Neo4j High Performance

Design, build, and administer scalable graph database systems for your applications using Neo4j

Sonal Raj
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 "Getting Started with Neo4j"
- A synopsis of the book’s content
- More information on Neo4j High Performance

About the Author

**Sonal Raj** is a hacker, Pythonista, big data believer, and a technology dreamer. He has a passion for design and is an artist at heart. He blogs about technology, design, and gadgets at [http://www.sonalraj.com/](http://www.sonalraj.com/). When not working on projects, he can be found traveling, stargazing, or reading.

He has pursued engineering in computer science and loves to work on community projects. He has been a research fellow at SERC, IISc, Bangalore, and taken up projects on graph computations using Neo4j and Storm. Sonal has been a speaker at PyCon India and local meetups on Neo4j and has also published articles and research papers in leading magazines and international journals. He has contributed to several open source projects.

Presently, Sonal works at Goldman Sachs. Prior to this, he worked at Sigmoid Analytics, a start-up where he was actively involved in the development of machine learning frameworks, NoSQL databases, including MongoDB and streaming using technologies such as Apache Spark.

I would like to thank my family for encouraging me, supporting my decisions, and always being there for me. I heartily want to thank all my friends who have always respected my passion for being part of open source projects and communities while reminding me that there is more to life than lines of code. Beyond this, I would like to thank the folks at Neo Technologies for the amazing product that can store the world in a graph. Special thanks to my colleagues for helping me validate my writings and finally the reviewers and editors at Packt Publishing without whose efforts this work would not have been possible. Merci à vous.
Neo4j High Performance

Welcome to the connected world. In the information age, everything around us is based on entities, relations, and above all, connectivity. Data is becoming exponentially more complex, which is affecting the performance of existing data stores. The most natural form in which data is visualized is in the form of graphs. In recent years, there has been an explosion of technologies to manage, process, and analyze graphs. While companies such as Facebook and LinkedIn have been the most well-known users of graph technologies for social web properties, a quiet revolution has been spreading across other industries. More than 30 of the Forbes Global 2000 companies, and many times as many start-ups have quietly been working to apply graphs to a wide array of business-critical use cases.

Neo4j, a graph database by Neo Technologies, is the leading player in the market for handling related data. It is not only efficient and easier to use, but it also includes all security and reliability features of tabulated databases.

We are entering an era of connected data where companies that can master the connections between their data—the lines and patterns linking the dots and not just the dots—will outperform the organizations that fail to recognize connectedness. It will be a long time before relational databases ebb into oblivion. However, their role is no longer universal. Graph databases are here to stay, and for now, Neo4j is setting the standard for the rest of the market.

This book presents an insight into how Neo4j can be applied to practical industry scenarios and also includes tweaks and optimizations for developers and administrators to make their systems more efficient and high performing.

What This Book Covers

Chapter 1, Getting Started with Neo4j, introduces Neo4j, its functionality, and norms in general, briefly outlining the fundamentals. The chapter also gives an overview of graphs, NOSQL databases and their features and Neo4j in particular, ACID compliance, basic CRUD operations, and setup. So, if you are new to Neo4j and need a boost, this is your chapter.

Chapter 2, Querying and Indexing in Neo4j, deals with querying Neo4j using Cypher, optimizations to data model and queries for better Cypher performance. The basics of Gremlin are also touched upon. Indexing in Neo4j and its types are introduced along with how to migrate from existing SQL stores and data import/export techniques.
Chapter 3, Efficient Data Modeling with Graphs, explores the data modeling concepts and techniques associated with graph data in Neo4j, in particular, property graph model, design constraints for Neo4j, the designing of schemas, and modeling across multiple domains.

Chapter 4, Neo4j for High-volume Applications, teaches you how to develop applications with Neo4j to handle high volumes of data. We will define how to develop an efficient architecture and transactions in a scalable way. We will also take a look at built-in graph algorithms for better traversals and also introduce Spring Data Neo4j.

Chapter 5, Testing and Scaling Neo4j Applications, teaches how to test Neo4j applications using the built-in tools and the GraphAware framework for unit and performance tests. We will also discuss how a Neo4j application can scale.

Chapter 6, Neo4j Internals, takes a look under the hood of Neo4j, skimming the concepts from the core classes in the source into the internal storage structure, caching, transactions, and related operations. Finally, the chapter deals with HA functions and master election.

Chapter 7, Administering Neo4j, throws light upon some useful tools and adapters that have been built to interface Neo4j with the most popular languages and frameworks. The chapter also deals with tips and configurations for administrators to optimize the performance of the Neo4j system. The essential security aspects are also dealt with in this chapter.

Chapter 8, Use Case – Similarity-based Recommendation System, is an example-oriented chapter. It provides a demonstration on how to go about building a similarity-based recommendation system with Neo4j and highlights the utility of graph visualization.
Graphs and graph operations have grown into prime areas of research in computer science. One reason for this is that graphs can be useful in representing several, otherwise abstract, problems in existence today. Representing the solution space of the problem in terms of graphs can trigger innovative approaches to solving such problems. It's simple. Everything around us, that is, everything we come across in our day-to-day life can be represented as graphs, and when your whiteboard sketches can be directly transformed into data structures, the possibilities are limitless. Before we dive into the technicalities and utilities of graph databases with the topics covered in this chapter, let's understand what graphs are and how representing data in the form of graph databases makes our lives easier. The following topics are dealt with in this chapter:

- Graphs and their use
- NoSQL databases and their types
- Neo4j properties, setup, and configurations
- Deployment on the Amazon and Azure cloud platforms
Graphs and their utilities

Graphs are a way of representing entities and the connections between them. Mathematically, graphs can be defined as collections of nodes and edges that denote entities and relationships. The nodes are data entities whose mutual relationships are denoted with the help of edges. Undirected graphs have two-way connections between edges whereas a directed graph has only a one-way edge between the nodes. We can also record the value of an edge and that is referred to as the weight of the graph.

