



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Neo4j Essentials

Leverage the power of Neo4j to design, implement, and deliver top-notch projects

Sumit Gupta

[PACKT] open source*
PUBLISHING community experience distilled

www.it-ebooks.info

Neo4j Essentials

Leverage the power of Neo4j to design, implement,
and deliver top-notch projects

Sumit Gupta



BIRMINGHAM - MUMBAI

Neo4j Essentials

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: February 2015

Production reference: 1190215

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78355-517-8

www.packtpub.com

Credits

Author

Sumit Gupta

Project Coordinator

Neha Bhatnagar

Reviewers

J. Paul Daigle

Endy Jasmi

Panayotis Matsinopoulos

Tiago Pires

Proofreaders

Paul Hindle

Bernadette Watkins

Indexer

Priya Sane

Commissioning Editor

Kunal Parikh

Graphics

Disha Haria

Acquisition Editor

Larissa Pinto

Production Coordinator

Nitesh Thakur

Content Development Editor

Arvind Koul

Cover Work

Nitesh Thakur

Technical Editor

Vivek Arora

Copy Editor

Laxmi Subramanian

About the Author

Sumit Gupta is a seasoned professional, innovator, and technology evangelist with over 100 plus man months of experience in architecting, managing, and delivering enterprise solutions revolving around a variety of business domains such as hospitality, healthcare, risk management, insurance, and so on. He is passionate about technology with over 14 years of hands-on experience in the software industry and has been using big data and cloud technologies over the past 5 years to solve complex business problems.

I want to acknowledge and express my gratitude to everyone who supported me in authoring this book. I am thankful for their inspiring guidance as well as valuable/constructive, and friendly advice.

About the Reviewers

J. Paul Daigle is a backend web developer in Atlanta, GA. His graduate research was in distributed graph algorithms. He is currently employed at BitPay, Inc.

Endy Jasmi is a software engineer at a semi government company in Malaysia. He received his bachelor's degree in microelectronic engineering from the National University of Malaysia. He loves giving back by open sourcing some of his projects on GitHub. He loves technology, be it hardware or software. In his spare time, he is currently working on a project related to the Internet of Things.

I want to thank my parents for their support and belief that I can review this book. I would also like to thank Neha Bhatnagar, the project coordinator for this book, who has been kind to me throughout the completion of this book.

Panayotis Matsinopoulos is a senior software engineer. He studied computer science at the University of Athens, Greece, and has an MSc in telecommunications from University College London. He has been designing and developing applications for the last 25 years. He started with C and C++ and later shifted to Java, C#, and VB.NET. For the last 4 years, he has been a Ruby on Rails fanatic. With lots of experience in various database technologies, both SQL and NoSQL, currently, he is in love with graphs and Neo4j. He works for various clients around the world, either start-ups or well-established companies with millions of customers. Occasionally, he teaches both kids and adults how to write computer programs and likes to play chess with people around the world.

Tiago Pires is a software engineer, programmer, and geek born in Portugal in 1985. He is full of energy, curiosity, and has an endless will to learn. Since he was a little child, he has been in love with computers, studied computer science at high school, and obtained a master's degree in software engineering from the Department of Informatics Engineering of the University of Coimbra , Portugal. An open source software and Docker enthusiast, Tiago is currently working as a full-stack developer with Portugal Telecom Inovação in Aveiro, Portugal, supporting multiple projects and researching recommendation systems and natural language processing.

I would like to thank my parents and family for all the love and faith, and my fiancée, Ana, for always supporting me unreservedly. Love you so much.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Chapter 1: Installation and the First Query	9
Licensing options	10
Community Edition	10
Enterprise Edition	11
A common feature set	11
Personal license	11
The start up program	12
Enterprise subscriptions	12
Applicability	12
System hardware requirements	12
Deployment options	13
Installing Neo4j Community Edition on Windows	14
Installing as a Windows service	15
Installing as a Windows archive / standalone application	18
Installing Neo4j Community Edition on Linux/Unix	18
Installing as a Linux tar / standalone application	19
Installing Neo4j as a Linux service	20
Installing the Neo4j Enterprise Edition	21
Configuring a Neo4j cluster on Windows	21
Configuring a Neo4j cluster on Linux/Unix	23
Tools and utilities for administrators/developers	24
Running your first Cypher query	27
Interactive console – Neo4j shell	27
Working with REST APIs	28
Java code API and embedding the Neo4j database	29
The Neo4j browser	31
Summary	32

Chapter 2: Ready for Take Off	33
Integration of the BI tool – QlikView	34
Creating a ready-to-use database – batch imports	39
The CSV importer	39
LOAD CSV with CREATE	40
LOAD CSV with MERGE	41
The spreadsheet way – Excel	42
HTTP batch imports – REST API	43
Java API – batch insertion	45
BatchInserter	45
Batch indexing	48
Understanding performance tuning and optimizations	50
Tuning JVM	52
LOAD CSV	52
Batch inserter / indexer	53
Summary	53
Chapter 3: Pattern Matching in Neo4j	55
Agile data modeling with Neo4j	56
Patterns and pattern matching	59
Pattern for nodes	60
Pattern for more than one node	60
Pattern for paths	61
Pattern for labels	61
Pattern for relationships	61
Pattern for properties	62
Expressions	62
Usage of pattern	63
Read-only Cypher queries	65
Creating a sample dataset – movie dataset	65
Working with the MATCH clause	68
Working with nodes	68
Working with relationships	69
Working with the OPTIONAL MATCH clause	70
Working with the START clause	71
Working with the WHERE clause	72
Working with the RETURN clause	73
Schema and legacy indexing	74
Using legacy indexes	74
Using schema-level indexing	75
Creating schema with Cypher	75

Movie Demo with GraphGists	76
Summary	78
Chapter 4: Querying and Structuring Data	79
Cypher write queries	80
Working with nodes and relationships	80
Working with MERGE	82
Writing data in legacy indexing	84
Writing data in a schema	88
Managing schema with Java API	88
Managing schema with REST	89
Unicity and other schema constraints	91
Applying unicity constraints with REST	91
Applying unicity constraints with Java	92
Cypher optimizations	94
Summary	96
Chapter 5: Neo4j from Java	97
Embedded versus REST	98
Embedding Neo4j in Java applications	98
Neo4j as a REST-based application	101
Which is best?	104
Unit testing in Neo4j	106
Testing frameworks for Neo4j	111
Java APIs	113
Graph traversals	114
Summary	119
Chapter 6: Spring Data and Neo4j	121
Spring Data philosophy	122
First step – Neo4jTemplate	125
Spring Data repositories and entities	137
Advanced mapping mode – AspectJ	141
Summary	144
Chapter 7: Neo4j Deployment	145
Neo4j architecture and advanced settings	146
High Availability and linear scalability	146
Fault tolerance	147
Data replication and data locality	148
Backup and recovery	149
Advanced settings	150

Neo4j cluster – principles and recommended setup	152
Scaling write throughputs	152
Introducing queues for write operations	152
Batch writes	153
Vertical scaling	153
Tuning Neo4j caches	154
Scaling read throughputs	154
Load balancer	154
Cache-based sharding	155
Monitoring	156
JConsole in local mode	156
JConsole in remote mode	158
Summary	160
Chapter 8: Neo4j Security and Extension	161
Neo4j security	162
Securing access to Neo4j deployment	162
Restricting access to Neo4j server / cluster with the proxy server	163
Feature deactivation	164
Fine-grained authorization	164
API extensions – server plugins and unmanaged extensions	168
Server plugins	169
Unmanaged extensions	174
Summary	177
Index	179

Preface

Understanding the data is the key to all systems and depicting analytical models on that data, which is built on the paradigm of real-world entities and relationships, are the success stories of large-scale enterprise systems.

Architects/developers and data scientists have been struggling hard to derive the real-world models and relationships from discrete and disparate systems, which consist of structured / un-structured data.

We would all agree that without any relationships, data is useless. If we cannot derive relationships between entities, then it will be of little or no use. After all, it's all about the connections in the data.

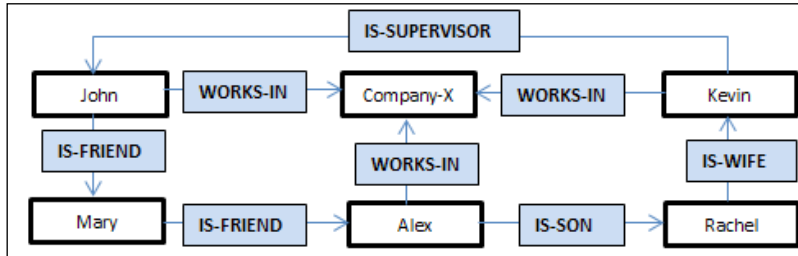
Coming from the relational background, our first choice would be to model this into relational models and use RDBMS.

No doubt we can use RDBMS, but relational models such as RDBMS focus more on entities and less on relationships between entities. There are certain disadvantages in using RDBMS for modeling the above data structure:

- Extensive Joins: While RDBMS can be made to replicate graphical ones, the extensive use of joins would make the technique quite slow.
- It is difficult to query and to handle complex queries and relationships where entities are linked deeply and at multiple levels.

Relationships are dynamic and evolving, which makes it more difficult to create models.

Let's consider the example of social networks:



Social networks are highly complex to define. They are agile and evolving. Considering the preceding example where John is linked to Kevin because of the fact that they work in the same company will be changed as soon as either of them switch companies. Moreover, their relationship IS_SUPERVISOR is dependent upon the company dynamics and is not static. The same is the case with relationship IS_FRIEND, which can also change at any time.

The following are some more use cases:

- Model and store 7 billion people objects and 3 billion nonpeople objects to provide an earth-view drill down from the planet to the sidewalk
- Network management
- Genealogy
- Public transport links and road maps

All the preceding examples state a need for a generic data structure, which can elegantly represent any kind of data and at the same time is it easy to query in a highly accessible manner.

Let's introduce a different form of database that focuses on relationships between the entities rather than the entities itself – Neo4j.

Neo4j as NoSQL database that leverages the theory of graphs. Though there is no single definition of graphs, here is the simplest one, which helps to understand the theory of graphs ([http://en.wikipedia.org/wiki/Graph_\(abstract_data_type\)](http://en.wikipedia.org/wiki/Graph_(abstract_data_type))):

"A graph data structure consists of a finite (and possibly mutable) set of nodes or vertices, together with a set of ordered pairs of these nodes (or, in some cases, a set of unordered pairs). These pairs are known as edges or arcs. As in mathematics, an edge (x,y) is said to point or go from x to y. The nodes may be part of the graph structure, or may be external entities represented by integer indices or references."

Neo4j is an open source graph database implemented in Java.

Its first version (1.0) was released in February 2010, and since then it has never stopped. It is amazing to see the pace at which Neo4j has evolved over the years. At the time of writing this book, the current stable version is 2.1.5, released in September 2014.

Let's move forward and jump into the nitty-gritties of Neo4j. In the subsequent chapters, we will cover the various aspects of Neo4j dealing with installation, data modeling and designing, querying, performance tuning, security, extensions, and many more.

What this book covers

Chapter 1, Installation and the First Query, details the installation process of Neo4j. It briefly explains the function of every tool installed together with Neo4j (shell, server, and browser). More importantly, the chapter will introduce and help developers to get familiar with the Neo4j browser and run the first basic Cypher query using different methods exposed by Neo4j (shell, Java, browser, and REST).

Chapter 2, Ready for Take Off, details the possible options at hand for integrating Neo4j with Business Intelligence (BI) tools and also inserting bulk data in Neo4j from various data sources such as CSV and Excel. It also talks about the usage of Java and REST APIs exposed by Neo4j for performing bulk data import and indexing operations and ends with the various strategies/parameters available and recommended for optimizing Neo4j.

Chapter 3, Pattern Matching in Neo4j, talks briefly about the data modeling techniques for graph databases such as Neo4j and then describes the usefulness of pattern and pattern matching for querying the data from the Neo4j database along with its syntactical details.

This will then dive into the syntactic details of the read-only Cypher queries and its new indexing capabilities.

Chapter 4, Querying and Structuring Data, starts with writing data in Neo4j using patterns, then it talks about structuring the data by applying various constraints, schema, and indexes on the underlying data in Neo4j, and ends with the strategies and recommendations for optimizing the Cypher queries.

Chapter 5, Neo4j from Java, talks about the available strategies for integrating Neo4j and Java and the applicability of these integration strategies in various real-world scenarios. It briefly talks about the integration and usage with various other open source unit testing APIs, and then discusses the Java packages / APIs / graph algorithms exposed by Neo4j along with examples of graph traversals and searching.

Chapter 6, Spring Data and Neo4j, talks about the best of both worlds, that is, Spring and Neo4j. It starts with the philosophy of Spring Data, key concepts, and then explores the possibilities offered by Spring and Spring Data Neo4j with appropriate examples.

Chapter 7, Neo4j Deployment, dives into the principles/strategies for deploying Neo4j in the distributed environment and recommended deployments for varied needs of scaling reads, writes, or both. It also talks about the monitoring parameters and APIs available in Neo4j.

Chapter 8, Neo4j Security and Extension, explains various ways to extend and customize Neo4j and provide new functionalities or secure our Neo4j deployments by implementing custom plugins or extensions.

What you need for this book

Readers should have some basic knowledge and understanding of any graph or NoSQL databases. It would also be good to have some prior knowledge of Java.

Who this book is for

This book is for expert programmers (especially those experienced in a graph-based or NoSQL-based database) who now want to learn Neo4j at a fast pace.

If you are reading this book, then you probably already have sufficient knowledge about the graph databases and you will appreciate their contribution in the complex world of relationships. This book will provide in-depth details in a fast-paced manner for learning and starting development with Neo4j. We will talk about various aspects of Neo4j – data structure, query, integrations, tools, performance parameters, deployments, and so on.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The `addData()` method is annotated with `@Transactional`, which will inform the Spring framework to start a new transaction while executing this method."


