Neo4j Graph Data Modeling

Neo4j is a graph database that allows you to model your data as a graph and find solutions to complex real-world problems that are difficult to solve using any other type of database.

This book is designed to help you understand the intricacies of modeling a graph for any domain.

The book starts with an example of a graph problem and then introduces you to modeling non-graph problems using Neo4j. Concepts such as the evolution of your database, chains, access control, and recommendations are addressed, along with examples and are modeled in a graph. Throughout the book, you will discover design choices and trade-offs, and understand how and when to use them. By the end of the book, you will be able to effectively use Neo4j to model your database for efficiency and flexibility.

Who this book is written for
If you are a developer who wants to understand the fundamentals of modeling data in Neo4j and how it can be used to model full-fledged applications, then this book is for you. Some understanding of domain modeling may be advantageous but is not essential.

What you will learn from this book
- Translate a problem domain from a whiteboard to your database
- Make design decisions based on the nature of data and how it is going to be used
- Use Cypher to create and query data
- Evolve your database in stages
- Optimize the performance of your application with data design
- Design paradigms to ensure flexibility, ease of querying, and performance
- Move from an existing model to a new model without losing consistency

Who this book is written for
If you are a developer who wants to understand the fundamentals of modeling data in Neo4j and how it can be used to model full-fledged applications, then this book is for you. Some understanding of domain modeling may be advantageous but is not essential.

What you will learn from this book
- Translate a problem domain from a whiteboard to your database
- Make design decisions based on the nature of data and how it is going to be used
- Use Cypher to create and query data
- Evolve your database in stages
- Optimize the performance of your application with data design
- Design paradigms to ensure flexibility, ease of querying, and performance
- Move from an existing model to a new model without losing consistency
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 2 'Modeling Flights and Cities'
- A synopsis of the book’s content
- More information on Neo4j Graph Data Modeling
About the Author

Mahesh Lal is a developer who has experience with various technologies. In 2011, while working on a social network for a client, he discovered the power of graphs, specifically Neo4j. Since then, he has been working with multiple clients across various domains for modeling their data as a graph. Currently working for ThoughtWorks, India, he is trying to help his clients look at their search problems in the form of a graph.
Graph databases have been gaining traction for a long time now and companies have adopted them for various use cases. Neo4j, the world's leading graph database, has been at the forefront of this trend and is widely used in production by companies that are world leaders in their respective domains. Advice on the usage of Neo4j using Cypher (the Neo4j query language), performance tuning of Neo4j, and general information can be sourced from various sources, including, but not limited to, blogs, the Neo4j website, the Neo4j mailing list, as well as books written by authors on these subjects. However, there is limited information regarding modeling information in Neo4j. This book aims to address this gap by giving examples of how various scenarios can be modeled in Neo4j. By sticking to a nonsocial graph example, this book steers clear of the stereotypical use case of graph databases. While we use Neo4j as an example to discuss graph database modeling, the concepts discussed can be applied to any graph database. We believe this book to be a useful tool for anyone wishing to understand graph database modeling.

What this book covers

Chapter 1, Graphs Are Everywhere, introduces you to the logical data representation of a property graph model, the various use cases of graph databases, and the advantages of using graph databases in general and Neo4j in particular.

Chapter 2, Modeling Flights and Cities, introduces you to basic modeling in Neo4j by discussing how flights and cities can be modeled in a graph database. We then create cities and flights in Neo4j using Cypher.

Chapter 3, Formulating an Itinerary, discusses some basic querying using Cypher for the purpose of creating a light itinerary from the existing data in Neo4j.

Chapter 4, Modeling Bookings and Users, discusses how to represent, in a graph database, a data model that is traditionally implemented in a RDBMS by modeling bookings in Neo4j.
Preface

Chapter 5, *Refactoring the Data Model*, covers refactoring the data model to accommodate changes in the business using Cypher. We do this as a multistep process and demonstrate how simple it is in Neo4j to change the data model.

Chapter 6, *Modeling Communication Chains*, discusses how communication chains can be modeled in Neo4j. This also covers how we can represent temporal relationships using this modeling technique, which allows for efficient retrieval of data while maintaining the integrity of the relationships between various pieces of information.