Modern datasets of science, government, or business are diverse and interrelated, and for years we have been developing data stores that have tabular schema. So, when it comes to highly connected data, tabular data stores offer retarded and highly complex operability. So, we started creating data stores that store data in the raw form in which we visualize them. This not only makes it easier to transform our ideas into schemas but the whiteboard friendliness of such data stores also makes it easy to learn, deploy, and maintain such data stores. Over the years, several databases were developed that stored their data structurally in the form of graphs. We will look into them in the next section.

Introducing NoSQL databases

Data has been growing in volume, changing more rapidly, and has become more structurally varied than what can be handled by typical relational databases. Query execution times increase drastically as the size of tables and number of joins grow. This is because the underlying data models build sets of probable answers to a query before filtering to arrive at a solution. NoSQL (often interpreted as Not only SQL) provides several alternatives to the relational model.
NoSQL represents the new class of data management technologies designed to meet the increasing volume, velocity, and variety of data that organizations are storing, processing, and analyzing. NoSQL comprises diverse different database technologies, and it has evolved as a response to an exponential increase in the volume of data stored about products, objects, and consumers, the access frequency of this data, along with increased processing and performance requirements. Relational databases, on the contrary, find it difficult to cope with the rapidly growing scale and agility challenges that are faced by modern applications, and they struggle to take advantage of the cheap, readily available storage and processing technologies in the market.

Often referred to as NoSQL, nonrelational databases feature elasticity and scalability. In addition, they can store big data and work with cloud computing systems. All of these factors make them extremely popular. NoSQL databases address the opportunities that the relational model does not, including the following:

- Large volumes of structure-independent data (including unstructured, semi-structured, and structured data)
- Agile development sprints, rapid iterations, and frequent repository pushes for the code
- Flexible, easy-to-use object-oriented programming
- Efficient architecture that is capable of scaling out, as compared to expensive and monolithic architectures due to the requirement of specialized hardware

**Dynamic schemas**

In the case of relational databases, you need to define the schema before you can add your data. In other words, you need to strictly follow a format for all data you are likely to store in the future. For example, you might store data about consumers such as phone numbers, first and last names, address including the city and state—a SQL database must be told what you are storing in advance, thereby giving you no flexibility.

Agile development approaches do not fit well with static schemas, since every completion of a new feature requires the schema of your database to change. So, after a few development iterations, if you decide to store consumers' preferred items along with their contact addresses and phone numbers, that column will need to be added to the already existing-database, and then migrate the complete database to an entirely new schema.
In the case of a large database, this is a time-consuming process that involves significant downtime, which might adversely affect the business as a whole. If the application data frequently changes due to rapid iterations, the downtime might be occurring quite often. Businesses sometimes wrongly choose relational databases in situations where the effective addressing of completely unstructured data is needed or the structure of data is unknown in advance. It is also worthy to note that while most NoSQL databases support schema or structure changes throughout their lifetime, some including graph databases adversely affect performance if schema changes are made after considerably large data has been added to the graph.

**Automatic sharding**

Because of their structure, relational databases are usually vertically scalable, that is, increasing the capacity of a single server to host more data in the database so that it is reliable and continuously available. There are limits to such scaling, both in terms of size and expense. An alternate approach is to scale horizontally by increasing the number of machines rather than the capacity of a single machine.

In most relational databases, **sharding** across multiple server instances is generally accomplished with **Storage Area Networks (SANs)** and other complicated arrangements that make multiple hardware act as a single machine. Developers have to manually deploy multiple relational databases across a cluster of machines. The application code distributes the data, queries, and aggregates the results of the queries from all instances of the database. Handling the failure of resources, data replication, and balancing require customized code in the case of manual sharding.

NoSQL databases usually support autosharding out of the box, which means that they natively allow the distribution of data stores across a number of servers, abstracting it from the application, which is unaware of the server pool composition. Data and query load are balanced automatically, and in the case of a node or server failure, it can quickly replace the failed node with no performance drop.

Cloud computing platforms such as Amazon Web Services provide virtually unlimited on-demand capacity. Hence, commodity servers can now provide the same storage and processing powers for a fraction of the price as a single high-end server.
Built-in caching

There are many products available that provide a cache tier to SQL database management systems. They can improve the performance of read operations substantially, but not that of write operations and moreover add complexity to the deployment of the system. If read operations dominate the application, then distributed caching can be considered, but if write operations dominate the application or an even mix of read and write operations, then a scenario with distributed caching might not be the best choice for a good end user experience.

Most NoSQL database systems come with built-in caching capabilities that use the system memory to house the most frequently used data and doing away with maintaining a separate caching layer.

Replication

NoSQL databases support automatic replication, which means that you get high availability and failure recovery without the use of specialized applications to manage such operations. From the developer’s perspective, the storage environment is essentially virtualized to provide a fault-tolerant experience.