A block of code is set as follows:


```
@Transactional
public void addData() {
    // Create Movie
    Movie rocky = new Movie();
    rocky.setTitle("Rocky");
    rocky.setYear(1976);
    rocky.setId(rocky.getTitle() + "-"
        +String.valueOf(rocky.getYear()));
}
```

Any command-line input or output is written as follows:

```
mvn install
mvn exec:java -Dexec.cleanupDaemonThreads=false -
Dexec.mainClass="org.neo4j.spring.samples.MainClass"
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Select the **Local Process** option and then select **org.neo4j.server.Bootstrapper** and click on **Connect**."

 Warnings or important notes appear in a box like this.

 Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1

Installation and the First Query

Neo4j comes with several flavors, thus embracing the diversity of its clients: from research projects to Fortune 500 companies through rapidly changing start-ups. Moreover, Neo4j is a multifaceted graph database, which can adapt to different technical constraints.

You now plan to start a project with it but you need some guidance. This chapter will help you understand the licensing options and the characteristics of each deployment model offered by Neo4j. Moreover, it will guide you through the installation process, starting from the download to the first actual query executed against your fully working instance.

At the end of this chapter, your work environment will be fully functional, and you will be able to write your first Cypher query to insert/fetch the data from the Neo4j database.

This chapter will cover the following topics:

- Licensing options and installing Neo4j
- Understanding all the deployment options
- Various tools and utilities
- Running your first Cypher query!

Licensing options

Neo4j is an open source graph database, which means all its sources are available to the public (on GitHub, at the time of writing). However, Neo Technology, the company behind Neo4j, distributes the latter in two different editions – Community Edition and Enterprise Edition.

Community Edition

Community Edition is a single node installation licensed under **General Public License (GPL) Version 3** – http://en.wikipedia.org/wiki/GNU_General_Public_License and is used for:

- Development and QA environments for fast-paced developments
- Small-to-medium-scale applications where it is preferred to embed the database within the existing application

Here are the main features of Neo4j Community Edition:

- Kernel API to intercept and customize Neo4j at a low level
- Core APIs to handle primitive operations on the graph
- APIs to query on the graph and retrieve data (Traversal API and Cypher Query Language)
- JMX monitoring primitives about the state of a Neo4j instance
- Index APIs to retrieve specific data using a fast and clean API
- Several REST APIs to expose the operations mentioned earlier
- Neo4j shell as a command line REST shell-like client
- Neo4j browser as a graphical Neo4j administration tool on your browser

You can benefit from the support of the whole Neo4j community on Stack Overflow, Google Groups, and Twitter.



If you plan to ask a question on Stack Overflow, do not forget to tag your question with #Neo4j hashtag.

Enterprise Edition

Enterprise Edition comes with three different kinds of subscription options and provides the distributed deployment of the Neo4j databases, along with various other features such as backup, recovery, replication, and so on.

Visit <http://neo4j.com/subscriptions/> for more information on the available subscriptions with Enterprise Edition.

Each of the subscriptions are subject to their own license and pricing but let's first see the common feature set available with all types of subscriptions.

A common feature set

Community Edition features are obviously a part of every enterprise package. Moreover, all enterprise options include:

- Maintenance and scheduled fixes
- Advanced monitoring options
- Hot backups (without interruption of service)
- Deployment in cluster and high-performance cache

Personal license

The personal license plan is free of charge and may look very similar to the Community Edition. It targets students as well as small businesses and is only applicable under the following conditions:

- Your company is made up of up to three employees or contractors
- Your company is self-funded
- Your company's annual revenue amounts to not more than \$100K

Depending on the company's location, the U.S. personal agreement at http://www.neotechnology.com/terms/personal_us/ or the world personal agreement at http://www.neotechnology.com/terms/personal_ne/ is to be accepted.

The start up program

Starting from this plan, you can benefit from enterprise support. The start up license allows workday support hours: 10 hours/5 business days.

The applicability conditions are similar to personal license, only the limits are raised:

- Your company is funded up to \$10 million
- Your company's annual revenue amounts to up to \$5 million

Depending on the company's location, the U.S. start up agreement at http://www.neotechnology.com/terms/startup_us/ or the World startup agreement at http://www.neotechnology.com/terms/startup_ne/ is to be accepted and the license fees amount to \$12K US dollar / €12K per instance.



The license defines an instance as the Java Virtual Machine hosting Neo4j server.



Enterprise subscriptions

With this plan, you can benefit from 24/7 support and emergency custom patches if needed. At this scale, your company will have to directly contact Neo Technology to assess the cost of your required setup.

Applicability

Neo4j is a database developed in Java and Scala – two programming languages that can only be executed within a Java Virtual Machine. As such, Neo4j can only be executed within a JVM, regardless of how you deploy it.

Therefore, embedding Neo4j is only available to JVM-based applications.

System hardware requirements

Neo4j does not restrict users to use certain hardware specifications but it does recommend minimum specifications for CPU, RAM, and disk that are required for deploying in development or production environments.

These specifications are explained as follows:

- **CPU:** Traversing in graphs and graph computations is CPU intensive and requires a good amount of CPU cycles. The following are the requirements for CPU:
 - Must have an Intel Core I3 processor
 - Good to have an Intel Core I7 processor
- **RAM:** The graphs in Neo4j are mapped to the OS memory so that the indexes and cached data can be traversed/searched from the memory itself and do not require any expensive I/O operation on disks. The following are the memory requirements:
 - Must have at least 2 GB
 - Good to have around 16 GB
- **Disk:** Anything that cannot be kept in memory will be flushed out to disks and will be required to read and load back into RAM on subsequent user's request. The higher the I/O, the better the response time will be for reading data from the disk. The following are some high-level specifications for the hard drive:
 - Must have SATA drives with 15k RPM
 - Good to have SSDs
- **Operating system:** The following are the various flavors of OSes recommended for deploying Neo4j:
 - For production: Linux, HP-UX, and Windows 2008
 - For development: Windows XP, Mac OS X, and Linux

Deployment options

Depending upon the license and operating system, Neo4j provides various deployment options. Although the nature of your project will probably rule out some licensing choices, the deployment plan remains open. Unfortunately, there cannot be universal recommendations about which one is going to be the best fit for you; it completely depends on the context of your project.

The following are the deployment options available with Neo4j:

- **Neo4j Community Edition on Windows/Linux:** Single node installation on Windows or Linux operating systems
- **Neo4j Enterprise Edition on Windows/Linux:** Single node or setting up cluster of nodes on Windows or Linux operating systems

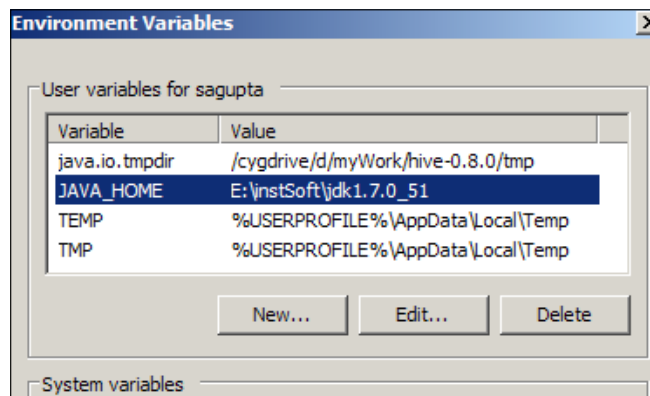
Let's move forward now and discuss in detail each of the earlier mentioned deployment options.

Installing Neo4j Community Edition on Windows

This section will guide you through the end-to-end process of Neo4j installation on the Windows operating system. At the end of this section, you will have a fully functional Neo4j instance running on your Windows desktop / server.

Perform the following steps in order to install Neo4j on Windows:

1. Download and install Oracle Java 7 available at <http://www.oracle.com/technetwork/java/javase/documentation/install-windows-152927.html> or Open JDK 7 available at <https://jdk7.java.net/download.html>.
2. Open the environment variables and set the JAVA_HOME variable.



Neo4j can be installed and executed as a Windows service or it can also be downloaded as a .zip file, where after installation you need to execute a .bat script to run it as a standalone application.

Further in this section, we will talk about the steps involved in installing Neo4j as a service and standalone archive.

Installing as a Windows service

Installing Neo4j as a Windows service had always been a preferred procedure for administrators, especially in production environments, where you want your critical applications to be available for use at the server start-up itself and should also survive user logons/logoffs. Not only in the production environment but also in other development / QA environments, users have always preferred ease of installation, configuration, and upgradation, which can only happen when we install Neo4j as a service.




You will require admin privileges on your system to install Neo4j as a Windows Service.

To install Neo4j as a Windows service, let's perform the following steps:

1. Depending upon your Windows platform (64 bit or 32 bit), download the latest stable release of Neo4j from <http://neo4j.com/download/other-releases/>.

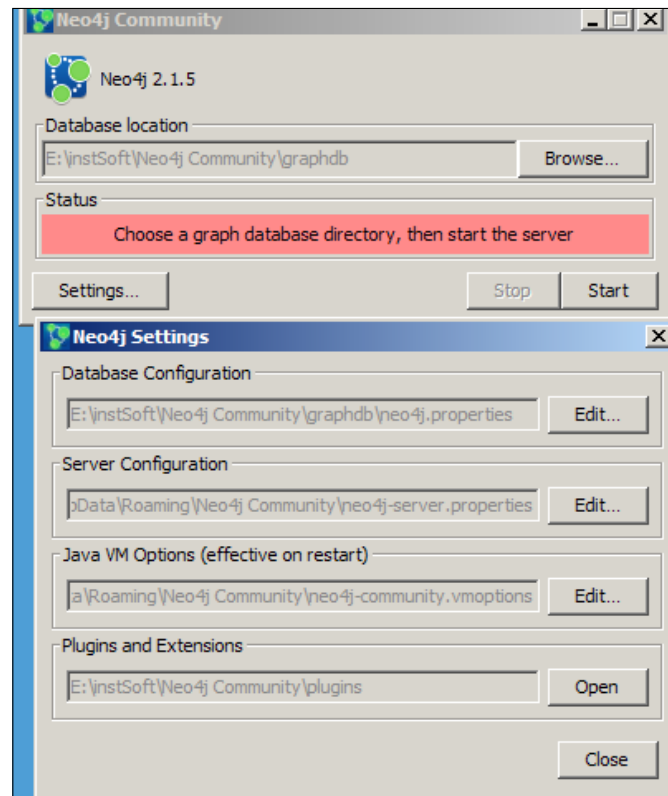
Neo4j Release	Community	Enterprise
Neo4j 2.1.5 4 September, 2014 Release Notes Read More		
Linux/Mac	Neo4j 2.1.5	Neo4j 2.1.5
Windows 64 bit	Neo4j 2.1.5 (exe) Neo4j 2.1.5 (zip)	Neo4j 2.1.5 (zip)
Windows 32 bit	Neo4j 2.1.5 (exe) Neo4j 2.1.5 (zip)	Neo4j 2.1.5 (zip)

2. Once downloaded, double-click on the .exe file and follow the instructions as they appear on your screen.
3. As soon as the installation process is finished, it will ask you to start the server and select the location where you want to create and store the graph database.

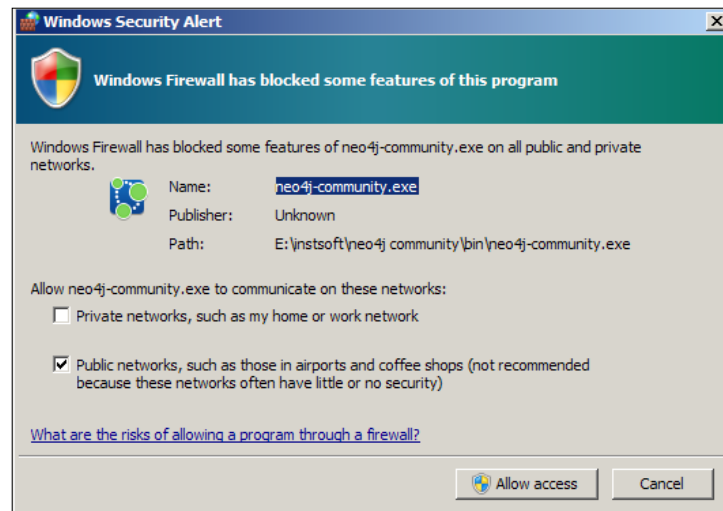
 Always provide an empty directory to store graph data for fresh installations.

By default it will choose to create data files at: %ROOT_DIR%\Users\%USER%\Documents\Neo4j\default.graphdb.

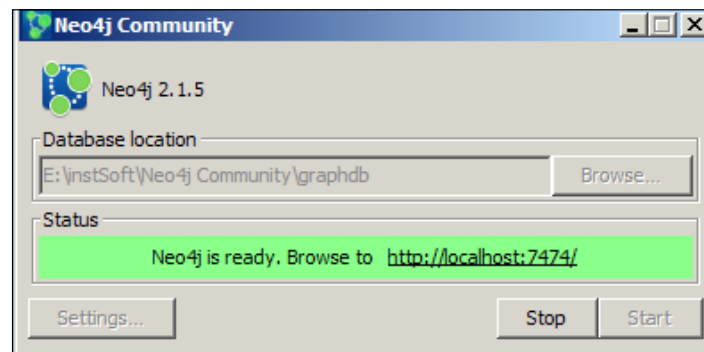
4. On the same window, click on **Settings** and it will show you the location of various configuration files and their corresponding settings.




5. In case you are behind the firewall, it will ask for permissions to allow the server to communicate with certain ports.



6. Click on **Allow access** and your Neo4j installation is complete and the server is ready for further development.



Browse <http://localhost:7474/browser/> and you should see the Neo4j browser.

[ The **Neo4j Community** link will also be available on your Windows Start menu for starting and stopping the server.]

Installing as a Windows archive / standalone application

There is no doubt that administrators/users have always preferred installing critical software such as Neo4j as a Windows service; however, there are instances where users do not have sufficient privileges to install software as a service, and for this reason, there is another way that users can download Neo4j and can do manual configuration and start using it.

To install Neo4j as a Windows archive, let's perform the following steps:

1. Depending upon your Windows platform (64 bit or 32 bit), download the latest stable release of Neo4j from <http://neo4j.com/download/other-releases/>.

Neo4j Release	Community	Enterprise
Neo4j 2.1.5 4 September, 2014 Release Notes Read More		
Linux/Mac	Neo4j 2.1.5	Neo4j 2.1.5
Windows 64 bit	Neo4j 2.1.5 (exe) Neo4j 2.1.5 (zip) ←	Neo4j 2.1.5 (zip)
Windows 32 bit	Neo4j 2.1.5 (exe) Neo4j 2.1.5 (zip) ←	Neo4j 2.1.5 (zip)


2. Once downloaded, extract the archive into any of the selected folders and refer to the top-level extracted directory as `NEO4J_HOME`.
3. Open the `<$NEO4J_HOME >\bin` directory and double-click on `Neo4j.bat` and you are done.
4. Stop the server by typing `Ctrl + C`.
5. Browse to <http://localhost:7474/browser/> and you should see the Neo4j browser.

Installing Neo4j Community Edition on Linux/Unix

This section will guide you through the end-to-end process of Neo4j installation on the Linux/Unix operating system. By the end of this section, you will have a fully functional Neo4j instance running on your Linux/Unix desktop / server.

Perform the following steps to install Neo4j on Linux/Unix:

1. Download and install Oracle Java 7 available at <http://www.oracle.com/technetwork/java/javase/install-linux-self-extracting-138783.html> or Open JDK 7 available at <https://jdk7.java.net/download.html>.
2. Set JAVA_HOME as the environment variable `export JAVA_HOME=<Path of Java install Dir>`.
3. Download the stable release of Linux distribution `neo4j-community-2.1.5-unix.tar.gz` from <http://neo4j.com/download/other-releases/>.

Neo4j Release	Community	Enterprise
Neo4j 2.1.5 4 September, 2014 Release Notes Read More		
Linux/Mac	Neo4j 2.1.5 	Neo4j 2.1.5
Windows 64 bit	Neo4j 2.1.5 (exe) Neo4j 2.1.5 (zip)	Neo4j 2.1.5 (zip)
Windows 32 bit	Neo4j 2.1.5 (exe) Neo4j 2.1.5 (zip)	Neo4j 2.1.5 (zip)

Neo4j can be installed and executed as a Linux service, or it can be downloaded as a `.tar` file, where after installation it needs to be started manually by executing the shell scripts.

Further in this section, we will talk about the steps involved in installing Neo4j as a service and standalone archive.

Installing as a Linux tar / standalone application

There are no second thoughts that administrators/users have always preferred installing critical software such as Neo4j as Linux service; however, there are instances where users do not have sufficient privileges to install software as a service, there is another way that users can download Neo4j, perform some manual configuration, and start using it.

Let's perform the following steps to install Neo4j as a Linux tar / standalone application:

1. Once the Neo4j archive is downloaded, browse the directory where you want to extract the neo4j server and untar the Linux/Unix Archive—`tar -xf <location of Archive file>`. Let's refer to the top-level extracted directory as `$NEO4J_HOME`.
2. Open Linux shell or console and execute the following commands to start the server:
 - `$NEO4J_HOME/bin/neo4j - start` - This command is used to run the server in a new process.
 - `$NEO4J_HOME/bin/neo4j - console` - This command is used to run the server in the same window without forking a new process.
 - `$NEO4J_HOME/bin/neo4j - restart` - This command is used to restart the server.
3. Browse `http://localhost:7474/browser/` and you should see the Neo4j browser.
4. Stop the server by typing `Ctrl + C` or `<$NEO4J_HOME>/bin/ neo4j - stop`.

Installing Neo4j as a Linux service

Installing Neo4j as a Linux service has always been a preferred procedure for administrators, especially in production environments where you want your critical applications to be available for use at the server start-up itself and to also survive user logons/logoffs. Not only in the production environment but in other development / QA environments, users have always preferred ease of installation, configuration, and up-gradation, which can only happen when we install Neo4j as a service.

To install Neo4j as a Linux Service, let's perform the following steps:

1. Once the Neo4j archive is downloaded, browse the directory where you want to extract the Neo4j server, untar the Linux / Unix Archive—`tar -xf <location of Archive file>`, and refer to the top-level extracted directory as `$NEO4J_HOME`.
2. Change the directory to `$NEO4J_HOME`, execute the `sudo bin/neo4j neo4j-installer install` command, and follow the steps as they appear on screen.



The installation procedure will provide an option to select the user that will be used to run the Neo4j server. You can supply any existing or new Linux user (it defaults to Neo4j). In case a user is not present, then it will be created as a system account and the ownership of `$NEO4J_HOME/data` will be moved to that user.

3. Once the installation is successfully completed, execute `sudo service neo4j-service start` on the Linux console to start the server and `sudo service neo4j-service stop` to gracefully stop the server.
4. Browse `http://localhost:7474/browser/` and you should see the Neo4j browser.



Accessing the Neo4j browser on a remote machine

To access the Neo4j browser on a remote machine, enable and modify `org.neo4j.server.webserver.address` in `neo4j-server.properties` and restart the server.

Installing the Neo4j Enterprise Edition

Enterprise Edition provides various features such as high availability (HA), fault tolerance, replication, backup, recovery, and many more. It also provides the features to set up a cluster of neo4j instances for achieving HA.

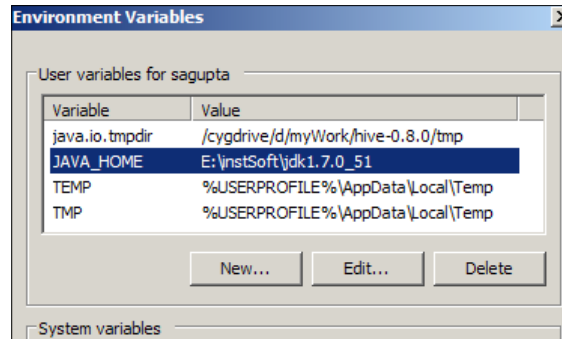
Setting up the Neo4j cluster will be quite similar to the single node setup except for a few properties that need to be modified for identification of the node in a cluster.

Configuring a Neo4j cluster on Windows

Let's perform the following steps to install the Neo4j Enterprise Edition on Windows:

1. Download and install Oracle Java 7, available at <http://www.oracle.com/technetwork/java/javase/install-linux-self-extracting-138783.html> or Open JDK 7, available at <https://jdk7.java.net/download.html>.

2. Open the environment variables and set the JAVA_HOME variable:



3. Depending upon your platform (64 or 32 bit), download the appropriate Windows archive from <http://neo4j.com/download/other-releases/>.
4. Once downloaded, extract the archive into any of the selected folders and let's refer to the top-level extracted directory as NEO4J_HOME.
5. Open <\$NEO4J_HOME>\conf\neo4j-server.properties and enable/modify the following properties:
 - #org.neo4j.server.database.mode=HA: This property allows us to keep this value as HA. You can run it as standalone too by providing the value as SINGLE.
 - #org.neo4j.server.webserver.address=0.0.0.0: This property allows us to enable and provide the IP of the Node for enabling remote access.
6. Open <\$NEO4J_HOME>\conf\neo4j.properties and enable/modify the following properties:
 - "#ha.server_id=": This is the Unique ID of each node that will participate in the cluster. It should be an integer (1, 2, or 3).
 - "#ha.cluster_server=192.168.0.1:5001"; this is the IP address and port for communicating cluster status information with other instances.
 - "#ha.server=192.168.0.1:6001": This is the IP address and port of the node for communicating transactional data with other instances.
 - "#ha.initial_hosts=192.168.0.1:5001,192.168.0.2:5001": This is a comma-separated list of the host port (ha.cluster_server), where all nodes will be listening. This would be the same for all nodes participating in the same cluster.

- `"#remote_shell_enabled=true"`: Enable this property for connecting the server remotely through Shell.
 - `"#remote_shell_host=127.0.0.1"`: Enable this property and provide the IP address where the remote shell will be listening.
 - `"#remote_shell_port=1337"`: Enable this property and provide the port at which the shell will listen. You can keep it as default in case the default port is not being used by any other process.
7. Open the `<$NEO4J_HOME>\bin` directory and double-click on `Neo4j.bat` and you are done. Stop the server by typing `Ctrl + C`.
 8. Browse `http://<IP>:7474/browser/` for interactive shell or browse `http://<IP>:7474/webadmin/` to perform administrative operations or to analyze server statistics.

Configuring a Neo4j cluster on Linux/Unix

Let's perform the following steps to install Neo4j Enterprise Edition on Linux:

1. Download and install Oracle Java 7, available at <http://www.oracle.com/technetwork/java/javase/install-linux-self-extracting-138783.html> or Open JDK 7, available at <https://jdk7.java.net/download.html>.
2. Set `JAVA_HOME` as the environment variable using the following command in your Terminal:
`export JAVA_HOME=<Path of Java install Dir>`
3. Download the stable release of the Linux distribution `neo4j-enterprise-2.1.5-unix.tar.gz` from <http://neo4j.com/download/other-releases/>.
4. Once downloaded, extract the archive into any of the selected folders and let's refer to the top-level extracted directory as `NEO4J_HOME`.
5. Open `<$NEO4J_HOME>\conf\neo4j-server.properties` and enable/modify the following properties:
 - `"#org.neo4j.server.database.mode=HA"`: This property allows us to keep this value as `HA`. You can run it as standalone too by providing the value as `SINGLE`.
 - `"#org.neo4j.server.webserver.address=0.0.0.0"`: This property allows us to enable and provide the IP of the node to enable remote access.

6. Open `<$NEO4J_HOME>\conf\neo4j.properties` and enable/modify the following properties:
 - `"#ha.server_id="`: This is the unique ID of each node that will participate in the cluster. It should be an integer (1, 2, or 3).
 - `"#ha.cluster_server=192.168.0.1:5001"`: This is the IP address and port for communicating cluster status information with other instances.
 - `"#ha.server=192.168.0.1:6001"`: This is the IP address and port of the node for communicating transactional data with other instances.
 - `"#ha.initial_hosts=192.168.0.1:5001,192.168.0.2:5001"`: This is a comma-separated list of the host port (`ha.cluster_server`), where all the nodes will be listening. This would be the same for all the nodes participating in the same cluster.
 - `"#remote_shell_enabled=true"`: This property enables us to connect to the server remotely through shell.
 - `"#remote_shell_host=127.0.0.1"`: This property enables and provides the IP address where remote shell will be listening.
 - `"#remote_shell_port=1337"`: This property enables and provides the port at which Shell will listen. You can keep it as default in case the default port is not being used by any other process.
7. Open `<$NEO4J_HOME>/bin` and execute `./neo4j start` and you are done. Stop the server by typing `Ctrl + C`.
8. Browse `http://<IP>:7474/browser/` for interactive shell or browse to `http://<IP>:7474/webadmin/` to perform administrative operations or to analyze server statistics.

Tools and utilities for administrators/developers

Neo4j comes with various tools and utilities which ease the life of a developer. Let's talk about each of them and see how they are helpful to developers:

- `<$NEO4J_HOME>/bin/neo4j`: This utility is used to start, stop, restart, or get the current status of the Neo4j server, which is not installed as a service and is executed as a standalone application. Here are the different options available with this utility:
 - `neo4j console`: This option starts the Neo4j server as a foreground process.

-
- `neo4j start`: This option starts the Neo4j server in a different process.
 - `neo4j stop`: This option shuts down the Neo4j server gracefully.
 - `neo4j status`: This option shows the current status of the Neo4j server.
 - `<$NEO4J_HOME>/bin/neo4j-installer`: This utility is used to install/uninstall the Neo4j server (as a service) on/from the local system. Here are the different options available with this utility:
 - `neo4j-installer install`: This option is used to install the Neo4j server as a service on the local system.
 - `neo4j-installer remove`: This option is used to uninstall/remove the server from the local system.
 - `neo4j-installer status`: This option provides the status of the Neo4j server.
 - `neo4j-installer usage`: This option prints the various parameters available for use.
 - `neo4j-installer start`: This option starts the service of the Neo4j server.
 - `neo4j-installer stop`: This option gracefully stops the service of the Neo4j server.
 - `<$NEO4J_HOME>/bin/neo4j-shell`: Here, `neo4j-shell` is a powerful interactive shell used to interact with the Neo4j database. It is used to perform CRUD operations on graphs.

Neo4jshell can be executed locally on the same machine where we have installed the Neo4j server or by remotely connecting Neo4j shell to a remote sever.

Here are the various options available with this utility for connecting to the local Neo4j server:

- `neo4j-shell -path <PATH>`: This option is the path to the database directory on a local filesystem. A new database will be created in case the given path does not contain any valid neo4j database.
- `neo4j-shell -pid <PID>`: This option is used to connect to a specific process ID.
- `neo4j-shell -readonly`: This option is used to connect to the local database in READONLY mode.
- `neo4j-shell -c <COMMAND>`: This option is used to execute a single statement and then shell exits.

- `neo4j-shell -file <FILE>`: This option is used to read the contents of the file (multiple CRUD operations) and then execute it.
- `neo4j-shell -config - <CONFIG>`: This option is used to read the given configuration file (`neo4j-server.properties`) from the specified location and then start the shell.

Here are the various options available with this utility for connecting to the remote Neo4j server:

- `neo4j-shell -port <PORT>`: This option is used to connect to the server running on a different port other than the default port (1337).
- `neo4j-shell -host <HOST>`: This is the IP address or domain name of the remote host on which the neo4j server is installed and is currently running.
- `neo4j-shell -name <NAME>`: This option is used to connect to the remote host by getting the reference of the object binded within the RMI registry.

The following configuration changes are required to enable your shell to connect remotely. Open `<$NEO4J_HOME>/conf/neo4j.properties` and enable the following properties, providing an appropriate value against each of them:

- `#remote_shell_enabled = true`
- `#remote_shell_host = 127.0.0.1`
- `#remote_shell_port = 1337`



All the utilities can be found under `<$NEO4J_HOME>/bin`.

There are two more utilities that are only available with the Enterprise Edition:

- `<$NEO4J_HOME>/bin/neo4jArbiter`: The Arbiter instances are used to take part in master elections with the single purpose of breaking ties in the election process. It will be difficult to understand at this point, but as of now, let's just remember the definition and we will talk about Arbiters a little later in *Chapter 7, Neo4j Deployment*.
- `<$NEO4J_HOME>/bin/backups`: This utility helps administrators/developers in scheduling backups or performing manual backups of Neo4j database.



Online backups can also be performed by enabling the `online_backup_enabled` and `online_backup_server` properties.

Neo4j also provides a rich UI for interacting with the database and performing various CRUD operations visually such as querying data, managing indexes, and so on. It is similar to the Neo4j shell, which is a command-line utility but does not provide a UI.

Neo4j provides two kinds of web interfaces:

- **Neo4j browser:** It is available at `http://<IP or Domain_name>:7474/browser/` and is the primary **User Interface (UI)** of Neo4j that provides a rich UI for visualizing the data in the form of graphs. It also facilitates in running your Cypher queries in an interactive UI-based console and showing the results in pretty graphs and tables.
- **Neo4j WebAdmin:** It is available at `http://<IP or Domain_Name>:7474/webadmin/` and is similar to the Neo4j browser but comes with a classic UI and does not provide rich UI.

Running your first Cypher query

After Neo4j installation, we are ready to get our hands dirty with the system.

To begin with and to make it simple, first we will insert the data and then try to fetch the same data in four different ways:

- Interactive console (Neo4j shell)
- REST APIs
- Custom Java code with embedded database
- Neo4j browser

Interactive console – Neo4j shell

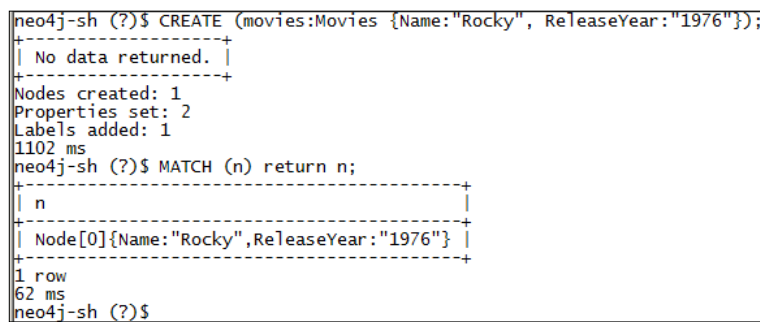
Neo4j shell is a command-line utility for performing CRUD operations on the Neo4j database. In this section, we will talk about the step-by-step process involved in executing Cypher queries using `neo4j-shell`.

Perform the following steps to run your Cypher queries in `neo4j-shell`:

1. Assuming that your server is already started, open a new Command Prompt, browse `<$NEO4J_HOME>/bin/`, and execute `neo4j-shell`. This will show you the shell prompt where you can execute your Cypher queries.
2. Next, execute the following set of statements on the console:

```
CREATE (movies:Movies {Name:"Rocky", ReleaseYear:"1976"});  
MATCH (n) return n;
```

3. You will see something like the following screenshot on your console:



```
neo4j-sh (?)$ CREATE (movies:Movies {Name:"Rocky", ReleaseYear:"1976"});  
+-----+  
| No data returned. |  
+-----+  
Nodes created: 1  
Properties set: 2  
Labels added: 1  
1102 ms  
neo4j-sh (?)$ MATCH (n) return n;  
+-----+  
| n |  
+-----+  
| Node[0]{Name:"Rocky",ReleaseYear:"1976"} |  
+-----+  
1 row  
62 ms  
neo4j-sh (?)$
```

WOW!!! We are done! Simple, isn't it?

In the preceding example, we created a node called `Movies` with two attributes—`Name` and `ReleaseYear`. After creation, we fetched the same data using the `Match` query.

Working with REST APIs

Neo4j also exposes several REST APIs for performing CRUD operations on the Neo4j database. In this section, we will talk about the step-by-step process involved in fetching data from the Neo4j database using REST APIs.

Perform the following steps to invoke the Neo4j REST APIs:

1. Download any tool such as SoapUI, which provides creation and execution of REST calls.
2. Open your tool and execute the following request and parameters:
 - Request method type: `POST`
 - Request URL: `http://localhost:7474/db/data/transaction`

- Request headers: Accept: application/json; charset=UTF-8 and Content-Type: application/json
- JSONRequest: {"statements": [{"statement" : "MATCH (n) return n;"}]}

The result would be a JSON response containing data inserted in the previous example.

Java code API and embedding the Neo4j database

Neo4j provides a rich set of Java APIs that help in embedding the Neo4j database within the Java/J2EE application as a headless application. Let's see how it works:

1. Shut down the Neo4j server by executing `$NEO4J_HOME/bin/neo4j stop`. Now using any IDE, create a new Java project and define the following dependencies to your project:
 - JDK 1.7
 - All JAR files in `$NEO4J_HOME/lib/`
2. Create a new Java file and add the following code:

```
import org.neo4j.graphdb.GraphDatabaseService;
import org.neo4j.graphdb.Label;
import org.neo4j.graphdb.Node;
import org.neo4j.graphdb.Transaction;
import org.neo4j.graphdb.factory.GraphDatabaseBuilder;
import org.neo4j.graphdb.factory.GraphDatabaseFactory;
import org.neo4j.tooling.GlobalGraphOperations;

public class FetchAllData {
    public static void main(String[] args) {
        //Complete path of the database files on local System
        //Use the same path "$NEO4J_HOME/data/graph.db"
        String location = "$NEO4J_HOME/data/graph.db";