Chapter 7, *Modeling Access Control*, focuses on how access control lists can be modeled in Neo4j. This also discusses how hierarchies and groups can be modeled in Neo4j.

Chapter 8, *Recommendations and Analysis of Historical Data*, demonstrates the construction of queries to recommend cities and hotels to travelers using the data that we have in the database. This also analyzes some historical data to discover patterns in the database. This chapter demonstrates queries that would normally require some heavy lifting in an RDBMS.

Chapter 9, *Wrapping Up*, is the final chapter and talks about potential issues that you might run into while using Neo4j or a graph database, and also how modeling for a current problem isn't future-proof.
Modeling Flights and Cities

We looked at what graphs are and what domains they might be suited for. Now, it is time to dive deeper into concepts that are related to graph databases and how we can go ahead and create our data in a graph. In this chapter, we will look at:

- How graphs can be used outside the social context modeling flights and cities for creating an itinerary
- Adding nodes, labels, properties, relationships, uniqueness constraints, and indices

Before we dive deep into modeling, we recommend that you download the code samples that you will need to run the examples in this chapter. The code can be downloaded from [https://github.com/maheshlal2910/neo4j_graph_data_modelling](https://github.com/maheshlal2910/neo4j_graph_data_modelling).

**Graphs are more than social**

Often, when we talk about graph databases and their most suited use cases, people point out that social networks are a good use case for graphs. While this is true, in a way, it pigeonholes graph databases. Graph databases are versatile tools that can be used to model various domains and problems. In this book, we pick up a nonsocial example—travel, and explain how we can model data for various subsystems that would be used in a travel website using Neo4j.
Designing a system to get a travel itinerary

The travel domain is interesting in terms of data modeling challenges. Throughout this book, we will be modeling systems that work together in a website that can be used for planning flight travel. Travelers would like to look at the options for an itinerary before booking any particular set of flights, especially if there is no direct flight from the traveler’s current city to the destination city. Normally, an itinerary includes total duration, layover duration, and the number of hops it takes to reach the destination. We cannot, however, derive the itinerary without modeling cities and flights, which brings us to our first data modeling problem.

Introduction to modeling flights and cities

If we were to explain the problem of cities and the number of flights between them, we could start with drawing cities as nodes. In case two cities have two or more direct flights between them, we connect those two cities with an undirected relationship, as shown in the following figure:

![Figure 2.1: Cities and flight routes](image)
In the situation depicted by Figure 2.1, there is a direct route between New York and the following cities: Chicago, Los Angeles, Washington, and Las Vegas, for instance. However, if a traveler has to fly to New York from Istanbul, they have to choose from among the following routes:

- Istanbul—London—New York
- Istanbul—Athens—Paris—New York

In Figure 2.1, an undirected relationship between cities that are connected by flights is an abstraction stating that there are at least two direct flights between those two cities. For example, between New York and Los Angeles, there should be one flight from New York to Los Angeles and vice versa. In reality, there might be multiple flights, operated by different operators, travelling to and fro between two cities that are directly connected in Figure 2.1. This complicates the problem of presenting the traveler with a good itinerary.

**Identifying the entities**

Before we jump to modeling, we need to identify the interacting entities involved in this problem. For the problem of creating an itinerary, we need cities and flights connecting them. Cities form hubs, which one can travel to or transit through. Flights fly from one city to another.

![Figure 2.2: Cities and flights](image)
The preceding figure represents the information that is needed and applicable to both the entities. While city names aren't unique in the real world, we could use city names to uniquely identify the city in our domain. In addition to the name, a city will also have the name of the country it belongs to.

A flight can be uniquely identified by its code. Other parameters include duration of the flight (in minutes), departure and arrival time (in minutes within the day where 0000 hrs will be 0 minutes and 2359 hrs will be 1439 minutes). We capture information about the airports between which the flight operates in source_airport_code and destination_airport_code.