Types of NoSQL databases

At one time, the answer to all your database needs was a relational database. With the rapidly spreading NoSQL database craze, it is vital to realize that different use cases and functionality call for a different database type. Based on the purpose of use, NoSQL databases have been classified in the following areas:

Key-value stores

Key-value database management systems are the most basic and fundamental implementation of NoSQL types. Such databases operate similar to a dictionary by mapping keys to values and do not reflect structure or relation. Key-value databases are usually used for the rapid storage of information after performing some operation, for example, a resource (memory)-intensive computation. These data stores offer extremely high performance and are efficient and easily scalable. Some examples of key-value data stores are Redis (in-memory data store with optional persistence), MemcacheDB (distributed, in-memory key-value store), and Riak (highly distributed, replicated key-value store). Sounds interesting, huh? But how do you decide when to use such data stores?
Let’s take a look at some key-value data store use cases:

- **Cache Data**: This is a type of rapid data storage for immediate or future use
- **Information Queuing**: Some key-value stores such as Redis support queues, sets, and lists for queries and transactions
- **Keeping live information**: Applications that require state management can use key-value stores for better performance
- **Distributing information or tasks**

## Column family stores

Column family NoSQL database systems extend the features of key-value stores to provide enhanced functionality. Although they are known to have a complex nature, column family stores operate by the simple creation of collections of key-value pairs (single or many) that match a record. Contrary to relational databases, column family NoSQL stores are schema-less. Each record has one or more columns that contain the information with variation in each column of each record.

Column-based NoSQL databases are basically 2D arrays where each key contains a single key-value pair or multiple key-value pairs associated with it, thereby providing support for large and unstructured datasets to be stored for future use. Such databases are generally used when the simple method of storing key-value pairs is not sufficient and storing large quantities of records with a lot of information is mandatory. Database systems that implement a column-based, schema-less model are extremely scalable.

These data stores are powerful and can be reliably used to store essential data of large sizes. Although they are not flexible in what constitutes the data (such as related objects cannot be stored!), they are extremely functional and performance oriented. Some column-based data stores are **HBase** (an Apache Hadoop data store based on ideas from BigTable) and **Cassandra** (a data store based on DynamoDB and BigTable).

So, when do we want to use such data stores? Let’s take a look at some use cases to understand the utility of column-based data stores:

- **Scaling**: Column family stores are highly scalable and can handle tons of information without affecting performance
- **Storing non-volatile, unstructured information**: If collections of attributes or values need to persist for extended time periods, column-based data stores are quite handy
Document databases

Document-based NoSQL databases are the latest craze that have managed to gain wide and serious acceptance in large enterprises recently. These DBMS operate in a similar manner to column-based data stores, incorporating the fact that they allow much deeper nesting of data to realize more complex data structures (for example, a hierarchal data format with a document, within another document, within a document). Unlike columnar databases that allow one or two levels of nesting, document databases have no restriction on the key-value nesting in documents. Any document with a complex and arbitrary structure can be stored using such data stores.

Although they have a powerful nature of storage, where you can use the individual keys for the purpose of querying records, document-based database systems have their own issues and drawbacks, for example, getting the whole record to retrieve a value of the record and similarly for updates that affect the performance in the long run.

Document-based databases are a viable choice for storing a lot of unrelated complex information with variable structure. Some document-based databases are **Couchbase** (a memcached compatible and JSON-based document database), **CouchDB**, and **MongoDB** (a popular, efficient, and highly functional database that is gaining popularity in big data scenarios).

Let's look at popular use cases associated with document databases to decide when to pick them as your tools:

- **Nested information handling**: These data stores are capable of handling data structures that are complex in nature and deeply nested
- **JavaScript compatible**: They interface easily with applications that use JavaScript-friendly JSON in data handling

Graph databases

A graph database exposes a graph model that has **create, read, update and delete** (CRUD) operation support. Graph databases are online (real time) in nature and are built generally for the purpose of being used in transactional (OLTP) systems. A graph database model represents data in a completely different fashion, unlike the other NoSQL models. They are represented in the form of tree-like structures or graphs that have nodes and edges that are connected to each other by relationships. This model makes certain operations easier to perform since they link related pieces of information.
Such databases are popular in applications that establish a connection between entities. For example, when using online social or professional networks, your connection to your friends and their friends' friends' relation to you are simpler to deal with when using graph databases. Some popular graph databases are Neo4j (a schema-less, extremely powerful graph database built in Java) and OrientDB (a speed-oriented hybrid NoSQL database of graph and document types written in Java; it is equipped with a variety of operational modes). Let's look at the use cases of graph databases:

- **Modeling and classification handling**: Graph databases are a perfect fit for situations involving related data. Data modeling and information classification based on related objects are efficient using this type of data store.

- **Complex relational information handling**: Graph databases ease the use of connection between entities and support extremely complex related objects to be used in computation without much hassle.
The following criteria can help decide when the use of NoSQL databases is required depending on the situation in hand:

- **Data size matters**: When large datasets are something you are working on and have to deal with scaling issues, then databases of the NoSQL family should be an ideal choice.

- **Factor of speed**: Unlike relational databases, NoSQL data stores are considerably faster in terms of write operations. Reads, on the other hand, depend on the NoSQL database type being used and the type of data being stored and queried upon.

- **Schema-free design approach**: Relational databases require you to define a structure at the time of creation. NoSQL solutions are highly flexible and permit you to define schemas on the fly with little or no adverse effects on performance.

- **Scaling with automated and simple replications**: NoSQL databases are blending perfectly with distributed scenarios over time due to their built-in support. NoSQL solutions are easily scalable and work in clusters.