        //Create instance of Graph Database.
        GraphDatabaseFactory fac = new GraphDatabaseFactory();
        GraphDatabaseBuilder build =
            fac.newEmbeddedDatabaseBuilder(location);
        GraphDatabaseService dbService =
            build.newGraphDatabase();
    }
}
```

```
//Starting a Transaction...All CRUD operations should be in
Transaction
try (Transaction tx = dbService.beginTx()) {
    GlobalGraphOperations operations =
        GlobalGraphOperations
            .at(dbService);

    //Getting All Nodes
    for (Node node : operations.getAllNodes()) {
        System.out.println("Node Labels.....");

        //Get All Labels associated with that Node
        for (Label label : node.getLabels()) {
            System.out.println(label.name());
        }
        System.out.println("Node Properties = ");

        //Get All Keys/ Value associated with that Node
        for (String key : node.getPropertyKeys()) {
            System.out.println(key + " = " +
                node.getProperty(key));
        }
    }
}
```

Downloading the example code



You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Compile the preceding code and execute it from the IDE itself. The preceding code will dump the complete data on the console.

In order to understand the preceding code, follow the comments provided before every line of the code.

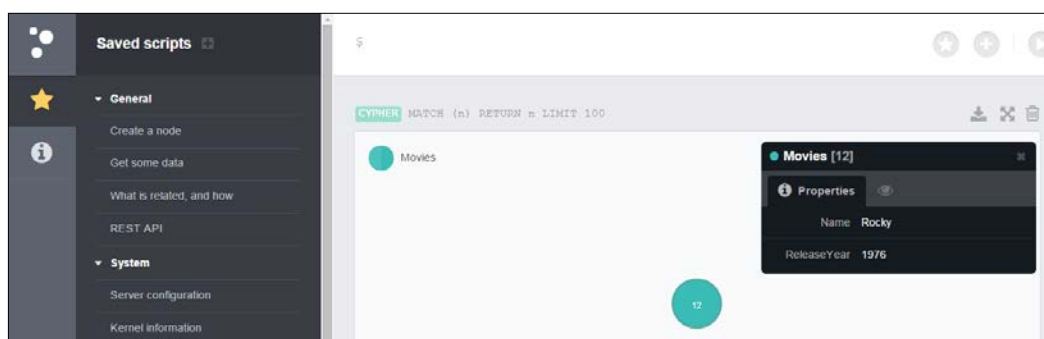
The Neo4j browser

Neo4j provides a UI-based interactive console for performing CRUD operations on the Neo4j database. The Neo4j browser not only provides the console for executing Cypher queries but also a REST client to invoke the REST endpoints exposed by the Neo4j server.

In this section, we will talk about the step-by-step process involved in executing Cypher queries using the Neo4j browser.

Perform the following steps for running your Cypher queries in the Neo4j browser:

1. Assuming that your server has already started, open a new browser window and type `http://localhost:7474` in the navigation bar of the browser. This will show you the Neo4j browser where you can execute your Cypher queries.
2. Next, click on the **Get some data** button from the left navigation pane and then execute the query by clicking on the right arrow sign that will appear on the extreme right corner, just below the browser navigation bar, which will look something like this:



You can also click on the specific node to see its properties and type/execute other Cypher queries in the space given next to \$ (below the browser navigation bar) as shown in the preceding screenshot. You can also invoke REST endpoints by clicking on **REST API** from the left navigation pane.

In this section, we walked you through the different ways to perform CRUD operation using various APIs exposed by Neo4j. You have not only executed your "first Cypher query" through the `neo4j-shell`, but you have also learned the other ways of accessing data through REST and Java APIs.

Summary

Throughout this chapter, we have gone through various licensing, installation, and step-by-step deployment options of the Neo4j database/server. We have also touched upon various ways to perform CRUD operations.

In the next chapter, we will talk about various bulk data import options and integration with third-party ETL tools.

2

Ready for Take Off

We never work in isolation and this applies even to enterprises where they always look for systems that can be efficiently and seamlessly integrated with many other tools and technologies, so that they can use the best fit technology for their use cases, also known as best of all possible worlds.

Apart from being a robust and massively scalable graph database, storing billions of nodes and their associations/relationships, Neo4j also provides several ways to integrate with **Business Intelligence (BI)** / Visualization tools. As per <http://businessintelligence.com/dictionary/business-intelligence-2/> BI is defined as:

"A term that refers to ideas, practices, and technologies for turning raw data into information that businesses can use to make better organizational decisions. Businesses that employ BI effectively can transform information into growth by gaining a clear understanding of their strengths and weaknesses, cutting costs and losses, streamlining internal processes, and increasing revenue."

BI helps in visualizing data in the form of intuitive dashboards for connected data, which helps businesses to take informed decisions for their organization's growth and future.

Neo4j also provides features to import/export data to/from other databases working for the same purpose or different purposes, in a performance-efficient manner.

At the end of this chapter, you will be able to insert data within your Neo4j database from other data sources and integrate with BI / Visualization and ETL tools.

This chapter will cover the following topics:

- Integration with BI / visualization and ETL tools
- Batch import
- Performance tuning / optimizations

Integration of the BI tool – QlikView

In this section, we will talk about the integration of the BI tool – QlikView with Neo4j. QlikView is available only on the Windows platform, so this section is only applicable for Windows users.

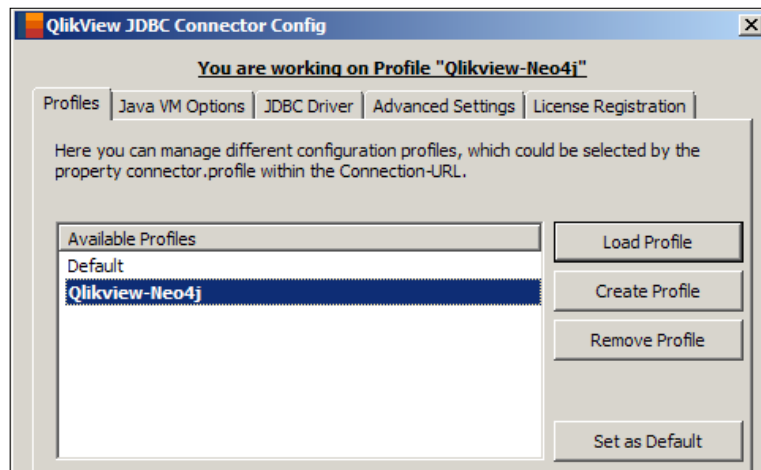
Neo4j as an open source database exposes its core APIs for developers to write plugins and extends its intrinsic capabilities.

Neo4j JDBC is one such plugin that enables the integration of Neo4j with various BI / visualization and ETL tools such as QlikView, Jaspersoft, Talend, Hive, Hbase, and many more.

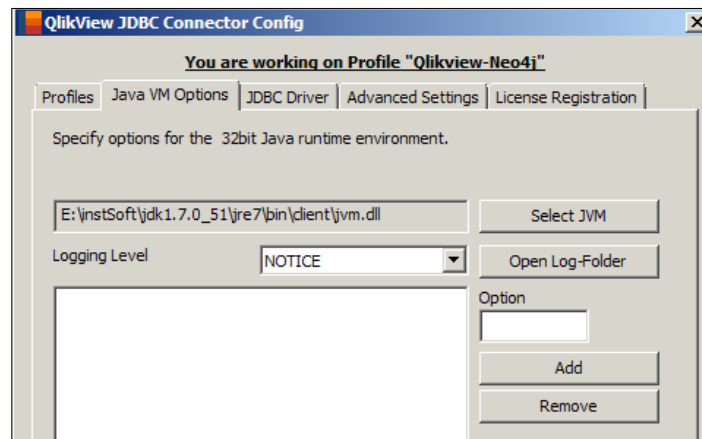
Let's perform the following steps for integrating Neo4j with QlikView on Windows:

1. Download, install, and configure the following required software:
 1. Download the Neo4j JDBC driver directly from <https://github.com/neo4j-contrib/neo4j-jdbc> as the source code. You need to compile and create a JAR file or you can also directly download the compiled sources from <http://dist.neo4j.org/neo4j-jdbc/neo4j-jdbc-2.0.1-SNAPSHOT-jar-with-dependencies.jar>.
 2. Depending upon your Windows platform (32 bit or 64 bit), download the QlikView Desktop Personal Edition. In this chapter, we will be using QlikView Desktop Personal Edition 11.20.12577.0 SR8.
 3. Install QlikView and follow the instructions as they appear on your screen. After installation, you will see the QlikView icon in your Windows start menu.
2. QlikView leverages QlikView JDBC Connector for integration with JDBC data sources, so our next step would be to install QlikView JDBC Connector. Let's perform the following steps to install and configure the QlikView JDBC Connector:
 1. Download QlikView JDBC Connector either from <http://www.tiq-solutions.de/display/enghome/ENJDBC> or from https://s3-eu-west-1.amazonaws.com/tiq-solutions/JDBCCConnector/JDBCCConnector_Setup.zip.
 2. Open the downloaded JDBCCConnector_Setup.zip file and install the provided connector.
 3. Once the installation is complete, open JDBC Connector from your Windows Start menu and click on Active for a 30-day trial (if you haven't already done so during installation).

4. Create a new profile of the name **Qlikview-Neo4j** and make it your default profile.



5. Open the **JVM VM Options** tab and provide the location of `jvm.dll`, which can be located at `<$JAVA_HOME>/jre/bin/client/jvm.dll`.

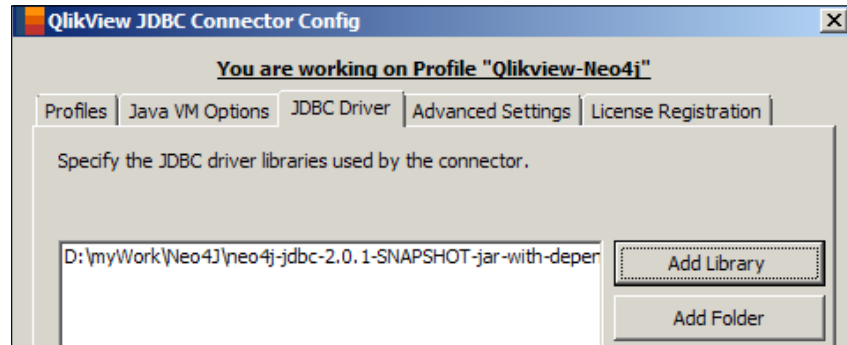


6. Click on **Open Log-Folder** to check the logs related to the Database connections.



You can configure the **Logging Level** and also define the JVM runtime options such as `-Xmx` and `-Xms` in the textbox provided for **Option** in the preceding screenshot.

7. Browse through the **JDBC Driver** tab, click on **Add Library**, and provide the location of your <\$NEO4J-JDBC driver>.jar, and add the dependent JAR files.

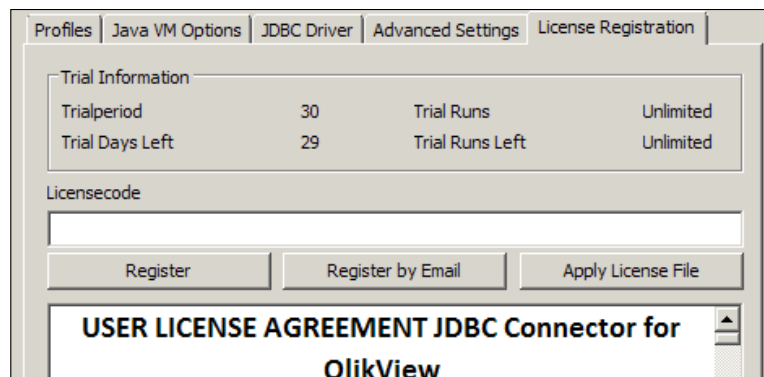


Instead of adding individual libraries, we can also add a folder containing the same list of libraries by clicking on the **Add Folder** option.



We can also use non JDBC-4 compliant drivers by mentioning the name of the driver class in the **Advanced Settings** tab. There is no need to do that, however, if you are setting up a configuration profile that uses a JDBC-4 compliant driver.

8. Open the **License Registration** tab and request Online Trial license, which will be valid for 30 days. Assuming that you are connected to the Internet, the trial license will be applied immediately.

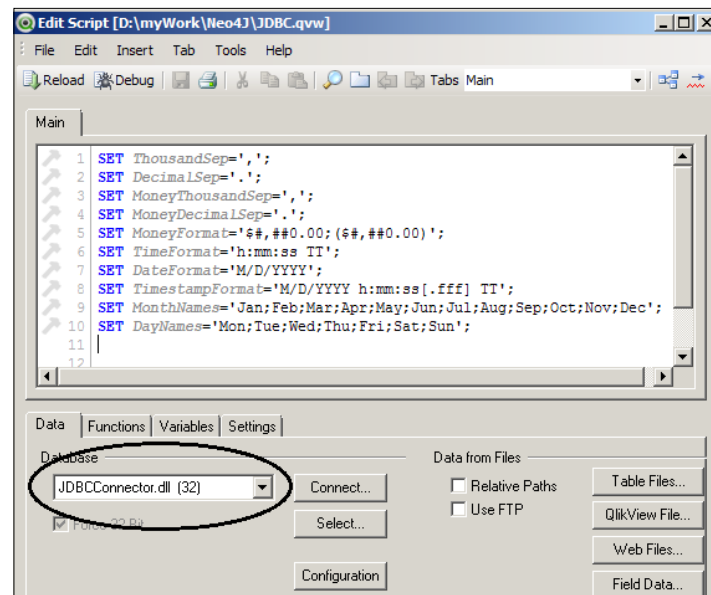


9. Save your settings and close QlikView JDBC Connector configuration.

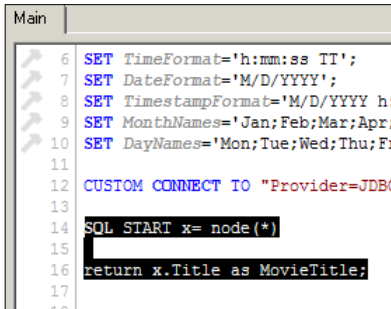
- Open <\$NEO4J_HOME>\bin\neo4jshell and execute the following set of Cypher statements one by one to create sample data in your Neo4j database; then, in further steps, we will visualize this data in QlikView:

```
CREATE (movies1:Movies {Title : 'Rocky', Year : '1976'});
CREATE (movies2:Movies {Title : 'Rocky II', Year :
'1979'});
CREATE (movies3:Movies {Title : 'Rocky III', Year :
'1982'});
CREATE (movies4:Movies {Title : 'Rocky IV', Year :
'1985'});
CREATE (movies5:Movies {Title : 'Rocky V', Year : '1990'});
CREATE (movies6:Movies {Title : 'The Expendables', Year :
'2010'});
CREATE (movies7:Movies {Title : 'The Expendables II', Year
: '2012'});
CREATE (movies8:Movies {Title : 'The Karate Kid', Year :
'1984'});
CREATE (movies9:Movies {Title : 'The Karate Kid II', Year :
'1986'});
```

- Open the QlikView Desktop Personal Edition and create a new view by navigating to **File | New**. The Getting Started wizard may appear as we are creating a new view. Close this wizard.
- Navigate to **File | EditScript** and change your database to **JDBCConnector.dll (32)**.

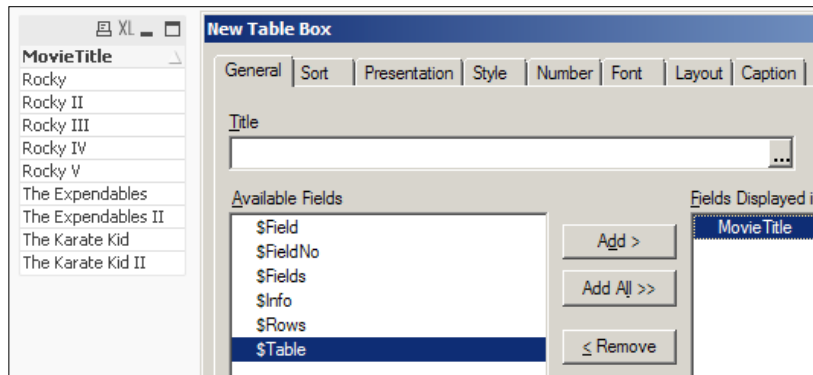


6. In the same window, click on **Connect** and enter "jdbc:neo4j://localhost:7474/" in the "url" box.
7. Leave the username and password as empty and click on **OK**. You will see that a CUSTOM CONNECT TO statement is added in the box provided.
8. Next insert the highlighted Cypher statements in the provided window just below the CUSTOM CONNECT TO statement.



