CHAPTER 7:

THE RISE OF GRAPH DATABASES

from Introducing Data Science

by Davy Cielen, Arno D. B. Meysman, and Mohamed Ali

This chapter cover

- 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

SECTION 7.1:

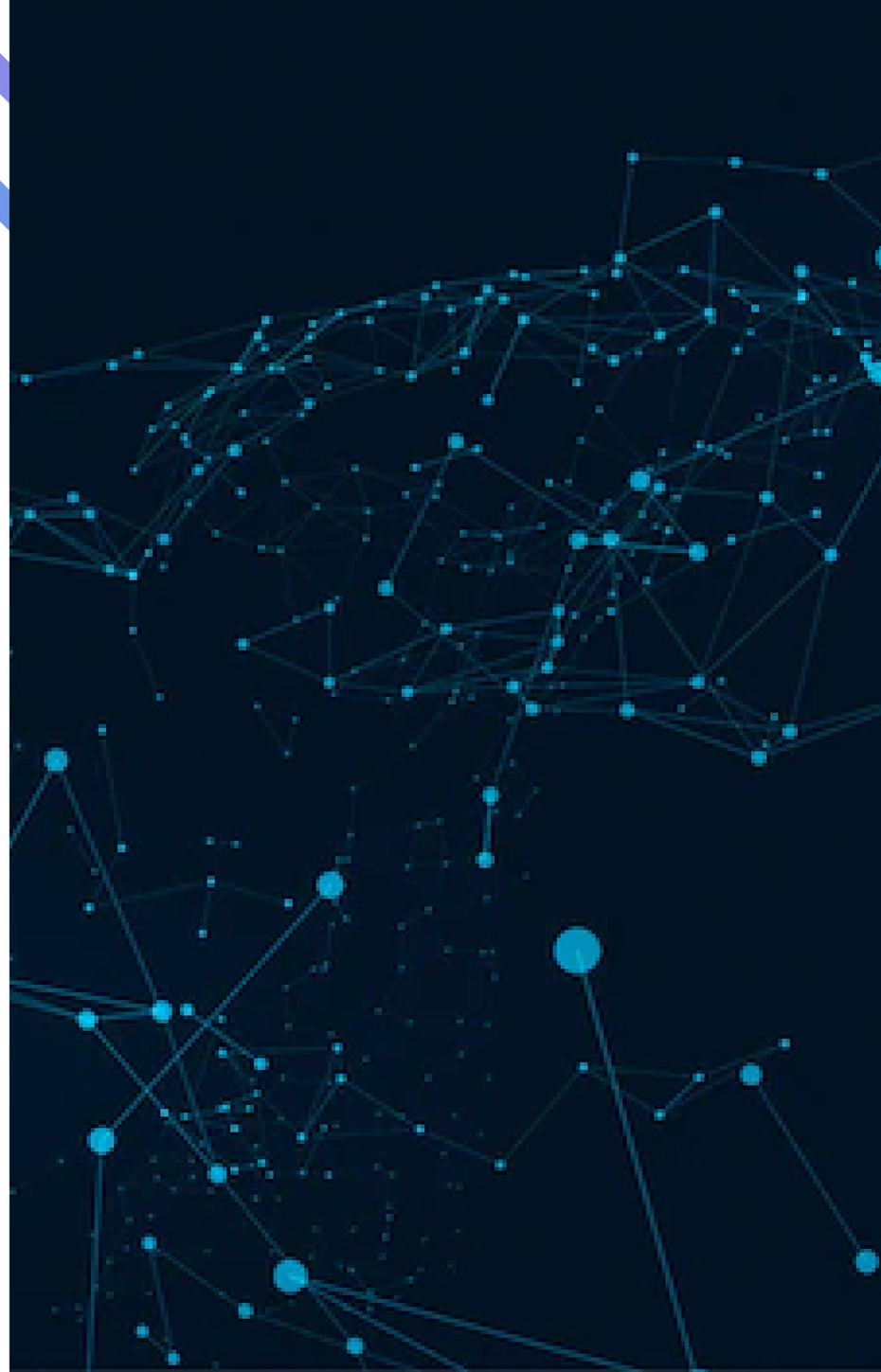
INTRODUCING CONNECTED DATA & GRAPH DATABASES

Connected data is a type of data that has links or relationships between its entities, such as people, places, products, events, and so on. In this section, you will learn how to use graphs and graph databases to model and store connected data in a natural and intuitive way.

What is Connected Data?

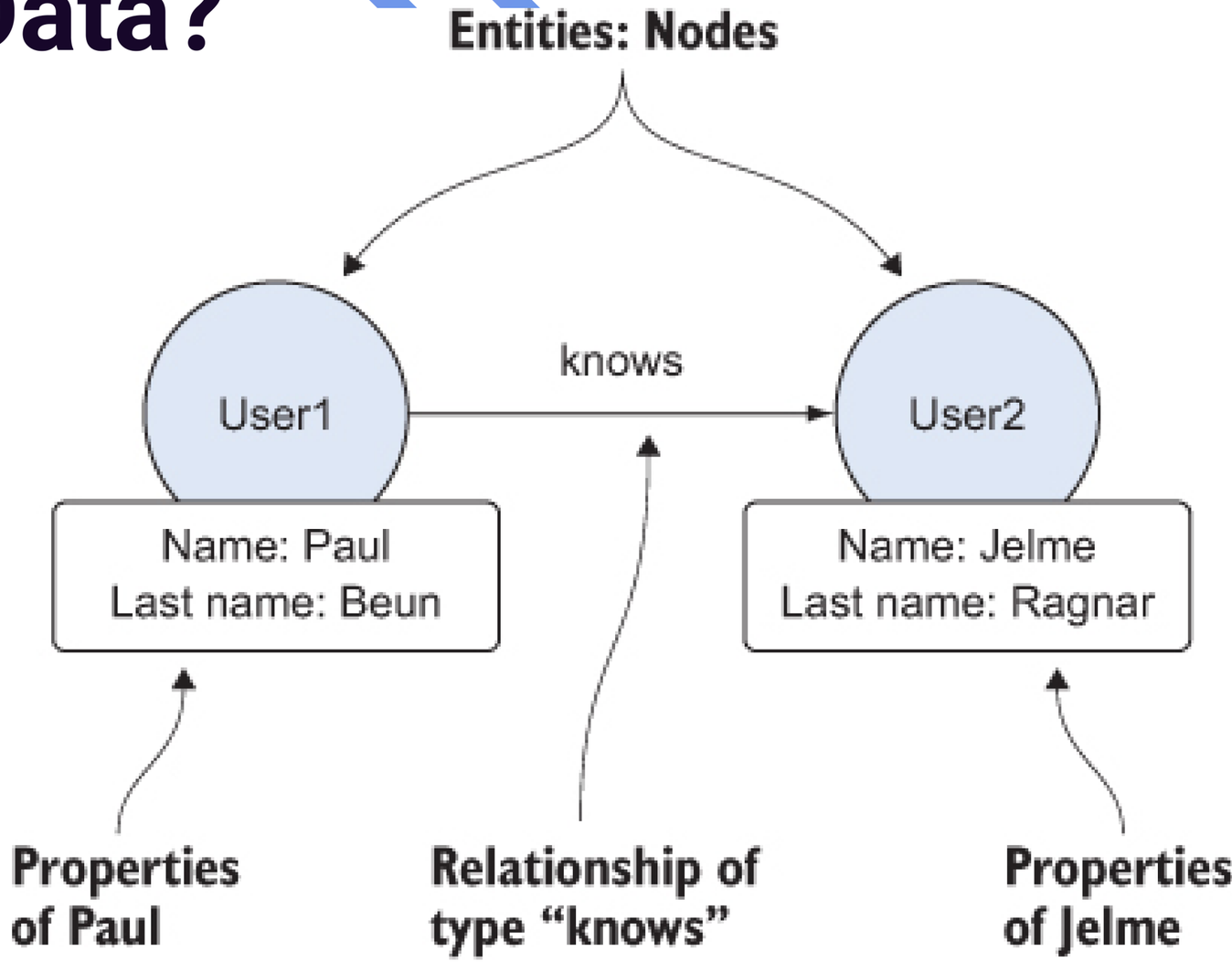
As the name indicates, **Connected Data** is characterized by the fact that the data at hand has a relationship that makes it connected.

And **Graph** often referred to in the same sentence as connected data. They are well suited to represent the connectivity of data in a meaningful way.



What is Connected Data?

