Neo4j). Neo4j Desktop also includes functionality for installing "Graph Apps", which are applications that run in Neo4j Desktop and connect to the active Neo4j instance. Neo4j Browser, installed by default, is an example of one of these Graph Apps. See https://install.graphapp.io for examples of other Graph Apps.

If you haven't yet downloaded Neo4j Desktop do so now at https://neo4j.com/download. Neo4j Desktop is available to download for Mac, Windows, and Linux systems. Other options for downloading and running Neo4j, such as Docker, Debian package, or standalone Neo4j Server options can be found at neo4j.com/download-center although the remaining instructions will assume Neo4j Desktop is used.

Once you have downloaded and installed Neo4j, create a new local Neo4j instance by selecting Add Graph. You'll be prompted for a database name and password. The password can be anything you want, just be sure to remember it for later. Once you've created the graph, click the Start button to activate it, then we'll use Neo4j Browser to start querying the database we just created.

3.5 Tooling: Neo4j Browser

Neo4j Browser is a query workbench for Neo4j that allows developers to interact with the database by writing Cypher queries and visualizing the results (figure 3.7). Start Neo4j Browser by selecting its application icon in the Application section of Neo4j Browser.

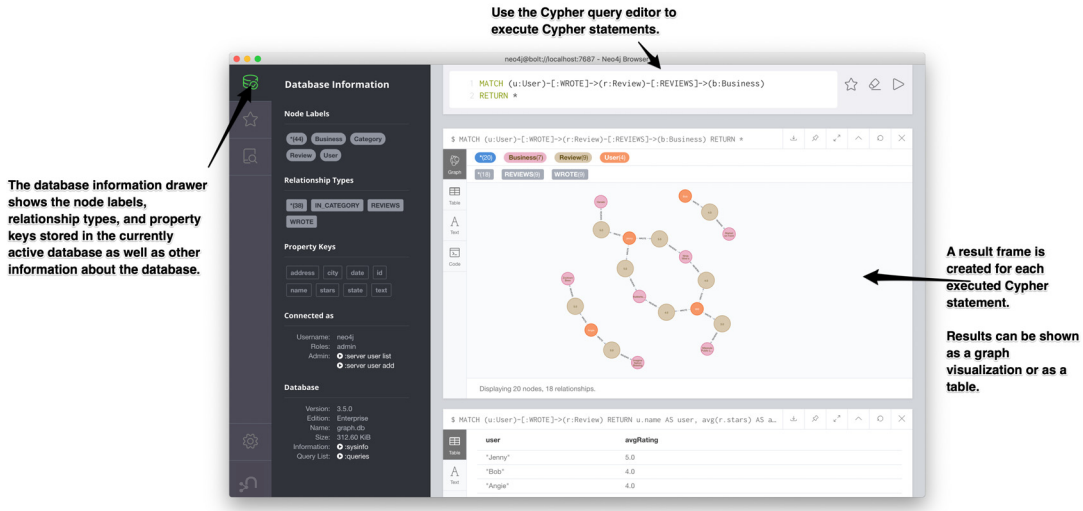


Figure 3.7 Neo4j Browser: A query workbench for Cypher and Neo4j.

Neo4j Browser allows us to run Cypher queries in Neo4j, but first let's review the Cypher query language.

3.6 Cypher

Cypher is a declarative graph query language with features that may be familiar from SQL. In fact, a good way to think of Cypher is as "SQL for graphs". Cypher uses pattern

matching, using an ASCII-art like notation for describing graph patterns. In this section, we'll look at basic Cypher functionality for creating and querying data, including making use of predicates and aggregations. We'll only cover a small part of the Cypher language. See the Cypher refcard https://r.neo4j.com/refcard for a through reference or consult the documentation at https://neo4j.com/docs/cypher-manual/current/ .

