aggregation across each group, but there's no `GROUP BY` operator in Cypher. Instead, with Cypher there's an *implicit* group by when returning the results of an aggregation function along with non-aggregated results. For example, to compute the average rating of each business:

```
MATCH (b:Business)<-[:REVIEWS]-(r:Review)
RETURN b.name, avg(r.stars)
```

and the results table:

```
╒═════════════╤══════════════╕
│"b.name"     │"avg(r.stars)"│
╞═════════════╪══════════════╡
│"Bob's Pizza"│4.0           │
└─────────────┴──────────────┘
```

Of course, this isn't too exciting because we only have one business and one review. In the exercise section of this chapter, we'll work with a larger dataset.

3.7 Using the client drivers

Until now we've used Neo4j Browser to execute our Cypher queries, which is usual for ad-hoc analysis or prototyping; however, typically we want to create an application that interacts with the database. To do this, we use the Neo4j client drivers. These client drivers are available in many languages such as JavaScript, Java, Python, .NET, and Go, and allow the developer to execute Cypher queries against a Neo4j instance with a consistent API that is idiomatic to the language. In chapter 1 we saw an example of using the Neo4j JavaScript driver to execute a Cypher query and work with the results. Neo4j client drivers also provide a fundamental building block for building framework-specific integrations with Neo4j, such as neo4j-graphql.js, which we'll explore in future chapters. Refer to the Drivers & Language guides for more information on Neo4j client drivers: https://neo4j.com/developer/language-guides/ .

In the next chapter, we'll see how to build a GraphQL API that connects to Neo4j using the Neo4j JavaScript driver and the neo4j-graphql.js library.

Exercises

To complete the following exercises, first run the following command in Neo4j Browser: `:play grandstack` to load a browser guide with embedded queries. This browser guide will walk you through the process of loading a larger, more complete sample dataset of businesses and reviews. After running the query to load the data in Neo4j:

1. Run the command `CALL db.schema()` to inspect the data model. What are the node labels used? What are the relationship types?
2. Write a Cypher query to find all the users in the database. How many users are there? What are their names?

3. Find all the reviews written by the user named "Will". What's the average rating given by this user?
4. Find all the businesses reviewed by the user named "Will". What's the most common category?
5. Write a query to recommend businesses to the user named "Will" that he hasn't previously reviewed.

You can find solutions to the exercises, as well as code samples, in the GitHub repository for this book: https://github.com/johnymontana/fullstack-graphql-book.

Summary

- A graph database allows you to model, store, and query data as a graph.
- The property graph data model is used by graph databases and consists of node labels, relationships, and properties.
- The Cypher query language is a declarative graph query language focused around pattern matching and is used for querying graph databases, including Neo4j.
- Client drivers are used for building applications that interact with Neo4j. These drivers enable application to send Cypher queries to the database and work with the results.

4 A GraphQL API for our graph database

This chapter covers

- Reviewing common issues that arise when building GraphQL backends
- Introducing database integrations for GraphQL that address common problems
- Building a GraphQL endpoint backed by Neo4j
- Extending the functionality of our GraphQL API with custom logic
- Inferring a GraphQL endpoint from an existing Neo4j database

GraphQL backend implementations commonly run into a set of issues that negatively impact performance and developer productivity. We've identified several of these problems previously (such as the ”n+1 query problem”), and in this chapter we look deeper at the common issues that arise and discuss how they can be mitigated using database integrations for GraphQL that make it easier to build efficient GraphQL APIs backed by databases.

Specifically, we look at using neo4j-graphql.js, a Node.js library designed to work with JavaScript GraphQL implementations such as Apollo Server for building GraphQL APIs backed by Neo4j. neo4j-graphql.js allows us to generate GraphQL APIs from type definitions, driving the database data model from GraphQL, auto-generate resolvers for data fetching and mutations, including complex filtering, ordering, and pagination. neo4j-graphql.js also enables adding custom logic.

In this chapter, we look at using neo4j-graphql.js to integrate our business reviews GraphQL API with Neo4j, adding a persistence layer to our API. In this initial look at neo4j-graphql.js, we focus on querying existing data, using the sample dataset in Neo4j used in the previous chapter. We'll explore creating and updating data (GraphQL mutations), as well as more complex GraphQL querying semantics, such as interfaces and fragments in future chapters, introducing these concepts in the context of building out our user interface. Figure 4.1 shows how neo4j-graphql.js fits into the larger architecture of our application. The goal of neo4j-graphql.js is to make it easy to build an API backed by Neo4j.

