THE EXPERT'S VOICE® IN DATABASES

Practical Neo4j

RELATIONSHIPS ARE AS IMPORTANT AS THE BIG DATA THAT CONNECTS THEM

Gregory Jordan
Foreword by Jim Webber

Apress®

Contents at a Glance

PART 1

Getting Started

CHAPTER 1

Introduction to Graphs

What do Cisco, Walmart, and eBay have in common with many academic and research projects? They all depend on graph databases as a core part of their technology stack.

Why have such a wide range of industries and fields found a common relationship through graph databases? The short answer is that graphs can offer superior and consistent speed when analyzing deep, dense relationships and can do so with a flexible data structure.

As many developers can attest, one of the most tedious pieces of a web application or software project is managing the schema for its database. Although relational databases are often the right tool for the job, certain limitations—particularly the time as well as the risk involved to make additions to or update the model—invite the use or consideration of alternatives and complementary data storage solutions. Enter NoSQL.

When NoSQL databases, such as MongoDB and Cassandra, came along, they brought with them a simpler way to model data, as well as a high degree of flexibility—or even a schema-less approach—for the model. While document and key-value databases remove many of the time and effort hurdles, they were mainly designed to handle simple data structures. However, the most useful, interesting and insightful applications require complex data and yield a deeper understanding of the connections and relationships between different data sets.

Graph databases—another branch of databases in the NoSQL family tree—can offer the blend of simplicity and speed while permitting data relationships to maintain a first-class status. For example, Twitter's graph database, called *FlockDB*, more elegantly solves the complex problem of storing and querying billions of connections than their prior relational database solution. In addition to simplifying the structure of the connections, FlockDB also ensures extremely fast access to this complex data. Twitter is just one use case of many that demonstrates why graph databases have become a draw for many organizations that need to solve scaling issues for their data relationships.

While offering fast access to complex data at scale is a primary driver for adoption of graph databases, they also offer the same tremendous flexibility found in so many other NoSQL options. The schema-free nature of a graph database permits the data model to evolve without sacrificing any of the speed of access or adding significant and costly overhead to development cycles.

Poised at the intersection of graph database capabilities, the growth of interest, and the trend toward more connected large sets of data, this chapter demonstrates how the graph database will affect future web and mobile application development—specifically, how graph databases will grow as a leading alternative to relational databases.

I start with a quick overview of graph theory and a look at the main elements of a graph database. I proceed to show how graph databases compare to relational databases as well as other NoSQL options. I conclude the chapter with a look at use cases for graph databases.

Graph Theory

The history of graph theory begins with Leonhard Euler (pronounced “oiler”), the Swiss mathematician and physicist. Euler made many significant contributions to pure and applied mathematics over a more than 50-year academic career. His solution to the Seven Bridges of Königsberg problem in 1735 is considered to be the first theorem of graph theory and one of his most important contributions.[1]

The Seven Bridges of Königsberg problem was to find a path through the city that would cross each of the seven bridges connecting two large islands. Figure 1-1 highlights the bridges connecting the mainland and the two islands with oval markers.

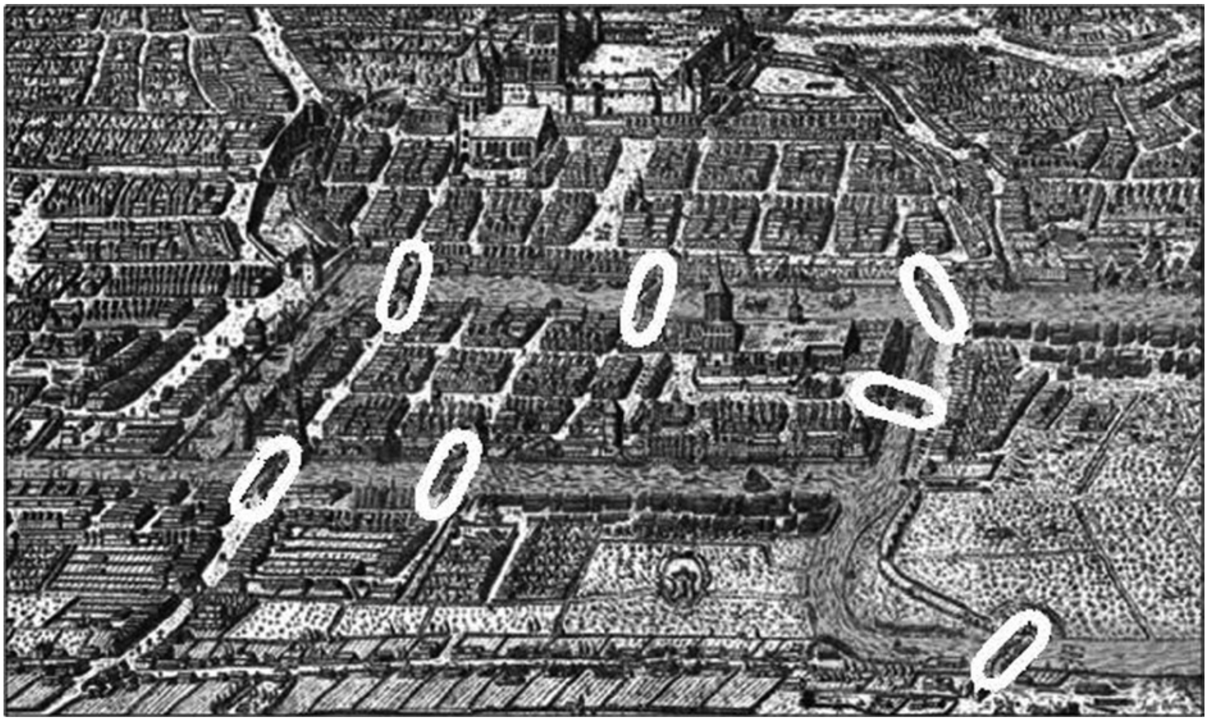


Figure 1-1. *The Seven Bridges of Königsberg*

Other conditions of the problem were that the bridges may not be crossed more than once and each bridge must be crossed completely. Euler’s subsequent treatise on the problem was written in 1736 and later published in 1741. Euler proved that the problem could not be solved but, more importantly, noted that the most relevant aspect of the problem is the order in which the bridges were crossed. In creating this singular, fundamental approach, Euler could examine the problem in abstract terms. His more focused methodology considered only the mainland, the islands, and the bridges that connected them.

[1]http://www.ams.org/journals/bull/2006-43-04/S0273-0979-06-01130-X/S0273-0979-06-01130-X.pdf

In graph theory, the mainland and islands are what is referred to as *vertices* (the plural of *vertex*). Each bridge that connects two vertices is known as an *edge*, which, for the purposes of graph theory, serves to identify which pair of vertices is connected by that bridge. As you can see in Figure 1-2, the components of the problem are broken down into four vertices connected by seven edges. The final mathematical structure that represents all the vertices and edges is called a *graph*.

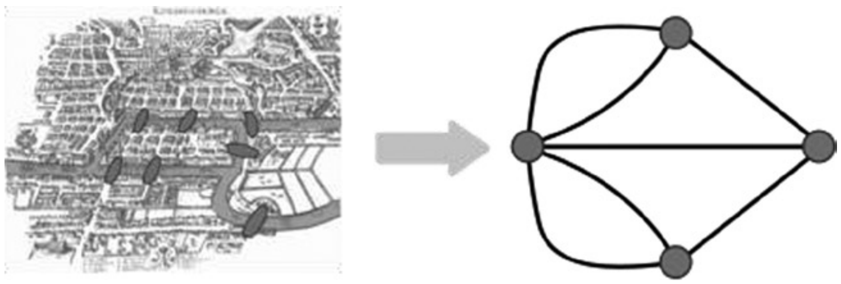


Figure 1-2. *The Seven Bridges of Königsberg problem displayed as Euler's graph representation*

■ **Note** A deep understanding of graph theory is not essential to working with graph databases. For those readers who want to dive further into graph theory, Richard J. Trudeau's *Introduction to Graph Theory* (Dover, 1993) provides a more thorough discussion.