3.6.1 Pattern matching

As a declarative graph query language, pattern matching is a fundamental tool used in Cypher, both for creating and querying data. Instead of telling the database the exact operations, we want it to take (an imperative approach); with Cypher, we describe the pattern we're looking for or want to create and the database figures out the series of operations that satisfies the statement in the most efficient way possible. Describing graph patterns using an ASCII-art like notation (also called motifs) is at the heart of this declarative functionality.

NODES

Nodes are defined within parentheses `()`. Optionally, we can specify node label(s) using a colon as a separator, for example `(:User)`.

RELATIONSHIPS

Relationships are defined within square brackets `[]`. Optionally we can specify type and direction: `(:Review)-[:REVIEWS]→(:Business)`.

3.6.2 Properties

Properties are specified as comma-separated name: value pairs within braces "{}", like the name of a business or user.

ALIASES

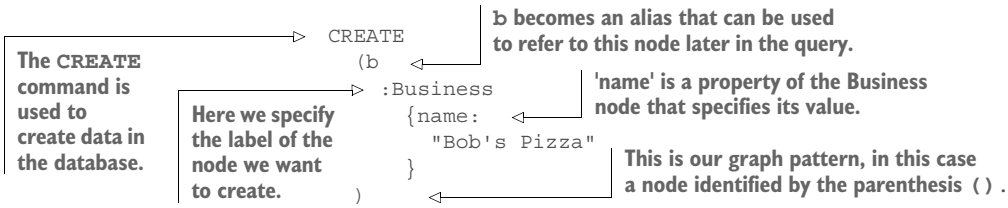
Graph elements can be bound to aliases that can be referred to later in the query: `(r:Review)-[a:REVIEWS]→(b:Business)`. The alias `r` becomes a variable bound to the review node matched in the pattern, `a` is bound to the `REVIEWS` relationship, and `b` is bound to the business node. These variables are only in scope for the Cypher query in which they're used.

Follow along by running the Cypher queries in Neo4j browser as we introduce Cypher commands for creating and querying data that matches the data model we built throughout this chapter.

3.6.3 CREATE

The first thing we need to do is create data in our database. Here's a look back at the objects we used in the previous chapter:

First, to create a single `Business` node in the graph, we start with the `CREATE` statement because we want to add data to the database. The `CREATE` clause takes a graph pattern upon which it operates:



And the result of running in Neo4j Browser shows:

```
Added 1 label, created 1 node, set 1 property, completed after 4 ms.
```

which means we've created 1 node with a new label in the database and set 1 node property value, in this case the `name` property on a node with the label `Business`.

Alternatively, we can use the `SET` command. This is equivalent:

```
CREATE (b:Business)
SET b.name = "Bob's Pizza"
```

To visualize the data being created we can add a `RETURN` clause to the Cypher statement, which will be rendered in Neo4j browser as a graph visualization. Running

```
CREATE (b:Business)
SET b.name = "Bob's Pizza"
RETURN b
```

gives the following visualization in Neo4j Browser (figure 3.8).

