in the variables `bham` and `eboro`, respectively.

The code then runs one of the shortest-path algorithms, Dijkstra's algorithm in this case. The algorithm is set up to work on the projection called `trains` and takes `bham` and `eboro` as its `sourceNode` and `destinationNode` node parameters. It also takes a value for the `relationshipWeightProperty` parameter, which is the `distance` property that the algorithm will use to compute path costs.

Example 6-4. Using the Python API for Graph Data Science to compute the shortest path between Birmingham New Street and Edinburgh

```
from graphdatascience import GraphDataScience

host = "bolt://127.0.0.1:7687"
user = "neo4j"
password= "yolo"

gds = GraphDataScience(host, auth=(user, password), database="neo4j")

bham = gds.find_node_id(["Station"], {"name": "Birmingham New Street"})
eboro = gds.find_node_id(["Station"], {"name": "Edinburgh"})

shortest_path = gds.shortestPath.dijkstra.stream(
    gds.graph.get("trains"),
    sourceNode=bham,
    targetNode=eboro,
    relationshipWeightProperty="distance"
)

print("Shortest distance: %s" % shortest_path.get('costs').get(0)[-1])

gds.close()
```

Once the algorithm is executed, you'll see that the shortest path between Birmingham New Street and Edinburgh is printed out: `Shortest distance: 298.0`. You can also visualize this path in the Neo4j Browser, as shown in Figure 6-4.

Figure 6-4. Shortest path between Birmingham New Street and Edinburgh

Curious readers might be wondering about the choice of railway stations in this example. As it happens, one of the authors (Jim Webber) grew up in Birmingham, England, which was one reason that Birmingham New Street was an obvious choice. Edinburgh just happens to be a beautiful city.

But Birmingham is geographically interesting.[1] It's in the center of England, and Birmingham New Street is a busy hub station that serves routes across Great Britain, including routes that intersect Edinburgh in the Scottish lowlands. As such, you might suppose that Birmingham New Street is the most critical station on the rail network in Great Britain. Based on that supposition, you might also advocate for significant investment in the station and its surrounding track to ensure that failures are few and gracefully handled.

But what if the supposition is wrong? Fortunately, you have the tools at your disposal to quickly verify that hunch.

The `trains` projection contains all the data you need to analyze the situation. To verify (or debunk) Birmingham New Street's importance in the rail network, you're going to have to compute its *centrality*. Happily, that is very straightforward, as you can see in Example 6-5.

Example 6-5. Using the Python API for Graph Data Science to compute centrality scores for all railway stations in Great Britain

```
from graphdatascience import GraphDataScience

host = "bolt://127.0.0.1:7687"
user = "neo4j"
password = "yolo"

gds = GraphDataScience(host, auth=(user, password), database="neo4j")
graph = gds.graph.get("trains")
result = gds.betweenness.stream(graph)
highest_score = result.sort_values(by="score", ascending=False)
    .iloc[0:1].get('nodeId')

n = gds.run_cypher(f"MATCH (s:Station) WHERE ID(s) = {int(highest_score)}
    RETURN s.name")
print("Station with highest centrality: %s" % n["s.name"][0])

gds.close()
```

Running the code in Example 6-5 reveals `Station with highest centrality: Tamworth`, which isn't what was expected from an informal understanding of the system. In fact, Tamworth's centrality score is `1,967,643` while Birmingham New Streets, is almost eight times lower at `254,706`.

1 But also has an undeserved reputation among the British for being boring. As the heart of the industrial revolution, it was the Silicon Valley of its day, and much of that heritage remains.

Now you know that it is Tamworth, just a few miles away from Birmingham, that has the highest centrality score and so the greatest impact on the rail network in the case of failures. If you are in charge of rail transport policy, that's where you might start.

Graph Data Science Reveals Counterintuitive Results

A puzzling fact about the result in Example 6-5 is that high-centrality Tamworth has far lower passenger numbers at 1.2 million per year versus Birmingham New Street's 47 million (plus 7 million passenger interchanges). In fact, Birmingham New Street is the eighth busiest station in Great Britain and the busiest station outside London. This would seem to be opposed to the finding that Tamworth is the most important.

In reality, two things might be occurring. The first is that the centrality computation didn't weight passengers or train routes, only connected track. From the point of view of the physical network, Tamworth really is important. Second, network effects are real. It's quite likely that if infrastructure at Tamworth fails, it will impact the large volume of passengers at Birmingham New Street and stations further afield, as faults ripple through the network.

Real life isn't always as as simple as a single algorithmic score. Graph data scientists should always look to refine their problems with richer data or more accurate projections, or they should be prepared to reformulate their hypotheses and bring more data and algorithms to bear. Fortunately, the experimental motion of Python and Graph Data Science allows for rapid reevaluation of test cases, giving you more opportunity to iterate toward a good answer.

Production Considerations

The setup in "Experimenting with Graph Data Science" on page 101 is great for experimentation, but there comes a time when you're happy with your projections and algorithm choices, and you want to move into production. At this point, you're no longer in the world of Python objects and Jupyter Notebooks but squarely in the domain of production data engineering.

A common setup to support knowledge graphs and graph algorithms is to separate your physical servers by role, providing physical isolation for different workloads. Neo4j supports the concept of *primary* and *secondary* servers for this purpose. Primary servers are responsible for transaction processing, keeping your knowledge graph clustered for scale and fault tolerance. They also typically serve Cypher queries.

In principle, primaries can also run graph algorithms, but doing so would cause contention between the computational workload of the algorithms and the ongoing database workload. Instead, consider deploying secondary instances for data science. Secondary instances have looser time guarantees about updates compared to

primaries since they receive their updates asynchronously rather than transactionally. They don't slow down transaction processing.

It is also normal to size secondaries differently from primaries and perhaps even to each other depending on the intended workload. For example, you could decide that your primary instances are equipped with large RAM and high I/O but modest CPU since you intend to run database queries and transactions that are rarely CPU bound. Your secondaries would then be configured with multiple fast CPUs and high RAM to enable fast parallel computation using graph algorithms or to support graph global queries, as shown in Figure 6-5.

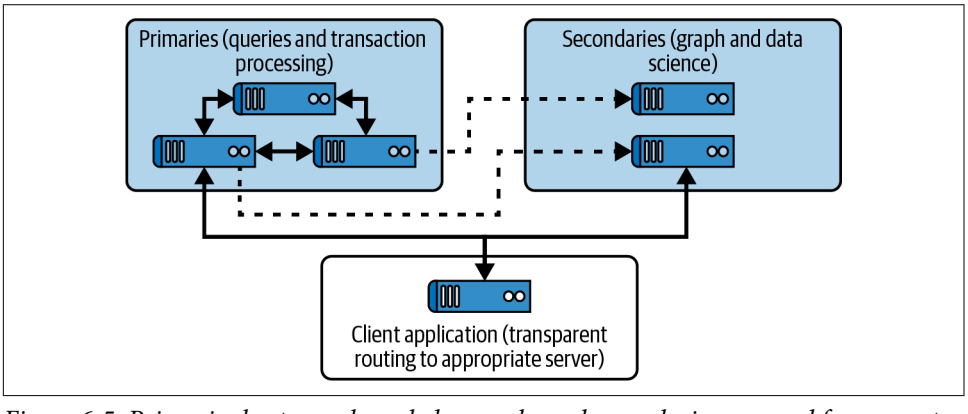


Figure 6-5. Primaries host your knowledge graph, and secondaries are used for compute-intensive graph data science

The infrastructure of Neo4j clusters allows you to tag servers with their intended roles so that users of the system send their jobs to the most appropriate physical server for their needs. All this is simple and *mostly* transparent.

To be clear, Neo4j makes operating hybrid transactional and analytics processing (HTAP) for knowledge graphs very straightforward. It's a single cluster with the same data throughout. There's no need for ETL or online analytical processing (OLAP) cubes or other complex data engineering.

But there will be times when you want guarantees about a secondary's freshness before you run a graph algorithm on it. In Chapter 3 you read that Neo4j provides an optional causal barrier, which means that users of the database always see at least their own writes, even with servers spread out over a wide area network. This casual barrier applies across both primaries and secondaries and for data science just as much as transactional workloads. Using the causal barrier is easy, as you can see in Example 6-6.

Example 6-6. Bookmarks work on secondaries, which are good workhorses for Graph Data Science

```
try ( Session session = driver.session( AccessMode.WRITE ) )
{
    try ( Transaction tx = session.beginTransaction() )
    {
        tx.run( "CREATE (user:User {userId: {userId}, passwordHash: {passwordHash}})",
                parameters( "userId", userId, "passwordHash", passwordHash ) );

        tx.success();
    }

    // Bookmark is the ID of the transaction you just committed.
    String bookmark = session.lastBookmark();
}
```

The first part of the causal barrier in Example 6-6 is where `session.lastBookmark()` is called. This gives you the ID of the transaction you just committed. You can subsequently present that ID to the database to ensure no operations will proceed unless that transaction has been processed by the receiving server (including secondaries), as shown in Example 6-7.

Example 6-7. Using a Bookmark to enforce a causal barrier

```
{
    try ( Transaction tx = session.beginTransaction( bookmark ) )
    {
        tx.run( "MATCH (user:User {userId: {userId}}) RETURN *",
                parameters( "userId", userId ) );

        tx.success();
    }
}
```

In Example 6-7, when opening the next transaction with `session.beginTransaction( bookmark )`, you provide the `bookmark` object from Example 6-6. Now the operation will not proceed until that transaction is processed at the current server. You will always see at least your own writes, even on secondaries.

Enriching the Knowledge Graph

The other place where production data science tends to differ from the experimental phase is what you choose to do with the results. For experiments, capturing results in a notebook (physical or digital) is often preferred. For production use, you need to utilize the results to improve the system.

Recall from Example 6-5 that results of the centrality computation were streamed back to the Python client using `result = gds.betweenness.stream(graph)` and then printed to the console. If those results were useful, then you'd like to capture them in a way that can be operationalized. Fortunately, this is straightforward; you simply swap out the `stream` method for `mutate`, which updates the projection with the results, or `write`, which enriches the underlying knowledge graph (irrespective of where the computation executed).

Example 6-8 shows how this is achieved.

Example 6-8. Using the Python API for Graph Data Science to compute centrality scores for all railway stations in Great Britain

```
from graphdatascience import GraphDataScience

host = "bolt://127.0.0.1:7687"
user = "neo4j"
password = "yolo"

gds = GraphDataScience(host, auth=(user, password), database="neo4j")
graph = gds.graph.get("trains")
result = gds.betweenness.write(gds.graph.get("trains"), writeProperty="betweenness" )

total_stations = gds.run_cypher("MATCH (s:Station) RETURN count(s) AS total_stations")
print(f'Total number of stations: {total_stations.iloc[0][0]}')

processed_stations = gds.run_cypher(
  """
  MATCH (s:Station)
  WHERE s.betweenness IS NOT NULL
  RETURN count(s) AS stations_processed
  """)

print(f'Number of stations with betweenness score: {processed_stations.iloc[0][0]}')

gds.close()
```

If you run the code in Example 6-8, it performs the centrality calculation, but instead of just sending results to the console, it writes them back to the underlying knowledge graph, enriching it for future use cases, such as graph feature engineering for ML. A message is also written to the console so that you can verify that the knowledge graph has been enriched:

```
Total number of stations: 2593
Number of stations with betweenness score: 2593
```

As you can see, moving from exploratory graph analysis to data science to refining experiments to production readiness is straightforward. Even readers who are not data scientists by training should be able to make good progress following this

pattern. As the underlying knowledge graph gets richer (and of course is kept up to date by frequent application of algorithms), the possibilities for deep understanding of data emerge.

Summary

In this chapter, you've learned (or perhaps even relearned) about graph algorithms. Far from being dull and computerish, experimentation with graph algorithms surfaces quantifiable insight. You've also seen that very little coding or data engineering is needed.

In the next chapter, you're going to mix graph algorithms with ML. You'll build better predictive models with graph features created by algorithms and use graph-native ML so that algorithms directly enrich the knowledge graph.

CHAPTER 7

Graph-Native Machine Learning

In this chapter, you'll learn about the intersection of graphs and ML. You'll see how ML techniques can automatically enrich an existing knowledge graph as well as how to mine features from a knowledge graph to create accurate predictive models.

This chapter builds on the skills you learned in Chapter 6, where you exploited the topology of a knowledge graph by using graph algorithms. In doing so, you discovered useful insights like the shortest paths between nodes and revealed the communities within the knowledge graph. The same skills, using Neo4j Graph Data Science, Cypher, and Python, will again be called to action in this chapter. You will build on those skills and learn how to add *graph-native machine learning* to your toolbox to create models that can enrich your knowledge graph.

Like Chapters 3, 5, and 6, this chapter is not intended to be a comprehensive guide to graph-based ML. But it gives you enough information to begin to use knowledge graphs as the basis for ML, with sufficient detail to enable you to explore further should you choose.

Machine Learning in a Nutshell

ML is a huge area of research and practice. But at its most abstract level, it is about deriving programs from data, and of course knowledge graphs are excellent data.

Traditionally, most software has been written to take some input and apply a function to produce some output data. The function is written by an expert human, often with a deep understanding of the domain. The function is usually a sophisticated program, which may require a team of professionals to build and maintain. Those people must handle all the potential edge cases and intricacies of the model so that its outputs remain high quality.

ML inverts these responsibilities. In training an ML model, the inputs are data and prior outputs while the output is essentially a program. The ML system learns what the program should do by cleverly correlating input and output data to create a new function or rule. Periodically, the model may be retrained as new data becomes available to ensure its output remains high quality.

The kind of program produced through ML depends on both the problem being solved and the preferences of the team that owns the problem domain. The program could be a statistical regression where historical data is mapped to a curve whose function has been determined by analyzing historical data so that future data points can be calculated. It could be a neural network (which underpins most deep learning approaches) where a program is trained over time to recognize or classify inputs (such as images) by creating a layered, linked network akin to a biological brain. It might also be a geometric learning system that can understand, which extends deep learning into the multidimensional topology of graphs.

The choice of model is an expert decision, but fueling that ML with knowledge graphs typically makes good sense. There are two ways you might choose to integrate graphs and ML:

- To predict how a knowledge graph will evolve over time, known as *in-graph machine learning*
- To extract features from the knowledge graph to build a high-quality external predictive model, known as *graph feature engineering*

Irrespective of which methods are used, the systems surrounding the ML model and knowledge graph are often arranged in a feedback loop. This means use of the model ultimately enriches the knowledge graph, which is then used to improve the model.

Topological Machine Learning

In-graph machine learning is a collection of techniques where the knowledge graph can be enriched by computing over itself. The process searches for opportunities to add missing relationships, node labels, and even properties.

In-graph machine learning capabilities are contained within the Neo4j Graph Data Science library, which is easily installed via the Neo4j desktop app for learning purposes.

With graph-native ML, you can execute algorithms using your knowledge graph's topology and data to gain insight. For example, using the built-in Neo4j movie knowledge graph as your basis, you can immediately run topological link predic-

tion algorithms. Start by running the Neo4j Browser on an empty database, then enter `:play movie graph` to start the built-in tutorial to build a small knowledge graph of movies, actors, and directors. Browsing around the graph centered on Keanu Reeves, you can see that he's had a busy career, including acting in all of the *Matrix* movies, as shown in Figure 7-1.

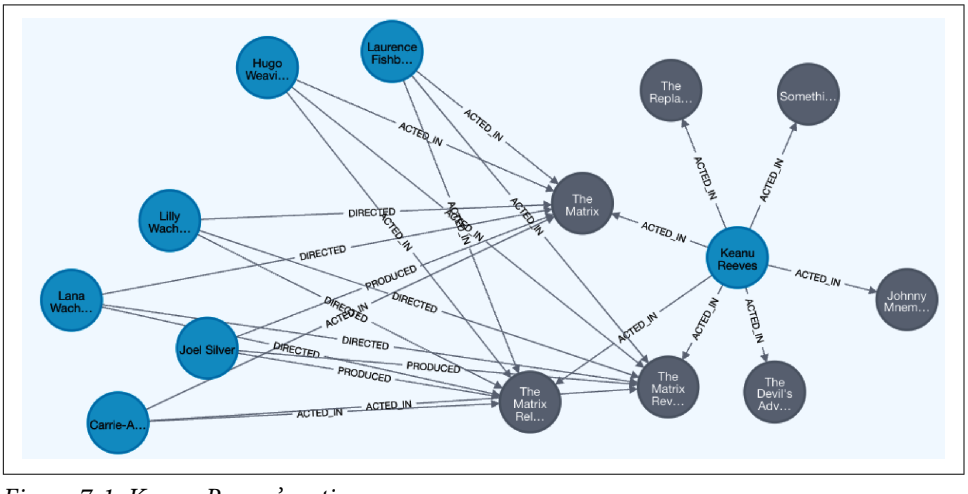


Figure 7-1. Keanu Reeves's acting career

In a real-world knowledge graph, you know that things are imperfect. Some data is missing, some has been mishandled, and some you don't even know about yet. While even imperfect knowledge graphs are useful, it is valuable to be able to use tools to increase the quality of the data. This is quite possible with graph data science, of course. To demonstrate topological link prediction, just delete a relationship you know exists and watch the algorithm recommend its re-creation. Remove the relationship between Keanu Reeves and the movie *The Matrix*, as shown in Example 7-1.

Example 7-1. Perturbing a knowledge graph to set up an algorithmic experiment

```
MATCH (:Person {name:'Keanu Reeves'})-[r:ACTED_IN]->(:Movie {title:'The Matrix'})
DELETE r
```

You know that the `ACTED_IN` relationship between Keanu Reeves and *The Matrix* should have a high probability of existing—after all, you just deleted it! This is easily tested by running a topological link prediction algorithm like preferential attachment by Barabási and Albert. Executing the algorithm can be done directly on the knowledge graph, as shown in Example 7-2.

Example 7-2. Invoking the Preferential Attachment algorithm on an actor and a movie to see if they should be linked

```
MATCH (Keanu:Person {name:'Keanu Reeves'})
MATCH (TheMatrix:Movie {title:'The Matrix'})
RETURN gds.alpha.linkprediction.preferentialAttachment
       (Keanu, TheMatrix, {relationshipQuery: "ACTED_IN"}) AS score
```

Running the code in Example 7-2 gives `28.0` as the score for the likelihood of Keanu Reeves having acted in *The Matrix*. This is a good indication that the relationship should exist, but it's not emphatic. Using another topological link prediction algorithms might help to improve the score, depending on the nature of the graph, as can using a larger graph (at the expense of longer compute times). But another avenue of exploration is to train and use ML pipelines to solve the link-prediction problem.

Graph-Native Machine Learning Pipelines

You can potentially improve link prediction (and prediction of labels and properties too) by moving to an ML pipeline. For this, you will need a reasonably large set of relevant training data from which features can be extracted by the ML pipeline to train a predictive model.

Graph Feature Engineering

Feature engineering is the activity of identifying useful data that can be used to train an ML model. Traditionally, those features came from columns in relational databases and might be labeled age, gender, or postal code.

Graph feature engineering involves identifying and extracting data from a knowledge graph to be integrated into an ML model. Since knowledge graphs have a topology as well as data, this enables you to extract more and typically higher-quality features than would otherwise be possible.

Features are often generated by running a graph algorithm over the knowledge graph. For example, you might compute the PageRank or community of each node in a graph and use those as features. Equally, you might simply encode the topology of the graph directly as a (non-human-readable) feature using node embeddings. Either way, you increase the number and quality of features available for ML that can lead to better performing predictive models.

With traditional ML, a significant amount of data engineering can be needed to set things up. However, if we use Neo4j to host our knowledge graph, then we can simply declare ML pipelines into existence, with only a little data wrangling needed.

The construction of a pipeline for link prediction is as follows:

1. Create a graph projection from the underlying knowledge graph.
2. Declare the pipeline into existence.
3. Add properties to the nodes in the graph projection containing values based only on the result of some computation on the relationships in the graph (e.g., by running one or more graph algorithms).
4. Add one or more *link features* to create a feature vector for each node pair in the graph using the properties computed in the previous step using a combiner.
5. Split the graph into disjoint test, train, and feature sets.
6. Add model candidates (for example, logistical regression).
7. Optionally estimate the computational requirements of the training phase to ensure your machines have enough memory.
8. Train the model, including evaluation metrics to judge the quality of the trained model.
9. If the model is deemed good enough via its metrics, then push it into production.

Instinctively, that seems like a lot of effort. But since the required functionality is already built into Neo4j, you only have to declare the configuration of the pipeline, rather than directly implement it. Working through an example illustrates this very well.

Recommending Complementary Actors

You've already seen the movies knowledge graph in "Topological Machine Learning" on page 114. It's a small knowledge graph intended for educational purposes, but it's realistic enough for you to build a predictive model which can recommend actors that might work well together based on whom they have previously worked with.

The first step is a small amount of data wrangling. Take the existing movie graph and match where actors have acted in the same movie. Then connect each pair of actors who've been in the same movie with an `ACTED_WITH` relationship (direction is unimportant). This is the only pretraining data wrangling you'll need, and it's simple, as you can see in Example 7-3.

Example 7-3. Adding `ACTED_WITH` relationships to the movie knowledge graph

```
MATCH (a:Person)-[:ACTED_IN]->(:Movie)<-[:ACTED_IN]-(b:Person)
MERGE (a)-[:ACTED_WITH]->(b)
```

Once your knowledge graph is enriched with `ACTED_WITH` relationships, you can easily visualize the network of actors who've worked together in the Neo4j Browser tool, as shown in Figure 7-2.

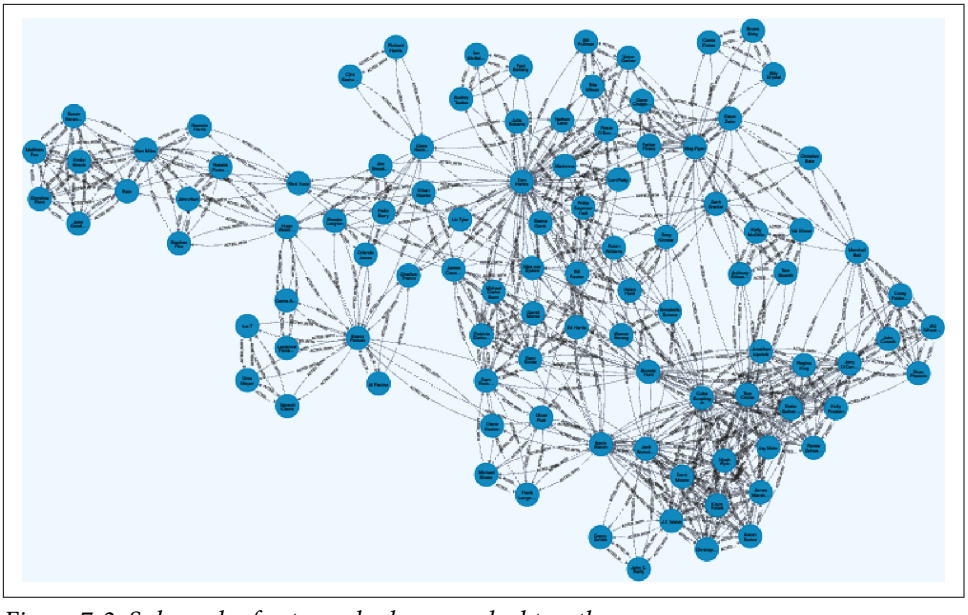


Figure 7-2. Subgraph of actors who have worked together

Now you can create a projection of that knowledge graph. The projection, just as you saw in Chapter 6, forms the basis for subsequent ML. This is a single procedure call, as shown in Example 7-4. You have to provide the name of the projected graph (`actors-graph`). You also need to supply the labels of any nodes to be included (`Person` in this case) and any properties on those nodes (empty `{}` in this case). Finally, you need to supply the names of any relationships (`ACTED_WITH`) and their orientation (`UNDIRECTED`).

Example 7-4. Creating a projection of actors who have worked with other actors

```
CALL gds.graph.project( 'actors-graph',
  {
    Person: {}
  },
  {
    ACTED_WITH: {
      orientation: 'UNDIRECTED'
    }
  }
)
```

The next step is to create your pipeline. This takes a single Cypher procedure call, `CALL gds.beta.pipeline.linkPrediction.create('actors-pipeline')`.

Now you have to think about the node properties that will be used to train the ML model. You can choose existing node properties from the underlying knowledge graph, or you can create new properties by running graph algorithms over the data. In this case, the topology shown in Figure 7-2 is important, so you need to encode that topology as a node property that can be read by the ML stages of the pipeline.

Using a node embedding algorithm, which encodes a node's local topology as a number, is a good choice here. Neo4j Graph Data Science has a handful of such algorithms, including Node2Vec and FastRP, which are potentially suitable. In Example 7-5 you can see how to configure the FastRP algorithm to calculate a node property called `embedding` on each of the nodes in the projection. Each property will be enriched with a numerical encoding of the node's position in the topology when the pipeline executes.

Example 7-5. Configuring the pipeline to inject a node embedding property into all nodes using the FastRP algorithm

```
CALL gds.beta.pipeline.linkPrediction.addNodeProperty('actors-pipeline', 'fastRP', {
  mutateProperty: 'embedding',
  embeddingDimension: 256,
  randomSeed: 42
})
```

The next step is to configure the part of the pipeline that deals with link features. Recall that a link feature is a single feature that joins a node pair together and is a necessity for topological ML. The link feature steps will be executed both at training time when the pipeline runs and in production when the model makes predictions.

Example 7-6 shows how this can be done. The feature step is added with the `addFeature` procedure, which takes the name of the pipeline and an algorithm for determining whether a node pair should be joined. In this case, `cosine` has been chosen, which generates the cosine similarity of the `embedding` node properties defined in Example 7-5. If the vectors are close enough, they are considered to be linked. Other options for link feature algorithms include Hadamard product and L2. In this small use case, after some experimentation `cosine` appears to give the best results.

Example 7-6. Adding `cosine` (closeness) as the link feature that determines if two nodes are linked

```
CALL gds.beta.pipeline.linkPrediction.addFeature('actors-pipeline', 'cosine', {
  nodeProperties: ['embedding']
}) YIELD featureSteps
```

In order to train and test the predictive model, positive and negative examples need to be fed into the pipeline. This is automated by the tooling, with the user only

having to specify the fraction of the graph used for testing and the fraction of the test complement set (data not used for testing) that will be used for training. This is shown in Example 7-7. If you don't want to tune the relationship splitting yourself, you can omit this configuration and use the defaults.

Example 7-7. Splitting the test and train data for the pipeline

```
CALL gds.beta.pipeline.linkPrediction.configureSplit('actors-pipeline', {
      testFraction: 0.25,
      trainFraction: 0.6,
      validationFolds: 3
    })
```

The next step in configuring the pipeline is to add one or more model candidates. At the time of writing, the choices are:

- Logistic regression
- Random forest
- Multilayer perceptron

These models can be added to the pipeline with a procedure call like `CALL gds.beta.pipeline.linkPrediction.addLogisticRegression(`*`actors-pipeline`*`)`. The model (or models) registered are then readied for auto tuning with a single procedure call `CALL gds.alpha.pipeline.linkPrediction.configureAutoTuning(`*`actors-pipeline`*`, {maxTrials: 100})`. In that call, `actors-pipeline` is the name of the pipeline to be tuned, and `maxTrials` declares the maximum number of tuning iterations each model will undergo before choosing the best parameters. Note that on small models like this, the computational time is negligible on modern computers, but on a large ML pipeline, the processing time of each iteration could be many minutes.

The pipeline is now ready for training. But before you do that on a production-sized data set, it's worth taking a moment to check the training memory requirements and, if necessary, running training on a machine with sufficient memory. The memory check is just a simple procedure call again, as shown in Example 7-8.

Example 7-8. Checking for sufficient memory to run the training task

```
CALL gds.beta.pipeline.linkPrediction.train.estimate('actors-graph', {
  pipeline: 'actors',
  modelName: 'actors-model',
  targetRelationshipType: 'ACTED_WITH'
})
YIELD requiredMemory
```

Once you're comfortable with the memory requirements for training, you can proceed to train the model as shown in Example 7-9. The code in Example 7-9 instantiates a pipeline using the configuration that you provided earlier and executes the training phase, producing a model called `actors-model` as its output. It also yields some metrics about the training phase, which a data scientist can use to reason about the quality of the model.

Example 7-9. Running the training task

```
CALL gds.beta.pipeline.linkPrediction.train('actors-graph', {
  pipeline: 'actors-pipeline',
  modelName: 'actors-model',
  metrics: ['AUCPR'],
  targetRelationshipType: 'ACTED_WITH',
  randomSeed: 73
}) YIELD modelInfo, modelSelectionStats

RETURN
  modelInfo.bestParameters AS winningModel,
  modelInfo.metrics.AUCPR.train.avg AS avgTrainScore,
  modelInfo.metrics.AUCPR.outerTrain AS outerTrainScore,
  modelInfo.metrics.AUCPR.test AS testScore,
  [cand IN modelSelectionStats.modelCandidates |
    cand.metrics.AUCPR.validation.avg] AS validationScores
```

Once you have a trained model, you can use it to make predictions. The starting point for that is to feed in a graph projection on which the model can be executed. The input graph `actors-input-graph-for-prediction` shown in Example 7-10 is based on the original movie graph but with some perturbations that make it slightly different. Those perturbations are simple Cypher statements to add in a few more synthetic actors and delete a few `ACTED_IN` relationships. In a production scenario, you would use your real knowledge graph instead.

Example 7-10. Creating an input graph on which the model will make predictions

```
CALL gds.graph.project(
  'actors-input-graph-for-prediction',
  {
    Person: {}
  },
  {
    ACTED_WITH: {
      orientation: 'UNDIRECTED'
    }
  }
)
```

The model can now execute on the input graph, as shown in Example 7-11. The parameters are somewhat self-explanatory:

- `actors-input-graph-for-prediction` is the projection used as the input graph.
- `actors-model` is the trained model produced by Example 7-9.
- `relationshipTypes` specifies that the execution should only concern itself with processing `ACTED_WITH` relationships.
- `mutateRelationshipType` specifies that the execution should write back `SHOULD_ACT_WITH` relationships into the projection when a certain confidence level is met.
- `topN` sets the number of recommendations required.
- `threshold` sets the confidence level below which a recommendation is discarded. In this case, it's a low threshold, since it's a small-scale problem with an immediate human end user (the person who casts roles for actors). A real production system is likely to have a higher threshold to prevent too many low-quality predictions being produced.

Example 7-11. Writing results back to the graph projection

```
CALL gds.beta.pipeline.linkPrediction.predict.mutate(
    'actors-input-graph-for-prediction', {
      modelName: 'actors-model',
      relationshipTypes: ['ACTED_WITH'],
      mutateRelationshipType: 'SHOULD_ACT_WITH',
      topN: 20,
      threshold: 0.4
})
YIELD relationshipsWritten, samplingStats
```

Once the code in Example 7-11 has executed, you are left with an enriched projection, but the recommendations have not yet been written back into the knowledge graph.

If you don't want to persist the recommendations from your model, you don't have to. Simply switch out the procedure call `gds.beta.pipeline.linkPrediction.predict.mutate` for `gds.beta.pipeline.linkPrediction.predict.stream` and drop the `mutateRelationshipType` parameter. Your results will now be streamed directly back to the user without enriching the projection.

You can see how this is achieved in Example 7-12. You stream `SHOULD_ACT_WITH` relationships from the `actors-input-graph-for-prediction` projection and then use some Cypher to take the source and target IDs from the streamed relationships and `MERGE` them back into the underlying knowledge graph.

Example 7-12. Persisting results in the knowledge graph

```
CALL gds.beta.graph.relationships.stream(
  'actors-input-graph-for-prediction',
  ['SHOULD_ACT_WITH']
)
YIELD
  sourceNodeId, targetNodeId

WITH gds.util.asNode(sourceNodeId) AS source, gds.util.asNode(targetNodeId) AS target

MERGE(source)-[:SHOULD_ACT_WITH]->(target)
```

An optional final step is one more piece of wrangling to tidy up the knowledge graph. Since `SHOULD_ACT_WITH` relationships are symmetric, there's no need to have two of them between actors. That is, `(a)-[:SHOULD_ACT_WITH]->(b)` is sufficient, and you don't necessarily need the reciprocal `(b)-[:SHOULD_ACT_WITH]->(a)`. If you want to normalize the knowledge graph in this way, the Cypher code is presented in Example 7-13.

Example 7-13. Normalizing the knowledge graph

```
MATCH (a:Person)-[:SHOULD_ACT_WITH]->(b:Person)-[d:SHOULD_ACT_WITH]->(a)
// Prevents a node matching as a first then b later, so deleting all rels
WHERE id(a) > id(b)
DELETE d
```

Once any wrangling is complete, the knowledge graph is enriched and ready for querying. A simple Cypher query like that shown in Example 7-14 is all that is needed to retrieve a set of recommendations. It just finds people connected by the `SHOULD_ACT_WITH` relationship that came from executing the predictive model over the graph and returns those people's names.

Example 7-14. Querying the knowledge graph for complementary actors

```
MATCH (a:Person)-[:SHOULD_ACT_WITH]->(b:Person)
RETURN a.name, b.name
```

After running Example 7-14, you'll obtain results like those in Table 7-1.

Table 7-1. Recommendations for actors to work with other actors

	a.name	b.name
0	Naomie Harris	Emile Hirsch
1	Naomie Harris	John Goodman
2	Naomie Harris	Susan Sarandon
3	Naomie Harris	Matthew Fox
4	Naomie Harris	Christina Ricci
5	Danny DeVito	Demi Moore
6	Danny DeVito	Noah Wyle
7	Danny DeVito	Kevin Pollak
8	Danny DeVito	James Marshall
9	Danny DeVito	Christopher Guest
10	Danny DeVito	Aaron Sorkin
11	John C. Reilly	Demi Moore
12	John C. Reilly	Noah Wyle
13	John C. Reilly	Kevin Pollak
14	John C. Reilly	James Marshall
15	John C. Reilly	Christopher Guest
16	John C. Reilly	Aaron Sorkin
17	Ed Harris	Sam Rockwell
18	Lori Petty	Julia Roberts

As these recommendations are persisted in your knowledge graph, you can reuse them and combine them into other queries, despite them being created by ML. Bear in mind that you'll need to repeat this process periodically (hourly, daily, monthly) to keep the data fresh depending on your use case needs. Finally, remember this is just a starting point, and the real time is spent optimizing parameters for the model, especially around properties, features, and metrics for the trained model. While the auto ML can help, human intelligence is still needed to bootstrap artificial intelligence!

If the scope of the in-graph ML pipelines isn't sufficient for your needs, you can always fall back on external tools like TensorFlow, PyTorch, and scikit-learn or cloud-hosted systems like Google's Vertex AI, Amazon SageMaker, and Microsoft's Azure Machine Learning. The bridge between the two domains is the set of feature vectors extracted from the knowledge graph which is used as input to train and test the externally hosted model. Through experimentation and iteration (either automatically or manually), it may be possible to use the greater depth of these tools and platforms to create better ML models, at the expense of implementation complexity, especially enriching the knowledge graph with the results from those external systems.

Summary

At this point in the book, you've covered enough of the technical details of building knowledge graphs to be competent. From creating and querying knowledge graphs, through applying graph algorithms, and finally using graph ML to enrich your data, you have covered a great deal of fundamental technologies. It's no mean feat to have come this far, so well done! From here on, you'll be applying your technical skills to specific knowledge graph use cases, starting with mapping the data in your enterprise.

CHAPTER 8

Mapping Data with Metadata Knowledge Graphs

Modern enterprises are extremely rich in data. But that data is distributed across silos, heterogeneous, and often of variable quality.

Being able to understand where your data resides, how it's processed, and who consumes it is an important part of running an enterprise. It's a key component of data governance and increasingly important for self-serve data consumers.

A metadata knowledge graph is an enterprise-wide map which records the shape and location of data, the systems which process that data, and its consumers. Importantly, a metadata knowledge graph links data, processes, and consumers so that the provenance of data is explicit and easy to reason about (e.g., for compliance and regulatory demands).

In this chapter, you'll learn about the challenges of stewarding data in the modern enterprise and how a metadata knowledge graph can help. You'll see how the modern distributed data landscape can be (logically) reunified by a metadata knowledge graph and how complex systems architectures can be tamed with the same technique. This is reinforced by an end-to-end example designed to resemble a typical enterprise. But first, the challenge of data stewardship must be addressed.

The Challenge of Distributed Data Stewardship

As organizations evolve, different departments implement applications and processes to solve their needs. Individual departments may store some of the same information, so it is not uncommon for duplicate or (worse still) nearly duplicate data to reside in silos.

Often these systems are implemented without consideration of the overall knowledge-organization needs of the business since that isn't owned by any individual department (and may not have an explicit owner at all). With data in a patchwork state, it's hard to understand a customer's journey or the success factors for a space mission.

Metadata (data that describes data) is a critical element in modern information systems management. A metadata map provides a lens over an organization's entire data ecosystem, allowing you to manage data across the enterprise. As such, metadata knowledge graphs are a very useful foundational layer in your information architecture.

High-quality metadata ensures that users can navigate vast, heterogeneous data assets whose quality and freshness are variable. It helps data analysts, data scientists, and ML engineers access the right data quickly. And it can explain the state of data assets for auditing and regulatory purposes.

But managing metadata has been and is a significant challenge. To address this challenge, some of the largest organizations are converging their data into metadata hubs backed by a knowledge graph.

Metadata Hubs

The trend for metadata hubs backed by knowledge graphs started in 2017 when Airbnb announced its Dataportal platform at their "Democratizing Data at Airbnb" talk at GraphConnect Europe 2017. This was followed by Lyft with Amundsen, the data discovery metadata engine, and LinkedIn with Datahub. Commercial solutions combining knowledge graphs for metadata management have since followed.

It is not only large organizations that benefit from knowledge graphs for metadata. With the support of frameworks like those in "Metadata Hubs", metadata knowledge graphs can be very helpful for building higher-order knowledge graph systems, for data science, and for compliance and regulatory purposes. The kind of entities captured in metadata knowledge graphs are the common data assets found in a modern data stack and typically include datasets, tasks and pipelines, and data sinks. The following sections serve as guidelines for building your own metadata knowledge graph.

Datasets Connected to Data Platforms

Tables, documents, streams—any data collection is a dataset. Datasets are connected to a system that processes or stores them. They can be broken down into a collection of data field descriptions that describe the public schema of the dataset.

Figure 8-1 shows the subgraph representing a table with customer information stored in Google BigQuery. The `Dataset` node is connected to the `DataPlatform` node through the `source` relationship. It has three `Field` nodes.

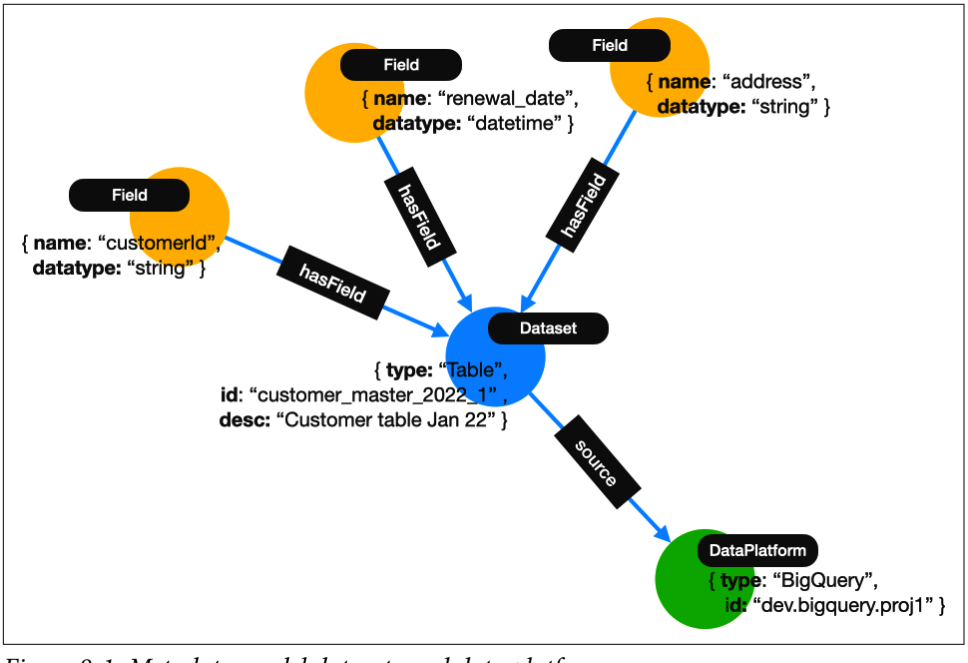


Figure 8-1. Metadata model datasets and data platforms

As you incorporate metadata like that in Figure 8-1, you begin to create a map of your data as it exists in systems. From there, it is possible to link logically common records across systems as well as to show how inputs from one system become outputs to another via tasks and pipelines.

Tasks and Data Pipelines

A *task* is any data job that processes a data asset. A simple example is an ETL task that, for example, standardizes the values of the `country` field in a CSV file. Tasks can be grouped in chains that form data pipelines or data flows. The chain determines the order in which tasks are executed and therefore makes the dependencies between tasks explicit, as in Figure 8-2.

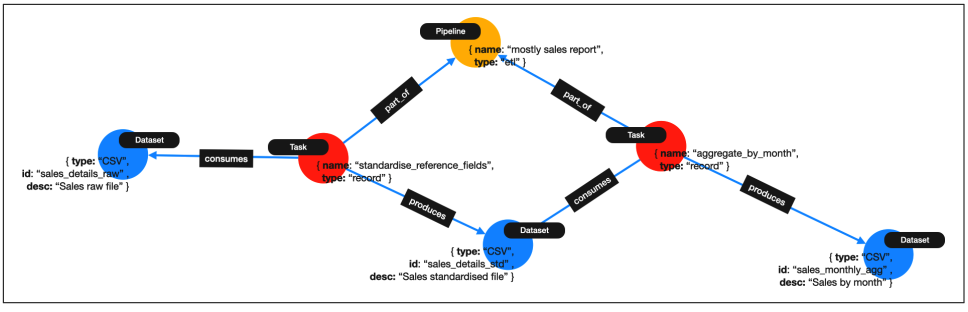


Figure 8-2. Tasks and data pipelines

Figure 8-2 shows how output data from a system becomes input to another. It can be used to understand which systems and users touch the data (e.g., for compliance and regulatory reasons) as well as to aid understanding of the provenance of the data. Once the start and end points for the pipeline are understood, they can be fed back as shortcut links into the data set part of the metadata graph from Figure 8-1.

Data Sinks

A *data sink* is any kind of terminal data consumer task, from a data visualization in a business intelligence tool to an ML training dataset. Data sinks are connected to datasets, but unlike the tasks described earlier, they don't produce new datasets. All elements in the metadata graph can be linked to data domains, have owners, and be connected to `catalog terms` representing related business terms from an enterprise ontology, glossary, or standard terminology in use, with descriptions or documentation attached to them.

Mapping data sinks to data allows you to understand which users and systems consume the data. It helps with planning for systems migrations and shows which data and pipelines are valued (and which aren't).

Metadata Graph Example

An example describing a data pipeline for standardizing and aggregating sales data by month is shown in Figure 8-3. Three datasets represent sales data in CSV format: the raw dataset (`sales_details_raw`) extracted from an operational system, the dataset after the standardization of some reference fields (`sales_details_std`), and the dataset aggregated by month (`sales_monthly_agg`).

