3 Graphs in the database

This chapter covers

- Introducing graph databases with a focus on Neo4j
- Understanding the property graph data model
- Using the Cypher query language to create and query data in Neo4j
- Using client drivers for Neo4j, specifically the JavaScript Node.js driver

Fundamentally, a graph database is a software tool that allows the user to model, store, and query data as a graph. Working with a graph at the database level is often more intuitive for modeling complex connected data and can be more performant when working with complex queries that require traversing many connected entities.

In this chapter we begin the process of creating a property graph data model using the business requirements from the previous chapter and show how to model that in Neo4j. Then we compare that to the GraphQL model created in the previous chapter. We then explore the Cypher query language, focusing on how to write Cypher queries to address the requirements of our application. Along the way, we show how to install Neo4j, use Neo4j Desktop to create new Neo4j projects locally, and use Neo4j Browser to query Neo4j, and visualize the results. Finally, we show how to use the Neo4j JavaScript client driver to create a simple Node.js application that queries Neo4j.

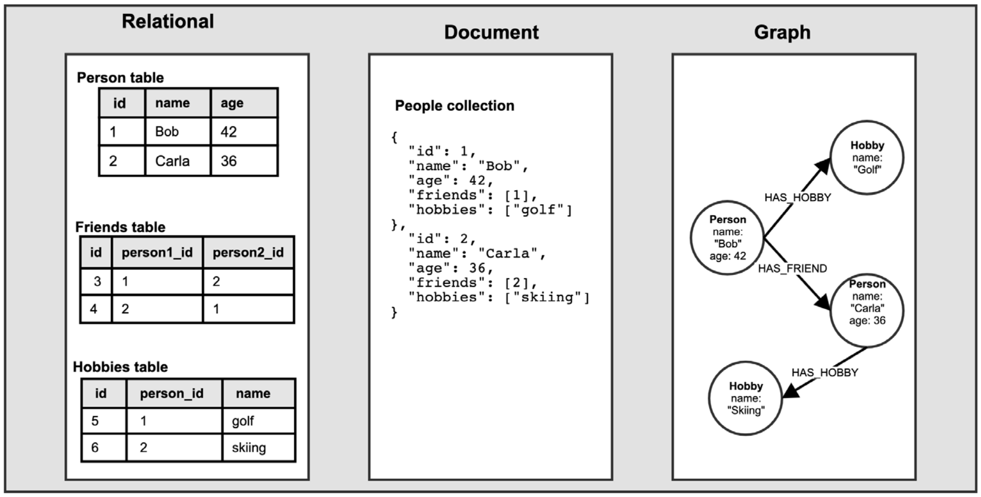
3.1 Neo4j overview

Neo4j is a native graph database that uses the property graph model for modeling data and the Cypher query language for interacting with the database. Neo4j is a transactional database with full ACID guarantees and can also be used for graph analytics. Graph databases such as Neo4j are optimized for working with highly connected data and queries that traverse the graph (think of the equivalent of multiple JOINs in a relational database) and are therefore the perfect backend for GraphQL APIs that describe connected data and often result in complex, nested queries. Neo4j is open-source and can be downloaded from https://neo4j.com/download.

We'll use Neo4j Desktop and Neo4j Browser in this chapter as we learn how to create and query data in Neo4j, but first let's dig into the property graph model used by Neo4j and see how it relates to the model used to describe GraphQL APIs we reviewed in the previous chapter.

3.2 Graph data modeling with Neo4j

Unlike other databases that use tables or documents to model data, graph databases such as Neo4j model, store, and allow the user to query data as a graph. In a graph, nodes are the entities and relationships connect them (see figure 3.1). In a relational database, we represent relationships with foreign keys and join tables. In a document database, we reference other entities using ids or even denormalizing and embedding other entities in a single document.