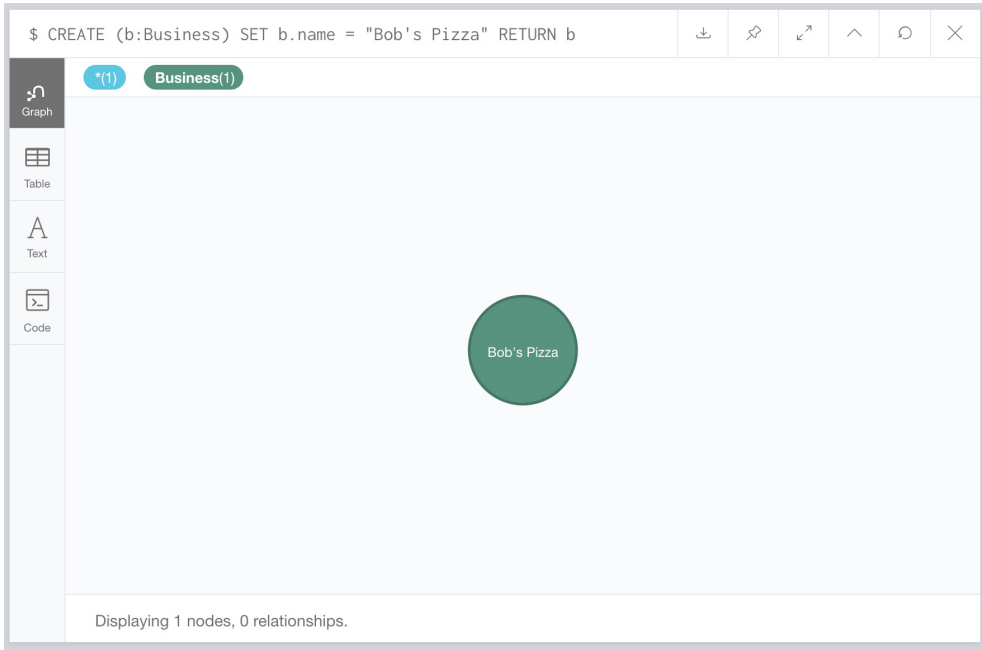


Figure 3.8 **Creating data with Cypher and Neo4j Browser.**

We can specify more complex patterns in the `CREATE` statement, such as relationships. Note the ASCII-art notation of defining a relationship using square brackets `←[]-`, including the direction of the relationship (figure 3.9).

```
CREATE (b:Business {name: "Bob's Pizza"})<-[:REVIEWS]-(r:Review {stars: 4,
     text: "Great pizza"})
RETURN b, r
```

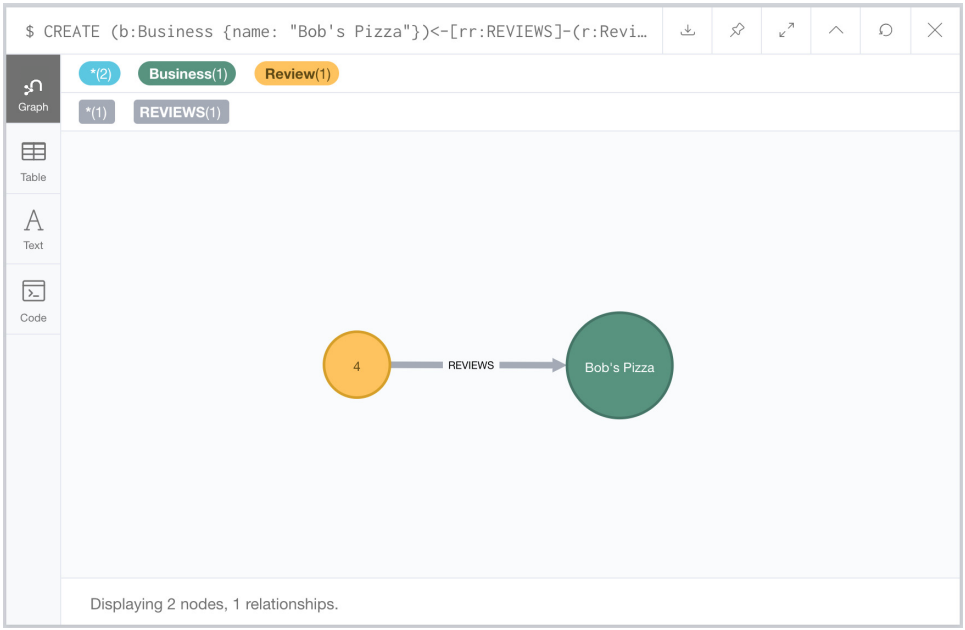


Figure 3.9 Creating data with Cypher and Neo4j Browser.

and even arbitrarily complex graph patterns (figure 3.10):

```
CREATE p=(b:Business {name: "Bob's Pizza"})<-[:REVIEWS]-(r:Review {stars: 4,
     text: "Great pizza"})<-[:WROTE]-(u:User {name: "Willie"})
RETURN p
```

Note that in the previous Cypher query we bind the entire graph pattern to a variable `p` and return that variable. In this case, `p` takes on the value of the entire path (a combination of nodes and relationships) being created.

