Now, in the GraphQL query result we see longitude and latitude included for each business.

```
{
  "data": {
    "Business": [
      {
        "name": "Ducky's Car Wash",
        "location": {
          "latitude": 37.575968,
          "longitude": -122.336041
        }
      },
      {
        "name": "Hanabi",
        "location": {
          "latitude": 37.582598,
          "longitude": -122.351519
        }
      },
      {
        "name": "Imagine Nation Brewing",
        "location": {
          "latitude": 46.876672,
          "longitude": -114.009628
        }
      }
    ]
  }
}
```

4.9.2 Distance filter

When querying using point data, often we want to find things that are close to other things. For example, what businesses are within 1.5km of me? We can accomplish this using the auto-generated filter argument in the following listing.

Listing 4.25 GraphQL query

```
{
  Business(
    filter: {
      location_distance_lt: {
        point: { latitude: 37.563675, longitude: -122.322243 }
        distance: 3500
      }
    }
  ) {
    name
    address
    city
    state
  }
}
```

For points using the Geographic coordinate reference system (latitude and longitude), distance is measured in meters.

```
{
  "data": {
    "Business": [
      {
        "name": "Hanabi",
        "address": "723 California Dr",
        "city": "Burlingame",
        "state": "CA"
      },
      {
        "name": "Ducky's Car Wash",
        "address": "716 N San Mateo Dr",
        "city": "San Mateo",
        "state": "CA"
      },
      {
        "name": "Neo4j",
        "address": "111 E 5th Ave",
        "city": "San Mateo",
        "state": "CA"
      }
    ]
  }
}
```

4.10 Adding custom logic

We've seen basic querying operations created by neo4j-graphql.js. Often, we want to add custom logic to our API. For example, we may want to calculate the most popular business or recommend businesses to users. There are two options for adding custom logic to your API using neo4j-graphql.js: 1) the `@cypher` schema directive, and 2) by implementing custom resolvers.

4.10.1 The @cypher directive

We expose Cypher through GraphQL via the `@cypher` directive. Annotate a field in your schema with the `@cypher` directive to map the results of that query to the annotated GraphQL field. The `@cypher` directive takes a single argument `statement` which is a Cypher statement. Parameters are passed into this query at runtime, including `this` which is the currently resolved node as well as any field-level arguments defined in the GraphQL type definition.

> **NOTE** The `@cypher` directive feature requires the use of the APOC standard library plugin. Be sure you follow the steps to install APOC in the Project Setup section of this chapter.

COMPUTED SCALAR FIELDS

We can use the `@cypher` directive to define a custom scalar field, defining a computed field in our schema. Here we add an `averageStars` field to the `Business` type, which calculates the average stars of all reviews for the business using this variable, as shown in the following listing.

Listing 4.26 index.js: GraphQL type definitions

```
type Business {
    businessId: ID!
    averageStars: Float! @cypher(statement:"MATCH (this)<-[:REVIEWS]-
     (r:Review) RETURN avg(r.stars)")
    name: String!
    city: String!
    state: String!
    address: String!
    location: Point!
    reviews: [Review] @relation(name: "REVIEWS", direction: "IN")
    categories: [Category] @relation(name: "IN_CATEGORY", direction: "OUT")
}
```

We need to restart our GraphQL server because we have modified the type definitions:

```
node index.js
```

Now, let's include the `averageStars` field in our GraphQL query, as shown in the following listing.

Listing 4.27 GraphQL query including averageStars field

```
{
  Business {
    name
    averageStars
  }
}
```

And we see in the results that the computed value for `averageStars` is now included.

```
{
  "data": {
    "Business": [
      {
        "name": "Hanabi",
        "averageStars": 5
      },
      {
        "name": "Zootown Brew",
        "averageStars": 5
      },
      {
        "name": "Ninja Mike's",
        "averageStars": 4.5
      }
    ]
  }
}
```

The generated Cypher query includes the annotated Cypher query as a sub-query, preserving the single database call to resolve the GraphQL request.

COMPUTED OBJECT AND ARRAY FIELDS

We can also use the @cypher schema directive to resolve object and array fields. Let's add a recommended business field to the Business type. We'll use a simple Cypher query to find common businesses that other users reviewed. For example, if a user likes "Market on Front", we could recommend other businesses that users who reviewed "Market on Front" also reviewed, as shown in the following listing.

Listing 4.28 Cypher

```
MATCH (b:Business {name: "Market on Front"})<-[:REVIEWS]-(:Review)<-[:WROTE]-
      (:User)-[:WROTE]->(:Review)-[:REVIEWS]->(rec:Business)
WITH rec, COUNT(*) AS score
RETURN rec ORDER BY score DESC
```

We can use this Cypher query in our GraphQL schema by including it in a @cypher directive on the recommended field in our Business type definition, as shown in the following listing.