```
6 SET TimeFormat='h:mm:ss TT';
7 SET DateFormat='M/D/YYYY';
8 SET TimestampFormat='M/D/YYYY h:
9 SET MonthNames='Jan;Feb;Mar;Apr;
10 SET DayNames='Mon;Tue;Wed;Thu;Fr
11
12 CUSTOM CONNECT TO "Provider=JDBC
13
14 SQL START x= node(*)
15
16 return x.Title as MovieTitle;
17
```

9. Save the script and close the **EditScript** window.
10. Now, on your Qlikviewsheet, execute the script by pressing **Ctrl + R** on our keyboard.
11. Next, add a new TableObject on your Qlikviewsheet, select "MovieTitle" from the provided fields and click on **OK**.



And we are done!!!!

You will see the data appearing in the listbox in the newly created Table Object. The data is fetched from the Neo4j database and QlikView is used to render this data.

The same process is used for connecting to other JDBC-compliant BI / visualization / ETL tools such as Jasper, Talend, Hive, Hbase, and so on. We just need to define appropriate JDBC Type-4 drivers in JDBC Connector.



We can also use ODBC-JDBC Bridge provided by EasySoft at http://www.easysoft.com/products/data_access/odbc_jdbc_gateway/index.html. EasySoft provides the ODBC-JDBC Gateway, which facilitates ODBC access from applications such as MS Access, MS Excel, Delphi, and C++ to Java databases. It is a fully functional ODBC 3.5 driver that allows you to access any JDBC data source from any ODBC-compatible application.

Creating a ready-to-use database – batch imports

Batch data import is one of the most widely used and integral processes in enterprise systems. It not only helps to create our ready-to-use database in a few minutes, but is also helpful in situations such as system failures / crashes where we want to recover our systems and reload the data as soon as possible.

Neo4j provides the following processes for importing data in batches:

- CSV importer
- The spreadsheet way – Excel
- HTTP batch imports – REST API
- Java API

Let's discuss in detail each of the batch import processes.

The CSV importer

Neo4j exposes the `LOAD CSV` command for importing data from the CSV files (with or without headers). `LOAD CSV` can either use the `CREATE` or `MERGE` commands for importing the data into the database.

Let's understand each of the available options.

LOAD CSV with CREATE

The following steps will guide you through the process of using the `LOAD CSV` command:

1. Let's create a file by the name `neo4j-import.csv` in the root of your `<$NEO4J_HOME>` folder with the following content:

```
Name, WorkedAs
"Sylvester Stallone", "Actor, Director"
"John G. Avildsen", "Director"
"Ralph Macchio", "Actor"
"Simon West", "Director"
```
2. Once the CSV file is created, ensure that it has the following characteristics:
 - The character encoding of a CSV file should be UTF-8
 - End line termination is system dependent, for example, it is `\n` on Unix and `\r\n` on Windows
 - The default field terminator is `,`, though you can change it by using the `FIELDTERMINATOR` option
 - Quoted strings are allowed in the CSV file and the quotes are removed while reading the data
 - The character for string quotation is double quotes `"`
 - The escape character is `\`

3. In Unix, open `<$NEO4J_HOME>/bin/neo4j-shell` and enter the following command; you will see that four nodes, four properties, and four labels are added:

```
LOAD CSV WITH HEADERS FROM "file://<$NEO4J_HOME>/neo4j-import.csv"
as csv
CREATE (cinestars:Cinestars {Name : csv.Name, WorkedAs : split
(csv.WorkedAs, ",")});
```

Make sure that you replace `<$NEO4J_HOME>` with the actual path or location of the directory.

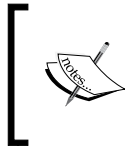
4. In Windows, open `<$NEO4J_HOME>\bin\neo4j-shell` and enter the following command; you will see that four nodes, four properties, and four labels are added:

```
LOAD CSV WITH HEADERS FROM "file:\\<$NEO4J_HOME>\\neo4j-import.
csv" as csv
```

```
CREATE (cinestars:Cinestar {Name : csv.Name, WorkedAs : split
(csv.WorkedAs, ",")});
```

Make sure that you replace `<$NEO4J_HOME>` with the actual path or location of the Neo4j installation directory..

And you are done!!! Easy isn't it?



LOAD CSV can import CSV files, which do not contain any headers. Replace the name of the header with their position in the file within `[]`. For example, the first column of `neo4j-import.csv` will be referred to as `csv[1]` and the second column would be `csv[2]` in the Cypher statement.

In subsequent sections and chapters, all examples will list down the steps for the Unix operating system, unless specified otherwise.

LOAD CSV with MERGE

In the previous example, we have used `CREATE` in `LOAD CSV`, which creates the new records in the database, but what happens when you have duplicate records in your CSV file?

That would create duplicate records in your database too, which will be a real problem.

In order to solve the issue with duplicate records, Neo4j provides the `MERGE` command, which if used will either match the existing nodes and bind them, or create new data and bind them. It is like a combination of `MATCH` and `CREATE` statements that additionally allow you to specify the action that needs to be performed in scenarios where data is matched or created.

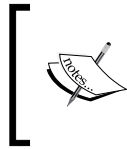
The behavior of `MERGE` is that it either matches the whole pattern, or it creates the whole pattern. It will not partially use existing patterns – it's all or nothing.

`MERGE` is accompanied and followed with `ON CREATE` and `ON MATCH`. These allow us to perform additional changes to the nodes, properties, and relationship depending upon the result of the `MERGE` command.

For example, the preceding `LOAD CSV` command can be re-written as:

```
LOAD CSV WITH HEADERS FROM "$NEO4J_HOME/neo4j-import.csv" as csv
MERGE (cinestars:Cinestar {Name : csv.Name})
ON CREATE SET cinestars.WorkedAs = split (csv.WorkedAs, ",");
```

The preceding command will load the data and set the properties of the node only if it is created. No changes will be performed in case the node already exists.



LOAD CSV only works with Cypher 2.1 and above. Check the version of kernel by executing `dbinfo -g Kernel KernelVersion` on `neo4j-shell` (in Unix) or `neo4jshell` (in Windows) and if required upgrade or try appending Cypher 2.1 before your `LOAD CSV` statement.

The spreadsheet way – Excel

Neo4j does not provide direct support for importing Excel documents; however, by leveraging the capabilities of Excel, we can create dynamic cypher queries and then execute all those queries in a batch mode.

Let's consider an example and see how it works.

Assume that your data is organized in an Excel sheet, as per the following screenshot where columns **A**, **B**, and **C** contain the data and column **D** contains the dynamic Cypher:

A	B	C	D	E
ID	Cinestars	Label	CYPHER QUERY	
1	Kate Winslet	Female	CREATE (n:Female {id:1, name: 'Kate Winslet'});	
2	Julia Roberts	Female	CREATE (n:Female {id:2, name: 'Julia Roberts'});	
3	Johnny Depp	Male	CREATE (n:Male {id:3, name: 'Johnny Depp'});	
4	Arnold Schwarzenegger	Male	CREATE (n:Male {id:4, name: 'Arnold Schwarzenegger'});	

Columns **A**, **B**, and **C** are straightforward to understand as they contain plain data. The tricky part is column **D** where we have entered the following Excel formula:

```
= "CREATE (n:"&C2&" {id:"&A2&", name: '"&B2&"'});"
```

Copy the same formula for all other rows and we get similar Cypher queries based on the data provided in Columns **A**, **B**, and **C**.

Next, perform the following steps to create the batch import file:

1. Create a new empty file and save it as `import.txt` in your `<$NEO4J_HOME>` folder.
2. Edit `import.txt` and add `BEGIN` at the top of the file.
3. Copy the values of column **D** into `import.txt`, below the `BEGIN` line.
4. Add `COMMIT` to the end of the file.
5. Save `import.txt`.

6. Stop your Neo4j server.

The Neo4j server needs to be stopped because in the next step we will be using Neo4j shell to start the Neo4j server in embedded mode and perform direct imports (also called low level imports). Now, for starting the Neo4j server in embedded mode, Neo4j shell needs to take a look at the database files, which otherwise would be with the Neo4j server.

7. Open your console, browse <\$NEO4J_HOME>, and execute the following command:

```
cat import.txt | $NEO4J_HOME/bin/neo4j-shell -config conf/neo4j.properties -path $NEO4J_HOME/data/graph.db
```

Now all you need to do is start your Neo4j server again and you will be able to query the data.

HTTP batch imports – REST API

Neo4j REST architecture is one of the most well-crafted and well-designed architectures. Designed on principles of service discoverability, it exposes the service root, and from there users can discover other URIs to perform various CRUD and search operations.

All endpoints are relative to `http://<HOST>:<PORT>/db/data/` which is also known as a service root for other REST end points. You can simply fire the service root, using GET operations, and it will show its other REST endpoints.

The default representation for all types of request (POST/PUT) and response is JSON.

In order to interact with the JSON interface, users need to explicitly set the request header as `Accept:application/json` and `Content-Type:application/json`.

One of the end points exposed by Neo4j REST architectures is `/batch/`. This is used to perform batch operations on the Neo4j database. It provides an interface for multiple API calls through a single HTTP request, which significantly improves performance for a large number of insert and update operations.

The API expects an array of job descriptions as input and each job description describes an action to be performed via the normal server API.

This service is executed in a transaction, and in case any operation fails, all modifications are rolled back.

Let's first understand the structure of a sample batch request, which defines multiple jobs within a single request and then we will also execute the same request against our local Neo4j database:

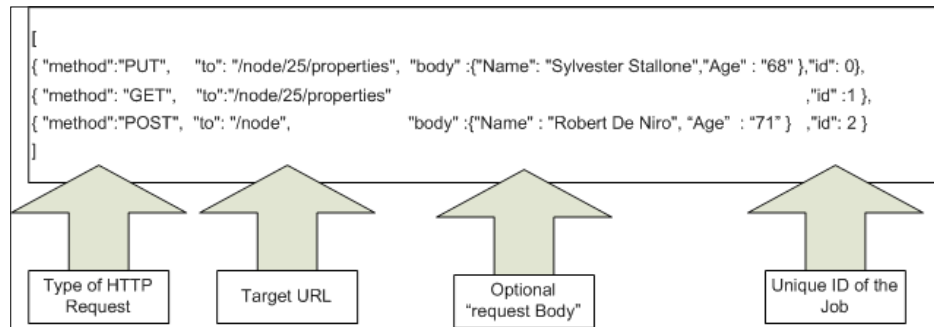
```
[ {
  "method": "PUT",
  "to": "/node/25/properties",
  "body" : {
    "Name": "Sylvester Stallone",
    "Age" : "68"
  }, "id": 0
},
{
  "method": "GET",
  "to": "/node/25/properties",
  "id" : 1
},
{
  "method": "POST",
  "to": "/node",
  "body" : {
    "Name" : "Robert De Niro",
    "Age" : "71"
  },
  "id": 2
}]
```

Let's discuss the role of each of the different sections of the preceding JSON request:

- **method:** This defines the type of operation (GET, PUT, POST, DELETE, and so on).
- **to:** This is the target URL where we submit our request. The URL can request any type of operation on nodes, attributes, or relationships. For example, in Job-0 ("id:"0") we request the PUT operation on node with node ID = 25, for Job-1 GET operation on node with node ID = 25 and for Job-2 we use the POST operation for adding a new node. Node ID is unique and has internal IDs used by Neo4j; they are not guaranteed to be the same for two different setups. So, before executing the preceding example, find the node ID of the node, which contains the Name property as *Sylvester Stallone* and then replace it in the preceding request. Node ID can be queried by executing the following Cypher query on Neo4j shell:

```
match (n) where n.Name = "Sylvester Stallone" return n;
```

- **body**: This is the optional attribute for sending the parameters.
- **Id**: This defines the unique ID of each job. Results, when returned, are identified by the ID. The order of the results is the same as the order of the jobs when submitted.



Use any tool such as SoapUI, which supports testing of REST-based services / representation, execute the preceding JSON request using the following configuration, and see the results:

- Request method type: POST
- Request URL: `http://<HOST>:<PORT>/db/data/batch`
- Headers: Accept: application/json; charset=UTF-8 and Content-Type: application/json

Java API – batch insertion

Neo4j exposes a low-level Java API for bulk insertion of data, namely `BatchInserter` and for bulk indexing of data, namely `BatchInserterIndex`. These APIs are packaged with the Neo4j kernel available at <https://github.com/neo4j/neo4j/tree/2.1.5/community/kernel/src/main/java/org/neo4j/unsafe/batchinsert>. So it is available with both flavors (Enterprise and Community Edition) of Neo4j.

Let's take a closer look at each of these APIs.

BatchInserter

The intention behind the batch insertion API is to provide a low-level interface for directly interacting with or inserting data into Neo4j and avoiding all overheads such as transactions and other checks.

This API is useful when we have to load huge datasets (TBs), and the priority is only performance and nothing else.

At the same time, this API is limited in functionality and needs to be used with caution due to the following reasons:

- Request method type is POST
- It is not thread safe
- It does not support a transaction, so it is non-transactional
- It repopulates all existing indexes and new indexes on shutdown
- It is imperative to invoke shutdown successfully, otherwise database files will be corrupted

The batch insertion API should be used in a single thread or synchronization needs to be used so that concurrent access to `BatchInserter` object is restricted.

The following code creates two nodes in Neo4j database, using the batch insertion API:

```
import java.util.HashMap;
import java.util.Map;

import org.neo4j.graphdb.DynamicLabel;
import org.neo4j.graphdb.DynamicRelationshipType;
import org.neo4j.graphdb.Label;
import org.neo4j.graphdb.RelationshipType;
import org.neo4j.unsafe.batchinsert.BatchInserter;
import org.neo4j.unsafe.batchinsert.BatchInserters;

public class Neo4jBatchInserter {

    public void batchInsert() throws Exception {
        //Getting Object of BatchInserter.
        //Should be called only once for Batch of statements
        // "/graph.db" should be replaced with the path of your Neo4j
        Database.
        //Neo4j Database should be located at following path
        <$NEO4J_HOME>/data/graph.db
        BatchInserter inserter = BatchInserters.inserter("/graph.db");

        try{
            //Create a Label for a Node
            Label personLabel = DynamicLabel.label( "Cinestars" );
```

```

//Enabling Index for a Label and Property
inserter.createDeferredSchemaIndex(personLabel ).on( "name"
).create();
//Creating Properties and the Node
//"<" or generics are available with Java>=1.5
// But it is recommended to use Java>=1.7 with Neo4j API's
Map<String, Object> properties = new HashMap<>();
properties.put( "Name", "Cate Blanchett" );
long cateNode = inserter.createNode( properties, personLabel
);

//Creating Properties and the Node
properties.put( "Name", "Russell Crowe " );
long russellNode = inserter.createNode( properties,
personLabel );

//Creating Relationship
RelationshipType knows = DynamicRelationshipType.withName(
"KNOWS" );