Up to now we've returned only the data we've created in each Cypher statement. How do we query and visualize the rest of the data in the database? To do this, we use the `MATCH` keyword. Let's match on all nodes in the database and return them:

```
MATCH (a) RETURN a
```

and we should see a graph that looks something like figure 3.11.

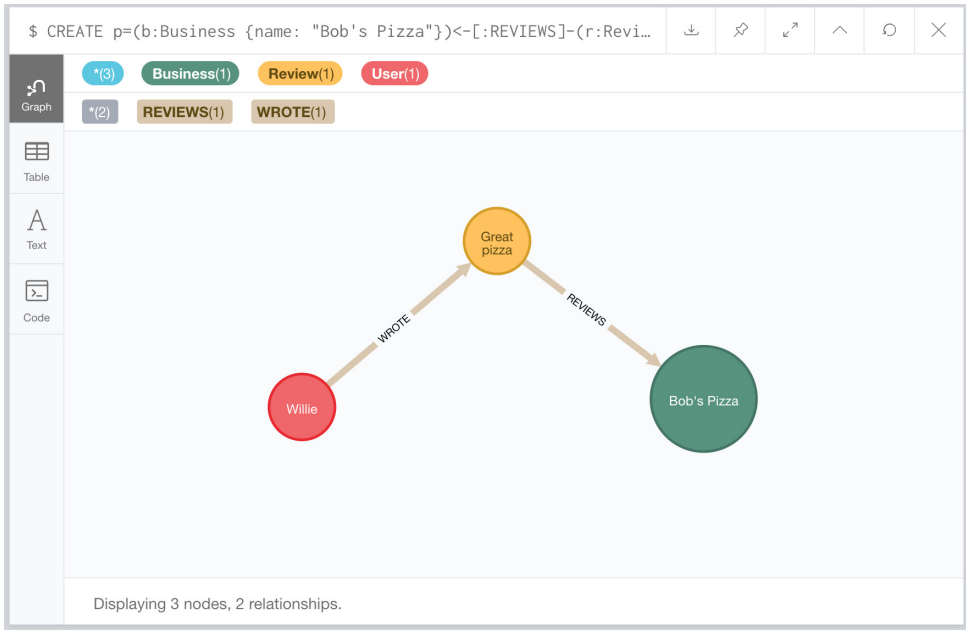


Figure 3.10 **Creating data with Cypher and Neo4j Browser.**

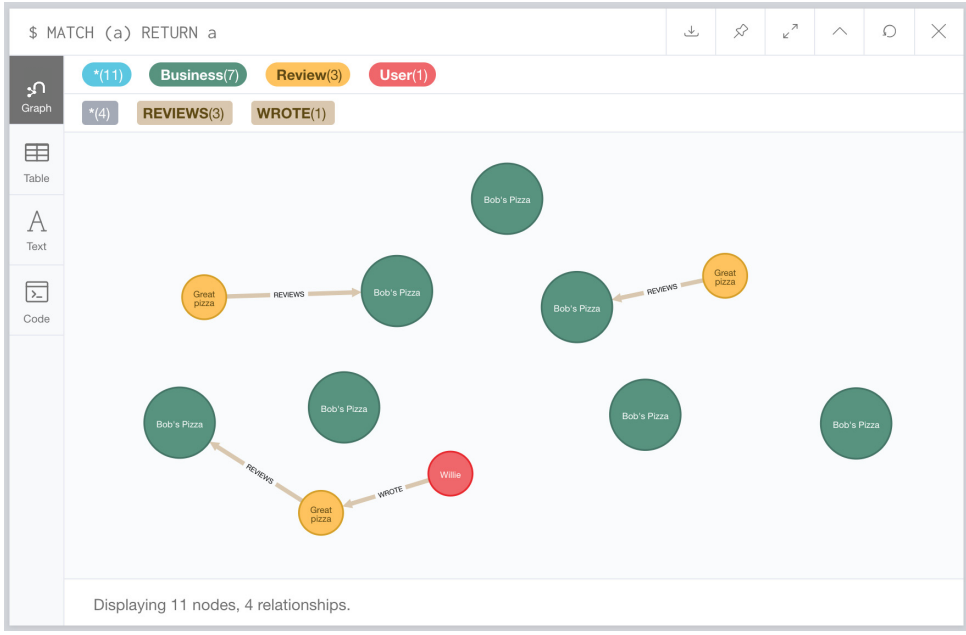


Figure 3.11 **Creating data with Cypher and Neo4j Browser.**

Right away, we can see something is wrong; we've created many duplicate nodes in our graph! Let's delete all data in the database:

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

This will match on all nodes and delete both the nodes and any relationships.

We should see output that tells us what we've deleted:

```
Deleted 11 nodes, deleted 4 relationships, completed after 23 ms.
```

Now let's start over and learn how to create data in the database without creating duplicates.