Listing 4.29 index.js: GraphQL type definitions

```
type Business {
    businessId: ID!
    averageStars: Float! @cypher(statement:"MATCH (this)<-[:REVIEWS]-
     (r:Review) RETURN avg(r.stars)")
    recommended(first: Int = 1): [Business] @cypher(statement: """
      MATCH (this)<-[:REVIEWS]-(:Review)<-[:WROTE]-(:User)-[:WROTE]-
     >(:Review)-[:REVIEWS]->(rec:Business)
      WITH rec, COUNT(*) AS score
      RETURN rec ORDER BY score DESC LIMIT $first
    """)
    name: String!
    city: String!
    state: String!
    address: String!
    location: Point!
    reviews: [Review] @relation(name: "REVIEWS", direction: "IN")
    categories: [Category] @relation(name: "IN_CATEGORY", direction: "OUT")
}
```

We also define a first field argument, which is passed to the Cypher query included in the @cypher directive and acts as a limit on the number of recommended businesses returned.

CUSTOM TOP-LEVEL QUERY FIELDS

Another helpful way to use the @cypher directive is as a custom query or mutation field. For example, let's see how we can add full-text query support to search for businesses. Applications often use full-text search to correct for things such as misspellings in user input using fuzzy matching.

In Neo4j, we can use full-text search by first creating a full-text index, as shown in the following listing.

Listing 4.30 Cypher: create full-text index

```
CALL db.index.fulltext.createNodeIndex("businessNameIndex",
     ["Business"],["name"])
```

Then to query the index, in this case we misspell "coffee" but including the ~ character enables fuzzy matching, ensuring we still find what we're looking for, as shown in the following listing.

Listing 4.31 Cypher: querying the full-text index

```
CALL db.index.fulltext.queryNodes("businessNameIndex", "cofee~")
```

Wouldn't it be nice to include this fuzzy matching full-text search in our GraphQL API? To do that let's create a Query field called `fuzzyBusinessByName` that takes a search string and searches for businesses, as shown in the following listing.

Listing 4.32 index.js: GraphQL type definitions

```
type Query {
    fuzzyBusinessByName(searchString: String): [Business] @cypher(
      statement: """
        CALL db.index.fulltext.queryNodes( 'businessNameIndex',
     $searchString+'~')
        YIELD node RETURN node
      """
   )
}
```

Again, since we've updated the type definitions, we must restart the GraphQL API application:

```
node index.js
```

If we check the Docs tab in GraphQL Playground, we'll see a new Query field `fuzzyBusinessByName`, and we can now search for business names using this fuzzy matching (see the following listing).

Listing 4.33 GraphQL query

```
{
  fuzzyBusinessByName(searchString: "libary") {
    name
  }
}
```

Because we're using full-text search, even though we spell "library" incorrectly, we still find matching results.

```
{
  "data": {
    "fuzzyBusinessByName": [
      {
        "name": "Missoula Public Library"
```

```
      }
    ]
  }
}
```

The @cypher schema directive is a powerful way to add custom logic and advanced functionality to our GraphQL API. We can also use the @cypher directive for authorization features, accessing values such as authorization tokens from the request object, a pattern that will be discussed in a later chapter when we explore different options for adding authorization to our API.

4.10.2 Implementing custom resolvers

While the @cypher directive is one way to add custom logic, in some cases we may need to implement custom resolvers that implement logic not able to be expressed in Cypher. For example, we may need to fetch data from another system, or apply custom validation rules. In these cases, we can implement a custom resolver and attach it to the GraphQL schema so that resolver is called to resolve our custom field instead of relying on the generated Cypher query by neo4j-graphql.js to resolve the field.

In our example, let's imagine there is an external system that can be used to determine current wait times at businesses. We want to add an additional waitTime field to the Business type in our schema and implement the resolver logic for this field to use this external system.

To do this, we first add the field to our schema, adding the @neo4j_ignore directive to ensure the field is excluded from the generated Cypher query. This is our way of telling neo4j-graphql.js that a custom resolver will be responsible for resolving this field and we don't expect it to be fetched from the database automatically, as shown in the following listing.

Listing 4.34 index.js: GraphQL type definitions

```
type Business {
    businessId: ID!
    waitTime: Int! @neo4j_ignore
    averageStars: Float!
      @cypher(
        statement: "MATCH (this)<-[:REVIEWS]-(r:Review) RETURN avg(r.stars)"
      )
    name: String!
    city: String!
    state: String!
    address: String!
    location: Point!
    reviews: [Review] @relation(name: "REVIEWS", direction: "IN")
    categories: [Category] @relation(name: "IN_CATEGORY", direction: "OUT")
  }
```

Next, we create a resolver map with our custom resolver. We didn't have to create this previously because neo4j-graphql.js generated our resolvers for us. Our wait time calculation will be selecting a value at random, but we could implement any custom logic here to determine the waitTime value, such as making a request to a third-party API, as shown in the following listing.

