the database, rather than in the client, so only the necessary data is returned from the database query. See the following listing.

Listing 4.16 Generated Cypher query

```
MATCH (`business`:`Business`) WITH `business` ORDER BY business.name ASC
RETURN `business` { .name } AS `business` LIMIT toInteger($first)
```

Note that this query includes a `$first` parameter, rather than including the value `3` inline in the query. Parameter usage is important here because it ensures a user can't inject potentially malicious Cypher code into the generated query and also it ensures the query plan generated by Neo4j can be reused, increasing performance.

To run this query in Neo4j Browser first set a value for the `first` parameter with the `:param` command:

```
:param first => 3
```

4.6 Nested queries

Cypher can easily express the types of graph traversals in our GraphQL queries, and neo4j-graphql.js is capable of generating the equivalent Cypher queries for arbitrary GraphQL requests, including nested queries.

In the following listing, we traverse from businesses to their categories.

Listing 4.17 GraphQL query

```
{
  Business(first: 3, orderBy: name_asc) {
    name
    categories {
      name
    }
  }
}
```

And the result shows each business is connected to one or more categories.

```
{
  "data": {
    "Business": [
      {
        "name": "Ducky's Car Wash",
        "categories": [
          {
            "name": "Car Wash"
          }
        ]
      },
      {
        "name": "Hanabi",
        "categories": [
          {
            "name": "Restaurant"
          },
```

```
          {
            "name": "Ramen"
          }
        ]
      },
      {
        "name": "Imagine Nation Brewing",
        "categories": [
          {
            "name": "Beer"
          },
          {
            "name": "Brewery"
          }
        ]
      }
    ]
  }
}
```


4.7 Filtering

The filter functionality is exposed by adding a filter argument with associated inputs based on the GraphQL type definitions that expose filtering criteria. You can see the full list of filtering criteria in the documentation at https://grandstack.io/docs/graphql-filtering.html.

4.7.1 Filter argument

In the following listing we use the filter argument to search for business names that contain "Brew".

Listing 4.18 GraphQL query

```
{
  Business(filter: { name_contains: "Brew" }) {
    name
    address
  }
}
```

Our results now show businesses that match the filtering criteria and only businesses that contain the string "Brew" in their name are returned.


```
{
  "data": {
    "Business": [
      {
        "name": "KettleHouse Brewing Co.",
        "address": "313 N 1st St W"
      },
      {
        "name": "Imagine Nation Brewing",
        "address": "1151 W Broadway St"
      },
```

```
      {
        "name": "Zootown Brew",
        "address": "121 W Broadway St"
      }
    ]
  }
}
```

4.7.2 Nested filter

To filter based on the results of nested fields applied to the root, we can nest our filter arguments. In the following listing we search for businesses where the name contains "Brew" and have at least one review with at least a 4.75 rating.

Listing 4.19 GraphQL query

```
{
  Business(
    filter: { name_contains: "Brew", reviews_some: { stars_gte: 4.75 } }
  ) {
    name
    address
  }
}
```

If we inspect the results of this GraphQL query we can see two matching businesses.

```
{
  "data": {
    "Business": [
      {
        "name": "KettleHouse Brewing Co.",
        "address": "313 N 1st St W"
      },
      {
        "name": "Zootown Brew",
        "address": "121 W Broadway St"
      }
    ]
  }
}
```

4.7.3 Logical operators: AND, OR

Filters can be wrapped in logical operators `OR` and `AND`. For example, we can search for businesses in either the Coffee or Breakfast category by using an `OR` operator in the filter argument, as shown in the following listing.

Listing 4.20 GraphQL query

```
{
  Business(
    filter: {
      OR: [
        { categories_some: { name: "Coffee" } }
```

```
        { categories_some: { name: "Breakfast" } }
      ]
    }
  ) {
    name
    address
    categories {
      name
    }
  }
}
```

This GraphQL query yields businesses that are connected to either the Coffee or Breakfast category.

```
{
  "data": {
    "Business": [
      {
        "name": "Ninja Mike's",
        "address": "200 W Pine St",
        "categories": [
          {
            "name": "Restaurant"
          },
          {
            "name": "Breakfast"
          }
        ]
      },
      {
        "name": "Market on Front",
        "address": "201 E Front St",
        "categories": [
          {
            "name": "Coffee"
          },
          {
            "name": "Restaurant"
          },
          {
            "name": "Cafe"
          },
          {
            "name": "Deli"
          },
          {
            "name": "Breakfast"
          }
        ]
      },
      {
        "name": "Zootown Brew",
        "address": "121 W Broadway St",
        "categories": [
          {
```

```
            "name": "Coffee"
          }
        ]
      }
    ]
  }
}
```

4.7.4 Filtering in selections

Filters can also be used throughout the selection set to apply the filter at the level of the selection. For example, let's say we want to find all Coffee or Breakfast businesses, but only view reviews containing the phrase "breakfast sandwich". See the following listing.