3.6.4 MERGE

To avoid creating duplicates, we can use the `MERGE` command. `MERGE` acts as an upset: only creating data specified in the pattern if it doesn't already exist in the database. When using `MERGE`, it's best to create a uniqueness constraint on the property that identifies uniqueness—often an id field. By creating a uniqueness constraint, this will also create an index in the database. See the next section for example of creating uniqueness constraints. For simple examples, it's fine to use `MERGE` without these constraints, so let's revisit our Cypher statement that created a business, a review and user, but this time we'll use `MERGE`:

```
MERGE (b:Business {name: "Bob's Pizza"})
MERGE (r:Review {stars: 4, text: "Great pizza!"})
MERGE (u:User {name: "Willie"})
MERGE (b)<-[:REVIEWS]-(r)<-[:WROTE]-(u)
RETURN *
```

And the resulting graph visualization, showing the data we've created in figure 3.12.

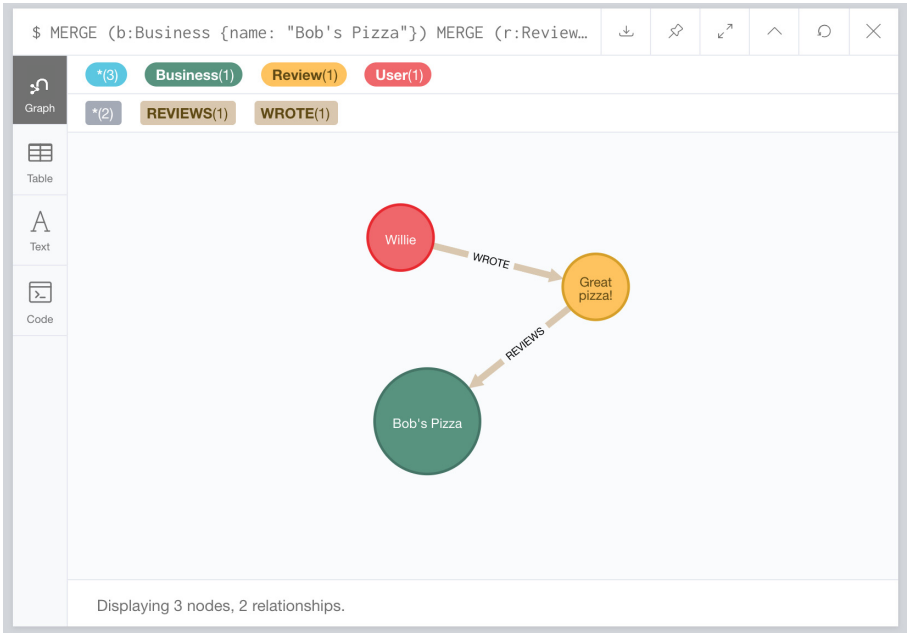


Figure 3.12 Using MERGE to create data.