Person table

id	name	age
1	Bob	42
2	Carla	36

Friends table

id	person1_id	person2_id
3	1	2
4	2	1

Hobbies table

id	person_id	name
5	1	golf
6	2	skiing

People collection

```
{
  "id": 1,
  "name": "Bob",
  "age": 42,
  "friends": [1],
  "hobbies": ["golf"]
},
  "id": 2,
  "name": "Carla",
  "age": 36,
  "friends": [2],
  "hobbies": ["skiing"]
}
```

Figure 3.1 Comparing datamodels across relational, document, and graph.

The first step when working with a database is to determine the datamodel that will be used. In our case our datamodel will be driven from the business requirements we've defined in the previous chapter, working with businesses, users, and reviews. Review the requirements listed in the first section of the previous chapter for a refresher. Let's take those requirements and our knowledge of the domain to create a "whiteboard model" (figure 3.2).

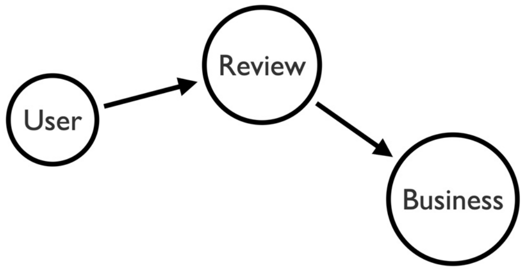


Figure 3.2 Building the Property Graph model: whiteboard model.

Whiteboard model

We'll use the term "whiteboard model" to refer to the diagram typically created when first reasoning about a domain, which is often a graph of entities and how they relate, drawn on a whiteboard.

How do we translate this mental model from the "whiteboard" model to the physical data model used by the database? In other systems, this might involve creating an ER diagram or defining the schema of the database. Neo4j is said to be "schema optional". While we can create database constraints to enforce data integrity, such as property uniqueness, we can also use Neo4j without these constraints or a schema. But the first step is to define a model using the Property Graph data model, which is the model used by Neo4j and other graph databases. Let's convert our simple whiteboard model in figure 3.2 into a property graph model that we can use in the database.

3.2.1 *The Property Graph model*

We gave a brief overview of the property graph data model in chapter 1. Here we'll go through the process of taking our whiteboard model and converting it to a property graph model used by the database.

The Property Graph datamodel

The property graph model is composed of:

- *Node Labels*: Nodes are the entities or objects in our data model. Nodes can have one or more labels that describe how nodes are grouped (think type or class).
- *Relationships*: Relationships connect two nodes and have a single type and direction.
- *Properties*: Arbitrary key-value pair attributes, stored on either nodes or relationships.

NODE LABELS

Nodes represent the objects in our whiteboard model. Each node can have one or more labels, which is a way of grouping nodes. Adding node labels (figure 3.3) to a whiteboard model is usually a simple process because a grouping will already have been defined during the whiteboard process. Here, we formalize the descriptors used to refer to our nodes into node labels (later we'll add node aliases and multiple labels so we use a colon as a separator to indicate the label).

Graph data model diagramming tools

There are many tools available for diagramming graph data models. Throughout this book we use the Arrows tool, a simple web-based application that allows for creating graph data models. Arrows is available online at http://www.apcjones.com/arrows/.

The Arrows user interface is minimal and is designed around creating property graph data models:

- Create new nodes with (+ Node).
- Drag relationships out of the halo of a node, either to an empty space for a new node or centered over an existing one to connect them.
- Double-click nodes and relationships to edit them and set names and properties (in a `key: "value" syntax).
- You can show the Markdown and also replace it with a previously saved fragment (useful for adding to documentation or checking into version control as your model evolves).
- You can export to SVG or take a screenshot.

For more on using Arrows, you can run the command `:play arrows` in Neo4j Browser to load a Neo4j Browser Guide showing how to use arrows or watch this quick video: bit.ly/33Nv9Mq.