In the RDBMS world, an attribute that uniquely identifies a record is called a primary key. More often than not, this primary key attribute is denoted by prefixing pk_ to its name. Also, primary keys are mostly some sort of long integer. In this book, we will use UUIDs as primary keys wherever other attributes can't be used as primary keys. However, in cases where we can use any property as a key, we will (for convenience) use that field as a key. Please note that in production systems, the key should be a distinct property serving a single purpose—uniquely identifying the node/relationship. Also, having either long integers or UUIDs as IDs allows us to use them in URLs where other properties might not be usable.

Introduction to modeling nodes and relationships

We looked at the data models in Figure 2.2. Without delving into technical details, we can say that "A person can fly from New York to London on carrier X". With this statement, we can start exploring possibilities in which this data can be modelled. A preliminary approach would be to mark cities as nodes and flights as relationships, as shown in the following figure:
Figure 2.3: A preliminary model with flights and cities—the property graph
This might seem to be a fair model, however, there are a few problems with the approach. Relationships, in graphs, are used to model how the entities' nodes are related to each other. As discussed earlier, flights are one of the two core entities in our model. Flights don't relate cities to each other, instead, they are a means to get from one city to another. Modeling flights as relationships, can work out in presenting a flight plan, but if we have to allow flight bookings in the future, then we need to change flights to nodes. In general, it's always a good practice to model any entity in the domain as a node. Figure 2.4 shows us an alternative approach to model flights and cities.

Modeling entities as relationships must be avoided. Relationships in Neo4j can’t have other relationships linked to them. Relationships should depict semantically relevant connections between two entities, not entities themselves.

Figure 2.4: A model for flights and cities
The preceding data model has :Flight and :City as labels for the nodes for flights and cities, respectively. :HAS_FLIGHT and :FLYING_TO are relationships that link a flight to its origin city and destination city, respectively.

When we represent a part of a graph that contains all possible nodes and relationships, we lay down a specification for how data is connected and structured. Thus, we can describe graphs using specification by example.

This graph seems to be a good starting point for us to begin feeding the data into Neo4j.

Before we move forward, it would be a good idea to install Neo4j so that we can work on the Cypher queries. Neo4j can be downloaded from http://neo4j.com/download/.

Once installed, we will use the neo4j-shell, a console tool for Neo4j to create nodes and relationships. Further in this book, we will make use of the Neo4j Browser tool, when we start dealing with traversal and exploring our graph. Before trying out any of the queries, ensure that the Neo4j server is started.

**Using Cypher to operate on Neo4j**

Operations on Neo4j are generally performed using a query language called Cypher. Cypher is a simple, expressive, SQL-like language that allows us to create, read, update, and delete nodes and relationships in Neo4j. To retrieve data from a Neo4j store, we write Cypher queries, which specify which nodes and which relationships to traverse.

Cypher is a declarative graph query language. Each query is built of clauses and each clause pipes/feeds the next clause with data. Cypher is designed to be a humane query language suitable for developers and operations professionals, and hence, elegantly combines simplicity, expressiveness, and efficiency.

There are ways in which you can, and should influence efficiency from a user perspective, particularly by writing queries that utilize and are sympathetic to the graph structure. However, each query is planned, costed, and executed by the query engine that tries to optimize queries. This allows users to focus on better modeling rather than worrying about the optimization of the queries being written.
Modeling Flights and Cities

To start with, we will need the following clauses:

- **CREATE**: This clause is used to create nodes and relationships.
- **MATCH**: This clause matches a certain set of nodes and relationships following the patterns specified.
- **RETURN**: This decides which part of the created data should be returned. It can be used to return nodes, relationships, or even individual properties.

Before we move forward, we need to ensure that we have a working Neo4j installation.

There are other tools such as Gremlin and the Java API that can be used to query and operate on Neo4j. We feel that Cypher is the most expressive among all of these options. Cypher, however, has limitations to processing which Gremlin addresses better. Gremlin, is not officially supported as of 2.2.0, and Neo4j requires a plugin to execute Cypher queries.

Creating cities in Neo4j

We will model cities as nodes, as shown in Figure 2.4, with the city's name and country as properties. Cities should have unique names. For this, we can add a constraint before we start creating cities in our graph.

