7 The rise of graph databases

This chapter covers

- Introducing connected data and how it's related to graphs and graph databases
- Learning how graph databases differ from relational databases
- Discovering the graph database Neo4j
- Applying the data science process to a recommender engine project with the graph database Neo4j

Where on one hand we're producing data at mass scale, prompting the likes of Google, Amazon, and Facebook to come up with intelligent ways to deal with this, on the other hand we're faced with data that's becoming more interconnected than ever. Graphs and networks are pervasive in our lives. By presenting several motivating examples, we hope to teach the reader how to recognize a graph problem when it reveals itself. In this chapter we'll look at how to leverage those connections for all they're worth using a graph database, and demonstrate how to use Neo4j, a popular graph database.

7.1 Introducing connected data and graph databases

Let's start by familiarizing ourselves with the concept of connected data and its representation as graph data.

- *Connected data*—As the name indicates, connected data is characterized by the fact that the data at hand has a relationship that makes it connected.
- *Graphs*—Often referred to in the same sentence as connected data. Graphs are well suited to represent the connectivity of data in a meaningful way.
- *Graph databases*—Introduced in chapter 6. The reason this subject is meriting particular attention is because, besides the fact that data is increasing in size, it's also becoming more interconnected. Not much effort is needed to come up with well-known examples of connected data.

A prominent example of data that takes a network form is social media data. Social media allows us to share and exchange data in networks, thereby generating a great amount of connected data. We can illustrate this with a simple example. Let's assume we have two people in our data, User1 and User2. Furthermore, we know the first name and the last name of User1 (first name: Paul and last name: Beun) and User2 (first name: Jelme and last name: Ragnar). A natural way of representing this could be by drawing it out on a whiteboard, as shown in figure 7.1.

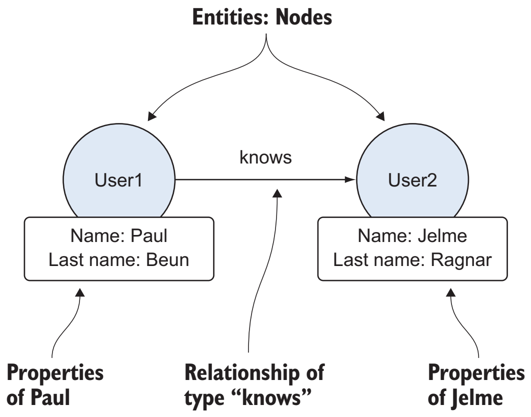


Figure 7.1 A simple connected data example: two entities or nodes (User1, User2), each with properties (first name, last name), connected by a relationship (knows)

The terminology of figure 7.1 is described below:

- *Entities*—We have two entities that represent people (User1 and User2). These entities have the properties "name" and "lastname".
- *Properties*—The properties are defined by key-value pairs. From this graph we can also infer that User1 with the "*name*" property Paul knows User2 with the "*name*" property Jelme.

- *Relationships*—This is the relationship between Paul and Jelme. Note that the relationship has a direction: it's Paul who "*knows*" Jelme and not the other way around. User1 and User2 both represent people and could therefore be grouped.
- *Labels*—In a graph database, one can group nodes by using labels. User1 and User2 could in this case both be labeled as "User".

Connected data often contains many more entities and connections. In figure 7.2 we can see a more extensive graph. Two more entities are included: Country1 with the name Cambodia and Country2 with the name Sweden. Two more relationships exist: "Has_been_in" and "Is_born_in". In the previous graph, only the entities included a property, now the relationships also contain a property. Such graphs are known as property graphs. The relationship connecting the nodes User1 and Country1 is of the type "Has_been_in" and has as a property "Date" which represents a data value. Similarly, User2 is connected to Country2 but through a different type of relationship, which is of the type "Is_born_in". Note that the types of relationships provide us a context of the relationships between nodes. Nodes can have multiple relationships.

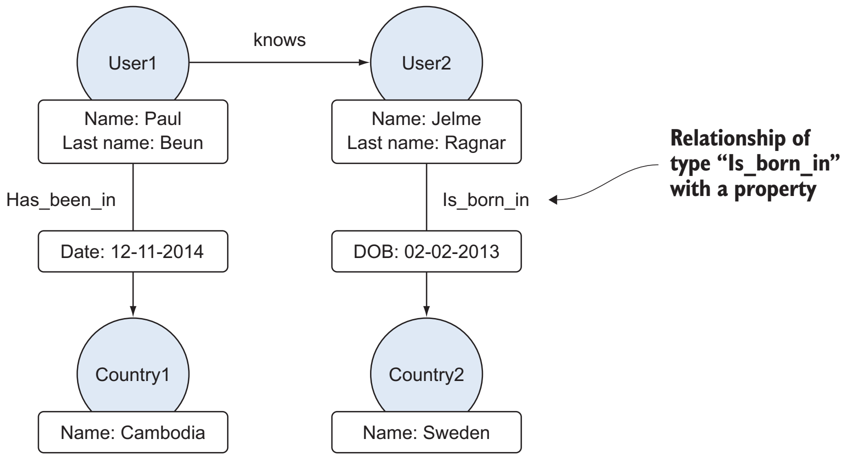


Figure 7.2 A more complicated connected data example where two more entities have been included (Country1 and Country2) and two new relationships ("Has_been_in" and "Is_born_in")

This kind of representation of our data gives us an intuitive way to store connected data. To explore our data we need to traverse through the graph following predefined paths to find the patterns we're searching for. What if one would like to know where Paul has been? Translated into graph database terminology, we'd like to find the pattern "Paul has been in." To answer this, we'd start at the node with the name "Paul" and traverse to Cambodia via the relationship "Has_been_in". Hence a graph traversal, which corresponds to a database query, would be the following:

1. *A starting node*—In this case the node with name property "Paul"
2. *A traversal path*—In this case a path starting at node Paul and going to Cambodia
3. *End node*—Country node with name property "Cambodia"

To better understand how graph databases deal with connected data, it's appropriate to expand a bit more on graphs in general. Graphs are extensively studied in the domains of computer science and mathematics in a field called graph theory. Graph theory is the study of graphs, where graphs represent the mathematical structures used to model pairwise relations between objects, as shown in figure 7.3. What makes them so appealing is that they have a structure that lends itself to visualizing connected data. A graph is defined by vertices (also known as nodes in the graph database world) and edges (also known as relationships). These concepts form the basic fundamentals on which graph data structures are based.

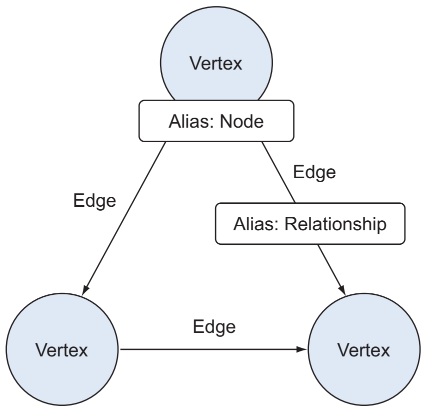


Figure 7.3 At its core a graph consists of nodes (also known as vertices) and edges (that connect the vertices), as known from the mathematical definition of a graph. These collections of objects represent the graph.

Compared to other data structures, a distinctive feature of connected data is its nonlinear nature: any entity can be connected to any other via a variety of relationship types and intermediate entities and paths. In graphs, you can make a subdivision between directed and undirected graphs. The edges of a directed graph have—how could it be otherwise—a direction. Although one could argue that every problem could somehow be represented as a graph problem, it's important to understand when it's ideal to do so and when it's not.

7.1.1 Why and when should I use a graph database?

The quest of determining which graph database one should use could be an involved process to undertake. One important aspect in this decision making process is finding the right representation for your data. Since the early 1970s the most common type of database one had to rely on was a relational one. Later, others emerged, such as the hierarchical database (for example, IMS), and the graph database's closest relative: the network database (for example, IDMS). But during the last decades the landscape has become much more diverse, giving end-users more choice depending on their specific needs. Considering the recent development of the data that's becoming available, two characteristics are well suited to be highlighted here. The first one is the size of the data and the other the complexity of the data, as shown in figure 7.4.

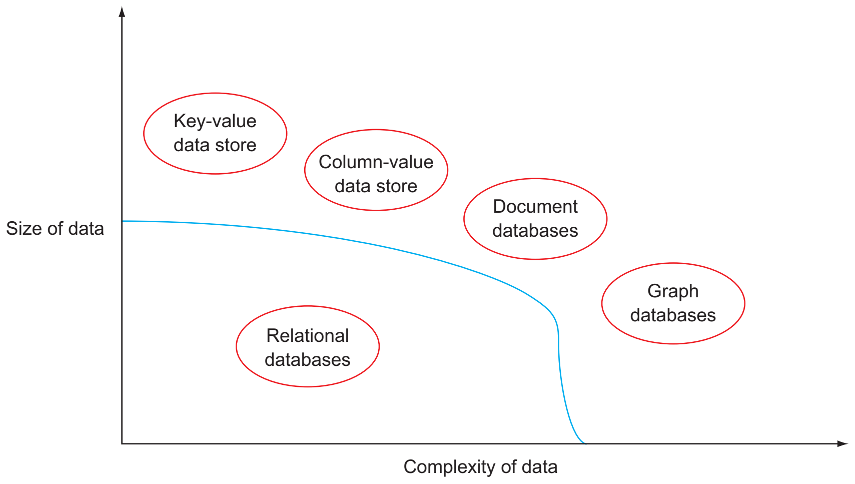


Figure 7.4 This figure illustrates the positioning of graph databases on a two dimensional space where one dimension represents the size of the data one is dealing with, and the other dimension represents the complexity in terms of how connected the data is. When relational databases can no longer cope with the complexity of a data set because of its connectedness, but not its size, graph databases may be your best option.