- **Variety of choices available**: Depending on your type of data and intensity of use, you can choose from a wide range of available database solutions to viably use your database management systems.

### Graph compute engines

A graph compute engine is a technology that enables global graph computational algorithms to be run against large datasets. The design of graph compute engines basically supports things such as identifying clusters in data, or applying computations on related data to answer questions such as how many relationships, on average, does everyone on Facebook have? Or who has second-degree connections with you on LinkedIn?
Because of their emphasis on global queries, graph compute engines are generally optimized to scan and process large amounts of information in batches, and in this respect, they are similar to other batch analysis technologies, such as data mining and OLAP, that are familiar in the relational world. Whereas some graph compute engines include a graph storage layer, others (and arguably most of them) concern themselves strictly with processing data that is fed in from an external source and returning the results.

The Neo4j graph database

Neo4j is one of the most popular graph databases today. It was developed by Neo Technology, Inc. operating from the San Francisco Bay Area in the U.S. It is written in Java and is available as open source software. Neo4j is an embedded, disk-based, fully transactional Java persistence engine that stores data structured in graphs rather than in tables. Most graph databases available have a storage format of two types:

- Most graph databases store data in the relational way internally, but they abstract it with an interface that presents operations, queries, and interaction with the data in a simpler and more graphical manner.
- Some graph databases such as Neo4j are native graph database systems. It means that they store the data in the form of nodes and relationships inherently. They are faster and optimized for more complex data.

In the following sections, we will see an overview of the Neo4j fundamentals, basic CRUD operations, along with the installation and configuration of Neo4j in different environments.
ACID compliance

Contrary to popular belief, ACID does not contradict or negate the concept of NoSQL. NoSQL fundamentally provides a direct alternative to the explicit schema in classical RDBMSes. It allows the developer to treat things asymmetrically, whereas traditional engines have enforced rigid sameness across the data model. The reason this is so interesting is because it provides a different way to deal with change, and for larger datasets, it provides interesting opportunities to deal with volumes and performance. In other words, the transition is about shifting the handling of complexity from the database administrators to the database itself.

Transaction management has been the talking point of NoSQL technologies since they started to gain popularity. The trade-off of transactional attributes for performance and scalability has been the common theme in nonrelational technologies that targeted big data. Some databases (for example, BigTable, Cassandra, and CouchDB) opted to trade-off consistency. This allowed clients to read stale data and in some cases, in a distributed system (eventual consistency), or in key-value stores that concentrated on read performance, where durability of the data was not of too much interest (for example, Memcached), or atomicity on a single-operation level, without the possibility to wrap multiple database operations within a single transaction, which is typical for document-oriented databases. Although devised a long time ago for relational databases, transaction attributes are still important in the most practical use cases. Neo4j has taken a different approach here. Neo4j’s goal is to be a graph database, with the emphasis on database. This means that you’ll get full ACID support from the Neo4j database:

- **Atomicity (A):** This can wrap multiple database operations within a single transaction and make sure that they are all executed atomically; if one of the operations fails, a rollback is performed on the entire transaction.
- **Consistency (C):** With this, when you write data to the Neo4j database, you can be sure that every client accessing the database afterwards will read the latest updated data.
- **Isolation (I):** This will make sure that operations within a single transaction will be isolated one from another so that writes in one transaction won't affect reads in another transaction.
- **Durability (D):** With this, you're certain that the data you write to Neo4j will be written to disk and available after a database restart or a server crash. If the system blows up (hardware or software), the database will pick itself back up.

The ACID transactional support provides seamless transition to Neo4j for anyone used to relational databases and offers safety and convenience in working with graph data.
Transactional support is one of the strong points of Neo4j, which differentiates it from the majority of NoSQL solutions and makes it a good option not only for NoSQL enthusiasts but also in enterprise environments. It is also one of the reasons for its popularity in big data scenarios.

**Characteristics of Neo4j**

Graph databases are built with the objective of optimizing transactional performance and are engineered to persist transactional integrity and operational availability. Two properties are useful to understand when investigating graph database technologies:

- **The storage within**: Some graph databases store data natively as graphs, which is optimized by design for storage, queries, and traversals. However, this is not practiced by all graph data stores. Some databases use serialization of the graph data into an equivalent general-purpose database including object-oriented and relational databases.

- **The processing engine**: Some graph databases definitions require that they possess the capability for index-free adjacency, which means that nodes that are connected must physically point to each other in the database. Here, let's take a broader view that any database which, from the user's perspective, behaves like a graph database (that is, exposes a graph data model through CRUD operations) qualifies as a graph database. However, there are significant performance advantages of leveraging index-free adjacency in graph data.

Graph databases, in particular native ones such as Neo4j, don't depend heavily on indexes because the graph itself provides a natural adjacency index. In a native graph database, the relationships attached to a node naturally provide a direct connection to other related nodes of interest. Graph queries largely involve using this locality to traverse through the graph, literally chasing pointers. These operations can be carried out with extreme efficiency, traversing millions of nodes per second, in contrast to joining data through a global index, which is many orders of magnitude slower. There are several different graph data models, including property graphs, hypergraphs, and triples. Let's take a brief look at them:

- **Property graphs**: A property graph has the following characteristics:
  - Being a graph, it has nodes and relationships
  - The nodes can possess properties (in the form of key-value pairs)
  - The relationships have a name and direction and must have a start and end node
  - The relationships are also allowed to contain properties
• **Hypergraphs**: A hypergraph is a generalized graph model in which a relationship (called hyperedge) can connect any number of nodes. Whereas the property graph model permits a relationship to have only one start node and one end node, the hypergraph model allows any number of nodes at either end of a relationship. Hypergraphs can be useful where the domain consists mainly of many-to-many relationships.