A common mistake is to refer to the item in Figure 1-3, and items similar to it, as a *graph*. Although graph data or diagrams may be contained within a chart, the terms *graph* and *chart* are not synonymous.

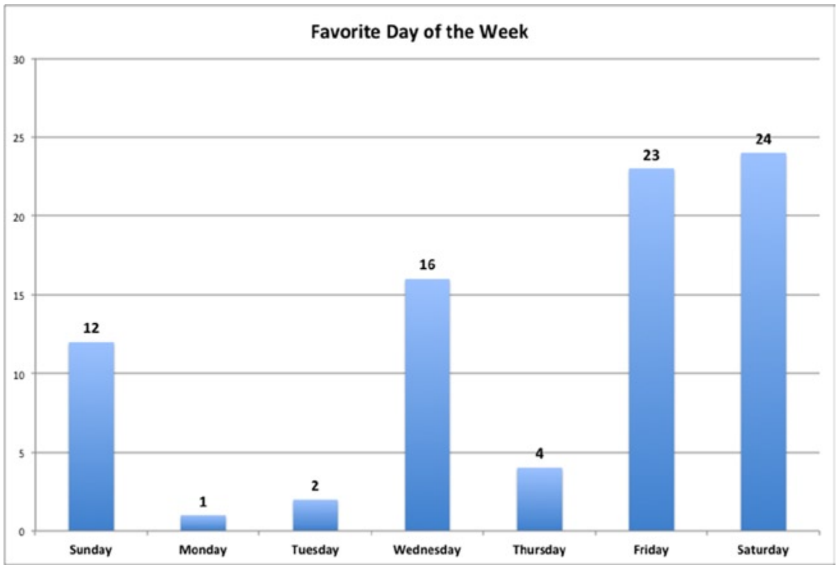


Figure 1-3. *Bar chart*

Graph Databases

In its simplest form, a *graph database* is a set of vertices and edges. Another way to picture graph databases is to view the data as an arbitrary set of objects connected by one or more kinds of relationships. This section defines and expands on the most essential components of a graph database—specifically, how they occur within and apply to the graph database, Neo4j.

Nodes and Relationships

When discussing graph databases, vertices are more commonly referred to as *nodes* and edges are more commonly referred to as *relationships* (Figure 1-4). While these two pairs of terms may be used interchangeably, this book follows the more common usage.

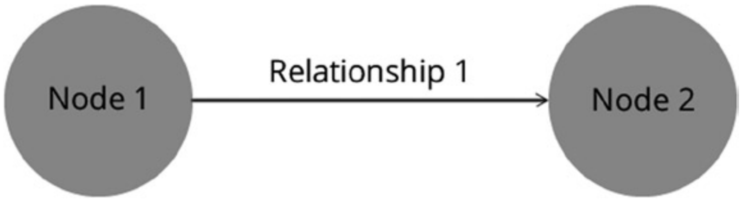


Figure 1-4. *Two nodes connected by a relationship*

A node can be thought of as an object with any number of properties. Unlike the keys that connect rows within a relational database, relationships within a graph database can also have properties.

Labels

Starting with the 2.0 version of Neo4j, the concept of labels was introduced as a way to group nodes. As the example in Figure 1-5 demonstrates, you can define a node as "Person" and then provide additional values for each property of the node as necessary. By grouping nodes in this way, we can query the graph to show common subsets of what are essentially node types. Labeling of nodes also offers a way to enforce modeling constraints when necessary, as well as to increase the speed at which data can be accessed through improved indexing.

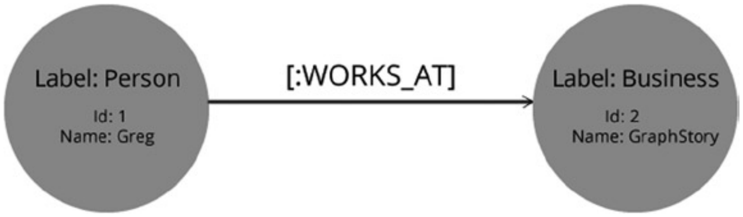


Figure 1-5. *Labels provide a way for nodes to be grouped*

Traversal

The most common method for querying a graph is by performing a *traversal*. In a traversal operation, the query begins with a single node that follows a path of relationships over connected nodes. Neo4j's traversal API allows you specify this path, essentially creating a subgraph of nodes and relationships. The shortest path has a length of zero, which is a single node without returning its relationships as part of the query. When a path has a length of one, the path can contain a relationship to another node or, as shown in Figure 1-6, even back to the same node.

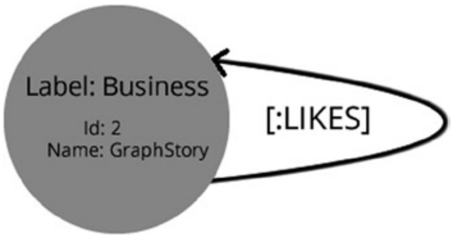


Figure 1-6. *A node that features a relationship back to itself*

Indexes

Like many other databases, Neo4j relies on an index to do an explicit look-up for a specific node or relationship. While it is possible to traverse the graph to find the node or relationship, it is sometimes more performant to allow indexing to handle the request. For example, when looking a specific "Person" node, you could query the index by a unique identifier such as a username or other unique key.

Relational Databases and Neo4j

When comparing graph databases to relational databases, one thing that should be clear upfront is that data affiliation does not have to be exclusive. That is, graph databases or other NoSQL options will likely not take over or replace relational databases. Clear and well-defined use cases will involve relational databases for the foreseeable future. Matt Aslett, research director of data management for 451 Research, has observed the growth of graph databases, specifically Neo4j, in which a relational database might have been otherwise used, and he notes that "there is a tipping point, but that will take some time."[2]

Undertaking the task of transforming an existing functional and manageable relational database into another database type is sometimes necessary. Relational databases may be poor fits for the goals of certain data for a number of reasons and use cases.

For example, the limitation on how a relationship is defined within a relational database is one reason to consider switching to a graph database such as Neo4j. As mentioned earlier in this chapter, relationships in the graph can, like nodes, have properties of their own. With that capability, it would be fairly trivial to add in a property on a graph relationship that was not defined when the relationship began. Although creating a *join table* (as it is known in the relational database world) that brings together two disparate tables is a common practice, doing so adds a layer of complexity. Chapter 3, which addresses data modeling with Neo4j, includes diagrams of graph models and how they compare to modeling with a relational database.

[2]http://techcrunch.com/2014/02/02/neo4j-a-graph-database-for-building-recommendation-engines-gets-a-visual-overhaul/

Another reason you might consider moving to a graph database is to avoid the half-measures and workarounds you must use to make your model fit within a relational database. A join table is created in order to have metadata that provides properties about relationships between two tables. When a similar relationship needs to be created among other tables, yet another join table must be created. Even if it has the same properties as the first join table, it must be created in order to ensure the integrity of the relationships. A certain type of relationship—such as "LIKES"—can exist among more than just two types of nodes. In fact, the relationship type could be applied to all types of nodes.

Another reason to favor graph databases over relational database is to avoid what might be referred to as "join hell." The joins required to connect two tables are often trivial, but those types of joins provide the least expressive data. When the application requires data that connects several tables, it is then that expense of joins begins to manifest itself in both the complexity and as well as diminished performance. In addition, the nature and depth of the query would need to be known ahead of time, or the query would need to be dynamically generated.

Despite the differences between graph and relational databases, there are a few similarities. A significant similarity is that both can achieve what is known as *ACID compliance. ACID*—Atomicity, Consistency, Isolation and Durability—is a set of principles guaranteeing that transactions completed by the database are processed reliably. In Neo4j, the Enterprise edition is fully ACID in high-availability clustering, whereas the Community edition is *eventually consistent*.