As figure 7.4 indicates, we'll need to rely on a graph database when the data is complex but still small. Though "small" is a relative thing here, we're still talking hundreds of millions of nodes. Handling complexity is the main asset of a graph database and the ultimate "why" you'd use it. To explain what kind of *complexity* is meant here, first think about how a traditional relational database works.

Contrary to what the name of relational databases indicates, not much is relational about them except that the foreign keys and primary keys are what relate tables. In contrast, relationships in graph databases are first-class citizens. Through this aspect, they lend themselves well to modeling and querying connected data. A relational database would rather strive for minimizing data redundancy. This process is known as database normalization, where a table is decomposed into smaller (less redundant) tables while maintaining all the information intact. In a normalized database one needs to conduct changes of an attribute in only one table. The aim of this process is to isolate data changes in one table. Relational database management systems (RDBMS) are a good choice as a database for data that fits nicely into a tabular format. The relationships in the data can be expressed by joining the tables. Their fit starts to downgrade when the joins become more complicated, especially when they become many-to-many joins. Query time will also increase when your data size starts increasing, and maintaining the database will be more of a challenge. These factors will hamper the performance of your database. Graph databases, on the other hand, inherently store data as nodes and relationships. Although graph databases are classified as a NoSQL type of database, a trend to present them as a category in their own right exists. One seeks the justification for this by noting that the other types of NoSQL databases are aggregation-oriented, while graph databases aren't.

A relational database might, for example, have a table representing "people" and their properties. Any person is related to other people through kinship (and friendship, and so on); each row might represent a person, but connecting them to other rows in the people table would be an immensely difficult job. Do you add a variable that holds the unique identifier of the first child and an extra one to hold the ID of the second child? Where do you stop? Tenth child?

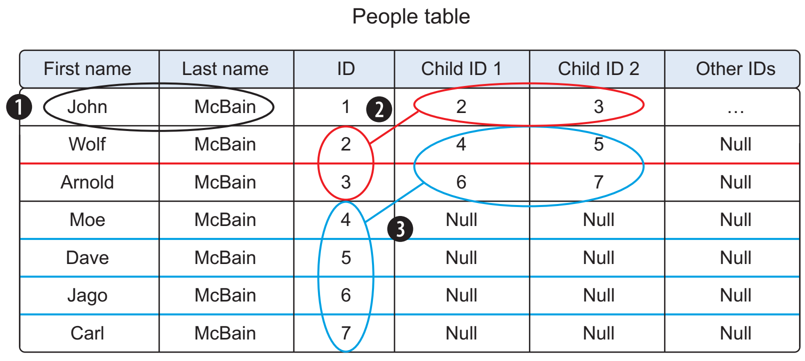
An alternative would be to use an intermediate table for child-parent relationships, but you'll need a separate one for other relationship types like friendship. In this last case you don't get column proliferation but table proliferation: one relationship table for each type of relationship. Even if you somehow succeed in modeling the data in such a way that all family relations are present, you'll need difficult queries to get the answer to simple questions such as "I would like the grandsons of John McBain." First you need to find John McBain's children. Once you find his children, you need to find theirs. By the time you have found all the grandsons, you have hit the "people" table three times:

1. Find McBain and fetch his children.
2. Look up the children with the IDs you got and get the IDs of their children.
3. Find the grandsons of McBain.

Figure 7.5 shows the recursive lookups in a relation database necessary to get from John McBain to his grandsons if everything is in a single table.

Figure 7.6 is another way to model the data: the parent-child relationship is a separate table.

Recursive lookups such as these are inefficient, to say the least.

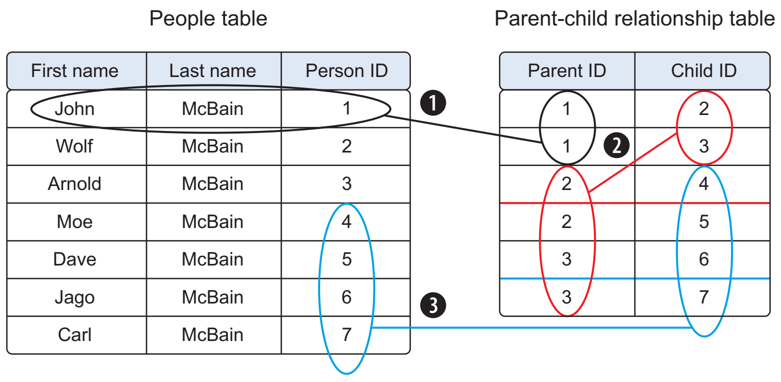


People table

First name	Last name	ID	Child ID 1	Child ID 2	Other IDs
John	McBain	1	2	3	…
Wolf	McBain	2	4	5	Null
Arnold	McBain	3	6	7	Null
Moe	McBain	4	Null	Null	Null
Dave	McBain	5	Null	Null	Null
Jago	McBain	6	Null	Null	Null
Carl	McBain	7	Null	Null	Null

1 Find John McBain **2 Use Child IDs to find Wolf and Arnold McBain** **3 Use Child IDs to find Moe, Dave, Jago, and Carl McBain**

Figure 7.5 Recursive lookup version 1: all data in one table



People table

First name	Last name	Person ID
John	McBain	1
Wolf	McBain	2
Arnold	McBain	3
Moe	McBain	4
Dave	McBain	5
Jago	McBain	6
Carl	McBain	7

Parent-child relationship table

Parent ID	Child ID
1	2
1	3
2	4
2	5
3	6
3	7

1 Find John McBain **2 Use Child IDs to find Wolf and Arnold McBain** **3 Use Child IDs to find Moe, Dave, Jago, and Carl McBain**

Figure 7.6 Recursive lookup version 2: using a parent-child relationship table

Graph databases shine when this type of *complexity* arises. Let's look at the most popular among them.

7.2 Introducing Neo4j: a graph database

Connected data is generally stored in graph databases. These databases are specifically designed to cope with the structure of connected data. The landscape of available graph databases is rather diverse these days. The three most-known ones in order of decreasing popularity are Neo4j, OrientDb, and Titan. To showcase our case study we'll choose the most popular one at the moment of writing (see http://db-engines.com/en/ranking/graph+dbms, September 2015).

Neo4j is a graph database that stores the data in a graph containing nodes and relationships (both are allowed to contain properties). This type of graph database is known as a property graph and is well suited for storing connected data. It has a flexible schema that will give us freedom to change our data structure if needed, providing us the ability to add new data and new relationships if needed. It's an open source project, mature technology, easy to install, user-friendly, and well documented. Neo4j also has a browser-based interface that facilitates the creation of graphs for visualization purposes. To follow along, this would be the right moment to install Neo4j. Neo4j can be downloaded from http://neo4j.com/download/. All necessary steps for a successful installation are summarized in appendix C.

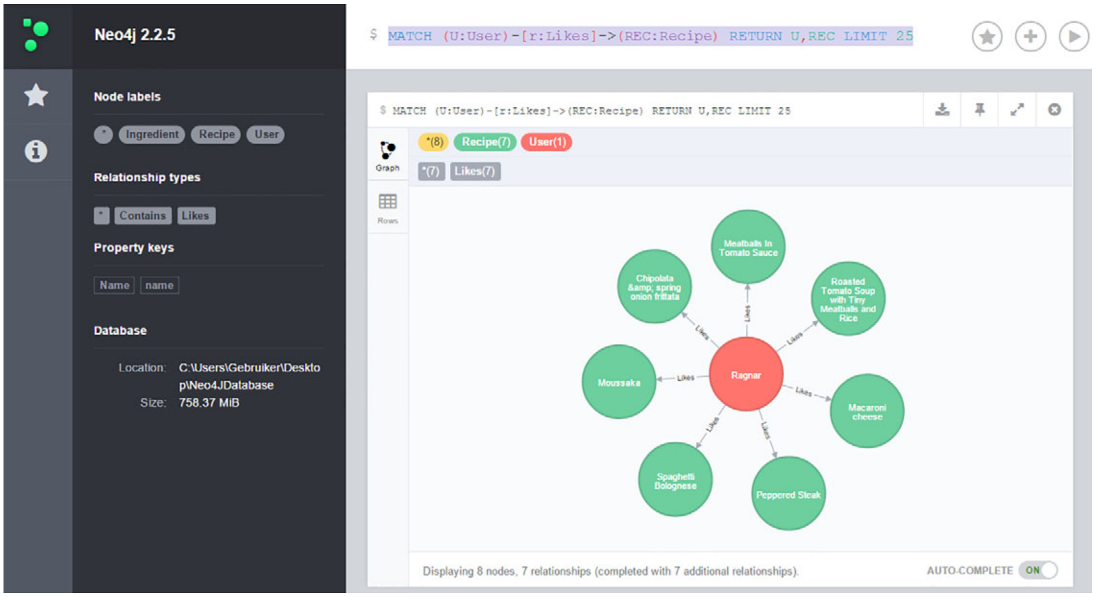
Now let's introduce the four basic structures in Neo4j:

- *Nodes*—Represent entities such as documents, users, recipes, and so on. Certain properties could be assigned to nodes.
- *Relationships*—Exist between the different nodes. They can be accessed either stand-alone or through the nodes they're attached to. Relationships can also contain properties, hence the name property graph model. Every relationship has a name and a direction, which together provide semantic context for the nodes connected by the relationship.
- *Properties*—Both nodes and relationships can have properties. Properties are defined by key-value pairs.
- *Labels*—Can be used to group similar nodes to facilitate faster traversal through graphs.