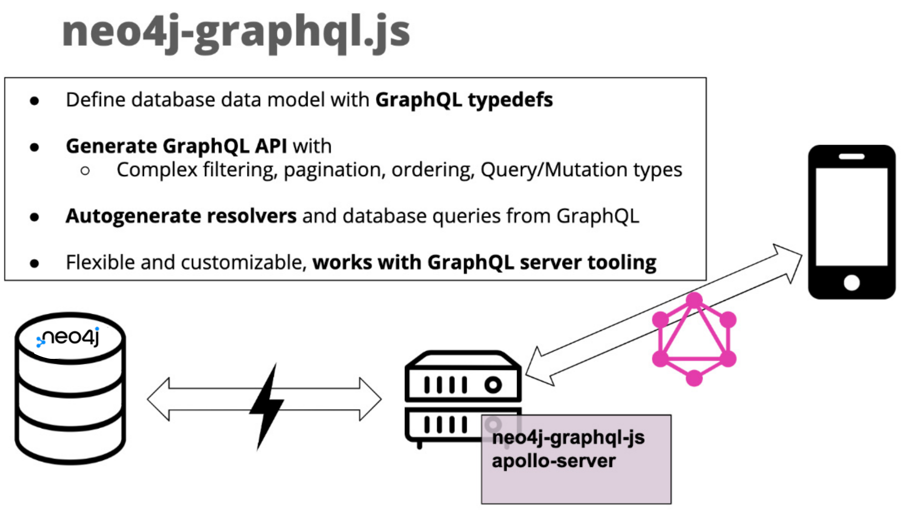

neo4j-graphql.js
Define database data model with GraphQL typedefs
Generate GraphQL API with
Complex filtering, pagination, ordering, Query/Mutation types
Autogenerate resolvers and database queries from GraphQL
Flexible and customizable, works with GraphQL server tooling
neo4j
neo4j-graphql-js
apollo-server

Figure 4.1 neo4j-graphql.js helps build the API layer between client and database.

4.1 Common GraphQL problems

When building GraphQL APIs, two types of common problems occur that developers can face: poor performance and writing lots of boilerplate code that can impact developer productivity.

4.1.1 Poor performance and the N+1 query problem

We've described the N+1 query problem previously. Because of the nested way that GraphQL resolvers are called, multiple database requests are often required to resolve a GraphQL query from the data layer. For example, imagine a query searching for businesses by name as well as all reviews for each business. A naive implementation would first query the database for all businesses matching the search phrase. Then for each matching business they send an additional query to the database to find any reviews for the business. Each query to the database incurs network and query latency, which can significantly impact performance.

A common solution for this is to use a caching and batching pattern known as DataLoader. This can alleviate part of the performance issues; however, it can still require multiple database requests and cannot be used in all cases, such as when the ID of an object isn't known.

4.1.2 Boilerplate and developer productivity

The term boilerplate is used to describe repetitive code that's written to accomplish a common task. In the case of implementing GraphQL APIs, often writing boilerplate code to implement data fetching logic in resolvers is required. This can negatively impact developer productivity, slowing down development as the developer is required to write simple data fetching logic for each type and field instead of focusing on the key components of their application. In the context of our business reviews application, this means manually writing the logic for finding businesses by name in the database, as well as finding reviews associated with each business and each user connected to each review, and so on, until we've manually defined the logic for fetching all fields of our GraphQL schema.

4.2 Introducing GraphQL database integrations

GraphQL database integrations are a class of tools that enable building GraphQL APIs that interact with databases. A handful of these tools exist with different feature sets and levels of integration—in this book we focus on neo4j-graphql.js. However, in general, the goal of these GraphQL "engines" is to address the common GraphQL problems identified previously in a consistent way by reducing boilerplate and addressing data fetching performance.

Throughout the rest of this chapter, we focus on using neo4j-graphql.js to build a GraphQL API backed by Neo4j. It's important to note that our GraphQL API serves as a layer between the client and the database; we don't want to directly query our database from the client. The API layer serves an important function where we can implement features such as authorization and custom logic that we don't want to expose to the client. Also, because GraphQL is an API query language (not a database query language), it lacks many semantics (such as aggregations and projections) that we expect in a database query language.

4.3 The neo4j-graphql.js library

Neo4j-graphql.js is a Node.js library that works with any JavaScript GraphQL implementation, such as GraphQL.js and Apollo Server, designed to make it as easy as possible to build GraphQL APIs backed by a Neo4j database. The two main functions of neo4j-graphql.js are *schema augmentation* and *GraphQL transpilation* (figure 4.2).