// To set properties on the relationship, use a properties
map
// instead of null as the last parameter.
inserter.createRelationship(cateNode, russellNode, knows,
null );
} catch (Exception e) {
//Print Exception on Console and shutdown the Inserter
e.printStackTrace();
//Shutdown the Inserter, so that your database is not
corrupted.
inserter.shutdown();
}
//Should be called only once for Batch of statements
inserter.shutdown();
}

public static void main(String[] args) {
    try {
        new Neo4jBatchInserter().batchInsert();
    } catch (Exception e) {
        //Print all and any kind of Exception on Console.
        e.printStackTrace();
    }
}
}

```

Stop your Neo4j server (if it is running); compile the preceding code and execute it from the IDE itself. You should now see two nodes inserted by the batch insertion API.



In order to understand the preceding code, follow the comments provided before each line of the code. This technique is adopted for every code example.

GraphDatabaseService is also supported by the batch insertion API for reusability of insertion code, which must have been written using the normal Neo4j Java API and not BatchInserter.

Simply replace `build.newGraphDatabase()` in your older code base with `BatchInserters.batchDatabase("/graph.db")`; and now you can use the APIs of GraphDatabaseService. Do not forget to add `dbService.shutdown()` at the end of the code and you are done; however, this ease and simplicity comes with a cost and the cost is that there are few operations that are not supported by this service.

The following is the list of such operations:

- `Transaction.finish()` / `Transaction.close()` or `Transaction.success()`
- `Node.delete()` and `Node.traverse()`
- `Transaction.failure()` will generate a `NotInTransaction` exception
- `Relationship.delete()`
- Event handlers and indexes are not supported
- `getRelationshipTypes()`, `getAllNodes()`, and `getAllRelationships()` of GraphDatabaseService are also not supported

Batch indexing

As we have BatchInserter for inserting the data in batches, in the same manner Neo4j provides BatchInserterIndex for indexing the dataset during the insertion process. The following sample code demonstrates the use of BatchInserterIndex:

```
import java.util.Map;

import org.neo4j.helpers.collection.MapUtil;
import org.neo4j.index.lucene.unsafe.batchinsert.
LuceneBatchInserterIndexProvider;
```

```

import org.neo4j.unsafe.batchinsert.BatchInserter;
import org.neo4j.unsafe.batchinsert.BatchInserterIndex;
import org.neo4j.unsafe.batchinsert.BatchInserterIndexProvider;
import org.neo4j.unsafe.batchinsert.BatchInserters;

public class Neo4jBatchIndexer {

    public void batchIndexer() throws Exception {
        // "/graph.db" should be replaced with the path of your Neo4j
        Database.
        //Neo4j Database should be located at following path
        <$NEO4J_HOME>/data/graph.db
        BatchInserter inserter = BatchInserters.inserter( "graph.db"
        );

        //Creating Object of Index Providers
        BatchInserterIndexProvider indexProvider=new
        LuceneBatchInserterIndexProvider( inserter );
        Try{
            //Getting reference of Index
            BatchInserterIndex actors = indexProvider.nodeIndex(
            "cinestars", MapUtil.stringMap( "type", "exact" ) );
            //Enabling Caching on Nodes
            actors.setCacheCapacity( "name", 100000 );
            //Creating Node and setting Properties
            // "<>" or generics are available with Java>=1.5
            // But it is recommended to use Java>=1.7 with Neo4j API's
            Map<String, Object> properties = MapUtil.map( "name", "Keanu
            Reeves" );
            long node = inserter.createNode( properties );
            //Adding Node to Index
            actors.add( node, properties );
            //make the changes visible for reading, use Carefully,
            requires IO!
            actors.flush();
            // Shut down the index provider and inserter
        } catch (Exception e) {
            //Print Exception on Console and shutdown the Inserter and
            Indexer
            e.printStackTrace();
            //Shutdown the Index and Inserter, so that your database is
            not corrupted.
        }
    }
}

```

```
        indexProvider.shutdown();
        inserter.shutdown();
    }
    indexProvider.shutdown();
    inserter.shutdown();
}

public static void main(String[] args) {
    try {
        new Neo4jBatchIndexer().batchIndexer();
    } catch (Exception e) {
        //Print all and any kind of Exception on Console
        e.printStackTrace();
    }
}
```

Stop your Neo4j server (if it is running); compile the preceding code and execute it from IDE itself. The preceding code will create a Node that can then be indexed using BatchInserterIndex.

Understanding performance tuning and optimizations

Neo4j provides various optimizations and tuning parameters, which, if used thoughtfully, can provide significant differences in the behavior and throughput of the database and server. There is no standard or universally accepted configuration and developers need to consider the various **non-functional requirements (NFRs)** of the use case such as SLA, efficiency, effectiveness, failure management, performance / response time, and the environment in which they need to operate, before they start tuning the system.

Neo4j provides the capability of using the OS mapped memory, instead of the JVM heap for reading/writing. It is always recommended to provide enough OS mapped memory so that the best performance can be achieved, but in case memory is not enough, then Neo4j will do its best to keep frequently accessed data in memory and the rest on disk.

It is also recommended to configure OS with large disk caches, which will help in cases where we get cache misses in the node and relationship caches.

The following table shows the configuration for tuning the memory of the Neo4j database, which needs to be modified in `<$NEO4J_HOME>/conf/neo4j.properties` irrespective of the process, function, or operation being used or considered (such as reading/writing or bulk insertion).

Parameter	Default value	Description
<code>use_memory_mapped_buffers</code>	True (except windows)	Enables the mapping of operating system memory for storing the file buffer cache windows.
<code>neostore.nodestore.db.mapped_memory</code>	25M	This is the memory required to store the nodes.
<code>neostore.relationshipstore.db.mapped_memory</code>	50M	This is the memory required to store the relationships.
<code>neostore.propertystore.db.mapped_memory</code>	90M	This is the memory required to store the properties of the nodes and relationships.
<code>neostore.propertystore.db.strings.mapped_memory</code>	130M	This is the memory allocated for storing strings.
<code>neostore.propertystore.db.arrays.mapped_memory</code>	130M	This is the memory allocated for storing arrays.
<code>mapped_memory_page_size</code>	1048576	Total size for pages of mapped memory.
<code>label_block_size</code>	60 bytes	This is the block size for storing labels exceeding in-lined space in node record. This parameter is only considered at the time of store creation, otherwise it is ignored.
<code>array_block_size</code>	120 bytes	This is the block size for storing arrays. This parameter is only considered at the time of store creation, otherwise it is ignored.
<code>string_block_size</code>	120 bytes	This is the block size for storing strings. This parameter is only considered at the time of store creation, otherwise it is ignored.
<code>node_auto_indexing</code>	False	This controls the autoindexing feature for nodes. We should turn it ON.

Tuning JVM

Neo4j is written in Java and as with other Java-based applications, we need to configure JVM for optimal performance.

First and foremost, here are a few considerations for tuning the memory of any Java-based systems, in our case Neo4j:

- Monitoring GC activity and ensuring that it is not spending too much time in garbage collection.
- For most of the use cases, CMS garbage collector (`-XX+UseConcMarkSweepGC`), which collects objects concurrently and has low pause time, works well but don't stop yourself from considering Parallel GC (`-XX:+UseParallelGC`). Parallel GC performs minor collections in parallel, which can significantly reduce garbage collection overhead.
- Start your Java process with `-server` flag and based on your hardware capacity and use case requirements, configure max memory (`-Xmx`) and min memory (`-Xms`).
- Providing too much memory for JVM can result in performance degradation due to the pauses caused by long GC cycles; this can happen with too less memory as well. Monitor the memory of your system and then decide upon the appropriate values for `-Xms` and `-Xmx`. Think twice in case you are considering anything above 16 GB for your JVM. There can be long GC cycles resulting in performance degradation. There are a bunch of other parameters that can be referred from <http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html>.

LOAD CSV

LOAD CSV is a memory-intensive operation, which if optimized, can provide significant improvements in the overall data-loading process. For one of our use cases, we were able to reduce the overall data insertion time by 40 percent by using appropriate configuration and tuning parameters.

Apart from the common memory parameters, the following are a few considerations for the LOAD CSV process:

- Having simple LOAD CSV statements and multiple passes across them or multiple `.csv` files, consumes less memory than complex LOAD CSV statements
- The MERGE command should not be used for nodes and relationships in a single LOAD CSV command

- In a single `LOAD CSV` statement, either use `CREATE` nodes or `MERGE` nodes, but not both
- Neo4j shell provides optimum performance for batch imports
- Make sure that auto indexing is ON (`set node_auto_indexing=true`) and also define the columns that need to be indexed (`node_keys_indexable=id,name,type`) in `<$NEO4J_HOME>/conf/neo4j.properties`

Batch inserter / indexer

Similar to `LOAD CSV`, batch inserter / indexer is also a memory-intensive operation. So it is imperative for inserter/indexer to tune the memory and JVM parameters. The following are few considerations for tuning the memory and JVM of batch inserter / indexer:

- `BatchInserter#shutdown` should be called only at the end of each phase of the batch writing operation, as it will commit all changes and make it visible to the querying APIs.
- Avoid executing `BatchInserterIndex.flush()` too often during batch indexing as flushing will commit all changes and make it available for the querying APIs, which could result in performance degradation.
- Depending upon your memory sizes, have big phases for the write operations.
- Enable indexing and caching (`BatchInserterIndex.html#setCacheCapacity`) for common and widely used keys so that it can increase performance during lookups. Though it will slightly degrade the performance, there will be no major impact unless you care for each millisecond.
- Avoid or keep to a minimum read/lookup operations during batch insertion or indexing.

Summary

In this chapter, we have walked through the process of BI integration with Neo4j, bulk data loading strategies and processes. In the end, we have also touched upon the various aspects of performance tuning and optimization of our Neo4j database.

In the next chapter, we will discuss the strategy for modeling data in Neo4j, read-only Cypher queries for fetching data from the database, and using legacy indexing queries.

3

Pattern Matching in Neo4j

Data is not simple, it is complicated and evolving.

It is imperative to have an efficient structure to store complicated and evolving data and predominantly it should focus on "What to retrieve?", instead of "How to retrieve?". Having such an efficient structure will also help developers to focus on the domain model instead of getting lost in learning procedures/functions for accessing the database.

We could have used object-oriented principles and designed some fixed set of functions for accessing the data, but remember data in itself is evolving and not fixed, so it is cumbersome to add methods every time you add some new dimensions to your data; to make it even more complicated, sometimes you may not even know about this new data added to your system.

So, "How do we do that?" The answer is patterns and pattern matching.

Neo4j provides powerful, declarative and yet relatively simple graph query language, based on the principles of patterns and pattern matching, known as **Cypher**.

Cypher is a declarative graph query language, which not only provides an efficient mechanism for querying and updating graphs but also helps developers to focus on "What to query?" and hides all the complexity of "How to query?".

In this chapter, you will learn about data modeling in Neo4j and retrieving data using pattern matching in Cypher.

This chapter will cover the following topics:

- Agile data modeling with Neo4j
- Patterns and pattern matching

- Read-only Cypher queries
- Schema and legacy indexing queries
- Movie demo with GraphGists

Agile data modeling with Neo4j

Data modeling in Neo4j is evolving and flexible enough to adapt to changing business requirements. It captures the new data sources, entities, and their relationships as they naturally occur, allowing the database to easily adapt to the changes, which in turn results in an extremely agile development and provides quick responsiveness to changing business requirements.

Data modeling is a multistep process and involves the following steps:

1. Define requirements or goals in the form of questions that need to be executed and answered by the domain-specific model.
2. Once we have our goals ready, we can dig deep into the data and identify entities and their associations/relationships.
3. Now, as we have our graph structure ready, the next step is to form patterns from our initial questions/goals and execute them against the graph model.

This whole process is applied in an iterative and incremental manner, similar to what we do in agile, and has to be repeated again whenever we change our goals or add new goals/questions, which need to be answered by your graph model.

Let's see in detail how data is organized/structured and implemented in Neo4j to bring in the agility of graph models.

Based on the principles of graph data structure available at [http://en.wikipedia.org/wiki/Graph_\(abstract_data_type\)](http://en.wikipedia.org/wiki/Graph_(abstract_data_type)), Neo4j implements the property graph data model at storage level, which is efficient, flexible, adaptive, and capable of effectively storing/representing any kind of data in the form of nodes, properties, and relationships.

Neo4j not only implements the property graph model, but has also evolved the traditional model and added the feature of tagging nodes with labels, which is now referred to as the labeled property graph.

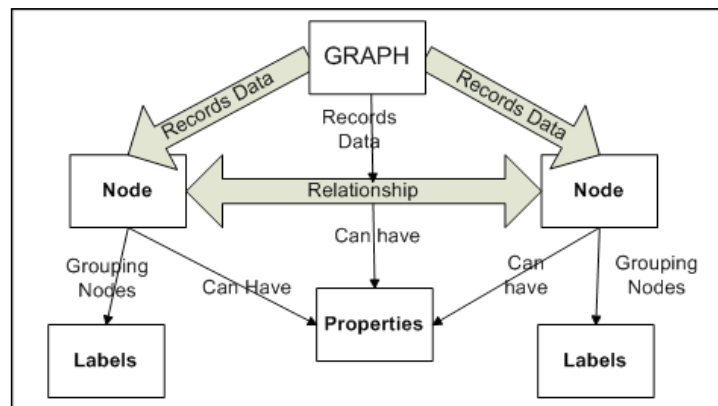
Essentially, in Neo4j, everything needs to be defined in either of the following forms:

- **Nodes:** A node is the fundamental unit of a graph, which can also be viewed as an entity, but based on a domain model, it can be something else too.

- **Relationships:** These defines the connection between two nodes. Relationships also have types, which further differentiate relationships from one another.
- **Properties:** Properties are viewed as attributes and do not have their own existence. They are related either to nodes or to relationships. Nodes and relationships can both have their own set of properties.
- **Labels:** Labels are used to construct a group of nodes into sets. Nodes that are labeled with the same label belong to the same set, which can be further used to create indexes for faster retrieval, mark temporary states of certain nodes, and there could be many more, based on the domain model.

Let's see how all of these four forms are related to each other and represented within Neo4j.

A graph essentially consists of nodes, which can also have properties. Nodes are linked to other nodes. The link between two nodes is known as a relationship, which also can have properties. Nodes can also have labels, which are used for grouping the nodes.

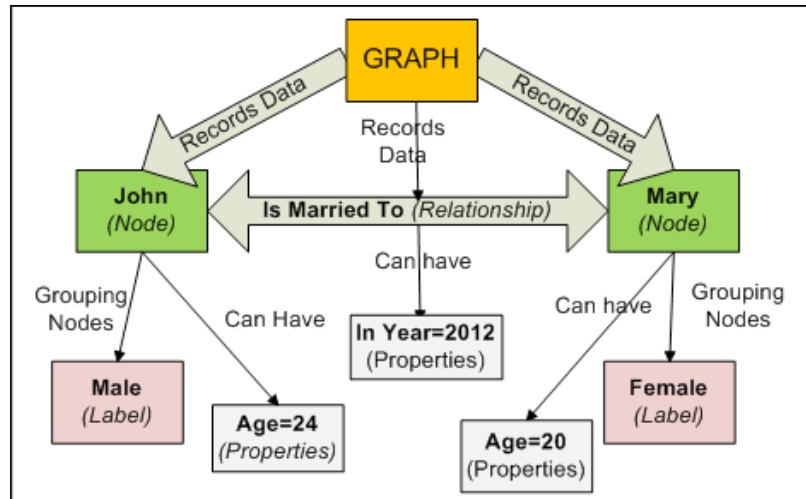


Let's take up a use case to understand data modeling in Neo4j. John is a male and his age is 24. He is married to a female named Mary whose age is 20. John and Mary got married in 2012.

Now, let's develop the data model for the preceding use case in Neo4j:

- **John** and **Mary** are two different nodes.
- Marriage is the relationship between John and Mary.
- Age of **John**, age of **Mary**, and the year of their marriage become the properties.

- **Male** and **Female** become the labels.



Easy, simple, flexible, and natural... isn't it?

The data structure in Neo4j is adaptive and effectively can model everything that is not fixed and evolves over a period of time.

The next step in data modeling is fetching the data from the data model, which is done through traversals. Traversals are another important aspect of graphs, where you need to follow paths within the graph starting from a given node and then following its relationships with other nodes. Neo4j provides two kinds of traversals: breadth first available at http://en.wikipedia.org/wiki/Breadth-first_search and depth first available at http://en.wikipedia.org/wiki/Depth-first_search. We will discuss traversals in detail in *Chapter 5, Neo4j from Java*.

If you are from the RDBMS world, then you must now be wondering, "What about the schema?" and you will be surprised to know that Neo4j is a schemaless or schema-optional graph database. We do not have to define the schema unless we are at a stage where we want to provide some structure to our data for performance gains. Once performance becomes a focus area, then you can define a schema and create indexes/constraints/rules over data. We will read more about indexing and its capabilities in the upcoming sections.

Unlike the traditional models where we freeze requirements and then draw our models, Neo4j embraces data modeling in an agile way so that it can be evolved over a period of time and is highly responsive to the dynamic and changing business requirements.

Patterns and pattern matching

Patterns and pattern matching is one of the important components for extracting data from the graph databases. Let's first talk about the definition of patterns and pattern matching and then we will see how these are implemented in the context of Neo4j:

- **Pattern:** In simple language and to make it even simpler to understand, a pattern is an occurrence of sequence of characters/numbers, word or group of words, literals that need to be found in a given dataset.
- **Pattern matching:** The process or algorithm used for finding or matching a pattern or sequence of patterns against a given dataset or data structure is called pattern matching. In other words, pattern matching consists of specifying patterns to which some data should conform and then checking to see if it does and deconstructing the data according to those patterns. Pattern matching should not be confused with pattern recognition, in which the match usually has to be exact.

There are well-known graph query languages which implement pattern matching and can be used to query graphs, but they are tedious and involve a lot of work in maintenance/modifications and enhancements. A few of the well-known graph query languages include:

- **SPARQL:** <http://en.wikipedia.org/wiki/SPARQL>
- **GREMLIN:** <http://gremlin.tinkerpop.com/>
- **Metaweb Query Language (MQL):** <https://developers.google.com/freebase/v1/mql-overview>

They also failed to meet one or more of the primary goals of the query language for the Neo4j database which are as follows:

- Declarative
- ASCII art pattern
- External DSL
- SQL familiarity
- Closures

Considering all the preceding goals, Neo4j implemented the concepts of patterns and pattern matching and provided a new declarative graph query language, Cypher, as a query language for the Neo4j graph database. Cypher is specifically designed to be a humane query language, which is focused on making things simpler for developers. Another benefit of being a declarative language is that it focuses on "What to retrieve?" and not "How to retrieve?", which is in contrast to the other imperative languages such as Java and Gremlin.

Patterns and pattern matching are the core components of Cypher; so, being effective with Cypher requires a good understanding of the patterns and their implementation in Cypher.

Patterns are used to describe the shape of the data and also provide the path from where it should start searching for occurrences of the provided pattern.

Let's see how the patterns are defined for nodes, properties, labels, and relationships followed by examples.

Pattern for nodes

Pattern for nodes is one of the most basic and simplest one. Pattern for nodes are defined using a pair of parentheses and a name is given between the parentheses as follows:

```
(n)
```

Here, a pair of parentheses define a single node and *n* is the identifier of that node.

Pattern for more than one node

More than one node is defined in the same way as we do for a single node, but when two or more nodes are used, then we also need to define the relationship between these two nodes. Consider the following code as an example:

```
(n) --> (n1)
```

In the preceding example, we have defined two nodes, *n* and *n1*, and both the nodes are connected to each other through a directed arrow `-->` which is known as the directed relationship between *n* and *n1*.

Pattern for paths

A series of connected nodes is known as a **path**. All the following examples define the path in a particular pattern:

- `(n) - [:REL_TYPE] -> (n1)`
- `(n) --> () <-- (n1)`
- `(n) --> (x) <-- (n1)`

We can also assign a path identifier and store the pattern in a variable as follows:

```
Var = (n) - [:REL_TYPE] -> (n1)
```

Pattern for labels

As we define the pattern for nodes, we can also define the pattern for labels. Labels are the identifiers for a node, which can also be used to group the related nodes.

A node can have one or more labels:

```
(n:Male) --> (b) or (a: Artist:Male) --> (b)
```

Pattern for relationships

Relationships can be of two types: unidirectional and bidirectional.

- **Unidirectional:** This is defined using an arrow sign, with an arrow head defining the direction of the relationship as shown in the following example:

```
(n) --> (n1)
```

This example defines a relationship between `n` and `n1`, which starts from node `n` and ends at node `n1`.

- **Bidirectional:** This is defined using an arrow sign with NO arrow head as shown in the following example:

```
(n) -- (n1)
```

This example defines a relationship between `n` and `n1`, which can flow from either end.

As with nodes, relationships can also have names or identifiers. Names or identifiers for relationships are defined in a pair of square brackets as follows:

- `(n) - [r] -> (n1)`
- `(n) - [r] - (n1)`

We can also define one or more than one relationship in a pattern:

- `(n) - [r:REL_TYPE] -> (n1)`
- `(n) - [r:REL_TYPE | REL_TYPE] - (n1)`

As we do in nodes, we can also omit the names or identifiers in relationships:

- `(n) - [:REL_TYPE] -> (n1)`
- `(n) - [] - (n1)`

Pattern for properties

Properties are defined in nodes and relationships, which in turn help in developing richer models.

Properties are defined in a key/value format separated with a comma, where the key is the string and the value can either be a primitive data type such as Boolean, int, double, long, char, string, and so on, or an array of primitive data type such as boolean[], int[], double[], long[], char[], string[], except NULL, which is not a valid value of a property. One or more properties can be defined using curly braces either within the node or within the relationship. The following examples show the format for defining properties in a node and in a relationship:

Structure / syntax for defining the property of a node.

```
(n:Artist {Name:"John"}) OR (n:Artist {Name:"John", Age: 24})
```

Structure / syntax for defining the property of a relationship.

```
[r:FRIEND {Since: 2004, LastVisited:"Jan-2015"}]
```

Expressions

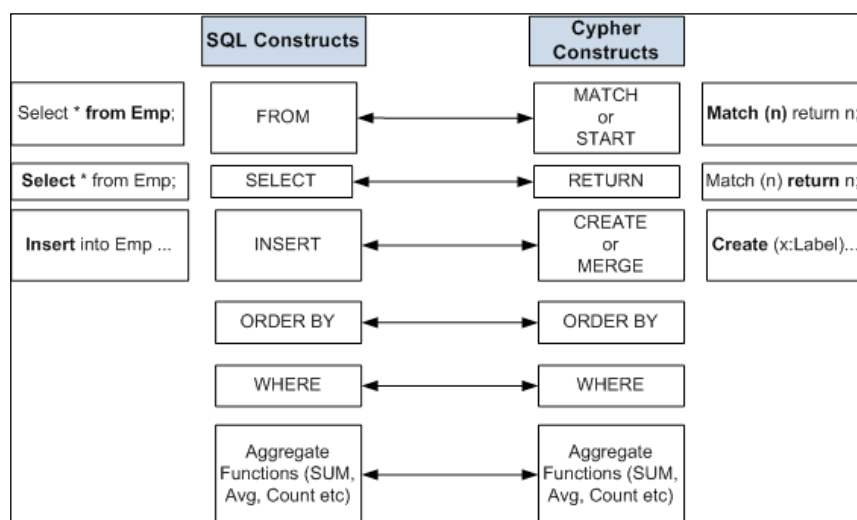
Cypher supports a variety of expressions. The following is the list of such expressions that can be used within Cypher queries:

- Decimal, hexadecimal, or octal integer literals

- String literals "Hello" or 'Hey'
- Boolean literals true/false or TRUE/FALSE.
- Collection of expressions: ["x", "z"] or [4,5,6] or ["a", 2, n.property, {param}], []
- Function call and aggregate functions such as count (*), avg (x), sum (y)
- Mathematical, comparison, Boolean, string, and collection operators
- CASE expressions

All the preceding defined patterns are used in conjunction with the CRUD operations within Cypher queries. Cypher pattern matching borrows the approach to expressions from SPARQL and a few of the other collection semantics have been borrowed from languages such as Haskell and Python. Cypher also borrows much of its structure from SQL itself and made it easy to understand for developers who are already familiar with SQL. It also helped them to achieve one of their critical goals of SQL familiarity.

Let's see the similarity between the structure of Cypher and SQL:



Usage of pattern

Let's see how queries are constructed and the way the patterns are used within the Cypher queries:

1. Open your Unix/ Linux shell and open Neo4j shell by executing
`<$NEO4J_HOME>/bin/neo4j-shell.`

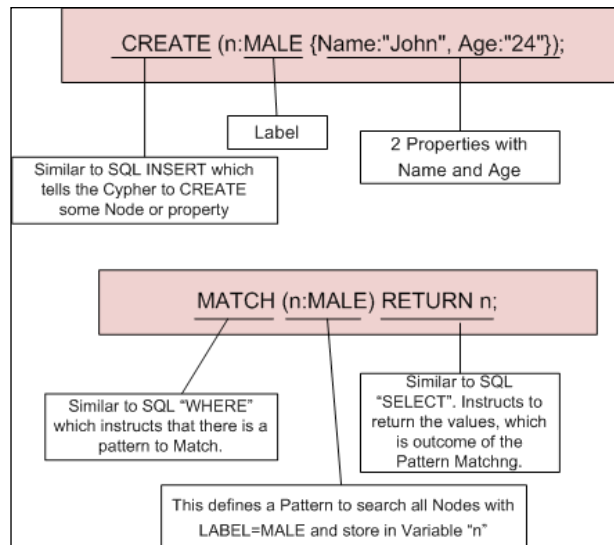
2. Execute the following commands on your Neo4j shell:

```
CREATE (n:MALE {Name:"John", Age:"24"});  
MATCH (n:MALE) RETURN n;
```

You will see the following output on your screen:

```
neo4j-sh (?)$ CREATE (n:MALE {Name:"John", Age:"24"});  
+-----+  
| No data returned. |  
+-----+  
Nodes created: 1  
Properties set: 2  
Labels added: 1  
12 ms  
neo4j-sh (?)$ MATCH (n:MALE) RETURN n;  
+-----+  
| n |  
+-----+  
| Node[95]{Name:"John",Age:"24"} |  
+-----+  
1 row  
25 ms  
neo4j-sh (?)$
```

In the previous step, we created a node with a label and properties, and then in the next statement we searched for the same node. Refer to the following illustration which explains the different parts of preceding Cypher query and also similarity with SQL:



In this section, we discussed the basic usage of patterns/pattern matching and implementation of these in Cypher, including its syntax. In the upcoming sections/chapter, we will get into the nitty-gritties of Cypher queries and will use various illustrations to show the usage of Cypher constructs.

Read-only Cypher queries

In the previous section, we have discussed the heart of Cypher, that is, patterns and pattern matching, and in this section, we will discuss one of the most important aspects of Neo4j, that is, read-only Cypher queries.

Read-only Cypher queries are not only the core component of Cypher but also help us in exploring and leveraging various patterns and pattern matching constructs. It either begins with `MATCH`, `OPTIONAL MATCH`, or `START`, which can be used in conjunction with the `WHERE` clause and further followed by `WITH` and ends with `RETURN`. Constructs such as `ORDER BY`, `SKIP`, and `LIMIT` can also be used with `WITH` and `RETURN`.

We will discuss in detail about read-only constructs, but before that, let's create a sample dataset and then we will discuss constructs/syntax of read-only Cypher queries with illustration.

Creating a sample dataset – movie dataset

Let's perform the following steps to clean up our Neo4j database and insert some data which will help us in exploring various constructs of Cypher queries:

1. Open your Command Prompt or Linux shell and open the Neo4j shell by typing `<$NEO4J_HOME>/bin/neo4j-shell`.
2. Execute the following commands on your Neo4j shell for cleaning all the previous data:

```
//Delete all relationships between Nodes
MATCH ()-[r]-() delete r;
//Delete all Nodes
MATCH (n) delete n;
```

3. Now we will create a sample dataset, which will contain movies, artists, directors, and their associations. Execute the following set of Cypher queries in your Neo4j shell to create the list of movies:

```
CREATE (:Movie {Title : 'Rocky', Year : '1976'});
CREATE (:Movie {Title : 'Rocky II', Year : '1979'});
CREATE (:Movie {Title : 'Rocky III', Year : '1982'});
CREATE (:Movie {Title : 'Rocky IV', Year : '1985'});
CREATE (:Movie {Title : 'Rocky V', Year : '1990'});
CREATE (:Movie {Title : 'The Expendables', Year : '2010'});
CREATE (:Movie {Title : 'The Expendables II', Year : '2012'});
```

```
CREATE (:Movie {Title : 'The Karate Kid', Year : '1984'});  
CREATE (:Movie {Title : 'The Karate Kid II', Year : '1986'});
```

4. Execute the following set of Cypher queries in your Neo4j shell to create the list of artists:

```
CREATE (:Artist {Name : 'Sylvester Stallone', WorkedAs : ["Actor",  
"Director"]});  
CREATE (:Artist {Name : 'John G. Avildsen', WorkedAs :  
["Director"]});  
CREATE (:Artist {Name : 'Ralph Macchio', WorkedAs : ["Actor"]});  
CREATE (:Artist {Name : 'Simon West', WorkedAs : ["Director"]});
```

5. Execute the following set of cypher queries in your Neo4j shell to create the relationships between artists and movies:

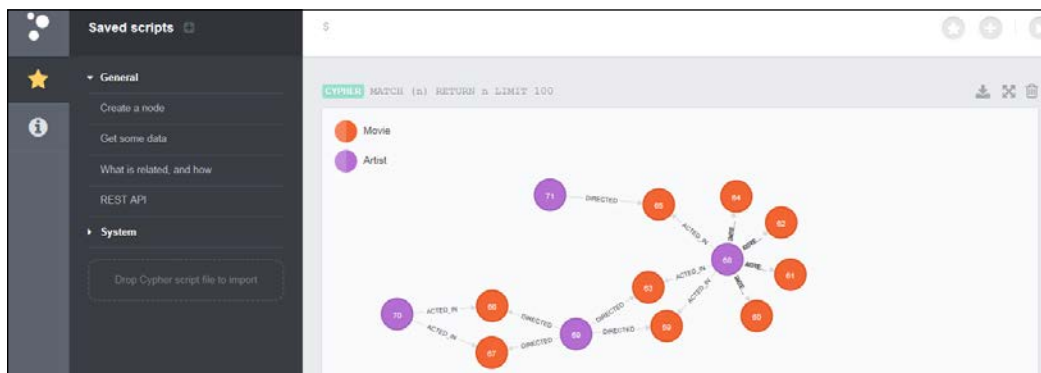
```
Match (artist:Artist {Name : "Sylvester Stallone"}), (movie:Movie  
{Title: "Rocky"}) CREATE (artist)-[:ACTED_IN {Role : "Rocky  
Balboa"}]->(movie);  
Match (artist:Artist {Name : "Sylvester Stallone"}), (movie:Movie  
{Title: "Rocky II"}) CREATE (artist)-[:ACTED_IN {Role : "Rocky  
Balboa"}]->(movie);  
Match (artist:Artist {Name : "Sylvester Stallone"}), (movie:Movie  
{Title: "Rocky III"}) CREATE (artist)-[:ACTED_IN {Role : "Rocky  
Balboa"}]->(movie);  
Match (artist:Artist {Name : "Sylvester Stallone"}), (movie:Movie  
{Title: "Rocky IV"}) CREATE (artist)-[:ACTED_IN {Role : "Rocky  
Balboa"}]->(movie);  
Match (artist:Artist {Name : "Sylvester Stallone"}), (movie:Movie  
{Title: "Rocky V"}) CREATE (artist)-[:ACTED_IN {Role : "Rocky  
Balboa"}]->(movie);  
Match (artist:Artist {Name : "Sylvester Stallone"}), (movie:Movie  
{Title: "The Expendables"}) CREATE (artist)-[:ACTED_IN {Role :  
"Barney Ross"}]->(movie);  
Match (artist:Artist {Name : "Sylvester Stallone"}), (movie:Movie  
{Title: "The Expendables II"}) CREATE (artist)-[:ACTED_IN {Role :  
"Barney Ross"}]->(movie);  
Match (artist:Artist {Name : "Sylvester Stallone"}), (movie:Movie  
{Title: "Rocky II"}) CREATE (artist)-[:DIRECTED]->(movie);  
Match (artist:Artist {Name : "Sylvester Stallone"}), (movie:Movie  
{Title: "Rocky III"}) CREATE (artist)-[:DIRECTED]->(movie);  
Match (artist:Artist {Name : "Sylvester Stallone"}), (movie:Movie  
{Title: "Rocky IV"}) CREATE (artist)-[:DIRECTED]->(movie);
```