Before conducting an analysis, a good habit is to design your database carefully so it fits the queries you'd like to run down the road when performing your analysis. Graph databases have the pleasant characteristic that they're whiteboard friendly. If one tries to draw the problem setting on a whiteboard, this drawing will closely resemble the database design for the defined problem. Therefore, such a whiteboard drawing would then be a good starting point to design our database.

Now how to retrieve the data? To explore our data, we need to traverse through the graph following predefined paths to find the patterns we're searching for. The Neo4j browser is an ideal environment to create and play around with your connected data until you get to the right kind of representation for optimal queries, as shown in figure 7.7. The flexible schema of the graph database suits us well here. In this browser you can retrieve your data in rows or as a graph. Neo4j has its own query language to ease the creation and query capabilities of graphs.

Cypher is a highly expressive language that shares enough with SQL to enhance the learning process of the language. In the following section, we'll create our own data using Cypher and insert it into Neo4j. Then we can play around with the data.



```
$ MATCH (U:User)-[r:Likes]->(REC:Recipe) RETURN U,REC LIMIT 25
```

```
Location: C:\Users\Gebruiker\Desktop\Neo4JDatabase
    Size: 758.37 MiB
```

```
Displaying 8 nodes, 7 relationships (completed with 7 additional relationships).
```

Figure 7.7 Neo4j 2.2.5 interface with resolved query from the chapter case study

7.2.1 Cypher: a graph query language

Let's introduce Cypher and its basic syntax for graph operations. The idea of this section is to present enough about Cypher to get us started using the Neo4j browser. At the end of this section you should be able to create your own connected data using Cypher in the Neo4j browser and run basic queries to retrieve the results of the query. For a more extensive introduction to Cypher you can visit http://neo4j.com/docs/stable/cypher-query-lang.html. We'll start by drawing a simple social graph accompanied by a basic query to retrieve a predefined pattern as an example. In the next step we'll draw a more complex graph that will allow us to use more complicated queries in Cypher. This will help us to get acquainted with Cypher and move us down the path to bringing our use case into reality. Moreover, we'll show how to create our own simulated connected data using Cypher.

Figure 7.8 shows a simple social graph of two nodes, connected by a relationship of type "knows". The nodes have both the properties "name" and "lastname".

Now, if we'd like to find out the following pattern, "Who does Paul know?" we'd query this using Cypher. To find a pattern in Cypher, we'll start with a `Match` clause. In

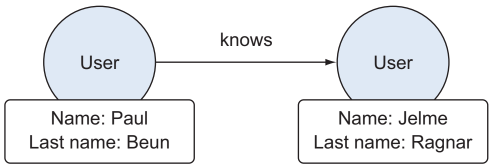


Figure 7.8 An example of a simple social graph with two users and one relationship

this query we'll start searching at the node User with the name property "Paul". Note how the node is enclosed within parentheses, as shown in the code snippet below, and the relationship is enclosed by square brackets. Relationships are named with a colon (:) prefix, and the direction is described using arrows. The placeholder p2 will contain all the User nodes having the relationship of type "knows" as an inbound relationship. With the return clause we can retrieve the results of the query.

```
 Match(p1:User { name: 'Paul' } )-[:knows]->(p2:User)
Return p2.name
```

Notice the close relationship of how we have formulated our question verbally and the way the graph database translates this into a traversal. In Neo4j, this impressive expressiveness is made possible by its graph query language, Cypher.

To make the examples more interesting, let's assume that our data is represented by the graph in figure 7.9.

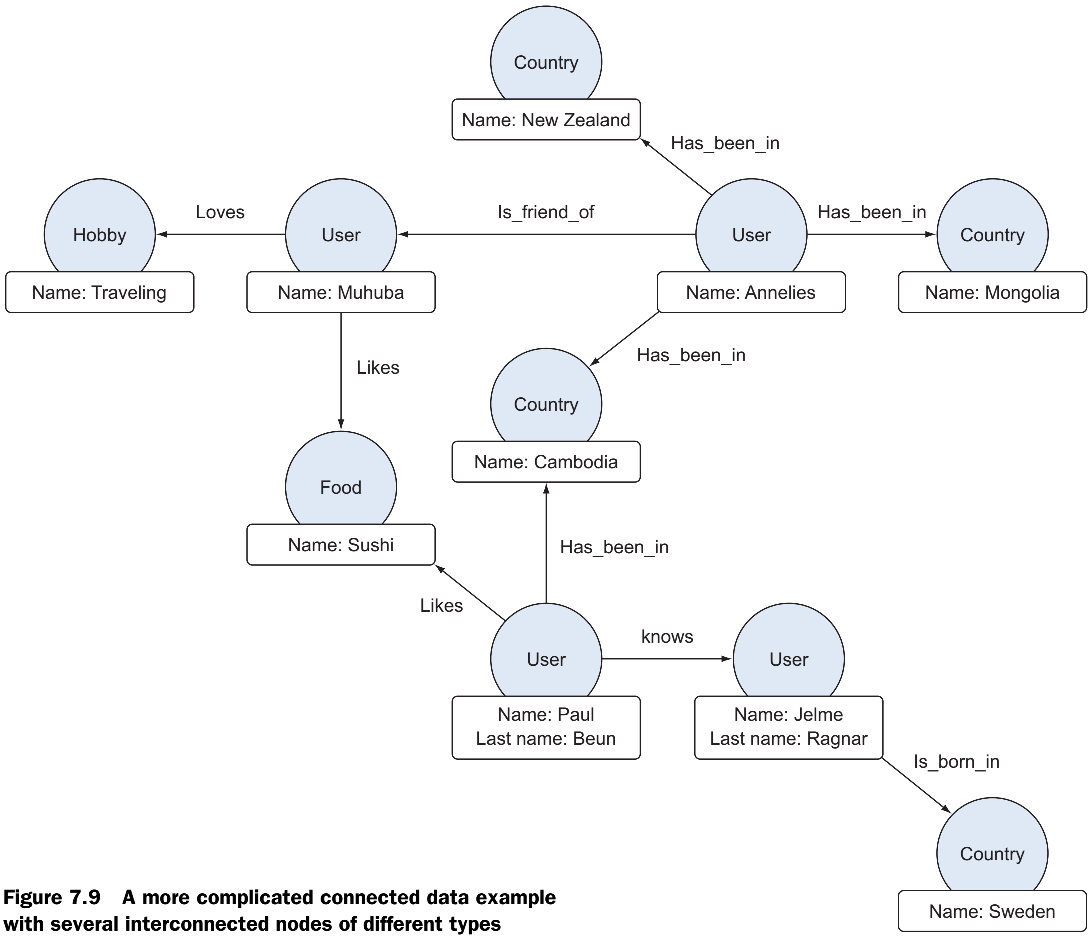


Figure 7.9 A more complicated connected data example with several interconnected nodes of different types

We can insert the connected data in figure 7.9 into Neo4j by using Cypher. We can write Cypher commands directly in the browser-based interface of Neo4j, or alternatively through a Python driver (see http://neo4j.com/developer/python/ for an overview). This is a good way to get a hands-on feeling with connected data and graph databases.

To write an appropriate create statement in Cypher, first we should have a good understanding of which data we'd like to store as nodes and which as relationships, what their properties should be, and whether labels would be useful. The first decision is to decide which data should be regarded as nodes and which as relationships to provide a semantic context for these nodes. In figure 7.9 we've chosen to represent the users and countries they have been in as nodes. Data that provides information about a specific node, for example a name that's associated with a node, can be represented as a property. All data that provides context about two or more nodes will be considered as a relationship. Nodes that share common features, for example Cambodia and Sweden are both countries, will also be grouped through labels. In figure 7.9 this is already done.

In the following listing we demonstrate how the different objects could be encoded in Cypher through one big create statement. *Be aware that Cypher is case sensitive.*

Listing 7.1 Cypher data creation statement

```
CREATE (user1:User {name :'Annelies'}),
 (user2:User {name :'Paul' , LastName: 'Beun'}),
 (user3:User {name :'Muhuba'}),
 (user4:User {name : 'Jelme' , LastName: 'Ragnar'}),
 (country1:Country { name:'Mongolia'}),
 (country2:Country { name:'Cambodia'}),
 (country3:Country { name:'New Zealand'}),
 (country4:Country { name:'Sweden'}),
 (food1:Food { name:'Sushi' }),
 (hobby1:Hobby { name:'Travelling'}),
 (user1)-[:Has_been_in]->(country1),
 (user1)-[: Has_been_in]->(country2),
 (user1)-[: Has_been_in]->(country3),
 (user2)-[: Has_been_in]->(country2),
 (user1)-[: Is_mother_of]->(user4),
 (user2)-[: knows]->(user4),
 (user1)-[: Is_friend_of]->(user3),
 (user2)-[: Likes]->( food1),
 (user3)-[: Likes]->( food1),
 (user4)-[: Is_born_in]->(country4)
```

Running this create statement in one go has the advantage that the success of this execution will ensure us that the graph database has been successfully created. If an error exists, the graph won't be created.