NoSQL and Neo4j

Graph databases are not the only alternative or complementary solutions to the shortcomings of relational databases. Although the first use of the term *NoSQL* dates from the late 1990s, it was only toward the end of the 2000s that NoSQL options became more focused and could be set into one of four different sectors or families: *key-value*, *column-family*, *document,* and *graph* databases.[3] Another group is the multimodel category, which includes combinations of concepts and features from at least two of the four main groups.

■ **Note** Contrary to the assumption in some quarters, *NoSQL* does not stand for "No to SQL." The proper sense of the acronym is "Not only SQL"—referring to alternatives to the relational database.

Key-value stores represent data by storing large sets of values, with each value based on a key. This simple data structure allows related applications to store its data in a schema-less way. The column-family database, modeled after Google's BigTable, can be described simply as rows of objects that contain columns of related data. As with key-value stores, column-family databases also have key values pairs that represent a row. Document databases represent a collection of "documents"; each one has its own collection of keys and values. In some ways, documents contained within a document database are like rows in relational database. In addition, querying against a unique id or key is a typical method used to retrieve a document.

The first big difference between graph databases and other NoSQL categories is the data model. Each type of node can have any number of properties. In addition, those properties can be changed over time, which provides a model that does not require a schema. This schema-less nature is certainly not unique in the NoSQL world, but when you consider that nodes can have arbitrary relationships that do not need to be determined ahead of time or carefully modeled in after an initial release, the difference between graphs and other NoSQL options begins to take shape. When you couple that with the fact that arbitrary relationships can also have any number of their own configurable properties, the difference is even clearer. Finally, because graphs can be quickly adapted to changes in business needs, especially in making connections between data, organizations are enabled to ask the right questions from the data as the needs arise, and those questions do not have to be precisely identified prior to data capture.

---

[3]`http://blog.monitis.com/index.php/2011/05/22/picking-the-right-Nosql-database-tool/`

Summary

This chapter provided a brief overview of graph theory as well as a look at the main elements of a graph database. Graph databases were compared to relational databases as well as to other NoSQL options, together with some use cases for graph databases. The next chapter covers how to install Neo4j quickly and how to test out its querying capability with its web-based UI and console tools.

CHAPTER 2

Up and Running with Neo4j

This chapter covers the requirements for running Neo4j as well as the steps for installing an instance of the Neo4j database on your computer. To set you on the path to mastering data management with Neo4j, I introduce the Neo4j Browser tool and walk you through the basics of the Neo4j query language, Cypher.

Neo4j

Neo4j began its life in 2000, when Emil Eifrem, Johan Svensson, and Peter Naubauer—the creators of Neo4j—began to notice a significant amount of overhead in both the performance and work required in one of their applications. The first and most significant aspect of the overhead could be traced to the mismatch of their content management system's model with the relational database. While the properties of the model could be stored in and retrieved from tables with relative ease, they observed that connections between the data imposed significant processing time for queries. Moreover, the performance of the queries grew worse as the connections among the data became more complex. Finally, the time and effort that was required to manage those relationships placed even more overhead on the application's development lifecycle.

After seeking out alternatives and performing a few rounds of research, they began to build out Project Neo. Neo aimed to introduce a database that offered a better way to model, store, and retrieve data while keeping all of the core concepts—such as ACIDity, transactions, and so forth—that made relational databases into a proven commodity.

Subsequent research and development has propelled the Neo4j to the top spot in popularity, justifying the tagline associated with the Neo4j logo on promotional materials, "The World's Leading Graph Database."[1] As you will come to see after working with Neo4j on your own, it fits extremely well with many different use cases, domains, and industries.

Requirements and Installation

The installation of Neo4j is straightforward and, regardless of whether you prefer Windows, Linux, or Mac, it should take very little time to get running once it has been downloaded. If you are ready to get started with the quick install, then browse to neo4j.com/download for a 30-day trial of the enterprise version. Click the download link and then choose the version for your operating system. If you run into problems downloading from the neo4j site, you can also visit `http://www.graphstory.com/practicalneo4j` and go to the download section to get the specific version as it applies to the remainder of this book.

[1]`http://db-engines.com/en/ranking/graph+dbms`

■ **Note** In addition to installing a version of Neo4j on your local machine, you can visit `http://www.graphstory.com/practicalneo4j` to setup a free, fully configured Neo4j instance of the enterprise version for personal use. You will be provided with your own free trial, a knowledge base, and email support from Graph Story.

Requirements

The requirements in Table 2-1 apply to a single instance of Neo4j. In terms of capability and performance for a single instance, memory and disk capability are the primary performance constraints. The amount of memory impacts the graph size that can fit in memory and disk I/O capability affects read/write performance.

Table 2-1. *Requirements for Running Neo4j*

	Minimum	Recommended
CPU	Intel Core i3	Intel Core i7
Memory	2GB	16-32GB
Disk	10GB SATA	SSD with SATA
Filesystem	ext4	ext4, ZFS

Versions

As of this writing, Neo Technology, the commercial entity that supports the ongoing development of Neo4j, offers a community license as well as enterprise subscriptions. This book uses the enterprise version, which includes the most critical features for exploring Neo4j. With the enterprise edition, the pricing and feature set has been set to match the current operational stage of a business. For example, the personal edition of Neo4j is in line with an early-stage or bootstrap company.

■ **Note** The types of licenses can be found in Table 2-2, which display only some of the more pertinent differences in capability and support with Neo4j. The license types are those available at time of writing publishing and are likely to evolve.

Table 2-2. *Neo4j License and Feature List*

	Community	Personal	Startup	Enterprise
Primary Features				
Property Graph Model	X	X	X	X
Native Graph Processing	X	X	X	X
Native Graph Storage	X	X	X	X
ACID	X	X	X	X
Cypher	X	X	X	X
Language Drivers	X	X	X	X
REST API	X	X	X	X
Memory	X	X	X	X
Disk	X	X	X	X
Filesystem	X	X	X	X
Performance and Scalability				
High-Performance Cache		X	X	X
Clustering		X	X	X
Online Backup		X	X	X
Advanced Monitoring		X	X	X
Support				
Commercial Email Support			X	X
Commercial Phone Support				X
Support Hours			10 x 5	Up to 24 x 7
License				
Production instances		3	3	3+
Test instances		3	3	3+
Developer Support		Up to 2	2	3+

Java

The "j" in Neo4j stands for *Java*, and the Java Development Kit (JDK) is required to run it. So before unpacking the download archive, make sure you have Oracle's JDK installed on your computer. If you already have the JDK installed, make sure it is at least version 7. If you need to install it, then be sure to use the latest stable version of JDK7. After you have installed JDK7 or verified that it has already been installed, you can proceed to the next section depending on your preferred operating system

■ **Note** To get you up and running as quickly as possible, this chapter uses the console to run Neo4j.

Installation

After downloading the version of Neo4j that is compatible with your operating system, follow the appropriate set of steps below.

■ **Note** Throughout this book, {NEO4J_ROOT} refers to the top-level installation directory for Neo4j.

Windows

Neo4j provides an installer version for Windows, but for the exercise in this chapter we will use the console:

1. Ensure that you have Java version 7 or higher running on your computer.
2. Extract the zip into a preferred directory on your computer.
3. Double-click on `{NEO4J_ROOT}\bin\Neo4j.bat`.
4. Open a browser and go to `http://localhost:7474`.
5. Stop the server by executing “Ctrl-C” in the corresponding open console window.

Linux/Unix

1. Ensure that you have Java version 7 or higher running on your computer.
2. Extract the archive into a preferred directory on your computer.
3. Open a command prompt and change directory to `{NEO4J_ROOT}\bin`.
4. Run the command `./neo4j start`.
5. Open a browser and go to `http://localhost:7474`.
6. Stop the server by executing `./neo4j stop` in the console.