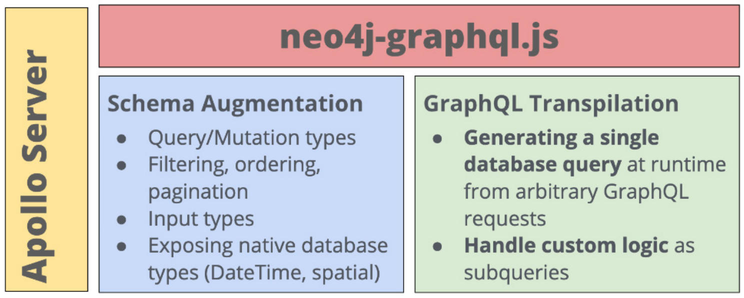

neo4j-graphql.js
Apollo Server
Schema Augmentation
Query/Mutation types
Filtering, ordering, pagination
Input types
Exposing native database types (DateTime, spatial)
GraphQL Transpilation
Generating a single database query at runtime from arbitrary GraphQL requests
Handle custom logic as subqueries

Figure 4.2 The two main functions of neo4j-graphql.js.

The schema augmentation process takes GraphQL type definitions and generates a GraphQL API with CRUD (Create, Read, Update, Delete) operations for the types defined. In GraphQL semantics, this includes adding a `Query` and `Mutation` type to the schema and generating resolvers for these queries and mutations. The generated API includes support for filtering, ordering, pagination, and native database types such as spatial and temporal types, without having to define these manually in the type definitions. The result of this process is a GraphQL executable schema object that can then be passed to a GraphQL server implementation, such as Apollo Server, to serve the API and handle networking and GraphQL execution processes. The schema augmentation process eliminates the need to write boilerplate code for data fetching and mapping the GraphQL and database schemas.

The GraphQL transpilation process happens at query time. When a GraphQL request is received, a single Cypher query is generated that can resolve the request and is sent to the database. Generating a single database query solves the n+1 query problem, assuring only one round trip to the database per GraphQL request.

You can find the documentation for neo4j-graphql.js at https://grandstack.io/docs.

4.3.1 Project setup

Throughout the rest of the chapter, we'll explore the features of neo4j-graphql.js by creating a new GraphQL API for Neo4j that uses the sample dataset of businesses and reviews from the Exercise section of the previous chapter. We'll first create a new Node.js project that uses neo4j-graphql.js and the Neo4j JavaScript driver to fetch data from Neo4j. Then we'll explore the various features of neo4j-graphql.js, adding additional code as we move along.

NEO4J

First, make sure a Neo4j instance is running (you can use Neo4j Desktop, Neo4j Sandbox, or Neo4j Aura, but we'll assume Neo4j Desktop for the purposes of this chapter). If using Neo4j Desktop, you need to install the APOC standard library plugin. Don't worry about this step if using Neo4j Sandbox or Neo4j Aura, because APOC is included by default in those services. To install APOC in Neo4j Desktop, click the "Plugins" tab in your project, then look for APOC in the list of available plugins, and click "Install".

Next, make sure your Neo4j database is empty by running the Cypher statement in listing 4.1.

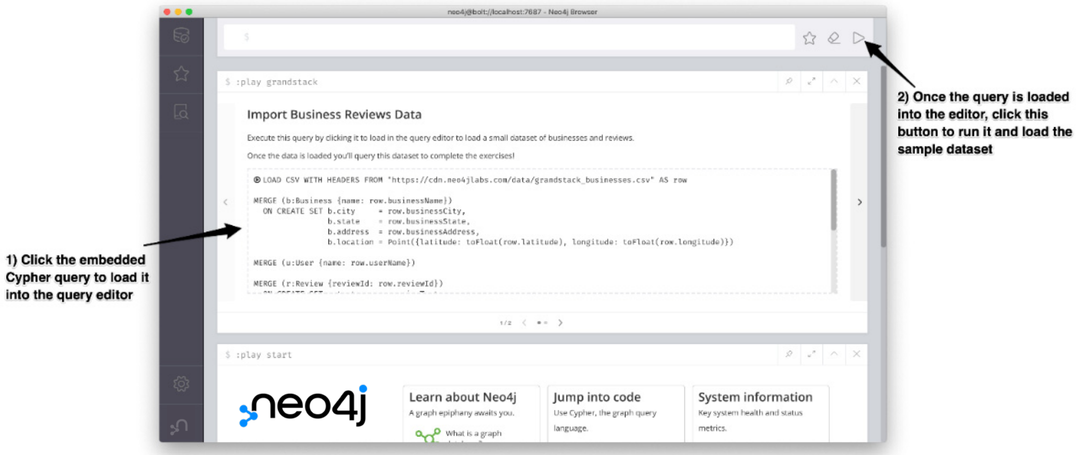
CAUTION This statement will delete all data in your Neo4j database so make sure this is the instance you want to use and not a database you don't want to delete.