The query is as follows:

```bash
eo4j-sh (?)$ CREATE CONSTRAINT ON (city: City) ASSERT city.name IS UNIQUE;
```

The output of the preceding query is as shown:

```
+-------------------+
| No data returned. |
+-------------------+
Constraints added: 1
```

This adds a constraint on all nodes that will henceforth be created with the label City to have a unique name property. The city in (city: City) is a placeholder, like a variable, for any node with label City. Note that the addition of a uniqueness constraint is idempotent—it can be repeated multiple times without throwing an error or changing the constraint after it first gets added.
It's a good practice to add a uniqueness constraint, like we did, before we start adding nodes that have a particular label. While a uniqueness constraint can be added anytime, creating one beforehand ensures that no two nodes with the same label will have the same identifier. Currently, there is no way to specify a uniqueness constraint that combines multiple fields. However, we can have multiple uniqueness constraints on the label. To emulate a uniqueness constraint that spans multiple properties, we can create a property that combines the values of two properties and creates a new constraint on this property with joint values. For example, if cities have unique names within the context of a country, we can create a property with the city name and country name appended and create a uniqueness constraint on that.

The primary function of a label is to provide semantic context for nodes. As discussed earlier, a node can have multiple labels, because a node can represent multiple things to the same system. As a corollary, a node can have multiple uniqueness constraints applied to it in context of the different labels applied to it. Labels can be added or removed from a node. While allowing for uniqueness constraints isn't the primary role of labels, it is important to note that without labels, adding uniqueness constraints isn't possible.

We can add our first city—New York as shown here:

```
neo4j-sh (?)$ CREATE (city:City{name:"New York",
country:"United States of America"}) RETURN n;
```

The output is as follows:

```
+-------------------------------------------------------------+
| n                                                           |
+-------------------------------------------------------------+
| Node[1]{name:"New York",country:"United States of America"} |
+-------------------------------------------------------------+
1 row
Nodes created: 1
Properties set: 2
Labels added: 1
```

In the query, `city` is a variable name just like `n` or `x`. While it can be anything, we recommend usage of readable and meaningful variable names in the query. We use variable names to ensure that we are referring to the same set of nodes, to use them in multiple parts of the query.
The output of each query on the Neo4j console can be divided into three parts:

- **Variable name**: This is the name of the value that is returned by the query.
- **Variable values**: These are the values that are returned by the query.
- **Modification summary**: This includes how much data is returned, everything that was (or not) modified, and the time taken to run the query.

The ID of the node created, is a part of the output, but should never be used to identify the node uniquely. Neo4j recycles IDs of any nodes that have been deleted. For example, if we delete the node we just created, the ID (1) of the node will be added to the free pool, and next time the server restarts, it might be reassigned to some other node we might create. Thus, using the generated Neo4j ID to identify a node is risky.

We can also create multiple cities in the same query with the following code:

```
neo4j-sh (?)$ CREATE
 (:City{name:"Mumbai", country:"India"}),
 (:City{name:"Chicago",
  country:"United States of America"}),
 (:City{name:"Las Vegas", country:"United States of America"}),
 (:City{name:"Los Angeles",
  country:"United States of America"}),
 (:City{name:"Toronto", country:"Canada"}),
 (:City{name:"London", country:"United Kingdom"}),
 (:City{name:"Madrid", country:"Spain"}),
 (:City{name:"Paris", country:"France"}),
 (:City{name:"Athens", country:"Greece"}),
 (:City{name:"Rome", country:"Italy"}),
 (:City{name:"Istanbul", country:"Turkey"}),
 (:City{name:"Singapore", country:"Singapore"}),
 (:City{name:"Sydney", country:"Australia"}),
 (:City{name:"Melbourne", country:"Australia"});
```

The output of the previous query is as follows:

```
+-------------------+
| No data returned. |
+-------------------+
Nodes created: 14
Properties set: 28
Labels added: 14
```
We can retrieve the cities we just created by using the following query:

```
neo4j-sh (?)$ MATCH (city:City{name:"New York"}) RETURN city;
```