Figure 7.1. A simple connected data example:

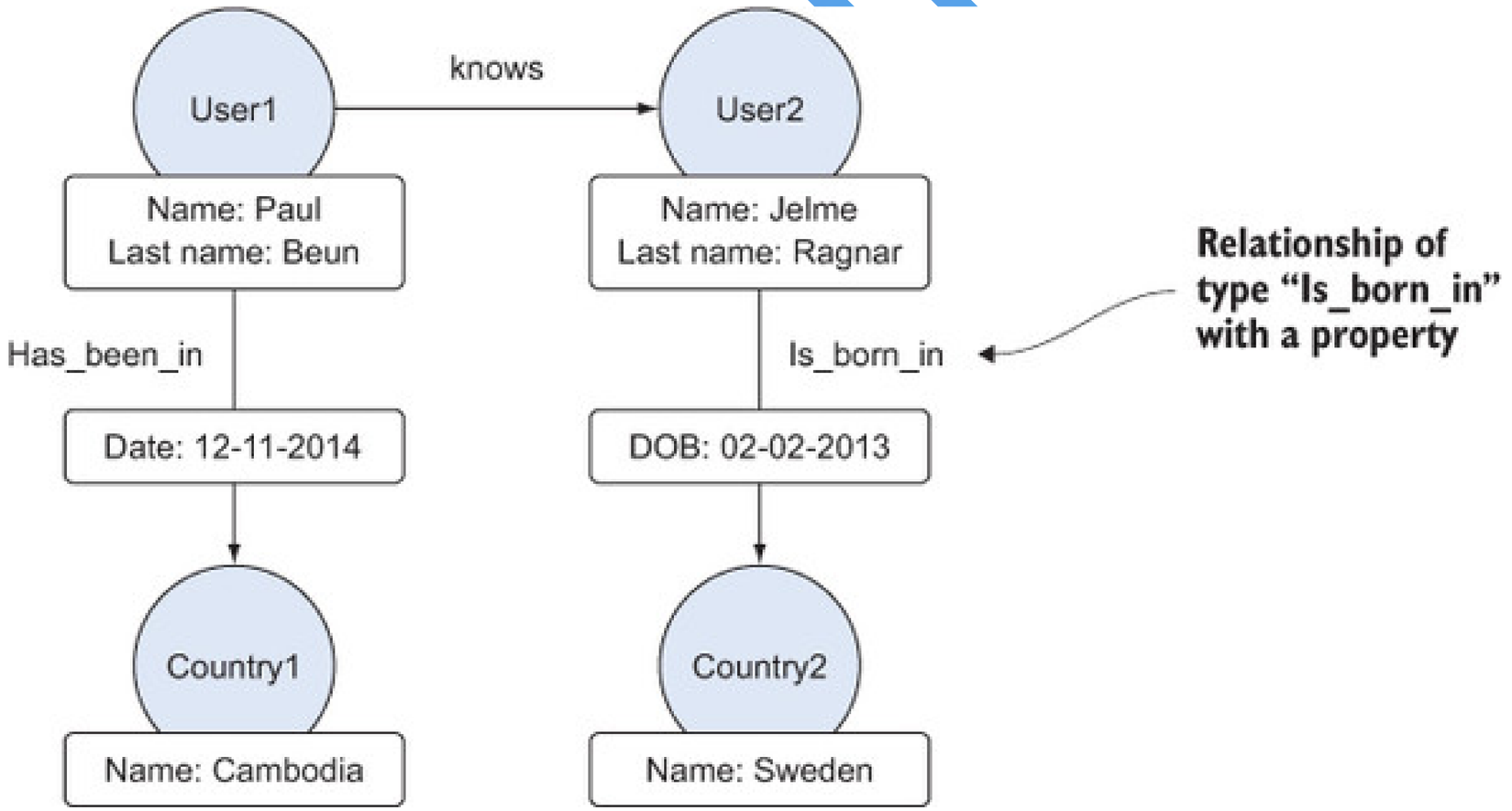


The terminology used:

- Entities
- Properties
- Relationships
- Labels

What is Connected Data?

Figure 7.2. Nodes can have multiple relationships.



When should I use it?

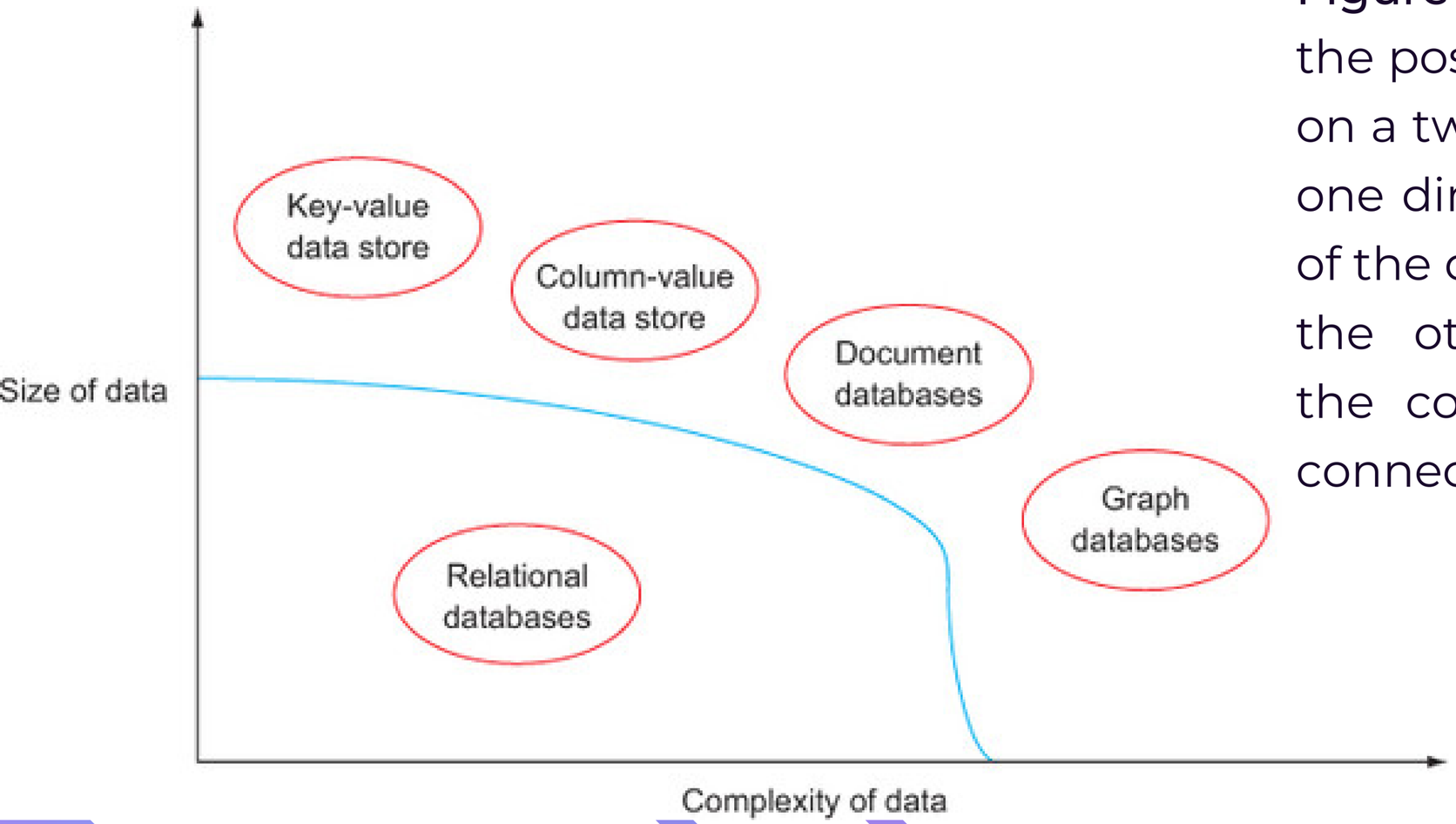


Figure 7.4. This figure illustrates the positioning of graph databases on a two dimensional space where one dimension represents the size of the data one is dealing with, and the other dimension represents the complexity in terms of how connected the data is.

Difficulties when using Relational databases

01

Complexity of Joins

Joining multiple tables and specifying the necessary conditions for the join can become challenging and error-prone.

02

Many-to-Many Joins

Requires the use of intermediate tables or additional queries to retrieve the desired data, leading to more complex and less efficient queries.

03

Query Performance

Joining large tables or performing complex queries can result in slower query execution times, leading to decreased performance.

04

Database Maintenance

Maintaining the integrity and consistency of a relational database with complex relationships can be challenging.

SECTION 7.2:

INTRODUCING NEO4J A GRAPH DATABASE

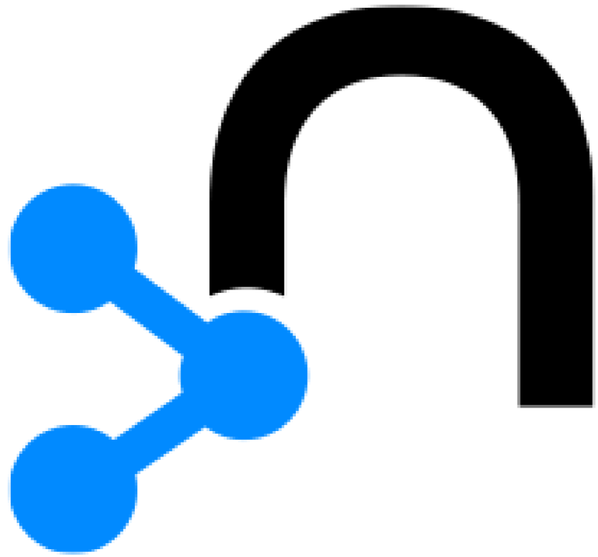
A most popular graph database today. Connected data is generally stored in graph databases. These databases are specifically designed to cope with the structure of connected data.