Listing 4.1 Clearing out our Neo4j database

```
MATCH (a) DETACH DELETE a;
```

Now we're ready to load our sample dataset (figure 4.3), which you may have done already if you completed the exercise section in the previous chapter. Run the following command in Neo4j Browser:

```
:play grandstack
```


neo4j@bolt://localhost:7687 - Neo4j Browser
$
$:play grandstack
Import Business Reviews Data
1/2
$:play start
neo4j
Learn about Neo4j
A graph epiphany awaits you.
What is a graph
Jump into code
Use Cypher, the graph query language.
System information
Key system health and status metrics.
1) Click the embedded Cypher query to load it into the query editor
2) Once the query is loaded into the editor, click this button to run it and load the sample dataset

Execute this query by clicking it to load in the query editor to load a small dataset of businesses and reviews.

Once the data is loaded you'll query this dataset to complete the exercises!

```
LOAD CSV WITH HEADERS FROM "https://cdn.neo4jlabs.com/data/grandstack_businesses.csv" AS row

MERGE (b:Business {name: row.businessName})
  ON CREATE SET b.city     = row.businessCity,
                b.state    = row.businessState,
                b.address  = row.businessAddress,
                b.location = Point({latitude: toFloat(row.latitude), longitude: toFloat(row.longitude)})

MERGE (u:User {name: row.userName})

MERGE (r:Review {reviewId: row.reviewId})
```

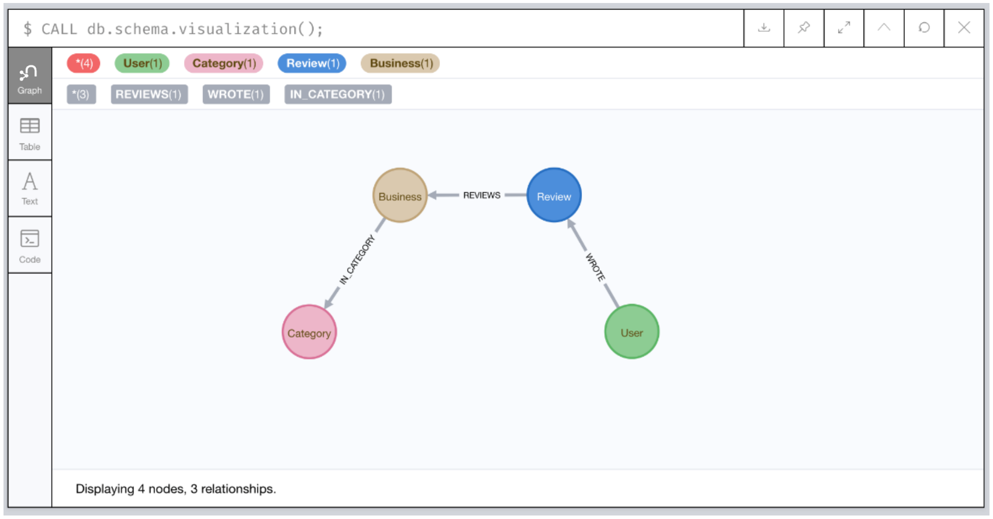
Figure 4.3 **Loading the sample dataset into Neo4j.**

This will load a sample dataset into Neo4j that we'll use as the basis for our GraphQL API. Next, we can explore the data a bit by running a command in the following listing that will give us a visual overview of the data included in the sample dataset (figure 4.4).

Listing 4.2 Visualizing the graph schema in Neo4j

```
CALL db.schema.visualization();
```

We see that we have four node labels: `Business`, `Review`, `Category`, and `User` connected by three relationship types: `IN_CATEGORY` (connecting businesses to the categories to which they belong), `REVIEWS` (connecting reviews to businesses), and `WROTE` (connecting users to reviews they have authored).



```
$ CALL db.schema.visualization();
```

Displaying 4 nodes, 3 relationships.

Figure 4.4 **The graph schema of our sample dataset.**

We can also view the node properties stored on the various node labels.

Listing 4.3 Inspect the node properties stored in Neo4j

```
CALL db.schema.nodeTypeProperties()
```

This command renders a table showing us the property names, their types and whether or not they're found on all nodes of that label.

"nodeType"	"nodeLabels"	"propertyName"	"propertyTypes"	"mandatory"
":`User`"	["User"]	"name"	["String"]	true
":`User`"	["User"]	"userId"	["String"]	true
":`Review`"	["Review"]	"reviewId"	["String"]	true
":`Review`"	["Review"]	"text"	["String"]	false
":`Review`"	["Review"]	"stars"	["Double"]	true
":`Review`"	["Review"]	"date"	["Date"]	true
":`Category`"	["Category"]	"name"	["String"]	true
":`Business`"	["Business"]	"name"	["String"]	true
":`Business`"	["Business"]	"city"	["String"]	true