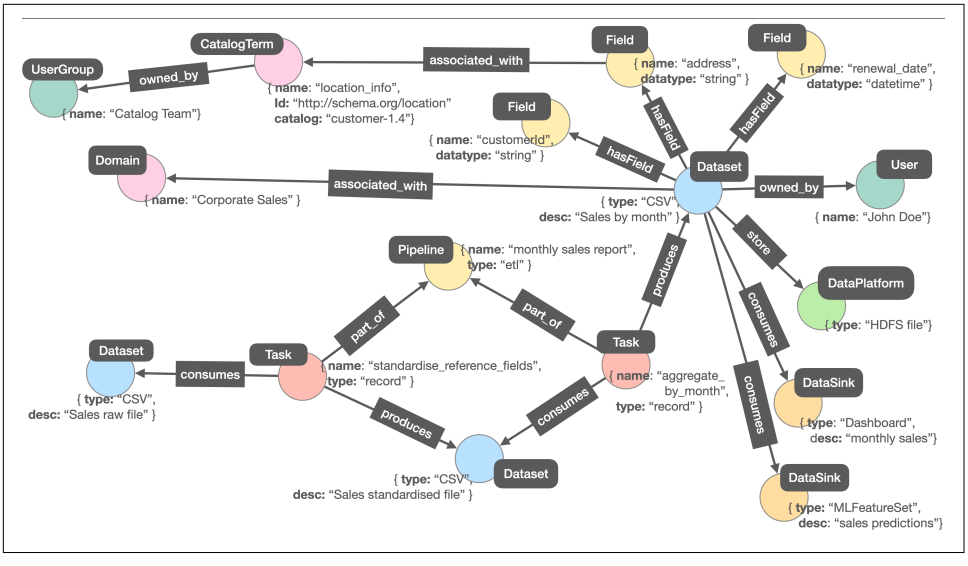


Figure 8-3. A complete view of a data pipeline

The aggregated dataset has nodes for three fields (`customerId`, `address`, and `renewal_date`). The `address` field is associated with a catalog term `location_info`, indicating that the field contains geolocation information. The catalog is owned by the `Catalog Team`.

The dataset itself is also connected to a user acting as the `owner`, although this kind of relationship can and should be qualified with an attribute indicating the role of the user, such as a data steward. Finally, the aggregated dataset is consumed by two data sinks: a `Dashboard` representing some kind of data visualization and an `MLFeatureSet` indicating that the dataset is used for an ML pipeline.

Querying the Metadata Graph Model

A model like Figure 8-3 can support data discovery queries like "What's the most popular corporate sales dataset containing customer location information?" One way to measure a dataset's popularity is to count the ways in which the data is consumed, by the number of data sinks consuming the dataset. That metric can be retrieved from the metadata graph using the Cypher query in Example 8-1.

Example 8-1. A Cypher query that measures dataset popularity by the number of consumers using that data

```
MATCH (d:Dataset)
WHERE (d)-[:associated_with]->(:Domain { name: 'Corporate Sales'}) AND
      (d)-[:has_field]->(:Field)-[:associated_with]->
```

```
      (:CatalogTerm {name:'location_info'})
RETURN  d.id AS dataset_id,
        d.desc AS dataset_desc,
        d.type AS dataset_type,
        count{ (d)<-[:consumes]-(d:DataSink) } AS dataset_usage_count
```

Similarly, you can explore the metadata graph to assess the impact of a failing task. In Example 8-2 you can see a query that returns the list of data consumers (`data sinks`) and their owners who would be affected by a given task failing.[1]

Example 8-2. Query that returns a list of data sinks and their owners

```
MATCH (t:Task)-[:produces|consumes*2..]-(:Dataset)
    <-[:consumes]-(s:DataSink)-[:owned_by]->(o)
WHERE t.name = 'standardise_reference_fields'
RETURN  s.id AS affectedDataConsumerID,
        s.type AS affectedDataConsumerType,
        s.desc AS affectedDataConsumerDesc,
        o.id AS ownerID,
        o.name AS ownerName
```

Another common type of analysis that uses a metadata knowledge graph is data lineage. This use case is the inverse of the previous one from a graph point of view. The simplest data lineage question could be "Which data platforms are the sources of the data for Dashboard X?" The answer comes from the Cypher query in Example 8-3.

Example 8-3. A simple data lineage query

```
MATCH (s:DataSink)-[:consumes]->(:Dataset)-[:produces|consumes*2..]->(raw:Dataset)
    -[:source]->(dp:DataPlatform)
WHERE s.type = 'Dashboard' AND s.id = 'X'
RETURN  raw.id AS sourceDatasetID,
        raw.type AS sourceDatasetType,
        dp.id AS sourcePlatformID,
        dp.type AS sourcePlatformType
```

There are many more questions related to metadata management that you could ask such a rich dataset. You can even analyze the graph using graph algorithms, like in Chapter 6, to surface strong connections (or disconnections) between teams and data domains because a map of how data flows in an organization (and that is exactly what this knowledge graph is) is a proxy for the operations of a company.

1 Chapter 11 gives more detail on impact analysis.

Using Relationships to Connect Data and Metadata

In a graph, you define relationships to describe how entities interrelate. For example, it is possible for a graph to state that a customer subscribes to a service by linking the customer node and the service node through a `SUBSCRIBES_TO` relationship. But relationships can also be used to connect data and metadata in the same graph.

Figure 8-4 shows a graph with two layers. The bottom layer shows the data pertaining to customers and the services to which they subscribe. The top layer has some metadata that describes the provenance of the customer and subscription data. This metadata shows how customer information is part of a dataset sourced from an external system, which is curated by a particular data steward.

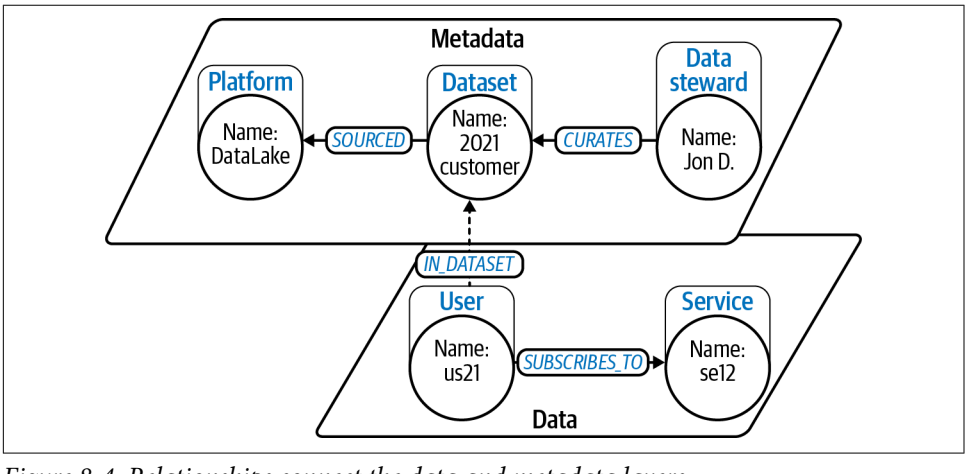


Figure 8-4. Relationships connect the data and metadata layers

Using a knowledge graph like the one in Figure 8-4, you can answer domain questions like "Which customers subscribe to service X?" You can also provide confidence in the answer by supplying provenance and governance metadata. Importantly, data architects can implement this technique in a noninvasive manner with respect to the source systems containing customer data, by building it as a layer above those systems.

A globally linked view of data unlocks many significant use cases. Figure 8-4 shows the powerful combination of the data and metadata planes in the graph, but it's easy to see how to construct a complete view of a customer by linking disparate partial views of customers across different systems. You could enrich the original customer view with the customer's behavior over their lifetime, detect patterns for good and bad customers, and adapt processes to suit their behavior.

At the metadata level, you can start with a catalog of well-described datasets connected to the data sources where they are hosted. But you can also use the techniques in

the rest of this chapter to enrich them by linking them to vocabularies, taxonomies, and ontologies to enable interoperability. Or you can add ownership or stewardship information to enhance data governance and map where data came from as well as which systems and individuals have handled it. A unified, deduplicated view of data enabled by a metadata graph is a solid foundation for several use cases, including semantic search.

Summary

This chapter has discussed the notion of metadata graphs as a foundational layer to track enterprise data assets, processes, and consumers. By integrating relevant data and metadata in a knowledge graph and defining the relationships within the data itself, you have seen how to push intelligence into the data. With metadata knowledge graphs, you've seen how to create knowledge graphs that allow consumers to search enterprise data for terms that don't even explicitly exist in text!

Given a metadata knowledge graph, you can discover, reason about, and steward data assets across the enterprise. While this is useful on its own, it can also serve as the basis for higher-level use cases. In the next chapter, you'll encounter some of those use cases as you look for fraudsters, anomalies, and skills and create recommendations.

CHAPTER 9

Identity Knowledge Graphs

Being able to confidently identify a person or thing is a bedrock of business information systems. Given two records, can you determine if they represent the same real-world thing (a person, an organization, a place, and so on)?

This problem commonly arises when integrating data from different systems, when you need to reason that a record in one system has a counterpart (or not) in the other. In that context, it is often referred to as *entity resolution* or *master data management*.

In this chapter, you will learn how knowledge graphs help you address the problem of identity. You will see how the topology of a knowledge graph can give you the confidence to link multiple entities and allow you to reason about them as if they were a single golden record.

Knowing Your Customer

Identity and its myriad of difficulties is part of our everyday experience. For example, Jane Coleman has been a happy and profitable customer of the Bank's credit card for several years.

Some months ago, Jane married her husband Peter and decided to take his name. Jane Downe and Peter Downe are the owners of a joint checking account with the Bank. But how can the Bank know that Jane Coleman and Jane Downe are the same person? There's no obligation on Jane to tell the Bank; indeed, she is free to continue to use her maiden name (a not unusual practice among certain professionals like doctors).

As a consequence, Jane regularly gets letters in her mailbox inviting her to open a credit card account, which she already has. This is clearly a waste of time and money, and probably a modest annoyance to Jane. But were Jane to be a fraudster, then

tracking down all her activities would be hampered by not being able to discern a single identity.

When records have a consistent strong identifier, then it's easy to connect them. It's one of the reasons why database practitioners are so concerned with things like unique identifiers to help solve that problem. At a human level, think of things like a Social Security number or a passport number for an individual, a product's unique SKU, or a unique company identifier issued by a government agency.

But not everything is as simple as a unique identifier. There are cases like the one in the banking example described earlier where strong identifiers may exist but are not consistent across all data. Maybe Jane opened her credit card account using her driver's license and the current account using her passport.

The data in Example 9-1 paints a typical picture of the problem. Rows 1 and 2 are easy to link. You can see they have the same `Passport` entry, so you can safely reason that Pete Downe and Peter J. Downe are the same person.

Example 9-1. Two records representing the same product

```
rowid, Source, AccountNo, Name, Passport, DriversLic, DOB
1, "Credit", 9918475, "Pete Downe", "VX83041",, 1987-03-12
2, "Current", 2436930, "Peter J. Downe", "VX83041",, 1987-03-12
3, "Credit", , "Jane Coleman", ,"49587640", 1989-10-28
4, "Current", 2436930, "Jane Downe", "RA14958", , 1989-10-28
```

Rows 3 and 4 in Example 9-1 are much harder to discern. There are different names (Jane's maiden and married names) and no common alphanumeric identifiers. Given just this data, you'd have to conclude there's a strong possibility of two people being represented in those rows.

Things become even more problematic in the absence of strong identifiers. In those cases, you will need to use whichever features are available, and often a combination, to try to determine if two similar references refer to the same thing.

When Does the Problem Appear?

There are several scenarios where you may face duplicated data. Some of the most common are:

Data integration
: This is probably the most frequent case. Different applications store variants of the same data for different functions, but often the data from multiple systems needs to be combined for analytics purposes. Sales applications, customer relationship management (CRM), marketing, and so on all hold customer data (or product data, or event data…), but that data is collected and managed

independently. This results in records being identified and described in different ways by each, without consistent strong identifiers across systems.

Any analytic workload (analysis, reporting, predictive model building) of the integrated data is unreliable without first having solved the duplication problem. Duplicate records must be identified in order to merge them into master data entities (golden records) to establish a single trusted view.

Anonymous activity
: Many publicly accessible systems allow users to interact with them without authenticating (e.g., to browse products). By definition, there is no authentication of the user that can be used to identify them, but users do leave traces (clickstream) of their activity and the devices they use to access those systems. These traces will include features like time, cookie IDs, and even the navigational choices they made, which can help identify patterns and eventually unique users and sessions.

Intentional/fraudulent duplicates
: Sometimes duplicates are created intentionally when bad actors are trying to beat a system. Examples include:

  - Individuals trying to get better insurance quotes by slightly adjusting some parameters like their address, their age, or others
  - In more sophisticated cases, users trying to abuse service trials by authenticating with different credentials (fake emails or names, etc.)
  - Online sellers posting multiple small variants of the same product listing in an attempt to maximize its chances of being seen and ultimately sold

All these scenarios share the same fundamental question: *given two records, do they refer to the same or to different entities?*

Graph-Based Entity Resolution Step by Step

The process of entity resolution includes three fundamental activities:

- Data preparation
- Entity matching
- Curation of a persisted record of master entities

You can see examples of each in Example 9-2. You have three overlapping data sets with personal information that need deduplication and integration into a single view.

Example 9-2. Three sample files

file: ds1.csv

system_id	full_name	email	ssn	passport_no	yob
"1_1"	"Sidney Bernardy"	"sbernardy0@va.gov"	"252-13-7091"	"A465901"	"1979"
"1_2"	"Ernestine Ouchterlony"	"eouchterlony1@sun.com"	"557-21-3938"	"CF3586"	"1999"
"1_3"	"Cherish Gosnall"	"cgosnall2@google.com.br"	"422-45-7305"	"BS945813"	"1983"
"1_4"	"Husein Sprull"	"hsprull3@va.gov"	"123-03-8992"	"FG45867"	"1980"
"1_5"	"Jonathan Pedracci"	"jpedraccig@gateway.com"	"581-96-2576"	null	"1991"

file: ds2.csv

system_id	name	email	ssn	passport_no	age
"2_1"	"Bernardy, Sydney Joanne"	null	null	"A465901"	"44"
"2_3"	"Gosnall, Cherise"	"cgosnall@outlook.com"	"422-45-7305"	null	"40"

file: ds3.csv

system_id	first_name	last_name	email	ssn	passport_no	dob
"3_1"	"Syd J"	"Bernardy"	"sjb@gmail.com"	"252-13-7091"	null	"1979-05-13"
"3_2"	"C"	"Gosnall"	null	"422-45-7305"	null	"1983-04-23"
"3_3"	"Jon"	"Pedracci"	"pedracci@outlook.com"	null	"TH834501"	"1991-12-05"
"3_4"	"Hana"	"Sprull"	"hsprull3@va.gov"	"465-63-9210"	null	"1984-10-13"

Data Preparation

The purpose of the data preparation phase is to make sure that the data meets the required quality characteristics. When there are multiple data sets from

independently governed systems, the different parts will also need to be harmonized to make them comparable. This will include activities like aligning data types and units (all distances in km, all prices in dollars, all scores between 0 and 1, etc.).

Even when all data comes from a single source and does not require those adjustments, all the features that you intend to use in the matching phase of the process still need to be cleansed, standardized, and so forth to maximize the quality of the results. This means making sure nulls are actually nulls (and not the string "null"), unifying string format (all uppercase or all lowercase and encoding), trimming, removing multiple spaces or special characters, and so on.

The first step is to model the data as a knowledge graph, storing it in a graph database. Since each record represents a person, the model will be the simplest possible. The obvious graph is with a `Person` node for each row and each column providing a property on that node. Though the files are quite similar, there are some file-specific columns (`yob`, `age`, or `dob`) which will introduce some variability in the node structure. This is fine for now and will be dealt with before the entity matching phase.

The Cypher script that loads each record into `Person` nodes is shown in Example 9-3. This generic script takes the structure (columns) in the source file and adds a property called `source` for data provenance.

There is never a single way of doing things in data management. Part of what was done in this phase could legitimately be carried out before the data is loaded into the graph. The data preparation work can be executed on the files directly using Python (and pandas is a great tool for this) or as SQL transformations on a cloud data warehouse after loading the files into tables. The scripts in Example 9-3 would just be loading into the graph database management system (DBMS) data in a format that is ready for entity matching (with the exception of the graph features that can only be extracted once the data is modeled as a graph). Simplicity, robustness, and performance should guide you when choosing your approach.

Example 9-3. Loading personal data files

```
LOAD CSV WITH HEADERS FROM "file:///ds1.csv" AS row
CREATE (p:Person) set p.source = "ds1", p += properties(row) ;

LOAD CSV WITH HEADERS FROM "file:///ds2.csv" AS row
CREATE (p:Person) set p.source = "ds2", p += properties(row) ;

LOAD CSV WITH HEADERS FROM "file:///ds3.csv" AS row
CREATE (p:Person) set p.source = "ds3", p += properties(row) ;
```

The most obvious difference between the nodes generated from the different files is the level of detail in the customer's date of birth information. While nodes from data set 1 include a year of birth (`yob`) property with a four-digit string representation of the year,[1] nodes from data set 2 will have a string property called `age` with a representation of the current age of the customer, and nodes from data set 3 will have a `dob` property with a string containing the customer's complete date of birth. To make these pieces of information comparable for the entity matching phase, the lowest common denominator in terms of information is the year of birth.

The Cypher script in Example 9-4 shows how to create a new property called `m_yob` (for matching year of birth) of type `integer` in all nodes. Notice how the `source` property is used to determine the type of transformation logic to be applied in each case, and number-casting functions, date functions, and arithmetic operators are used on the original data.

Example 9-4. Normalizing dates in personal data files

```
MATCH (p:Person) WHERE p.source = "ds1" SET p.m_yob = toInteger(p.yob) ;

MATCH (p:Person)
  WHERE p.source = "ds2" SET p.m_yob = date().year - toInteger(p.age) ;

MATCH (p:Person) WHERE p.source = "ds3" SET p.m_yob = date(p.dob).year ;
```

The `name` element also needs alignment, as each data set uses a different way to store names. You can see separate fields for first and last name, CSVs, and a single string containing the full name. The Cypher script in Example 9-5 creates a new property called `m_fullname` (for matching full name) in all nodes with type `string` and a "canonical" representation of a customer name: `firstname lastname`. The script uses the `source` property to apply the relevant logic implemented by using string functions (`trim`,`toLower` and tokenizer `split`).

Example 9-5. Normalizing names in personal data files

```
MATCH (p:Person)
  WHERE p.source = "ds1" SET p.m_fullname = toLower(trim(p.full_name)) ;

MATCH (p:Person) WHERE p.source = "ds2"
WITH p, split(p.name,",") AS parts
  SET p.m_fullname = toLower(trim(parts[1]) + ' ' + trim(parts[0]));

MATCH (p:Person) WHERE p.source = "ds3"
  SET p.m_fullname = toLower(trim(p.first_name) + ' ' + trim(p.last_name)) ;
```

1 `LOAD CSV` reads all fields until coerced to other types.

Once the graph data has been harmonized and normalized, it is ready for the application of the matching rules.

Blocking Keys

Optionally, the data preparation phase may include the generation of blocking keys. *Blocking* is a technique to reduce the search space and so improve performance. When looking for duplicates, you will need to compare every record to every other record requiring n^2 operations for n elements.

To reduce this to a more manageable computational workload, you need to avoid comparing pairs that are not likely to produce matches. This is the job of the blocking keys.

Only candidates with the same blocking key are compared with each other. Sometimes features in the data can be used as blocking keys. For instance, you could use postal codes and then only compare records representing entities within a reasonable proximity.

Where "natural" blocking keys don't exist, you can still generate them synthetically, including using algorithms or ML models to suggest them.

Entity Matching

Matching is the central activity in the entity resolution process, where the identity logic is applied. You will define when two entities are considered to be the same, based on your business heuristics. This is done through the application of a set of rules based on the features identified and prepared in the preparation phase. These rules can vary in complexity but generally will be based on exact matches (useful for strong identifiers), will approximate matches based on a definition of distance (numeric, dates, embeddings), and may also introduce elements of fuzziness (string similarity, value approximation, etc.). Some rules will give unambiguous matches (e.g., a strong identifier) while others will provide weightings for probabilistic scoring.

The following algorithm provides a systematic approach to applying the matching rules:

1. Create a `SAME_AS` relationship between nodes for which there is a match on a strong identifier feature.
2. Create a weighted `SIMILAR` relationship between nodes for which there is a match above threshold on a weak identifier feature and a `SAME_AS` relationship does not exist between the two nodes.

3. Discard the SIMILAR relationships for which there is a mismatch in one of the strong identifier features. The weight in the relationship will be the similarity score.
4. Apply a correcting factor to the weight in a SIMILAR relationship when there is a match above threshold on a nonidentifying feature.
5. Discard the SIMILAR relationships with a similarity score below a minimum threshold.

The application of these rules generates a graph of related nodes connected through SAME_AS and SIMILAR relationships in which connected nodes will identify unique entities. These four steps implicitly define three types of features: strong identifiers, weak identifiers, and nonidentifying features.

In Example 9-6, the two strong identifiers in the data set are the passport number (passport_no) and the Social Security number (ssn). The application of step 1 will instantiate a relationship of type SAME_AS between any pair of Person nodes that share the same value for either of the two strong identifiers.

Note the introduction of the filter on id(p1) > id(p2), avoiding the comparison of p1 versus p2 and then the superfluous p2 versus p1. This is because the SAME_AS relationship is symmetrical (if p1 is the same as p2, then p2 is the same as p1), so there is no need to compare them in both directions and create two SAME_AS relationships between the nodes.

Example 9-6. Creating SAME_AS relationships between nodes matching on strong IDs

```
MATCH (p1:Person), (p2:Person)
WHERE p1.source <> p2.source
  AND (p1.ssn = p2.ssn OR p1.passport_no = p2.passport_no)
  AND id(p1) > id(p2)
CREATE (p1)-[:SAME_AS { ssn_match : p1.ssn = p2.ssn,
              passport_match : p1.passport_no =
              p2.passport_no}]->(p2)
```

The script in Example 9-7 shows an implementation of step 2, taking care of the weak identifiers. The m_fullname property is a weak identifier that was created by amalgamating data across sources and by standardizing all name information available. Normally, a match on the name is not sufficient to derive that two persons are the same. That's why it is treated as a secondary identifier, and a SIMILAR relationship is created only when a match above threshold is found. In Example 9-7, one standard string similarity metric called *Jaro–Winkler distance* is used, and when that metric is below 0.2 (depending on the use case), the relationship is created and annotated with a similarity metric.

Example 9-7. Creating similar relationships between nodes matching on weak IDs

```
MATCH (p1:Person), (p2:Person)
WHERE NOT (p1)-[:SAME_AS]-(p2)
      AND p1.source <> p2.source
      AND id(p1) > id(p2)
      AND apoc.text.jaroWinklerDistance(p1.m_fullname, p2.m_fullname) < 0.2
CREATE (p1)-[:SIMILAR { sim_score : 1 -
  apoc.text.jaroWinklerDistance(p1.m_fullname, p2.m_fullname)}]->(p2)
```

In Example 9-7 the metric is inverted because it measures distance instead of similarity, so to trigger the creation of the relationship, the actual value needs to be under a given distance value instead of over threshold as stated earlier.

The similarity score (`sim_score`) used to qualify the `SIMILAR` relationship is created by subtracting the Jaro–Winkler distance from 1. It is important to keep consistent semantics in the score used in this metric to make them comparable and composable later on. The script in Example 9-7 sets a score between 0 and 1.0. The higher the score, the stronger the similarity.

Step 3 is about filtering detected similarities based on weak identifiers (those materialized by `SIMILAR` relationships) by discarding the ones that exist between entities with discrepancies in the strong identifier features. This is what the Cypher script in Example 9-8 implements.

Example 9-8. Removing similar relationships between nodes matching on weak IDs when they differ on strong ones

```
MATCH (p1:Person)-[sim:SIMILAR]->(p2:Person)
WHERE p1.ssn <> p2.ssn OR p1.passport_no <> p2.passport_no
DELETE sim
```

Step 4 applies a correcting factor to the weights in the `SIMILAR` relationships. The code in Example 9-9 provides an example of this. For every pair of nodes at both ends of a `SIMILAR` relationship, if there is a reasonable similarity on the year of birth (`m_yob`, a nonidentifying feature in this case), then the similarity score will be adjusted up or down.

The logic in steps 3 and 4 can also be implemented together with step 2 and, in some cases, achieve an improvement in performance by running a single pass on the data. The disadvantage would be the lack of detailed traceability of the effect of each step. In this chapter, the steps are separated for clarity.

The logic in Example 9-9 is based on an absolute threshold (`abs(p1.m_yob - p2.m_yob) > $yob_threshold`). If the years of birth are not too dissimilar, the

similarity score is increased by 10%, but if the difference is too significant, then the score is decreased by 10%. The adjustment function can be as sophisticated as you need for your use case.

Example 9-9. Boosting or damping similar relationships between nodes matching on weak IDs when they differ on other features

```
MATCH (p1:Person)-[sim:SIMILAR]->(p2:Person)
WITH sim, abs(p1.m_yob - p2.m_yob) AS yob_diff
SET sim.sim_score = sim.sim_score * CASE WHEN yob_diff > $yob_threshold
  THEN .9 ELSE 1.1 END
```

Step 5 removes the `SIMILAR` relationships that, after applying the correcting factor in the previous step, would result in a similarity score below acceptable. The code in Example 9-10 shows how this can be implemented straightforwardly using Cypher.

Example 9-10. Removing similar relationships between nodes matching on weak IDs when below minimum score

```
MATCH (p1:Person)-[sim:SIMILAR]->(p2:Person)
WHERE sim.sim_score < $sim_score_threshold
DELETE sim
```

The result of processing all five steps produces the graph in Figure 9-1.

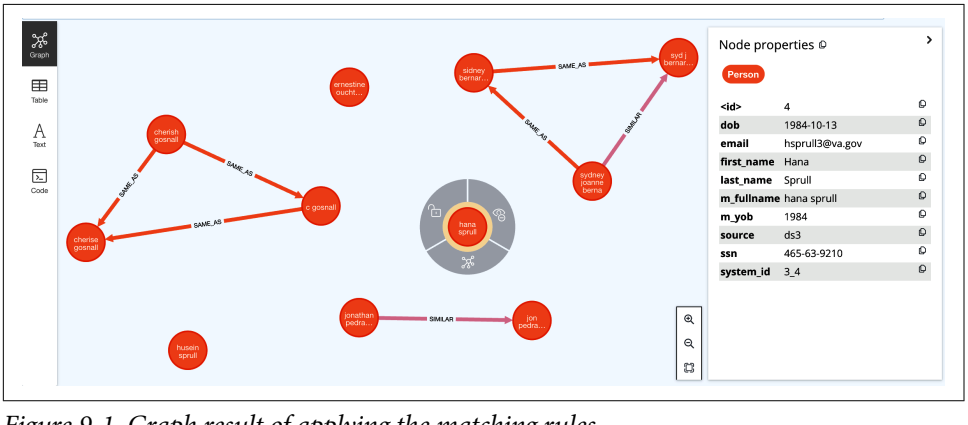


Figure 9-1. Graph result of applying the matching rules

It's worth mentioning that after the application of step 2, additional `SIMILAR` relationships were created, which were subsequently removed when executing step 3—for example, the nodes representing the individuals named Husein Sprull and Hana Sprull. On execution of step 2, a relationship would be created based on the similarity in their names, which would later be removed in step 3 due to the mismatch in their Social Security numbers. This process is shown in Figure 9-2.

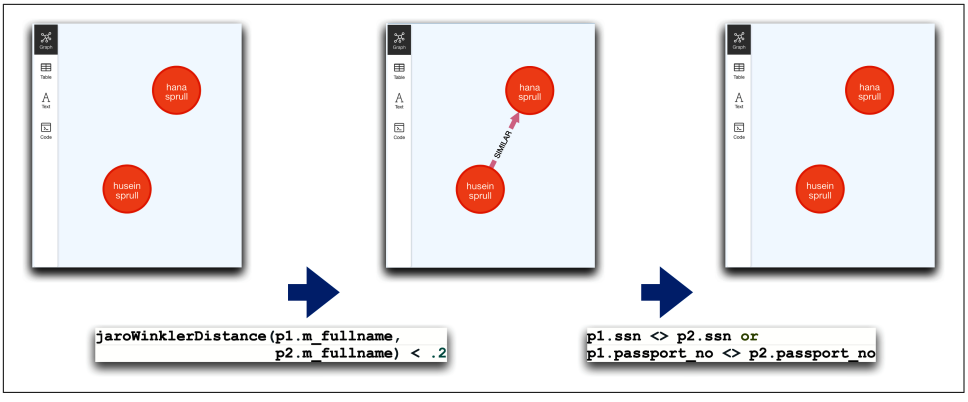


Figure 9-2. Relationship of type `SIMILAR` added and subsequently removed

In the next section, the graph in Figure 9-1 will be analyzed topologically to identify related entities.

Build/Update a Persisted Record of Master Entities

Once the set of related entities has been determined and the relationships made explicit in the graph, the final step is to create a persisted representation of what's typically called *master entities*. Essentially, this means detecting sets of connected nodes in the graph either through `SAME_AS` or `SIMILAR` relationships resulting from applying the matching rules.

All nodes in a set form a connected component and are equivalent to a unique entity. Detecting such sets is what the Weakly Connected Components (WCC) algorithm does. The adjective *weakly* comes from the fact that the algorithm works on undirected graphs. This nicely fits the objective of this exercise because the direction of both `SAME_AS` and `SIMILAR` relationships is irrelevant, as both are symmetric.

You can see how this can be implemented using the concepts from Chapter 6. A prerequisite to the application of any algorithm is the creation of an in-memory projection of the subgraph to be used. This is what the code in Example 9-11 does.

Example 9-11. Creating a projection on `person` nodes and `SIMILAR` and `SAME_AS` relationships

```
CALL gds.graph.project(
  'identity-wcc',
  'Person',
  ['SAME_AS','SIMILAR']
)
```

The projected graph is added to the catalog under the name `identity-wcc` and will include all nodes of type `Person` and all relationships of type `SAME_AS` and `SIMILAR`. A first execution of the WCC algorithm in stream mode (streaming results but not persisting anything in the graph) can be executed with the Cypher fragment in Example 9-12. Note that the invocation of an algorithm refers to the projected graph by its name in the catalog.

Example 9-12. Running the WCC algorithm to identify nodes in the same identity group

```
CALL gds.wcc.stream('identity-wcc')
YIELD nodeId, componentId
WITH gds.util.asNode(nodeId) AS person, componentId AS golden_id
RETURN golden_id, person.m_fullname, person.passport_no, person.ssn
ORDER BY golden_id
```

The results produce an identifier per connected component. For readability, some details about each node in the component are added. You can see that in Example 9-13.

Example 9-13. Results of running the WCC algorithm

"golden_id"	"person.m_fullname"	"person.passport_no"	"person.ssn"
0	"sidney joanne bernardy"	"A465901"	null
0	"sj bernardy"	null	"252-13-7091"
0	"sidney bernardy"	"A465901"	"252-13-7091"
1	"cherise gosnall"	null	"422-45-7305"
1	"c gosnall"	null	"422-45-7305"
1	"cherish gosnall"	"BS945813"	"422-45-7305"
4	"jon pedracci"	"TH834501"	null
4	"jonathan pedracci"	null	"581-96-2576"
5	"hana sprull"	null	"465-63-9210"
7	"ernestine ouchterlony"	"CF3586"	"557-21-3938"
9	"husein sprull"	"FG45867"	"123-03-8992"

Once you're happy with the results, you can persist them in the knowledge graph. You could simply use the *write* version of the algorithm (`gds.wcc.write`), which annotates each node with the connected component identifier, and then use that to create master entities.

Instead, the Cypher script in Example 9-14 uses the output of the *stream* version of the algorithm to create a customized persisted representation of the connected components. The subgraph created is itself a representation of the master entity (the golden record). The same script connects all matched records to the master entity, enabling traceability.

Example 9-14. Creating entity nodes (golden records) from the result of the WCC algorithm and linking to original ones

```
CALL gds.wcc.stream('identity-wcc')
YIELD nodeId, componentId
WITH gds.util.asNode(nodeId) AS person, componentId AS golden_id
MERGE (pg:PersonMaster { uid: golden_id })
    ON CREATE SET pg.fullname = person.m_fullname,
                  pg.ssn = person.ssn, pg.passport_no = person.passport_no
    ON MATCH SET pg.ssn = coalesce(pg.ssn,person.ssn),
                 pg.passport_no = coalesce(pg.passport_no,person.passport_no)
MERGE (pg)-[:HAS_REFERENCE]->(person)
```

Once the entity nodes have been created and linked to the original references, the identity graph will resemble Figure 9-3.

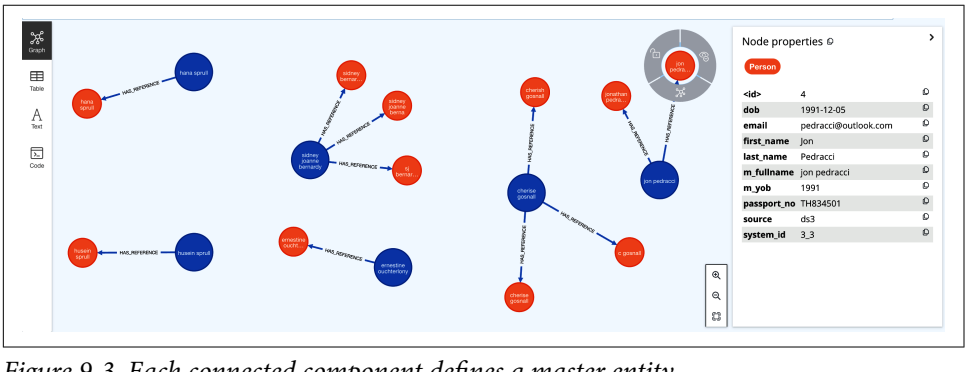


Figure 9-3. Each connected component defines a master entity

Graphs like Figure 9-3 can be queried to produce an aggregated view of a master entity, including all details by source. This is shown in Example 9-15.

Example 9-15. Querying the master entities and producing a detailed summary of all the related references

```
MATCH (p:PersonMaster)-[:HAS_REFERENCE]->(ref)
WHERE p.passport_no = 'A465901'
WITH p, collect( { source: ref.source , details : properties(ref)}) AS refs
RETURN { master_entity_id : p.uid, references: refs }
```

For interoperability, you can also produce the JSON serialization of the results directly from the Cypher query, as in Example 9-16.

Example 9-16. JSON serialization of an aggregated view of a master entity

```
{
  "master_entity_id": 1,
  "references": [
    {
      "details": {
        "m_fullname": "sydney joanne bernardy",
        "system_id": "2_1",
        "m_yob": 1979,
        "name": "Bernardy, Sydney Joanne",
        "passport_no": "A465901",
        "source": "ds2",
        "age": "44"
      },
      "source": "ds2"
    },
    {
      "details": {
        "m_fullname": "sidney bernardy",
        "full_name": "Sidney Bernardy",
        "system_id": "1_1",
        "m_yob": 1979,
        "passport_no": "A465901",
        "source": "ds1",
        "yob": "1979",
        "email": "sbernardy0@va.gov",
        "ssn": "252-13-7091"
      },
      "source": "ds1"
    },
    {
      "details": {
        "m_fullname": "syd j bernardy",
        "dob": "1979-05-13",
        "system_id": "3_1",
        "m_yob": 1979,
        "last_name": "Bernardy",
        "source": "ds3",
        "first_name": "Syd J",
```

```
        "email": "sjb@gmail.com",
        "ssn": "252-13-7091"
      },
      "source": "ds3"
    }
  ]
}
```

There are two final important things to note.

The data sources to which the entity resolution is applied tend to be living and change over time. The process is rarely a one-off exercise and will usually require future iterations. In cases where the data sets being deduplicated are large, it is common to get the deltas (new elements and removed ones) at regular intervals instead of full new versions of the data sets. This complicates things because it means that the addition (or removal) of new data points from a data set may affect previous matches. It may also create new or alter existing connected components. To accommodate this, a preliminary step is needed where any removed elements in the delta are removed from the graph.

It is often helpful to use information from the graph to improve entity resolution results. The approach can be extended to include properties in relationships or in neighboring nodes. You can also include structural metrics like the degree, centrality, or other graph measure. In fact, one of the most powerful things about graphs is that all these features are automatically available for entity resolution.

Working with Unstructured Data

The absence of strong identifiers in the data will force you to do more sophisticated work on the weak identifiers and the nonidentifying features. These will often be text descriptions for which only the string similarity metrics have been proposed so far. What other options do graph offer?

The Amazon-Google product catalog data set provides a good setup for showing how to solve this problem. The data set contains 1,363 product descriptions from Amazon.com and 3,226 from Google products. The common attributes between the two data sources are product name, product description, manufacturer, and price. The data is loaded in the usual way from CSV files in Example 9-17.

Example 9-17. Loading product data

```
LOAD CSV WITH HEADERS FROM "file:///amz.csv" AS row
CREATE (p:Product { sid: row.id }) SET p.source = "AMZ", p += properties(row) ;

LOAD CSV WITH HEADERS FROM "file:///ggl.csv" AS row
CREATE (p:Product { sid: row.id }) SET p.source = "GGL", p += properties(row) ;
```

The preparation phase tokenizes the product name by breaking it into words after lowercasing and removing all nonalphanumeric characters from the text (`replace(tolower(p.name),"[^a-zA-Z0-9]", " ")`). In this example, the `split` function is used, which means that a word is anything separated from other words by a whitespace. Other methods could equally be used, like tokenizer functions in natural language software packages (such as NLTK, spaCy, Hugging Face, and others). The complete Cypher script is in Example 9-18.

Example 9-18. Normalizing and extracting words from all products

```
CREATE INDEX FOR (w:Word) ON w.txt ;

MATCH (p:Product { source : "GGL" })
UNWIND [x in split(apoc.text.replace(tolower(p.name),"[^a-zA-Z0-9]", " "),"  ")
  WHERE x <> "" ] AS txt
MERGE ( w:Word { txt:  txt }) merge (p)-[:includes]->(w) ;

MATCH (p:Product { source : "AMZ" })
UNWIND [x in split(apoc.text.replace(tolower(p.title),"[^a-zA-Z0-9]", " "),"  ")
  WHERE x <> "" ] AS txt
MERGE ( w:Word { txt:  txt }) merge (p)-[:includes]->(w) ;
```

The resulting graph contains the nodes representing the products that need to be deduplicated as well as the ones representing the tokens (the words). This forms a network like the one seen in the excerpt in Figure 9-4. You can see how some tokens are shared across products.

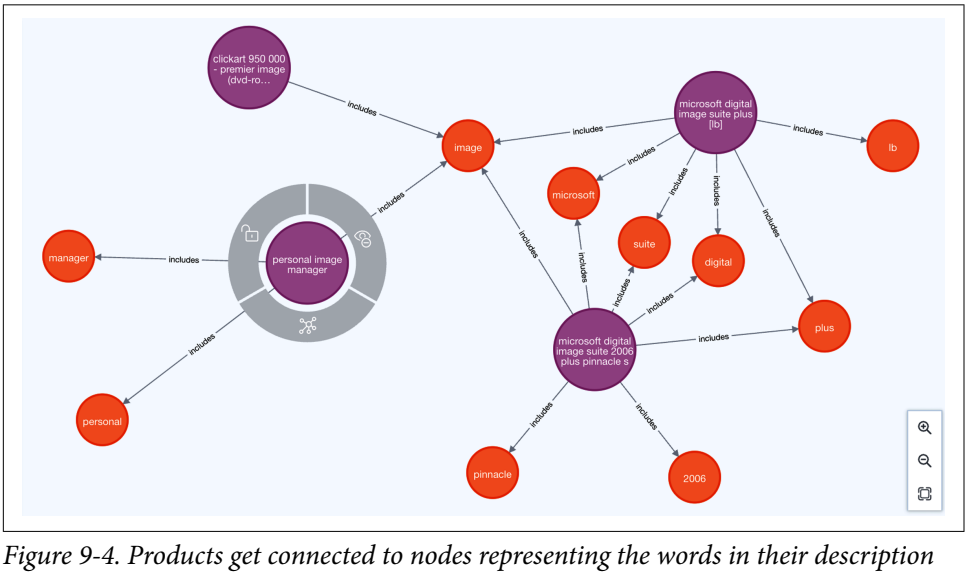


Figure 9-4. Products get connected to nodes representing the words in their description

You can now execute a structural similarity algorithm on the graph in Figure 9-4. The Node Similarity algorithm compares a set of nodes based on the nodes they are connected to. Two nodes are considered similar if they share many of the same neighbors. In this case, that means sharing words in the product name. The algorithm computes pair-wise similarities based on either the Jaccard metric (also known as the Jaccard Similarity Score) or the overlap coefficient (also known as the Szymkiewicz–Simpson coefficient).

The implementation requires the creation of the in-memory projected graph (Example 9-19). This is a bipartite graph including products and words. Its implementation is highly memory optimized so that billions of graph elements can be performantly processed on commodity hardware.

Example 9-19. Creating projection by matching on product title

```
CALL gds.graph.project(
    'identity-sim',
    ['Product', 'Word'],
    ['includes']
);
```

The projection is followed by the execution of the algorithm where it is possible to set a similarity cutoff threshold (`similarityCutoff: 0.8`), discarding all cases where similarity is not high enough to be acceptable. By default, the `gds.nodeSimilarity` algorithm uses the Jaccard metric, but it is possible to override that and use the overlap metric instead (just add the parameter `similarityMetric: 'OVERLAP'`). The code in Example 9-20 uses the `stream` version of the algorithm, producing the results in the Example 9-21 table.

Example 9-20. Running similarity algorithm on projection

```
CALL gds.nodeSimilarity.stream('identity-sim', { similarityCutoff: 0.8 })
YIELD node1, node2, similarity
WITH similarity, gds.util.asNode(node1) AS node1, gds.util.asNode(node2) AS node2
WHERE node1.source = "GGL" AND node2.source = "AMZ"
RETURN similarity AS tk_sim,
       apoc.text.jaroWinklerDistance(node1.name, node2.title) AS str_sim,
       node1.name AS Prod1,
       node2.title AS Prod2
```

The results are shown in Example 9-21. An additional column with string similarity score calculated using the Jaro–Winkler distance has been added to show that the token-based approach is more robust than a string-similarity one. Something as benign as a formatting change in a product name (see last record: `apple apple mac os x server 10 4 7 10 client` versus `mac os x server v10.4.7 10-client`)

could have a significant impact on the string similarity and would be missed by setting a high threshold, depending on use case.

Example 9-21. Entity-matching results from running similarity algorithm

tk_sim	str_sim	Prod1	Prod2
0.8	0.7969	"allume internet cleanup 3.0"	"internet cleanup 3.0"
0.8	0.7831	"marware project x project management software"	"project x project management software"
0.8	0.7703	"allume morpheus photo animation suite"	"morpheus photo animation suite"
0.8	0.7627	"mobile media converter (pc) pinnacle"	"pinnacle mobile media converter"
0.8	0.7593	"extensis suitcase fusion 1u box"	"suitcase fusion 1u box"
0.8	0.7423	"sp linux we 50 lic/cd 3.0c"	"hp sp linux we 50 lic/cd 3.0c (t3586a)"
0.8	0.7423	"sp linux we 50 lic/cd 3.0c"	"hp sp linux we 50 lic/cd 3.0c (t3586a)"
0.8	0.7406	"serverlock manager - 100 servers"	"watchguard serverlock manager (100 servers)"
0.8	0.7234	"print shop deluxe 21"	"broderbund print shop 21 deluxe"
0.8	0.6965	"apple apple mac os x server 10 4 7 10 client"	"mac os x server v10.4.7 10-client"

The next step would be the creation of the `SIMILAR` relationships (because the product name is a weak identifier). The Cypher script in Example 9-22 takes care of this by setting the similarity score as the value returned by the algorithm.