Mac OSX

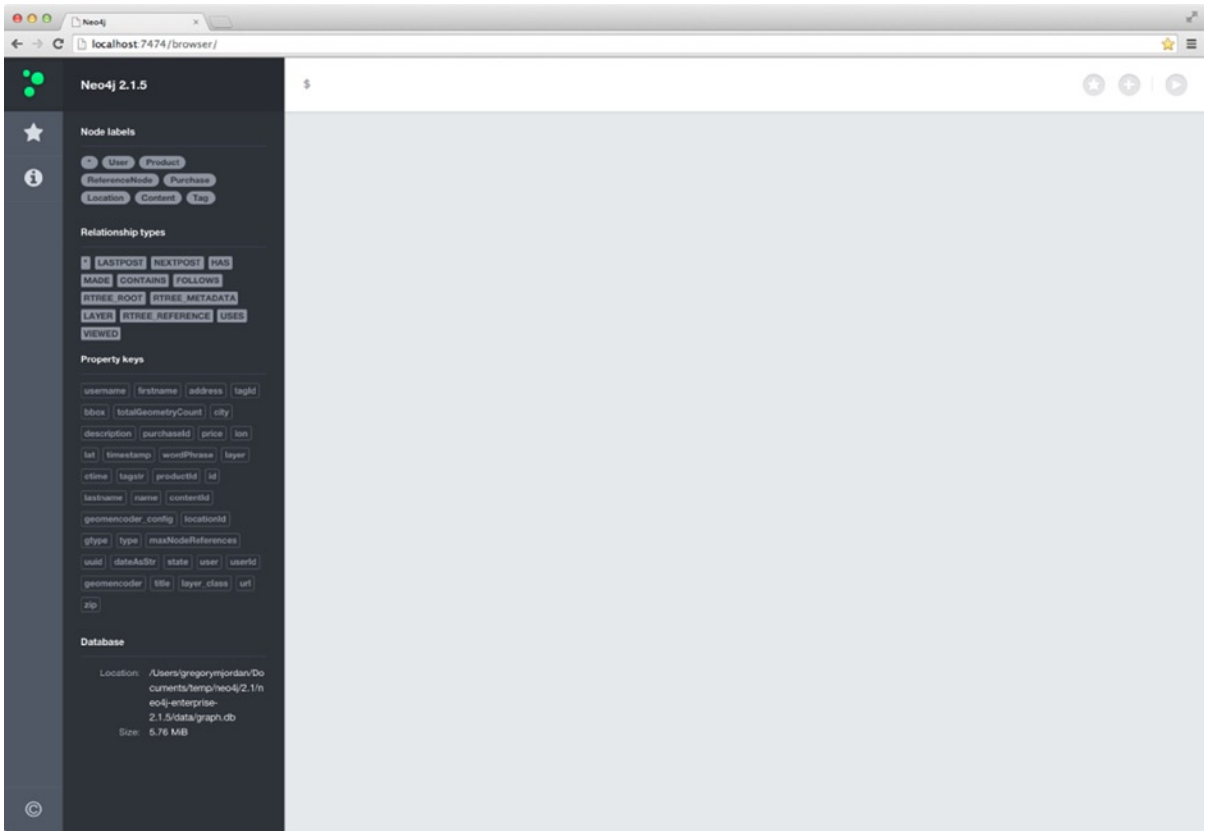
Ensure that you have Java version 7 or higher running on your computer. While it is possible to follow the Linux/Unix install instructions for Mac OS, users familiar with using Homebrew can install the latest stable version of Neo4j with the command, `brew install neo4j && neo4j start`.

This will provide a Neo4j instance running on `http://localhost:7474`. The installation files will reside in `/usr/local/Cellar/neo4j/community-{NEO4J_ROOT}/libexec/` —available to tweak settings and symlink the database directory if desired. After the installation has completed, you can run Neo4j from the terminal.

The server can be started in the background from the terminal with the command `neo4j start` and then stopped again with `neo4j stop`. The server can also be started in the foreground with the neo4j console, and it can send the log output to the terminal.

The Neo4j Browser

One of the most useful tools included with the database is the Neo4j Browser, a web-based shell (Figure 2-1). Version 2.x of Neo4j contains significant enhancements to the features, speed, and visualization tools over the previous incarnations of the web-based tool.


Neo4j
localhost:7474/browser/
Neo4j 2.1.5
$
Node labels
* User Product ReferenceNode Purchase Location Content Tag
Relationship types
* LASTPOST NEXTPOST HAS MADE CONTAINS FOLLOWS RTREE_ROOT RTREE_METADATA LAYER RTREE_REFERENCE USES VIEWED
Property keys
username firstname address tagId bbox totalGeometryCount city description purchaseId price lon lat timestamp wordPhrase layer ctime tagstr productId id lastname name contentId geomencoder_config locationId gtype type maxNodeReferences uuid dateAsStr state user userId geomencoder title layer_class url zip
Database

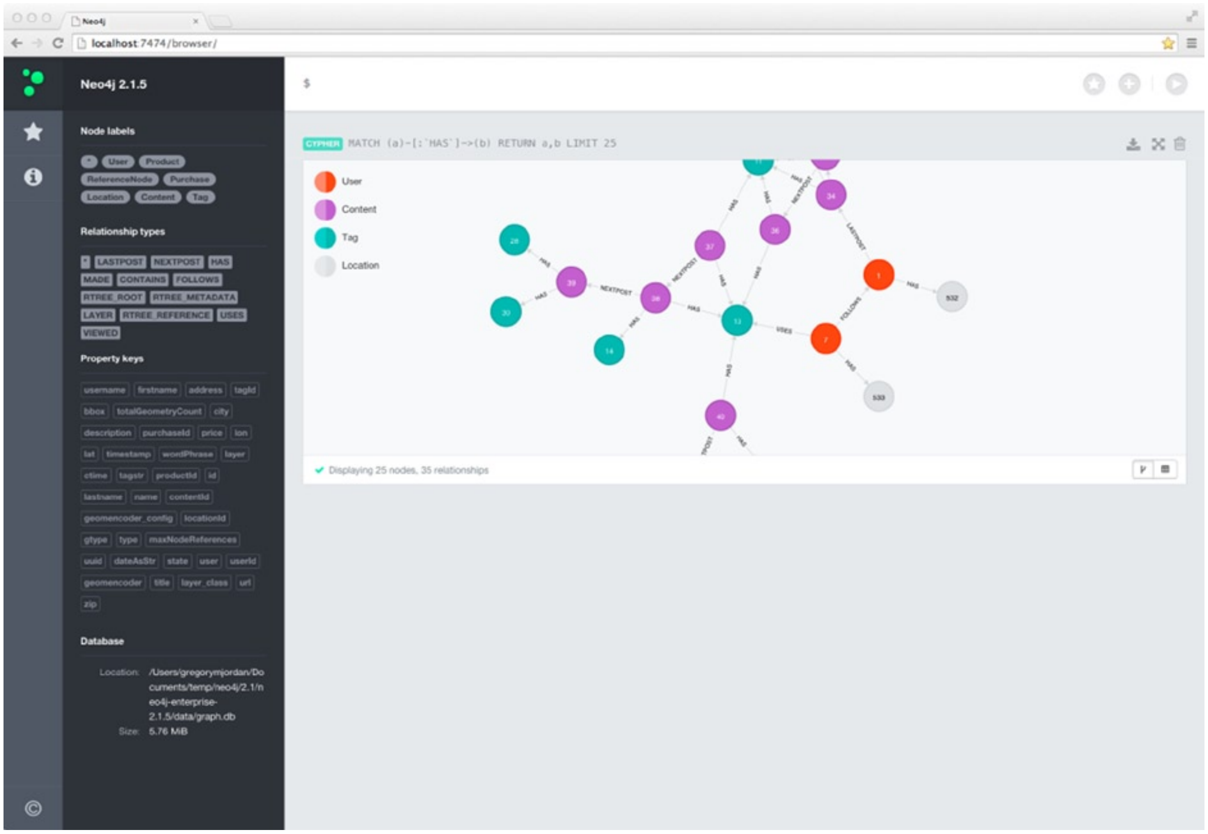
Location: /Users/gregorymjordan/Documents/temp/neo4j/2.1/neo4j-enterprise-2.1.5/data/graph.db
Size: 5.76 MiB

Figure 2-1. *The Neo4j Browser*

In addition to execution of the commands to perform CRUD (Create, Read, Update, and Delete) operations against the Neo4j database, the web interface provides helpful features to inspect the connected database instance as well as the system configuration settings. As in Figure 2-1, the Neo4j Browser shows labels, relationship types, and property keys that are contained within the data.