• **Triples**: Triple stores come from the Semantic Web movement, where researchers are interested in large-scale knowledge inference by adding semantic markup to the links that connect web resources. To date, very little of the web has been marked up in a useful fashion, so running queries across the semantic layer is uncommon. Instead, most efforts in the Semantic Web movement appear to be invested in harvesting useful data and relationship information from the web (or other more mundane data sources, such as applications) and depositing it in triple stores for querying.

Some essential characteristics of the Neo4j graph databases are as follows:

• They work well with web-based application scenarios including metadata annotations, wikis, social network analysis, data tagging, and other hierarchical datasets.

• It provides a graph-oriented model along with a visualization framework for the representation of data and query results.

• A decent documentation with an active and responsive e-mail list is a blessing for developers. It has a few releases and great utility indicating that it might last a while.

• Compatible bindings are written for most languages including Python, Java, Closure, and Ruby. Bindings for .NET are yet to be written. The REST interface is the recommended approach for access to the database.

• It natively includes a disk-based storage manager that has been completely optimized to store graphs to provide enhanced performance and scalability. It is also ready for SSDs.

• It is highly scalable. A single instance of Neo4j can handle graphs containing billions of nodes and relationships.

• It comes with a powerful traversal framework that is capable of handling speedy traversals in a graph space.

• It is completely transactional in nature. It is ACID compliant and supports features such as JTA or JTS, 2PC, XA, Transaction Recovery, Deadlock Detection, and so on.
• It is built to durably handle large graphs that don't fit in memory.
• Neo4j can traverse graph depths of more than 1,000 levels in a fraction of a second.

The basic CRUD operations

Neo4j stores data in entities called nodes. Nodes are connected to each other with the help of relationships. Both nodes and relationships can store properties or metadata in the form of key-value pairs. Thus, inherently a graph is stored in the database. In this section, we look at the basic CRUD operations to be used in working with Neo4j:

```
CREATE ( gates  { firstname: 'Bill', lastname: 'Gates'} )
CREATE ( page  { firstname: 'Larry', lastname: 'Page'}), (page) - [r:WORKS_WITH] -> (gates)
RETURN gates, page, r
```

In this example, there are two queries; the first is about the creation of a node that has two properties. The second query performs the same operation as the first one, but also creates a relationship from page to gates.

```
START n=node(*) RETURN "The node count of the graph is \"+count(*)+\" !" as ncount;
```

A variable named ncount is returned with the value; it's basically the same as `SELECT count(*)`.

```
START self=node(1) MATCH self<--friend
RETURN friend
```

Assuming that we are using this simple database as an example, these commands will return the page node keeping in mind the direction of the relationship:

```
START person=node(*)
MATCH person
WHERE person.firstname = 'Bill'
RETURN person
```
This query searches through all nodes and matches the ones with the `firstname` property that is equal to Bill. The `!` symbol makes sure that only nodes that possess the property are to be taken into consideration, to prevent errors.

```
START person=node(*)
MATCH person
WHERE person.firstname!= 'Bill'
SET person.age = '60'
RETURN person
```

The node that has the `firstname` property as Bill is searched and adds another property called `age` that has the value 60.

```
START person = node(*)
MATCH person
WHERE person.firstname= "Larry"
DELETE person
```

In this query, we match all nodes that have `firstname` equal to Larry and perform a delete operation on them.

```
START node = node(*)
MATCH node-[r]-()
DELETE node, r
```

This query is used to fetch all nodes and relationships and performs a delete operation on them.

So, you now know how to perform basic CRUD operations on a Neo4j graph. We will encounter more of these queries in more complex forms in later chapters in the book.

**The Neo4j setup and configurations**

Neo4j is versatile in terms of usability. You can include and package Neo4j libraries in your application. This is referred to as the embedded mode of operation. For a server setup, you install Neo4j on the machine and configure it as an operating system service. The latest releases of Neo4j come with simple installer scripts for different operating systems. Let’s take a look at how to configure Neo4j in the different modes of operation.
Modes of setup – the embedded mode

Neo4j in the embedded mode is used to include a graph database in your application. In this section, we will see how to configure Neo4j embedded into your application in Eclipse IDE. Ensure that you have the proper version of eclipse IDE from https://www.eclipse.org/downloads/ and the Neo4j Enterprise edition TAR archive from the other downloads section at http://www.neo4j.org/download.

Within Eclipse, navigate to File | New | Java Project, give your project a preferred name, and then click on Finish.

Under the Project Properties page, select the option for Java Build Path (1) on the sidebar, proceed to the Libraries tab (2), and then click on the button for Add External JARs (3). You can now locate the external JAR files of the libraries you want to add from here.

 Navigate to the directory you extracted Neo4j under and look under the libs directory. Select all the *.jar files and click on Add. Click on Finish to complete the package addition process.
In the Eclipse navigation sidebar, right-click on the `src` folder of the newly created project and navigate to **New | Package**. In the dialog that appears, add a new package name. In the example, we have added `com.neo4j.chapter1`. Click on the **Finish** button.