What is Neo4J?

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.

7.2.1. Cypher: A Graph Query Language

Cypher — is a declarative query language, built on the basic concepts and clauses of SQL but with added graph-specific functionality. And the main idea to understand is a concept of Graph Pattern Matching. There are no tables, where we store our data, instead, we have nodes and each one itself know its type by having a Node Label. You can imagine, that we have an infinity plain surface and all nodes just placed on it somehow.



Comparing with SQL

In SQL we say — give me this SELECTed information FROM this table. In Cypher we say — RETURN me data, that MATCH this pattern. So, as you can see, the pattern for Cypher is like FROM and RETURN is like a SELECT.

SELECT	MATCH pattern
FROM	RETURN it

When SQL want to combine data from many sources, we use JOINs. In Cypher there are no tables and no joins, but nodes connected by relationships. So, for us, this would be just a more complex pattern to express what many things we want to RETURN.

SELECT	MATCH more complex pattern
FROM	RETURN it
JOIN	

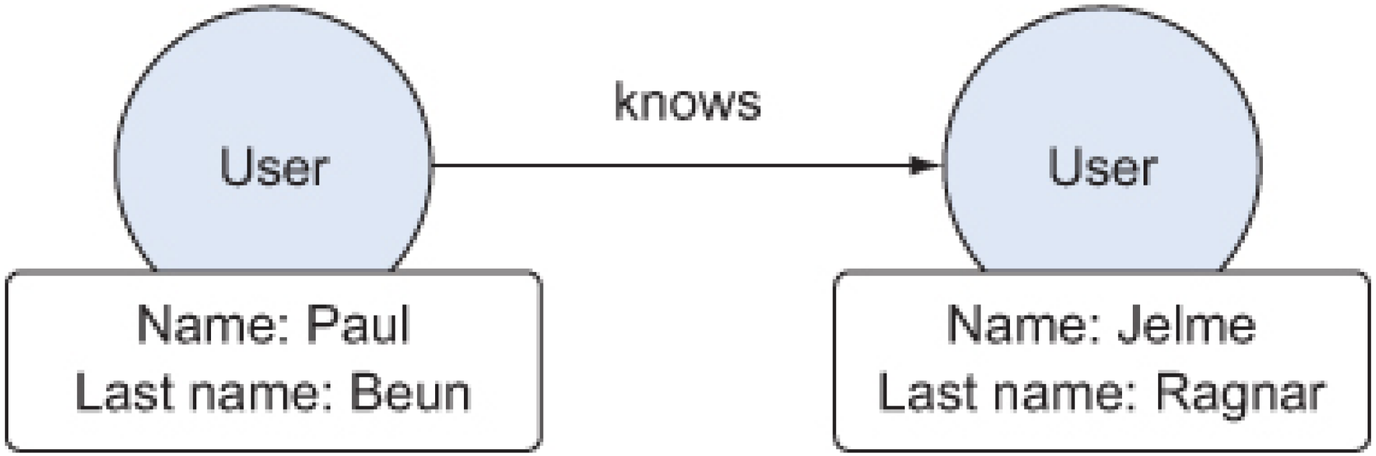
Comparing with SQL

When we need to apply to filter in both SQL and Cypher we simply use WHERE clause. In Cypher we can set not only condition, like, for example, name = “Robert”, but also filter by additional pattern together with existed MATCH pattern.

SELECT	MATCH pattern
FROM	WHERE condition/pattern
WHERE	RETURN it

Knowing the idea, the next required thing is a basic syntax to be able to build simple MATCH patterns.

Cypher Fundamentals



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.

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

Cypher Fundamentals

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

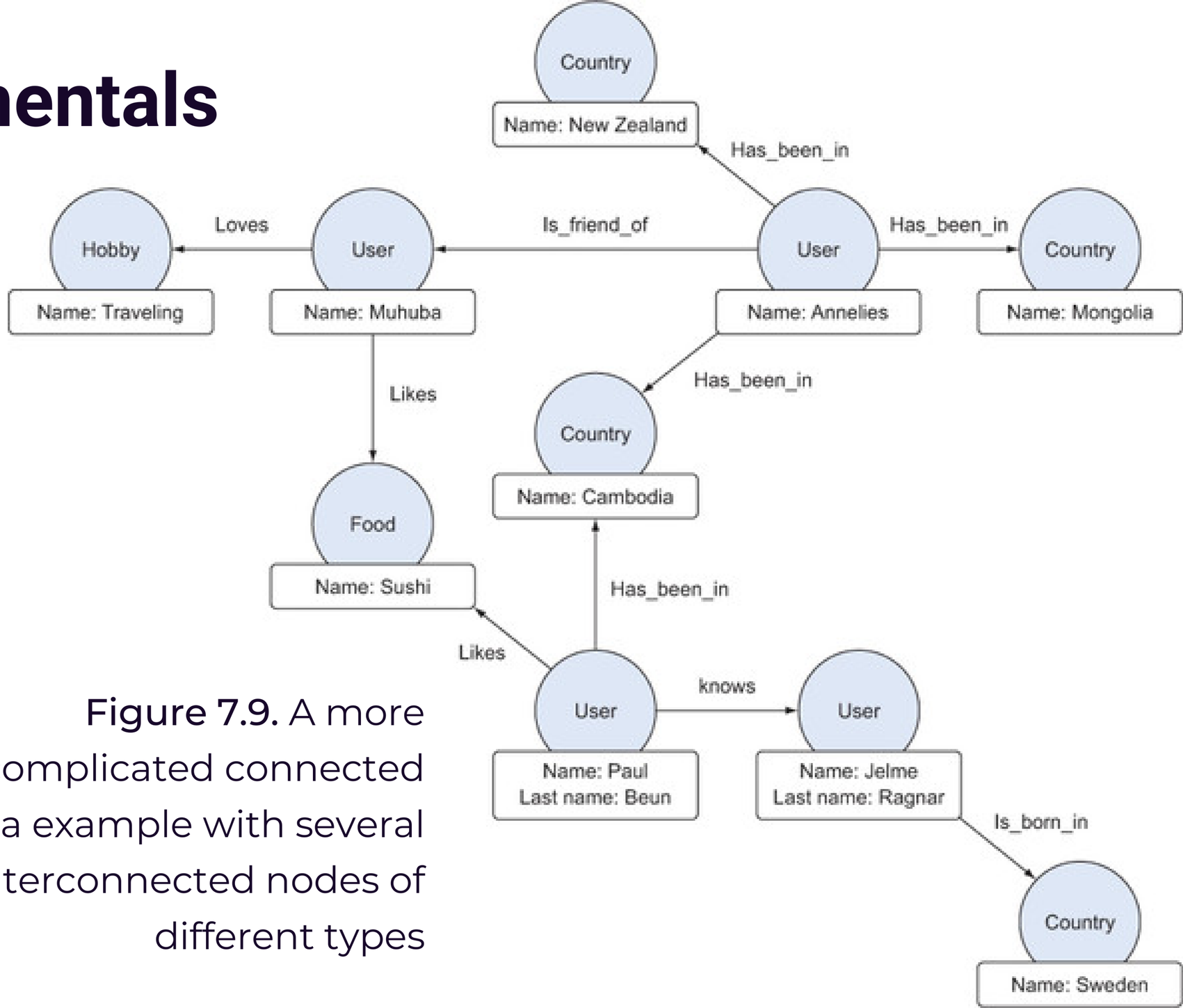
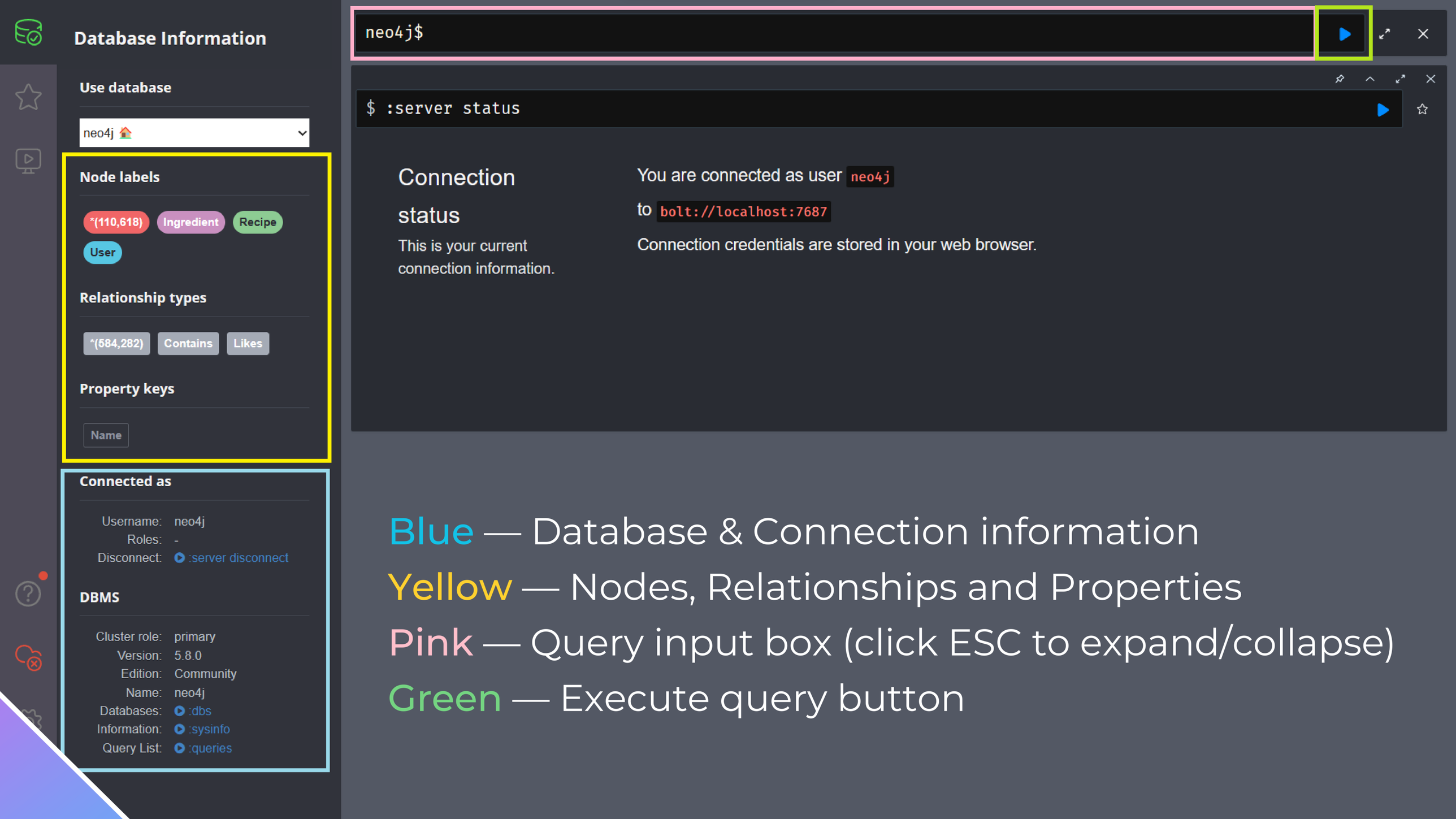