Example 9-22. Matching using product title

```
CALL gds.nodeSimilarity.stream('identity-sim', { similarityCutoff: 0.8 })
YIELD node1, node2, similarity
WITH similarity, gds.util.asNode(node1) AS node1, gds.util.asNode(node2) AS node2
WHERE node1.source = "GGL" AND node2.source = "AMZ"
MERGE (node1)-[:SIMILAR { sim_score : similarity }]->(node2)
```

From here on, the algorithm will continue to build the graph where the connected components yield matching entities.

Case Study: The Meredith Corporation Identity Graph

Meredith Corporation was a media conglomerate with $3.2 billion in annual revenue. With over 30 top consumer brands including *Parents*, *People*, *Real Simple*, and *Coastal Living*, Meredith's digital presence reached more than 180 million users a month across dot-coms, apps, websites, podcasts, and video.

Meredith sought to give users just the right content throughout the day, personalized just for them. But that meant really knowing your user—a challenge when most people don't log in.

Meredith identified anonymous users through unique cookies that drop on the user's device. But cookie loss, diverse devices, and browsers that block cookies by default increased the difficulty of getting a 360-degree view. Even when they work, cookies have a short lifespan.

Graph queries offer rapid answers no matter how large your graph is. But if you are running the same query over and over to build up user profiles, there is a better way: run a graph algorithm over the entire graph.

The Meredith data science team chose the WCC algorithm to identify unique subgraphs (connected components) within the larger graph. In production, the Meredith Identity Graph incorporated more than 20 months of user data from both first- and third-party sources. The database had more than 4.4 terabytes of data for 30 billion nodes, 67 billion properties, and 35 billion relationships.

The average length of touch points exploded from 14 days with a cookie to 241 days with user profiles. Average visits increased from 4 per cookie to 23.8 per profile.

Nearly 350 million profiles that would have been considered unique individuals with different interests and patterns were consolidated into 163 million richer and more accurate profiles. A high-definition view of user interests and preferences fueled stronger models, which led to more relevant content and more users returning over time. It created a virtuous cycle.

Meredith commented, "We basically have increased our understanding of a customer by 20 to 30% by looking at how the data connects over time, rather than just looking at individual cookies themselves. Instead of *advertising in the dark*, we now better understand our customers, which translates into significant revenue gains and better-served consumers."

Summary

Identity is a key pillar for connecting knowledge graphs to the real world. In this chapter, you've learned about strong and weak identifiers for data. You've also discovered how, by using knowledge graphs and graph algorithms, a set of weak identifiers can be aggregated in a strong identifier. With these techniques, you can take on even unruly data sets and truly master that data. Whether that's by connected components, node similarity, or custom graph matching, you have the necessary tools at your disposal. From here, with data quality much improved through mastering, you'll move onto large-scale pattern-matching graphs.

CHAPTER 10

Pattern Detection Knowledge Graphs

Managing enterprise data as a knowledge graph provides plentiful benefits. Data is (logically) centralized, curated, and contextualized. A knowledge graph can be mined for patterns that originate from interesting business events. Those patterns give historical insight into the business, but they can also be used to look forward.

This chapter focuses on finding patterns in knowledge graphs and using those patterns to improve future outcomes. Starting with simple pattern-matching approaches, you'll see how the patterns in your knowledge graph can be easily exploited to stop fraud and build better teams. In each of the use cases, you'll also see how pattern detection can be augmented by graph data science to enrich the knowledge graph and surface new, valuable patterns.

Fraud Detection

Online fraud is a widespread and pernicious problem in the era of ubiquitous computer systems. These systems host some of the most critical aspects of our personal lives and are the lifeblood of modern businesses. They are tempting targets for criminals who might want to defraud individuals or organizations.

The level of that temptation is eye-watering. In the United States alone, fraud cost the productive economy $5.8 billion in 2021 according to the Federal Trade Commission, with around $1 billion being lost in the banking sector alone. Worse, the amount of money lost to fraud is increasing at a staggering 70% year on year. The picture is similar in other advanced economies, with the United Kingdom's banks losing around £700 million ($856 million) to fraud.

But it gets worse. For each $1 lost to fraud, it is estimated that a further $4.23 (over four times more) is lost dealing with attacks and liabilities, loss of reputation and customers, and the general work involved in trying to defend, contain, manage, and redress fraud. It is an expensive problem that necessitates a continuous arms race.

Even marginal improvements in fraud detection bring huge benefits in absolute dollar terms. Fortunately, knowledge graphs are an excellent platform upon which to mount your defense against fraudsters.

First-Party Fraud

Take credit cards as an example. For many of us, they are a necessary and useful part of modern life, but for fraudsters they're a tantalizing opportunity. The value being transacted is huge, and the volume of transactions across cards and providers can provide enough cover for them to hide while leeching money out of the system.

In Figure 10-1, you can see the recent purchases of a credit card customer. Some transactions have been queried for possible fraud, and there is even a declined transaction in the recent history. This happens to all of us from time to time because the systems that process payments are imperfect and human behavior is erratic. Sometimes you really do want to spend thousands of dollars on a laptop, and sometimes you are making payments in a foreign currency because you're on vacation. Sometimes your card has been stolen and a criminal is spending on your card without your permission. With the information available, it's hard to know if this is a good customer, a customer in distress, or a fraudster.

"person"	"card"	"purchase"
{"firstname":"Timmy","id":"2313579","lastname":"Sewell"}	{"number":"1000 0000 0000 0000"}	{"date":"2022-09-02","amount":"71.60","declined":true,"retailer":"Lakeland","id":"ccffc09c-1d79-4e43-9fa9-7699bab21833"}
{"firstname":"Timmy","id":"2313579","lastname":"Sewell"}	{"number":"1000 0000 0000 0000"}	{"date":"2022-09-04","amount":"123.82","declined":"false","retailer":"Monsoon","id":"8c61ee5b-1d8f-4da4-a928-52ddc7295203"}
{"firstname":"Timmy","id":"2313579","lastname":"Sewell"}	{"number":"1000 0000 0000 0000"}	{"date":"2022-09-22","amount":"125.08","declined":"false","retailer":"Boden","id":"33542ef5-01bd-43d8-9d92-6c469003f0bf"}
{"firstname":"Timmy","id":"2313579","lastname":"Sewell"}	{"number":"1000 0000 0000 0000"}	{"date":"2022-10-27","amount":"270.40","declined":"false","retailer":"Lakeland","id":"934cf536-a244-45de-9921-06686a2837a5"}

Figure 10-1. Data in tables obfuscates structure, making it hard to distinguish between customers and fraudsters

Traditionally, finance companies have used rule-based strategies, such as assigning risk scores by searching for certain words, accounts, and locations. Alerts can then be sent to fraud analysts for manual review. This solution is slow, expensive, and inflexible. Conversely, using a knowledge graph to understand customers' broader connected context provides a great deal more insight at scale.

Uncovering Fraud from Data

Loading the tabular data from Figure 10-1 into a knowledge graph is the first step in getting value from it. Figure 10-2 shows a better view of the same customer, which exemplifies the graph data model chosen. The credit card, with its unique card number, is at the center of each subgraph. It is surrounded by purchases made `WITH` the card, and each `Purchase` has a `NEXT` relationship to create a a purchase history. The customer is connected to the card via an `ACCOUNT_HOLDER` relationship and also `LIVES_AT` an address and `OWNS` a phone number. They exist in their own disconnected subgraph, with no addresses or phone numbers held in common between customers.

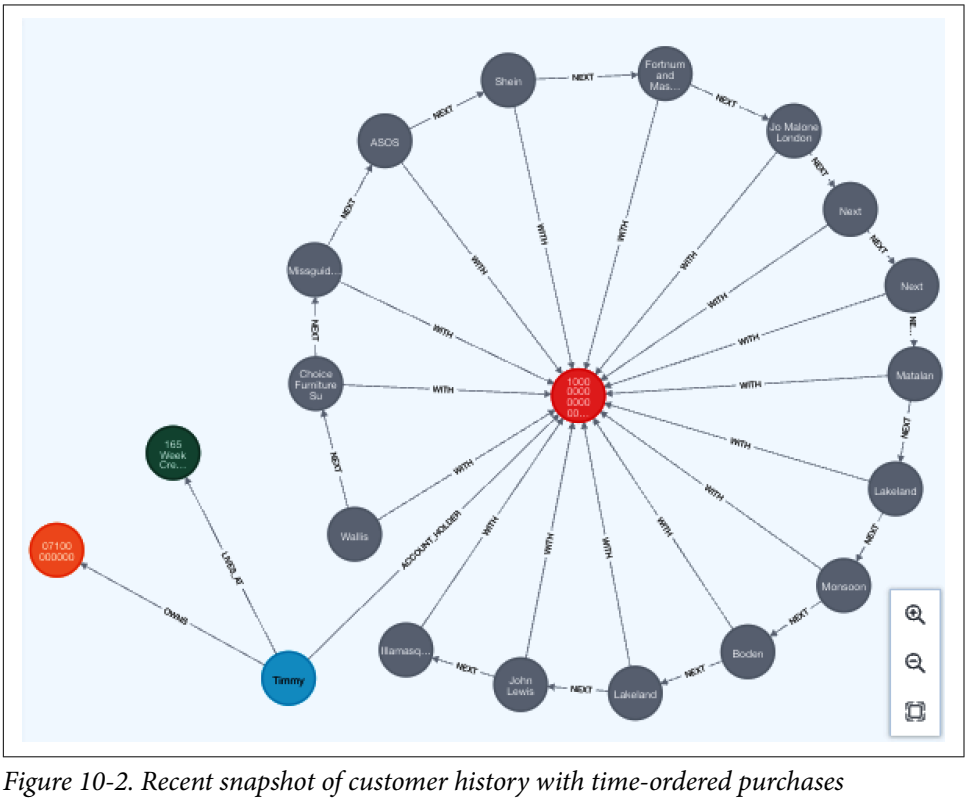


Figure 10-2. Recent snapshot of customer history with time-ordered purchases

Zooming in on one customer isn't enough to validate the knowledge graph, and it is time consuming to do this for all customers. If the data seems to be as uniform as Figure 10-2, you should be able to verify that uniformity with some simple tests.

One way of coming to grips with a data set that isn't entirely known is to verify some of its general qualities. An obvious starting point is to query for the number of people in the graph, as shown in Figure 10-3, to see if it matches the number of subgraphs.

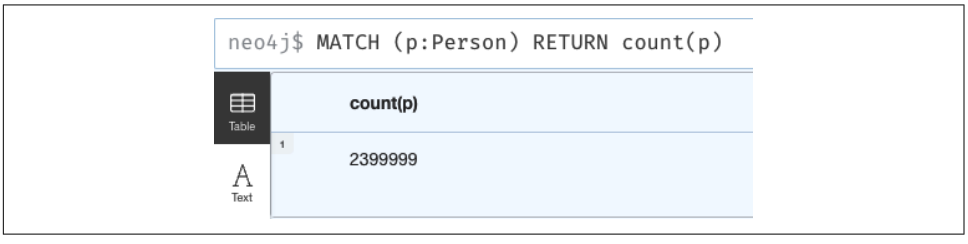


Figure 10-3. Count of people in the Customer-Purchases knowledge graph

In Figure 10-3 you can see that there are 2,399,999 `Person` nodes in the graph. If the model is regular, like in Figure 10-2, then there should be 2,399,999 communities in the graph, one for each detached subgraph. You can check that with a community detection algorithm from Neo4j Graph Data Science, which you first saw in Chapter 6. Start by creating a projection on which to run the community detection algorithm, as shown in Example 10-1.

Example 10-1. Create a projection of people and their means of contact

```
CALL gds.graph.project(
    'fraud-wcc',
    ['Person','Phone','Address'],
    ['LIVES_AT','OWNS']
)
```

The procedure call in Example 10-1 creates an in-memory projection of some elements of the knowledge graph. In this case it takes `Person`, `Phone`, and `Address` nodes along with `LIVES_AT` and `OWNS` relationships. It takes no property data or other nodes or relationships. The projection needs a very short time to prepare, and then you are ready to run one of the community detection algorithms, as shown in Example 10-2.

Example 10-2. Run the Louvain algorithm to compute communities in the graph

```
CALL gds.louvain.stream('fraud-wcc')
    YIELD nodeId, communityId
    WITH gds.util.asNode(nodeId) AS person, communityId
    SET person.communityId = communityId
```

Example 10-2 uses the Louvain algorithm to detect distinct communities in a knowledge graph. In this case, its job is simple because there should be 2,399,999 people in disjoint subgraphs, which should become 2,399,999 communities. For each community, the query takes its ID and writes it as a `communityId` property into each `Person` node in the community. But as you can see in Figure 10-4, things don't quite tally up. There are only 1,803,467 community identifiers in the knowledge graph, around half a million too few.

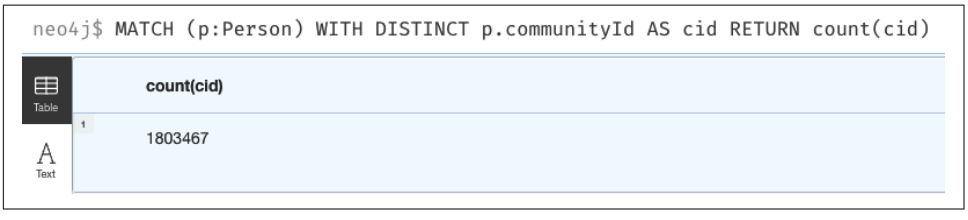


Figure 10-4. Fewer communities than people in the knowledge graph

What Figure 10-4 tells you is that some communities must contain more than one person. It would be interesting to know more about those communities since they may or may not be benign.

Fraud Rings

To understand what's happening, you need to sample the graph. You can easily craft a query to drill down into communities that have more than one `Person` node and inspect them, as shown in Example 10-3.

Example 10-3. Sampling query to check why `Person` nodes might be linked

```
MATCH (:Person) WITH count(*) AS count
MATCH (a:Person)-[*2..4]-(b:Person)
WHERE rand() < 10.0/count
RETURN a
```

Example 10-3 starts by matching all `Person` nodes in the graph and providing a count of all items. It then matches paths bewteen `Person` nodes at depth 2 to 4, so that we don't match on small, well-behaved subgraphs. The query then chooses to include, or not, the current match based on a random number so that you get a sample of the knowledge graph rather than the full version.

Running the query in Example 10-3 yields the results shown in Figure 10-5 in which there are two disjoint and dissimilar subgraphs. They're both quite different from the the common pattern of a single person owning a phone and a credit card, living alone at a single address.

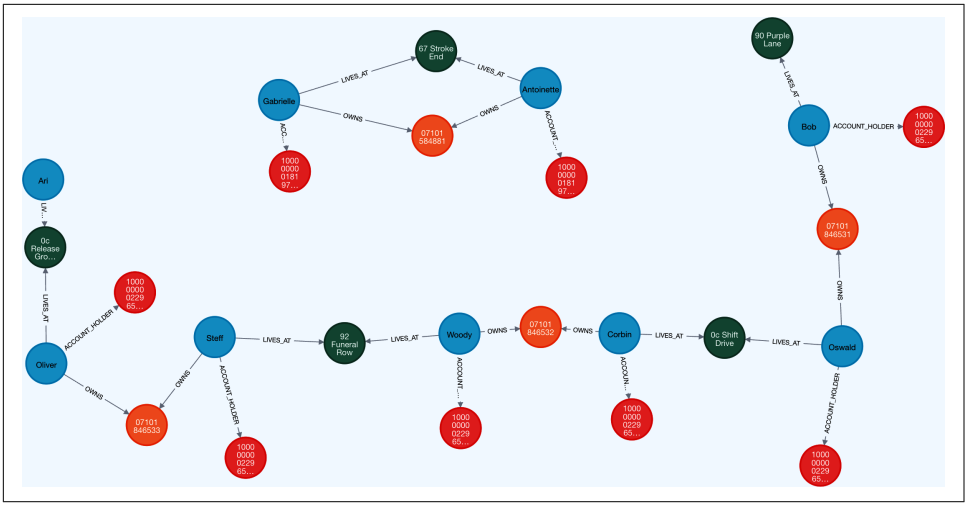


Figure 10-5. Reviewing a sample of the graph

But are the patterns in Figure 10-5 fraudulent? They are certainly different from each other and the expected form. Here you need subject matter expertise, at least to bootstrap that decision. For example, an analyst might observe a strong correlation between several short, consecutive Uber rides or a burst of expensive similar transactions, both suggesting a stolen or cloned card (or a phone on which the card is hosted).

In this case, the small subgraph is a household with a shared phone number and shared address. It's reasonably common and is unlikely to be fraudulent. The larger subgraph is unlikely to represent ordinary customers; it is probably a *fraud ring* that has created synthetic identities anchored with real addresses and phone numbers. Knowing these patterns (and indeed discovering others through human expertise and machine learning) allows you to check for them as transactions are processed in a payments system and reject those that match.

But you can go further than that. If you find a fraudulent pattern, then it's simple with a knowledge graph to traverse all of it to find all cards or accounts that might be compromised. To illustrate, take a step back and try to think like a fraudster.

Stealing cards must be stressful (though no sympathy for the fraudsters is necessary). The rightful owner tends to find out very soon that the card is missing or compromised and can act to cancel it quickly, perhaps even before any theft has taken place. As a criminal, it's far better to own a card (or other lines of credit like bank accounts with an overdraft) in good faith so that you can take your time planning your fraud and maximize the haul. Simply operate the account normally, and you will often be rewarded with higher lending limits. This vertically increases the gain from the eventual fraud. Better still, you can horizontally scale your crime by creating

compelling fake identities to gain access to many lines of credit. Then operate them normally until it's time to commit fraud at scale.

The fraud-ring pattern is hard to spot with conventional data technology, but with a knowledge graph it becomes much easier. The pattern repeats, from a `Person` via a shared `PhoneNumber` to another `Person` to a shared `Address` to another `Person` to a shared `PhoneNumber` and so on, encompassing an entire fraudulent network. You can see this pattern clearly isolated in Figure 10-6.

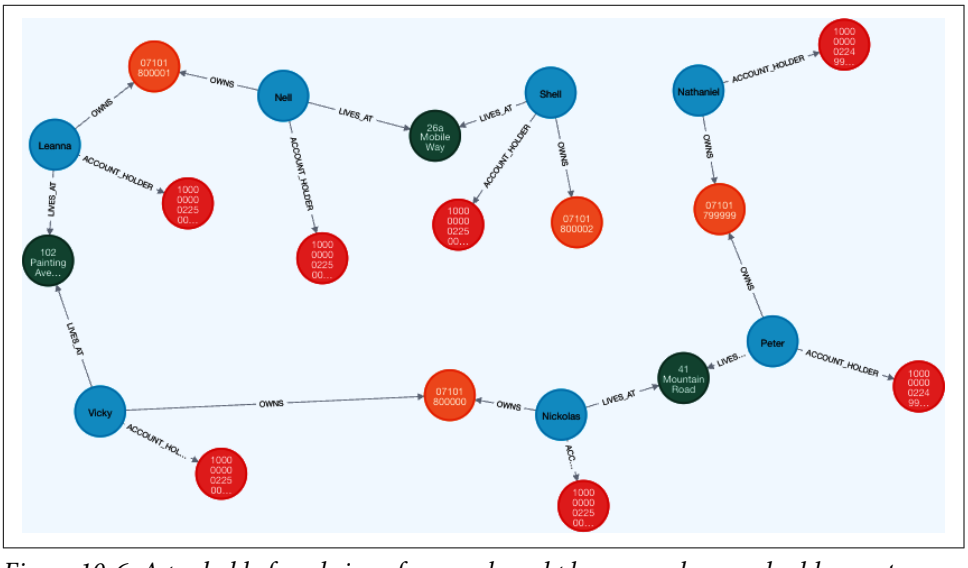


Figure 10-6. A probable fraud ring of many shared phone numbers and addresses to create synthetic identities

It now becomes clear that at least one of the subgraphs found in the sample shown in Figure 10-5 is likely to be a fraud ring. It's a pattern you can now use to discover potential criminality.

To find those chains or rings of potentially fraudulent identities and the cards (or accounts) that are affected, run the Cypher code in Example 10-4. The `MATCH` statement sets out the pattern of interest, being a `Person` node connected at depth 5 or greater to another `Person` node via `OWNS` or `LIVES_AT` relationships (for shared phones and addresses, respectively). From the `MATCH` clause, paths through the knowledge graphs are returned. To omit cycles in the graph, the `WHERE` predicate ensures that the path doesn't cross over itself. To find any rings (or rings within rings), simply omit that predicate. Finally, the query returns matched paths of potential fraudsters to the caller. Note that the `CALL ... IN TRANSACTIONS OF 100 ROWS` simply breaks down the query into smaller units for efficient execution and does not alter its semantics.

Example 10-4. Cypher code for finding fraud rings in a knowledge graph

```
CALL {
    MATCH ring=(p1:Person)-[:OWNS|LIVES_AT*5..]-(p2:Person)
    WHERE p1<>p2
    RETURN ring
} IN TRANSACTIONS OF 100 ROWS
RETURN ring
```

On a large knowledge graph, the code in Example 10-4 may take many seconds or minutes to execute. That's OK since it's an analytical query run periodically (e.g., hourly, daily) to highlight unusual customer patterns.

Allowing synthetic identities to grow permits fraudsters to build a substantial aggregate credit line. Once the fraudsters have built their criminal empire as big as they dare, they will cash out by rapidly using all the credit facilities at their disposal. Without a knowledge graph, the cash-out can appear to be random and unstoppable. Lines of credit are drawn down and left unpaid. When letters or debt collectors are sent to the synthetic identities, they are unanswered or, worse, affect genuine people who happen to live at those addresses. The money is lost.

With a knowledge graph, a cash-out in one part of a ring structure can be used very rapidly to find all other accounts in that structure and block or scrutinize transactions for those identities. This is a cheap operation since only a few hundreds or thousands of linked records need to be considered and can be performed in the transaction flow if necessary.

Innocent Bystanders

You know now that in the sample in Figure 10-5, the larger subgraph is likely to be a fraud ring. But the other subgraph seems more benign. Not all linked identities may be fraudsters, since people legitimately share addresses, and even in modern times, people may also share phone numbers (especially landlines at their home address or place of work). Branding a household as fraudulent usually isn't the right thing to do, and so in this use case you have to be aware of the pattern shown in Figure 10-7.

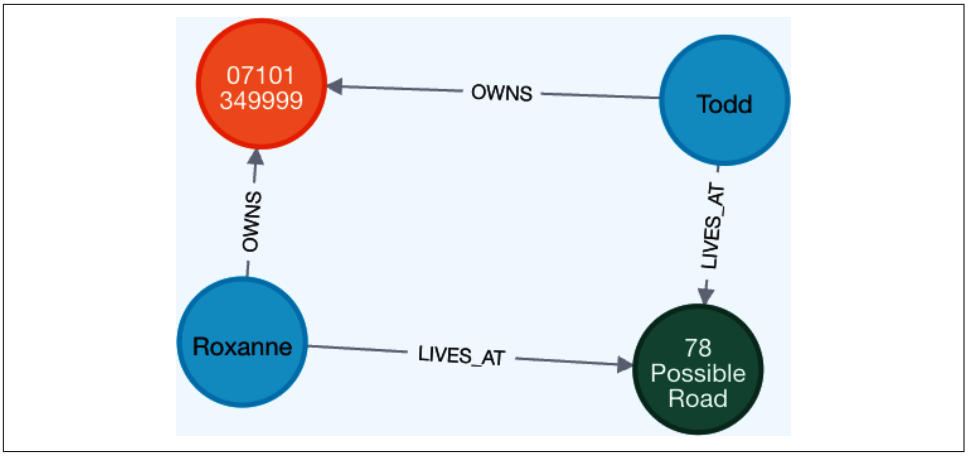


Figure 10-7. A household can share both a phone number and an address

Finding these patterns is a straightforward process—it's actually a simplified version of the fraud-ring query with a specific path length. In Example 10-5 the `MATCH` query pattern matches against two people who live at an address and share a phone. Note that the variable `p` is used to ensure that the same person `LIVES_AT` an address and `OWNS` the same phone as the other anonymous `Person`.

Example 10-5. Cypher code for finding legitimate households in a knowledge graph

```
MATCH households=(p:Person)-[:LIVES_AT]->(:Address)
                  <-[:LIVES_AT]-(:Person)-[:OWNS]->(:Phone)<-[:OWNS]-(p)
RETURN households
```

The Cypher query in Example 10-5 is simple, though the `MATCH` clause takes a little unpacking. It returns paths of all households that fit the pattern of two `Person` nodes connected by two `LIVE_AT` relationships to an `Address` node and then connected via an `OWNS` relationship to the same `Phone` node. The first person matched is stored as variable `p`, which is used to end the pattern to ensure it's the same person connected to the shared address and shared phone. Running this query for the current transaction in your payments system gives rapid confirmation that the holder is in a regular household. Another check is passed, and so confidence in the legitimacy of the transaction is increased.

Operationalizing the Fraud Detection Knowledge Graph

At this point, you understand some positive and negative patterns for fraud. In a production deployment, these patterns would be surfaced by subject matter experts, who can use their own knowledge and experience, supported by graph algorithms, graph machine learning, and visualization, to maintain a robust arsenal against fraud. Matching these patterns against a card's subgraph is the key to operationalizing this kind of knowledge graph.

In general, you will prefer subgraph-local operations since they are very low latency. Because the data is stored in a knowledge graph of cardholders, it's fast to access the subgraph centered on each customer during a transaction. It's therefore possible to search the customer subgraph for suspicious patterns as customers are making purchases, since a relatively small number of records need to be considered. Doing so adds minimal latency during a purchase and, if you have a good understanding of fraudulent patterns, offers good protection.

In cases of particularly high risk or value, you might decide to run more intensive pattern matching across a larger area of the graph. Or you might even choose to run some data science processes over a subgraph. Since these may add many milliseconds or even seconds to your decision making, they should be used relatively sparingly, with the low-latency pattern matching handling much of the work. Using both subgraph-local pattern matching with occasional graph algorithms can be an effective way to turn your knowledge about your customers into a way to protect yourself, and them, from fraud.

Case Study: Banking Circle

Banking Circle has a fraud management solution similar to that described in this chapter. At its core is a knowledge graph of accounts connected to payments. Banking Circle then uses community detection to generate features for its ML pipeline that detects high-risk clusters of customers. The knowledge graph topology is a critical input to the ML pipeline, combining graph features such as risk scores of near neighbors, distances to tax havens, and known fraudsters. It has had significant positive impact on Banking Circle's fraud detection efforts. It has turned a slow and manual process into a scalable, flexible solution that can evolve to meet the ever-changing challenges that fraudsters present. Today, Banking Circle continues to experiment with graph algorithms and plans to add more graph features to further tune its predictive model.

Skills Matching

An organization that can recruit and retain skilled workers has an edge in the marketplace. But skilled workers, in all fields of human endeavor, require active support so that their skills continue to align with the organization's ongoing needs. If an organization has the ability to recruit and retain skilled workers, the challenge of successfully deploying those workers into appropriate departments or projects remains. Not only do departments or projects require a specific mix of skills, but they also need different levels of seniority and experience. The best-run departments or projects will also have a diverse set of participants who (mostly) get along well.

Building robust teams is no trivial task, but knowledge graphs can help. There are many dimensions to building high-performing teams:

- Organizational chart, including cost centers and locations
- Expertise
- Project history
- Social engagement
- Tertiary skills (e.g., language)

Each of these can be modeled as a separate layer in a knowledge graph and brought together at query time to answer sophisticated questions. Those questions will help create and manage effective teams.

Organizational Knowledge Graph

The most obvious place to start to look for skills is the organizational (org) chart. As well as positional hierarchy, the org chart often identifies communities of practice or projects towards the bottom of the org chart where it fans out. Org charts also give some insight about tenure or level of competency based on vertical position.

Org charts are usually trees, from a single CEO through management to many workers. Since trees are graphs, it's quite easy to import an org chart as a fundamental layer in a knowledge graph. In Figure 10-8 you see a visualization from Neo4j Bloom showing part of an organizational hierarchy. The CEO *Alice* sits at the top of the tree with her senior vice presidents (SVPs) underneath, connected via `REPORTS_TO` relationships. The SVPs are connected to their VPs via incoming `REPORTS_TO` relationships, and the same pattern extends to directors and ultimately individual contributors (ICs).

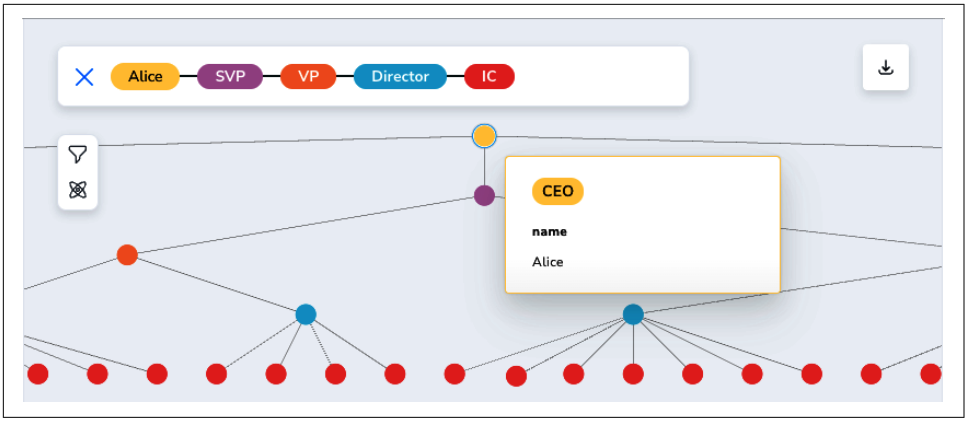


Figure 10-8. An organizational chart is a good basis for a skills knowledge graph

While Figure 10-8 doesn't distinguish the levels of seniority of ICs (for example, senior analyst or principal consultant), it's sufficient to enable you to reason about the *intended* structure of the organization. For example, you can easily run queries like that shown in Example 10-6 to understand how many employees are in a department or project, and from there be able to aggregate their costs.

Example 10-6. Finding people with Java project experience

```
MATCH (:VP {name:'Harry'})<-[:REPORTS_TO*1..2]-(n)
RETURN count(n) AS numberOfEmployees
```

When you do, you get a result like that shown in Figure 10-9.

Knowledge graphs are a great fit for HR teams and managers looking to understand or change an organizational hierarchy. It's easy to see how the knowledge graph in Figure 10-8 could be augmented to include a range of organizational and HR data for the employees.

But org charts are the *intended* structure of the organization, which is generally an approximation of its *real* structure, how it functions day to day. In most organizations there is an informal org chart which is rarely captured and even more rarely put into action. This is the org chart of how people actually work together, based on their skills and experience with delivering solutions.

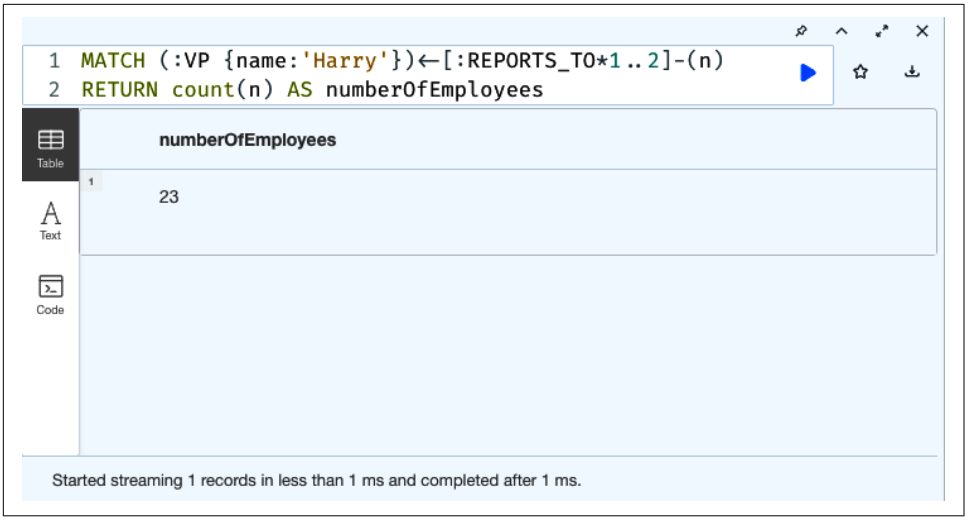


Figure 10-9. Aggregating the number of employees in a particular department

Skills Knowledge Graph

At a basic level, finding skills in a knowledge graph is simple. For example, the query `MATCH (ic:IC)-[:HAS_SKILL]->(s:Skill {name:'Java'}) RETURN ic.name` will return the names of the ICs who have had Java programming experience, like in Figure 10-10.

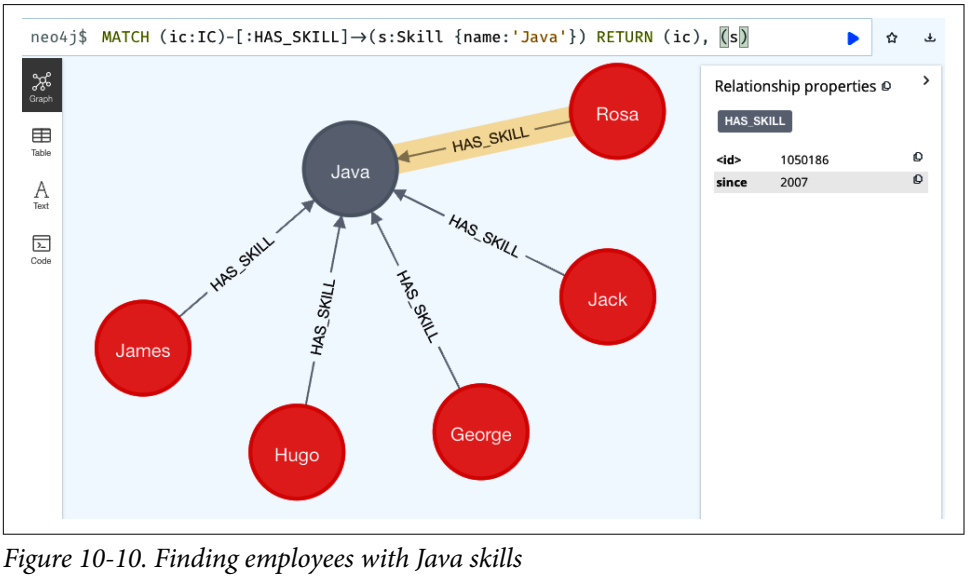


Figure 10-10. Finding employees with Java skills

But the results of that query are quite superficial. The results in Figure 10-10 say nothing about the level of proficiency, recency, or exposure of the individuals to real projects that use Java. Nor does it give any context for how those skills have been used over time. All you have is a name and a year when that person's experience began. That's probably not rich enough to make good staffing decisions, since you don't even know when that skill was last exercised.

A better way to understand skills is to inspect an individual's project experience, to aggregate the time spent honing particular skills and the time periods during which those skills were used. That connected context provides a richer view of an individual's competencies than any checklist could.

Of course, with a knowledge graph it's easy to add in those connections with minimal fuss. After some refactoring, a more effective knowledge graph emerges, as shown in Figure 10-11.

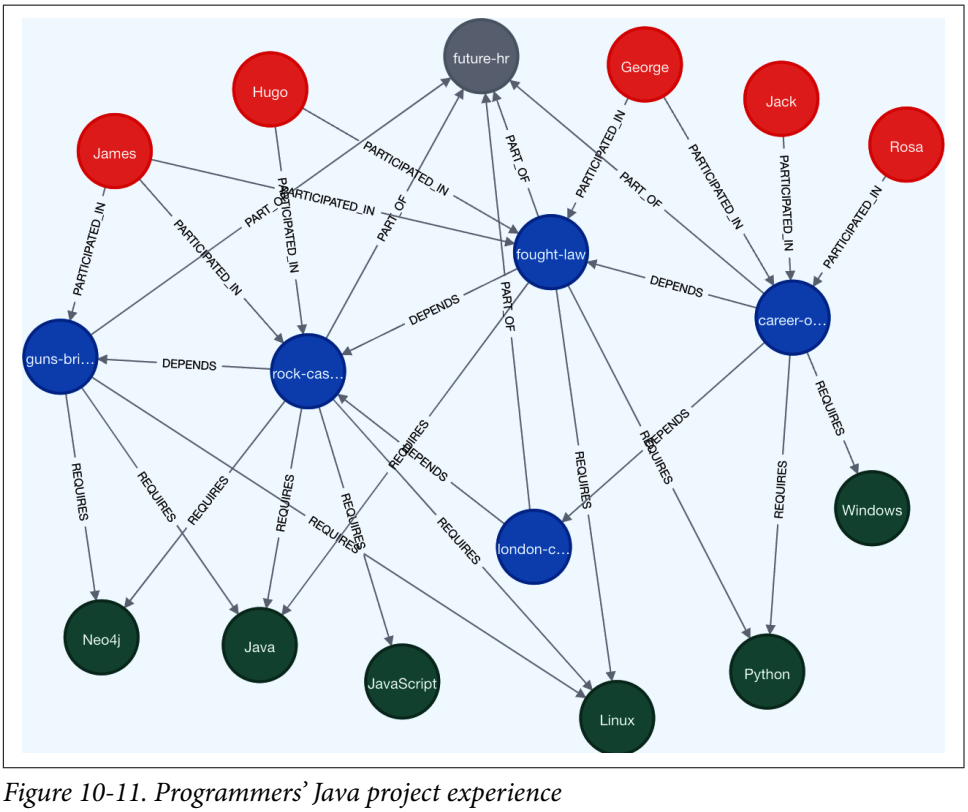


Figure 10-11. Programmers' Java project experience

Figure 10-11 is a much more realistic example of the kind of enterprise knowledge graph of projects, people, and expertise. In this case, the knowledge graph serves two purposes: a detailed repository of who worked on what and a way of finding

skills. In this subgraph, there is a program of work called `future-hr`, which has been broken down into several projects. The inaugural project is `guns-brixton`, which appears to have been a solo effort using `Java`, `Neo4j`, and `Linux` by engineer `James`. This is explicit in the graph structure where `(James)-[:PARTICIPATED_IN]-(guns-brixton)` and `(guns-brixton)-[:REQUIRES]->(Neo4j)`. The inaugural project in the program must have been `guns-brixton` because it has no outgoing project dependencies, only an incoming `DEPENDS` relationship from the next project `rock-casbah`. The project dependency and staffing structure continues to the last project, `career-opportunities`, which has no dependencies and is therefore the last project in the program.

Now you can use projects and even programs to understand which people have been involved with various skills. The aggregate time with a particular skill across projects gives a reasonable approximation for skill level, and the recency of using that skill can be used to reason about how up to date a particular skill is.

As an example, using Figure 10-11 as a guide, you can create a query to find people who have been on projects involving Java. This is shown in Example 10-7, where the query transitively matches from people to skills via projects.

Example 10-7. Finding people with Java project experience

```
MATCH (java:Skill {name:'Java'})<-[:REQUIRES]-(:Project)<-[:PARTICIPATED_IN]-(ic:IC)
RETURN DISTINCT ic.name
```

Example 10-7 is fine as a first approximation but lacks detail. A next useful step, shown in Example 10-8, is simply to count the number of Java projects that a person has participated in as a proxy for tenure.

Example 10-8. Finding people with Java project experience

```
MATCH (:Skill {name:'Java'})<-[:REQUIRES]-(:Project)<-[:PARTICIPATED]-(e:Employee)
CALL {
        WITH e
        MATCH (e)-[:PARTICIPATED]->(p:Project)
        RETURN collect(duration.inMonths(p.start, p.end).months) AS duration
}
RETURN DISTINCT e.name,
    reduce(total=0, number in duration | total + number) AS monthsOfExperience
```

The query in Example 10-8 builds on Example 10-7, adding aggregation so that the matching programmers are ranked by the number of Java projects in which they have participated. The `CALL` idiom is used to compose an aggregating subquery that computes the duration of individual employees' Java experience. When run in this

case, it returns that `James` and `Hugo` appear to be experienced Java programmers, with `George` having a little less experience, as shown in Figure 10-12.

```
MATCH (:Skill {name:'Java'})<-[:REQUIRES]-(:Project)<-[:PARTICIPATED_IN]-(ic:IC)
CALL {
    WITH ic
    MATCH (ic)-[:PARTICIPATED_IN]->(p:Project)
    RETURN collect(p.duration) AS duration
} RETURN DISTINCT ic.name, reduce(total=0, number in duration | total + number) AS
monthsOfExperience
```

	ic.name	monthsOfExperience
1	"James"	40
2	"Hugo"	36
3	"George"	18

Started streaming 3 records after 20 ms and completed after 20 ms.

Figure 10-12. A program history

From here, it's not hard to see how you could enrich this knowledge graph to fit other valuable production uses. It already serves to surface latent skills knowledge, but with the addition of the workers' personal profiles, project evaluations, and so forth, you can create a knowledge graph system that can help staff the best people on the most challenging or critical projects.

Expertise Knowledge Graph

So far, you've seen how knowledge graphs can be populated with data from people and project data to discover skills explicitly associated with patterns of work. This mirrors a traditional way of organization, where people from various parts of the org chart work on projects and get increasing seniority through tenure. Such metrics are a valuable signal, but they are not the only valuable signal.

Particularly in larger organizations, staff don't only orient themselves around the org chart or project team. There are often long-lived communities of interest in particular tools or skills. Finding the experts in these communities can help to provide a complementary signal to your org chart and project statistics; in fact, it might even be considered the *real* organizational chart for your business.

For information workers, at least, the modern enterprise can be remarkably flat. While they still have to adhere to some protocols and be somewhat mindful of the org chart hierarchy, peer collaboration is often strongly encouraged and facilitated by enterprise collaboration systems. Systems like Slack, Microsoft Teams, and so forth allow employees not only to communicate in a point-to-point fashion but also to

form communities around long-lived topics. Within those communities are people with deep expertise and experience looking to help others and those who are seeking to participate to upgrade their skills or find help.

If the communities and their experts can be discovered, they can be amplified to the benefit of all concerned. However, platforms like Slack and Microsoft Teams don't offer structured data beyond users, conversations, and threading. Finding good information or experts is difficult and often relies on word of mouth. As such, experience is varied, with a distinct bias against new employees who haven't yet developed their work network.

Enriching your knowledge graph by adding a layer from the data in your collaboration platform can help. After all, wouldn't it be nice if your skills record was automatically updated with expertise scores based on the quality of your interactions on Slack or Teams?

The kind of model that enterprise collaboration platforms have is very amenable to knowledge graphs. Moreover, some of the work that you might do with graph data science, such as community detection, is implicit in the channels that employees join. A knowledge graph representation of Slack's core idioms is shown in Figure 10-13. It allows for patterns like `(:User)-[:WROTE]->(:Message)-[:POSTED_TO]->(:Channel)` and `(:Message)-[:MENTIONS]->(:Skill)`, which transitively allow you to understand those users who are writing or responding to messages about a particular skill. It also contains the platform structural data around channels and topics to allow generalization from simply skills to channels containing a community of people who possess (or at least discuss) those skills.