In a real scenario, one should also define indexes and constraints to ensure a fast lookup and not search the entire database. We haven't done this here because our simulated data set is small. However, this can be easily done using Cypher. Consult the

Cypher documentation to find out more about indexes and constraints (http://neo4j.com/docs/stable/cypherdoc-labels-constraints-and-indexes.html). Now that we've created our data, we can query it. The following query will return all nodes and relationships in the database:

```
MATCH (n)-[r]-()          <-- Find all nodes (n) and all their relationships [r].
RETURN n,r                <-- Show all nodes n and all relationships r.
```

Figure 7.10 shows the database that we've created. We can compare this graph with the graph we've envisioned on our whiteboard. On our whiteboard we grouped nodes of people in a label "User" and nodes of countries in a label "Country". Although the nodes in this figure aren't represented by their labels, the labels are present in our database. Besides that, we also miss a node (Hobby) and a relationship of type "Loves". These can be easily added through a merge statement that will create the node and relationship if they don't exist already:

```
Merge (user3)-[: Loves]->( hobby1)
```

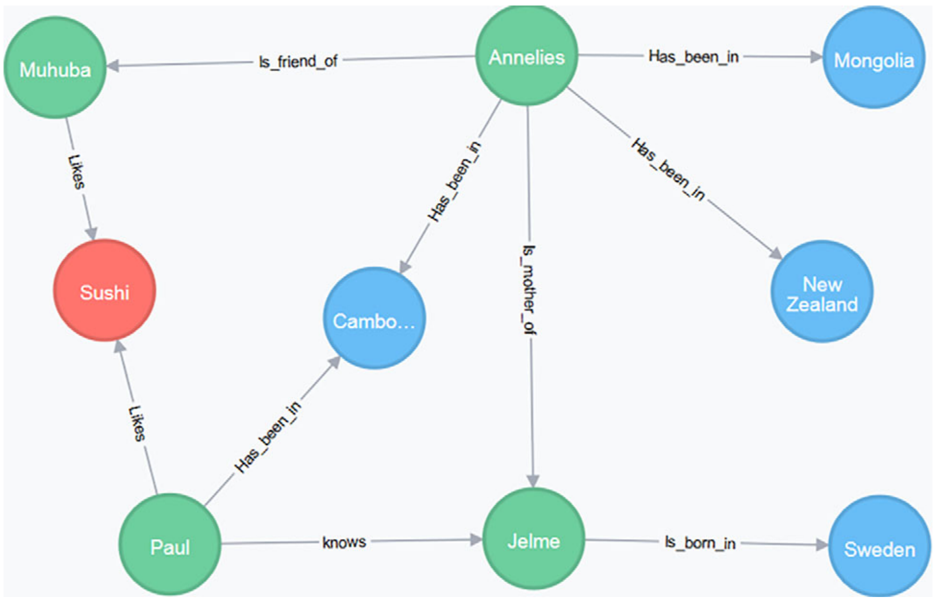


Figure 7.10 The graph drawn in figure 7.9 now has been created in the Neo4j web interface. The nodes aren't represented by their labels but by their names. We can infer from the graph that we're missing the label *Hobby* with the name *Traveling*. The reason for this is because we have forgotten to include this node and its corresponding relationship in the create statement.

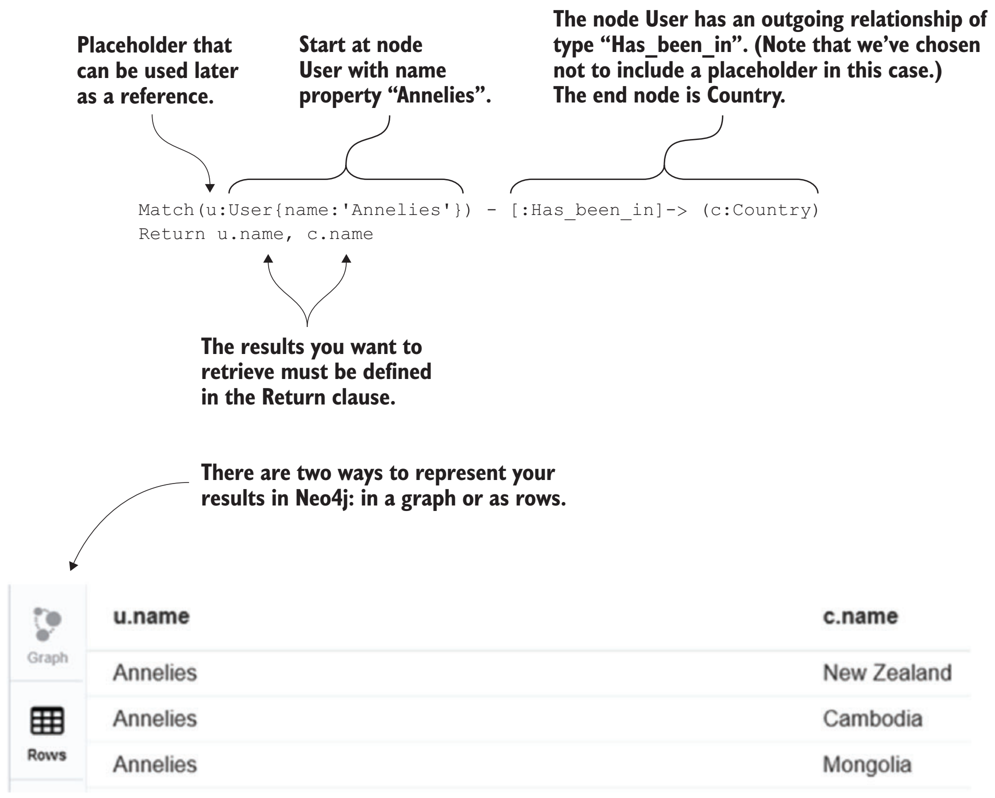
We can ask many questions here. For example:

- Question 1: Which countries has Annelies visited? The Cypher code to create the answer (shown in figure 7.11) is

```
Match(u:User{name:'Annelies'}) - [:Has_been_in]-> (c:Country)
Return u.name, c.name
```

- Question 2: Who has been where? The Cypher code (explained in figure 7.12) is

```
Match ()-[r: Has_been_in]->()
Return r LIMIT 25
```


Placeholder that can be used later as a reference.
Start at node User with name property "Annelies".
The node User has an outgoing relationship of type "Has_been_in". (Note that we've chosen not to include a placeholder in this case.) The end node is Country.
The results you want to retrieve must be defined in the Return clause.
There are two ways to represent your results in Neo4j: in a graph or as rows.
Graph
Rows

```
Match(u:User{name:'Annelies'}) - [:Has_been_in]-> (c:Country)
Return u.name, c.name
```

u.name	c.name
Annelies	New Zealand
Annelies	Cambodia
Annelies	Mongolia

Figure 7.11 Results of question 1: Which countries has Annelies visited? We can see the three countries Annelies has been in, using the row presentation of Neo4j. The traversal took only 97 milliseconds.

This query is asking for all nodes with an outgoing relationship with the type "Has_been_in".

```
MATCH ()-[r:Has_been_in]->()
RETURN r LIMIT 25
```

The end nodes are all nodes with an incoming relationship of the type "Has_been_in".

Figure 7.12 Who has been where? Query buildup explained.

When we run this query we get the answer shown in figure 7.13.

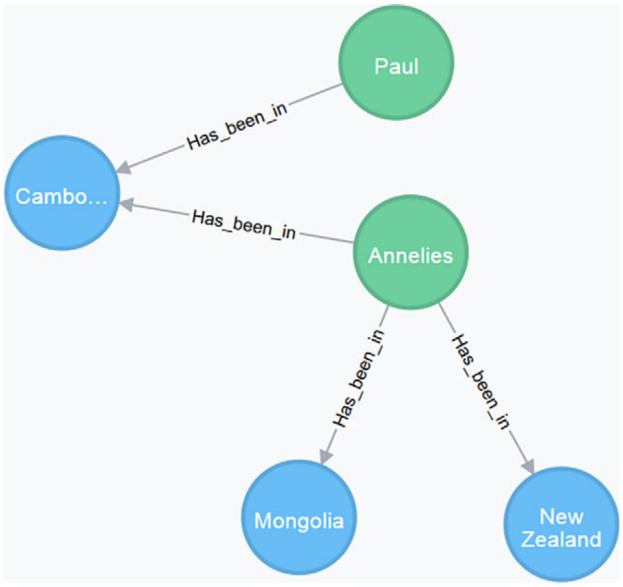


Figure 7.13 Results of question 2: Who has been where? The results of our traversal are now shown in the graph representation of Neo4j. Now we can see that Paul, in addition to Annelies, has also been to Cambodia.

In question 2 we have chosen not to specify a start node. Therefore, Cypher will go to all nodes present in the database to find those with an outgoing relationship of type "Has_been_in". One should avoid not specifying a starting node since, depending on the size of your database, such a query could take a long time to converge. Playing around with the data to obtain the right graph database also means a lot of data deletion. Cypher has a delete statement suitable for deleting small amounts of data. The following query demonstrates how to delete all nodes and relationships in the database:

```
MATCH(n)
Optional MATCH (n)-[r]-()
Delete n,r
```

Now that we're acquainted with connected data and have basic knowledge of how it's managed in a graph database, we can go a step further and look into real, live applications of connected data. A social graph, for example, can be used to find clusters of tightly connected nodes inside the graph communities. People in a cluster who don't know each other can then be introduced to each other. The concept of searching for tightly connected nodes, nodes that have a significant amount of features in common, is a widely used concept. In the next section we'll use this idea, where the aim will be to find clusters inside an ingredient network.