Right-click on the package created and create a new Java class by navigating to **New | Java Class** and name it accordingly (use `HelloNeo` to run the following example). Click on **Finish**. Add the following code into your project. This is a sample program to test whether our embedded setup is working fine:

```java
package com.neo4j.chapter1;

import org.neo4j.graphdb.GraphDatabaseService;
import org.neo4j.graphdb.Node;
import org.neo4j.graphdb.Direction;
import org.neo4j.graphdb.Relationship;
import org.neo4j.graphdb.Transaction;
import org.neo4j.graphdb.RelationshipType;
import org.neo4j.graphdb.factory.GraphDatabaseFactory;

public class HelloNeo {
    // change the path according to your system and OS
    private static final String PATH_TO_DB = "path_to_your.neo4j_installation";

    String response;
    GraphDatabaseService graphDBase;
    Node node_one;
    Node node_two;
    Relationship relation;
```
private static enum RelationTypes implements RelationshipType {
    HATES
}

public static void main( final String[] args )
{
    HelloNeo neoObject = new HelloNeo();
    neoObject.createGraphDb();
    neoObject.removeGraph();
    neoObject.shutDownDbServer();
}

void createGraphDb()
{
    graphDBase = new GraphDatabaseFactory().newEmbeddedDatabase(
        PATH_TO_DB);

    Transaction tx = graphDBase.beginTx();
    try{
        node_one = graphDBase.createNode();
        node_one.setProperty( "name", "Bill Gates, Microsoft" );
        node_two = graphDBase.createNode();
        node_two.setProperty( "name", "Larry Page, Google" );

        relation = node_one.createRelationshipTo( node_two,
            RelationTypes.HATES );
        relation.setProperty( "relationship-type", "hates" );

        response = ( node_one.getProperty( "name" ).toString() )
            + " " + ( relation.getProperty( "relationship-type" ).toString() )
            + " " + ( node_two.getProperty( "name" )
            ).toString() );
        System.out.println(response);
        tx.success();
    }
    finally
    {
        tx.finish();
    }
}

void removeGraph()
{
Chapter 1

```java
Transaction tx = graphDBase.beginTx();
try {
    node_one.getSingleRelationship(RelationTypes.HATES,
        Direction.OUTGOING).delete();
    System.out.println("Nodes are being removed . . .");
    node_one.delete();
    node_two.delete();
    tx.success();
} finally {
    tx.finish();
}

void shutDownDbServer()
{
    graphDBase.shutdown();
    System.out.println("graphdb is shutting down.");
}
```

On running the program, you will see the different stages of operation if your configuration is correct. In fact, there is an easier way to set up this configuration if you are familiar with Maven.

Apache Maven is a software project management and comprehension tool. Based on the concept of **Project Object Model** (**POM**), Maven can manage a project's build, reporting, and documentation from a central piece of information. You can learn more about Maven from the official website at [http://maven.apache.org/](http://maven.apache.org/).

Start a new Maven project on Eclipse and edit `pom.xml` to have the following lines for the dependencies:

```xml
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```
Getting Started with Neo4j

<groupId>org.neo4j</groupId>
<artifactId>neo4j</artifactId>
<version>2.0.1</version>
</dependency>
</dependencies>

When you save the pom.xml file, the Neo4j dependencies are installed into the project. You can now run the preceding script and test the configuration.

Modes of setup – the server mode

To develop applications on single machines locally, the embedded database is efficient and serves the purpose. Most of the examples in this book can be tested with the embedded setup. However, for larger applications that deal with rapidly scaling data, the server mode of Neo4j provides the necessary functionality.

Setting up a Neo4j server is relatively easy. You can include Neo4j startup and shutdown as a normal operating system process. For most Linux distributions, the following procedure would suffice:

1. The latest release of Neo4j can be downloaded from http://www.neo4j.org/download. Select the compressed archive (tar.gz) distribution for your operating system.
2. The archive contents can be extracted using tar -cf <filename>.
   The master directory housing Neo4j can be referred to as NEO4J_HOME.
3. Move into the $NEO4J_HOME directory using cd $NEO4J_HOME and run the installer script using the following command:
   sudo ./bin/neo4j-installer install
4. If prompted, you will be required to enter your user password for super-user access privileges to restricted directories:
   sudo service neo4j-service status
   This indicates the state of the server, which in this case is not running.
5. The following command starts the Neo4j server:
   sudo service neo4j-service start
6. If you need to stop the server, you can run this from the terminal:
   sudo service neo4j-service stop
During installation, you will be asked to select the user under which Neo4j will. You can specify a username (the default is neo4j), and if that user does not exist on that system, a system account in that name will be created and the ownership of the $NEO4J_HOME/data directory will be assigned (chown) to that user. It is a good practice to create a dedicated user to run this service, and hence it is suggested that the downloaded archive is extracted under /opt or the package directory for optional packages on your system.

If you want the Neo4j server to no longer be a part of the system startup service, the following commands can be used to remove it:

```bash
cd $NEO4J_HOME
sudo ./bin/neo4j-installer remove
```

If the server is running, it is stopped and removed.

**Neo4j high availability**

In this section, we will learn how to set up Neo4j HA onto a production cluster. Let's assume that our cluster has three machines to be set up with Neo4j HA.

Download Neo4j Enterprise from [http://neo4j.org/download](http://neo4j.org/download), extract the archive into the machines on the production cluster, and perform the following configurations to the local property files of the HA servers:

**Machine #1 – neo4j-01.local**

File: conf.neo4j.properties:

```
# A unique Id for this machine, must be non-negative
ha.server_id = 1

# Specify other hosts that make up this database cluster.
ha.initial_hosts = neo4j-01.local:5001,neo4j-02.local:5001,neo4j-03.local:5001

# You can also specify the hosts using their IP addresses
# ha.initial_hosts = 192.168.0.61:5001, 192.168.0.62:5001, 192.168.0.63:5001
```
Getting Started with Neo4j

File: conf/neo4j-server.properties:

# Mention the IP address to which this database server will listen
# to. 0.0.0.0 means it will listen to all incoming connections.
org.neo4j.server.webserver.address = 0.0.0.0

# Specify the mode of operation as HA if the mode is High
# Availability or set to SINGLE if using a cluster of 1 Node
# (This is default setting)
org.neo4j.server.database.mode=HA

Machine #2 – neo4j-02.local

File: conf/neo4j.properties:

# A unique Id for this machine, must be non-negative
ha.server_id = 2

# Specify other hosts that make up this database cluster.
ha.initial_hosts = neo4j-01.local:5001,neo4j-02.local:5001,neo4j-03.
local:5001

# You can also specify the hosts using their IP addresses
# ha.initial_hosts = 192.168.0.61:5001, 192.168.0.62:5001,
192.168.0.63:5001

File: conf/neo4j-server.properties:

# Mention the IP address to which this database server will listen
# to. 0.0.0.0 means it will listen to all incoming connections.
org.neo4j.server.webserver.address = 0.0.0.0

# Specify the mode of operation as HA if the mode is High
# Availability or set to SINGLE if using a cluster of 1 Node
# (This is default setting)
org.neo4j.server.database.mode=HA

Machine #3 – neo4j-03.local

File: conf/neo4j.properties:

# A unique Id for this machine, must be non-negative
ha.server_id = 3

# Specify other hosts that make up this database cluster.
Chapter 1

```
ha.initial_hosts = neo4j-01.local:5001, neo4j-02.local:5001, neo4j-03.
local:5001

# You can also specify the hosts using their IP addresses
# ha.initial_hosts = 192.168.0.61:5001, 192.168.0.62:5001,
192.168.0.63:5001
```

File: conf/neo4j-server.properties:

```
# Mention the IP address to which this database server will listen
# to. 0.0.0.0 means it will listen to all incoming connections.
org.neo4j.server.webserver.address = 0.0.0.0

# Specify the mode of operation as HA if the mode is High
# Availability or set to SINGLE if using a cluster of 1 Node
# (This is default setting)
org.neo4j.server.database.mode = HA
```

Use the following commands on the neo4j script on each server to start up the
servers. The order in which the servers are started is not important:

```
neo4j-01$ ./bin/neo4j start  (# to start first server)
neo4j-02$ ./bin/neo4j start  (# to start second server)
neo4j-03$ ./bin/neo4j start  (# to start third server)
```

If the database mode has been set to HA, the startup script does not wait for the
server to become available, but returns immediately. The reason being that each
machine does not accept requests till the setup of a cluster has been completed. For
example, in the preceding configuration, this happens when the second machine
starts up. In order to monitor the state of the startup process, you can trace messages
in the console.log file created during the setup. You can find the location of the log
file printed before the startup script terminates.

**Configure Neo4j for Amazon clusters**

The most popular thing among the cloud deployment platforms has been Amazon
Web Services (AWS), particularly on their EC2 cluster-computing systems. These
services are not only easy to set up but offer a wide range of services and support
that make the life of admins a lot easier in the long run.
Neo4j, like a lot of other databases, is quite easy to configure and set up on an AWS server. In this section, we outline the deployment process for a Neo4j instance on Amazon EC2 (short for Elastic Compute Cloud). This process requires you to have a valid AWS account and be familiar with launching instances. If you feel you need to level up your experience with AWS, I would recommend that you follow the official guide of Amazon so that you are able to connect with your instance with SSH; the official guide is available at http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EC2_GetStarted.html.

You will also need a copy of the latest stable version of Neo4j for Unix Systems. The community edition will suffice for developmental purposes. The latest downloads can be found at http://www.neo4j.org/download. You can then perform the following steps:

1. You need to open the AWS management console and select Ubuntu Server 12.04.1 LTS 64-bit or start a basic 32-bit Linux instance. You could start an Ubuntu AMI but it does not include a Java installation, which is a key dependency of Neo4j, so you’ve got to install it manually.

2. In the Instance Details section, select m1.large as the type and make sure that Availability Zone is set to any of the us-east regions. A new security group needs to be created or you can use the default one and configure a new security rule for the port to be used by the Neo4j server.

3. When the instance is launched, a TCP rule is created on the 7474 port used by Neo4j with 0.0.0.0/0 as the source address. What we did was open the 7474 port for all external access (with 0.0.0.0 being the universal identifier). If you intend to use the Neo4j REST API by remote calls from another server, then for security reasons you can change the source field to that of the external server. The 22 port also needs to be open for SSH.

Now, it's time to install Neo4j into the system; let's do this by performing the following steps:

1. Open a terminal on your local system where we downloaded Neo4j. We now copy or transfer the archive to the AWS server using the scp command:

   ```
   scp -i filename.pem neo4j-community-2.1.2-unix.tar.gz ec2-user@PUBLIC_DNS_OF_INSTANCE:/home/ec2-user
   ```

2. You will need to provide the absolute path to your pem key file, which is typically found in ~/.ssh, the filename of the Neo4j server, and the public DNS of your EC2 instance (ec2-user by default). Next, we establish a connection with our EC2 instance using SSH:

   ```
   ssh -i filename.pem ec2-user@PUBLIC_DNS_OF_INSTANCE
   ```
3. Extract the archive contents for the Neo4j server:
   `tar xvfz neo4j-community-2.1.2-unix.tar.gz`

4. You need to move the content into `/usr/local` and change the folder name to `neo4j`:
   `sudo mv neo4j-community-2.1.2-unix.tar.gz /usr/local/neo4j`

5. You now need to enable external access to the Neo4j server by editing the Neo4j configurations file. You need to open `neo4j-server.properties` under the `conf` directory of the master folder and append the following line:
   `org.neo4j.server.webserver.address = 0.0.0.0`
   This creates an open connection for anyone to access the Neo4j server. For restricted access, you can specify the IP of the machine, which will act as the source.

6. Finally, the server is started from the installation directory using the following command:
   `sudo ./bin/neo4j start`

A startup script can be created to automate the server initiation. To check whether the deployment succeeded, you need to pop up a browser on your local machine and key in `http://PUBLIC_DNS_OF_INSTANCE:7474`.

This should direct you to the Monitoring and Management console of Neo4j on your AWS server. Voilà! We're done.

**Cloud deployment with Azure**

In this section, you will learn how to deploy Neo4j to a Linux VM hosted on Azure. Azure has wizards to guide you, but we will be using **Command-line Interface (CLI)** tools for our setup. If CLI tools are not installed on the system, you can install them using the Node package manager with the following command:

`npm install azure-cli`

When the tools are installed, we open a terminal, type `azure`, and we are greeted with cool ASCII art and some common commands. Now, to create our new Linux VM on Azure, we need the following information:

- The DNS name of our VM, which will later be used to access your app as `DNS_name.cloudapp.net`. We will be using `myDNS`.

---
The name of an existing Ubuntu distribution image that can be selected from existing ones (type `azure vm image list` to view all images) or a custom image can be uploaded. Here, we use `z12k$9b3b3w66g78t94r$v5b73d$rd23__Ubuntu-12_04_1-LTS-amd64-server-20140618-en-us-50GB`.

You can now create the Linux VM with the following command in the terminal:

```
azure vm create myDNS z12k$9b3b3w66g78t94r$v5b73d$rd23__Ubuntu-12_04_1-LTS-amd64-server-20140618-en-us-50GB username -e -l "West US"
```

In this command, `username` is the default user account that will be created whose username is specified later. The `-e` flag enables SSH on the default port 22. The `-l` flag permits specifying the region where the VM will be deployed. Now we have the VM created and we can easily access it with SSH.

```
ssh username@myDNS.cloudapp.net
```

Since we are using an Ubuntu instance, we will install Neo4j using the Debian repository by performing the following steps:

1. Add the repository to your system configuration:
   ```
echo 'deb http://debian.neo4j.org/repo stable/' > /etc/apt/sources.list.d/neo4j.list
   
   sudo apt-get update
   
   sudo apt-get install neo4j
   ```

2. If we need to access Neo4j from external applications or servers, we need to configure the Neo4j properties accordingly by performing the following steps:

   1. Open the `/etc/neo4j/neo4j-server.properties` file. Add the following line to the file:
      ```
      org.neo4j.server.webserver.address = 0.0.0.0
      ```

   2. Also, confirm that the SSL port is enabled:
      ```
      org.neo4j.server.webserver.https.enabled = true
      ```

If the server was already started, we need to restart it with the following command:

```
sudo /etc/init.d/neo4j-service restart
```
We will now navigate to the Azure portal and the port that Neo4j runs on (7474 by default) has to be opened if the server is intended to be used as a database server. In this case, we map the 7474 port with the 80 port so that the port need not be specified with the requests. We will be using the `add new endpoint` function of Azure for this, as shown below:

In order to test whether our installed application has successfully deployed, we can test it with the following call:

```bash
curl http://myDNS.cloudapp.net
```
Getting Started with Neo4j

If it works, we have successfully set up Neo4j on Azure. However, the fun does not end there. If your Azure subscription gives you access to apps for the Azure store, then you will find that Neo4j has been included as an app there. So, the first thing you need to do is install Apps for Azure.

Search and select the latest version of Neo4j that is available in the store and then click on **Deploy To Cloud** in the screen that appears. We then need to select the data center and provide our Windows Azure Subscription details in the form of our publishsettings file.

We then select the size of our VM and specify a password for the administrator that will be mailed after the completion of the deployment.
Next, once the deployment completes, you can RDP into the VM using the admin credentials from http://manage.windowsazure.com. Similar to the previous process, if we want our server to be accessible from external hosts, we will need to add the following line to the neo4j-server.properties file:

    org.neo4j.server.webserver.address = 0.0.0.0

**Summary**

In this chapter, you learned about what NoSQL databases are and how important a role graph databases play when datasets are large, complex, and inter-related. You also learned about the different modes of operation of Neo4j, namely, embedded, server, and high availability, and how to configure each of them. Also, Neo4j is easy to set up in cloud deployment environments such as Amazon Clusters and Windows Azure, which offer native built-in support for Neo4j as a scalable database management system.

In the next chapter, we will be dealing with how to efficiently query Neo4j and also study the indexing support that can be used to optimize traversals.
Where to buy this book
You can buy Neo4j High Performance 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.