■ **Tip** The web-based shell uses a default value and can be accessed using the port number 7474. However, you can change the port address by updating the server configuration located in the `{NEO4J_ROOT}/conf/neo4j-server.properties` file using the setting for `org.neo4j.server.webserver.port`. Changing this setting might be necessary if there are restrictions on your network for port ranges.

When a populated database is accessed through the Browser, many of the top-level properties of Neo4j are displayed. For example, by clicking on one of the relationship types in Figure 2-2, a query is executed and displays sets of related nodes that contain the node ID of both the “start” node and “end” node.

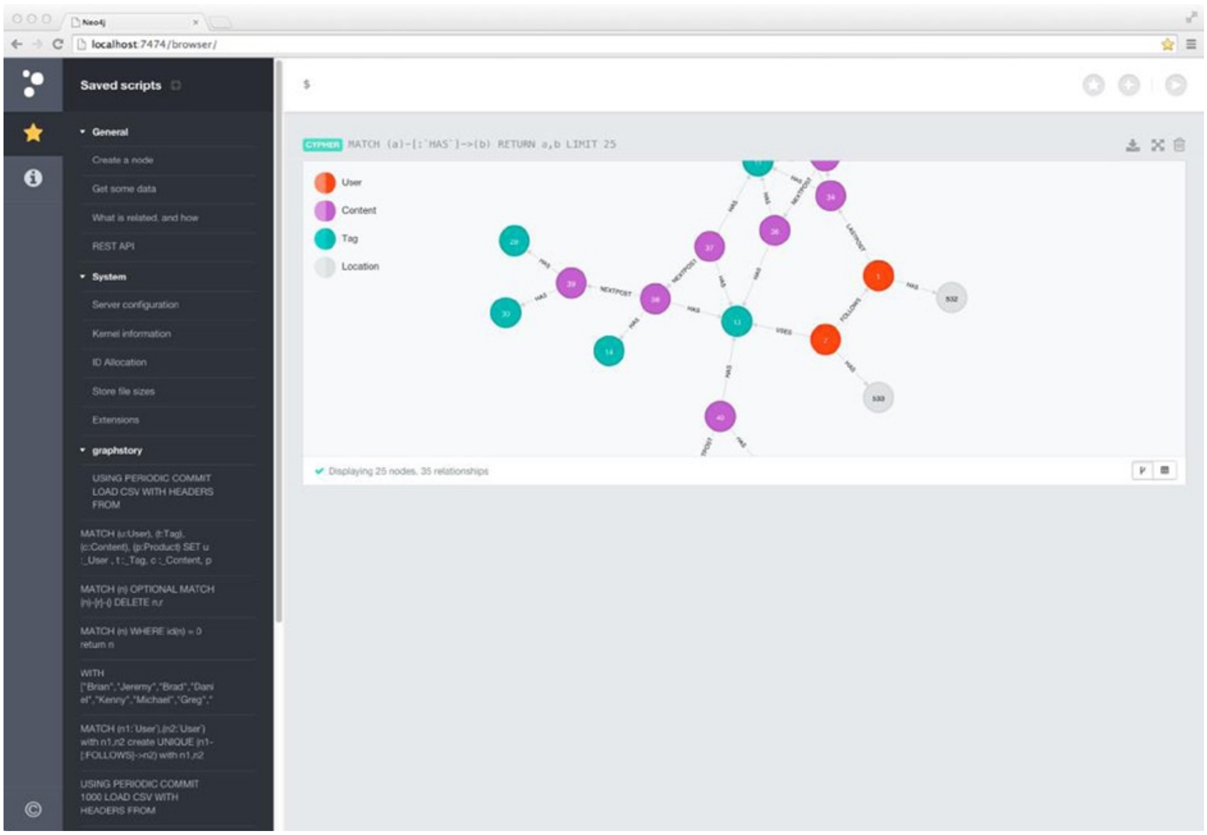

Neo4j
localhost:7474/browser/
Neo4j 2.1.5
Node labels
* User Product ReferenceNode Purchase Location Content Tag
Relationship types
* LASTPOST NEXTPOST HAS MADE CONTAINS FOLLOWS RTREE_ROOT RTREE_METADATA LAYER RTREE_REFERENCE USES VIEWED
Property keys
username firstname address tagId bbox totalGeometryCount city description purchaseId price lon lat timestamp wordPhrase layer ctime tagstr productId id lastname name contentId geomencoder_config locationId gtype type maxNodeReferences uuid dateAsStr state user userId geomencoder title layer_class url zip
Database
Location: /Users/gregorymjordan/Documents/temp/neo4j/2.1/neo4j-enterprise-2.1.5/data/graph.db
Size: 5.76 MiB
$
User
Content
Tag
Location
HAS
NEXTPOST
LASTPOST
FOLLOWS
USES

```
MATCH (a)-[:`HAS`]->(b) RETURN a,b LIMIT 25
```

Displaying 25 nodes, 35 relationships

Figure 2-2. *The Neo4j Browser showing a visual graph result after executing a Cypher command*

Figure 2-3 displays new tools in 2.x that offer shortcuts to perform common tasks. For example, one the new features available is the ability to save and archive Cypher queries for later use. In addition, some shortcuts provide a stubbed-out version of Cypher statements, such as the “Create a node” option under the *General* section.



```
MATCH (a)-[:`HAS`]->(b) RETURN a,b LIMIT 25
```

Displaying 25 nodes, 35 relationships

Figure 2-3. *The Neo4j Browser showing quick commands and saved scripts*

Introducing Cypher

Cypher is the declarative query language used for data manipulation in Neo4j. It is similar in many ways to how a relational database depends on Structured Query Language (SQL) to perform data operations. However, Cypher is not yet a standard graph database language that can interact with other graph database platforms. If you have some familiarity with SQL, you will probably be able grasp Cypher quickly. In addition, the expressive and relatively simple nature of Cypher allows it to be a tool that can be used beyond the confines of an organization's technology-centered groups, similarly to the way SQL is used in an ad hoc way outside many IT departments.

■ **Note** A *declarative language* is a high-level type of language in which the purpose is to instruct the application on what needs to be done or what you want from the application, as opposed to how to do it. A *procedural language*, by contrast, instructs the application what to do, step by step.

While there are a number of language drivers as well as a native API to execute CRUD operations, Cypher is the primary access tool for Neo4j.

Cypher will be covered in much greater detail in Chapter 4, but it is apposite at this point to get a feel for this centerpiece of the Neo4j world from the following simple examples of Cypher queries.

Create

CREATE is analogous to an INSERT statement in SQL. Listing 2-1 is a very basic example of a CREATE operation.

Listing 2-1. Example CREATE query statement

```
CREATE (n:Business { name : 'GraphStory', description : 'Graph as a Service' })
```

Start

In the latest version of Neo4j, the START clause has become an optional part of a read operation. The counterparts in SQL are portions of the FROM and WHERE clauses. In Listing 2-2, the lowercase business represents the variable being returned, which is closer to the SELECT clause in SQL, but in this case the business variable also returns all of the properties (or *columns*, as they are referred to in a relational database). The Business index is equivalent to a table in the relational database world, and the name='GraphStory' portion is similar to a WHERE clause.

Listing 2-2. Example START query statement on the index Business

```
START business=node:Business (name = 'GraphStory')
RETURN business
```

Match

A MATCH clause represents a similar operation as a JOIN would in SQL. The Cypher statement in Listing 2-3 displays how to return a collection of people who like GraphStory.