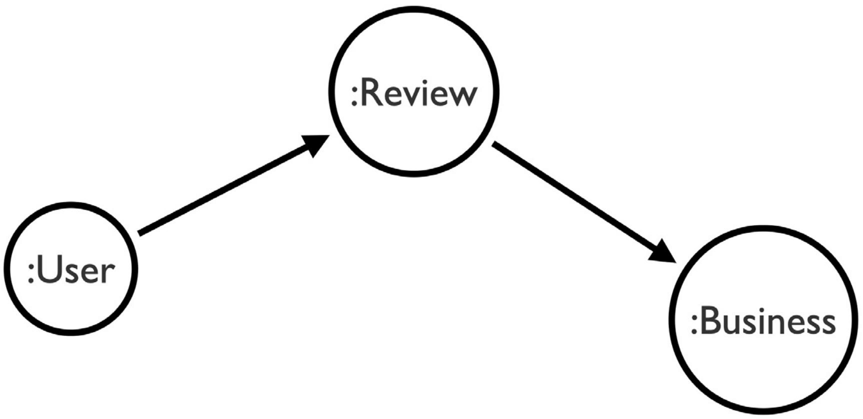


Figure 3.3 Building the Property Graph model: node labels.

The convention used for casing node labels is `PascalCase`. See the openCypher style guide for more examples of naming convention. Nodes can have multiple labels and allow us to represent type hierarchies, roles in different contexts, or even multi-tenancy.

Later on, we'll see how to use multiple node labels with GraphQL abstract types such as interface and union types.

Relationships

Once we've identified our nodes labels, the next step is to identify the relationships in our data model. Relationships have a single type and direction but can be queried in either direction (figure 3.4).

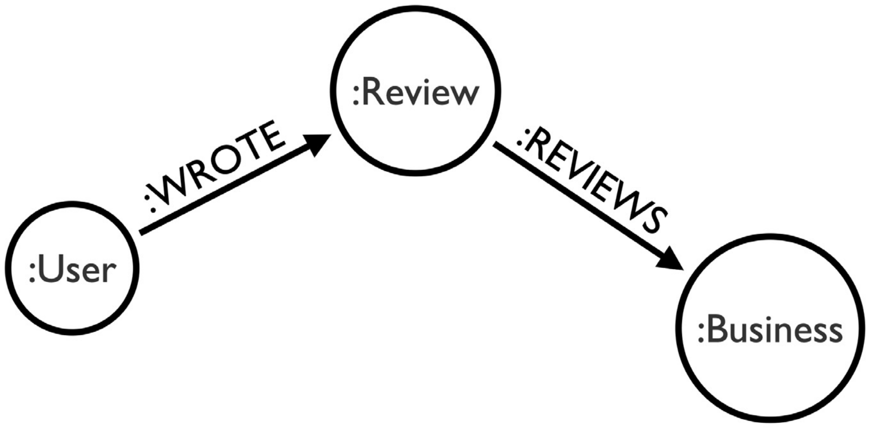


Figure 3.4 **Building the Property Graph model: relationship types.**

Dealing with undirected relationships

While every relationship has a single direction, we can treat the relationship as undirected at query time by not specifying a direction.

A good guideline for naming relationships is that the traversal from a node along a relationship to another node should read as a somewhat comprehensible sentence (figure 3.5). For example, "User wrote review" and "Review reviews business". You can read more about best practices for naming and conventions in the Cypher Style guide: https://neo4j.com/developer/cypher-style-guide

Properties

Properties are arbitrary key-value pairs stored on nodes and relationships. These are the attributes or fields of entities in our datamodel. Here we store `userId` and `name` as string properties on the User node, as well as other relevant properties on the Review and Business nodes.