Figure 7.9. A more complicated connected data example with several interconnected nodes of different types


Database Information
Use database
neo4j
Node labels
*(110,618)
Ingredient
Recipe
User
Relationship types
*(584,282)
Contains
Likes
Property keys
Name
Connected as
Username: neo4j
Roles: -
Disconnect: :server disconnect
DBMS
Cluster role: primary
Version: 5.8.0
Edition: Community
Name: neo4j
Databases: :dbs
Information: :sysinfo
Query List: :queries
neo4j$
$:server status
Connection status
This is your current connection information.
You are connected as user neo4j
to bolt://localhost:7687
Connection credentials are stored in your web browser.
Blue — Database & Connection information
Yellow — Nodes, Relationships and Properties
Pink — Query input box (click ESC to expand/collapse)
Green — Execute query button

Cypher Fundamentals

```
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.

Listing 7.1. Cypher data creation statement

Cypher Fundamentals

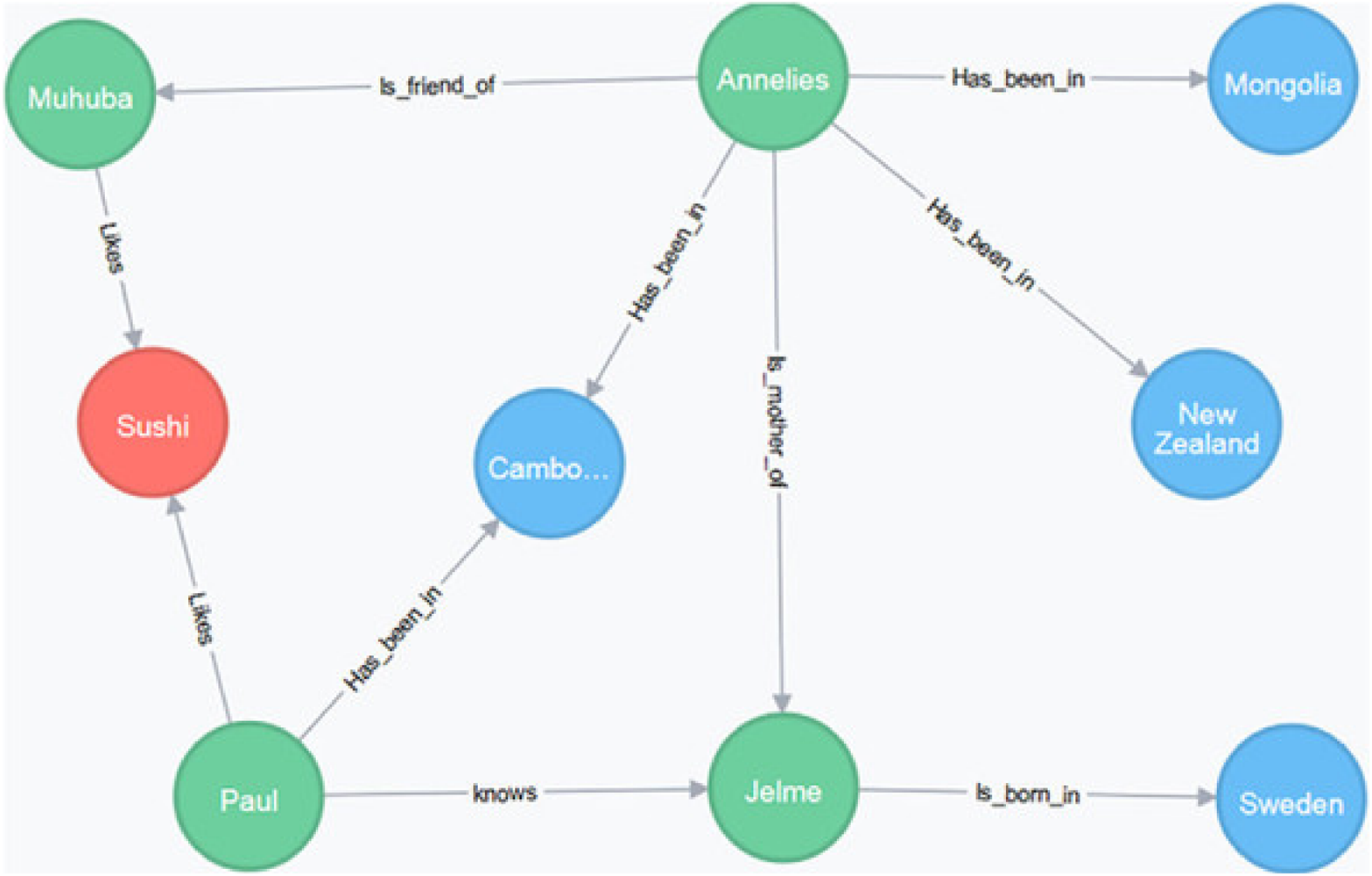


Figure 7.10. The graph drawn in figure 7.9 now has been created in the Neo4j web interface.

Cypher Fundamentals

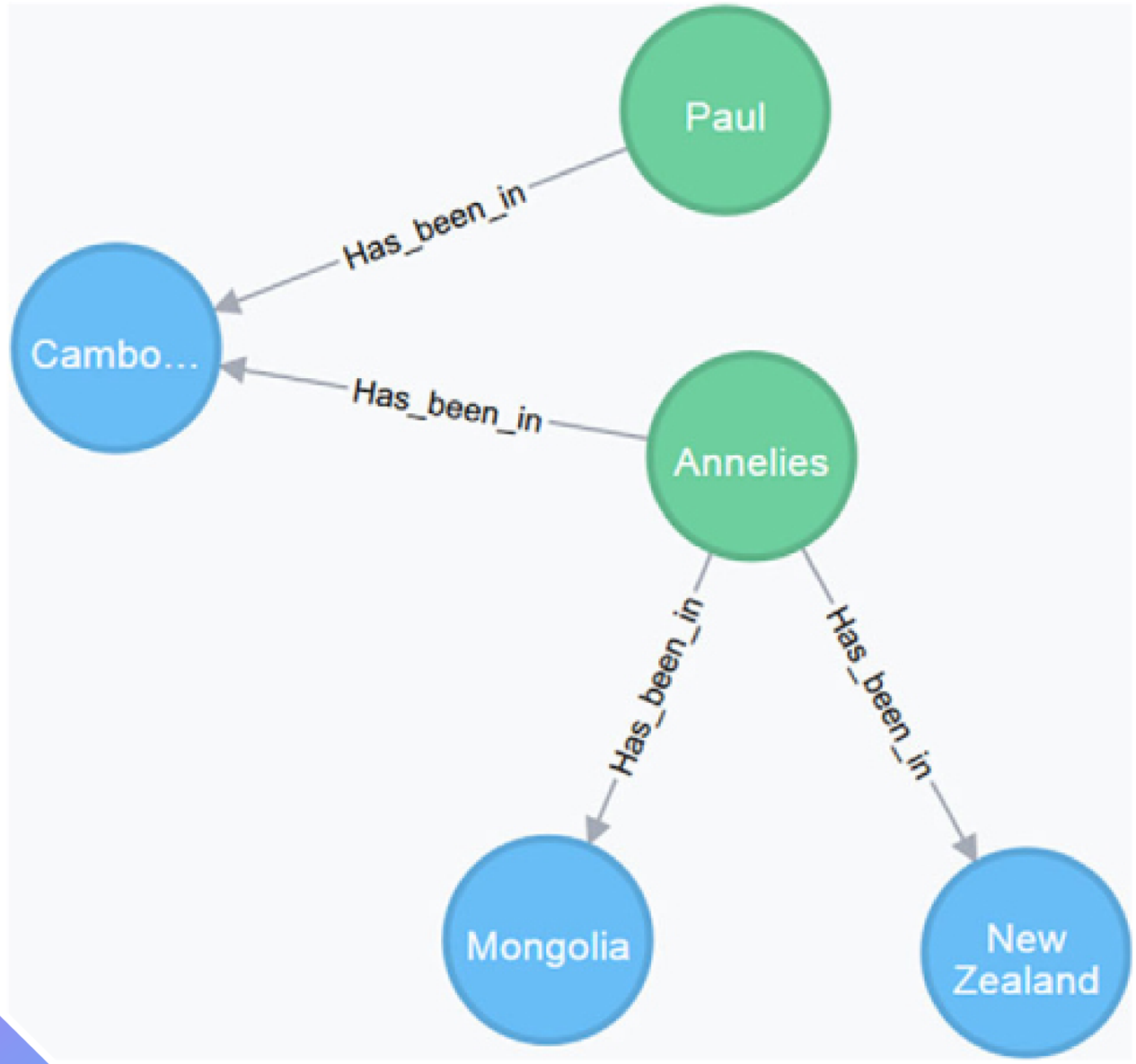


Figure 7.13. Results of question 2: Who has been where? The results of our traversal are now shown in the graph representation of Neo4j. Now we can see that Paul, in addition to Annelies, has also been to Cambodia.

SECTION 7.3:

MAKE 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

ITEM REQUEST

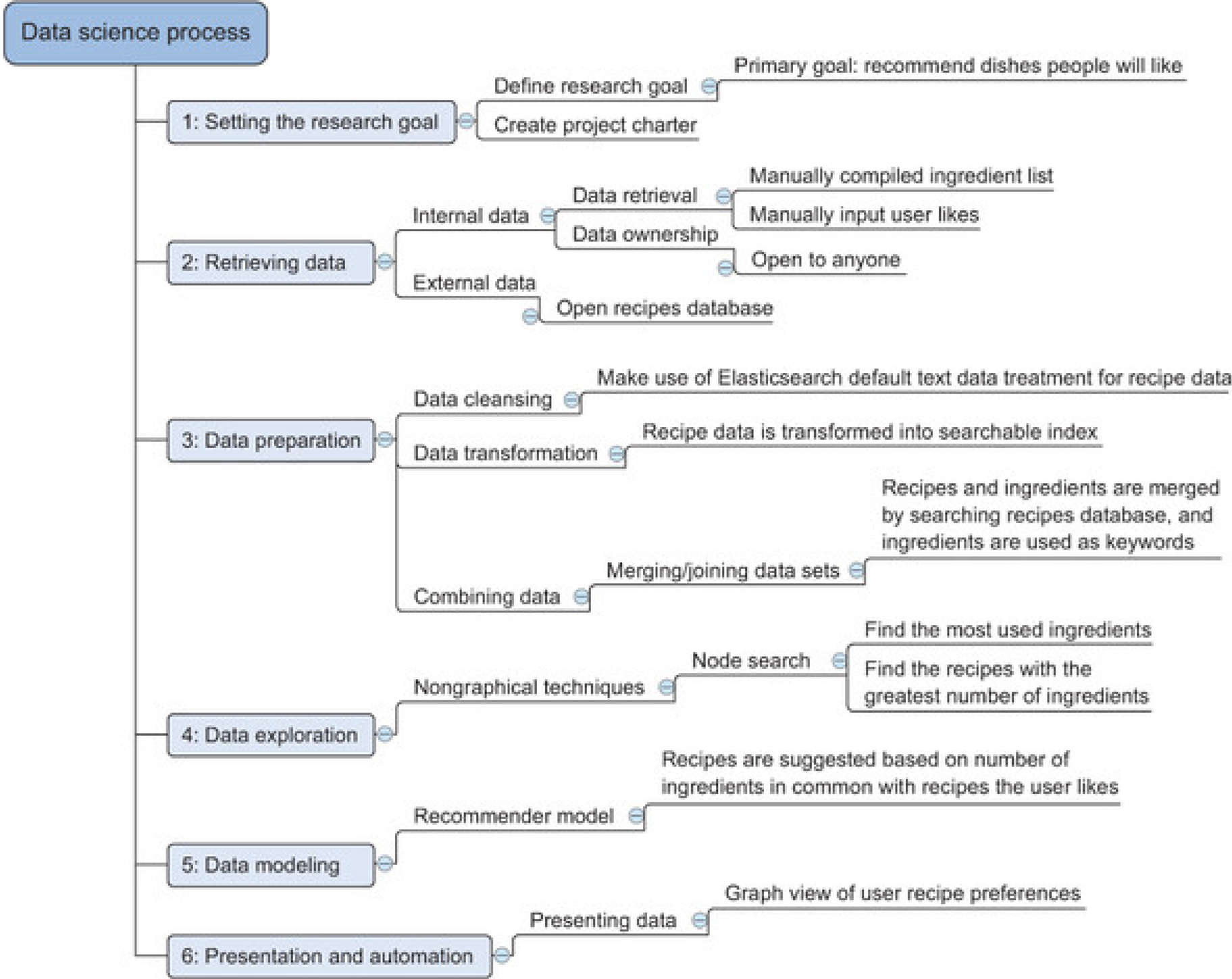
PYTHON3

ELASTICSEARCH

NEO4J

Elasticsearch is a distributed, open-source search and analytics engine that is built on top of Apache Lucene. It provides a scalable search solution that can be used to index and search large volumes of data in real-time. Elasticsearch is designed to be highly available, fault-tolerant, and easy to use. It supports a wide range of use cases, including full-text search, structured search, analytics, and machine learning. Elasticsearch is commonly used in applications such as e-commerce websites, log analysis tools, and enterprise search solutions.

DATA SCIENCE PROCESS



STEP 2 & 3: DATA RETRIEVAL & PREPARATION

We have two possible sources: internal data and external 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 fictivekin/openrecipes repo on Github.

STEP 2 & 3: DATA RETRIEVAL & PREPARATION

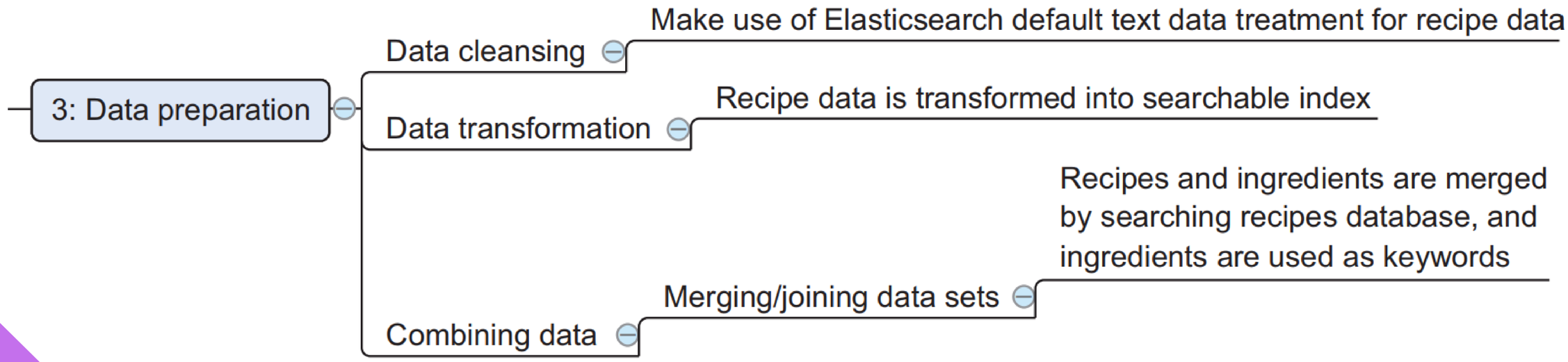
Listing 7.2. Ingredients list text file sample

```
1 Ditalini
2 Egg Noodles
3 Farfalle
4 Fettuccine
5 Fusilli
6 Lasagna
7 Linguine
8 Macaroni
9 Orzo
```

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 hom
     with the soccer-playing girls, and I had returned home with the..."
}
```

STEP 2 & 3: DATA RETRIEVAL & PREPARATION



STEP 2 & 3: DATA RETRIEVAL & PREPARATION

Import Module and create Client used to communicate with database.