Listing 2-3. Sample MATCH query statement in earlier versions of Neo4j

```
START business=node:Business (name = 'GraphStory')
MATCH people-[:LIKE]->business
RETURN people
```

A shorter way to represent the same result is to use Label, which excludes the START clause. The example shown in Listing 2-4 is the current recommended way of executing a MATCH result.

Listing 2-4. The recommended way to execute a MATCH query statement

```
MATCH person-[:LIKE]->(b:Business { name: "Graph Story"})
RETURN person
```

Set

The SET statement is analogous to an UPDATE statement in SQL. Listing 2-5 is a basic example of a SET operation.

Listing 2-5. Example MATCH query statement

```
MATCH (b:Business { name: 'GraphStory' })
SET b.description = 'The Leading Graph Database as a Service Provider'
RETURN b
```

Summary

This chapter provided a quick overview of Neo4j, including the requirements for running the server in your local environment, as well as the steps to install for Windows, Linux/Unix, and Mac OSX. It also introduced the Cypher query language. The next chapter will discuss modeling for Neo4j and will begin to explore the Cypher language a bit more.

PART 2

Managing Your Data with Neo4j

CHAPTER 3

Modeling

This chapter reviews the elements that make up Neo4j and the proper way to view relationships within the model. This chapter recurs to some of the concepts first addressed in earlier chapters from the perspective of how modeling is handled within Neo4j. It also explores a little more of the Cypher language, but only as it applies to our modeling effort. Chapter 4 will deal with Cypher in greater depth. Finally, this chapter goes over some common models found in various domains and looks at some of the common issues that data architects and developers face when modeling for a graph. The chapter will begin with an overview of data modeling and why it can help ensure your application starts on a solid foundation.

Data Modeling

If you are comfortable with the concepts of modeling, feel free to skip ahead to the next section. If, however, you are still fairly new to data modeling or just need a refresher, this section will provide a quick conceptual overview and cover the basics for proper modeling.

Data Modeling Overview

Data models serve as visual representations of the specific data that will reside within database and almost exclusively in support of an external application. The models represent objects, such as a User or Shopping Cart, the connections between the objects, and the rules that determine how the objects are stored within the database. The model typically concentrates on what data will be stored and how it will be organized. The specific functions or how the application will operate on the model should be considered separate from the modeling tasks. One common analogy of the model are the blueprints of a house, where there is direction as to how the spaces are defined but the exact contents remain to be determined after the main construction is completed.

In addition, for the some areas the data model is independent from the constraints of the database platform. As you will see in the later sections of this chapter, there is a divergence that takes place when modeling relationships within a relational database versus modeling within Neo4j. In either event, the model still serves as the high-level, conceptual representation for all of the data points.

Why Is Data Modeling Important?

Regardless of whether you are using a graph database like Neo4j or a relational database, modeling is a critical part in helping to ensure your application's data can be stored and retrieved as efficiently as possible. In the case where there is a dedicated *database administrator* (DBA), the model is provided as a diagram—almost like a set of "blueprints"—to use as a guide while creating the actual database. In most cases, the model represents the basics of the tables, the primary and foreign keys, and the meta-information on properties, such as their type. The model might also contain constraint information, such as whether a value of field is required or can be null or empty.

Although the model can and likely will evolve over time, maintaining it in a diagram format or similar way is important to ensure an efficient and cohesive design. It could be argued that for some applications either the domain is limited enough or the objects representing the model are so well defined and documented that a model diagram is unnecessary. In addition, it has been suggested that the time involved to create a model diagram can slow down the development process.

However, most applications that start small will grow over time and the object code will—at some point—probably be passed from the initial developers to a new set of developers. Without a diagram to quickly demonstrate all of the data points represented within an application, the time and effort involved to explain the model will likely grow as well as make it much more difficult to most efficiently add, update, or remove specific pieces of the model.

Data Model Components

The data model is developed in the first stage of the project and will evolve over time. Even as relational databases have changed over the past forty years, they have retained certain design limitations, which, in turn, makes the initial data modeling task a critical path within the scope of an application development project. Although NoSQL options have helped the outcome of projects by lowering the risk of modifications to the model, the task of modeling is still critical to successful application development.

In the data modeling stage, whether with an agile focus or otherwise, the project team, specifically analysts and developers, will usually begin by having discussions with the application owners to understand the requirements of the model. These discussions should yield at least one important result, which is an *entity-relationship* (ER) diagram. The ER diagram is an important resource for an application project team because it provides a common understanding of how the application's data will be represented.

Entity-Relationship Model

Although many variants of the theme existed prior to it, the entity-relationship model is credited to Peter Chen in his 1976 paper, "The Entity-Relationship Model: Toward a Unified View of Data."[1] Chen's original description and design was adapted to more common usage today for data analysts and administrators. The ER model is specifically useful because of how well it maps to the structure of a relational model.

In addition, the ER model is fairly simple to create and can be understood by all members of the team and wider organization with minimal instruction as well as act as the instructions to one or more team members on how to specifically construct the database as it applies to the platform in use. Perhaps the most important aspect of the ER model is that it acts as a universal way to communicate. Without its ubiquity, the method and manner of describing and visualizing data models could vary from project to project.

Entities

Entities are characteristically viewed as the central objects within the ER model. Most often data modelers will strive to use terms that are easily recognizable to each member of the project team in order to describe the entity. Conversely, you should stay away from terminology that is not commonly used or not the default within the domain or industry. For example, when modeling applications that deal with constructing residential areas, it would be more common to use the word "house" rather than "abode"—even though they are synonyms.

We can see in Figure 3-1 that the model diagram employs the use of a box—a standard shape to symbolize entities. In some model diagrams, the entities—as well as the relationships and attributes—will be shown in specific colors to further visually distinguish each part of the model.

[1]Peter Chen, "The Entity-Relationship Model: Toward a Unified View of Data," September 22–24, 1975, ACM Transactions on Database Systems, Vol. 1, No. 1 (March 1976), pp. 9–36.

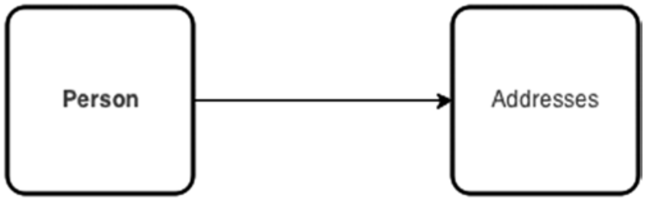


Figure 3-1. *A simple ER model*

Relationships

As you might surmise from Figure 3-1, relationships represent the connections or associations between entities. In most cases, the relationship can be expressed using a verb. For example, if you were going to connect people who use your application with where they live, you would typically express it as "a user has addresses." In addition, you would normally want to address cardinality, which measures how many times one entity type might be connected to another distinct entity type. To express that a user has many addresses, the cardinality would be denoted as "1:M" (one-to-many).

The relationship objects within the ER diagram usually address the optionality and direction of the association between entities as well. Addressing the optionality of the relationship can be handled conveniently through its cardinality. For example, you can express an optional relationship by showing its cardinality as "0:1". The direction of the relationship—often referred to as the *parent-child*—is shown by using an arrow pointing from the parent entity to the child entity, e.g. Person ➤ Address. In addition to arrows and lines with numeric representations, cardinality, direction, and optionality can be expressed graphically. Figure 3-2 displays the special symbols that are often used in ER diagrams to express relationships between entities: in this case, a relationship of one to many as one person could have many addresses.