Listing 4.35 index.js: creating a resolver map

```
const resolvers = {
  Business: {
    waitTime: (obj, args, context, info) => {
      const options = [0, 5, 10, 15, 30, 45];
      return options[Math.floor(Math.random() * options.length)];
    }
  }
};
```

Then we add this resolver map to the parameters passed to makeAugmentedSchema, as shown in the following listing.

Listing 4.36 index.js: generating the GraphQL schema

```
const schema = makeAugmentedSchema({
  typeDefs,
  resolvers
});
```

Now we restart the GraphQL API application because we've updated the code:

```
node index.js
```

After restarting, in GraphQL Playground if we check the Docs for the Business type, we'll see our new field waitTime on the Business type.

Now, let's search for restaurants and see what their wait times are by including the waitTime field in the selection set in the following listing.

Listing 4.37 GraphQL query

```
{
  Business(filter: { categories_some: { name: "Restaurant" } }) {
    name
    waitTime
  }
}
```

In the results we now see a value for the wait time. Your results will of course vary because the value is randomized.

```
{
  "data": {
    "Business": [
      {
        "name": "Ninja Mike's",
        "waitTime": 5
      },
      {
        "name": "Market on Front",
        "waitTime": 45
      },
```

```
        {
          "name": "Hanabi",
          "waitTime": 45
        }
      ]
    }
  }
```

4.11 Inferring GraphQL schema from an existing database

Typically, when we start a new application, we don't have an existing database and follow the GraphQL-First development paradigm by starting with type definitions. However, in certain cases we may have an existing Neo4j database populated with data. In those cases, it can be convenient to generate GraphQL type definitions based on the existing database that can then be fed into `makeAugmentedSchema` to generate a GraphQL API for the existing database. We can do this with the use of the `inferSchema` functionality in neo4j-graphql.js.

This Node.js script will connect to our Neo4j database and infer the GraphQL type definitions that describe this data, then write those type definitions to a file named schema.graphql, as shown in the following listing.

Listing 4.38 infer.js: inferring GraphQL type definitions

```
const neo4j = require("neo4j-driver");
const { inferSchema } = require("neo4j-graphql-js");
const fs = require("fs");

const driver = neo4j.driver(
  "bolt://localhost:7687",
  neo4j.auth.basic("neo4j", "letmein")
);

const schemaInferenceOptions = {
  alwaysIncludeRelationships: false
};

inferSchema(driver, schemaInferenceOptions).then(result => {
  fs.writeFile("schema.graphql", result.typeDefs, err => {
    if (err) throw err;
    console.log("Updated schema.graphql");
    process.exit(0);
  });
});
```

Then we can load this schema.graphql file in the following listing and pass the type definitions into `makeAugmentedSchema`.

Listing 4.39 Initial GraphQL API code

```
// Load GraphQL type definitions from schema.graphql file
const typeDefs =
     fs.readFileSync(path.join(__dirname,"schema.graphql")).toString("utf-8");
```

Up to now, all of our GraphQL querying has been done using GraphQL Playground, which is great for testing and development, but typically our goal is to build an application that queries the GraphQL API. In the next few chapters, we'll start to build out the user interface for our business reviews application using React and Apollo Client. Along the way, we'll learn more about GraphQL concepts such as mutations, fragments, interface types, and more!

Exercises

1. Query the GraphQL API we created in this chapter using GraphQL Playground to find:
   - Which users have reviewed the business named "Hanabi"?
   - Find any reviews that contain the word "comfortable". What business(es) are they reviewing?
   - Which users have given no five-star reviews?
2. Add a `@cypher` directive field to the `Category` type that computes the number of businesses in each category. How many businesses are in the "Coffee" category?
3. Create a Neo4j Sandbox instance at https://sandbox.neo4j.com choosing from any of the pre-populated datasets. Using the `inferSchema` method from neo4j-graphql.js, create a GraphQL API for this Neo4j Sandbox instance without manually writing GraphQL type definitions. What data can you query for using GraphQL?

Refer to the book's GitHub repository to see the exercise solutions: https://github.com/johnymontana/fullstack-graphql-book.

Summary

- Common problems that arise when building GraphQL APIs include the n+1 query problem, schema duplication, and a large amount of boilerplate data-fetching code.
- GraphQL database integrations like neo4j-graphql.js can help mitigate these problems by generating database queries from GraphQL requests, driving database schema from GraphQL type definitions, and auto-generating a GraphQL API from GraphQL type definitions.
- neo4j-graphql.js makes it easy to build GraphQL APIs backed by a Neo4j database by generating resolvers for data fetching and adding filtering, ordering, and pagination to the generated API.
- Custom logic can be added by using the `@cypher` schema directive to define custom logic for fields, or by implementing custom resolvers and attaching them to the GraphQL schema.
- If we have an existing Neo4j database, we can use the `inferSchema` functionality of neo4j-graphql.js to generate GraphQL type definitions and a GraphQL API on top of the existing database.

index

G

H

I

M

N