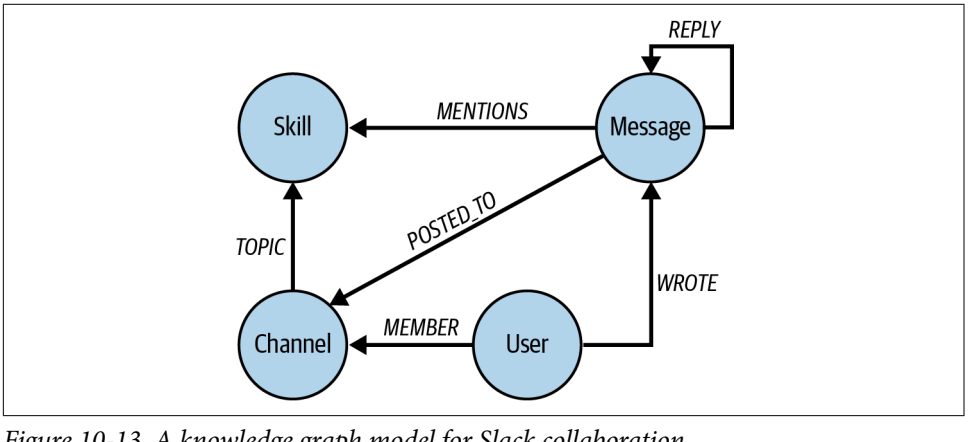


Figure 10-13. A knowledge graph model for Slack collaboration

Figure 10-13 isn't difficult to understand; it's probably very similar to how you'd draw the domain on a whiteboard. But despite its simplicity, it does not lack capability. For

any given skill, you can trivially determine who engages most across the organization, as shown in Example 10-9.

Example 10-9. Finding the most engaged people for Intermediate Language (IL) techniques

```
MATCH (m:Message)-[:POSTED]->(c:Channel)-[:TOPIC]->
        (s:Skill {name:'IL'})<-[:MENTIONS]-(m)
MATCH (m)<-[:REPLY*]-(r:Message)<-[:POSTED]-(u:User)
WITH u AS user, count(r) AS replies, s AS skill,
    collect(DISTINCT c.name) AS channel ORDER BY replies DESC
RETURN skill.name, user.name, replies, size(channel) AS numberOfChannels
```

By now, the Cypher code in Example 10-9 should feel familiar. It first finds original postings on a particular `Skill`, which can appear in any channel. From there, it matches against a path of `Message` nodes joined by `REPLY` relationships to get conversation threads. Then the query aggregates the results. To do so, it uses the number of replies to order the results in a descending fashion. It collects the channels in which each thread appears into a list using `DISTINCT` to ensure no duplicates. Finally, it returns the skill, the users who have participated in threads about that skill along with how many times they have replied, and the number of channels across which they have replied. If you run the query, you get a result like that shown in Figure 10-14.

	skill.name	user.name	replies	numberOfChannels
1	"IL"	"James"	7	2
2	"IL"	"Hugo"	2	1
3	"IL"	"Jack"	2	1
4	"IL"	"George"	1	1
5	"IL"	"Rosa"	1	1

Figure 10-14. Users who have significantly participated in discussions on a specific topic across one or more channels

It is not difficult to see how this generalizes so that you can identify experts in your organization. In a way, the expertise graph is the complement to your official organizational chart. As Andy Grove states in *High Output Management*, 2nd edition

(Vintage Books), employees have *organizational power* because of the organizational chart or *knowledge power* because of their skills. Sometimes they have both, in the case of skilled managers. A knowledge graph is an excellent way of merging these two axes for a significantly richer understanding of your enterprise's staffing.

Individual Career Growth

A knowledge graph like the one in Figure 10-13 is hugely useful for individuals. For any given topic, the system can easily recommend a likely expert who can advise to reduce project risk and time and transfer knowledge to help upskill others.

But there are other benefits of a skills knowledge graph for the individual. The foundational knowledge contained in such a system is the employee's project history. As an individual, it becomes simple to look back at your project and skills history, which gives you a feel for your general career direction. This is shown in Example 10-10.

Example 10-10. Project history for individuals

```
MATCH (:Employee {name:'Rosa'})-[part:PARTICIPATED]->
       (p:Project)-[r:REQUIRES]->(s:Skill)
WITH p AS proj, s AS skill, part.end AS lastUsed,
       duration.inDays(part.start, part.end).days AS days,
RETURN skill.name AS skill, sum(days) AS daysOfExperience,
       max(lastUsed) ORDER BY daysOfExperience DESC
```

The `MATCH` clause in Example 10-10 finds a specific employee and the projects in which they have participated. From there, it finds the skills deployed on that project. It then calculates the duration in days that the employee spent on each project in the `WITH` clause. The `RETURN` clause aggregates that data into a short report. It returns the skill, how many days' experience in total the employee has with that skill, and the last date when it was used (on the basis that recently used skills are fresher). The results are presented in descending order by days of experience in the skill. An example of the output from running the query in Example 10-10 is shown in Figure 10-15.

	skill	daysOfExperience	lastUsed
1	"Neo4j"	1306	"2022-10-03"
2	"Linux"	1244	"2022-10-03"
3	"Python"	791	"2021-07-06"
4	"JavaScript"	789	"2022-10-03"
5	"Java"	515	"2022-10-03"
6	"Windows"	62	"2020-12-12"

Figure 10-15. Quantifying an individual's project history and skills exposure

Clearly, the query in Example 10-10 provides useful results in itself, but it also forms the bedrock of more sophisticated analyses. One of the most compelling analyses for employees is to recommend skills for career paths. Career recommendations draw heavily on being able to reason about the individual's project history and accumulated skills, but within the context of others whose skills and career progression can act as guides. This is a pattern that is readily exploitable, as shown in Example 10-11.

Example 10-11. Project history for individuals

```
MATCH (me:Employee {name:'Rosa'})-[:PARTICIPATED]->(:Project)
        <-[:PARTICIPATED]-(other:Employee)
MATCH (other)-[:PARTICIPATED]->(:Project)-[:REQUIRES]->(s:Skill)
WHERE NOT (me)-[:PARTICIPATED]->(:Project)-[:REQUIRES]->(s)
WITH s AS skill, count(s) AS popularity
RETURN DISTINCT skill.name, popularity ORDER BY popularity DESC
```

The key aspect of the Cypher query in Example 10-11 is to find people with whom you have a shared project history who have gone on to other projects where they have been exposed to new skills that you have not. Those skills to which you have not been exposed are probably good things to learn.

The first `MATCH` clause simply surfaces projects in which Rosa and other employees have participated. The second `MATCH` finds projects that others have worked on and the skills they have encountered during those projects. It is narrowed by the `WHERE` clause, which includes only those projects that Rosa hasn't worked on. This leaves only the skills possessed by people Rosa has worked with but that Rosa herself does not have. Finally, the query ends by returning those skills ordered by their popularity in Rosa's peer group.

You can see clearly in Figure 10-16 that the primary skill Rosa should consider learning is Java if she wants to mimic most of her project peers. This is followed by a list of other technologies her peer group has been exposed to, in descending order of popularity.

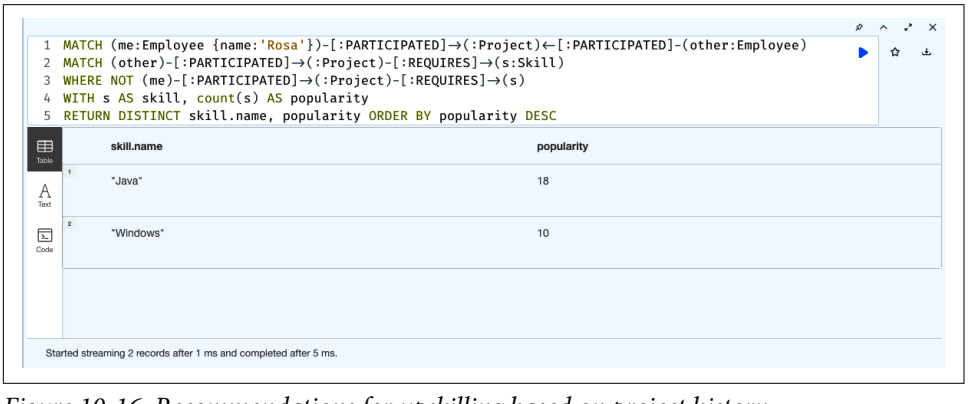


Figure 10-16. Recommendations for upskilling based on project history

The examples you've seen provide immediate value for solving important HR problems. But they also form the basis for more sophisticated analyses.

For example, it's straightforward to add a simple taxonomy of skills so that staffing can be reasoned about at a generalized and specialized level. This means you can think about the general skill of databases versus the specific skill of NoSQL, or the general skill of functional programming versus the specific skill of F#.

Organizational Planning

Recall the individual employee project history query from Example 10-10. As it happens, with minor tweaks this is precisely the kind of query that managers in an organization use when looking across the employee base for particular skills and experience. Whereas Example 10-10 curated the results around an individual employee by using their `name` property in the `(:Employee)` node, the manager's query

is more general. Managers are looking for specific expertise, so their equivalent query simply is centered on skills rather than employees, as shown in Example 10-12.

Example 10-12. Project history for individuals

```
MATCH (e:Employee)-[part:PARTICIPATED]->(p:Project)-[r:REQUIRES]->
        (s:Skill {name:'Java'})
WITH p AS proj, e AS employee, part.end AS lastUsed,
        duration.inDays(part.start, part.end).days AS days
RETURN employee.name AS skill, sum(days) AS daysOfExperience,
        max(lastUsed) ORDER BY daysOfExperience DESC
```

If you run the query in Example 10-12, you will get results similar to those in Figure 10-17.

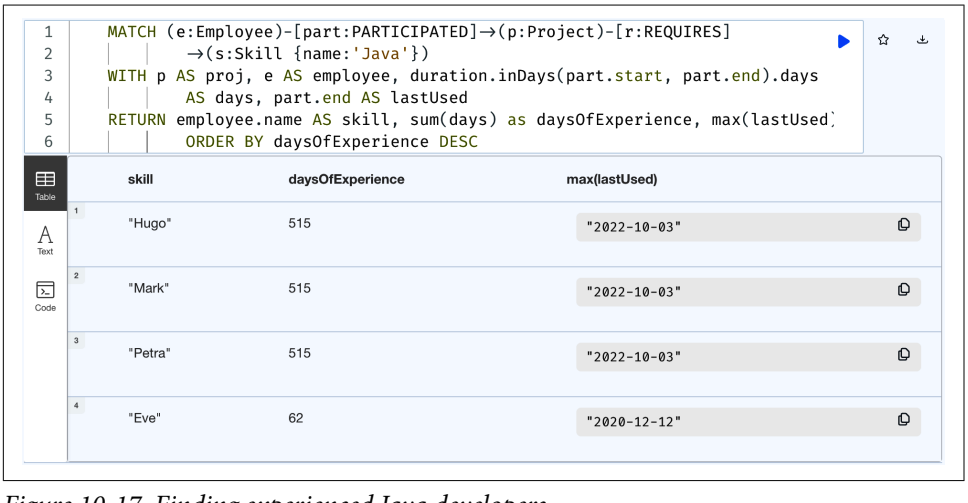


Figure 10-17. Finding experienced Java developers

You can add other dimensions into the staffing mix, not just skills, to improve the outcome of projects. For example, it is well understood that team members who enjoy working with each other tend to produce better results. Adding a social layer to the people-skills knowledge graph allows the organization not only to assemble the necessary skills for a project but also to create a cohesive team ready to embrace those challenges together.

For example, if members of a project were allowed to rate their experience working with one another, then you have a ready source of information from which to create high-impact teams. This is a rich social network based not only on "follows" or "likes" but on a project-by-project (or quarter-by-quarter) score of who gels with whom. A subgraph that gives an example of the topology is shown in Figure 10-18.

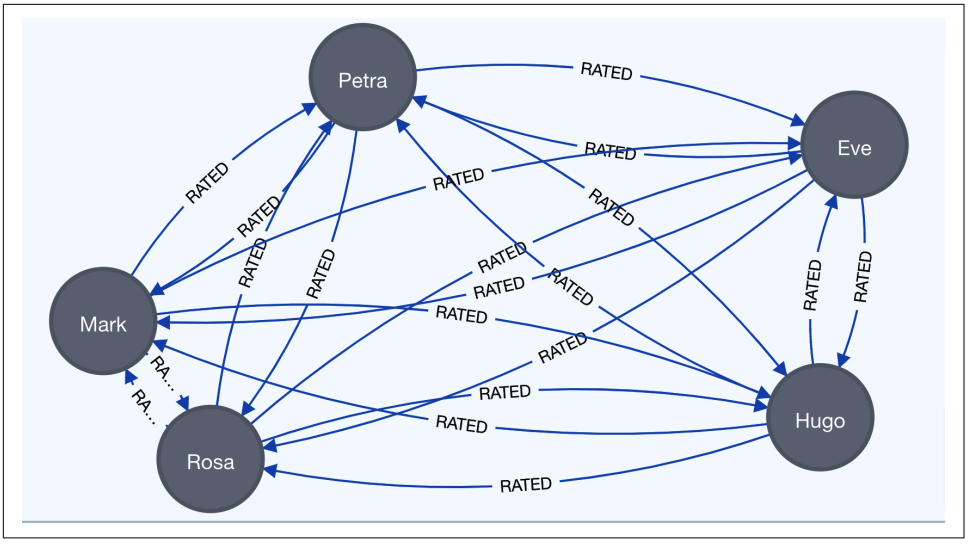


Figure 10-18. Developers ranking each others' qualities on past projects

For pedagogical reasons, the peer rankings in Figure 10-18 are captured as a list of scores on the `RATED` relationships. The average of these scores gives some insight into the working preferences of team members. Harnessing that data to provide peer recommendations is straightforward, as shown in Example 10-13.

Example 10-13. Project history for individuals

```
UNWIND ["Rosa", "Hugo", "Eve", "Petra", "Mark"] AS candidate
MATCH (me:Employee {name:candidate})
MATCH (me)-[r1:RATED]->(other:Employee)
MATCH (me)<-[r2:RATED]-(other:Employee)
WITH me.name AS myself,
    reduce (score = 0, r in r1.ratings | score + r)/size(r1.ratings) AS s1,
    reduce (score = 0, r in r2.ratings | score + r)/size(r2.ratings) AS s2,
    other.name AS them
RETURN myself, them, (s1+s2)/2 AS score ORDER BY myself, score DESC
```

The Cypher query in Example 10-13 uses the `UNIWND` clause to turn a list into individual rows for the query. It is a convenient way of specifying parameters, which in this case are people who have been involved in peer rating. The `MATCH` clauses build a pattern where mutual ratings have been made between the current employee and another employee. The `WITH` clause computes the average score for the incoming and outgoing ratings using `reduce`. Finally, the `RETURN` clause takes the scores for incoming and outgoing ratings and divides them by two for an aggregate compatibility score. The result of running the query is shown in Figure 10-19.

```
UNWIND ["Rosa", "Hugo", "Eve", "Petra", "Mark"] AS candidate
MATCH (me:Employee {name:candidate})
MATCH (me)-[r1:RATED]→(other:Employee)
MATCH (me)←[r2:RATED]-(other:Employee)
WITH me.name AS myself,
    reduce (score = 0, r in r1.ratings | score + r)/size(r1.ratings) AS s1,
    reduce (score = 0, r in r2.ratings | score + r)/size(r2.ratings) AS s2,
    other.name AS them
RETURN myself, them, (s1+s2)/2 AS score ORDER BY myself, score DESC
```

Table | Text | Code

	myself	them	score
1	"Eve"	"Rosa"	7
2	"Eve"	"Hugo"	6
3	"Eve"	"Petra"	1
4	"Eve"	"Mark"	1
5	"Hugo"	"Rosa"	8
6	"Hugo"	"Eve"	6
7			

Figure 10-19. Analyzing the developers skills ranking network

The query shown in Figure 10-19 provides an exhaustive list of mutual rankings. Sometimes it's more useful to see the communities of people that have ranked one another highly. A slight tweak to Figure 10-19 is shown in Example 10-14, where the cliques in the overall population are discovered.

Example 10-14. Productive cliques

```
UNWIND ["Rosa", "Hugo", "Eve", "Petra", "Mark"] AS candidate
MATCH (me:Employee {name:candidate})
MATCH (me)-[r1:RATED]->(other:Employee)
MATCH (me)<-[r2:RATED]-(other:Employee)
WITH me AS myself,
    reduce (score = 0, r in r1.ratings | score + r)/size(r1.ratings) +
    reduce (score = 0, r in r2.ratings | score + r)/size(r2.ratings)/2 AS score,
    other AS them, r1, r2
WHERE score >= 5
RETURN myself, them, score, r1, r2
```

The main change in Example 10-14 is that only high mutal ratings are returned using `WHERE score >= 5`, and the `RATING` relationships are also returned. This allows the

graph to be rendered visually, as shown in Figure 10-20, which can be a much more intuitive view of the data.

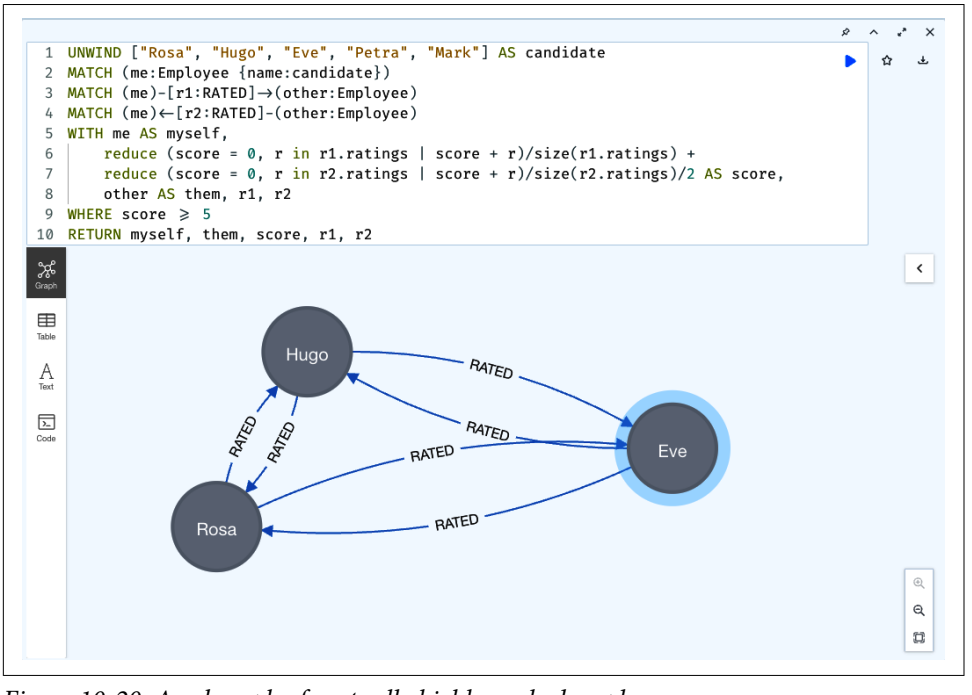


Figure 10-20. A subgraph of mutually highly ranked employees

The scoring mathematics in this example is on the simple side, but for your organization, you can make that scoring as sophisticated as you need. Whatever scoring arithmetic you choose, the notion of using the social graph to fine-tune team composition is a solid foundation.

Predicting Organizational Performance

The benefits of a skills knowledge graph rooted in the organization's people and projects is clearly very useful. But knowledge graphs can go far beyond workforce development. They can also be used to evaluate and reason about the likely outcomes of future projects based on past outcomes. For example, Figure 10-21 shows the subgraph containing workers and their managers around two incomplete projects.

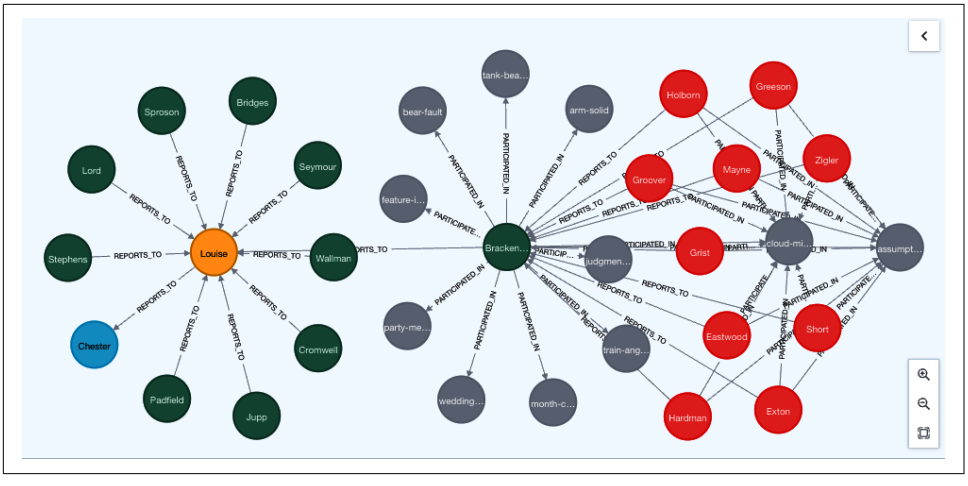


Figure 10-21. Incomplete projects in the organization

In Figure 10-21, the staffing on the projects is interesting. In fact, you can use it as the basis for reasoning—in broad terms—about the likely outcomes for the projects underway. For instance, poor managers or low-skilled workers could have a negative impact on the project, whereas a strong management structure and skilled workers might tend to have a positive effect. To get a sense of the history of failed projects and their chains of management, you can run a query like the one shown in Example 10-15.

Example 10-15. Looking for the leadership of failed projects

```
MATCH (p:Project)<-[:PARTICIPATED_IN]-(:IC)-[:REPORTS_TO*3..3]->(svp:SVP)
WHERE p.rating < 30.0
RETURN svp, count(p) AS failedProjects ORDER BY failedProjects DESC
```

The Cypher query in Example 10-15 binds to nodes with `Project` labels that satisfy the `WHERE` predicate that there is a property key `rating` (for completed projects only) and a value less than `30.0`, which is considered a failed score. From there, it matches individual contributors who `PARTICIPATED` in the project and the leader three levels of management above them at SVP level via `REPORTS_TO*3..3`. It suggests that the project teams under SVP `Lynn Smyth` are much more often troubled than their contemporaries, as shown in Figure 10-22.

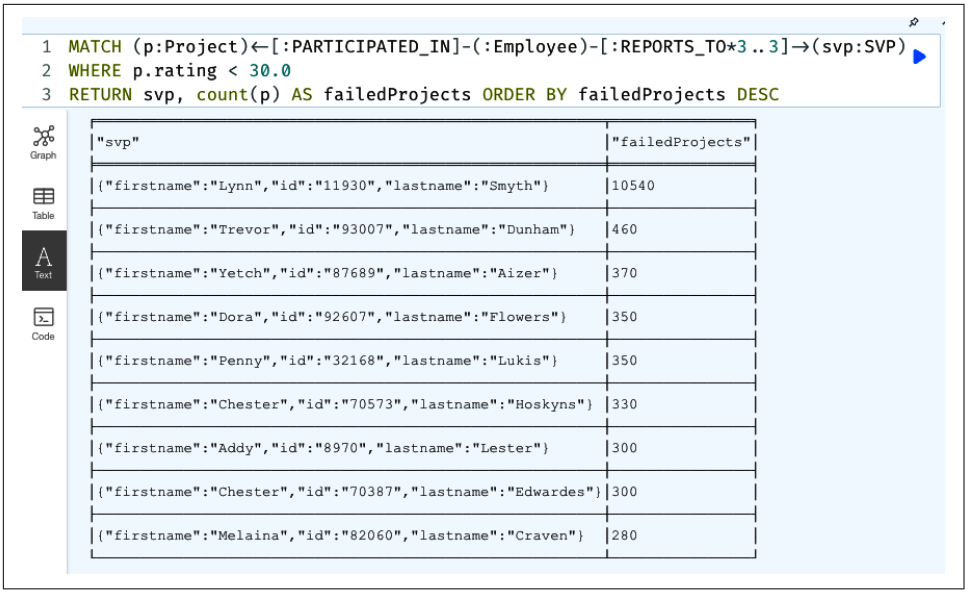


Figure 10-22. Failed projects that roll up to `Lynn Smyth`

Of course, at such a small scale the data can be made to tell almost any story. If you go deeper into Lynn's history, there might be compelling reasons why project failure is more prominent. For example, she might run a research organization where even failed projects have value. Lynn is not the only SVP with some part of their portfolio having gone awry, but hers is somewhat more significant compared to the others.

Exhaustively pattern matching like this at a global scale is impractical and limiting. It would be much nicer if the knowledge graph could simply make good recommendations about project leaders that might be successful or give early warning about projects that might go astray. Graph data science gives you the tools to tackle this problem.

As you recall from Chapter 6, the first step is to create a projection from the knowledge graph that contains the nodes, relationships, and properties that you want to analyze. In this case, you want the entire knowledge graph structure of `Employee`, `Project`, and `Incomplete` nodes and `PARTICIPATED_IN` and `REPORTS_TO` relationships but only the `rating` property of `Project` nodes, as shown in Example 10-16. Ultimately, you want to be able to predict a `rating` property for projects that haven't yet completed to predict likely outcomes.

Example 10-16. Creating a graph projection for historical and current projects

```
CALL gds.graph.project(
    'projects',
```

```
    {
        Incomplete: {},
        Project: { properties: ['rating'] },
        Employee: {}
    },
    {
        PARTICIPATED_IN: {orientation: 'UNDIRECTED'},
        REPORTS_TO: {orientation: 'UNDIRECTED'}
    }
)
```

The next step is to create an ML pipeline with `CALL gds.alpha.pipeline.nodeRegression.create('projects-pipeline')`. Now you can add a node regression procedure to the `projects-pipeline`, as shown in Example 10-17. In this case, the `fastRP` algorithm is chosen because it encodes knowledge graph topologies and data as numeric values suitable for ML. Here it encodes the topology of `Project` nodes as a numerical property on each of those nodes in the projection.

Example 10-17. Adding a node regression procedure into the ML pipeline

```
CALL gds.alpha.pipeline.nodeRegression.addNodeProperty(
    'projects-pipeline',
    'fastRP', {
        embeddingDimension: 256,
        iterationWeights: [0, 1],
        mutateProperty:'fastrp-embedding',
        contextNodeLabels: ['Project']
    }
)
```

Next, you choose the features on which you'll train the model. Since organizational topology is important in this case, the `fastrp-embedding` from Example 10-17 property is chosen, since FastRP encodes nodes' topologies into a number suitable for use as a feature in ML. An example is shown in Example 10-18.

Example 10-18. Choosing features on which to train the model

```
CALL gds.alpha.pipeline.nodeRegression.selectFeatures(
    'projects-pipeline',
    'fastrp-embedding'
)
```

The next step is to declare the split between training and testing data. In Example 10-19 `testFraction` is set to `0.2`, meaning that 20% of the graph is reserved for testing the model (with the remainder being used for other purposes like training).

Example 10-19. Declare the ratio between training and testing data

```
CALL gds.alpha.pipeline.nodeRegression.configureSplit('projects-pipeline', {
    testFraction: 0.2
})
```

Next, you need to add a regression model to your pipeline. In this case, random forest is used. Random forest is a popular supervised ML method for classification and regression that consists of using several decision trees (to prevent overfitting on a single tree) and combining the trees' individual predictions into an overall prediction. The configuration is shown in Example 10-20.

Example 10-20. Adding a regression model into the pipeline

```
CALL gds.alpha.pipeline.nodeRegression.addRandomForest('projects-pipeline', {
    numberOfDecisionTrees: 10
})
```

Now that the regression model and data are in place, the system is ready to tune. Since Neo4j Graph Data Science supports auto-tuning, it's as simple as declaring the number of iterations the auto-ML system should execute before picking the best parameters. In this case, it's computationally cheap enough to try `100` iterations, as shown in Example 10-21.

Example 10-21. Add configuration for regression parameter auto-tuning

```
CALL gds.alpha.pipeline.nodeRegression.configureAutoTuning('projects-pipeline', {
    maxTrials: 100
})
```

The penultimate step is to train a predictive model using all of the configuration specified previously. This is shown in Example 10-22. The model is trained on nodes with the `Project` label and the `rating` property in preparation for predicting the rating property of incomplete projects.

Example 10-22. Train the predictive model

```
CALL gds.alpha.pipeline.nodeRegression.train('projects', {
    pipeline: 'projects-pipeline',
    targetNodeLabels: ['Project'],
    modelName: 'projects-pipeline-model',
    targetProperty: 'rating',
    randomSeed: 1,
    concurrency: 4,
    metrics: ['MEAN_SQUARED_ERROR']
})
```

Finally, you have a trained model that can be used to make predictions about the health of incomplete projects. To use it, run the the example code in Example 10-23, where the `projects-pipeline-model` is used to make predictions on nodes labeled `Incomplete` whose results are then streamed back to the user.

Example 10-23. Make predictions about project outcomes

```
CALL gds.alpha.pipeline.nodeRegression.predict.stream('projects', {
    modelName: 'projects-pipeline-model',
    targetNodeLabels: ['Incomplete']
}) YIELD nodeId, predictedValue
WITH gds.util.asNode(nodeId) AS projectNode, predictedValue AS predictedRating
RETURN projectNode.name AS name, predictedRating
    ORDER BY predictedRating ASC LIMIT 10
```

The results from an execution of Example 10-23 are shown in Figure 10-23. In it, you can see the 10 projects with the poorest predicted scores. From here, you can continue to perform further data science on the knowledge graph to better understand why specific projects might fail, or you might take action at a business level to confirm the findings. Either way, at least you now have guidance on which projects could benefit from early intervention.

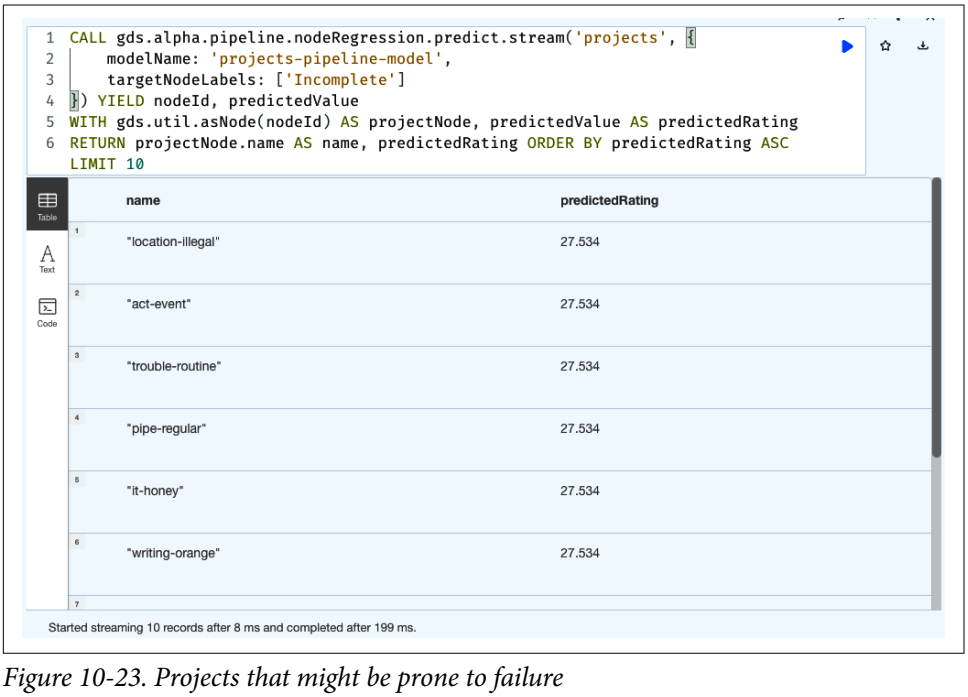


Figure 10-23. Projects that might be prone to failure

This approach has two key benefits. For the organization looking to improve, it helps to track outcomes by department/leader and skill set and to predict future outcomes based on a notion of past competence held in the knowledge graph. It can help leaders in difficulty to improve or help to identify systemic barriers that lead to prolonged poor performance. This approach can also be a starting point for organizational redesign if the knowledge graph reveals persistent structural or individual failings.

For individuals looking to upskill, this approach can help navigate to areas where project delivery is mature, so teams have the breathing space to help new members learn. For those individuals looking to show their mettle, it can point to opportunities across their organization where the skill set matches but where the opportunities to join a strong team or strengthen a failing team are explicit. Whichever of the paths is taken, the individual employee is given significantly better visibility into what will be expected of them.

What is very clear at this point is that the skills knowledge graph takes you from a narrow world of "five years Java and Agile practices" to a world where individuals have a rich history of skills in context of delivery. An individual's past and future career skills all enrich the knowledge graph, helping those individuals get the best from their careers as well as helping the organization to plan for the future.

Case Study: DXC Technology

DXC Technology is a global IT services company employing 130,000 people. To thrive in a competitive marketplace, DXC has to attract, retain, and upskill its team so that it can provide the utmost level of consultancy service and delivery expertise to clients.

DXC wanted to create a system where employees can easily get recommended career-development advice, based on the roles and career pathways that would be most valuable to the organization. The system should make it simple for employees to upskill and cross-skill based on available roles within the company.

However, like many other large enterprises, DXC has information silos and disconnected systems. While those systems are all valuable in their own right, they weren't able to provide a connected view to employees. This led to challenges around hiring and promotions as well as in creating career pathways to support the growth of employees and the business. In times of high demand, employees were lost to attrition, and in times of workforce reductions, suboptimal decisions were made.

To remedy these problems, creating better opportunities for DXC employees and better outcomes for customers, DXC developed the Career Navigator. The DXC Career Navigator connects all the siloed data and generates intelligent recommendations, underpinned by an employee knowledge graph similar to the examples in this chapter.

Now all an employee's interactions with the knowledge graph are mediated by an application that starts by making recommendations based on skill progressions of similar employees. For example, the tool recommends skills for a software engineer based on the the gaps between their skill sets and those of similar software engineers.

DXC's Career Navigator uses a combination of Neo4j's graph database and Graph Data Science library. The knowledge graph hosted on Neo4j contains all the structural information about the employees and their skills. The graph algorithms executed by Graph Data Science enrich the knowledge graph by identifying similarly skilled employees, which can be used as a prototype for others' career progression.

DXC's Career Navigator has been so successful that the company is now offering it as a solution to its customers.

Summary

Uncovering and understanding patterns in your domain is a key competitive differentiator. Using pattern detection knowledge graphs is a powerful way of unlocking that skill.

The examples you have seen in this chapter provide a firm foundation upon which you can build your own pattern detection knowledge graphs. You can freely mix human expertise with insight from graph data science practices to enrich your knowledge graph. In turn, that enriched knowledge graph can be put into production to serve online and analytical use cases with equal competence.

But the techniques you've learned in this chapter have broader applicability. They are not only for social networks like fraudsters or business networks of colleagues and projects. In the following chapter, you'll learn how to process patterns of dependencies within systems for such varied domains as supply chain and risk management.

CHAPTER 11

Dependency Knowledge Graphs

What do the following three scenarios have in common?

- You are trying to come up with the best plan for your project, and you need to understand the dependencies between the different tasks involved. "Task B cannot start before A is completed" means B depends on A. "Task B and task C are both carried out by team member X" means that B and C both depend on X.
- A vulnerability has been reported on a popular software library. To determine whether your projects are directly affected by the vulnerability, you will have to verify that your code doesn't directly use that vulnerable library. You also have to check that your chosen libraries don't use the compromised library themselves.
- A company owns 25% of the shares in another company, which itself owns part of another and so on. Finding the ultimate beneficial owner (UBO) of a transaction carried out by an institution requires you to traverse the ownership chain. Financial institutions and regulators have to do this when implementing anti-money-laundering or antiterrorism checks or when they try to assess risk in general.

The common theme for each of these, and many more, is *dependency modeling*. The idea is quite intuitive: each individual direct dependency is modeled as a relationship between nodes that build out into networks of transitive dependencies. These dependency networks form a graph, and many dependency problems can be resolved with graph pattern matching and graph algorithms, as you will learn in this chapter.

Dependencies as a Graph

Table 11-1 is a common representation of dependencies. Each entry represents an element that directly depends on some other element. These elements might be

routers in a network, parts in a supply chain, or tasks in a project, but the way you process those dependencies is identical in all use cases.

Table 11-1. Tabular representation of a set of dependencies

Element	Depends On
A	B
A	C
A	D
C	H
D	J
E	F
E	G
F	J
G	L
H	I
J	N
J	M
L	M

In this scenario, basic dependency analysis questions like "Is there a *hidden* dependency between `A` and `F`?" can be answered, albeit not entirely conveniently. If you follow the individual dependencies by exploring the table and taking some notes along the way, you will eventually come up with an answer, perhaps even in under a minute: there is no direct or indirect dependency that exists between `A` and `F`.

If such a query were based on Table 11-1, it would look something like Example 11-1. Note that the query only goes down two levels in the exploration—that's why it contains an ellipsis—to indicate that the number of joins would need to be extended for as many levels of depth required in the exploration. In the pathological case of a negative answer like the absence of dependencies between `A` and `F`, a valid answer can only be provided when a complete exploration of all possible dependency chains has been carried out, but not before.

Example 11-1. SQL query template returning all recursive dependencies for a given element

```
SELECT count(*) >0 dependency_exists

FROM dependencies d

  INNER JOIN dependencies d1
    ON d.depends_on = d1.elem
```

```
  INNER JOIN dependencies d2
    ON d1.depends_on = d2.elem

  [...]

WHERE d.elem = 'A'
  AND d2.elem = 'F'
```

The approach in Example 11-1 doesn't work well, even at small scales, because SQL and relational technology weren't designed for recursive path analysis. But knowledge graphs are designed for arbitrary depth path analysis. The same dataset can be converted into a knowledge graph with the simple script in Example 11-2, where the input file contains a CSV serialization of the data in Table 11-1.

Example 11-2. Cypher script to import the dependencies

```
LOAD CSV WITH HEADERS FROM "file:///dependencies.csv" AS row
MERGE (a:Element { id: row.element })
MERGE (b:Element { id: row.depends_on})
MERGE (a)-[:DEPENDS_ON]->(b)
```

The resulting graph is shown in Figure 11-1. Now each entry in Table 11-1 becomes a directed relationship between nodes that indicates a dependency.

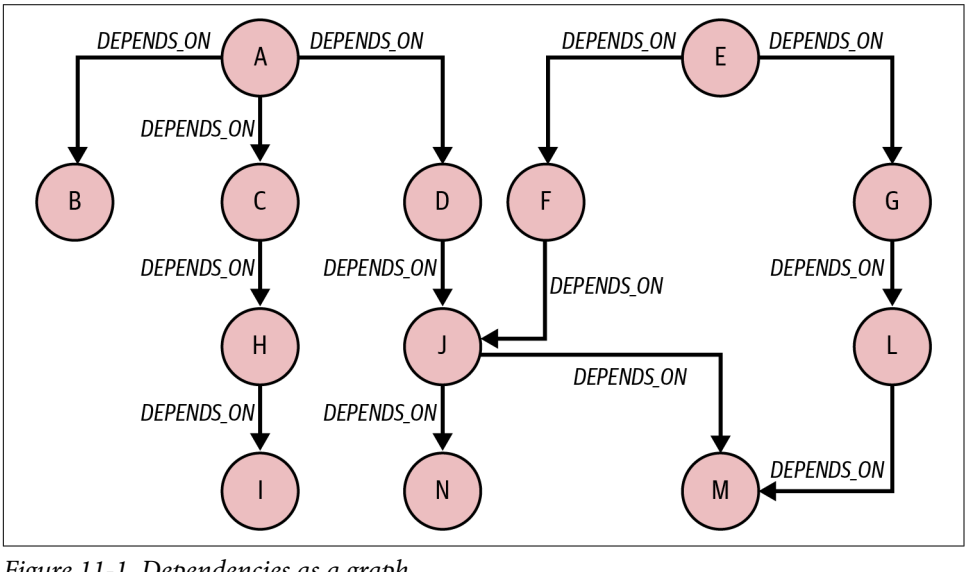


Figure 11-1. Dependencies as a graph

With a knowledge graph, answering the exact same question ("Is there a *hidden* dependency between `A` and `F`?") becomes a trivial task because the dependencies are

explicit. A human will need no more than a few seconds to correctly spot that there is no dependency between `A` and `F`. A graph database supporting such a knowledge graph does the same thing, just much more quickly and at much greater scale.

The Cypher query in Example 11-3 is a complete, runnable query, as opposed to the abbreviated template used in Example 11-1. It will work for any arbitrary depth in a dependency chain without changes because it's natively a variable-length expression. The `DEPENDS_ON*` pattern tells the database to traverse `DEPENDS_ON` relationships to any depth in a path. In addition to the robustness that this introduces, the pattern-based query is significantly more intuitive and compact. In a software development context, this means lower cost of maintenance.

Example 11-3. Cypher query returning all recursive dependencies for a given element

```
MATCH path = (:Element { id: 'A'})-[:DEPENDS_ON*]->(:Element { id: 'F'})
RETURN path
```

Of course, Example 11-3 is a simple query. But the same benefits hold for even complex dependency graphs, as you will learn in the following sections.

Advanced Graph Dependency Modeling

Dependencies are sometimes simple. All you care about is whether a dependency exists or not, just like in Table 11-1. But more complex problems require more data, like how critical or expensive a dependency is or how multiple dependencies combine. From a data modeling point of view, the property graph model offers a natural way to represent qualified dependencies by using different types of relationship combined with properties in them.

Qualified Dependencies

The diagrams in Figure 11-2 show examples of much richer dependencies. In each, the relationship representing the dependency between two entities is qualified by an attribute that describes the strength of the dependency.

In the case of the company ownership graph, you need to know not only that one company owns a part of another company but also the percentage of ownership. Similarly, in the capacity utilization graph, if a service uses a portion of the capacity provided by another lower-level service, it will be necessary to know that percentage. This is what the `capacity` and `part` attributes of the `CONSUMES` and `OWNS` relationships encode. The property will typically be an absolute value or a percentage but always a numeric (or worst case, a value from an enumerated list) in a way that multiple values are composable. This is important, as you will see in the following sections, to

calculate complex impact propagation or to be able to validate the correctness of a dependency model.