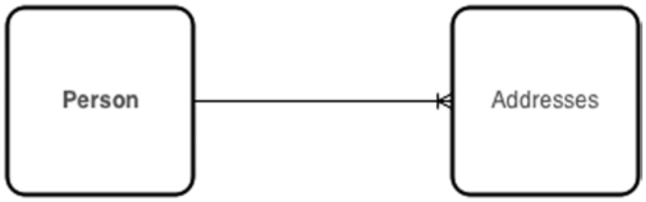


Figure 3-2. *A simple ER model*

Attributes

Attributes act as an identity, characteristic, or descriptor for an entity. For example, a User entity might use an identity attribute (also known as a *key*) which is named "Person ID". The "Person ID" attribute can be used to identify a specific instance of that entity type. In the case of descriptor attribute, the User entity might include "Person Name" or "Person Email."

In some entities, a single attribute might contain one or more of its sibling attributes, which is referred to as a composite attribute. For example, the Address entity could have the attributes number, street, city, state, and ZIP code, which together form the composite attribute called "Address", shown in Figure 3-3.

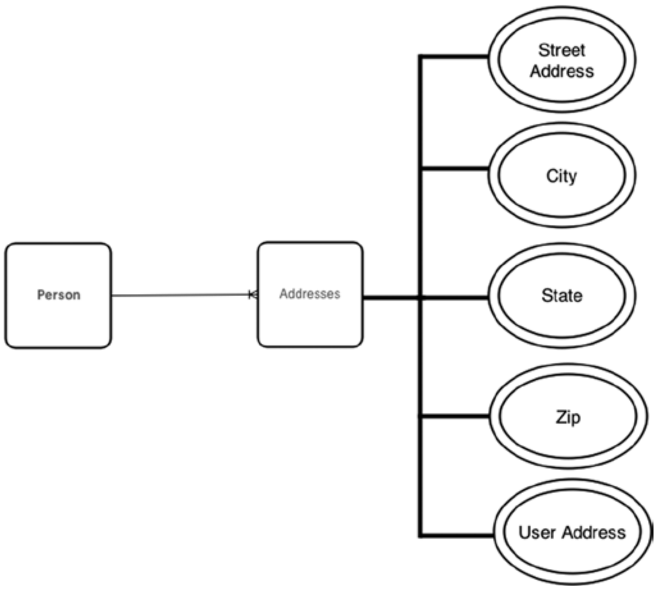


Figure 3-3. *A ER model with attributes*

Challenges in Using Entity-Relationship Modeling with Neo4j

Traditional entity-relationship models accept information and content that can be freely and easily contained within a relational database and are typically only a good match for a relational structure. In fact, they are insufficient for models in which the data cannot be suitably represented in relational form, as is the case with frequently changing, semi-structured data. One of the biggest challenges for many applications is the possible frequency and scope of change to the way model is structured. As detailed in Chapter 1, these types of modifications for relational systems are nontrivial, involve at least moderate risk, and are often significant causes for changes from one database platform to another.

Modeling with Neo4j

This section begins to build out the model for the application to be discussed in the later chapters of the book. The model contains some likely familiar themes in terms of its structure and includes five areas that have been identified as the most significant portions of both consumer and business data: *social*, *intent*, *consumption*, *interest*, and *location* graphs. These five graph types are certainly not the only use cases that make sense for Neo4j, but they are in wide use and intrinsically shaped.

As part of our examination of the graph model for these areas, we will examine the companion model structure as designed for a relational database. As noted in the data model overview section, a divergence takes place when modeling relationships within a relational database versus modeling within Neo4j. The divergence is not significant in terms of the data being captured, but, as Table 3-1 shows, the main components of an entity-relationship model in Neo4j may be known by different names and take vastly different shapes.

Table 3-1. *The Main Components of the ER Model Compared to Neo4j*

Entity—Relationship	Neo4j
Entity	Node
Relationship	Relationship
Attribute	Property

In either event, the model still serves as the high-level, conceptual representation for all of the data points. The companion model will also allow us to see the transformation that would occur when moving from the relational setting to the graph setting. Again, we will explore a bit more of the Cypher language in this chapter but only as it applies to our modeling aim. In addition, your preferred programming language is important to how you might consider some aspects of your model, but it is not critical to understanding the essential concepts for modeling with Neo4j.

Modeling Relationships

As you will likely find in working more frequently with graphs, the node types can seem more natural than tables, especially when creating and managing relationships. However, there are some common pitfalls or issues that can surface during the first exercises in modeling.

Directed relationships are an important aspect of graph databases and understanding how they should be modeled is necessary to improving the design, efficiency and manageability of your Neo4j database. The example in Figure 3-4 clearly denotes the direction to infer that "Greg works at GraphStory." In turn, this relationship implies that "GraphStory is an employer of Greg."

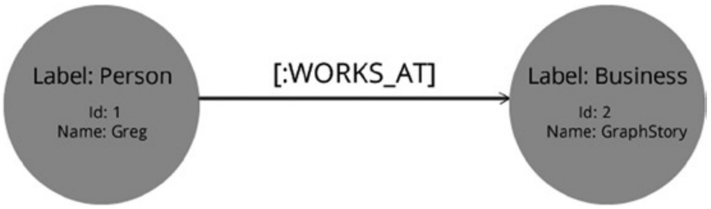


Figure 3-4. *Directed relationship type*

It is not necessary to explicitly add both relationship types, as shown in Figure 3-5, because one directed connection, by definition, suffices for the other direction. In fact, the speed of traversing the graph is not dependent on the direction.

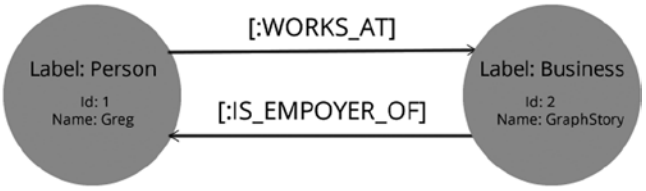


Figure 3-5. *Two relationship connections are unneccesary as the first implies the other*

While some connections between nodes naturally suggest how the direction should be set, others have a mutual or bidirectional relationship. Consider Figure 3-6, in which “GraphStory is a partner with NeoTechnology.” In these bidirectional relationships, a second relationship connection, as with directed relationship, is unnecessary. Again, as is the case with directed relationships, it is faster to have a single relationship with an arbitrary direction.

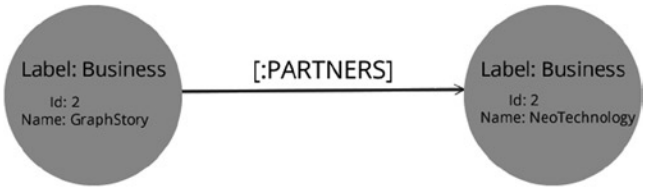


Figure 3-6. *Bidirectional relationship with an arbitrary direction*

Modeling Constraints

Ensuring that specific properties within the model remain unique is an important feature of any database and Neo4j is no different. With Neo4j 2.0, the concept of adding unique constraints based on labels was added. You can use unique constraints, as shown in Listing 3-1, to ensure that property values are unique for all nodes with a specific label. If you are creating the constraint after nodes have been created, then be aware that the new constraint could take some time to become enforced as any existing data must be scanned beforehand.

Listing 3-1. Creating a Unique Constraint

```
CREATE CONSTRAINT ON (business:Business) ASSERT business.businessname IS UNIQUE
```

When adding a unique constraint on a node's property, please note that this process will also create an index on the specific property and, therefore, you will not be able to add a separate index for the property. The index can be used to perform lookups for specific nodes. If you need for some reason to remove the constraint, as shown in Listing 3-2, and require an index on that property, then you will need to create a new index to support lookups.

Listing 3-2. Dropping a Unique Constraint

```
DROP CONSTRAINT ON (business:Business) ASSERT business.businessname IS UNIQUE
```

Modeling Use Cases

To begin building out the model for the application to be developed in the later chapters of the book, the following sections examine in turn the five areas identified as the most significant portions of consumer and business data—namely, social, interest, consumption, location, and intent graphs.

Social Graph

The social graph is the most widely discussed type of graph in the list of graph use cases. In its more well-known incarnation, the social graph represents the degree of connection between users on a particular application, such as Twitter. Facebook’s social graph is the largest social graph in the world, developed on a mostly proprietary technology stack.