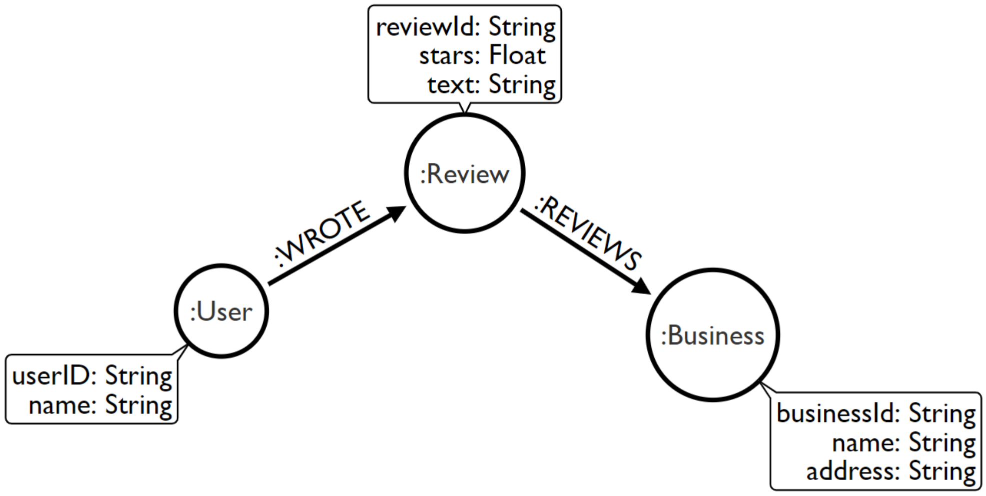


Figure 3.5 Building the Property Graph model: properties.

Property types

The following property types are supported by Neo4j:

- String
- Float
- Long
- Date
- DateTime
- LocalDateTime
- Time
- Point
- Lists of the above types

3.2.2 Database constraints and indexes

Now that we've defined our data model, how do we use it in the database? As mentioned earlier, unlike other databases that require us to define a complete schema before inserting data, Neo4j is said to be schema optional and doesn't require the use of a pre-defined schema. We can define database constraints that ensure the data adheres to the rules of the domain. We can create uniqueness constraints that ensure property values are unique across a node label (for example, guaranteeing that no two users have a duplicate id property value), property existence constraints (ensuring that a set of properties exist when a node or relationship is created or modified), and node key constraints that are similar to a composite key.

Database constraints are backed by indexes, which can be created separately as well. In a graph database indexes are used to find the starting point for a traversal, not to traverse the graph. We'll cover database constraints and indexes in more detail in the following section introducing Cypher.

3.3 Data modeling considerations

Graph data modeling can be an iterative process. In general, this is the process followed:

1. What are the entities? How are they grouped? These become nodes and node labels.
2. How are these entities connected? These become relationships.
3. What are the attributes of the nodes and relationships? These become properties.
4. Can you identify the graph traversal that answers your questions? These become Cypher queries. If not, iterate on the graph model.

However, there are often nuances not covered by this general approach. In the following sections, we address several common graph data modeling questions.

3.3.1 Node vs. property

Sometimes it's difficult to determine if a value should be modeled as a node or as a property on the node. A good guideline to follow here is to ask yourself the question, "Can I discover something useful by traversing through it if it was a node?" If the answer is yes, then it should be modeled as a node; if not, then treat it as a property. For example, consider if we were to add the category of business to our model. Finding businesses with overlapping categories is potentially useful and easier to discover if category is modeled as a node.

3.3.2 Node vs. relationship

In the case where we have a piece of data that seemingly connects two nodes (such as a review of a business, written by a user), should we model this data as a node or as a relationship? At first glance it seems as if we might want to create a `REVIEWS` relationship connecting the user and business, storing the review information, such as `stars` and `text` as relationship properties. However, we might want to extract data from the review, such as keywords mentioned through some natural language processing technique and connect that extracted data to the review. Or perhaps we want to use the review nodes as the starting point for a traversal query? These are two examples of why we may want to choose to model this data as an intermediate node instead of a relationship.

3.3.3 Indexes

Indexes are used in graph databases to find the starting point of a traversal, and not during the actual traversal. This is an important performance characteristic of graph databases such as Neo4j known as index-free adjacency. Only create indexes for properties that will be used to find the starting point of a traversal, such as a username or business id.