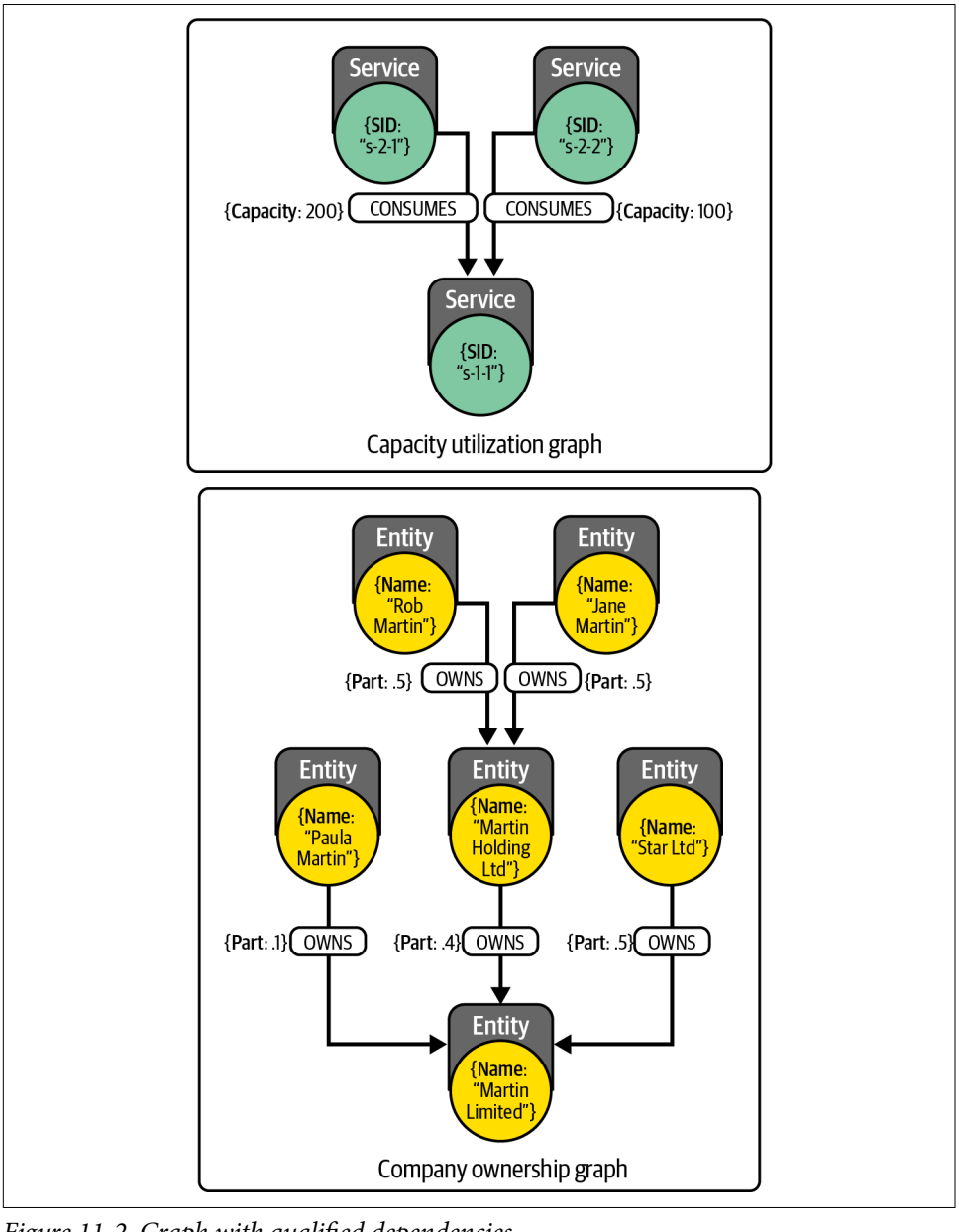


Figure 11-2. Graph with qualified dependencies

Another type of qualifier for a dependency is its temporal validity. This will indicate when the dependency is valid, and therefore, given a date/time, the dependency must be possible to determine whether or not it is active. This case is captured in Figure 11-3 and follows the same modeling approach used to capture the strength of the dependency: properties in the relationships indicating start and end of validity for each dependency.

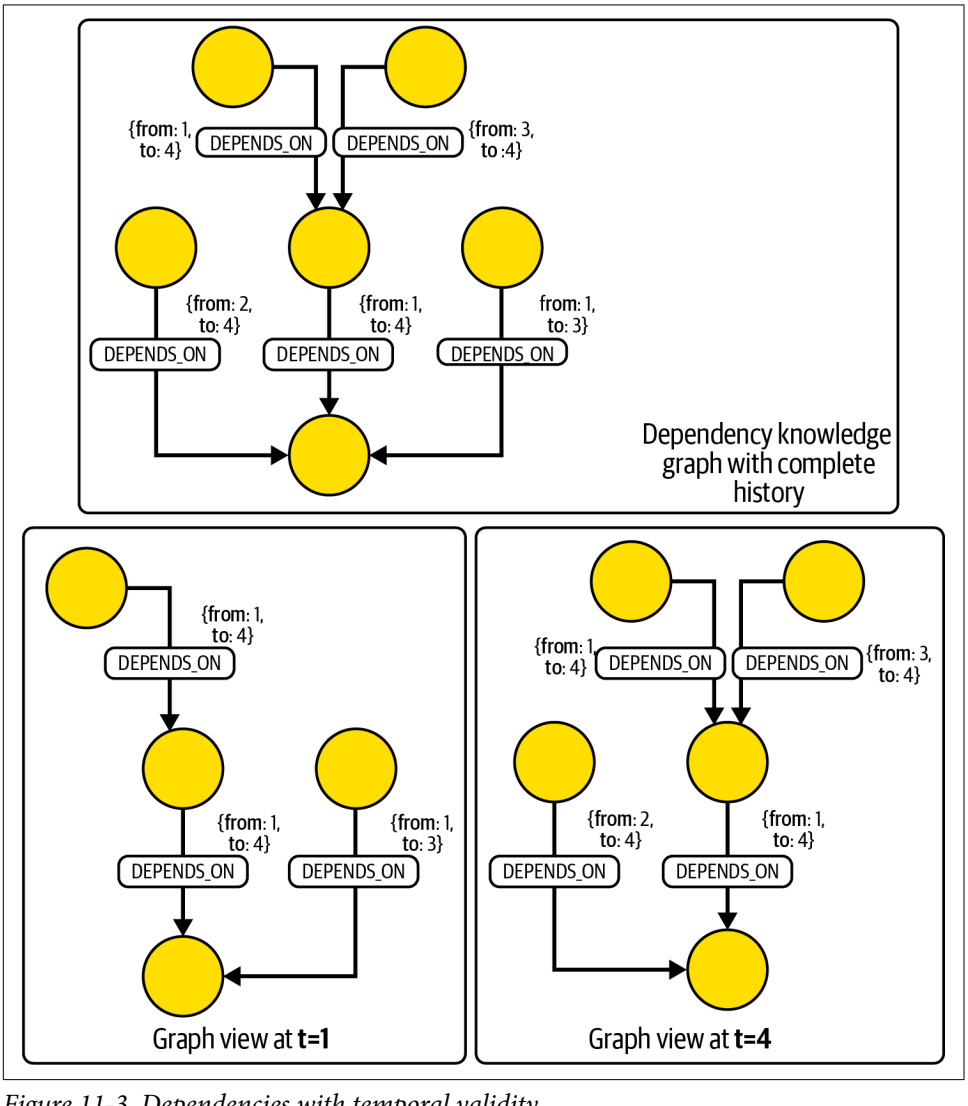


Figure 11-3. Dependencies with temporal validity

Queries in a historized dependency graph like Figure 11-3 require a timestamp parameter `t` that will contextualize the query by exploring only the dependencies that were (or will be[1]) active at time `t`.

Semantics of Multidependency

In the examples in Figures 11-2 and 11-3, all dependent entities (or services) have a dependency on a single other entity through a single outgoing relationship. But in the general case, multiple dependencies are commonplace—for example, when an investor spreads their investments around so that their exposure to any one type of asset is limited or when a supply chain creates multiple routes for the transportation of goods for resiliency. Both are shown in Figure 11-4.

Both examples include elements (nodes) that depend on multiple other nodes, but this multidependency can have different interpretations. Is the logic additive? Or is it an *at least one* type of dependency? These form the two basic interpretations of multidependencies: additive/aggregate or redundant/protection.

Additive dependencies are interpreted as *simultaneously active*, which means that the dependency is spread over the dependent entities. An element *A* depends on another element *B* weighted by *B*'s contribution to *A*. In the example on the left side of Figure 11-4, the portfolio is composed of a number of investments in different types of assets. The fact that one asset contributes to the total with a given percentage (say 30%) implies that the portfolio depends on it up to that percentage. In other words, the impact on the portfolio of something happening to that investment (positive or negative) would be weighted (say by 0.3).

In the case of a negative impact (often the main focus in dependency analysis problems), the damage propagated would be limited to that amount. There are other scenarios where this same logic would apply, such as the capacity of a logical communications link that is the result of bonding the capacity of multiple underlying physical links or a virtualized data storage drive backed by multiple underlying physical drives. Again, if any of the dependent elements (links or drives) were to fail, the result would be a degradation to the capacity of the parent element limited to the failing element's contribution to the total capacity provided by the parent. There may even be a tipping point where the parent element's provision fails entirely when enough of the capacity of the dependent elements fail.

1 Nothing stops you from capturing how the future will look in the dependency knowledge graph. Think of a scheduling application where users can book resources in advance.

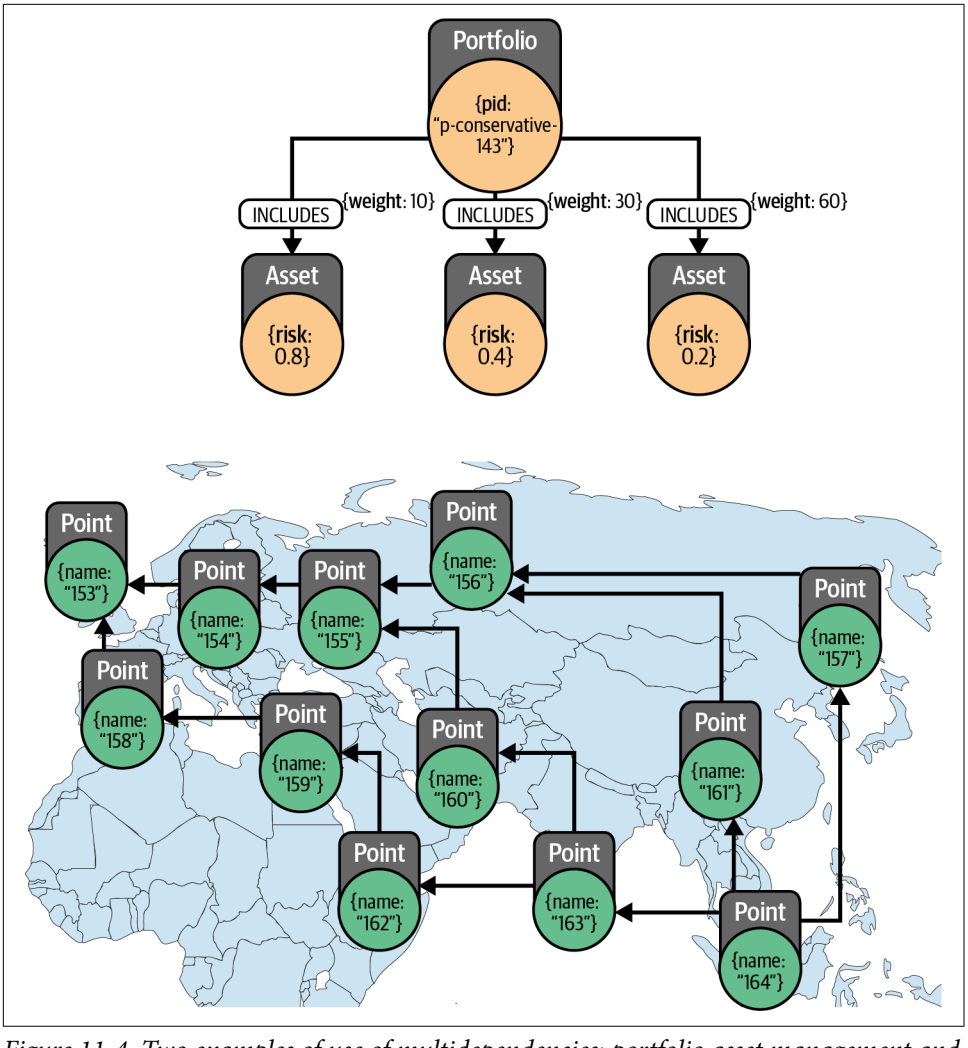


Figure 11-4. Two examples of use of multidependencies: portfolio asset management and supply chain routing

Impact propagation in the dependency graph under this model is computed recursively following a simple expression at each step along the dependency path:

*parent_impact = child1_impact * child1_weight + child2_impact * child2_weight …*

This is visualized in Figure 11-5. The impacts that are measured or detected are introduced in the graph as statements. You will recognize them in the diagram by the magnifying glass icons: a hard drive is down, the value of asset X has dropped by 50%, there is a loss of signal in link X. The derived impacts are shown with their

arithmetic underpinnings and a calculator icon. Finally, the impact propagation paths are highlighted with the background shading.

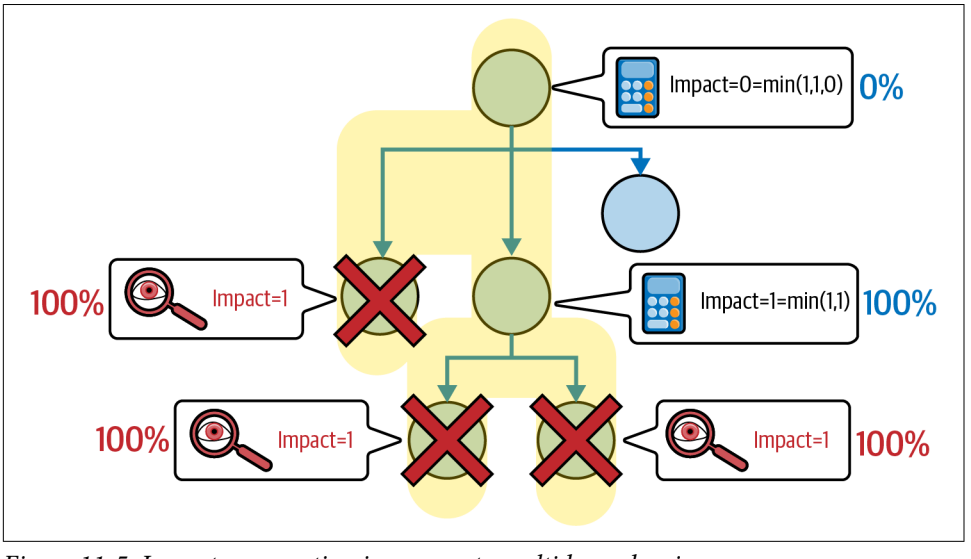


Figure 11-5. Impact propagation in aggregate multidependencies

Redundant dependencies provide fault tolerance. Of all dependent entities in a redundant set, only one is active at any point in time, and the rest are held in reserve until needed. The dependent entity will not be affected as long as at least one of the dependees is available. The consequence is that certain failures will be absorbed by "protection nodes" (those with multiple redundant dependees) and will not be propagated further. These types of situations are common in highly available architectures for software systems where critical servers required to operate 24-7 are protected by other servers that will provide continuity of service if the first one fails.

Something similar happens in broadcast television and communication networks. To guarantee connectivity between two points, multiple alternative routes are created so that in case of failure, one of the remaining alternative pathways is used.

Supply chains work in a similar way with alternative routes used in case of disruption. In Figure 11-4, this is depicted in the diagram on the right-hand side. The preferred route between Singapore and Felixstowe goes through the Suez Canal, but there are alternatives, like the one over the Trans-Siberian Railway.

Energy networks and project plans are yet further examples of scenarios where redundant dependencies are used. These kinds of knowledge graphs have genuinely vast utility.

In all of these scenarios, impact propagation is the key characteristic. It is computed recursively using this expression along the dependency path:

parent_impact = min(child1_impact, child2_impact, …)

This is visualized in Figure 11-6. Like in Figure 11-5, the stated impacts are highlighted with a magnifying glass icon, derived impacts with their underlying arithmetic and a calculator icon, and the impact propagation paths with background shading.

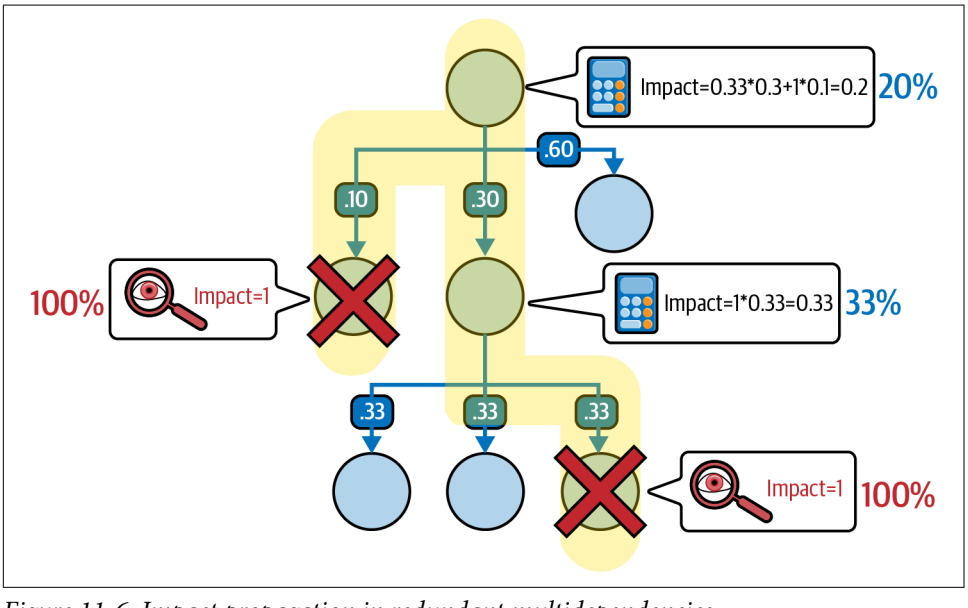


Figure 11-6. Impact propagation in redundant multidependencies

In more complicated systems, there may be a minimum number of dependent entities for a parent entity to operate. Beyond that threshold, guarantees fail. The underlying principle does not change, but the expression to compute the impact is generalized to reflect the presence of an arbitrary threshold:

parent_impact = 0 if sum((1 - child1_impact), (1 - child2_impact), …)
>= parent_threshold else 1

A good example of this could be a database cluster which, in order to be fully functional, requires a majority of the servers to be available provide a quorum for fault-tolerant udpates. The dependency model looks something like the diagram in Figure 11-7.

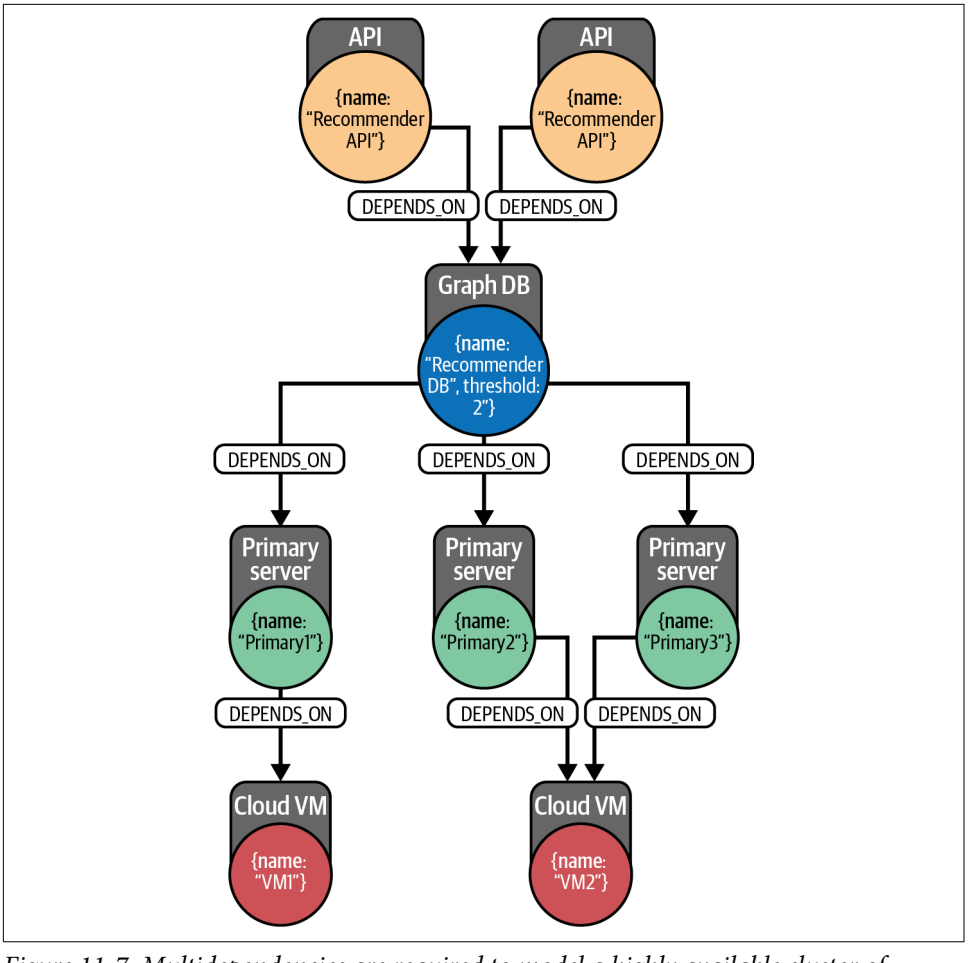


Figure 11-7. Multidependencies are required to model a highly available cluster of servers

In the dependency graph, you can see the database management system has a triple dependency on servers 1, 2, and 3, which themselves depend on underlying virtual machines. The threshold is stored in a property in the `GraphDB` node, which indicates that two out of the three primary servers need to be operational for it to remain available. In the graph, you also see two recommender APIs that depend directly on the `GraphDB` that provides functionality to the end user, if its underlying dependencies are transitively satisfied (that is, the database and therefore its infrastructure are available). The two interpretations of multidependencies described in this section are the two foundational cases, but you can often find them coexisting in the same dependency graph in real-life systems.

Impact Propagation with Cypher

The next step is to be able to analyze a complete dependency graph that includes multidependencies with the two possible interpretations previously described. The dataset (of which an excerpt is shown in Example 11-4) consists of a single file in which every record describes a direct dependency between two elements, its type (`'AGG'` for aggregate and `'RED'` for redundant), and an absolute qualifier.

Example 11-4. CSV file containing the individual direct dependencies forming the graph

```
╒═══════════╤══════════════╤═══════╤═══════╕
│"element"  │"depends_on"  │"mode" │"abs"  │
╞═══════════╪══════════════╪═══════╪═══════╡
│"A"        │"B"           │"AGG"  │"80"   │
├───────────┼──────────────┼───────┼───────┤
│"A"        │"C"           │"AGG"  │"10"   │
├───────────┼──────────────┼───────┼───────┤
│"A"        │"D"           │"AGG"  │"10"   │
├───────────┼──────────────┼───────┼───────┤
│"A"        │"E"           │"AGG"  │"10"   │
├───────────┼──────────────┼───────┼───────┤
  ...
├───────────┼──────────────┼───────┼───────┤
│"Q"        │"H"           │"AGG"  │"30"   │
└───────────┴──────────────┴───────┴───────┘
```

All dependencies are modeled with `()-[:DEPENDS_ON]->()` relationships, and the properties `mode` and `abs` in them indicat, respectively, the type of relationship and the weight. The complete Cypher script to ingest the dataset from the the CSV file in Example 11-4 is presented in Example 11-5.

Example 11-5. Cypher script to load the qualified dependency knowledge graph

```
WITH "file:///qualified-dependencies.csv" AS data
LOAD CSV WITH HEADERS FROM data AS row
MERGE (a:Element { id: row.element })
MERGE (b:Element { id: row.depends_on})
MERGE (a)-[do:DEPENDS_ON]->(b)
    ON CREATE SET do.abs = toInteger(row.abs), do.mode=row.mode
```

Running Example 11-5 creates a knowledge graph like the one shown in Figure 11-8.

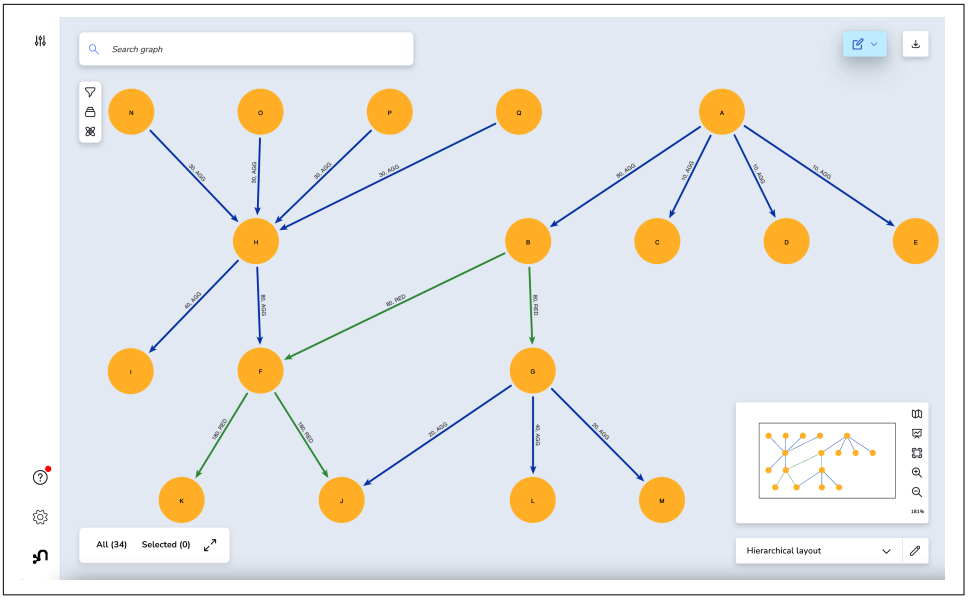


Figure 11-8. Graph combining additive and redundant multidependencies

A generic impact analysis query will take input parameters from the collection of known affected nodes and produce an extended list with the derived impacts (based on the dependency graph) as its result. The most basic type of impact propagation would be the computation of the potential collection of impacted nodes. This analysis is purely topological and returns the list of nodes in the graph for which there exists a direct or indirect dependency (a directed path), given the initial known list of affected nodes. This impact scope analysis is often a first step to detect the subset of nodes to which a more detailed analysis is to be applied. In other words, not all of them will ultimately be impacted, or they will be impacted only partially, given the redundant and aggregate dependencies in the graph. The Cypher for this analysis is shown in Example 11-6.

Example 11-6. Cypher query calculating impact propagation ignoring the semantics and qualifiers of individual dependencies

```
:params declared: [ "I", "K", "J" ]

MATCH (e:Element)-[:DEPENDS_ON*]->(impacted)
WHERE impacted.id IN $declared
RETURN collect(distinct e.id) AS max_impact_list
```

The query in Example 11-6 returns the results in Example 11-7, which are depicted visually in Figure 11-9.

Example 11-7. Results returned by the Cypher query calculating the maximum potential impact of a list of failing nodes

```
╒═══════════════════════════════════════╕
│"max_impact_list"                      │
╞═══════════════════════════════════════╡
│["H","Q","P","O","N","G","B","A","F"]  │
└───────────────────────────────────────┘
```

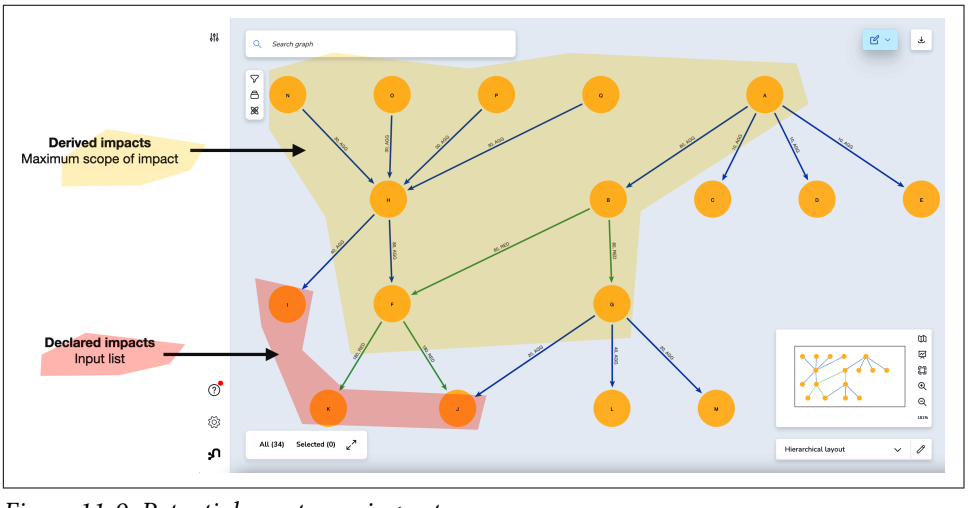


Figure 11-9. Potential worst-case impact

Once the analysis space has been restricted with the previous query, the semantics of the dependencies (qualifiers, multidependencies, and so on) can be taken into account to provide a detailed impact assessment. The query in Example 11-8 shows the Cypher syntax for impact propagation on dependencies in redundant mode. For practical reasons, the assumption is that the threshold for nodes with redundant multidependencies is `1` and that impact is binary—that is, a node is affected `1` or not affected `0`. The parameter `$declared` contains the input list of impacted nodes.

Example 11-8. Cypher query calculating impact propagation in cases of multiple redundant dependencies

```
:params declared: [ "I", "K", "J" ] , current: "F"

MATCH (e:Element { id: $current})-[d:DEPENDS_ON {mode:"RED"}]->(dependee)
WITH e.id AS element, dependee.id AS dependee,
  CASE WHEN dependee.id IN $declared THEN 1 ELSE 0 END AS partial_impact
RETURN element, min(partial_impact) AS derived_impact
```

The query in Example 11-8 returns the result shown in Example 11-9.

Example 11-9. Results of Cypher query calculating impact propagation in cases of multiple redundant dependencies

```
╒═══════════╤══════════════════╕
│"element"  │"derived_impact"  │
╞═══════════╪══════════════════╡
│"F"        │1                 │
└───────────┴──────────────────┘
```

The query in Example 11-10 shows the Cypher syntax for impact propagation on dependencies in aggregate mode. In this case, the impact is augmented with numeric values representing the severity in the input parameters. The data structure used is a map with the ID of the element as key and the numeric representation of the impact to the element between 0 (no impact) and 1 (total impact).

Example 11-10. Cypher query calculating impact propagation in cases of multiple aggregate dependencies

```
:params declared: { "I": 0.5, "K": 1, "J": 0.33 } , current: "G"

MATCH (e:Element { id: $current })-[d:DEPENDS_ON {mode:"AGG"}]->()
WITH e.id AS element, coalesce($declared[endNode(d).id],0) * d.abs AS partial_impact
RETURN element, sum(partial_impact) AS derived_impact
```

The query returns data like that shown in Example 11-11 as its result.

Example 11-11. Results of Cypher query calculating impact propagation in cases of multiple aggregate dependencies

```
╒═══════════╤═══════════════════╕
│"element"  │"derived_impact"   │
╞═══════════╪═══════════════════╡
│"G"        │6.6000000000000005 │
└───────────┴───────────────────┘
```

These implementations in Cypher of the generic dependency propagation rules can give you a solid foundation for any kind of dependency analyis. They are extensible to the particulars of virtually any dependency modeling scenario.

Validating a Dependency Knowledge Graph

An important characteristic of a dependency graph model is that it can be easily validated for correctness. A well-formed dependency graph will have to verify a number of conditions that are scenario specific. However, there are three conditions that are generic enough to be commonly used as the foundation atop which specific conditions can be built.

Validation 1: No Cycles

Technically, a dependency graph is a *directed acyclic graph* (DAG). The presence of a cycle means some component ultimately depends on itself. That is obviously undesirable from a computation point of view but also reflects a bug in the knowledge graph or in the domain itself. The good news is that graphs make structures explicit, and the detection of cycles/loops (and any other shape in your data) is generally as easy as drawing that shape in a query pattern. Example 11-12 is the Cypher query that will detect cycles in a dependency graph. Since a cycle is a path that starts and ends in the same node, the pattern captures any directed path that starts and ends at the same node.

Example 11-12. Cypher query checking correctness of aggregate multidependencies (relative weights)

```
MATCH cycle = (e:Element)-[d:DEPENDS_ON*]->(e)
RETURN cycle
```

It's worth noting that in a large graph this can be an expensive query. That is why you will normally want to:

- Reduce its scope by restricting it to a portion of the graph, maybe providing a set of nodes as input to run the cycle detection around them (just need to add a filter to the path `where e.id in $node_subset`).
- Control the depth of the exploration. The asterisk indicates "path of any arbitrary length" but it is possible to specify a minimum and a maximum with the following notation: `()-[d:DEPENDS_ON*3..45]->`.

Validation 2: Aggregate Multidependencies Add Up to the Expected Total

This type of validation is important to make sure that the model does not describe anomalous situations. It should ensure, for example, that combined, shareholders comprise exactly 100% of an organization, or a link in an optical transmission network with five 20 Gbps bonded lines is reported as 100 Gbps.

The logic is simple: if the weight of each of the component dependencies in an aggregate set is described in relative terms, then the sum of the weights should be 1 (or 100%). This is what the Cypher query in Example 11-13 captures. If the weight is expressed in absolute terms, then probably the model will include each node's total aggregate as an attribute, and in that case, the query shown in Example 11-14 applies.

Example 11-13. Cypher query checking correctness of aggregate multidependencies (aggregate weights)

```
MATCH (e:Element)-[d:DEPENDS_ON { mode:'AGG'}]->()
WITH e, sum(d.rel) AS agg_sum
RETURN e.id AS element_id, agg_sum = 1 AS valid
```

The query in Example 11-13 simply matches on every generic `Element` node and returns `true` if the sum of the `rel` property in all outgoing `DEPENDS_ON` relationships is `1`.

Example 11-14. Cypher query checking correctness of aggregate multidependencies (absolute weights)

```
MATCH (e:Element)-[d:DEPENDS_ON { mode:'AGG'}]->()
WITH e, sum(d.abs) AS agg_sum
RETURN e.id AS element_id, agg_sum = e.total AS valid
```

The query in Example 11-14 applies an identical pattern to the query in Example 11-13, but the sum is calculated on the `abs` property in all outgoing `DEPENDS_ON` relationships, and the total is compared against the `total` property in the element. If these two values match, then the dependencies are well-formed for that node.

Validation 3: consumption does not exceed production. Some scenarios are highly dynamic, and dependencies are added and removed frequently. In these situations, maintaining a property with the current aggregate can be costly and a potential source of inconsistencies if the addition or removal of relationships and the update of the capacity are done in separate transactions. When there is no property to store the capacity, the only way to check that the dependency graph is well-formed at any point in time is by checking if incoming and outgoing dependencies are balanced. This is what the Cypher query in Example 11-15 does.

Example 11-15. Cypher query to check the graph has balanced producers and consumers

```
MATCH (e:Element)
OPTIONAL MATCH ()<-[dependee:DEPENDS_ON { mode:'AGG'}]-(e)
WITH e, sum(dependee.abs) AS agg_sum
OPTIONAL MATCH ()<-[dependee:DEPENDS_ON { mode:'RED'}]-(e)
WITH e, agg_sum, coalesce(min(dependee.abs),0) AS red_sum
WITH e, agg_sum, red_sum, agg_sum + red_sum AS total_cap WHERE total_cap > 0
MATCH (e)<-[dependent:DEPENDS_ON]-()
WITH e.id AS elem, agg_sum, red_sum, total_cap, sum(dependent.abs) AS used
RETURN elem, agg_sum, red_sum, total_cap, used,
  used *100.0 / total_cap AS percentage_used
```

When applied to the graph in Figure 11-8, the results shown in Example 11-16 are produced. For each element in the dependency knowledge graph, the incoming

and outgoing dependencies will be well-formed if the `percentage_used` does not go beyond 100%. In addition to the validation for correctness, the report provides business-level guidance to optimize resource utilization.

Example 11-16. Summary of balanced producers and consumers in the dependency graph

"elem"	"agg_sum"	"red_sum"	"total_cap"	"used"	"percentage_used"
"B"	0	80	80	80	100.0
"H"	120	0	120	120	100.0
"F"	0	180	180	160	88.88888888888889
"G"	80	0	80	80	100.0

Validation 4: redundant dependency configuration aligns with threshold definitions. In multidependency scenarios that use redundant dependencies and where thresholds are defined, the number of members in a redundant set has to be, at a minimum, equal to the threshold; otherwise, the dependency would never be satisfied. This is what the query in Example 11-17 verifies.

A good example of this would be a database cluster like the one described by Figure 11-7 where only one `PrimaryServer` is found. By definition, the threshold of `2` would be impossible to meet, and the `GraphDB` node would be impacted, and that impact would propagate over the dependency network.

The problems uncovered by the validations can be the result of an actual issue in the environment, like the cluster being incorrectly provisioned, or a problem in the data ingestion and dependency graph construction pipeline. Identifying whether it's one or the other is required in order to determine subsequent actions.

Example 11-17. Cypher query checking correctness of redundant multidependencies and threshold definitions

```
MATCH (e:Element)-[d:DEPENDS_ON { mode:'RED'}]->()
WITH e, count(d) AS available_redundant_elements
RETURN e.id AS element_id, available_redundant_elements > e.threshold AS valid
```

Multiple additional validations can be run on a dependency knowledge graph. As you have seen, the combination of the graph representation and the expressivity of Cypher makes it straightforward to write pattern-based expressions to detect them.

Complex Dependency Processing

The impact propagation rules described in "Impact Propagation with Cypher" on page 198 are combined algorithmically to produce a detailed impact assessment of one or several failing nodes. This is often computed as events are being streamed to the graph for real-time event enrichment. It is useful for filtering and prioritization of large volumes of events over a complex system. But the question arises: which results should be looked at first, and which can be ignored? For example, something has happened in your IT infrastructure, and you want to understand if it's low priority because you're protected against it or if it's high priority because it's impacting critical services. A dependency knowledge graph can help to solve problems like single-point-of-failure detection for planning purposes and root cause analysis for postmortem improvements. The next two sections look at these in detail.

Single-Point-of-Failure Analysis

A single point of failure (*SPOF*) is a single element on which a whole system depends, and failure of that element results in failure of the system. The principle can be generalized to parts of a dependency knowledge graph. The diagram in Figure 11-7 shows a simple example. In it you can see how two `PrimaryServer` nodes (`Primary2` and `Primary3`) depend directly on the same `CloudVM` node (`VM2`). The obvious consequence is that the `GraphDB` node (`recommender DB`) actually has a strong dependency on `VM2`. This is a common mistake in real life.

If you deploy multiple instances of a fault-tolerant software system on the same hardware, you will be exposed to a major failure if there is a hardware failure. In real life this isn't limited to virtual machines but can affect all infrastructure, including networks, cooling, power, and even entire data centers. In other words, the level of fault tolerance is superficial because of an underlying SPOF dependency. While this kind of scenario may go unnoticed (and it often does) when systems are designed or when investments are placed, it will not go unnoticed using knowledge graphs.

In topological terms, a SPOF is nothing but another pattern in the graph. As such, you can describe it with a Cypher pattern and surface any instance of it in the dependency knowledge graph.

Example 11-18 shows a generic Cypher expression to detect SPOF situations. Given an element (`selected_node_id`) in the dependency graph, a SPOF is defined as multiple variable-length dependency chains (paths) that converge on any given element `spof`.

Example 11-18. Cypher query checking for SPOF patterns

```
MATCH alertPath = (spof)<-[:DEPENDS_ON*]-(e:Element)-[:DEPENDS_ON*]->(spof)
WHERE e.id = $selected_node_id
RETURN alertPath
```

Not all SPOFs will be fatal. Detailed analysis that takes into account aggregate and redundant multidependencies will be required to determine the ultimate impact in case of failure (and partial failures of a system may be deemed acceptable). But it's fair to say that SPOFs are generally undesirable patterns that, if not worth solving, should at least be monitored. Think, for example, of an energy network where the geography imposes a particular topology that includes a bottleneck (an SPOF). Even if it's an unsolvable geographic feature, it's worth understanding how it impacts the whole network in order to have the right reactive measures in place, like backup diesel generation for critical facilities such as hospitals.

Root Cause Analysis

Root cause analysis is the inverse of the impact propagation problem. Given a collection of impacts, the objective is to try to find whether or not they all depend on a single element. If that is proved, the consequence would be that this common ancestor is the root cause of the observable impacts.

A common scenario where this type of analysis is quite prominent is service assurance systems. They normally are built to monitor complex systems and receive continuous alarms/events/metrics from the components in the system. In a complex system (like communications networks, supply chains, IT infrastructure, and so on), components will depend on others in some complex way, as you have seen several times in this chapter. The alarms indicate some kind of malfunctioning in an element. Sometimes the malfunction is inherent to the element, but it can be caused by some dependent element failing and that failure impacting the others. The monitoring system needs to reduce the volume of alarms so that it is high signal/low noise and can direct interventions accurately. One way to do this is to group all derived alarms with their root cause and only present the root causes. This is often referred to as *alarm correlation* or *event correlation*, and you can intuit how it involves discovering an alarm's direct and indirect dependencies.

To create a pattern for root cause analysis, the first step is to identify the candidates. This is what the Cypher query in Example 11-19 does. Given a collection of impacted nodes (the symptoms), the exploration of the dependencies in the graph will identify the leaf nodes that could potentially explain groups within the collection of symptoms. This will uncover disjoint groups. It is possible and often the case that simultaneous although independent faults are detected. This exploration will immediately group independent symptoms in separate clusters.

Example 11-19. Clustering symptoms around potential root causes using Cypher

```
:param symptoms: ["N","O","P","H","I","A","E"]

MATCH (e:Element)-[:DEPENDS_ON*0..]->(x)
WHERE e.id in $symptoms
  AND NOT (x)-[:DEPENDS_ON]->()
WITH x, collect(distinct e.id) AS explains_faults
WHERE size(explains_faults) > 1
RETURN x.id AS candidate_root, explains_faults
  ORDER BY size(explains_faults) desc
```

The query in Example 11-19 looks for leaf nodes—those that don't depend on others, `NOT (x)-[:DEPENDS_ON]->()`, and can therefore be root causes—in the dependency chain of all symptom nodes. For each of them, the aggregate step `collect(distinct e.id)` identifies which of the symptoms would be explained by the candidate root cause in question. When run on the graph in Figure 11-8, the query produces the results shown in Example 11-20.

Example 11-20. Result of clustering symptoms around potential root causes using Cypher

```
╒══════════════╤═══════════════════════╕
│"candidate_root"│"cluster"              │
╞══════════════╪═══════════════════════╡
│"I"           │["H","N","O","P","I"]  │
├──────────────┼───────────────────────┤
│"K"           │["H","N","O","P","A"]  │
├──────────────┼───────────────────────┤
│"J"           │["H","N","O","P","A"]  │
├──────────────┼───────────────────────┤
│"E"           │["E","A"]              │
└──────────────┴───────────────────────┘
```

You can immediately see that there are two independent clusters: the one potentially explained by `E` and the one potentially explained by `I`, `K`, or `J`.

In a second step, common information retrieval metrics (precision, recall, and F-score) are used to rank and determine which are the more likely ones. The idea is quite intuitive: for each candidate root cause produced in the previous step, three metrics will be calculated:

Precision
: What portion of the elements in the symptom list does this candidate root cause explain?

Recall
: Should this be the actual root cause of the problem, what would have been its maximum impact? How does it compare to the actual observation (the initial list of failing nodes)?

F-score
: Performance metric calculated from the precision and recall.

These metrics are described in Figure 11-10.