7.3 Connected data example: a recipe recommendation engine

One of the most popular use cases for graph databases is the development of recommender engines. Recommender engines became widely adopted through their promise to create relevant content. Living in an era with such abundance in data can be overwhelming to many consumers. Enterprises saw the clear need to be inventive in how to attract customers through personalized content, thereby using the strengths of recommender engines.

In our case study we'll recommend recipes based on the dish preferences of users and a network of ingredients. During data preparation we'll use Elasticsearch to quicken the process and allow for more focus on the actual graph database. Its main purpose here will be to replace the ingredients list of the "dirty" downloaded data with the ingredients from our own "clean" list.

If you skipped ahead to this chapter, it might be good to at least read appendix A on installing Elasticsearch so you have it running on your computer. You can always download the index we'll use from the Manning download page for this chapter and paste it into your local Elasticsearch data directory if you don't feel like bothering with the chapter 6 case study.

You can download the following information from the Manning website for this chapter:

Three .py code files and their .ipynb counterparts

- *Data Preparation Part 1*—Will upload the data to Elasticsearch (alternatively you can paste the downloadable index in your local Elasticsearch data folder)
- *Data Preparation Part 2*—Will move the data from Elasticsearch to Neo4j
- Exploration & Recommender System

Three data files

- *Ingredients (.txt)*—Self-compiled ingredients file
- *Recipes (.json)*—Contains all the ingredients
- *Elasticsearch index (.zip)*—Contains the "gastronomical" Elasticsearch index you can use to skip data preparation part 1

Now that we have everything we need, let's look at the research goal and the steps we need to take to achieve it.

7.3.1 Step 1: Setting the research goal

Let's look at what's to come when we follow the data science process (figure 7.14).

Our primary goal is to set up a recommender engine that would help users of a cooking website find the right recipe. A user gets to like several recipes and we'll base

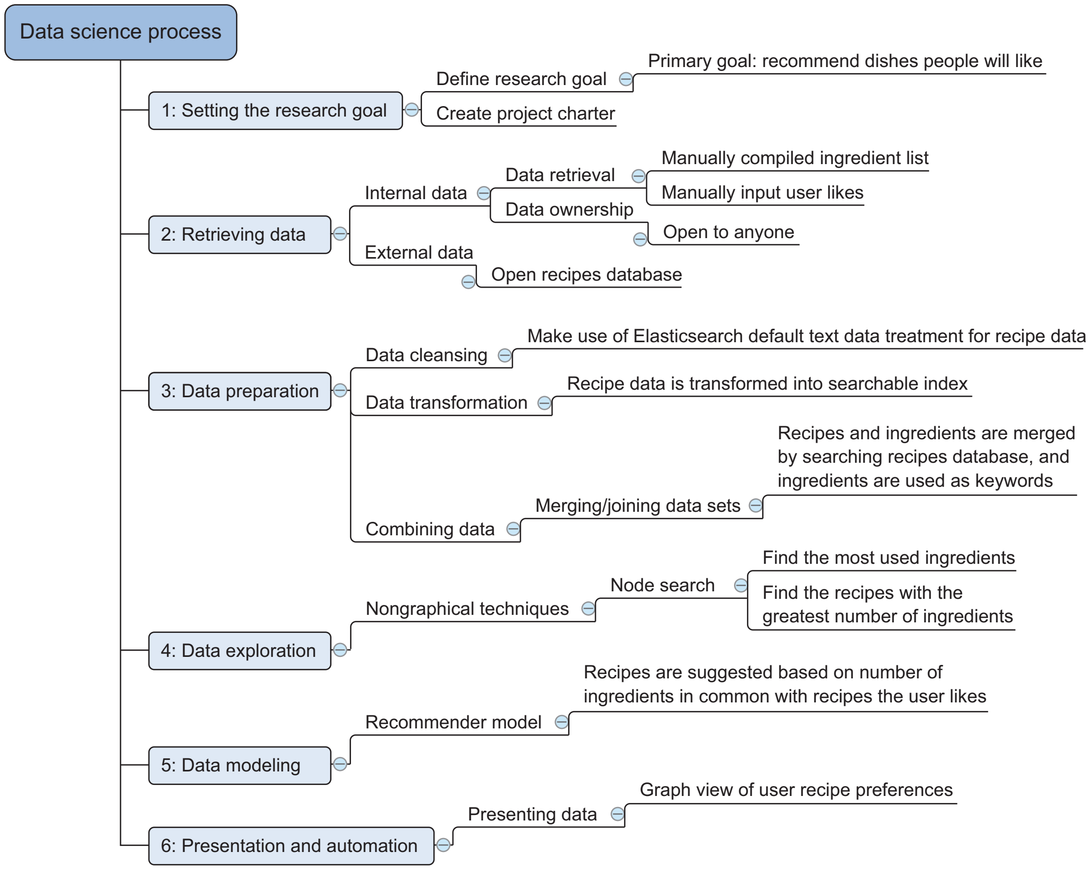

Data science process
1: Setting the research goal
Define research goal
Primary goal: recommend dishes people will like
Create project charter
2: Retrieving data
Internal data
Data retrieval
Manually compiled ingredient list
Manually input user likes
Data ownership
Open to anyone
External data
Open recipes database
3: Data preparation
Data cleansing
Make use of Elasticsearch default text data treatment for recipe data
Data transformation
Recipe data is transformed into searchable index
Combining data
Merging/joining data sets
Recipes and ingredients are merged by searching recipes database, and ingredients are used as keywords
4: Data exploration
Nongraphical techniques
Node search
Find the most used ingredients
Find the recipes with the greatest number of ingredients
5: Data modeling
Recommender model
Recipes are suggested based on number of ingredients in common with recipes the user likes
6: Presentation and automation
Presenting data
Graph view of user recipe preferences

Figure 7.14 Data science process overview applied to connected data recommender model

our dish recommendations on the ingredients' overlap in a recipes network. This is a simple and intuitive approach, yet already yields fairly accurate results. Let's look at the three data elements we require.

7.3.2 Step 2: Data retrieval

For this exercise we require three types of data:

- Recipes and their respective ingredients
- A list of distinct ingredients we like to model
- At least one user and his preference for certain dishes

As always, we can divide this into internally available or created data and externally acquired data.

- *Internal data*—We don't have any *user preferences* or ingredients lying around, but these are the smallest part of our data and easily created. A few manually input preferences should be enough to create a recommendation. The user gets more interesting and accurate results the more feedback he gives. We'll input user preferences later in the case study. A *list of ingredients* can be manually compiled and will remain relevant for years to come, so feel free to use the list in the downloadable material for any purpose, commercially or otherwise.
- *External data—Recipes* are a different matter. Thousands of ingredients exist, but these can be combined into millions of dishes. We are in luck, however, because a pretty big list is freely available at https://github.com/fictivekin/openrecipes. Many thanks to Fictive Kin for this valuable data set with more than a hundred thousand recipes. Sure there are duplicates in here, but they won't hurt our use case that badly.

We now have two data files at our disposal: a list of 800+ ingredients (ingredients.txt) and more than a hundred thousand recipes in the recipes.json file. A sample of the ingredients list can be seen in the following listing.

Listing 7.2 Ingredients list text file sample

```
Ditalini
Egg Noodles
Farfalle
Fettuccine
Fusilli
Lasagna
Linguine
Macaroni
Orzo
```

The "openrecipes" JSON file contains more than a hundred thousand recipes with multiple properties such as publish date, source location, preparation time, description, and so on. We're only interested in the name and ingredients list. A sample recipe is shown in the following listing.

Listing 7.3 A sample JSON recipe

```
{ "_id" : { "$oid" : "5160756b96cc62079cc2db15" },
    "name" : "Drop Biscuits and Sausage Gravy",
    "ingredients" : "Biscuits\n3 cups All-purpose Flour\n2 Tablespoons Baking
     Powder\n1/2 teaspoon Salt\n1-1/2 stick (3/4 Cup) Cold Butter, Cut Into
     Pieces\n1-1/4 cup Butermilk\n SAUSAGE GRAVY\n1 pound Breakfast Sausage,
     Hot Or Mild\n1/3 cup All-purpose Flour\n4 cups Whole Milk\n1/2 teaspoon
     Seasoned Salt\n2 teaspoons Black Pepper, More To Taste",
    "url" : "http://thepioneerwoman.com/cooking/2013/03/drop-biscuits-and-
     sausage-gravy/",
    "image" : "http://static.thepioneerwoman.com/cooking/files/2013/03/
     bisgrav.jpg",
    "ts" : { "$date" : 1365276011104 },
    "cookTime" : "PT30M",
    "source" : "thepioneerwoman",
    "recipeYield" : "12",
    "datePublished" : "2013-03-11",
    "prepTime" : "PT10M",
    "description" : "Late Saturday afternoon, after Marlboro Man had returned
     home with the soccer-playing girls, and I had returned home with the..."
}
```

Because we're dealing with text data here, the problem is two-fold: first, preparing the textual data as described in the text mining chapter. Then, once the data is thoroughly cleansed, it can be used to produce recipe recommendations based on a network of ingredients. This chapter doesn't focus on the text data preparation because this is described elsewhere, so we'll allow ourselves the luxury of a shortcut during the upcoming data preparation.