If you have a certificate issue, add line in your code.

```python
from elasticsearch import Elasticsearch
import json

ELASTIC_PASSWORD = "DONT_SHARE_THIS_PASSWORD"

client = Elasticsearch(
    "http://localhost:9200",
    basic_auth=("elastic", ELASTIC_PASSWORD),
    verify_certs=False,
)
```

STEP 2 & 3: DATA RETRIEVAL & PREPARATION

Save the location of Json file, Create Index and Mapping

```
file_name = '~/PATH_OF_YOUR_FOLDER/recipes.json'

indexName = "gastronomical"
client.indices.create(index=indexName)
recipeMapping = {
    'properties': {
        'name': {'type': 'text'},
        'ingredients': {'type': 'text'}
    },
}
client.indices.put_mapping(
    index=indexName,
    body=recipeMapping)
```

STEP 2 & 3: DATA RETRIEVAL & PREPARATION

Load Json file to an Array
Use a loop to pass data to index.
If everything went well, we now have an Elasticsearch index by the name “gastronomical” populated by thousands of recipes.

```
25  #Load Json file
26  with open(file_name) as data_file:
27      recipeData = json.load(data_file)
28
29  #Index the recipes
30  for recipe in recipeData:
31      print(recipe.keys())
32      print(recipe['_id'].keys())
33      client.index(
34          index=indexName,
35          id=recipe['_id']['$oid'],
36          document={
37              "name": recipe['name'],
38              "ingredients": recipe['ingredients']
39          }
40      )
```

STEP 2 & 3: DATA RETRIEVAL & PREPARATION

Import necessary py2neo modules and especially add NodeMatcher. Then initiate connection to Neo4J graph database with username and password. Password is required to be changed the first time you log in

```
from py2neo import Graph, Node, Relationship, GraphService, NodeMatcher


graph_db = Graph(
    "http://neo4j:neo4ja@localhost:7474/db/data/",
    user="neo4j",
    password="your_neo4j_password",
    name="neo4j"
)
```

STEP 2 & 3: DATA RETRIEVAL & PREPARATION

Get data from txt file and save it in the ingredients array. Use strip because we have new line with each ingredients in the file

```
23  filename = 'your_path_to_ingredients.txt'
24
25  ingredients = []
26
27  with open(filename) as f:
28      for line in f:
29          ingredients.append(line.strip())
```

STEP 2 & 3: DATA RETRIEVAL & PREPARATION

```
31  ingredientnumber = 0
32  grandtotal = 0
33  matcher = NodeMatcher(graph_db)
34
35  for ingredient in ingredients:
36      try:
37          IngredientNode = matcher.match("Ingredient", Name=ingredient).first()
38          if IngredientNode is None:
39              IngredientNode = Node("Ingredient", Name=ingredient)
40              graph_db.create(IngredientNode)
41      except:
42          continue
```

STEP 2 & 3: DATA RETRIEVAL & PREPARATION

```
44      ingredientnumber += 1
45      searchbody = {
46          "size": 10000,
47          "query": {
48              "match_phrase": {
49                  "ingredients": {
50                      "query": ingredient,
51                  }
52              }
53          }
54      }
55
56      result = client.search(index=indexName, body=searchbody)
```

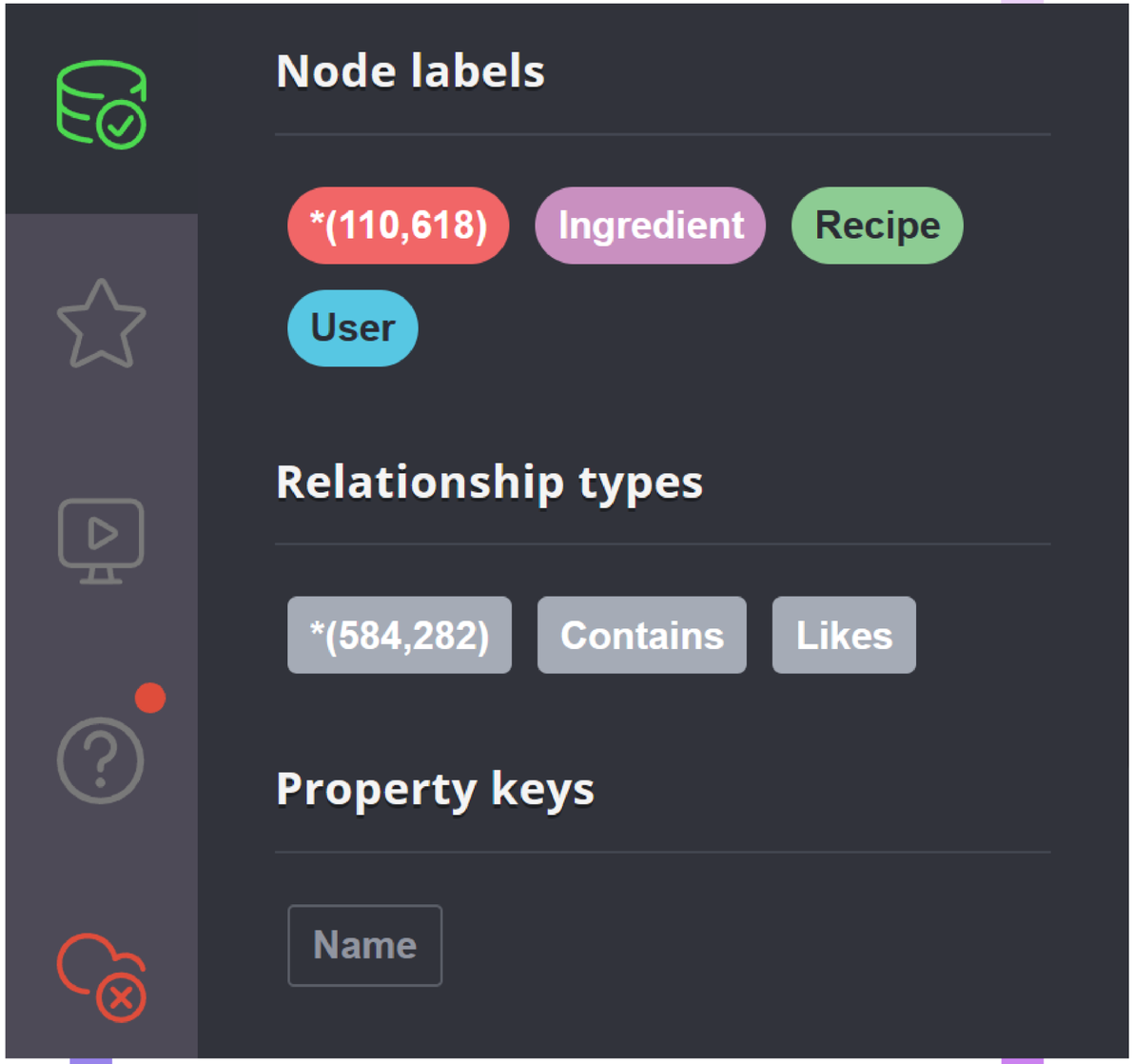
STEP 2 & 3: DATA RETRIEVAL & PREPARATION

```
65    for recipe in result['hits']['hits']:
66        try:
67            RecipeNode = matcher.match("Recipe", Name=recipe['_source']['name']).first()
68            if RecipeNode is None:
69                RecipeNode = Node("Recipe", Name=recipe['_source']['name'])
70                graph_db.create(RecipeNode)
71
72            NodesRelationship = Relationship(RecipeNode, "Contains", IngredientNode)
73            graph_db.merge(NodesRelationship)
74            # print("added: " + recipe['_source']['name'] + " contains " + ingredient)
75        except:
76            continue
```