The output of this query is as follows:

```
+-------------------------------------------------------------+
| city                                                        |
| +-------------------------------------------------------------+
| Node[1]{country:"United States of America",name:"New York"} |
+-------------------------------------------------------------+
1 row
```

We can also retrieve multiple cities in the same query as shown here:

```
neo4j-sh (?)$ MATCH (c1:City{name:"Athens"}),
       (c2:City{name:"Mumbai"}) RETURN c1, c2;
```

The output is as follows:

```
+------------------------ +------------------------ +
| c1                     | c2                     |
| +------------------------ +------------------------ +
| Node[7]{name:"Athens",country:"Greece"} |
| Node[9]{name:"Mumbai",country:"India"} |
+------------------------------ ----------------------------+
1 row
```

While we can return whole nodes as our result, it's generally advised to return only the data that is needed. It's also possible to alias the data that is returned. The following query returns the names of two cities as shown:

```
neo4j-sh (?)$ MATCH (c1:City{name:"Athens"}),
       (c2:City{name:"Mumbai"}) RETURN c1.name as first_city, c2.name as second_city;
```

The output is as follows:

```
+-------------------------- +-------------------------- +
| first_city | second_city |
| +-------------------------- +-------------------------- +
| "Athens"   | "Mumbai"    |
+-------------------------- +-------------------------- +
1 row
```
Indices

We have added a property `country` to every node labeled as `City`. We can search for cities belonging to a country as well. Searching without indexes is inefficient, and hence, it's a good practice to add an index for properties which we anticipate the nodes will be searched by. Let's see how this is done using an example as shown:

```
neo4j-sh (?)$ CREATE INDEX ON :City(country);
```

The output is as follows:

```
+-------------------+
| No data returned. |
+-------------------+
Indexes added: 1
```

Creating an index returns `Indexes added: 1`. However, at this point, an index may not have been added—but will be created. In our database, an index would already have been created. In larger datasets, the indexing will take time.

The index we have just created is called `schema index`.

Whenever we send Neo4j a Cypher query for execution, Neo4j will try reducing the queried graph to a small subgraph, and then try comparing which nodes have `properties:value` queried. In larger databases, the subgraph itself might have millions of nodes, and checking each node within the subgraph for the presence of the `property:value` pair would be time consuming. To avoid store scans, and to improve discrete lookup performance, we can declare an index on properties for a given label. It is also a good practice to restrict the subgraph using the label of the nodes we want the query to operate on. Since labels are indexed by default, finding nodes using labels is fast for Neo4j.

We can now search for cities in a country without worrying about performance. For example, to search all cities in the United States of America we will use the following query:

```
neo4j-sh (?)$ MATCH (c:City{country:"United States of America"}) RETURN c.name as City;
```
Adding flights to Neo4j

Since we have identified flights as entities, we will create them as nodes. To begin with, we should create a uniqueness constraint on the property code for the label :Flight as shown:

```sql
neo4j-sh (?)$ CREATE CONSTRAINT ON (flight:Flight)
    ASSERT flight.code IS UNIQUE;
```

The output is as follows:

```
+-------------------+
| No data returned. |
+-------------------+
Constraints added: 1
```

We can create a flight with its information as a standalone entity:

```sql
neo4j-sh (?)$ CREATE (flight:Flight {code:"AA9",
    carrier:"American Airlines", duration:314,
    source_airport_code:"JFK", departure:1300,
    destination_airport_code:"LAX", arrival:114})
RETURN flight.code as flight_code,
    flight.carrier as carrier, flight.source_airport_code
    as from, flight.destination_airport_code as to;
```
The output is as follows:

<table>
<thead>
<tr>
<th>flight_code</th>
<th>carrier</th>
<th>from</th>
<th>to</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;AA9&quot;</td>
<td>&quot;American Airlines&quot;</td>
<td>&quot;JFK&quot;</td>
<td>&quot;LAX&quot;</td>
</tr>
</tbody>
</table>

1 row
Nodes created: 1
Properties set: 5
Labels added: 1

This flight can now be connected to the source and destination cities by means of relationships. Relationships in Neo4j must have a direction while being created. In the absence of a specified direction while creation, the query will throw an error. Here's an example:

```cypher
MATCH (source:City {name:"New York"}),
    (destination:City {name:"Los Angeles"}),
    (flight:Flight{code:"AA9"})
CREATE (source)-[:HAS_FLIGHT]->(flight)-[:FLYING_TO]-(destination);
```