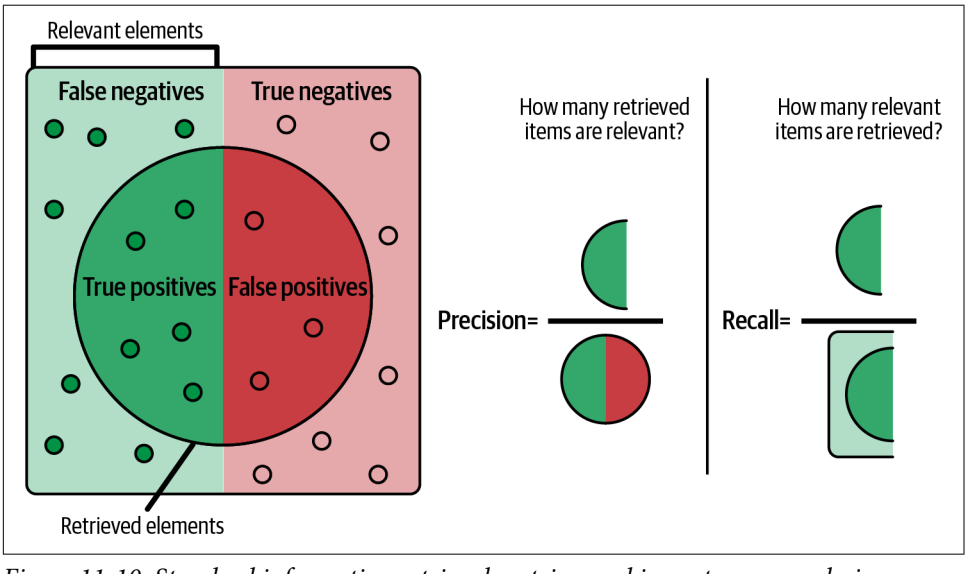


Figure 11-10. Standard information retrieval metrics used in root cause analysis computation

A Cypher implementation using retrieval metrics to refine potential root causes is shown in Example 11-21.

Example 11-21. Complete root cause analysis using Cypher

```
MATCH (e:Element)-[:DEPENDS_ON*0..]->(x)
WHERE e.id IN $symptoms
  AND NOT (x)-[:DEPENDS_ON]->()
WITH x, collect(distinct e.id) AS explains_faults
WHERE size(explains_faults) > 1
WITH x AS candidate_root, explains_faults
MATCH (candidate_root)<-[:DEPENDS_ON*0..]-(x)
WITH candidate_root, explains_faults, collect(distinct x.id) AS potential_max_impact
WITH candidate_root, toFloat(size(explains_faults))/size($symptoms) AS precision,
  toFloat(size([x in potential_max_impact WHERE x in $symptoms]))
    / size(potential_max_impact) AS recall
```

```
RETURN candidate_root.id, precision, recall,
  (2 * precision * recall) / (precision + recall ) AS fscore
  ORDER BY fscore DESC
```

The Example 11-21 query extends the code in Example 11-19 by computing the three metrics and produces the results shown in Example 11-22.

Example 11-22. Complete root cause analysis results using Cypher

"candidate_root.id"	"precision"	"recall"	"fscore"
"I"	0.7142857142857143	0.8333333333333334	0.7692307692307692
"K"	0.7142857142857143	0.5555555555555556	0.6250000000000001
"J"	0.7142857142857143	0.5	0.588235294117647
"E"	0.2857142857142857	1.0	0.4444444444444445

From Example 11-22, you can infer that the most likely root cause is `I` based on the collected symptoms. Candidates like `I`, `J`, and `K` have very similar precision, or in other words, they could be the cause of a good portion of the symptoms detected.

However, if `J` was the root cause, then a significantly higher number of failing nodes should be in the symptom list because they indirectly depend on `J` and would therefore be affected. The fact that they are not in the input list reduces the likelihood of `J` being the actual root cause. The same applies to `K`. Only `I` has a more balanced combination of precision and recall, which is reflected in its F-score.

You can see that the approach depends on the quality of the observed symptoms. The more complete the list, the more accurate the results. In real-life scenarios, it is uncommon to have a complete picture of the current status of the environment, and the consequence is that the results of these kinds of analysis will always produce a list of most likely candidates rather than a single answer. But those results can be combined with other elements like time or any other domain-specific heuristics to improve your chances of making a correct diagnosis.

Vanguard Group Migrating Monoliths to Microservices

The Vanguard Group is one of the world's largest investment management companies. It is the largest provider of mutual funds and the second largest provider of exchange-traded funds.

Vanguard was facing the challenge of refactoring its existing monolithic Java-based systems into a microservices architecture. For that, the company required visibility

into all components in the monolith and, most importantly, the interdependencies between those components. Some of Vanguard's legacy Java code modules contained up to four million lines of code, making the task absolutely unmanageable without software support. Furthermore, as part of the process, Vanguard wanted to prune dead code to reduce technical debt.

The team understood that they were facing a graph problem, and the scale and complexity of their analytic requirements lent itself to knowledge graphs. They automated the construction of the knowledge graph as part of their build process, adding all their existing code artifacts, along with their dependencies. In addition to the "discovered graph" from the build process, they added information from their architecture diagrams to detect misalignments.

Using graph analytics and visual exploration of the graph, the team now evaluates their code against best practices, and derives key metrics to enforce best practices such as constraining the number of service-to-service calls to reduce risk and latency. Also based on metrics derived from the knowledge graph, a code quality scorecard has been produced, which enables collaboration across the business.

Summary

Capturing and exploiting dependencies is a common and important capability. The natural way to represent and process dependencies is to use dependency-centric knowledge graphs because of their expressive power, intuitiveness, and efficiency.

The ideas in this chapter show the basic principles of building dependency-centric knowledge graphs to solve complex problems like impact analysis and root cause analysis. The examples provided also give you an idea of broad applicability of these graphs and will help you extrapolate these principles to your own use cases. From here, you can readily apply and extend these techniques to meet your own needs.

CHAPTER 12

Semantic Search and Similarity

A good proportion of data available in the world is in the form of documents—documents created by humans for consumption by humans and therefore expressed in natural language. But natural language is not easy to exploit programmatically because it does not have a well-defined structure like a table (database or CSV file) or a hierarchy (JSON or XML document). Any automated use of a natural language document will require some preprocessing to extract structured information from it. If you want to go past the basics of text processing (word count, text-based analysis), this can only be achieved using technology called natural language processing (NLP). In this chapter, you will see how the types of structures that result from applying NLP techniques fit naturally into a graph structure and how building knowledge graphs from unstructured data enables more sophisticated exploitation.

Search over Unstructured Data

The first obvious way you want to make programmatic use of the content in natural language documents is to enable search. Search is an area that has had an incredible recent history. In its earliest days, just two decades ago (and surprisingly still today for many services), a search engine would have been a simple index over a set of natural language documents, sometimes even human curated. To use it, you had to type a keyword and hope it matched an index term. This does not particularly help, given the many lexical variations in most human languages (a search for the term *country* would ignore potentially relevant documents containing its plural *countries*). If you add to this inherent complexity of natural languages other elements like the use of alternative spellings (sometimes due to typos), abbreviations, and acronyms, the task does not get any easier. The consequence is that search results were (are) often less than good.

The second part of the search problem is ranking the results. In this basic scenario, the engine cannot do better than sorting by simple relevance metrics like number of appearances of the search term.

Graphs Supercharge Search

You use search every day: to find documents on your hard drive, a product in an online shop, a song on your favorite music streaming service, or virtually anything on the web using a search engine.

Google has revolutionized Web search on both fronts over the years, first with the PageRank algorithm, used to rank search results (web pages) in the Google search engine by leveraging existing connections (hyperlinks) between pages. With PageRank, relevance of web pages to queries is based on whether the pages contain the search term as well as how many relevant pages link to it.

Years later, Google also introduced the use of knowledge graphs to enhance search. Google search today is underpinned by a series of sophisticated algorithms and data, but both PageRank and the Google Knowledge Graph remain fundamental for making sense of the web today.

Search engines have evolved over the years. Keyword-based search has been enhanced with features like wildcard searches, fuzzy searches, and the possibility of combining keywords using logical expressions with different ways of ranking results (like term frequency–inverse document frequency).

The use of knowledge graphs to complement index-based search makes modern engines far superior. Knowledge graphs capture the "intelligence" required to resolve synonyms, adjacent keywords, or domain-related concepts. In fact, they can take search into the territory of question answering. Not only can they return semantically relevant documents for a given search, but they can also be used to provide sophisticated responses to natural language questions. This chapter will cover the first problem, and the second will be explored in Chapter 13.

From Strings to Things: Annotating Documents with Entities

The natural language *text* in documents is a human way of conveying information by describing *entities* and relationships between them. But the same *entities* and relationships can be described in many different ways because human language is very rich (but also ambiguous and contextual). That's why basing any kind of automated processing (search, in this case) of the documents on the *text* representation is both inaccurate and inefficient as hinted at in the previous section. You need to move this

processing to a higher level of abstraction. In fact, the goal is to search for "things not strings" (as described in the Google blog by Amit Singhal: "Introducing the Knowledge Graph: Things, Not Strings").

The first step to achieve this is to apply the most basic type of processing of natural language: the extraction of the entities "hidden" in the text. This process is often called *text annotation*, and the NLP technique is *named-entity recognition* (NER).

To understand what NER will do for us, take the example of a document that contains the sentence "The *New York Times* is a daily newspaper and its headquarters is on the west side of Midtown Manhattan in New York City." NER will unambiguously uncover that both an organization "The New York Times" and a location "New York" are mentioned in the text even though both are described with the same words. Making these results explicit in a knowledge-graph-powered search engine would then make it possible to do both location- and company-based searches, whereas a text-based search would only be able to find a match on the text "New York" but would not be able to distinguish whether the text refers to one or the other.

To illustrate the application of NER for building a knowledge graph that enables entity-based search, see Table 12-1. It contains a list of documents (articles) with their titles and summaries.

Table 12-1. A list of articles to put in a semantic search knowledge graph

Article list
Title: Twitter chair Patrick Pichette joins graph data platform Neo4j board of directors. **Fragment:** Pichette is currently a partner at Inovia Capital and is also currently chair of commerce platform Lightspeed. His 30 years of financial and operating expertise includes roles at Google, Sprint Canada and Bell Canada, with a focus on digital transformation and hyper-growth. According to Pichette, "Neo4j's graph technology offers a truly unique solution to solve some of the world's most complex challenges, with a clear focus on promoting transparency and positive social change." **author:** Digital Nation Staff **url:** *https://www.itnews.com.au/digitalnation/news/twitter-chair-patrick-pichette-joins-graph-data-platform-neo4j-board-of-directors-572498*
Title: JupiterOne Unveils Starbase for Graph-Based Security **Fragment:** JupiterOne has announced Starbase, an open source tool for security analysts to collect information about the organization's assets and their relationships and pull them into an intuitive graph view for cyber asset management. The graph data model, based on open source graph data platform Neo4j, makes it easier to see relationships between different assets and to perform complex relationship analysis, the company said in a statement. "As a CSO, I'm a big advocate of bringing graph-based technology to security, given its power to reshape how we think about security threat defense" said Sean Catlett, CSO at Slack. **author:** Dark Reading Staff **url:** *https://www.darkreading.com/dr-tech/jupiterone-unveils-starbase-for-graph-based-security*

You will need two basic components, as depicted in Figure 12-1: a software package that can carry out the entity extraction from the natural language text in the article collection and a graph platform to store the results of the process as connected entities.

For this first example, you will use Hugging Face to take care of the entity extraction and Neo4j for the storage and analysis of the results in the form of a knowledge graph.

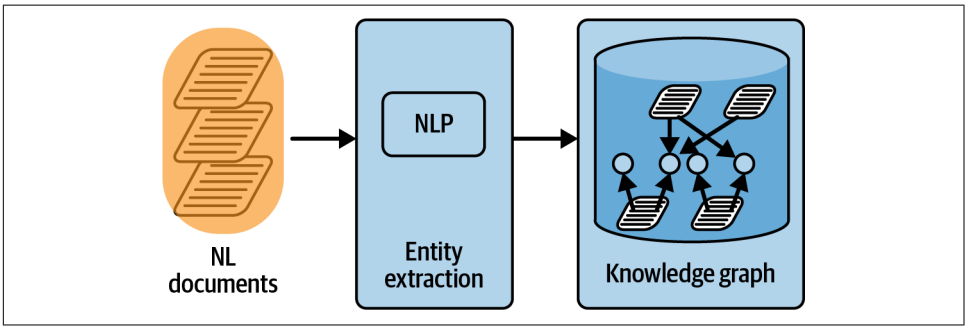


Figure 12-1. Extracting entities from natural language text and storing them in a graph database

Under its Token Classification subtasks, Hugging Face includes NER. Its use is quite straightforward if you stick to the defaults. Example 12-1 shows a Python code snippet that extracts the entities from the title of the first of the articles in the list. In a complete example, you would iteratively apply this type of analysis to all articles in your collection.

Example 12-1. Extracting entities from natural language text using Python

```
from transformers import pipeline

ner_pipe = pipeline("ner", aggregation_strategy="simple")

title = """Twitter chair Patrick Pichette joins graph data
  \ platform Neo4j board of directors."""

for entity in ner_pipe(title):
  print(entity)
```

When not overridden, the pipeline uses the bert-base-NER model, which performs great in the example with general English language. If you are working on a very specific domain and your natural language documents use highly technical terms, you will want to use an NER model trained specifically for that domain. Both the creation and the use of a custom model is beyond the scope of this book, but the Hugging Face documentation is very useful.

The output from the code is shown in Example 12-2 and is basically the list of entities identified in the text, with their types and a score associated with each of them.

Example 12-2. Entities extracted

```
{'entity_group': 'ORG', 'score': 0.9961534, 'word': 'Twitter', 'start': 0, 'end': 7}
{'entity_group': 'PER', 'score': 0.9972605, 'word': 'Patrick Pichette',
  'start': 14, 'end': 30}
{'entity_group': 'ORG', 'score': 0.8314789, 'word': 'Neo4j', 'start': 62, 'end': 67}
```

Because entities can be of different types and not all are equally important in the context of the natural language text being analyzed, it is quite common for NER processors to return the following in addition to a list of entities:

type
: Is it a person? Is it a location? Is it an organization? The set of categories will depend on the specific model used. The bert-base-NER distinguishes four types of entities: location (`LOC`), organization (`ORG`), person (`PER`), and miscellaneous (`MISC`).

salience
: The relative importance in the text analyzed or, in other words, the entity's relevance. Is the entity central to the text (higher score/salience), or is it just mentioned tangentially (lower score/salience)?

Now that you know how to extract entities from natural language, you can store these entities along with the information about the articles they come from. When doing that, you will make the connection between articles and entities explicit and weighted (the salience) and form a knowledge graph like the one in Example 12-3.

As in the previous example, for simplicity the code only shows the process for a single entry in the article list (the title, the author, the URL, and a fragment of text). A complete version would iterate over the list of articles and invoke the following logic for each of them.

Example 12-3. Writing entities to a knowledge graph

```
from neo4j import GraphDatabase

driver = GraphDatabase.driver("bolt://your.db.ip:7687",
  auth=("neo4j", "pwd"))

title = "the title of the article"
fragment = "article fragment"
author = "the author of the article"
url= "https://the.url.of.the.article"

entity_list = []
for entity in ner_pipe(title + fragment):
    entity_list.append(entity)
```

```
cypher_query = '''
MERGE (a:Article { url:$url}) ON CREATE SET a.title= $title, a.text= $frg
MERGE (p:Person { name: $author})
MERGE (a)-[:has_author]->(p)
WITH a UNWIND $entityList AS entity
MERGE (e:Entity { name: entity.word , type: entity.type })
MERGE (a)-[:references { salience: entity.score }]->(e)
'''

with driver.session(database="neo4j") as session:
  session.write_transaction(
    lambda tx: tx.run(cypher_query, url=url, author=author, title=title,
                      frg=fragment,
                      entityList=[{ "word": x["word"], "type":x["entity_group"] }
                                  for x in entity_list])
  )

driver.close()
```

This code produces a knowledge graph like the one in Figure 12-2. Such a graph enables a much richer use of the data in the documents.

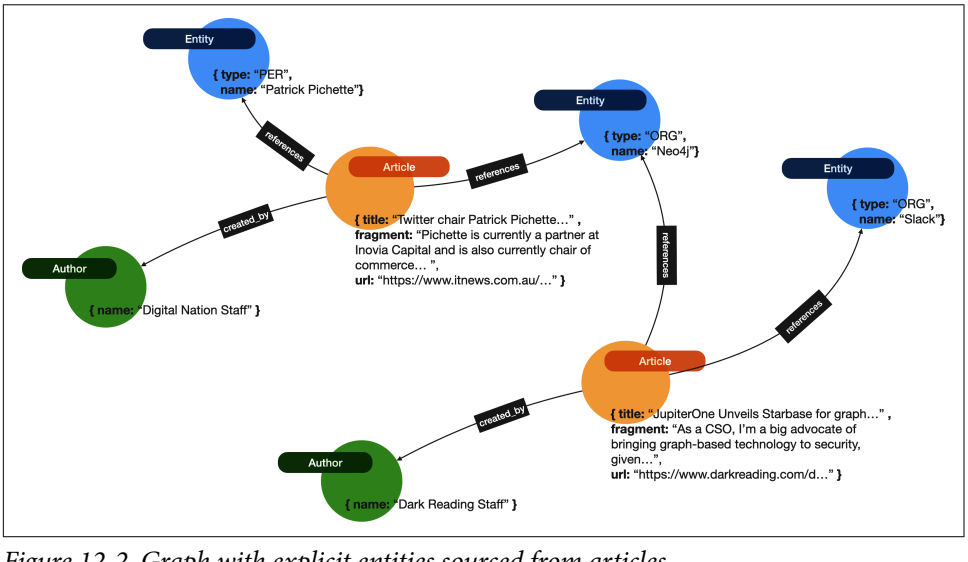


Figure 12-2. Graph with explicit entities sourced from articles

Having a collection of entities that make the content of the articles explicit substantially enriches the options for search. You can now list the entities mentioned in the articles, or in other words, answer the question "What is this document about?" with the simple Cypher query shown in Example 12-4.

Example 12-4. A Cypher query shows entities referenced in articles

```
MATCH (a1:Article)-[:references]->(e:Entity)
WHERE a1.url = $url
RETURN e.name AS entityName, e.type AS entityType
```

The exact same pattern also answers the inverse question "Which articles mention this entity?"" The complete Cypher would be the one shown in Example 12-5.

Example 12-5. A Cypher query shows articles that reference specific entities

```
MATCH (a:Article)-[:references]->(e:Entity)
WHERE e.name = $entityName AND e.type = $entityType
RETURN a.url AS articleLink, a.title AS articleTitle
```

This technique also helps determine popularity and trends at the entity level, as opposed to the much less meaningful word level. Think of an entity frequency metric versus a basic word count. You can aggregate mentions of entities to compute the most or least popular entities of all time or over a particular time range. You can even analyze trends for entities over time and see how their popularity evolves, just by counting how frequently they appear in documents over a given time period.

Example 12-6 shows how to get the data points to populate a monthly time series with the frequency of the appearance of a given entity in articles published each month.

Example 12-6. A Cypher query to find popular entities over a period of 12 months

```
MATCH (c:Entity { name: $entityName, type: $entityType })
WITH c, date() - duration("P1Y") AS startdate
UNWIND range(0,11) AS increment
MATCH (c)<-[:references]-(a:Article)
WHERE startdate + duration("P"+ (increment - 1) +"M")
  < a.published < startdate + duration("P"+ increment +"M")
RETURN startdate + duration("P"+ increment +"M") AS date, count(a)
```

Navigating the Connections: Document Similarity for Recommendations

You have seen how document annotation surfaces a graph that enables search at a higher level of abstraction (entities versus words). Now you will see that the analysis of the structure of this graph, as Figure 12-3 shows, can help you uncover similarities between documents with very high precision.

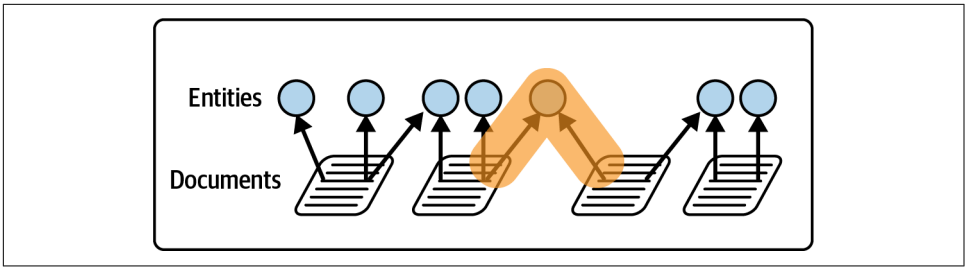


Figure 12-3. Enabling navigation from one document to another via entities

Content-based recommender systems are centered on the features of the items to recommend—"You may also like X"—as opposed to collaborative filtering ones that focus on the analysis of the behavior of the consumers—"Customers like you also bought/read/watched X." If you are browsing for desks on your favorite furniture ecommerce and you click on a white solid wood one, it should not surprise you if you see on the side of your screen a list of "similar items" with more white wood desks or items that share a good number of features with the one you are looking at. It's easy to extrapolate this approach: the items are the articles, and the features are the entities that you extracted from them.

All you need now is a systematic way to combine the feature matches to come up with a similarity metric. Ordering the results by that metric will give you the top *n* elements to recommend. In the case of the furniture ecommerce, it might be tricky to determine whether a match on the color is more important than a match on the material, so the weights would probably be determined by domain experts. In this case, the salience of the entity in a document gives a useful hint.

Taking this step by step, the starting point is that articles are related to each other by the entities they share. Given two articles, a simple graph pattern like the one shown in Example 12-7 will find the collection of entities that are referenced in both of them.

Example 12-7. A Cypher query shows entities that are referenced in both articles

```
MATCH (a1:Article)-[:references]->(e:Entity)<-[:references]-(a2:Article)
WHERE a1.url = $url1 AND a2.url = $url2
RETURN e.name AS entityName, e.type AS entityType
```

The Cypher query takes only the URLs (unique identifiers) of the articles. In the example, the results would show that for the two articles in Table 12-1, Neo4j Inc. is the only entity that happens to relate to both.

This simple pattern can be extended to compute the similarity metric for any pair of documents. In the query in Example 12-8, the similarity is defined as the weighted sum of entities shared between two articles. The bigger the entity overlap, the greater the similarity between the articles. But not all entities are counted in the same way—remember that the relationships between articles and entities are weighted with the salience of the entity in that particular document.

Example 12-8. A Cypher query calculating a similarity metric between a pair of articles

```
MATCH (a1:Article)-[r1:references]->(e:Entity)<-[r2:references]-(a2:Article)
WHERE a1.url = $url1 AND a2.url = $url2
RETURN sum(r1.salience * r2.salience) AS similarity_metric
```

This is a very simple graph pattern but an extremely powerful metric that will help you offer a recommendation about what to read next, based on the assumption that the reader will want to continue reading about similar topics.

Nearly as important as having a way of measuring *how similar two items are* is a precise description of *how exactly they are related* because that will help you produce not only a relevant recommendation but also an explainable one. It turns out that it is easily in reach of the knowledge graph. Example 12-9 shows the complete recommendation query returning the top five similar articles along with an enumeration of the shared entities that explain the similarity.

Example 12-9. A Cypher query finds similar articles computing the similarity metric on the fly (at query time)

```
MATCH (a1:Article)-[r1:references]->(e:Entity)<-[r2:references]-(recommendation)
WHERE a1.url = $url1
RETURN recommendation,
       sum(r1.salience * r2.salience) AS similarity_metric,
       collect(e) AS explanation
ORDER BY similarity_metric DESC LIMIT 5
```

In some cases, it will make sense to compute the recommended set on the fly as in Example 12-9, but sometimes you will want to enrich the graph by materializing the similarities between articles with a new weighted relationship directly connecting similar articles where the weight in the relationship would encode the similarity metric computed with the previous formula. This approach is described visually in Figure 12-4.

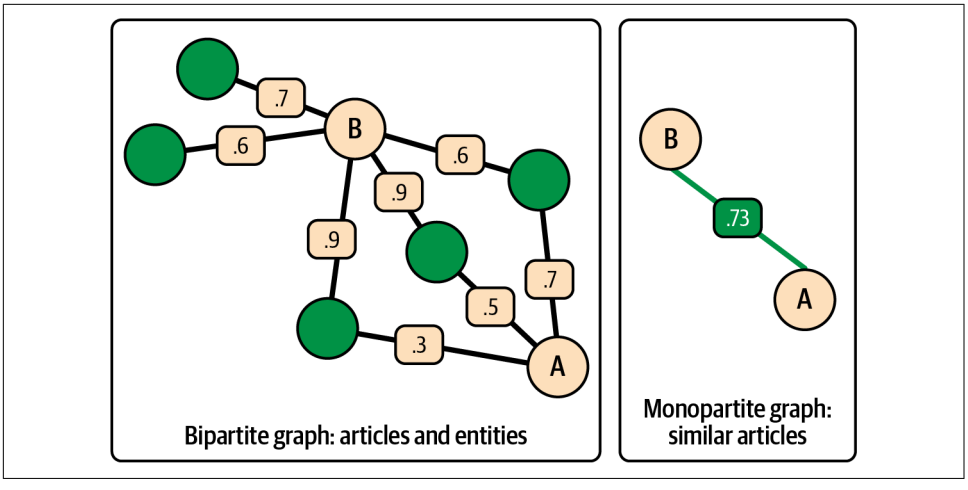


Figure 12-4. Creating a new weighted similarity relationship between articles

The creation of such relationship would require just a minor change to the previous query, as seen in Example 12-10.

Example 12-10. A Cypher query creates a new weighted similarity relationship between articles

```
MATCH (a1:Article)-[r1:references]->(e:Entity)<-[r2:references]-(a2:Article)
WITH a1, a2, sum(r1.salience * r2.salience) AS similarity_metric
MERGE (a1)-[:similar { metric: similarity_metric }]-(a2)
```

If you want to run this type of query on a large graph, you'll want to batch it instead of running it in a single transaction. You can achieve this with the Cypher clause `CALL { … } IN TRANSACTIONS` or using the periodic execution methods in the APOC library.

Once the metric is persisted in the graph, generating recommendations based on a selected article becomes a lot simpler, as shown in Example 12-11.

Example 12-11. A Cypher query finds similar articles using a precomputed similarity metric

```
MATCH (a1:Article)-[r:similar]-(recommendation)
WHERE a1.url = $url1
RETURN recommendation,
       r.metric AS similarity_metric
ORDER BY similarity_metric DESC LIMIT 5
```

The Cold Start Problem

Document similarity is useful for recommending what to show, read, or buy next, but it requires some context, something to anchor your queries on. The previous recommendations selected an article (or a set of recently read articles) as the anchor points.

But what about a "cold start" where there are no anchor points in your knowledge graph? How could you generate an initial recommendation, for example, to populate a home page and overcome the cold start problem? In that case, it might be reasonable to display currently trending topics. Those can be found using the query shown in Example 12-12, which is similar to the trend analysis query in Example 12-6.

Example 12-12. A Cypher query that ranks the most popular topics over the last six months

```
MATCH (c:Concept)<-[:refers_to]-(a:Article)
WHERE date() - duration("P6M") < a.datetime < date()
RETURN c.label, count(a) AS freq LIMIT 10
```

Example 12-12 doesn't directly provide a semantically rich answer to a question since at cold start there is no such question. It does provide some way of making forward progress into the world of linked entities and semantic search.

While not as computationally cheap as the approach in Example 12-12, there are other ways you could lessen the effects of the cold start problem. For example, you could run a similarity algorithm (as per Chapter 5) on the existing graph. The results of that algorithm would immediately be able to suggest similar things to the user, once the user anchors to a particular topic/article/etc. Moreover, with careful design, the similarity algorithm can be run periodically (on a secondary server), with its results being written directly into the production graph.

Making the Annotation Semantic with an Organizing Principle

Document annotation based on NER has been a significant step in the right direction, but it's limited in two obvious ways:

- Entities are not disambiguated. The consequence is that if one of the documents in my collection mentions the United Kingdom and another one does the same using the acronym UK, the NER software will probably successfully identify the entities as locations but will not be able to determine that they both represent the same thing.
- There are no relationships linking entities and capturing the domain knowledge. To continue the geography-based example, a third document mentioning Wales would have the location entity identified but the fact that there is a `PART_OF` relationship between Wales and the United Kingdom is not known to search/recommendation systems.

The way to overcome these limitations is to match the entities to known entries in an organizing principle to form a semantic knowledge graph. Recall that an organizing principle can be an ontology, a concept scheme, or a vocabulary, and it describes the different entities in a particular domain as well as how they relate to one another.

The adoption of standard organizing principles varies by industry. One of the more prolific ones in this area is the pharmaceutical, health care, and life sciences industry. Examples of well-known and mature organizing principles are Medical Subject Headings (MeSH), a comprehensive controlled vocabulary for indexing journal articles and books in the life sciences, and the Disease Ontology, a standardized ontology for human disease.

A visual intuition of this idea can be produced as an extension of the image in Figure 12-3 by adding to it an organizing principle. The extra layer would effectively offer new ways to explore the data by bringing in explicit domain knowledge as seen in Figure 12-5.

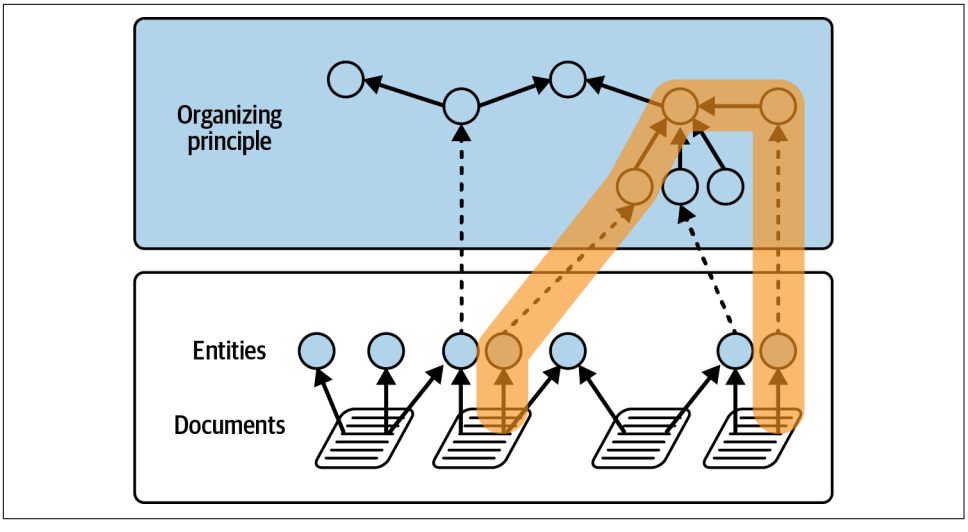


Figure 12-5. An organizing principle offers additional paths for search and exploration

The process of matching entities to uniquely defined entries in a shared concept scheme is often referred to as *named-entity linking* or *entity disambiguation*.

This task is often bundled with the entity extraction process and carried out by specifically trained extractors that are aware of the target organizing principle and that generate entities prematched to elements in that organizing principle. When using general-purpose NER engines like the ones described in the previous sections, we will have to apply custom algorithms, often based in heuristics, to connect the entities resulting from the NER with the elements in the organizing principle. That use case will be explained later in the chapter, but first you should understand the more common approach.

For this example, you will use a different natural language package: the Google Cloud Platform (GCP) Natural Language API. The GCP Natural Language API includes an entity extraction feature very similar to the one offered by Hugging Face but with one interesting addition: the API adds some metadata to the entities it extracts, including Wikipedia URLs. These URLs will help you to match them to entities in an organizing principle in a nonambiguous way.

Just like in Example 12-1 with Hugging Face, in Example 12-13 you can see a small code fragment that calls the GCP Natural Language API to run entity recognition on the title of one of the articles used in the previous section.

Example 12-13. Python code shows how to call the GCP Natural Language API

```
from google.cloud import language_v1

client = language_v1.LanguageServiceClient.from_service_account_json
  ('services.json')

text = u"Twitter chair Patrick Pichette joins
  graph data platform Neo4j board of directors"
document = language_v1.Document(
    content=text, type_=language_v1.Document.Type.PLAIN_TEXT
)

response = client.analyze_entities(request={"document": document})

for entity in response.entities:
    print(entity)
```

When you run this code, you get a list of entities with their type and salience, as shown in Example 12-14. Note that they are returned with some additional metadata that effectively disambiguates them by linking them to Wikipedia entries (other elements like the Google Knowledge Graph ID have been removed for clarity).

Example 12-14. A list of entities with Wikipedia links

```
name: "Patrick Pichette"
type_: PERSON
metadata {
  key: "wikipedia_url"
  value: "https://en.wikipedia.org/wiki/Patrick_Pichette"
}
salience: 0.6320402026176453

name: "Twitter"
type_: ORGANIZATION
metadata {
  key: "wikipedia_url"
  value: "https://de.wikipedia.org/wiki/Twitter"
}
salience: 0.22149702906608582

name: "Neo4j"
type_: ORGANIZATION
metadata {
  key: "wikipedia_url"
  value: "https://en.wikipedia.org/wiki/Neo4j"
}
salience: 0.020158693194389343
```

The solution to the linking problem would be complete if you could find an organizing principle that would include in its elements the same metadata (a reference to the Wikipedia page). Interestingly, some public ontologies and knowledge bases like Wikidata or DBPedia are cross-referenced, and they do include the Wikipedia page reference in their elements.

As an example, Example 12-15 shows a fragment of the Wikidata graph describing the entity Neo4j (identified as `wd:Q1628290`). The description includes a statement indicating that the Wikipedia page *https://en.wikipedia.org/wiki/Neo4j* is about (schema:about) that particular entity.

Note that Wikidata uses numeric values to identify all elements in it. Just like Neo4j is identified as `wd:Q1628290`, it is classified as a graph database, which is itself identified as `wd:Q595971`.

Example 12-15. A fragment of the code from the Wikidata graph for Neo4j

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix schema: <http://schema.org/> .
@prefix wdp: <http://www.wikidata.org/prop/direct/> .
@prefix wd: <http://www.wikidata.org/entity/> .

wd:Q1628290
  rdfs:label "Neo4j"@de ;
  schema:description "graph database implemented in Java"@en ;
  wdp:P31 wd:Q595971 .

<https://en.wikipedia.org/wiki/Neo4j> schema:about wd:Q1628290 .
```

With these two parts, it's straightforward to build a semantic knowledge graph, as described in Figure 12-6.

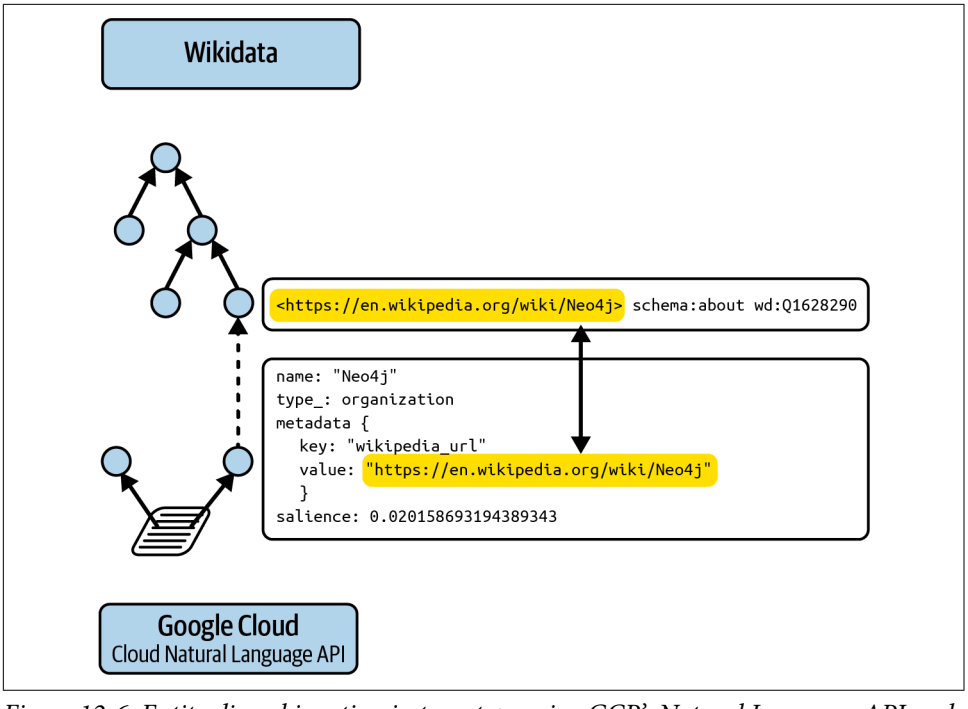


Figure 12-6. Entity disambiguation in two steps using GCP's Natural Language API and Wikidata

The following example shows a Cypher-only version of this process. You can do that because the APOC library in Neo4j offers wrapper methods that invoke the GCP Natural Language API without having to write any external code as in previous examples in this chapter.

1. Import articles as nodes in the graph. The Cypher statement in Example 12-16 parses the list of articles as a CSV file (*articles.csv*) with four columns: `uri`, `title`, `body`, and `datetime`. It also creates a node for each of the records.

   Example 12-16. Loading articles into a knowledge graph

   ```
   LOAD CSV WITH HEADERS FROM 'file:///articles.csv' AS row
   CREATE (a:Article { uri: row.uri})
   SET a.title = row.title, a.body = row.body, a.datetime =
     datetime(row.date)
   ```

2. Import the organizing principle into the graph. Assume that the organizing principle is described using any of the common W3C standards: `OWL`, `SKOS` or

RDFS. You then rely on the neosemantics library for the import, as shown in Example 12-17.

Example 12-17. Import the organizing principle for the knowledge graph using neosemantics

```
CALL n10s.graphconfig.init({ handleVocabUris: "IGNORE",
  classLabel: "Concept", subClassOfRel: "broader"})

CALL n10s.skos.import.fetch
  ("path-to-file-containing-organizing-principle","RDF/XML")
```

3. Run entity extraction on the contents of the articles and link the returned entities with the organizing principle. Notice the use of the `apoc.nlp.gcp.entities.stream` method in Example 12-18 that is in charge of calling the GCP Natural Language API used in this example.

Example 12-18. Call the GCP Natural Language API from Cypher to perform entity extraction

```
CALL apoc.periodic.iterate(
  "MATCH (a:Article)
  WHERE a.processed IS NULL            // (1)
  RETURN a",
  "CALL apoc.nlp.gcp.entities.stream([item in $_batch | item.a], {   // (2)
    nodeProperty: 'body',
    key: $key
  })
  YIELD node, value
  SET node.processed = true            // (3)
  WITH node, value
  UNWIND value.entities AS entity      // (4)
  WITH entity, node
  WHERE NOT (entity.metadata.wikipedia_url is null)
  MATCH (c:Concept {altLabel: entity.metadata.wikipedia_url})  // (5)
  MERGE (node)-[rt:refers_to]->(c) SET rt.salience = entity.salience", // (6)
  {batchMode: "BATCH_SINGLE", batchSize: 10, params: {key: $key}})
YIELD batches, total, timeTaken, committedOperations
RETURN batches, total, timeTaken, committedOperations;
```

A few comments on the previous code fragment:

- The logic is organized as an iterative producer-consumer structure `apoc.periodic.iterate`. The first block `"MATCH ..."` streams batches of size `batchSize: 10` to the second block `"CALL ..."` that processes them

- In (1) you iterate over all articles that have not been processed yet. In (3) you flag articles as `processed` to make sure you do it only once per article.
- In (2) you invoke the GCP Natural Language API for each article in the batch. The method call includes two additional parameters: the name of the property containing the natural language text to be analyzed (`body` in this example) and the API key required to use the service.
- In (4) you iterate over the entities returned by the API, and you create nodes of type `Concept` for each one of them in (5).
- Finally, in (6) the `Article` nodes are linked to the `Concept` nodes through a `refers_to` relationship, weighted with the salience also returned by the API.

The result of this process is a graph quite similar to the one created in the previous section but now with the explicit semantics overlay added by the organizing principle:

- Entities become unambiguous and uniquely identifiable using a *concept scheme*. Such schemes can be public or private, use case or industry specific, departmental or enterprise wide.
- Entities become more interconnected in a meaningful and explainable way, offering new paths for exploration and analysis.

What kind of semantic exploitation can you carry out now? Well, now if you search for articles related to "NoSQL database management systems," you would get results even though no single article explicitly mentions that set of terms.

Why is that? Because by bringing the organizing principle into the graph, you make it explicit that Neo4j is a graph database, and a graph database is a kind of NoSQL database (along with column stores, document databases, and key-value stores), as shown in Figure 12-7. This is what is often referred to as *semantic search*. You have managed to make your search independent of the ambiguous natural language text and base it on well-defined domain entities. In other words, you have effectively implemented the "things, not strings" principle.

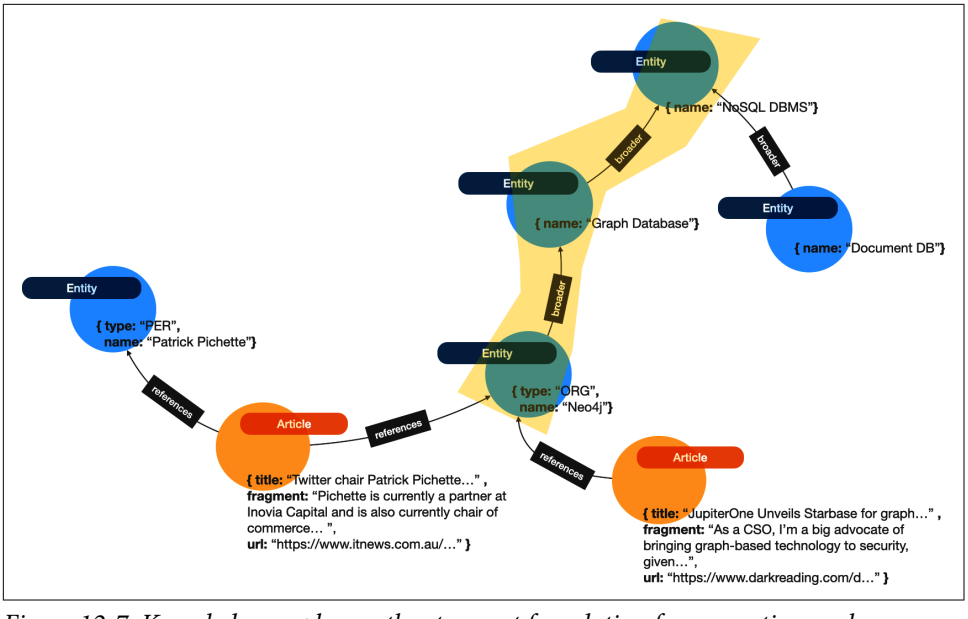


Figure 12-7. Knowledge graphs are the strongest foundation for semantic search

In making the semantics (your knowledge about a domain) explicit, you are making them programmatically exploitable. A simple Cypher query like Example 12-19 captures the basic logic: You select the concept you are interested in, and the pattern `(c:Concept)<-[:broader*0..]-(sc)` recursively navigates the organizing principle to find any other concept that is a subconcept, at any level of depth. As a consequence, the query will return all the `Article` nodes directly or indirectly connected to the selected concept.

Example 12-19. A Cypher query finds articles based on concepts

```
MATCH (c:Concept)
  <-[:broader*0..]-(sc)<-[:refers_to]-(article:Article)
WHERE c.prefLabel = "NoSQL database management system"
RETURN article.title AS searchResult
```

This approach can be extended with multiple organizing principles. The articles in the example, just as products in a catalog can be classified according to multiple dimensions, effectively creating multiple complementary paths for both semantic search and similarity. This is an extremely powerful pattern and supercharges business information systems with search, in much the same way that knowledge graphs have supercharged Web search.

Here are some real-world scenarios where you can see these ideas in action.

In the field of health care, document annotation is applied to discharge summaries, clinical notes, scientific literature, and clinical trial protocols, among others. All of these documents are typically unstructured text. The results can be used to do advanced search and analysis of patient data, and it is also a step toward building a 360-degree view of the patient.

In news and media, document annotation can associate articles with searchable metadata describing key entities like persons, objects, locations, and events. Recommendations and personalization are the most obvious uses—"watch/read next…"—but not the only ones. When used on transcripts of audio or video media, annotation enables more sophisticated (and nonlinear) ways of consuming such media, like zooming in on specific moments and language in the timeline.

You have covered the case where the entity linkage is taken care of by the NER software. When that is not the case, some manual matching is required. Techniques for disambiguation include using textual analysis and graph analysis. One approach is annotating the concepts in the organizing principle with sets of keywords or even regular expressions to apply to the identifiers of the entities extracted by the NLP, as in Example 12-20.

Example 12-20. Annotate concepts with keywords and regular expressions