7.3.3 Step 3: Data preparation

We now have two data files at our disposal, and we need to combine them into one graph database. The "dirty" recipes data poses a problem that we can address using our clean ingredients list and the use of the search engine and NoSQL database Elasticsearch. We already relied on Elasticsearch in a previous chapter and now it will clean the recipe data for us implicitly when it creates an index. We can then search this data to link each ingredient to every recipe in which it occurs. We could clean the text data using pure Python, as we did in the text mining chapter, but this shows it's good to be aware of the strong points of each NoSQL database; don't pin yourself to a single technology, but use them together to the benefit of the project.

Let's start by entering our recipe data into Elasticsearch. If you don't understand what's happening, please check the case study of chapter 6 again and it should become clear. Make sure to turn on your local Elasticsearch instance and activate a Python environment with the Elasticsearch module installed before running the code snippet in the following listing. It's recommended not to run this code "as is" in Ipython (or Jupyter) because it prints every recipe key to the screen and your browser can handle only so much output. Either turn off the print statements or run in another Python IDE. The code in this snippet can be found in "Data Preparation Part 1.py".

Listing 7.4 Importing recipe data into Elasticsearch

```
from elasticsearch import Elasticsearch
import json
```

Import modules.

```
client = Elasticsearch ()
indexName = "gastronomical"
docType = 'recipes'
```

Elasticsearch client used to communicate with database.

```
client.indices.create(index=indexName)
```

Create index.

```
file_name = 'C:/Users/Gebruiker/Downloads/recipes.json'
```

Location of JSON recipe file: change this to match your own setup!

```
recipeMapping = {
        'properties': {
            'name': {'type': 'string'},
            'ingredients': {'type': 'string'}
        }
    }
```

Mapping for Elasticsearch "recipe" doctype.

```
client.indices.put_mapping(index=indexName,doc_type=docType,body=recipeMapping )

with open(file_name, encoding="utf8") as data_file:
    recipeData = json.load(data_file)

for recipe in recipeData:
    print recipe.keys()
    print recipe['_id'].keys()
    client.index(index=indexName,
        doc_type=docType,id = recipe['_id']['$oid'],
        body={"name": recipe['name'], "ingredients":recipe['ingredients']})
```

Load JSON recipe file into memory. Another way to do this would be:
recipeData = []
with open(file_name) as f:
for line in f:
recipeData.append(json.loads(line))

Index recipes. Only name and ingredients are important for our use case. In case a timeout problem occurs it's possible to increase the timeout delay by specifying, for example, timeout=30 as an argument.

If everything went well, we now have an Elasticsearch index by the name "gastronomical" populated by thousands of recipes. Notice we allowed for duplicates of the same recipe by not assigning the name of the recipe to be the document key. If, for instance, a recipe is called "lasagna" then this can be a salmon lasagna, beef lasagna, chicken lasagna, or any other type. No single recipe is selected as the prototype lasagna; they are all uploaded to Elasticsearch under the same name: "lasagna". This is a choice, so feel free to decide otherwise. It will have a significant impact, as we'll see later on. The door is now open for a systematic upload to our local graph database. Make sure your local graph database instance is turned on when applying the following code. Our username for this database is the default Neo4j and the password is Neo4ja; make sure to adjust this for your local setup. For this we'll also require a Neo4j-specific Python library called py2neo. If you haven't already, now would be the time to install it to your virtual environment using `pip install py2neo` or `conda install py2neo` when using Anaconda. Again, be advised this code will crash your browser when run directly in Ipython or Jupiter. The code in this listing can be found in "Data Preparation Part 2.py".

Listing 7.5 Using the Elasticsearch index to fill the graph database

```
from elasticsearch import Elasticsearch
from py2neo import Graph, authenticate, Node, Relationship
```

Import modules

```
client = Elasticsearch ()
indexName = "gastronomical"
docType = 'recipes'
```

Elasticsearch client used to communicate with database

```
authenticate("localhost:7474", "user", "password")
graph_db = Graph("http://localhost:7474/db/data/")
```

Authenticate with your own username and password

Graph database entity

```
filename = 'C:/Users/Gebruiker/Downloads/ingredients.txt'
ingredients =[]
with open(filename) as f:
    for line in f:
        ingredients.append(line.strip())

print ingredients
```

Ingredients text file gets loaded into memory

Strip because of the /n you get otherwise from reading the .txt

```
ingredientnumber = 0
grandtotal = 0
for ingredient in ingredients:
```

Loop through ingredients and fetch Elasticsearch result

```
    try:
        IngredientNode = graph_db.merge_one("Ingredient","Name",ingredient)
    except:
        continue

    ingredientnumber +=1
    searchbody = {
        "size" : 99999999,
        "query": {
            "match_phrase":
                {
                    "ingredients":{
```

Create node in graph database for current ingredient

Phrase matching used, as some ingredients consist of multiple words

```
                            "query":ingredient,
                        }
                    }
            }
    }
    result = client.search(index=indexName,doc_type=docType,body=searchbody)

    print ingredient
    print ingredientnumber
    print "total: " +  str(result['hits']['total'])

    grandtotal = grandtotal + result['hits']['total']
    print "grand total: " +  str(grandtotal)

    for recipe in result['hits']['hits']:          <- Loop through recipes found for this particular ingredient

        try:
            RecipeNode =
     graph_db.merge_one("Recipe","Name",recipe['_source']['name'])     <- Create node for each recipe that is not already in graph database
            NodesRelationship = Relationship(RecipeNode, "Contains",
     IngredientNode)
            graph_db.create_unique(NodesRelationship)     <- Create relationship between this recipe and ingredient
            print "added: " + recipe['_source']['name'] + " contains " +
     ingredient

        except:
            continue

    print "*************************************"
```

Great, we're now the proud owner of a graph database filled with recipes! It's time for connected data exploration.

7.3.4 Step 4: Data exploration

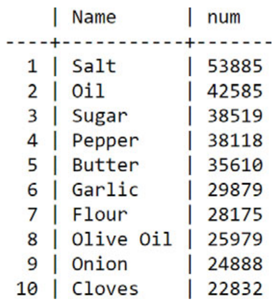
Now that we have our data where we want it, we can manually explore it using the Neo4j interface at http://localhost:7474/browser/.

Nothing stops you from running your Cypher code in this environment, but Cypher can also be executed via the py2neo library. One interesting question we can pose is which ingredients are occurring the most over all recipes? What are we most likely to get into our digestive system if we randomly selected and ate dishes from this database?

```
from py2neo import Graph, authenticate, Node, Relationship
authenticate("localhost:7474", "user", "password")
graph_db = Graph("http://localhost:7474/db/data/")graph_db.cypher.execute("
   MATCH (REC:Recipe)-[r:Contains]->(ING:Ingredient) WITH ING, count(r) AS num
   RETURN ING.Name as Name, num ORDER BY num DESC LIMIT 10;")
```

The query is created in Cypher and says: for all the recipes and their ingredients, count the number of relations per ingredient and return the ten ingredients with the most relations and their respective counts. The results are shown in figure 7.15.

Most of the top 10 list in figure 7.15 shouldn't come as a surprise. With salt proudly at the top of our list, we shouldn't be shocked to find vascular diseases as the number one killer in most western countries. Another interesting question that comes to mind now is from a different perspective: which recipes require the most ingredients?

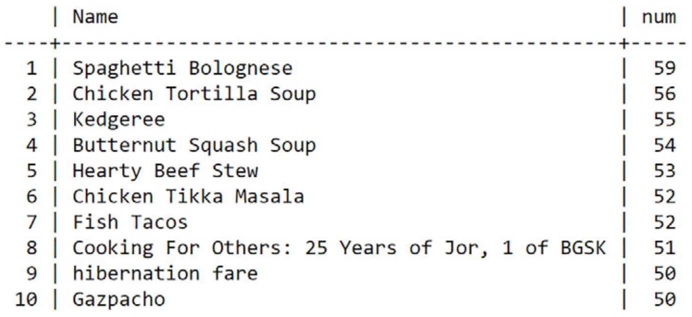


```
    | Name       | num
----+------------+-------
  1 | Salt       | 53885
  2 | Oil        | 42585
  3 | Sugar      | 38519
  4 | Pepper     | 38118
  5 | Butter     | 35610
  6 | Garlic     | 29879
  7 | Flour      | 28175
  8 | Olive Oil  | 25979
  9 | Onion      | 24888
 10 | Cloves     | 22832
```

Figure 7.15 Top 10 ingredients that occur in the most recipes

```
from py2neo import Graph, Node, Relationship
graph_db = Graph("http://neo4j:neo4ja@localhost:7474/db/data/")
graph_db.cypher.execute("
    MATCH (REC:Recipe)-[r:Contains]->(ING:Ingredient) WITH REC, count(r) AS num
    RETURN REC.Name as Name, num ORDER BY num DESC LIMIT 10;")
```

The query is almost the same as before, but instead of returning the ingredients, we demand the recipes. The result is figure 7.16.



```
    | Name                                         | num
----+----------------------------------------------+-----
  1 | Spaghetti Bolognese                          |  59
  2 | Chicken Tortilla Soup                        |  56
  3 | Kedgeree                                     |  55
  4 | Butternut Squash Soup                        |  54
  5 | Hearty Beef Stew                             |  53
  6 | Chicken Tikka Masala                         |  52
  7 | Fish Tacos                                   |  52
  8 | Cooking For Others: 25 Years of Jor, 1 of BGSK |  51
  9 | hibernation fare                             |  50
 10 | Gazpacho                                     |  50
```