The results of this Cypher statement look identical to the version using `CREATE`; however, there's an important difference: this query is now idempotent. No matter how many times we run the query, we won't create duplicate nodes, because we're using `MERGE` instead of `CREATE`. We'll revisit `MERGE` again in the next chapter when we show how to create data in the database via our GraphQL API.

Indexes In Neo4j

It's important to understand how indexes are used in a graph database like Neo4j. We said earlier that Neo4j has a property called index-free adjacency, which means that traversing from a node to any other connected node does not require an index lookup. How are indexes used in Neo4j? Indexes are used to find the starting point for a traversal only, unlike relational databases that use an index to compute set (table) overlap, graph databases are simply computing offsets in the file store, essentially chasing pointers, which we know computers are good at doing quickly.

3.6.5 Defining database constraints with Cypher

We mentioned database constraints and how they relate to (optionally) defining a schema in Neo4j earlier in the chapter as we built up our data model. Here we show the Cypher syntax for creating database constraints relevant to our data model.

Uniqueness constraints are used to assert that the database will never contain more than one node with a specific label and property value. In this case we create a uniqueness constraint ensuring that the value of the `businessId` property on nodes with the label `Business` is unique.

```
CREATE CONSTRAINT ON (b:Business) ASSERT b.businessId IS UNIQUE;
```

Property existence constraints ensure that all nodes with a certain label have a certain property. Here we create a constraint to guarantee that all `Business` nodes will have a `name` property, however the value for this property does not need to be unique across all `Business` nodes.

```
CREATE CONSTRAINT ON (b:Business) ASSERT exists(b.name);
```

Node key constraints ensure that all nodes with a certain label have a set of defined properties whose combined value is unique and that all properties in this set exist on the node. Here we create a node key constraint ensuring that the combination of `firstName` and `lastName` is unique for `Person` nodes, and that every `Person` node has both `firstName` and `lastName` properties.

```
CREATE CONSTRAINT ON (p:Person) ASSERT (p.firstName, p.lastName) IS NODE KEY;
```

Note that if you still have duplicate data in the database that conflict with any of these constraints, then you'll receive an error message saying the constraint cannot be created.

3.6.6 MATCH

Now that we've created our data in the graph, we can start to write queries to address several of the business requirements of our application.

The MATCH clause is similar to CREATE in that it takes a graph pattern; however, we can also use a WHERE clause for specifying predicates to be applied in the pattern.

```
MATCH (u:User)
RETURN u
```

We can, of course, use more complex graph patterns in a MATCH clause:

```
MATCH (u:User)-[:WROTE]->(r:Review)-[:REVIEWS]->(b:Business)
RETURN u, r, b
```

The previous query matches on all users that have written a review of any business. What if instead we only want to query for reviews of a certain business? In that case, we need to introduce predicates into our query. Notice how the relationship arrows are reversed here.

WHERE

The WHERE clause can be used to add predicates to a MATCH statement. To search for the business named "Bob's Pizza":

```
MATCH (b:Business)
WHERE b.name = "Bob's Pizza"
RETURN b
```

For equality comparisons, an equivalent shorthand notation is available:

```
MATCH (b:Business {name: "Bob's Pizza"})
RETURN b
```

3.6.7 Aggregations

Often, we want to compute an aggregation across a set of results. For example, we may want to calculate the average rating of all the reviews of Bob's Pizza. To do this, we use the avg aggregation function:

```
MATCH (b:Business {name: "Bob's Pizza"})<-[:REVIEWS]-(r:Review)
RETURN avg(r.stars)
```

Now in Neo4j Browser because we're not returning graph data, and rather tabular data, instead of a graph visualization, we're presented with a table showing the results of our query:

```
╒═══════════════╕
│"avg(r.stars)"│
╞═══════════════╡
│4.0            │
└───────────────┘
```

What if we wanted to calculate the average rating of *each* business? In SQL, we might use a GROUP BY operator to group the reviews by business name and calculate the