In Neo4j, the social graph is typically defined in one of two manners. The first is a direct connection that implies a mutual connection, which is similar to the approach user connections are made on Facebook. The second approach is where one user follows another user, similar to the connections created on Twitter. In Figure 3-7 and 3-8, we can see how both of these connections methods might be modeled within a relational database.

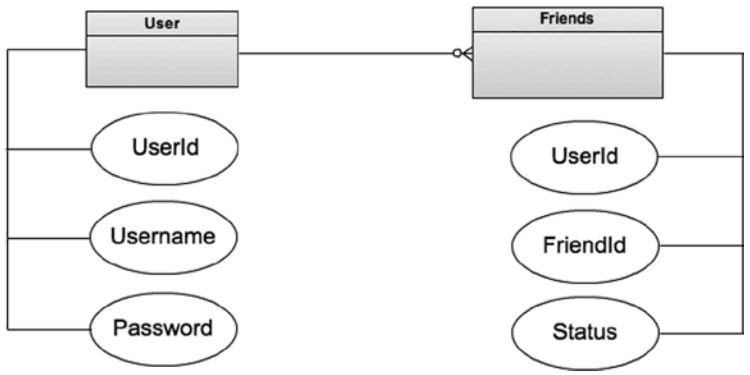


Figure 3-7. *Entity-relationship diagram with mutual connections*

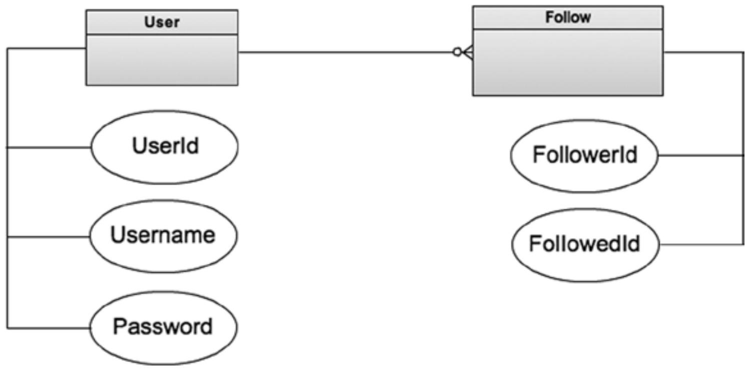


Figure 3-8. *Entity-relationship diagram with a one-way connection*

Figures 3-9 and 3-10 show how the same relationships would be modeled for Neo4j. In Figure 3-9, the direction is shown as a single relationship between two nodes. As mentioned earlier in this chapter, you should avoid duplicating a typed relationship between two nodes. However, this is one exception to the directionality of relationship modeling, as it is necessary to define whether the relationship is mutual and, indirectly, allows for certain features to be enabled.

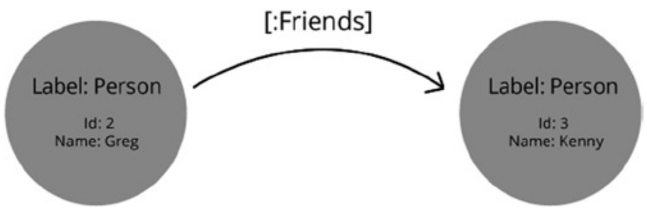


Figure 3-9. *Graph diagram with mutual connection. The direction implies who made the request*

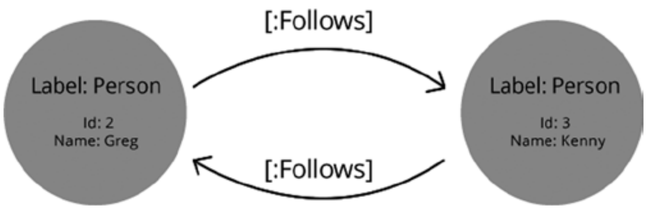


Figure 3-10. *Graph diagram with specific directed connections*

While deciding the manner in which your social model should be established, it is important to consider that there is more than just a technology decision at stake, but, potentially, a business decision as well. While both models allow for exploring connections in either direction from a technical standpoint, the bidirectional relationship implies that only one user action needs to occur in order to establish a mutual connection.

In addition, using the bidirectional or mutual option, by definition, will reduce the number of relationships comparatively by 50 percent. The problem of dense nodes—think of any celebrity who might have millions of followers but only follows a few other users—is less a factor in performance in the latest version of Neo4j. However, directional relationships can sometimes have an impact and need to be considered carefully. For the purposes of the book's example application, we will consider the directional relationship for the social aspect, such as the connection method found in applications such as Twitter.

Interest Graph

The interest graph is closely connected to the intent graph. However, the interest graph is principally concerned with the connecting a person with her specific interests. In that sense, the interest graph would allow for an application to make recommendations regarding related items of interest much in the same way a thesaurus can offer synonyms of a specific word. When combining the interest graph with a person's demographic or social graph, an application can make recommendations that typically have a higher degree of connectedness and relevance. Figure 3-11 demonstrates how an interest graph could be created within a relational model.

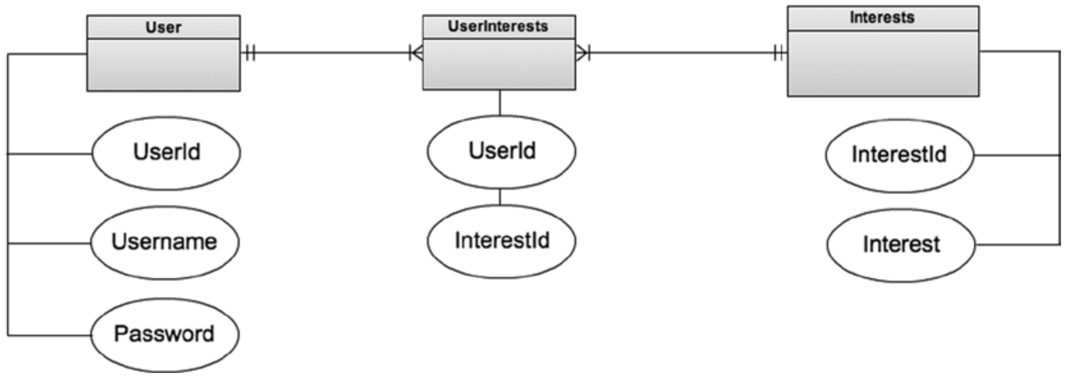


Figure 3-11. *Entity relationship diagram with a user's interests*

Figure 3-12 shows the interest graph as it could be modeled for Neo4j. The interesting aspect in this graph type is how the named relationship in this model, "UserInterests", could be quickly modified to show a degree of interest and the date and time when the interest was established.

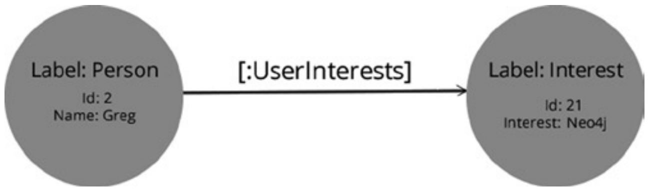


Figure 3-12. *Graph diagram with a user's interests*

As you can see in Figure 3-13, adding a simple measurement for frequency is fairly trivial. Although adding the same measurement in the relational model is possible, the change would probably not happen as easily. More importantly, connecting people with those who have similar interests will be even easier and much faster as the degrees of connection begin to increase.

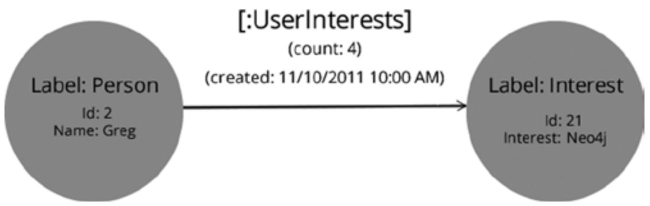


Figure 3-13. *Graph diagram with a user's interests, including properties for the named relationship*