```
// keyword based
CREATE (c:Category { name: "Person", alts: ["Human","Pax", "Pers"]})

//regex based (inclusion/exclusion list)
CREATE (c:Category { name: "COVID-19", inc: ["covid.*","corona.*"],
  excl: ["sars.*"]})
```

One limitation of this approach is its strong reliance on the precision of such keywords or regexes. You would need to write some custom code to take care of the linkage, with logic similar to Example 12-21.

Example 12-21. Linking categories in Cypher

```
MATCH (e:Entity { name: $entityname })
MATCH (c:Category)
WHERE e.name IN c.alts
WITH e, collect(c) AS candidate_cats
WITH e, selection_logic(candidate_cats) AS selected_cat
MERGE (e)-[:references]->(selected_cat)
```

The `selection_logic` function abstracts the problem of selecting the most plausible candidate in case there are multiple matches. It can be as simple as picking one random one, or assigning weights based on the relevance in the context, which could be computed using domain-specific heuristics or structural features like distance, centrality, or others.

Case Study: NASA Lessons Learned

Semantic search knowledge graphs aren't science fiction—they're science fact. Dating all the way back to the Apollo moon missions in the 1960s and spanning more than half a century of collective knowledge, NASA's "lessons learned" is a treasure trove of knowledge gained from positive experiences, such as successful missions and tests, and negative experiences, as in failures and mishaps.

NASA's lessons learned information system (LLIS) contains hundreds of millions of documents, reports, and scientific research findings from multiple databases across the agency. Previously, it held less than 1% of the organization's 20 million documents. With a total of 80,000 employees, the volume, variety, and velocity of data were taxing the system. NASA needed a better way for end users to access this information.

In the early 2000s, the LLIS was significantly underutilized because it was not easy to navigate or search. The biggest challenge was accessing the data, which was in silos strewn across groups, departments, programs, and products. Nothing connected or even cross-referenced the data. Keyword search queries across all 20 million documents took a long time to run. Search results returned 1,100 documents. The user would then need to read those articles (or at least scan them) to get to an answer.

But there was hope. There is a great deal of metadata associated within the lessons learned, so NASA was able correlate the topics (by extracting entities) based on their self-assigned categories. They could see each lesson with its topic as well as correlations between topics, so they could also see how topics correlated to one another. This allowed users to look at trends, which can potentially help NASA engineers prevent disastrous outcomes.

Fast forward to today, and LLIS at NASA is now a rich knowledge graph. Search is faster and provides fewer—but much more relevant—results. The knowledge graph plays a fundamental role in two main ways.

First, it gathers data from various siloed sources (as per Chapter 5). It then links the data together via entities using the techniques in this chapter. The lessons learned database has already generated significant value. One NASA engineer said, "This has saved us at least a year and over $2M in research and development towards our Mission to Mars planning." If your semantic search knowledge graphs are as good, who knows how far they'll take you?

Summary

The ability to search for concepts rather than specifying index keys is revolutionary. Using a semantic search knowledge graph, this becomes possible. Not only does your organization keep more of its knowledge, but accessing that knowledge becomes straightforward through the knowledge graph.

From semantic search, you can go one level further and try to understand natural language. In the next chapter, you'll see how search and NLP go hand in hand, using real-world examples to demonstrate.

CHAPTER 13

Talking to Your Knowledge Graph

In the previous chapter, you saw NLP applied to the construction of knowledge graphs to support semantic search over collections of things (articles, products, documents, and so forth). This relied on an NLP task called *entity extraction*, or NER. But NER is only one of the ways in which NLP can interact with knowledge graphs, out of three broad categories:

1. Where knowledge graphs are populated with entities, facts and knowledge come from applying NLP techniques to natural language text. This includes what you learned in Chapter 12 as well as the cases where NLP is used for fact extraction to build question-answering knowledge graphs. This category is *natural language as input* to a knowledge graph.
2. You will see in this chapter how natural language can also be generated from knowledge graphs. This can be used to produce a conversational answer to a query or for automated report generation, as some examples. This category is *natural language as output*.
3. Somewhere in between, knowledge graphs can be a *tool providing structured context for NLP tasks*, either at the lexical or at the conceptual level (or both).

Figure 13-1 depicts the three categories.

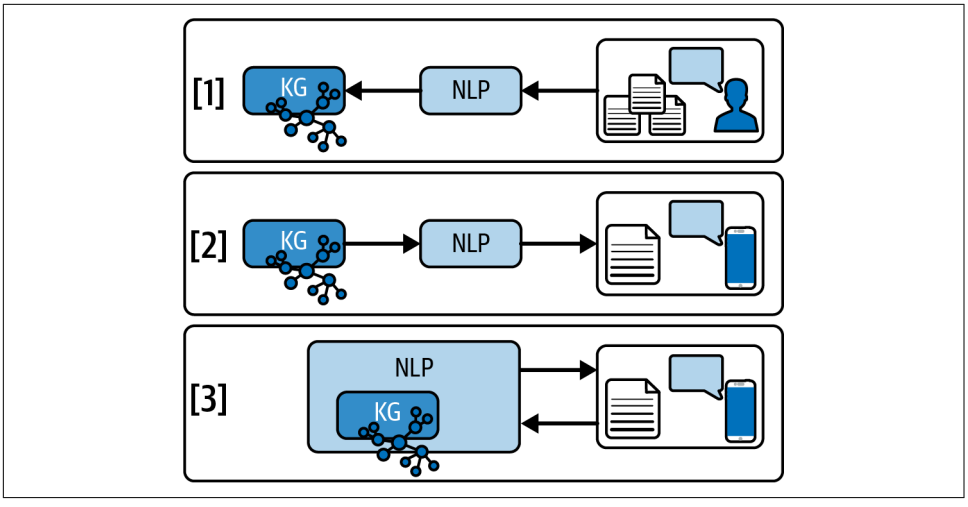


Figure 13-1. Types of interaction between natural language and knowledge graphs

This chapter is split into three parts. In the first part, you will learn how NER can be complemented with fact extraction to build a knowledge graph for answering questions as opposed to document search and recommendation. The second part explores techniques for implementing conversational interfaces to interact with the knowledge graph. This includes both processing natural language questions and translating them into structured queries on the graph, as well as generating text from the graph to produce the answers. The last section focuses on knowledge graphs for lexical databases where you need to build a custom entity extractor which has specialized domain knowledge. You will learn how a public lexical database (like WordNet) can be used to provide a detailed analysis of the use of such specialized knowledge graphs and the implementation of semantic similarity metrics in them.

Question Answering: Natural Language as a Source of Facts for a Knowledge Graph

The techniques described in the previous chapter were document centric. You applied NER and entity disambiguation to the items (products, articles, documents…) in a collection to annotate them and build a knowledge graph capable of doing effective semantic search over large sets of documents or computing semantic similarity between documents to drive recommendation and penalization.

In this section, you will explore the case when documents are just a source of facts that get interconnected in a knowledge graph with the objective of answering domain questions rather than carrying out sophisticated tasks on the documents they are

extracted from. In this type of scenario, you query the knowledge graph expecting an answer, not a list of documents where you will find that answer.

Note that this is not incompatible with keeping a reference to the source documents for provenance and trust purposes, but the ultimate objective is the creation of a knowledge graph that will not answer the question "Which documents talk about Tesla?" but rather "Who is the CEO of Tesla?" or "Where is Tesla's headquarters?"

The nature and complexity of the NLP used will vary and with it the richness of the graph created from the unstructured data. The role of the NLP moves from entity extraction to fact extraction. This effectively means that you will want to extract statements from the natural language text that will then be modeled in the graph as either node properties or relationships connecting nodes.

To show this in action, the example will use yet another NLP package: Diffbot. Diffbot's Natural Language API offers fact extraction in addition to entity extraction. Example 13-1 shows some sample code that takes the first sentence in one of the articles and extracts facts from it by invoking the Diffbot Natural Language API.

Example 13-1. Using Diffbot's Natural Language API to extract facts from text

```
import json
import requests

payload = {
    "content": "Pichette is a partner at Inovia Capital and chair of" \
               "commerce platform Lightspeed",
    "lang": "en"
    }
res = requests.post("https://{}/v1/?fields={}&token={}".format
  (HOST, "entities,facts", TOKEN), json=payload)

for ent in res.json()["entities"]:
  print(ent)
  //write to graph

for fact in res.json()["facts"]:
  print(fact)
  //write to graph
```

The extracted facts are structured as *entity-property-value* triplets. The real output is a rich JSON structure like the one shown in Example 13-2 that includes unique identifiers, cross references to public knowledge bases like Wikidata, and additional attributes like salience, confidence, sentiments, and others.

Example 13-2. Raw facts extracted with the Diffbot Natural Language API

```
{"sentiment": -0.32881057,

  "entities": [{"name": "Patrick Pichette", "confidence": 0.81,
                "salience": 0.64345313, "sentiment": 0.0, "isCustom": ... }],

  "facts": [ {
    "humanReadable": "[Patrick Pichette] employee or member of [Innovia Capital]",
    "entity": { "name": "Patrick Pichette",
        "diffbotUri": "https://diffbot.com/entity/EYwbLa__MNVGtPMryqqCCEA",
        "confidence": 0.9999268,
        "allUris": ["http://www.wikidata.org/entity/Q3369779"],
        "allTypes": [{"name": "person",
            "diffbotUri": "https://diffbot.com/entity/E4aFoJie0MN6dcs_yDRFwXQ",
            "dbpediaUri": "http://dbpedia.org/ontology/Person"}],
        "isCustom": False, "entityIndex": 5},
    "property": {"name": "employee or member of",
        "diffbotUri": "https://docs.diffbot.com/ontology#Person.memberOf"},
    "value": {"name": "Innovia Capital",
        "confidence": 0.9989188, "allUris": [],
         "allTypes": [{"name": "date",
            "diffbotUri": "https://diffbot.com/entity/EGTSOJhZ0NnqIjbjgQ8pyLg"}],
            "isCustom": False, "entityIndex": 0},
    "confidence": 0.9498123,
    "evidence": [{"passage": "Pichette is a partner at Inovia Capital",
                  "entityMentions": [{
                    "text": "Pichette", "beginOffset": 111, "endOffset": 113,
                    "isPronoun": False, "confidence": 0.9985091}],
                    "valueMentions": [{"text": "Innovia Capital",
                                       "beginOffset": 119, "endOffset": 132,
                                       "confidence": 0.9989188}]}]},
    {"humanReadable": ... }] ,
    ...
}
```

A visual and more intuitive view of the same output is presented in Figure 13-2.

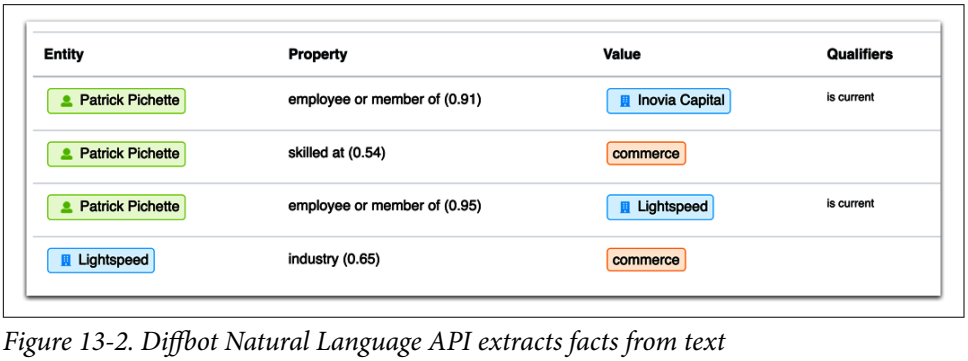


Figure 13-2. Diffbot Natural Language API extracts facts from text

That output can be directly consumed to populate the knowledge graph, as shown in Example 13-3.

Example 13-3. Importing facts from Diffbot into your knowledge graph

```
//code incomplete...
entity_list  = res.json()["entities"]
fact_list  = res.json()["facts"]

//call write transaction with params entities and facts and Cypher below

cypher= "
UNWIND $facts AS fact
MERGE (source:Entity { id: fact.entity.diffbotUri })
MERGE (target:Entity { id: fact.value.diffbotUri })
WITH source, target, fact
CALL apoc.create.relationship(source,fact.property.name,{},target)
//add salience etc
"
```

A knowledge graph of these characteristics can then be queried (see Example 13-4) to answer domain questions like "Which individuals are directly related to Inovia Capital and what's the nature of their relationship?"

Example 13-4. A Cypher query shows who is related to Inovia Capital

```
MATCH (:Organization { name : "Inovia Capital"})-[rel]-(p:Person)
RETURN p.name AS person, type(rel) AS rel_type
```

Running the Cypher query in Example 13-4 produces the expected results shown in Example 13-5.

Example 13-5. Results of a Cypher query showing who is related to Inovia Capital

```
╒═══════════════════╤═══════════════════════╕
│"person"           │"rel_type"             │
╞═══════════════════╪═══════════════════════╡
│"Patrick Pichette"│"EMPLOYEE_OR_MEMBER_OF"│
└───────────────────┴───────────────────────┘
```

The challenge with this approach is the harmonization of relationships. To avoid an undesired proliferation of relationships between entities, fact extraction tools need additional input about the types of relationships that are relevant and how to identify and disambiguate them.

Using Natural Language Query with a Knowledge Graph

You have seen how information rich a knowledge graph can be, and you have also seen that you can get value from it through both visual exploration with tools like Bloom and programmatic access with Cypher. While Bloom is more universal and only requires a basic understanding of the domain, Cypher is only really accessible to technical specialists. One way to widen the spectrum of potential users getting value from a knowledge graph is by enabling natural-language-based interfaces. You are no doubt used to interacting with personal assistants like Siri or Alexa using your voice or having conversations with chatbots when contacting the customer support of any of your service providers. While you are aware that you are exploring a knowledge base, being able to articulate your questions in natural language lowers the barrier for accessing the content. In the world of knowledge graphs, conversational interactions require the translation of natural language into a structured query to the graph.

In this section, you will see a motivating example to guide you through the process. The example shows how to implement a basic natural language interface for Neo4j's movie database. The NLP library used in this case will be spaCy, using its rule-based matching capability.

If you are familiar with regular expressions, spaCy's rule-based matcher engines take this approach to the next level. In addition to finding words and phrases, they give you access to the tokens within the document and their relationships. This means you can easily access and analyze the surrounding tokens, combine them, process them to make sense of the meaning of the natural language question, and ultimately be able to translate it into a structured query in Cypher.

Before getting into the details, you can initialize the knowledge graph with the movies dataset with a simple instruction on the Neo4j Browser, `:play movies`, and then click through to have the nodes and relationships created. The resulting graph contains two types of entities, `Person` and `Movie`, connected through relationships of type `ACTED_IN`, `DIRECTED`, `PRODUCED`, `WROTE`, and `REVIEWED`. Figure 13-3 shows a small excerpt of the resulting graph.

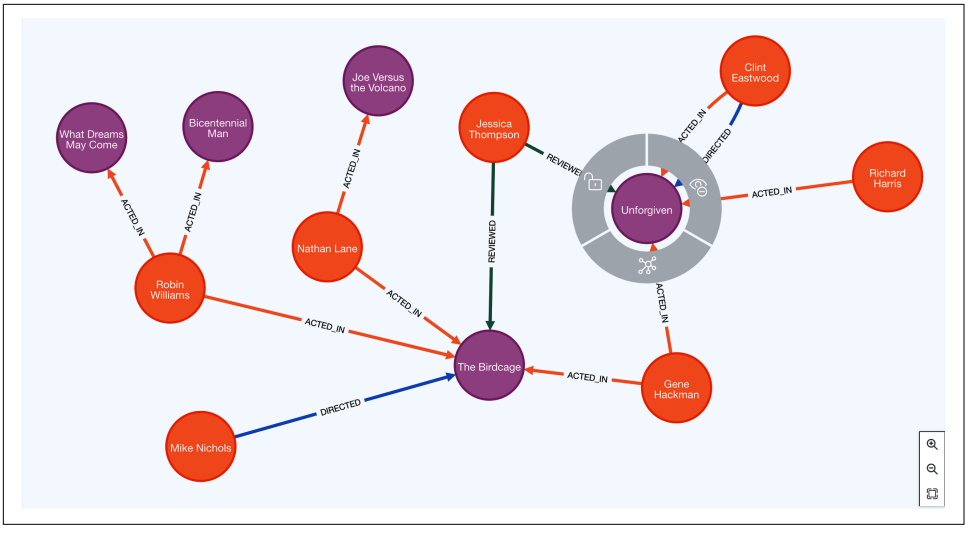


Figure 13-3. Excerpt of the movie dataset

The central element of the rule-based matching capability in spaCy is called a `Matcher`. A `Matcher` takes a configuration in the form of a rich pattern that describes the type of structure you want to detect in the natural language text received as input. Example 13-6 shows how to initialize and configure a `Matcher`.

Example 13-6. A `Matcher` capturing simple questions about movies (part 1)

```
import spacy
from spacy.matcher import Matcher

nlp = spacy.load('en_core_web_sm')

matcher = Matcher(vocab=nlp.vocab)

q1_pattern = [{"LOWER":"who"},
          {"LEMMA":
             {"IN": ["direct", "produce", "write", "review"]}, "POS": "VERB"},
          {'IS_ASCII': True, 'OP': '+'},
          {'IS_PUNCT': True, 'OP': '?'}]


matcher.add("q_1", patterns=[q1_pattern]) .          # (1)

doc = nlp("do you know who wrote a few good men?")  # (2)

result = matcher(doc, as_spans=True)                # (3)

print(result)
```

The sentence types that this pattern will capture are of the form "Who directed *The Da Vinci Code*?" or "Do you know who wrote *A Few Good Men*?" You can get all the details in the spaCy documentation, but you are defining a pattern with four components (tokens):

- The word *who* described in its lowercase form. This means that if the input uses any case variant of it (*Who*, *WHO*, …), the pattern will still work.
- A verb—that is, what the part of speech configuration specifies `"POS": "VERB"`--having as lemma any of the ones in the list: `direct`, `produce`, `write`, `review`. A *lemma* is the canonical form of a set of words. In this case, *direct* would be the lemma for *directs* or *directed*.
- A word, any alphanumeric word (`IS_ASCII`), with one important modifier: it can appear *one or more times*. That is what the optionality modifier `'OP': '+'` specifies.
- The pattern ends with an optional punctuation sign. This would correspond to the question mark when used. It has been made optional (again with the optionality modifier `'OP': '?'`) for the cases where the request is formulated in an assertive way rather than as a question: "Tell me who wrote X."

The pattern is added to the `Matcher` in (1), and when the sentence `do you know who wrote a few good men?` is passed to the `nlp` method in (2), it does match the pattern and will therefore be included in the results in (3). The result includes all the information you need to transform the detected meaning into a valid Cypher query. All you need is a way to map the different verbs that the second token in the pattern can match to the relevant schema elements in your graph model.

A simple way to do this is to build a dictionary with the correspondences. In this pattern, the verbs correspond to relationships in the graph so the code in Example 13-7 should as well.

Example 13-7. A `Matcher` capturing simple questions about movies (part 2)

```
# map tokens to graph schema elements
q1_verb_to_rel = {"direct" : "DIRECTED", "produce" : "PRODUCED",
                  "write" : "WROTE", "review" : "REVIEWED"}

max_match = result[-1]
verb = max_match[1].lemma_ # (1)
title = ' '.join([tk.text for tk in max_match[2:] if tk.pos_ != 'PUNCT']) # (2)
query_as_cypher = "MATCH (p:Person)-[:{rel_type}]->(m:Movie) " \
                  "WHERE toLower(m.title) CONTAINS '{movie_title}' " \
                  "WITH collect(p.name) AS answer_as_list " \
                  "RETURN CASE WHEN size(answer_as_list) > 0 THEN " \
                  "  substring(reduce" \
```

```
                "(result='', x in answer_as_list | result + ', ' + x),2) " \
                "  ELSE \"I cannot answer your question about " \
                "'{movie_title}' \" end AS answer " \
    .format(rel_type=q1_verb_to_rel[verb], movie_title=title.lower()) # (3)

print(query_as_cypher)
```

You can extract the relevant tokens from the `Span` (a `Span` is a sequence of tokens in spaCy terminology). In (1) the verb is extracted, followed by (2) where the collection of words that represent the movie title in the sentence are also extracted. With the results of (1) and (2), it's easy to build a query that encodes the semantics of the natural language received as input.

Ignoring the last four lines that take care of the formatting of the output, the query synthesized by this process is the one you would expect from visually inspecting the graph and is shown in Example 13-8.

Example 13-8. The Cypher query generated from a natural language fragment

```
MATCH (p:Person)-[:WROTE]->(m:Movie)
WHERE toLower(m.title) CONTAINS 'a few good men'
WITH collect(p.name) AS answer_as_list
RETURN CASE WHEN size(answer_as_list) > 0 THEN
        substring(reduce(result='', x in answer_as_list | result + ', ' + x),2)
        ELSE "I cannot answer your question about 'a few good men' "
    end as answer
```

You may be thinking that this is a quite scripted approach because you have to be explicit about all the patterns that you want your `Matcher` to be aware of. That is correct to some extent, but in the next section you will see how to fully automate this by annotating the organizing principle in your knowledge graph. To understand how this is possible, remember that the graph is a self-describing data structure. You can *ask* the graph the types of entities, properties, and relationships that it's aware of just by calling the `db.schema.*` procedures. Some simple annotation on the common natural language terms for the elements in the schema is all that's needed to dynamically create a rich natural language interface and have it evolve as the knowledge graph evolves.

To complete this section, Example 13-9 shows the entire code of a rule-based conversational interface with multiple different patterns interacting with a Neo4j graph database.

Example 13-9. A conversational interface

```
import spacy
from spacy.matcher import Matcher
from neo4j import GraphDatabase, basic_auth
```

```
driver = GraphDatabase.driver(
  "bolt://localhost:7687",
  auth=basic_auth("neo4j", "neo"))
session = driver.session(database="data") // 1


def query_db(session, cypher): // 2
    results = session.read_transaction(
        lambda tx: tx.run(cypher).data())
    return results[0]["answer"]


nlp = spacy.load('en_core_web_sm')

matcher = Matcher(vocab=nlp.vocab)

q1_pattern = [{"LOWER":"who"},
           {"LEMMA": {"IN": ["direct", "produce", "write", "review"]}, "POS": "VERB"},
           {'IS_ASCII': True, 'OP': '+'},
           {'IS_PUNCT': True, "OP":"?"}]

q1_verb_to_rel = {"direct" : "DIRECTED", "produce" : "PRODUCED",
    "write" : "WROTE", "review" : "REVIEWED"}
matcher.add("q_1", patterns=[q1_pattern]) // 3


q2_pattern = [{"LOWER": "when"},
           {"LEMMA": "be"},
           {'IS_ASCII': True, 'OP': '+'},
           {"LEMMA": {"IN": ["release","premiere"]}, "POS": "VERB", "OP":"?"},
           {"LEMMA": "out", "OP":"?"},
           {'IS_PUNCT': True, "OP":"?"}]

q2_word_to_prop = {"release" : "released",
                   "premiere" : "released", "out" : "released"}
matcher.add("q_2", patterns=[q2_pattern])) // 4

q3_pattern = [{"LOWER": "who"},
           {"LEMMA": {"IN": ["act", "perform", "appear", "be"]},
              "POS": {"IN": ["VERB", "AUX"]}},
           {"LEMMA": "in"},
           {'IS_ASCII': True, 'OP': '+'},
           {'IS_PUNCT': True, "OP":"?"}]

matcher.add("q_3", patterns=[q3_pattern])) // 5


def process_question(question): // 6
    doc = nlp(question)

    result = matcher(doc, as_spans=True)
```

```
    if (len(result) > 0):
        max_match = result[-1]
        pattern_detected = nlp.vocab[max_match.label].text // 7
        if (pattern_detected == 'q_1'):
            verb = max_match[1].lemma_
            title = ' '.join([tk.text for tk in max_match[2:] if tk.pos_ != 'PUNCT'])
            query_as_cypher = "MATCH (p:Person)-[:{rel_type}]->(m:Movie) " \
                              "WHERE toLower(m.title) CONTAINS '{movie_title}' " \
                              "WITH COLLECT(p.name) AS answer_as_list " \
                              "RETURN CASE WHEN size(answer_as_list) > 0 THEN " \
                              "  substring(reduce(result='', x in answer_as_list" \
                              " | result + ', ' + x),2) " \
                              "  ELSE \"I cannot answer your question about " \
                              "'{movie_title}' \" end AS answer " \
                .format(rel_type=q1_verb_to_rel[verb], movie_title=title.lower())

        elif (pattern_detected == 'q_2'):
            word_pos = -2 if max_match[-1].is_punct else -1
            word = max_match[word_pos].lemma_
            title = ' '.join([tk.text for tk in max_match[2:word_pos]])
            query_as_cypher = "MATCH (m:Movie) " \
                              "WHERE toLower(m.title) CONTAINS '{movie_title}' " \
                              "WITH collect(m.{movie_prop}) AS answer_as_list " \
                              "RETURN CASE WHEN size(answer_as_list) > 0 THEN " \
                              "  substring(reduce(result='', x in answer_as_list" \
                              " | result + ', ' + x),2) " \
                              "  ELSE \"I cannot answer your question about " \
                              "'{movie_title}' \" end AS answer " \
                .format(movie_prop=q2_word_to_prop[word], movie_title=title.lower())

        elif (pattern_detected == 'q_3'):
            title = ' '.join([tk.text for tk in max_match[3:-1]])
            query_as_cypher = "MATCH (p:Person)-[:ACTED_IN]->(m:Movie) " \
                              "WHERE toLower(m.title) CONTAINS '{movie_title}' " \
                              "WITH collect(p.name) AS answer_as_list " \
                              "RETURN CASE WHEN size(answer_as_list) > 0 THEN" \
                              " substring(reduce(result='', x in answer_as_list |" \
                              " result + ', ' + x),2) " \
                              "  ELSE \"I cannot answer your question about " \
                              "'{movie_title}' \" end AS answer " \
                .format(movie_title=title.lower())

        print("Q:", question)
        print("A:", query_db(session, query_as_cypher))
        print("Explain:", query_as_cypher[:90], "...\n")

questions = ["could you tell me who reviewed the Da Vinci code?",
             "who directed unforgiven ?",
             "do you happen to know who appears in Jerry Maguire?",
             "Who acts in Apollo 13?",
             "tell me when was the birdcage released",
```

```
            "do you know who produced the matrix reloaded?",
            "do you remember who was in cloud atlas?",
            "name the person who wrote When Harry Met Sally",
            "when was top gun out?"] // 8

for question in questions:
    process_question(question) // 9


driver.close()
```

Example 13-9 starts by initializing a driver and a session to talk to the knowledge graph in (1). In (2) a function `query_db` is defined to use the session created in (1) and run a read query. It takes only one parameter, the Cypher query itself, and returns a string with the result of running the query on the graph. In (3), (4), and (5) you register each pattern with a unique identifier for the type of queries that it describes. This will be useful in (7) to identify which of the patterns has been detected in the text and generate the right Cypher query by following the relevant branch in the `if` structure. The whole processing of the text has been bundled in a function called `process_question` defined in (6), which takes the natural language question and prints out three values: the original question in natural language, the answer to the question returned from running the Cypher translation against the movies knowledge graph, and the Cypher query itself as an explanation.

A few natural language sentences are defined in (8) and used as a test in (9), producing the results shown in Example 13-10. (Note that the queries have been truncated for brevity, but the final fragment is very similar to the one generated in Example 13-8 and can also be gleaned from the source code in Example 13-9.)

Example 13-10. Results of processing natural language questions on the movie database

```
Q: could you tell me who reviewed the Da Vinci code?
A: Jessica Thompson, James Thompson
Explain: MATCH (p:Person)-[:REVIEWED]->(m:Movie)
         WHERE toLower(m.title) CONTAINS 'the da vinci code ...

Q: who directed unforgiven ?
A: Clint Eastwood
Explain: MATCH (p:Person)-[:DIRECTED]->(m:Movie)
         WHERE toLower(m.title) CONTAINS 'unforgiven' WITH  ...

Q: do you happen to know who appears in Jerry Maguire?
A: Bonnie Hunt, Jay Mohr, Cuba Gooding Jr., Jonathan Lipnicki, ...
Explain: MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
         WHERE toLower(m.title) CONTAINS 'jerry maguire' wi ...

Q: Who acts in Apollo 13?
A: Tom Hanks, Ed Harris, Gary Sinise, Kevin Bacon, Bill Paxton
```

```
Explain: MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
         WHERE toLower(m.title) CONTAINS 'apollo 13' WITH c ...

Q: tell me when was the birdcage released
A: 1996
Explain: MATCH (m:Movie)
         WHERE toLower(m.title) CONTAINS 'the birdcage'
         WITH collect(m.released) AS ...

Q: do you know who produced the matrix reloaded?
A: Joel Silver
Explain: MATCH (p:Person)-[:PRODUCED]->(m:Movie)
         WHERE toLower(m.title) CONTAINS 'the matrix reload ...

Q: do you remember who was in cloud atlas?
A: Tom Hanks, Jim Broadbent, Halle Berry, Hugo Weaving
Explain: MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
         WHERE toLower(m.title) CONTAINS 'cloud a ...

Q: name the person who wrote When Harry Met Sally
A: Nora Ephron
Explain: MATCH (p:Person)-[:WROTE]->(m:Movie)
         WHERE toLower(m.title) CONTAINS 'when harry met sally ...

Q: when was top gun out?
A: 1986
Explain: MATCH (m:Movie)
         WHERE toLower(m.title) CONTAINS 'top gun'
         WITH collect(m.release...
```

Natural Language Generation from Knowledge Graphs

Often there's a need to generate natural language out of a knowledge graph. Examples of this can be generating a report in natural language from an investigative exercise in a graph (such as the Panama Papers investigation) or just the results of a query as the output of a conversational interface.

In this context, the self-describing nature of graph data gives a substantial advantage. Start with the basic (but still largely valid) modeling approach that transforms nouns in nodes in the graph and verbs in relationships. A good example is the already famous sentence "Dan loves Ann." When modeled as a graph, it becomes two nodes representing the two nouns *Dan* and *Ann* and one relationship connecting Dan to Ann representing the verb *to love*.

If you invert this logic and generalize it, you can produce a simple protosentence of the form "subject-predicate-object" by just concatenating the words identifying any pair of nodes in a graph, connected by a word identifying the relationship. An example of a minimal subgraph extracted from the Panama Papers database can be created with the script in Example 13-11.

Example 13-11. A simple fragment from the Panama Papers dataset

```
MERGE (x:Entity { name: "Euroyacht Limited" , jurisdiction: "BM" })
MERGE (y:Entity { name: "TUC LIMITED", jurisdiction: "MLT",
        incorporation_date: date("2013-10-07")})
MERGE (z:Entity {name: "GLOBAL TUITION & EDUCATION INSURANCE CORPORATION",
        jurisdiction: "BRB", incorporation_date: date("1998-04-03") })
MERGE (x)-[:OFFICER_OF]->(y)
MERGE (x)-[:OFFICER_OF]->(z)
```

Now you can write a completely generic query capturing any pair of nodes connected by a relationship (pattern `(x)-[r]->(y)`) and returning a string built by concatenating the three elements to form the basic sentence shown in Example 13-12.

Example 13-12. A completely generic query producing basic sentences from the graph structure and data

```
MATCH (x)-[r]->(y)
RETURN x.name + " " + toLower(replace(type(r),"_"," ")) + " " + y.name AS sentence
```

Notice that the query contains some basic formatting like adding a couple of whitespaces to separate words and replacing the underscores in names with whitespaces to make it even more natural. Nothing very sophisticated for this initial example. The query produces the results shown in Example 13-13, which are surprisingly close to correct English.

Example 13-13. Result of running a completely generic query producing a basic English sentence from the graph structure and data

```
╒═══════════════════════════════════════════════════════════════════════╕
│"sentence"                                                             │
╞═══════════════════════════════════════════════════════════════════════╡
│"Euroyacht Limited officer of GLOBAL TUITION & EDUCATION INSURANCE CORPORATION"│
├───────────────────────────────────────────────────────────────────────┤
│"Euroyacht Limited officer of TUC LIMITED"                             │
└───────────────────────────────────────────────────────────────────────┘
```

Given the fact that the query is totally generic in the sense that it does not contain any schema-specific elements, it would be valid for any graph. Taking this same query and running it on the movie database shipped with Neo4j (just run the `:play movies` command) would produce sentences like "Keanu Reeves acted in *Johnny Mnemonic*."

It's quite good value for two lines of code. But it does not stop here. The same concept can be applied to the properties of a node too, as shown in Example 13-14.

Example 13-14. Another completely generic query producing basic sentences from the graph structure and data

```
MATCH (n:Entity) UNWIND keys(n) AS property
RETURN n.name + "'s " + replace(property,"_"," ") + " is " + n[property] AS sentence
```

This property-oriented query produces results shown in Example 13-15 that are once again decent English fragments.

Example 13-15. Result of running a completely generic query producing a basic sentence from the graph structure and data

```
╒═══════════════════════════════════════════════════════════════════════════════════╕
│"sentence"                                                                         │
╞═══════════════════════════════════════════════════════════════════════════════════╡
│"Euroyacht Limited's jurisdiction is BM"                                           │
├───────────────────────────────────────────────────────────────────────────────────┤
│"TUC LIMITED's jurisdiction is MLT"                                                │
├───────────────────────────────────────────────────────────────────────────────────┤
│"TUC LIMITED's incorporation date is 20113-10-07"                                  │
├───────────────────────────────────────────────────────────────────────────────────┤
│"GLOBAL TUITION & EDUCATION INSURANCE CORPORATION's jurisdiction is BRB"           │
├───────────────────────────────────────────────────────────────────────────────────┤
│"GLOBAL TUITION & EDUCATION INSURANCE CORPORATION's incorporation date is          │
│ 1998-04-03"                                                                       │
└───────────────────────────────────────────────────────────────────────────────────┘
```

This is a great starting point that reflects practically how the structure captured by a knowledge graph can be directly read and used to generate natural language. This is empiric confirmation that graphs capture the understanding of a domain similar to humans. It's quite straightforward to read the information from a knowledge graph and express it in natural language.

But there are a couple of points that have been temporarily ignored in the previous examples and that are important in this process. If you paid close attention, you might have noticed that the previous queries were not completely generic; they did actually contain one schema element: the property `name`. Generating subject-predicate-object sentences requires having a way to name things (nodes). The previous queries assumed that there would always be a `name` property, which is not necessarily the case. For a generally valid solution, a way of referring to nodes would be required.

Another thing to keep in mind is that, depending on how relationships and properties are named (the graph's schema), the resulting sentences will be more or less readable. Think of the sentence that would result from a relationship called `sub_109`. A generally valid solution will want to detach itself from the actual schema element names. While it can be practical to have a relationship named `sub_109` to be able

to write compact Cypher queries, a natural-language-oriented annotation of the property would be required to make the two independent of each other.

Finally, the previous queries navigated outgoing relationships (from source to target). But sometimes exploring incoming relationships is required. Linguistically, that's a challenge since `Person - ACTED_IN - Movie` or `Entity - HAS_JURISDICTION - Jurisdiction` only reads correctly one way around. You solve this problem by providing additional annotation for relationships for navigation in both directions containing the reciprocal natural language: `MOVIE - FEATURED - Actor` for example. But a more general approach can be achieved through annotating the knowledge graph's ontology, which is explored next.

Annotating the Knowledge Graph's Organizing Principle to Drive Natural Language Generation

This section presents an automated approach for generating natural language from a property graph by creating and annotating an ontology with information that a general-purpose natural language generator uses to produce high-quality natural language from the data. It's assumed that such ontology will be stored along with the instance data in such a way that the natural language engine will have access to both, but it's important to understand that this approach could be generalized to follow alternative architectures: the ontology could be read directly from an external repository, stored in a separate graph in a fabric configuration, and so on.

The starting point is to create and store the ontology of the knowledge graph. An ontology can be formalized using W3C standards like RDF Schema or OWL and then imported into Neo4j using the neosemantics plugin or, alternatively, created as a set of nodes and relationships directly in Neo4j.

The ontology contains descriptions for the types of entities that will be described using natural language. This declarative description of the schema can also be seen as the configuration for a general-purpose natural language generation engine to work.

The examples use the movies database that contains nodes of type `Person` and `Movie`. The first requirement is to identify for each node type the property that will be used to refer to the node. In the case of nodes of type `Person`, it will be the property `name`, and for nodes of type `Movie`, it will be the property `title`. This can be formally described using OWL, as shown in Example 13-16.

Example 13-16. Ontology annotated with information on entity names

```
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix talk: <http://www.neo4j.org/2022/07/talkable#> .
```

```
@prefix mv: <http://www.neo4j.org/sch/movies#> .

    mv:Movie rdf:type owl:Class ;
        talk:name "title" ;
        rdfs:label "Movie" .

    mv:Person rdf:type owl:Class ;
        talk:name "name" ;
        rdfs:label "Person" .
```

This can be generated either manually or with any ontology modeling tool and then imported into Neo4j using the neosemantics plugin in one line, as shown in Example 13-17.

Example 13-17. Code to import an ontology into Neo4j

```
CALL n10s.onto.import.fetch("file containing the onto","Turtle")
```

If compliance with W3C standards is important to your project and/or reuse of existing ontologies, then the previous two steps will be extremely useful. If that does not apply to you, the same can be achieved with Cypher, as shown in Example 13-18.

Example 13-18. Creating an organizing principle with plain Cypher

```
CREATE (:Class { "name":"title","label":"Movie" })
```

These annotated class definitions help the engine determine how to refer to nodes when producing a natural language sentence. When talking about a `Movie`, you will use its `title`, and when talking about a `Person`, you will use its `name` property. The next thing is to describe and annotate relationships. Example 13-19 shows how.

Example 13-19. Ontology annotated with information on relationships

```
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix talk: <http://www.neo4j.org/2022/07/talkable#> .
@prefix mv: <http://www.neo4j.org/sch/movies#> .

mv:ACTED_IN rdf:type owl:ObjectProperty ;
      rdfs:domain mv:Person ;
      rdfs:range mv:Movie ;
      talk:direct "acted in"@default ,
                  "is in the cast of"@long ,
                  "worked in"@short ;
      talk:inverse "has $o in it"@default ,
                   "includes"@short ,
                   "includes $o in its cast"@long ;
```

```
      rdfs:label "ACTED_IN" .

mv:WROTE rdf:type owl:ObjectProperty ;
      rdfs:domain mv:Person ;
      rdfs:range mv:Movie ;
      talk:direct "is the author of"@default ,
                  "wrote"@short ,
                  "wrote the script of"@long ;
      talk:inverse "is authored by"@default ,
                  "is written by"@long ,
                  "is by"@short ;
      rdfs:label "WROTE" .
```

For each of the relationships in the knowledge graph that you'd like to express in natural language, you need to provide the source and target in the form of `domain` and `range` and annotate them with both a `direct` and `inverse` way of navigating it. What that means is that when traversing the relationship in the natural direction—in the case of `ACTED_IN`, that would be from the `Person` to the `Movie`—use one of the direct expressions: "person X is in the cast of movie Y." But when navigating the relationship in the inverse way (from `Movie` to `Person`), take one of the inverse expressions: "movie Y includes person X." Multiple versions of the natural language to be generated from the relationship are provided. In the example, there is a long and a short form, but you could have a formal and a casual one chosen, depending on context. Finally, parameters are used to add more flexibility to the natural language generated (noted with the `$` sign). Sometimes the basic pattern subject-predicate-object is too limiting to generate richer expressions. When you want to describe the `ACTED_IN` connection between Keanu Reeves and the movie *The Matrix* in natural language, for example, you express it with a sentence like "*The Matrix* has Keanu Reeves in it." The object (Keanu Reeves) is embedded in the sentence rather than appended at the end.

Finally, a similar definition for node properties is provided, as shown in Example 13-20.

Example 13-20. Ontology annotated with information on node properties

```
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix talk: <http://www.neo4j.org/2022/07/talkable#> .
@prefix mv: <http://www.neo4j.org/sch/movies#> .

mv:born rdf:type owl:DatatypeProperty ;
        rdfs:domain mv:Person ;
        talk:direct "The birth year of $s was"@long ,
                    "spawned in"@short ,
                    "was born in"@default ;
```

```
        rdfs:label "born" .

mv:released rdf:type owl:DatatypeProperty ;
            rdfs:domain mv:Movie ;
            talk:direct "premiered in"@short ,
                        "The release year of $s was"@long ,
                        "was released in"@default ;
            rdfs:label "released" .

mv:tagline rdf:type owl:DatatypeProperty ;
           rdfs:domain mv:Movie ;
           talk:direct  "'s tagline goes: '$o'"@default ,
                        "'s tagline is"@short ,
                        "the tagline for $s was"@long ;
           rdfs:label "tagline" .
```

With this type of description stored in the graph along with the data, the Cypher expression in Example 13-21 can dynamically generate a natural language description of any given node.

Example 13-21. A Cypher natural language generating engine driven by the ontology

```
MATCH (n)-[r]-(o) WHERE id(n) = $entity_id
MATCH (cn:Class)<-[:domain|range]-(op:ObjectProperty)-[:domain|range]->(co:Class)
WHERE type(r) IN op.label
      AND (op.direct IS NOT NULL OR op.inverse IS NOT NULL)
      AND [x in labels(n) where x <> "Resource"][0] IN cn.label
      AND [x in labels(o) where x <> "Resource"][0] IN co.label
WITH  n[cn.name[0]] AS subj ,
      n10s.rdf.getLangValue("default" ,
        op[case when startNode(r) = n THEN "direct" else "inverse" end]) AS pred ,
      substring(reduce(res="", x IN collect(o[co.name[0]]) | res +","+ x),1) AS obj
WITH  CASE WHEN pred CONTAINS '$s' THEN '' ELSE subj end AS subj ,
      replace(replace(pred,'$o',obj),'$s',subj) AS pred,
      CASE WHEN pred CONTAINS '$o' THEN '' ELSE obj end AS obj
RETURN subj + " " + pred + " " + obj AS sentence
```

Example 13-21 can produce competent sentences like those in Example 13-22.

Example 13-22. A Cypher natural language generating engine driven by the ontology

```
╒═══════════════════════════════════════════════════════════════════════════════╕
│"sentence"                                                                     │
╞═══════════════════════════════════════════════════════════════════════════════╡
│"Lilly Wachowski wrote V for Vendetta,Speed Racer"                             │
├───────────────────────────────────────────────────────────────────────────────┤
│"Laurence Fishburne worked in The Matrix Revolutions,The Matrix Reload│
│ed,The Matrix"                                                                 │
├───────────────────────────────────────────────────────────────────────────────┤
│"Hugo Weaving worked in Cloud Atlas,V for Vendetta,The Matrix Revoluti│
│ons,The Matrix Reloaded,The Matrix"                                            │
├───────────────────────────────────────────────────────────────────────────────┤
│"Lana Wachowski wrote V for Vendetta,Speed Racer"                              │
├───────────────────────────────────────────────────────────────────────────────┤
│"Emil Eifrem worked in The Matrix"                                             │
└───────────────────────────────────────────────────────────────────────────────┘
```

This approach offers a pure Cypher solution to the generation of natural language from a knowledge graph by using an ontology as a configuration artifact. For any given X, it returns a natural language description of "all the knowledge graph knows about X". While this can be useful for generating documentation and similar use cases, it's still quite limited.

More sophisticated approaches can include the use of lexical databases like WordNet or ML elements that can learn from common expressions and eventually reconfigure themselves instead of requiring manual sentence pattern definitions.

Example 13-23 shows a small fragment of a real-world example using the open source bot framework Rasa. Rasa includes "knowledge base actions" to leverage information from external databases and can be used to interact with a knowledge graph built on Neo4j.

Example 13-23. Connecting the Rasa open source bot framework to a Neo4j knowledge graph