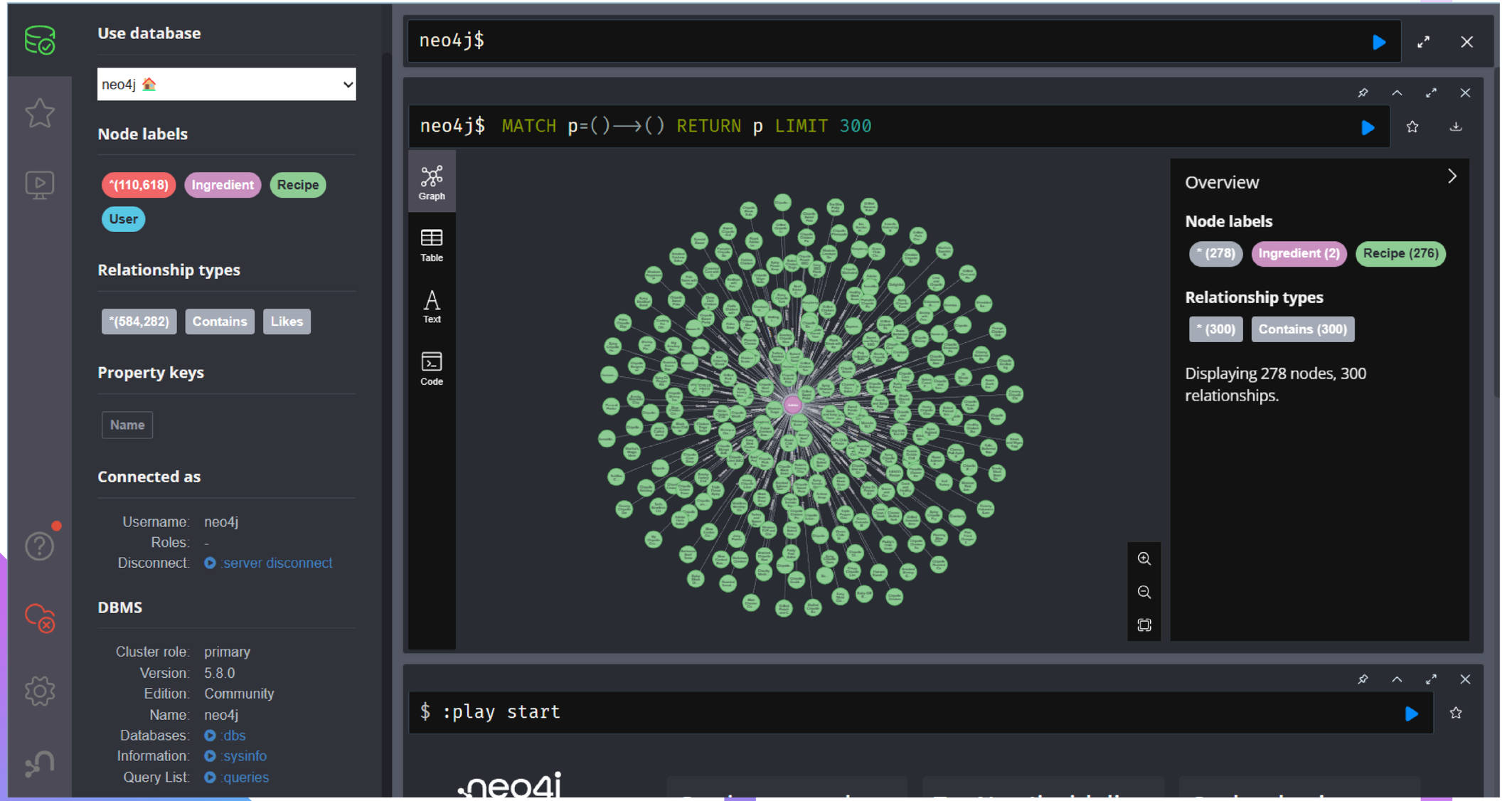
STEP 2 & 3: DATA RETRIEVAL & PREPARATION

Check result on Neo4J

at localhost:7474/browser



STEP 4: DATA EXPLORATION



STEP 4: DATA EXPLORATION

Top 10 Ingredients

```
graph_db.run("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;").to_table()
```

[3] Python

...

Name	num
Onion	9539
All Purpose Flour	9516
Garlic	9442
Butter	9422
Eggs	9410
Flour	9386
Milk	9382
Lemon	9369
Cheese	9365
Onions	9318

STEP 4: DATA EXPLORATION

Top 10 Recipes with the biggest number of ingredients

[4]

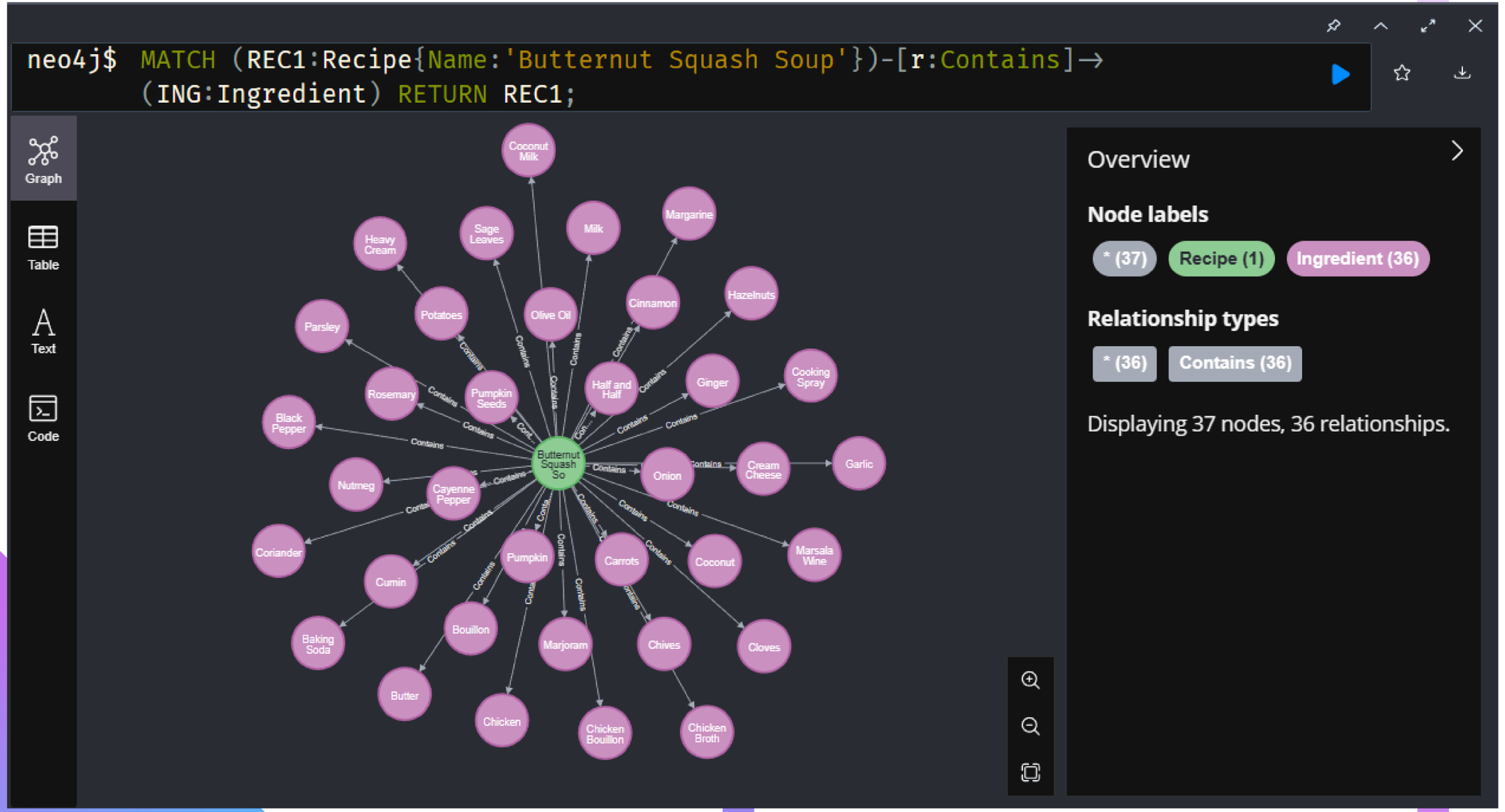
```
graph_db.run("MATCH (REC:Recipe)-[r:Contains]->(ING:Ingredient) WITH REC, count(r) AS num
RETURN REC.Name as Name, num ORDER BY num DESC LIMIT 10;").to_table()
```

Python

...

Name	num
Butternut Squash Soup	36
Fish Tacos	34
Cooking For Others: 25 Years of Jor, 1 of BGSK	34
Chicken Tortilla Soup	33
Chicken Tikka Masala	33
Kedgeree	32
Fattoush	32
Chicken and Dumplings	31
Hearty Beef Stew	31
Spaghetti Bolognese	31

STEP 4: DATA EXPLORATION



STEP 5: DATA MODELING

Add User

```
matcher = NodeMatcher(graph_db)
user = matcher.match("User", Name="Ragnar").first()
if not user:
    user = Node("User", Name="Ragnar")
    graph_db.create(user)
```

```
RecipeRef = matcher.match("Recipe", Name="Spaghetti Bolognese").first()


if RecipeRef:
    NodesRelationship = Relationship(UserRef, "Likes", RecipeRef)
    graph_db.merge(NodesRelationship)
```