The output of this query is as follows:

```
WARNING: Only directed relationships are supported in CREATE (line 1, column 164 (offset: 163))

"MATCH (source:City {name:"New York"}),
    (destination:City {name:"Los Angeles"}),
    (flight:Flight{code:"AA9"})CREATE (source)-[:HAS_FLIGHT]-(flight)-[:FLYING_TO]-(destination)"
```

In the preceding example, we don't have a direction specified on the [:FLYING_TO] relationship.

If we provide a direction to the relationship in the Cypher query, Neo4j will create the relationship in the database.

Input the following query:

```cypher
MATCH (source:City {name:"New York"}),
    (destination:City {name:"Los Angeles"}),
    (flight:Flight{code:"AA9"})CREATE (source)-[:HAS_FLIGHT]-(flight)-[:FLYING_TO]->(destination);
```
Chapter 2

The output obtained is as follows:

```
| No data returned. |
+-------------------+
 Relationships created: 2
 Properties set: 2
```

The structure of the query is simple. We select the nodes that need to be linked and then create a relationship between them. In the preceding example, source, destination and flight represent variables that are temporarily used to hold the nodes between which the relationships have to be formed. :HAS_FLIGHT and :FLYING_TO are both relationship types. As discussed earlier, relationships can have properties.

![Image](image.png)

The type of a relationship is tied to the relationship. They are like labels in the semantic sense, but once a relationship is created with a type, it cannot be changed, nor can more types be added.

The direction of the relationship is denoted by the direction in which the arrow (\(\rightarrow\)) points. Both the direction and type are intrinsic to the relationship, and to change any of these, we need to delete the relationship and recreate it with the desired type and direction.

These two steps of creating the flight and then connecting them to the cities can be condensed into one by writing a slightly longer query, as shown here:

```
neo4j-sh (?)$ CREATE (flight:Flight {code:"AA920",
carrier:"American Airlines", duration:305,
source_airport_code:"LAX", departure:505,
destination_airport_code:"JFK", arrival:990})
WITH flight
MATCH (source:City {name:"Los Angeles"}),
(destination:City {name:"New York"})
CREATE (source)-[:HAS_FLIGHT]->(flight)-[:FLYING_TO]->(destination);
```

The output of this query is as follows:

```
| No data returned. |
+-------------------+
 Nodes created: 1
 Relationships created: 2
 Properties set: 7
 Labels added: 1
```
In the preceding query, we used `WITH` to pipe the result of the first part of the query to the second.

Let's create a few more flights using the queries in `flights.cqy`, which we downloaded at the beginning of this chapter.

**Traversing relationships**

Traversing relationships in Neo4j is done by specifying the path that we want to be matched. Queries can be open ended, like the one here, in which we haven't highlighted the direction in which we want the relationship to be traversed. Other open-ended queries might refrain from specifying the relationship type to be traversed or the node labels that would identify the subgraph that needs to be traversed.

Input the following query:

```
neo4j-sh (?)$ MATCH (source:City {name:"Los Angeles"})
-[:HAS_FLIGHT]->(f:Flight)-[:FLYING_TO]->
(destination:City {name:"New York"})
RETURN f.code as flight_code, f.carrier as carrier;
```

The output of this query is as follows:

```
+-----------------------------------+
| flight_code | carrier             |
+-----------------------------------+
| "UA1262"    | "United"            |
| "AA920"     | "American Airlines" |
+-----------------------------------+
2 rows
```

**Summary**

In this chapter, you learned that graphs can be described by describing a subgraph that contains all possible relationships, nodes, and properties; this is called specification by example. You also learned that entities should be modeled as nodes, and relationships must be used to denote semantic correlation between two entities. Nodes can have multiple labels, while relationships can have only one type. We also discussed the advantages of using labels apart from the semantic context they provide.
Where to buy this book

You can buy Neo4j Graph Data Modeling from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.