```
from rasa_sdk.knowledge_base.storage import KnowledgeBase
from rasa_sdk.knowledge_base.actions import ActionQueryKnowledgeBase

class MyKnowledgeBaseAction(ActionQueryKnowledgeBase):
    def __init__(self):
        knowledge_base = KnowledgeBase("ref to neo4j")
        super().__init__(knowledge_base)
```

Working with Lexical Databases

WordNet is a lexical database of English. Nouns, verbs, adjectives, and adverbs are grouped into sets of cognitive synonyms (called *synsets*), each expressing a distinct concept. Synsets are interlinked by means of conceptual-semantic and lexical relations. The result is a network of elements that can be stored naturally in a graph.

Before exploring some of the possibilities for analyzing the WordNet database as a graph, it's worth understanding how to construct that graph. Use a public version of WordNet published by the Global WordNet Association. The latest release can be found in the WordNet GitHub repository.

One of the formats available for WordNet is RDF. This simplifies things significantly because you can automate importing RDF into Neo4j, as shown in Example 13-24.

Example 13-24. Importing RDF into your knowledge graph using Cypher

```
CREATE CONSTRAINT n10s_unique_uri ON (r:Resource)
  ASSERT r.uri IS UNIQUE

CALL n10s.graphconfig.init( {handleVocabUris : "IGNORE"});

CALL n10s.rdf.import.fetch(".../english-wordnet-2021.ttl.gz","Turtle");
```

Figure 13-4 shows a fragment of the structure (schema) of the graph, whose form is Lexical Entry–Lexical Sense–Lexical Concept.

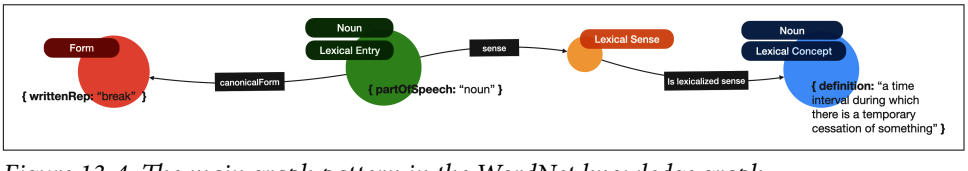


Figure 13-4. The main graph pattern in the WordNet knowledge graph

An explicit and detailed graph representation of a lexical database like WordNet has many different uses, but most importantly, it gives control over its content and makes it possible to adapt and extend it for specific uses. But first, it's interesting to try some basic queries over the graph that return meanings of words or reveal the nature of certain words, such as their precision or their ambiguity.

WordNet can be used to get all possible meanings of a word or a set of words. Most queries in this analysis will be based on a traversal of the main graph pattern Form–Lexical Entry–Lexical Sense–Lexical Concept, shown in Figure 13-4.

The Cypher query shown in Example 13-25 returns all the meanings for the word *clear*.

Example 13-25. A Cypher query that returns all the meanings for the word clear

```
MATCH (lemma:Form)<-[:canonicalForm]-(le:LexicalEntry)
  -[:sense]->()-[:isLexicalizedSenseOf]->(concept)
WHERE lemma.writtenRep = "clear"
RETURN le.partOfSpeech AS PoS, concept.definition AS definition
```

The results are numerous since *clear* is a word that can act as a noun, a verb, an adjective, or an adverb. The query returns the results in tabular form, and they match the ones you can get from querying WordNet through their public online search interface (see Figure 13-5).

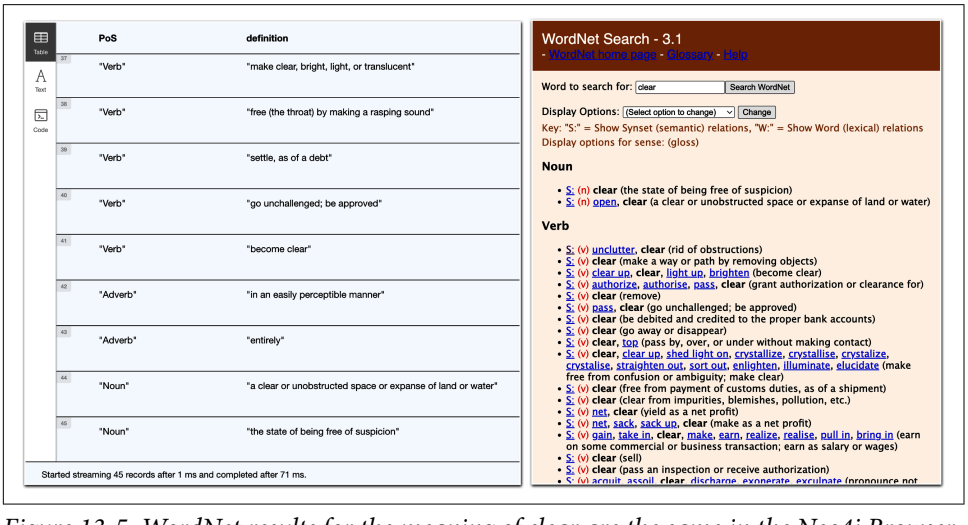


Figure 13-5. WordNet results for the meaning of clear are the same in the Neo4j Browser (left) and WordNet's web interface (right)

Example 13-26 shows how a slightly modified version of the query can be used to return a graph visualization of meanings (`LexicalConcepts`) of the word *clear*.

Example 13-26. A Cypher query that retrieves a graph visualization of meanings of the word clear

```
MATCH path = (lemma:Form)<-[:canonicalForm]-(:LexicalEntry)
  -[:sense]->()-[:isLexicalizedSenseOf]->()
WHERE lemma.writtenRep = "clear"
RETURN path
```

In Figure 13-6, you can see the lexical concepts playing the role of a verb, adjectives, adverbs, and nouns in different components of the graph.

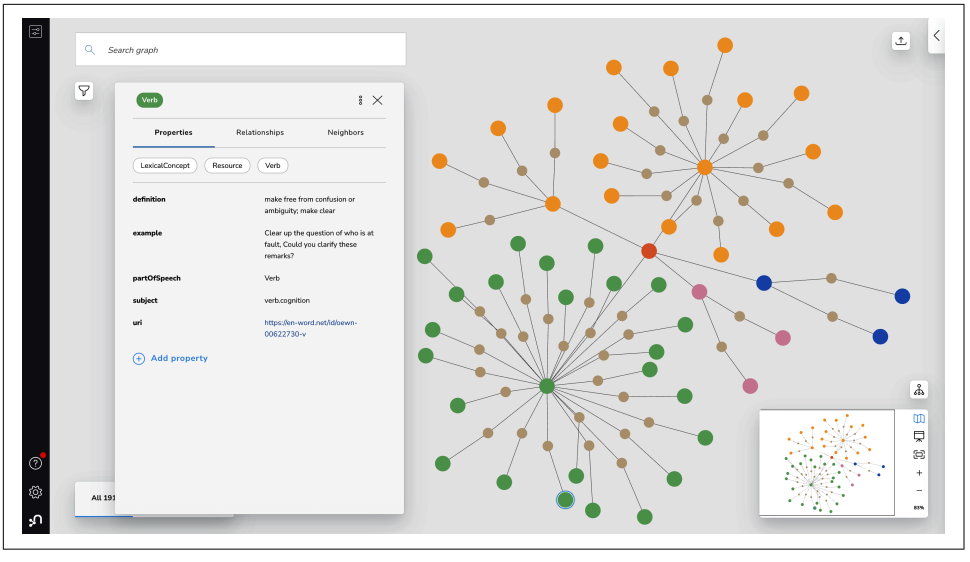


Figure 13-6. A graph visualization of the meanings of the word clear in Neo4j Bloom, with verbs, adjectives, adverbs, and nouns

Aggregations can be used to find the most *polysemic* words (words with multiple meanings) in English. It's just about counting the number of Lexical Entry–Lexical Sense–Lexical Concept patterns coming out of a given word or set of words (WordNet uses the term *lemma*, or a canonical or dictionary form for this). The query in Example 13-27 produces a list of the five most polysemic words.

Example 13-27. A Cypher query to find the top five words with the most meanings

```
MATCH (lemma:Form)
RETURN lemma.writtenRep AS lemma,
       size((lemma)<-[:canonicalForm]-(:LexicalEntry)-[:sense]->())
       AS senseCount
ORDER BY senseCount DESC LIMIT 5
```

The top five polysemic words in the current version of WordNet (2021) are shown in Table 13-1.

Table 13-1. Top five words with the most meanings

lemma	senseCount
break	75
cut	70
run	57
play	52
make	51

An inverse lookup is more interesting. With WordNet in a knowledge graph, you can find the opposite pattern by traversing the same graph pattern in the opposite direction, starting from a concept and navigating out to find all the words in the English language that can be used to express it. Figure 13-7 shows the three ways of expressing the concept of *being unsuccessful*.

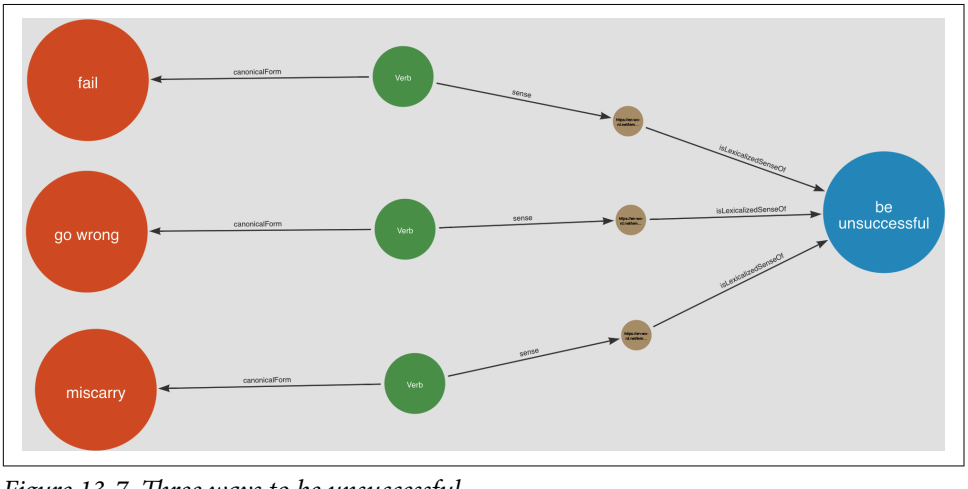


Figure 13-7. Three ways to be unsuccessful

But which concepts have the most ways to express them? The query in Example 13-28 performs the traversals, aggregates the results, and sorts them by lemmas per lexical concept.

Example 13-28. A Cypher query that finds concepts that have many words to express them

```
MATCH (lemma:Form)<-[:canonicalForm]-(:LexicalEntry)-[:sense]->
  (s)-[:isLexicalizedSenseOf]->(con:LexicalConcept)
RETURN con.definition AS concept, count(lemma.writtenRep)
  AS wordCount, collect(lemma.writtenRep) AS words
ORDER BY wordCount DESC LIMIT 10
```

The results are quite interesting, should you choose to run that code snippet. Now you can dive into some of the more advanced features of WordNet.

Our WordNet graph is enriched with semantic relationships between lexical concepts. The list of relationships is quite extensive. You will find *hypernymy* (being more general than), *hyponymy* (being more specific than), *meronymy* (being part of), and many more. The complete list can be explored using the `CALL db.relationshipTypes()` method. The first two (hyper/hypo) form a taxonomy in the graph. You can navigate the graph to expand the analysis of a concept and find more specific (or alternatively, more generic) meanings for it. At the lemma level, this can be used to find more generic or more specific words.

The path in Figure 13-8 shows that *sphere* is a more general term for *globe* for the specific meaning *a sphere on which a map (especially of the earth) is represented.*

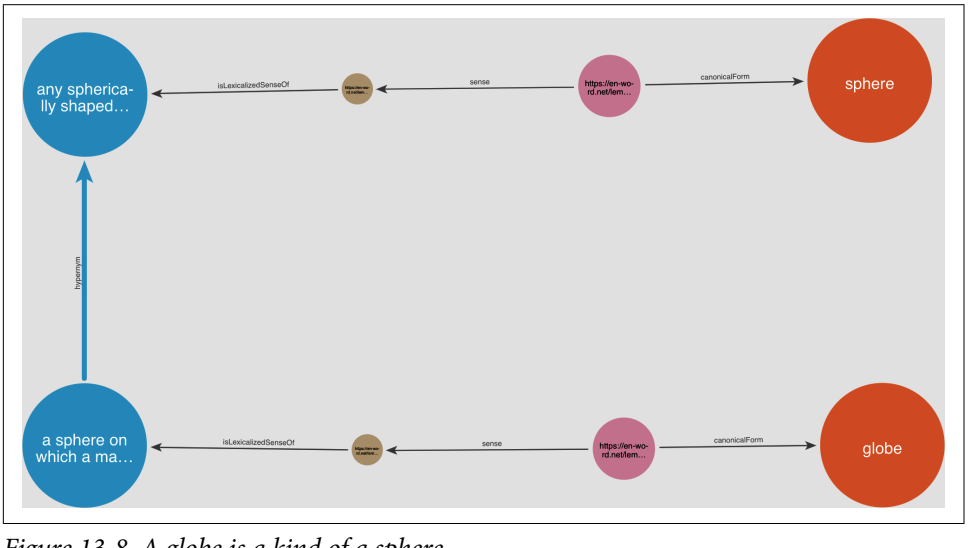


Figure 13-8. A globe is a kind of a sphere

On a more practical level, the exploration of these taxonomies has been used to define semantic similarity metrics between lexical concepts.

Graph-Based Semantic Similarity

Some of the most popular NLP libraries implement a number of standard semantic similarity metrics. These metrics turn out to be based on the exploration of paths in the lexical knowledge graph. You will explore three of the most commonly used ones and reproduce them using Cypher. You will also see how being able to customize the graph by adding terms and concepts specific to your domain can help you to use these metrics in multiple scenarios.

Path Similarity

The path similarity metric is based on the shortest path that connects the two lexical concepts in the taxonomy. Figure 13-9 shows how it is calculated.

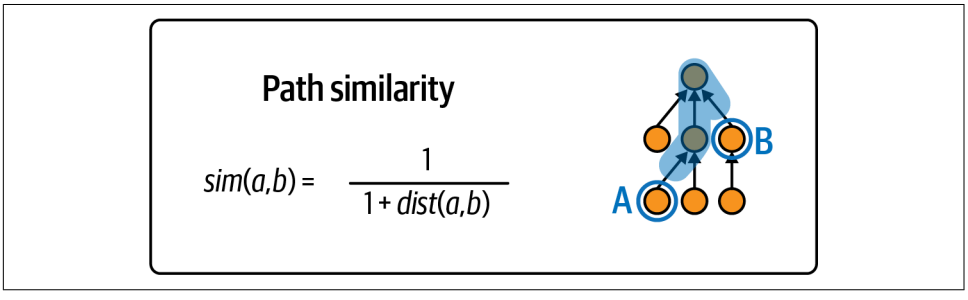


Figure 13-9. How path similarity is calculated

You can test it by comparing the similarity between the concepts denoted by the words *dog* and *lion*. The graph representation in Figure 13-10 shows the `hypernym` taxonomy between the concepts denoted by *dog* and *lion*, and you can see that the path is of length 5.

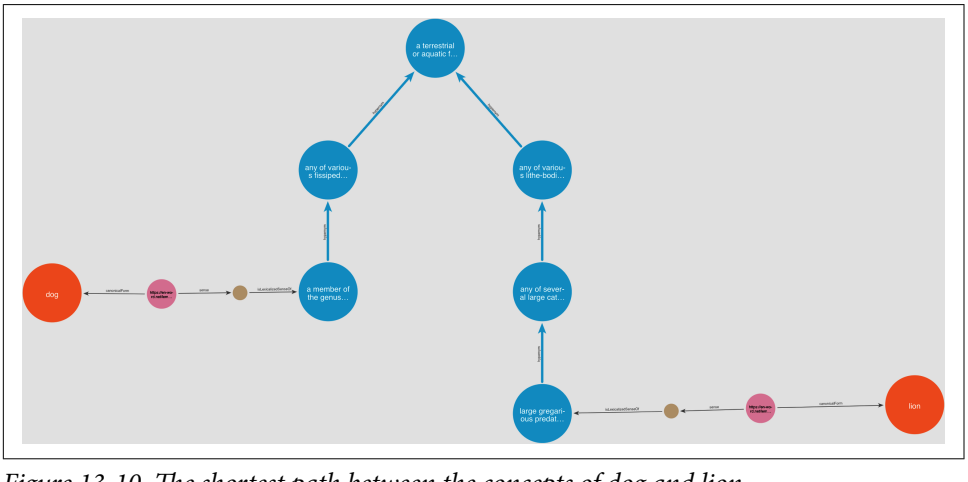


Figure 13-10. The shortest path between the concepts of dog and lion

The Cypher implementation of path similarity is shown in Example 13-29.

Example 13-29. A Cypher implementation of path similarity

```
MATCH (a:LexicalConcept {uri: $a_id})
MATCH (b:LexicalConcept {uri: $b_id })
MATCH p = shortestPath((a)-[:hypernym*0..]-(b))
WITH a, b, length(p) AS pathLen
```

```
RETURN a.definition AS a_def, b.definition
  AS b_def, pathLen, 1.0/(1+pathLen) AS pathSim
```

When run for params `{a_id: "https://en-word.net/id/oewn-02086723-n", b_id: "https://en-word.net/id/oewn-02131817-n" }` (the unique identifiers of the lexical concepts denoted by *dog* and *lion*, respectively), the Cypher query returns the results in Table 13-2.

Table 13-2. Dog and lion concepts, path length, and similarity

a_def	b_def	pathLen	pathSim
"a member of the genus Canis (probably descended from the common wolf) that has been domesticated by man since prehistoric times; occurs in many breeds"	"large gregarious predatory feline of Africa and India having a tawny coat with a shaggy mane in the male"	5	0.16666666666666666

The same results can be achieved using some popular NLP libraries. In this section, the Natural Language Toolkit for Python, more commonly known as NLTK, is used. The Python code snippet in Example 13-30 calls the `path_similarity` method in a synset to return that metric.

Example 13-30. Calling the path similarity method in Python via NLTK

```
from nltk.corpus import wordnet as wn

dog = wn.synset('dog.n.01')
lion = wn.synset('lion.n.01')
print(dog.definition())
print(lion.definition())
print(dog.path_similarity(lion))
```

That code produces exactly the same results you got with the Cypher query, shown in Example 13-31.

Example 13-31. The results for path similarity from calling NLTK are the same as those from Cypher

```
a member of the genus Canis (probably descended from the common wolf) that has been
domesticated by man since prehistoric times; occurs in many breeds

large gregarious predatory feline of Africa and India having a tawny coat with a
shaggy mane in the male

0.16666666666666666
```

Why use Cypher instead of NLTK? There are several advantages to being able to control the implementation of the similarity algorithm:

- There is no dependency on the version of WordNet currently supported by NLTK. At this writing, NLTK supports WordNet 3.0 only, which is years behind the current version. Though in the example the results from the graph and those from NLTK are identical, it is easy to find cases where they are different.
- New lexical terms and associated lexical concepts can be added to the graph to provide domain-specific detail in areas not covered by WordNet. All that's needed is to extend the graph by adding the relevant nodes and the relationships representing these entities.
- The previous point taken to the extreme means it's possible to apply these metrics to any other taxonomic organization not necessarily based on WordNet.
- The algorithm itself can be modified and adapted if desired.

Some of these advantages will be evident on analysis of the metrics in the following sections.

Leacock-Chodorow Similarity

The Leacock-Chodorow similarity metric adds one extra element to the computation, and that is the depth of the taxonomy. You can easily compute the depth of a taxonomy as the greatest length of the path between any element and the root element. The way to get the depth of a taxonomy in Cypher is by using variable-length path expressions, as shown in Example 13-32.

Example 13-32. A Cypher query that finds the depth of the taxonomy

```
MATCH path = (leaf:LexicalConcept)-[:hypernym*0..]->(root)
WHERE NOT EXISTS (()-[:hypernym]->(leaf))
  AND NOT EXISTS ((root)-[:hypernym]->())
RETURN max(length(path)) AS maxTaxonomyDepth
```

The query returns the value `19` for this graph, the global maximum.

There is not always a single root category that all categories are connected directly or indirectly to. There are graphs (WordNet is one of them) where there are multiple root elements or, in other words, multiple disconnected taxonomies. In those cases, you need to compute the depth of the taxonomy containing the elements being compared.

The Leacock-Chodorow similarity metric is defined by the formula in Figure 13-11.

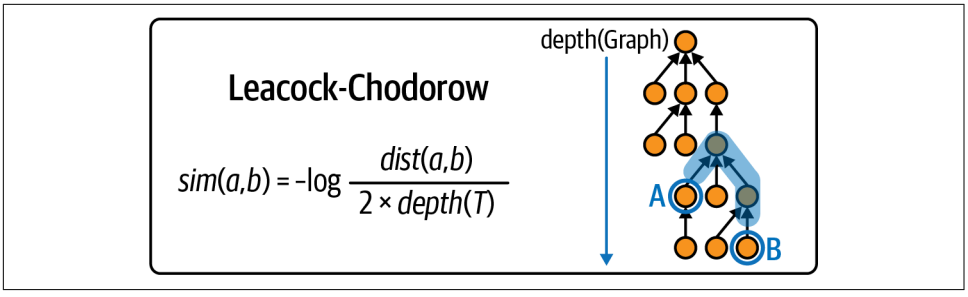


Figure 13-11. The Leacock-Chodorow distance formula finds the depth of the taxonomy

This metric boosts similarities that happen in deeper hierarchies. And as in the previous case, the metric can be implemented using Cypher over your WordNet knowledge graph, as shown in Example 13-33.

Example 13-33. A Cypher implementation of the Leacock distance formula

```
MATCH (a:LexicalConcept {uri: $a_id})
MATCH (b:LexicalConcept {uri: $b_id })
MATCH p = shortestPath((a)-[:hypernym*0..]-(b))
WITH a, b, length(p) AS pathLen
RETURN a.definition AS a_def, b.definition AS b_def,
  pathLen, -log10(pathLen/(2.0*$depth)) AS LCSim
```

Again, using Cypher produces results similar to the ones obtained by using the NLTK version, shown in Example 13-34.

Example 13-34. Cypher and NLTK produce similar results

```
from nltk.corpus import wordnet as wn

dog = wn.synset('dog.n.01')
lion = wn.synset('lion.n.01')
print(dog.definition())
print(lion.definition())
print(wn.lch_similarity(dog, lion))
```

Wu and Palmer Similarity

The Wu and Palmer similarity metric is based on the notion of the least common subsumer (LCS). The LCS is the most specific ancestor node of the two elements that are being compared.

The metric combines the depths of the two elements being compared and that of their LCS as defined by the formula in Figure 13-12.

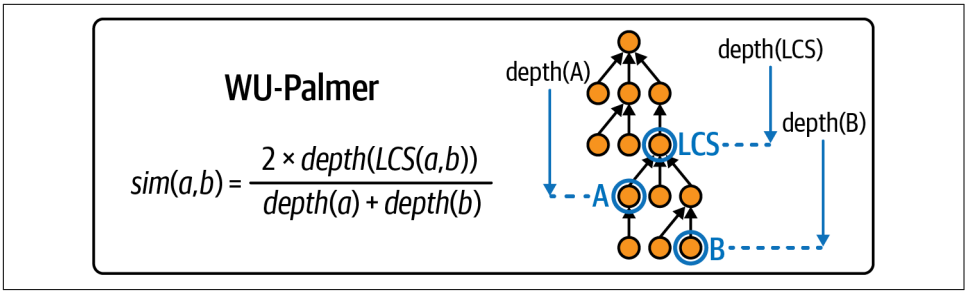


Figure 13-12. The Wu-Palmer distance formula

To explain the calculation of the metric with the example you have been using in the previous sections, you need to provide additional context on the position in the overall WordNet taxonomy of the lexical concepts denoted by *dog* and *lion*. The diagram in Figure 13-13 shows the key elements for the Wu and Palmer metric.

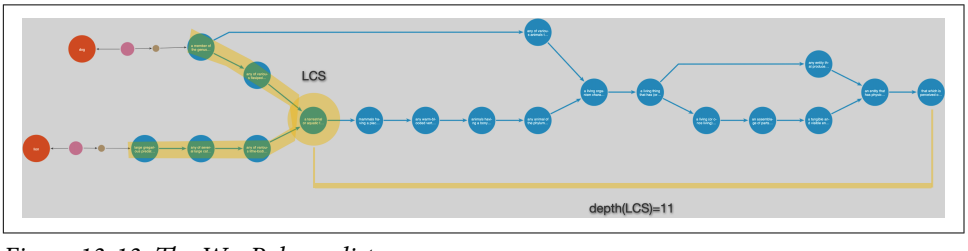


Figure 13-13. The Wu-Palmer distance

To get the LCS with Cypher, you need to draw two variable-length patterns converging on the LCS, as shown in Example 13-35.

Example 13-35. Cypher implementation of the Wu-Palmer distance formula

```
MATCH (a:LexicalConcept {uri: $a_id})
MATCH (b:LexicalConcept {uri: $b_id })
MATCH p = (a)-[:hypernym*0..]->(lcs)<-[:hypernym*0..]-(b)
WITH a, b, lcs, length(p) AS pathLen
MATCH p = (lcs)-[:hypernym*0..]->(root)
  WHERE NOT (root)-[:hypernym]->()
RETURN lcs.definition AS lcs_def, lcs.uri AS lcs_id,
  length(p) AS lcs_depth
ORDER BY pathLen LIMIT 1
```

Table 13-3 shows the results.

Table 13-3. Using the Cypher implementation of Wu-Palmer distance formula to determine the LCS

lcs_def	lcs_id	lcs_depth
a terrestrial or aquatic flesh-eating mammal	*https://en-word.net/id/oewn-02077948-n*	11

Once you have the LCS, calculating the Wu and Palmer similarity metric is straightforward, as shown in Example 13-36.

Example 13-36. Calculating the Wu and Palmer similarity metric in Cypher

```
MATCH (a:LexicalConcept {uri: $a_id})
MATCH (b:LexicalConcept {uri: $b_id })
MATCH (lcs:LexicalConcept {uri: $lcs_id })
MATCH a_to_lcs = (a)-[:hypernym*0..]->(lcs)
MATCH b_to_lcs = (b)-[:hypernym*0..]->(lcs)
WITH a, b, lcs, length(a_to_lcs) AS depth_a,
  length(b_to_lcs) AS depth_b
RETURN (2.0 * $lcs_depth) / (2.0 * $lcs_depth + depth_a + depth_b)
  AS wp_sim
```

As in the previous metrics, we'll show how to achieve the same using NLTK directly. It's not very different from what you saw for Leacock-Chodorow, just a different function name, as shown in Example 13-37.

Example 13-37. Calculating the Wu and Palmer similarity metric in Python

```
from nltk.corpus import wordnet as wn

dog = wn.synset('dog.n.01')
lion = wn.synset('lion.n.01')
print(dog.definition())
print(lion.definition())
print(wn.wup_similarity(dog, lion))
```

One of the advantages of storing the lexical database in a knowledge graph is that it introduces transparency in an otherwise black box solution. This is reflected in the fact that the knowledge graph is a dynamic entity that allows for the addition of new elements.

What is the process for adding a new entity to the WordNet knowledge graph in Neo4j? Well, first you need to *construct* the pattern Form–Lexical Entry–Lexical Sense–Lexical Concept. This requires the concepts of *proprietary software* and *open source software* as hyponyms of *software*.

In Example 13-38, the set of `CREATE` statements in Cypher adds the nodes and relationships representing the lemmas (`Form` and `LexicalEntry`) that identify the

sets of words *proprietary software* and *closed-source software*, both denoting the same meaning (`LexicalConcept`) with the following definition: *computer software released under a license restricting use, study, or redistribution*. To complete the pattern defined in WordNet, you must also include the intermediate `LexicalSense`.

Example 13-38. Adding lexical concepts to the knowledge graph in Cypher

```
CREATE (lc:LexicalConcept:Noun
    { subject:"noun.communication",
      partOfSpeech:"Noun",
      definition:"computer software released under a license
      restricting use, study or redistribution",
      uri:"https://custom.extension/id/15349-n",
      example:["the use of proprietary software is not allowed in our organization"]
    })
CREATE (ls1:LexicalSense
  {uri: "https://custom.extension/lemma/prop-soft#15349"})
CREATE (lc)<-[:isLexicalizedSenseOf]-(ls1)
CREATE (le1:LexicalEntry:Noun
    { canonicalForm: "proprietary software",
      partOfSpeech: "Noun",
      uri: "https://custom.extension/lemma/prop-soft#prop-soft-n"
    })
CREATE (ls1)<-[:sense]-(le1)
CREATE (f1:Form
    { writtenRep: "proprietary software",
      uri: "https://custom.extension/lemma/prop-soft"
    })
CREATE (le1)-[:canonicalForm]->(f1)

CREATE (ls2:LexicalSense {uri: "https://custom.extension/lemma/closed-soft#15348"})
CREATE (lc)<-[:isLexicalizedSenseOf]-(ls2)
CREATE (le2:LexicalEntry:Noun
    { canonicalForm: "closed-source software",
      partOfSpeech: "Noun",
      uri: "https://custom.extension/lemma/closed-soft#closed-soft-n"
    })
CREATE (ls2)<-[:sense]-(le2)
CREATE (f2:Form
    { writtenRep: "closed-source software",
      uri: "https://custom.extension/lemma/closed-soft"
    })
CREATE (le2)-[:canonicalForm]->(f2)
```

A similar script (excluded for brevity) would introduce the sets of words *OSS* and *open source software* denoting the same meaning (`LexicalConcept`) with the definition: *software whose source code is available under an open source license*.

Once the two new lexical concepts have been introduced, they need to be hooked to the existing hierarchy in WordNet with `hypernym` and the corresponding symmetric `hyponym` relationships, as shown in Example 13-39.

Example 13-39. Hooking added concepts to the existing hierarchy in Cypher

```
MATCH (prop:LexicalConcept { uri:"https://custom.extension/id/15349-n"})
MATCH (oss:LexicalConcept { uri:"https://custom.extension/id/15350-n"})
MATCH (sw:LexicalConcept { uri:"https://en-word.net/id/oewn-06578068-n"})
MERGE (prop)-[:hypernym]->(sw)-[:hyponym]->(prop)
MERGE (oss)-[:hypernym]->(sw)-[:hyponym]->(oss)
```

The subgraph created by the previous scripts can be visually explored with the query in Example 13-40, which yields a graph like that shown in Figure 13-14.

Example 13-40. Query to explore the new lexical subgraph

```
MATCH p = (lc:LexicalConcept {uri:"https://en-word.net/id/oewn-06578068-n"})
    <-[:hypernym]-()<-[:isLexicalizedSenseOf]-(ls)
    <-[:sense]-(le)-[:canonicalForm]->(f)
RETURN p
```

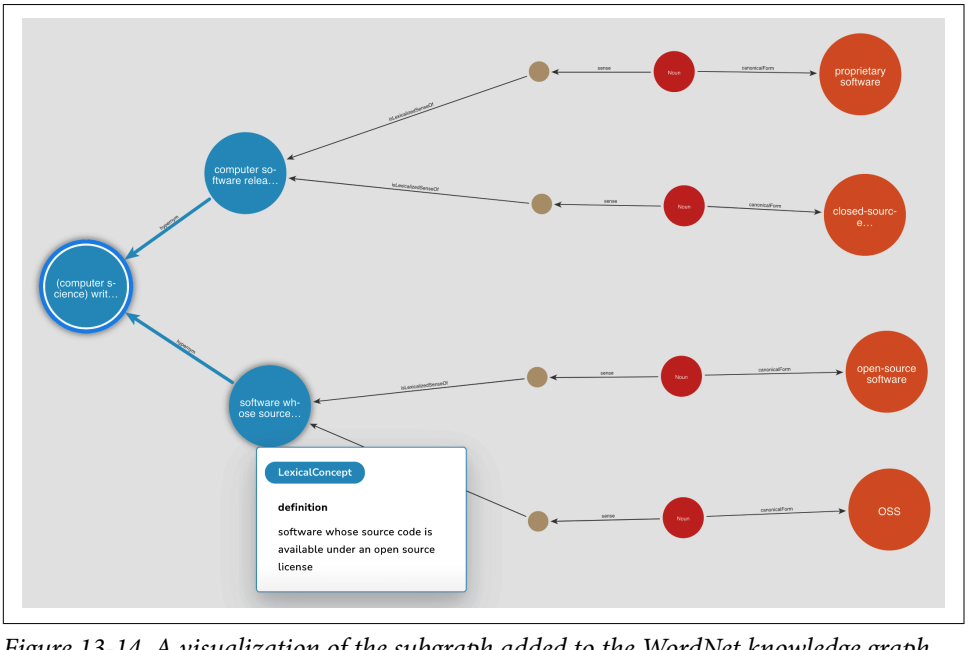


Figure 13-14. A visualization of the subgraph added to the WordNet knowledge graph

The extended graph is ready for semantic similarity computations, and the semantic similarity metrics discussed in this section can now be computed seamlessly (with exactly the same queries) on the newly created entities, producing the results in Table 13-4.

Table 13-4. Similarity metrics for the concepts added to the WordNet knowledge graph

concept A	concept B	Path sim	Leacock-Chodorov sim	Wu and Palmer sim
software whose source code is available under an open source license	*computer software released under a license restricting use, study, or redistribution*	0.4	0.846	0.973

This idea of modifying the graph can be generalized and the similarity metrics can be applied to a completely different knowledge graph as long as it contains some kind of hierarchical organization. This is the case with large knowledge graphs like Wikidata or DBPedia that have their entities classified in taxonomies using the `rdf:SubClassOf` (DBPedia) and `wd:P279` (Wikidata) relationships, respectively.

You have seen that knowledge graphs import any kind of textual information, including any existing relationships, as with the many semantic relationships defined in WordNet. With your text in a knowledge graph, you can perform all the operations available in other tools using Cypher. You can add to your knowledge graph, adding domain-specific terms as needed. It's a pattern often seen in the real world. For example, the German Center for Diabetes Research (DZD) has incorporated clinical trial data, its own research data, and the entirety of the PubMed database of published medical research into its knowledge graph.

Summary

In this chapter, you've seen a collection of powerful techniques and patterns for powering conversational systems with knowledge graphs. You've seen how specialist third-party tools and interoperability formats can be used to create compelling systems. Moreover, you've seen how straightforward building systems can be with commonplace graph database technology. You've reached an impressive level of understanding.

But that begs the question, "What's next for knowledge graphs?" Chapter 14 makes some predictions about the future of knowledge graphs, based on everything you've learned in this book.

CHAPTER 14

From Knowledge Graphs to Knowledge Lakes

Data is one of a modern enterprise's most valuable assets. With an organizing principle in hand, you can build a knowledge graph that drives significant business value for many use cases. This final chapter examines potential future directions for knowledge graphs. You'll see how knowledge graphs may move down the enterprise stack into a more foundational role and how in that role their scope will broaden. This intriguing pattern is known as a *knowledge lake*, and it might be one possible future for knowledge graph technology.

Conventional Knowledge Graph Deployments

In the last few years, the deployment of knowledge graphs has proliferated, mirroring the advances in modern graph technology. Knowledge graphs for individual use cases or departments are becoming commonplace. Indeed, knowledge graphs for significant cross-functional business activities (e.g., compliance) or entities (e.g., customers) are now becoming increasingly common. At each step, as the scope of the knowledge graph grows, the infrastructure to support it moves further down the stack, as shown in Figure 14-1.

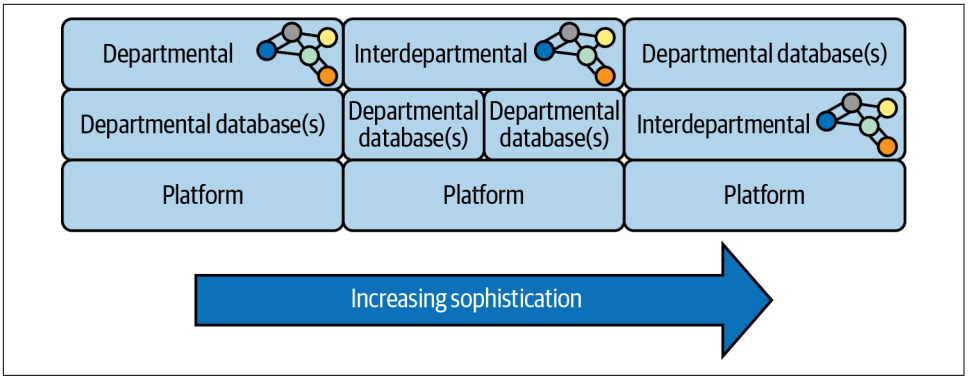


Figure 14-1. Knowledge graph proliferation to date

The high-level architecture shown in Figure 14-1 is uncontroversial. Some knowledge graphs have narrow use, and some have broader use. Each drives some business value.

The knowledge graph starts as a departmental asset fueled by that department's data. But as your needs become more sophisticated, those graphs may be interlocked via ontologies and middleware to create a knowledge graph that spans data from multiple departments.

At the most sophisticated level, the knowledge graph not only spans data from multiple departments but becomes a foundational technology layer for enterprise-wide reuse. Using a large knowledge graph system as a foundation conveniently allows other knowledge-intensive systems to be built atop its curated data. This is a success insofar as knowledge graphs have been broadly adopted and have themselves become a foundation for other useful systems.

But the question arises: what happens when the proliferation of knowledge graphs becomes a victim of its own success? How does an enterprise manage with many knowledge graphs without them becoming an ungovernable mess? Does the enterprise architecture devolve into a system of recursive knowledge graphs all the way down? Hopefully not.

From Knowledge Graphs to Knowledge Lakes

The prospect of more and better-quality data is intriguing. It implies a bigger and broader role for knowledge graphs and, consequently, an increasingly foundational role for them. As knowledge graphs become more fundamental for businesses, the deployment architecture changes. You move from (potentially many) knowledge graphs for individual use cases and specific cross-functional activities into a general-purpose approach. This general-purpose approach is known as a *knowledge lake* whose architecture is shown in Figure 14-2.

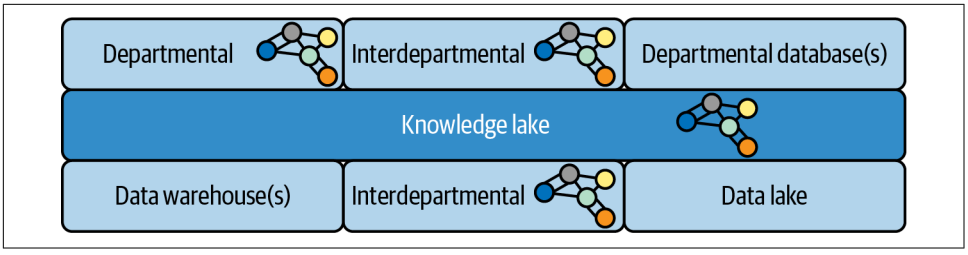


Figure 14-2. A knowledge lake is a general-purpose contextualized information system at scale

A knowledge lake is an architecture and usage pattern that is itself a knowledge graph and can subsume other knowledge graphs and nongraph data. It sits right next to an enterprise's data warehouses and data lakes. It stores a curated subset of the organization's data, represented as a graph, which can then be served up to multiple projects inside that enterprise. It is effectively a collection of knowledge graphs designed for reuse.

It's important to understand that a knowledge lake isn't a replacement for a data lake or (to a lesser extent) a data warehouse. Data lakes have the property that data can be poured into them at high volume, on the basis that meaning can be established later. Volume and throughput are key characteristics.

A knowledge lake brings meaning to this data after the fact. It makes the huge volumes of data in the lake or warehouse meaningful so that it has value to the business. It is therefore the knowledge lake that becomes the primary lens through which business-meaningful data is consumed. In effect, you get the best of both worlds.

Throughout this book, you've seen how knowledge graphs help businesses harness connected data to drive business value. You've also seen how this can be done without high-cost, high-risk, rip-and-replace technical projects. Instead, you now understand that knowledge graphs can be deployed as a nondisruptive technology sitting alongside existing systems while simultaneously enhancing their utility and value.

Despite their scope, knowledge lakes are no different. While the end goal might be to provide contextualized understanding and linkage with broad coverage across the enterprise, the starting point can be as small as a single system. Like any graph, your knowledge lake can grow over time and accrete more data and use cases. While it needs governance and automation to ensure that it doesn't become stale or patchy, the end game is a knowledge lake that can map out the entirety of the enterprise's data, so it can be discovered, used (and reused), and curated.

Looking to the Future

> We are drowning in information but starved for knowledge.
>
> —John Naisbitt, *Megatrends*

Naisbitt is right. Our businesses are not short of data. The challenge is to understand that data and put it to good use.

Knowledge graphs and, in the future, knowledge lakes, represent a significant opportunity to understand data in context. The patterns and use cases you've seen in this book provide the foundation you need to tackle this monumental challenge. Today, you have the ability to create systems of immense business value. Tomorrow, you may well be able to tame the complexity of the most challenging data-intensive systems and extract business value from them as easily as querying a single database. There's an exciting journey ahead!

Index

D

E

F

G

H

I

J

K

L

M

Q

R

S

T

U

V

W

About the Authors

Dr. Jesús Barrasa an expert in semantic technologies and graph databases, is head of the solutions architecture team in EMEA at Neo4j and leads the development of neosemantics (a Neo4j plugin for RDF). He cowrote *Knowledge Graphs: Data in Context for Responsive Businesses* (O'Reilly) and is cohost of the *Going Meta* live webcast.

Dr. Jim Webber is chief scientist at Neo4j, where he works on fault-tolerant graph databases. He coauthored *Graph Databases for Dummies* (Wiley) and *Graph Databases* and *Knowledge Graphs: Data in Context for Responsive Businesses*, both for O'Reilly. He's also a visiting professor at Newcastle University, UK.

Colophon

The animal on the cover of *Building Knowledge Graphs* is a sable antelope. It is a large antelope native to the wooded savannas stretching from Kenya to South Africa. Females and juveniles usually range in color from chestnut to dark brown, while males turn dark brown to black as they mature. They have white underparts and mostly white faces, with dark stripes on the nose and the sides of the snout. Both sexes have a dark mane and ringed, backward-curving horns. These horns can reach up to 3 feet long in females and over 5 feet long in males.

Sable antelopes live in herds of up to 20 indivuduals in a matriarchal structure, with one adult male per herd. Juvenile males are forced to leave the herd at about three years of age, and they congregate in bachelor groups of up to about a dozen individuals until they can take the position of the adult male in a herd.

The sable antelope is considered a species of least concern, though one subspecies, the giant sable antelope, is critically endangered. Sable antelope numbers are declining due to deforestation, development, and trophy hunting. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on an antique line engraving from *The Museum of Natural History*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.