3.3.4 Specificity of relationship types

Relationship types are a way of grouping relationships and should convey just enough information to make it clear how two nodes are connected without being overly specific. For example, `REVIEWS` is a good relationship type connecting `Review` and `Business` nodes, `REVIEW_WRITTEN_BY_BOB_FOR_PIZZA` is an overly specific relationship type, the name of the user and restaurant are stored elsewhere and don't need to be duplicated in the relationship type.

3.3.5 Choosing a relationship direction

All relationships in the property graph model have a single direction, but can be queried in either direction, or queried without consideration of direction. There's no need to create duplicate relationships to model bidirectionality. In general, you should choose relationship directions that allow for a consistent reading of the data model.

3.4 Tooling: Neo4j Desktop

Now that we understand the property graph data model and have defined a simple version of the model we'll use for our business reviews application, let's create a Neo4j database and start executing Cypher queries. To do this we'll use Neo4j Desktop, which is the mission control center for Neo4j. In Neo4j Desktop (figure 3.6), we can create projects and instances of Neo4j. We can start, stop, and configure Neo4j database instances in Neo4j Desktop, as well as install optional database plugins, such as GraphQL Algorithms, and APOC (a standard library of database procedures for

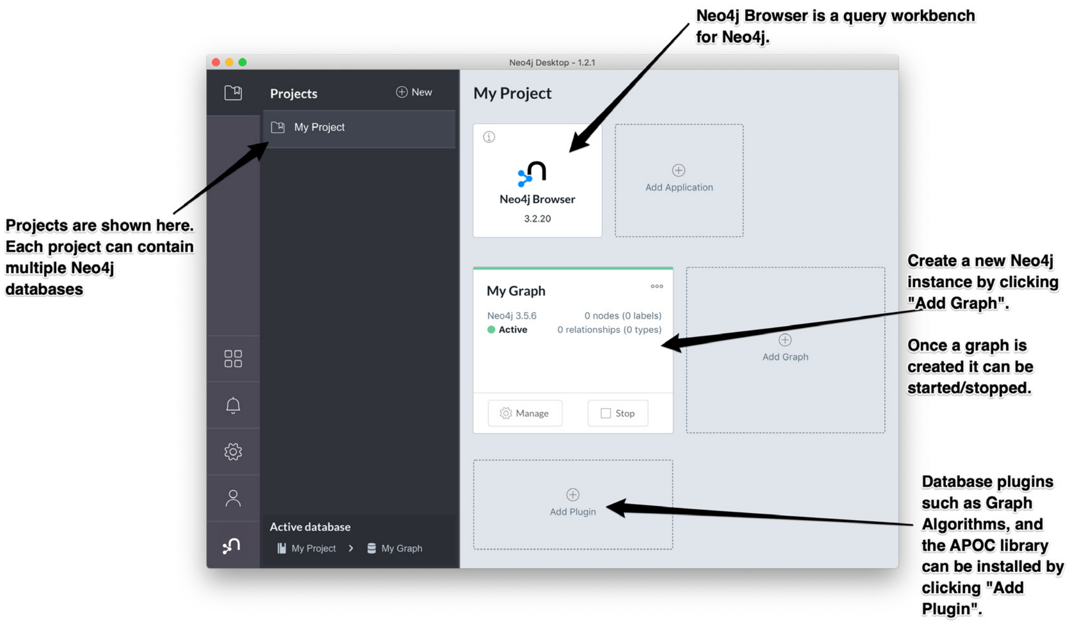

Neo4j Browser is a query workbench for Neo4j.
Neo4j Desktop - 1.2.1
Projects
New
My Project
My Project
Neo4j Browser
3.2.20
Add Application
Projects are shown here. Each project can contain multiple Neo4j databases
My Graph
Neo4j 3.5.6
Active
0 nodes (0 labels)
0 relationships (0 types)
Add Graph
Create a new Neo4j instance by clicking "Add Graph".
Once a graph is created it can be started/stopped.
Manage
Stop
Add Plugin
Active database
My Project
My Graph
Database plugins such as Graph Algorithms, and the APOC library can be installed by clicking "Add Plugin".

Figure 3.6 Neo4j Desktop: "Mission Control" for Neo4j.