Listing 4.21 GraphQL query

```
{
  Business(
    filter: {
      OR: [
        { categories_some: { name: "Coffee" } }
        { categories_some: { name: "Breakfast" } }
      ]
    }
  ) {
    name
    address
    reviews(filter: { text_contains: "breakfast sandwich" }) {
      stars
      text
    }
  }
}
```

Because the filter was applied at the `reviews` selection, businesses that don't have any reviews containing the phrase "breakfast sandwich" are still shown in the results; however, only reviews containing that phrase are shown.

```
{
  "data": {
    "Business": [
      {
        "name": "Ninja Mike's",
        "address": "200 W Pine St",
        "reviews": [
          {
            "stars": 4,
            "text": "Best breakfast sandwich at the Farmer's Market. Always
     get the works."
          }
        ]
      },
      {
```

```
        "name": "Market on Front",
        "address": "201 E Front St",
        "reviews": []
      },
      {
        "name": "Zootown Brew",
        "address": "121 W Broadway St",
        "reviews": []
      }
    ]
  }
}
```

4.8 Working with temporal fields

Neo4j supports native temporal types as properties on nodes and relationships. These types include Date, DateTime, and LocalDateTime. With neo4j-graphql.js you can use these temporal types in your GraphQL schema.

4.8.1 Using temporal fields in queries

Temporal types expose their date components (such as day, month, year, hour, etc.) as fields, as well as a formatted field which is the ISO-8601 string representation of the temporal value. The specific fields available vary depending on which temporal is used. See the following listing.

Listing 4.22 GraphQL query

```
{
  Review(first: 3, orderBy: date_desc) {
    stars
    date {
      formatted
    }
    business {
      name
    }
  }
}
```

Because we specified the `formatted` field on our date property, we see that in the results.

```
{
  "data": {
    "Review": [
      {
        "stars": 3,
        "date": {
          "formatted": "2018-09-10"
        },
        "business": {
          "name": "Imagine Nation Brewing"
        }
      },
```

```
      {
        "stars": 5,
        "date": {
          "formatted": "2018-08-11"
        },
        "business": {
          "name": "Zootown Brew"
        }
      },
      {
        "stars": 4,
        "date": {
          "formatted": "2018-03-24"
        },
        "business": {
          "name": "Market on Front"
        }
      }
    ]
  }
}
```

4.8.2 DateTime filters

Temporal fields are also included in the generated filtering enums, allowing for filtering using dates and date ranges. In the following listing we search for reviews created before January 1, 2017.

Listing 4.23 GraphQL query

```
{
  Review(
    first: 3
    orderBy: date_desc
    filter: { date_lte: { year: 2017, month: 1, day: 1 } }
  ) {
    stars
    date {
      formatted
    }
    business {
      name
    }
  }
}
```

We can see the results are now ordered by the `date` field.

```
{
  "data": {
    "Review": [
      {
        "stars": 5,
        "date": {
          "formatted": "2016-11-21"
        },
```

```
        "business": {
          "name": "Hanabi"
        }
      },
      {
        "stars": 5,
        "date": {
          "formatted": "2016-07-14"
        },
        "business": {
          "name": "KettleHouse Brewing Co."
        }
      },
      {
        "stars": 4,
        "date": {
          "formatted": "2016-01-03"
        },
        "business": {
          "name": "KettleHouse Brewing Co."
        }
      }
    ]
  }
}
```

4.9 Working with spatial data

Neo4j currently supports the spatial Point type, which can represent both 2D (such as latitude and longitude) and 3D (such as x,y,z or latitude, longitude, height) points. neo4j-graphql.js makes available the Point type for use in your GraphQL type definitions. The GraphQL schema augmentation process will translate the location field to a `_Neo4jPoint` type in the augmented schema.

4.9.1 The point type in selections

`Point` type fields are object fields in the GraphQL schema, so let's retrieve the latitude and longitude fields for our matching businesses by adding those fields to our selection set as shown in the following listing.

Listing 4.24 GraphQL query

```
{
  Business(first: 3, orderBy: name_asc) {
    name
    location {
      latitude
      longitude
    }
  }
}
```