[19] Python

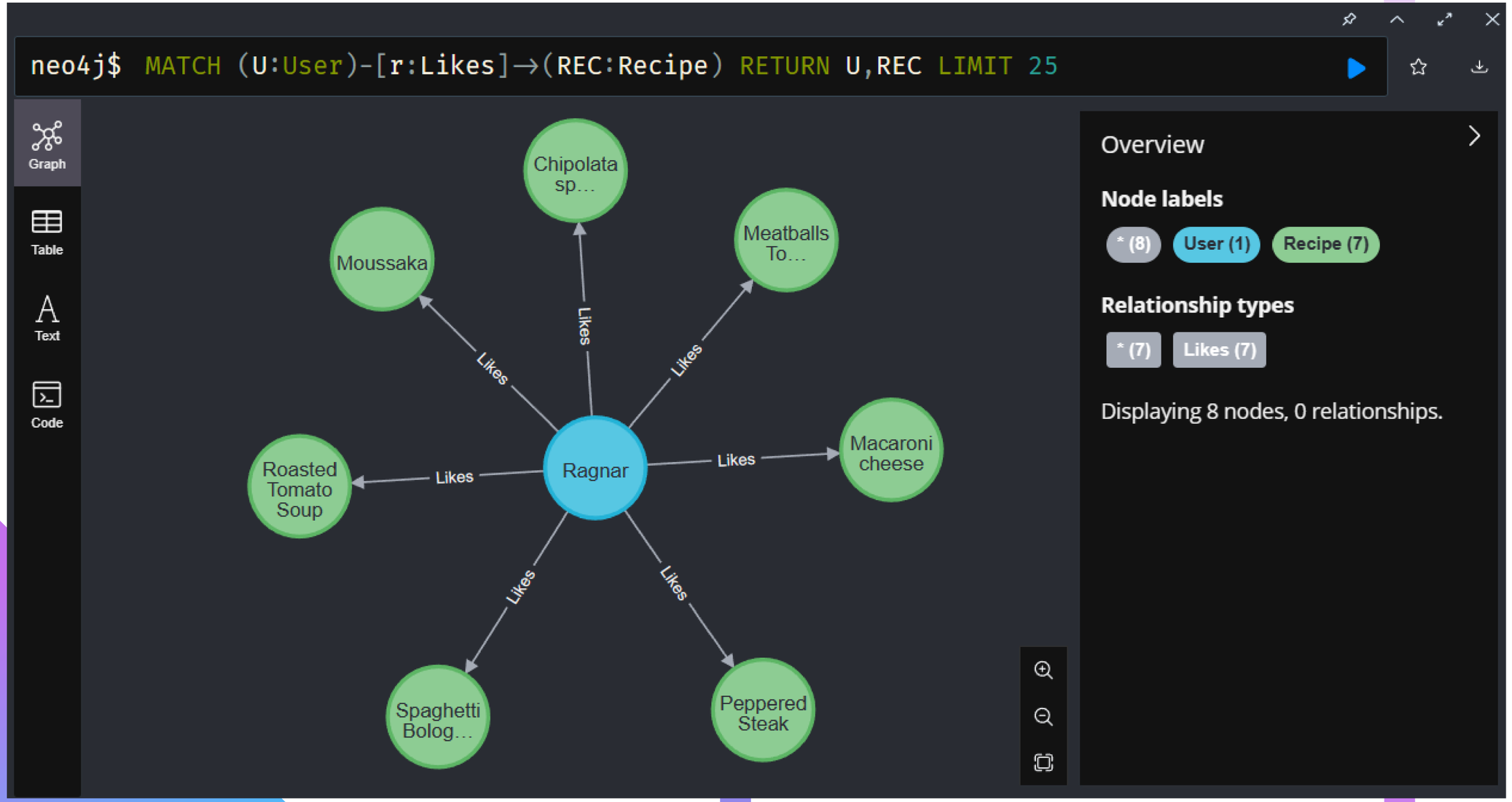
```
(_110617)-[:Likes {}]→(_17521)
```

STEP 5: DATA MODELING

```
recipe1 = matcher.match("Recipe", Name="Roasted Tomato Soup with Tiny Meatballs and Rice").first()
recipe2 = matcher.match("Recipe", Name="Moussaka").first()
recipe3 = matcher.match("Recipe", Name="Chipolata &amp;amp; spring onion frittata").first()
recipe4 = matcher.match("Recipe", Name="Meatballs In Tomato Sauce").first()
recipe5 = matcher.match("Recipe", Name="Macaroni cheese").first()
recipe6 = matcher.match("Recipe", Name="Peppered Steak").first()

if recipe1:
    graph_db.create(Relationship(UserRef, "Likes", recipe1))
if recipe2:
    graph_db.create(Relationship(UserRef, "Likes", recipe2))
if recipe3:
    graph_db.create(Relationship(UserRef, "Likes", recipe3))
if recipe4:
    graph_db.create(Relationship(UserRef, "Likes", recipe4))
if recipe5:
    graph_db.create(Relationship(UserRef, "Likes", recipe5))
if recipe6:
    graph_db.create(Relationship(UserRef, "Likes", recipe6))
```

STEP 5: DATA MODELING



STEP 5: DATA MODELING

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.

STEP 5: DATA MODELING

Recommend recipes to User

```
graph_db.run(
    "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;"
).to_table()
```

[21]

Python

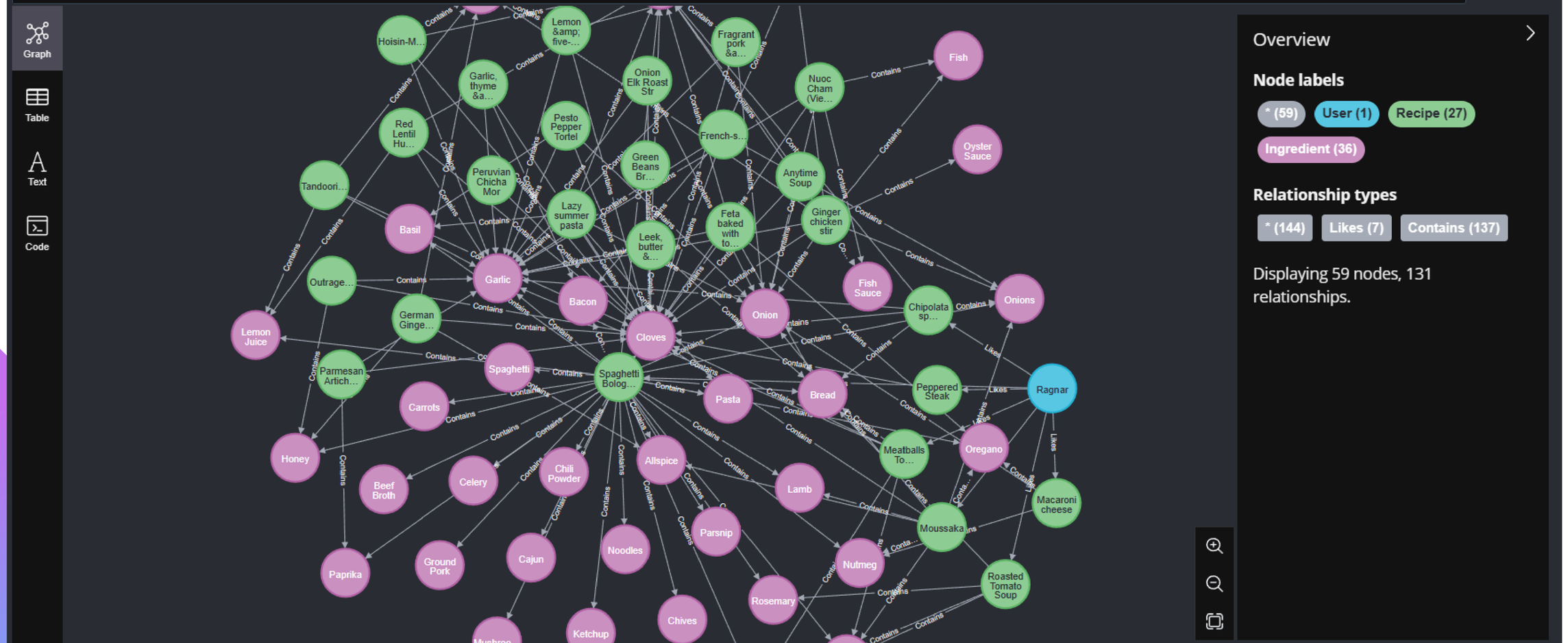
STEP 5: DATA MODELING

REC2.Name	IngCount
Italian Wedding Soup	38
Lasagne	38
Spaghetti and Meatballs	38
Gazpacho	35
Roasted Butternut Squash Soup	34
Coq au vin	34
Butternut Squash Soup	34
Hearty Beef Stew	33
Turkey Meatballs	32
Spaghetti & Meatballs	32
Cincinnati Chili	32
Cassoulet	31
Lasagna	30
Spaghetti Sauce	30
Ratatouille	30

From result, we can deduce it's time for Ragnar to try Italian Wedding Soup. 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.

STEP 6: PRESENTATION

```
1  MATCH (USR1:User{Name:'Ragnar'})-[l1:Likes]→(REC1:Recipe),(REC1)-[c1:Contains]→(ING1:Ingredient)
2  WITH USR1, ING1, REC1
3  MATCH (REC2:Recipe)-[c2:Contains]→(ING1:Ingredient)
4  WHERE REC1 <> REC2
5  RETURN USR1, REC2, REC1 LIMIT 20
6
```



Thank You

Summary by:

A41316 - Nguyễn Hữu Khoa
huukhoa203@gmail.com

A42718 - Lê Thảo Quyên
irisle2003@gmail.com