Figure 7.16 Top 10 dishes that can be created with the greatest diversity of ingredients

Now this might be a surprising sight. Spaghetti Bolognese hardly sounds like the type of dish that would require 59 ingredients. Let's take a closer look at the ingredients listed for Spaghetti Bolognese.

```
from py2neo import Graph, Node, Relationship
graph_db = Graph("http://neo4j:neo4ja@localhost:7474/db/data/")
graph_db.cypher.execute("MATCH (REC1:Recipe{Name:'Spaghetti Bolognese'})-
     [r:Contains]->(ING:Ingredient) RETURN REC1.Name, ING.Name;")
```

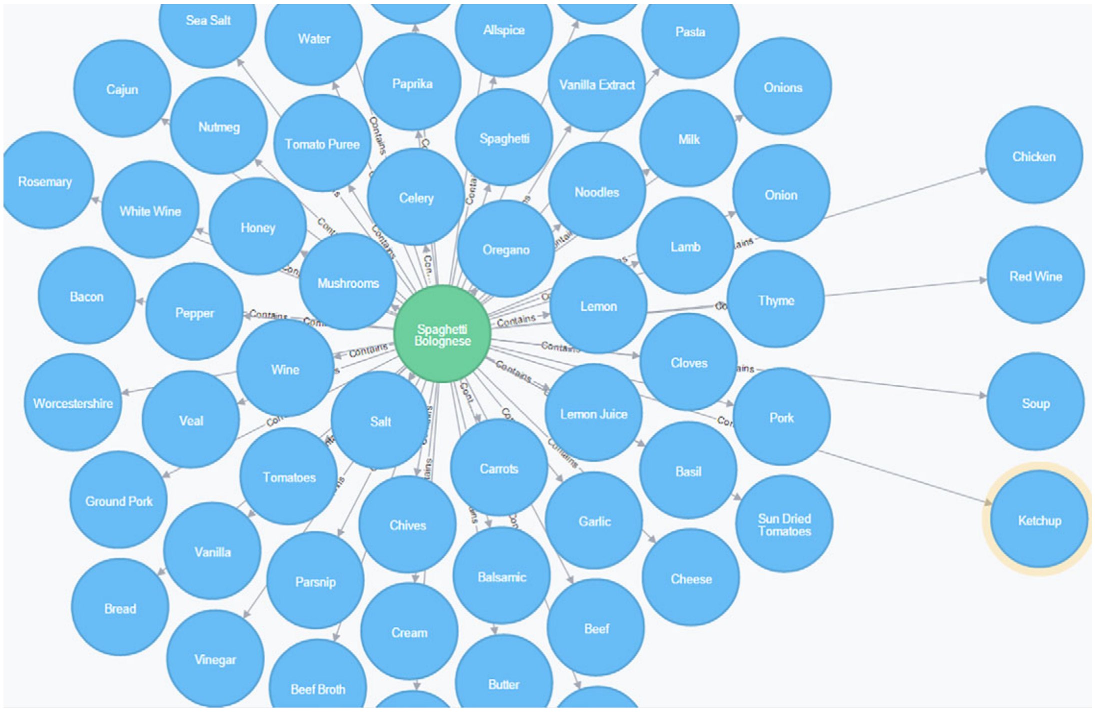


Figure 7.17 Spaghetti Bolognese possible ingredients

The Cypher query merely lists the ingredients linked to Spaghetti Bolognese. Figure 7.17 shows the result in the Neo4j web interface.

Let's remind ourselves of the remark we made when indexing the data in Elasticsearch. A quick Elasticsearch search on Spaghetti Bolognese shows us it occurs multiple times, and all these instances were used to link ingredients to Spaghetti Bolognese as a recipe. We don't have to look at Spaghetti Bolognese as a single recipe but more as a collection of ways people create their own "Spaghetti Bolognese." This makes for an interesting way to look at this data. People can create their version of the dish with ketchup, red wine, and chicken or they might even add soup. With "Spaghetti Bolognese" as a dish being so open to interpretation, no wonder so many people love it.

The Spaghetti Bolognese story was an interesting distraction but not what we came for. It's time to recommend dishes to our gourmand "Ragnar".

7.3.5 Step 5: Data modeling

With our knowledge of the data slightly enriched, we get to the goal of this exercise: the recommendations.

For this we introduce a user we call "Ragnar," who likes a couple of dishes. This new information needs to be absorbed by our graph database before we can expect it to suggest new dishes. Therefore, let's now create Ragnar's user node with a few recipe preferences.

Listing 7.6 Creating a user node who likes certain recipes in the Neo4j graph database

```
from py2neo import Graph, Node, Relationship

graph_db = Graph("http://neo4j:neo4ja@localhost:7474/db/data/")

UserRef = graph_db.merge_one("User","Name","Ragnar")

RecipeRef = graph_db.find_one("Recipe",property_key="Name",
     property_value="Spaghetti Bolognese")
NodesRelationship = Relationship(UserRef, "Likes", RecipeRef)
graph_db.create_unique(NodesRelationship) #Commit his like to database

graph_db.create_unique(Relationship(UserRef, "Likes",
     graph_db.find_one("Recipe",property_key="Name",
     property_value="Roasted Tomato Soup with Tiny Meatballs
     and Rice")))
graph_db.create_unique(Relationship(UserRef, "Likes",
     graph_db.find_one("Recipe",property_key="Name",
     property_value="Moussaka")))
graph_db.create_unique(Relationship(UserRef, "Likes",
     graph_db.find_one("Recipe",property_key="Name",
     property_value="Chipolata &amp; spring onion frittata")))
graph_db.create_unique(Relationship(UserRef, "Likes",
     graph_db.find_one("Recipe",property_key="Name",
     property_value="Meatballs In Tomato Sauce")))
graph_db.create_unique(Relationship(UserRef, "Likes",
     graph_db.find_one("Recipe",property_key="Name",
     property_value="Macaroni cheese")))
graph_db.create_unique(Relationship(UserRef, "Likes",
     graph_db.find_one("Recipe",property_key="Name",
     property_value="Peppered Steak")))
```

Import modules

Make graph database connection object

Create new user called "Ragnar"

Find recipe by the name of Spaghetti Bolognese

Ragnar likes Spaghetti Bolognese

Create a like relationship between Ragnar and the spaghetti

Repeat the same process as in the lines above but for several other dishes

In listing 7.6 our food connoisseur Ragnar is added to the database along with his preference for a few dishes. If we select Ragnar in the Neo4j interface, we get figure 7.18. The Cypher query for this is

```
MATCH (U:User)-[r:Likes]->(REC:Recipe) RETURN U,REC LIMIT 25
```

No surprises in figure 7.18: many people like Spaghetti Bolognese, and so does our Scandinavian gastronomist Ragnar.

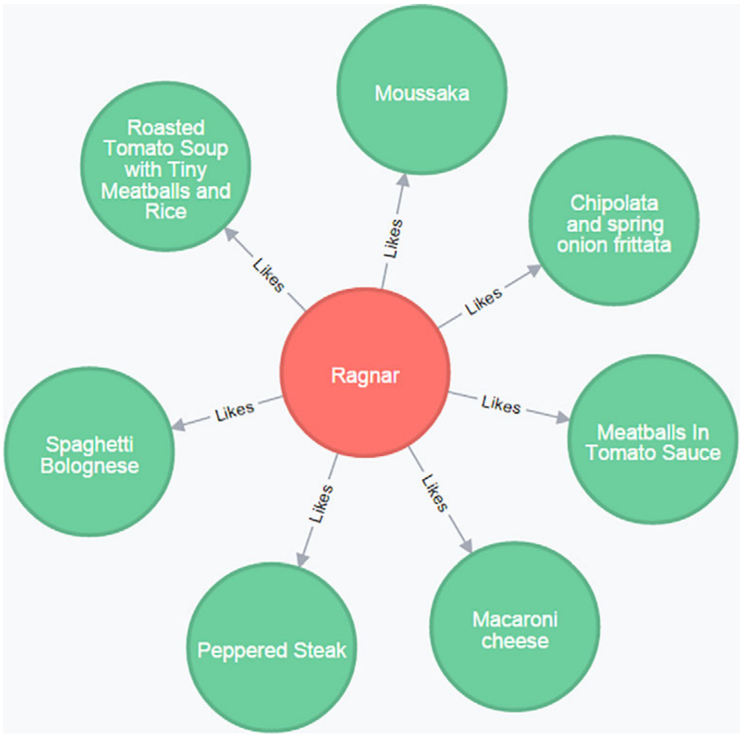

Moussaka
Roasted Tomato Soup with Tiny Meatballs and Rice
Chipolata and spring onion frittata
Ragnar
Spaghetti Bolognese
Meatballs In Tomato Sauce
Peppered Steak
Macaroni cheese
Likes

Figure 7.18 The user Ragnar likes several dishes

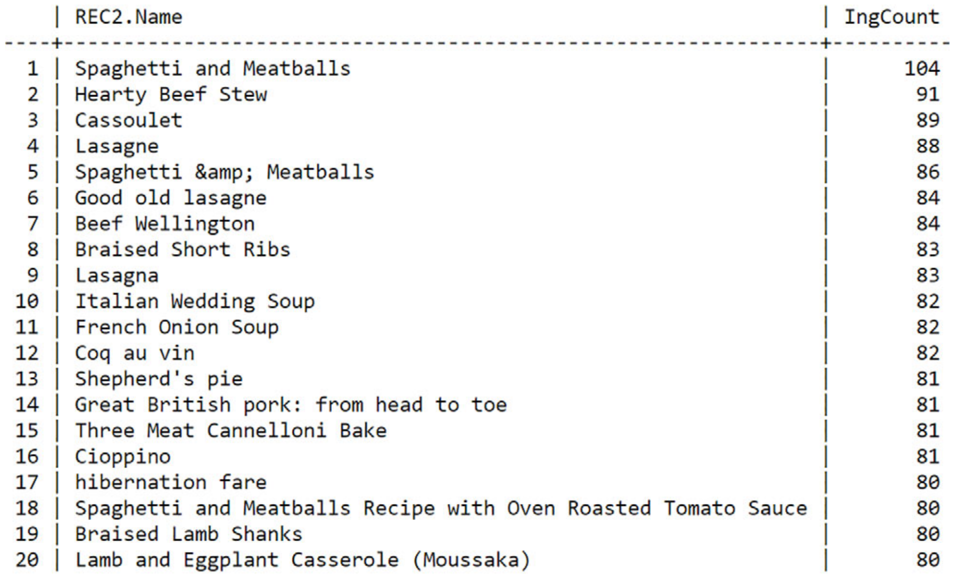
For the simple recommendation engine we like to build, all that's left for us to do is ask the graph database to give us the nearest dishes in terms of ingredients. Again, this is a basic approach to recommender systems because it doesn't take into account factors such as

- Dislike of an ingredient or a dish.
- The amount of like or dislike. A score out of 10 instead of a binary like or don't like could make a difference.
- The amount of the ingredient that is present in the dish.
- The threshold for a certain ingredient to become apparent in its taste. Certain ingredients, such as spicy pepper, will represent a bigger impact for a smaller dose than other ingredients would.
- Food allergies. While this will be implicitly modeled in the like or dislike of dishes with certain ingredients, a food allergy can be so important that a single mistake can be fatal. Avoidance of allergens should overwrite the entire recommendation system.
- Many more things for you to ponder about.

It might come as a bit of a surprise, but a single Cypher command will suffice.

```
from py2neo import Graph, Node, Relationship
graph_db = Graph("http://neo4j:neo4ja@localhost:7474/db/data/")
graph_db.cypher.execute("
   MATCH (USR1:User{Name:'Ragnar'})-[l1:Likes]->(REC1:Recipe),
         (REC1)-[c1:Contains]->(ING1:Ingredient)
      WITH  ING1,REC1 MATCH (REC2:Recipe)-[c2:Contains]->(ING1:Ingredient)
      WHERE REC1 <> REC2
   RETURN REC2.Name,count(ING1) AS IngCount ORDER BY IngCount DESC LIMIT 20;")
```

First all recipes that Ragnar likes are collected. Then their ingredients are used to fetch all the other dishes that share them. The ingredients are then counted for each connected dish and ranked from many common ingredients to few. Only the top 20 dishes are kept; this results in the table of figure 7.19.



```
    | REC2.Name                                                 | IngCount
----+-----------------------------------------------------------+----------
  1 | Spaghetti and Meatballs                                   |      104
  2 | Hearty Beef Stew                                          |       91
  3 | Cassoulet                                                 |       89
  4 | Lasagne                                                   |       88
  5 | Spaghetti &amp; Meatballs                                 |       86
  6 | Good old lasagne                                          |       84
  7 | Beef Wellington                                           |       84
  8 | Braised Short Ribs                                        |       83
  9 | Lasagna                                                   |       83
 10 | Italian Wedding Soup                                      |       82
 11 | French Onion Soup                                         |       82
 12 | Coq au vin                                                |       82
 13 | Shepherd's pie                                            |       81
 14 | Great British pork: from head to toe                      |       81
 15 | Three Meat Cannelloni Bake                                |       81
 16 | Cioppino                                                  |       81
 17 | hibernation fare                                          |       80
 18 | Spaghetti and Meatballs Recipe with Oven Roasted Tomato Sauce |   80
 19 | Braised Lamb Shanks                                       |       80
 20 | Lamb and Eggplant Casserole (Moussaka)                    |       80
```

Figure 7.19 Output of the recipe recommendation; top 20 dishes the user may love

From figure 7.19 we can deduce it's time for Ragnar to try Spaghetti and Meatballs, a dish made immortally famous by the Disney animation *Lady and the Tramp*. This does sound like a great recommendation for somebody so fond of dishes containing pasta and meatballs, but as we can see by the ingredient count, many more ingredients back up this suggestion. To give us a small hint of what's behind it, we can show the preferred dishes, the top recommendations, and a few of their overlapping ingredients in a single summary graph image.

7.3.6 Step 6: Presentation

The Neo4j web interface allows us to run the model and retrieve a nice-looking graph that summarizes part of the logic behind the recommendations. It shows how recommended dishes are linked to preferred dishes via the ingredients. This is shown in figure 7.20 and is the final output for our case study.

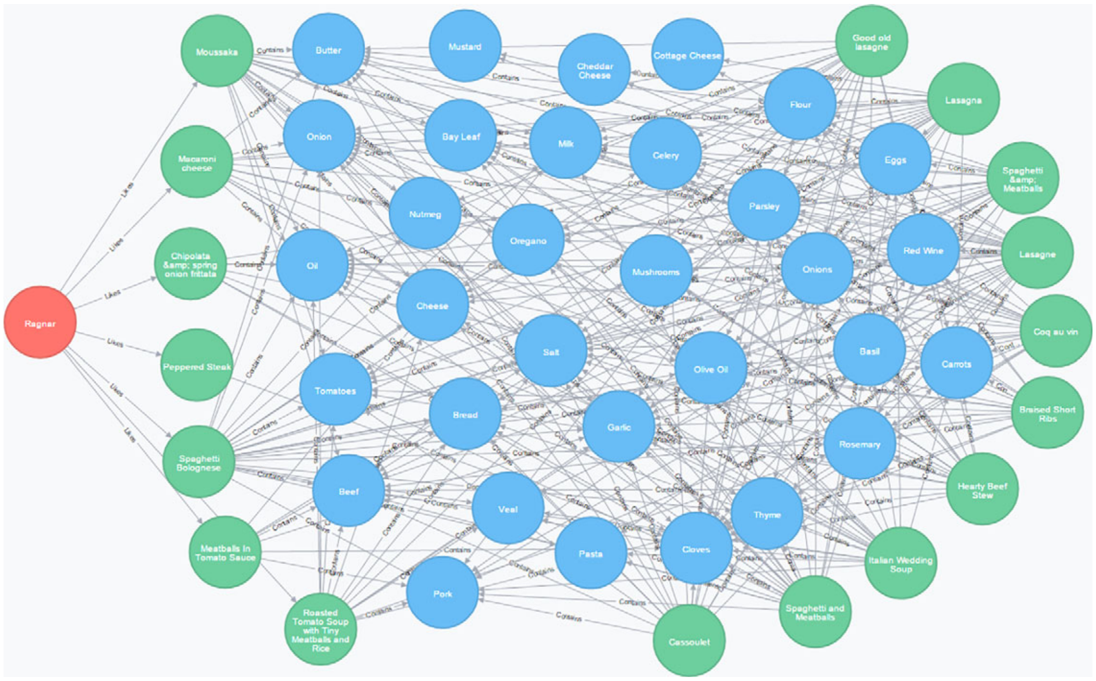


Figure 7.20 Interconnectedness of user-preferred dishes and top 10 recommended dishes via a sub-selection of their overlapping ingredients

With this beautiful graph image we can conclude our chapter in the knowledge that Ragnar has a few tasty dishes to look forward to. Don't forget to try the recommendation system for yourself by inserting your own preferences.

7.4 Summary

In this chapter you learned

- Graph databases are especially useful when encountering data in which relationships between entities are as important as the entities themselves. Compared to the other NoSQL databases, they can handle the biggest complexity but the least data.

- Graph data structures consist of two main components:
  - *Nodes*—These are the entities themselves. In our case study, these are recipes and ingredients.
  - *Edges*—The relationships between entities. Relationships, like nodes, can be of all kinds of types (for example "contains," "likes," "has been to") and can have their own specific properties such as names, weights, or other measures.
- We looked at Neo4j, currently the most popular graph database. For instruction on how to install it, you can consult appendix B. We looked into adding data to Neo4j, querying it using Cypher, and how to access its web interface.
- Cypher is the Neo4j database-specific query language, and we looked at a few examples. We also used it in the case study as part of our dishes recommender system.
- In the chapter's case study we made use of Elasticsearch to clean a huge recipe data dump. We then converted this data to a Neo4j database with recipes and ingredients. The goal of the case study was to recommend dishes to people based on previously shown interest in other dishes. For this we made use of the connectedness of recipes via their ingredients. The py2neo library enabled us to communicate with a Neo4j server from Python.
- It turns out the graph database is not only useful for implementing a recommendation system but also for data exploration. One of the things we found out is the diversity (ingredient-wise) of Spaghetti Bolognese recipes out there.
- We used the Neo4j web interface to create a visual representation of how we get from dish preferences to dish recommendations via the ingredient nodes.