```

Match (artist:Artist {Name : "Sylvester Stallone"}), (movie:Movie
{Title: "The Expendables"}) CREATE (artist)-[:DIRECTED]->(movie);
Match (artist:Artist {Name : "John G. Avildsen"}), (movie:Movie
{Title: "Rocky"}) CREATE (artist)-[:DIRECTED]->(movie);
Match (artist:Artist {Name : "John G. Avildsen"}), (movie:Movie
{Title: "Rocky V"}) CREATE (artist)-[:DIRECTED]->(movie);
Match (artist:Artist {Name : "John G. Avildsen"}), (movie:Movie
{Title: "The Karate Kid"}) CREATE (artist)-[:DIRECTED]->(movie);
Match (artist:Artist {Name : "John G. Avildsen"}), (movie:Movie
{Title: "The Karate Kid II"}) CREATE (artist)-[:DIRECTED]-
>(movie);
Match (artist:Artist {Name : "Ralph Macchio"}), (movie:Movie
{Title: "The Karate Kid"}) CREATE (artist)-[:ACTED_IN
{Role:"Daniel LaRusso"}]->(movie);
Match (artist:Artist {Name : "Ralph Macchio"}), (movie:Movie
{Title: "The Karate Kid II"}) CREATE (artist)-[:ACTED_IN
{Role:"Daniel LaRusso"}]->(movie);
Match (artist:Artist {Name : "Simon West"}), (movie:Movie {Title:
"The Expendables II"}) CREATE (artist)-[:DIRECTED]->(movie);

```

- Next, browse your data through the Neo4j browser. Click on **Get some data** from the left navigation pane and then execute the query by clicking on the right arrow sign that will appear on the extreme right corner just below the browser navigation bar, and it should look something like this:



Now, let's understand the different pieces of read-only queries and execute those against our movie dataset.

Working with the MATCH clause

MATCH is the most important clause used to fetch data from the database. It accepts a pattern, which defines "What to search?" and "From where to search?". If the latter is not provided, then Cypher will scan the whole tree and use indexes (if defined) in order to make searching faster and performance more efficient.

Working with nodes

Let's start asking questions from our movie dataset and then form Cypher queries, execute them on `<$NEO4J_HOME>/bin/neo4j-shell` against the movie dataset and get the results that will produce answers to our questions:

- How do we get all nodes and their properties?
 - **Answer:** `MATCH (n) RETURN n;`
 - **Explanation:** We are instructing Cypher to scan the complete database and capture all nodes in a variable `n` and then return the results, which in our case will be printed on our Neo4j shell.
- How do we get nodes with specific properties or labels?
 - **Answer:** Match with label, `MATCH (n:Artist) RETURN n;`
or `MATCH (n:Movies) RETURN n;`
 - **Explanation:** We are instructing Cypher to scan the complete database and capture all nodes, which contain the value of label as `Artist` or `Movies`.
 - **Answer:** Match with a specific property `MATCH (n:Artist {WorkedAs: ["Actor"]}) RETURN n;`
 - **Explanation:** We are instructing Cypher to scan the complete database and capture all nodes that contain the value of label as `Artist` and the value of property `WorkedAs` is `["Actor"]`. Since we have defined the `WorkedAs` collection, we need to use square brackets, but in all other cases, we should not use square brackets.

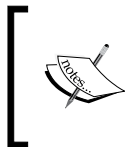


We can also return specific columns (similar to SQL). For example, the preceding statement can also be formed as `MATCH (n:Artist {WorkedAs:["Actor"]}) RETURN n.name as Name;`

Working with relationships

Let's understand the process of defining relationships in the form of Cypher queries in the same way as you did in the previous section while working with nodes:

- How do we get nodes that are associated or have relationships with other nodes?
 - **Answer:** `MATCH (n) - [r] - (n1) RETURN n, r, n1;`
 - **Explanation:** We are instructing Cypher to scan the complete database and capture all nodes, their relationships, and nodes with which they have relationships in variables `n`, `r`, and `n1` and then further return/print the results on your Neo4j shell. Also, in the preceding query, we have used `-` and not `->` as we do not care about the direction of relationships that we retrieve.
- How do we get nodes, their associated properties that have some specific type of relationship, or the specific property of a relationship?
 - **Answer:** `MATCH (n) - [r:ACTED_IN {Role : "Rocky Balboa"}] -> (n1) RETURN n, r, n1;`
 - **Explanation:** We are instructing Cypher to scan the complete database and capture all nodes, their relationships, and nodes, which have a relationship as `ACTED_IN` and with the property of `Role` as `Rocky Balboa`. Also, in the preceding query, we do care about the direction (incoming/outgoing) of a relationship, so we are using `->`.



For matching multiple relations replace `[r:ACTED_IN]` with `[r:ACTED_IN | DIRECTED]` and use single quotes or escape characters wherever there are special characters in the name of relationships.

- How do we get a coartist?
 - **Answer:** `MATCH (n {Name : "Sylvester Stallone"})-[r]->(x)<-[r1]-(n1) return n.Name as Artist,type(r),x.Title as Movie, type(r1), n1.Name as Artist2;`
 - **Explanation:** We are trying to find out all artists that are related to Sylvester Stallone in some manner or the other. Once you run the preceding query, you will see something like the following image, which should be self-explanatory. Also, see the usage of `as` and `type`. `as` is similar to the SQL construct and is used to define a meaningful name to the column presenting the results, and `type` is a special keyword that gives the type of relationship between two nodes.

```
neo4j-sh (?)$ MATCH (n {Name : "Sylvester Stallone"})-[r]->(x)<-[r1]-(n1) return n.Name as Artist,type(r),x.Title as Movie, type(r1), n1.Name as Artist2;
```

Artist	type(r)	Movie	type(r1)	Artist2
"Sylvester Stallone"	"ACTED_IN"	"The Expendables II"	"DIRECTED"	"Simon West"
"Sylvester Stallone"	"ACTED_IN"	"The Expendables"	"DIRECTED"	"Sylvester Stallone"
"Sylvester Stallone"	"ACTED_IN"	"Rocky V"	"DIRECTED"	"John G. Avildsen"
"Sylvester Stallone"	"ACTED_IN"	"Rocky IV"	"DIRECTED"	"Sylvester Stallone"
"Sylvester Stallone"	"ACTED_IN"	"Rocky III"	"DIRECTED"	"Sylvester Stallone"
"Sylvester Stallone"	"ACTED_IN"	"Rocky II"	"DIRECTED"	"Sylvester Stallone"
"Sylvester Stallone"	"ACTED_IN"	"Rocky"	"DIRECTED"	"John G. Avildsen"
"Sylvester Stallone"	"DIRECTED"	"The Expendables"	"ACTED_IN"	"Sylvester Stallone"
"Sylvester Stallone"	"DIRECTED"	"Rocky IV"	"ACTED_IN"	"Sylvester Stallone"
"Sylvester Stallone"	"DIRECTED"	"Rocky III"	"ACTED_IN"	"Sylvester Stallone"
"Sylvester Stallone"	"DIRECTED"	"Rocky II"	"ACTED_IN"	"Sylvester Stallone"

```
11 rows
3313 ms
neo4j-sh (?)$
```

- How do we get the path and number of hops between two nodes?
 - **Answer:** `MATCH p = (:Movie{Title:"The Karate Kid"})-[:DIRECTED*0..4]-(:Movie{Title:"Rocky V"}) return p;`
 - **Explanation:** Paths are the distance between two nodes. In the preceding statement, we are trying to find out all paths between two nodes, which are between 0 (minimum) and 4 (maximum) hops away from each other and are only connected through the relationship `DIRECTED`. You can also find the path, originating only from a particular node and re-write your query as `MATCH p = (:Movie{Title:"The Karate Kid"})-[:DIRECTED*0..4]-() return p;`

Working with the OPTIONAL MATCH clause

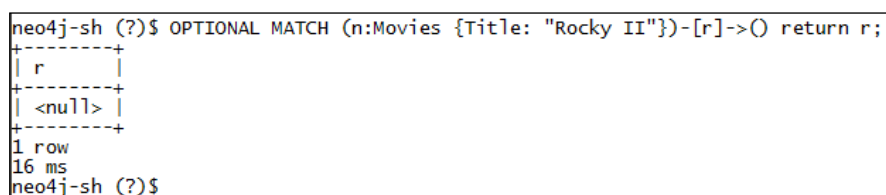
The `OPTIONAL MATCH` clause is similar to `MATCH` but the only difference is that `OPTIONAL MATCH` is "not strict" while evaluating the pattern against a given dataset.

In `MATCH`, it is all or nothing but `OPTIONAL MATCH` uses `NULLs` for missing parts of the pattern for those scenarios where no match is found. It is similar to the functionality provided by the outer joins in SQL.

Let's consider that in the movie dataset, we want to get all outgoing relationships from the movie `Rocky II`. The following is our Cypher query:

```
OPTIONAL MATCH (n:Movies {Title: "Rocky II"})- [r] -> ()
return r;
```

The result is shown in the following screenshot:



```
neo4j-sh (?)$ OPTIONAL MATCH (n:Movies {Title: "Rocky II"})- [r] -> () return r;
+-----+
| r      |
+-----+
| <null> |
+-----+
1 row
16 ms
neo4j-sh (?)$
```

The preceding query returned 1 row and the `<null>` values because there were no matching records for the given pattern.

Working with the `START` clause

`START` is used in cases where we have multiple starting points and you need to specifically provide a starting point for evaluating your pattern. Starting points are introduced by legacy index lookups or by the ID. However, trying to use a legacy index that doesn't exist will generate an exception.

`START` is only used when we are dealing with legacy indexes (refer to the next section for more information on legacy indexes), otherwise Cypher is intelligent enough to find out the starting point from your pattern by looking at nodes or labels and its predicates.

Starting Neo4j 2.0 `START` is not recommended and is only used for backward compatibility with the prior versions of Neo4j. It is advised to not use `START` for all versions of Neo4j 2.0+.

Let's consider an example where we need to get details of a node with ID 10. Your query would be:

```
start n=node(10) return n;
```

The results of the preceding query may vary from database to database as the IDs are internal to Neo4j and are re-used whenever a node or relationship is deleted.

Working with the WHERE clause

WHERE, as the name suggests, is used to filter the given set of results.

Let's consider our movie dataset and take an example where we need to fetch all relationships that have some value of attribute `Role`, so our Cypher query would be:

```
MATCH (n)-[r]-() where r.Role is NOT NULL return n,r;
```

Another example would be where we want a count of relationships for a particular node, which does have a value of attribute `Role`:

```
MATCH (n)-[r]-() where r.Role is NOT NULL return n,count(r);
```

We could also consider an example where we want only those nodes that have `count (r)` is greater than 1:

```
MATCH (n)-[r]-() WITH n, count(r) AS countRel  
WHERE countRel>1 return n,countRel;
```

See the usage of the `WITH` clause in the preceding statement, which is used to introduce the aggregate function `count (r)` for counting the relationships for all nodes within our database.

`WITH` is like the event horizon—it is a barrier between a plan and the finished execution of that plan. Apart from the aggregate functions, `WITH` can also be used to chain together two reading query parts. For example, let's assume that we need to get the count of `Movie` for all `Artist` that are related to `Movie` by `ACTED_IN` relationship, but we should consider only for those `Artist` that are linked to movies by the `DIRECTED` relationship:

```
MATCH (m:Movie)<- [r:ACTED_IN] - (a:Artist) - [d:DIRECTED] -> (m)  
WITH a  
MATCH (a)-[:ACTED_IN]->(m1:Movie)  
WITH a,count( DISTINCT m1) as TotalMovies  
return a.Name as Artist,TotalMovies;
```

In the preceding query, we have chained two reading query parts: the first one does the filtering and gets the `Artist` that are related to `Movie` by `DIRECTED` and `ACTED_IN` relationships, and the second one counts the movies of the selected `Artist` that are related to `Movie` by the `ACTED_IN` relationship. We have also used `DISTINCT` so that, while aggregating, we consider only unique records.

There is no limit and depending on the available memory you can chain together as many query parts as you like using `WITH`.

Working with the RETURN clause

`RETURN` is used to return the results of Cypher statements to the program, which is requesting Neo4j to execute queries or statements. It is similar to SQL `SELECT`. But in Cypher it needs to be the last statement of the query.

Except the `CREATE` statement, every other Cypher statement should end with `RETURN`, otherwise it is treated as an invalid statement and the compiler will throw an error.

There are clauses such as `ORDER BY`, `SKIP`, and `LIMIT`, which can be used with `RETURN` for getting the results in a desired format.

Let's take an example where we need to get all nodes marked with the label `Artist` in a particular order (ascending or descending) and also retrieve only the top two rows:

```
MATCH (n:Artist) return n order by n.Name desc LIMIT 2;
```

Remove `desc` from the preceding statement to get the results in ascending order. You can also use `SKIP` with `RETURN` for ignoring a particular number of specified rows starting from the top of the results.

Let's rewrite our previous query and use to `SKIP 1` record:

```
MATCH (n:Artist) return n order by n.Name desc SKIP 1 LIMIT 2;
```

The results of the preceding query will skip a record from the top; then it will show the second and third rows of the result set, as shown in the following screenshot:

```
neo4j-sh (?)$ match (n:Artist) return n order by n.Name desc LIMIT 2;
+-----+
| n |
+-----+
| Node[9]{Name:"Sylvester Stallone",WorkedAs:["Actor","Director"]} |
| Node[12]{Name:"Simon West",WorkedAs:["Director"]} |
+-----+
2 rows
41 ms
neo4j-sh (?)$ MATCH (n:Artist) return n order by n.Name desc SKIP 1 LIMIT 2;
+-----+
| n |
+-----+
| Node[12]{Name:"Simon West",WorkedAs:["Director"]} |
| Node[11]{Name:"Ralph Macchio",WorkedAs:["Actor"]} |
+-----+
2 rows
25 ms
neo4j-sh (?)$
```

Schema and legacy indexing

Neo4j 2.0+ supports two different ways of indexing the data; legacy indexing and schema level indexing. Let's discuss both the indexing techniques.

Using legacy indexes

Legacy indexes are also known as manual indexes that are managed at the application level via `IndexManager` API available at <http://neo4j.com/docs/2.1.5/javadocs/org/neo4j/graphdb/index/package-summary.html>. Legacy indexes are created by a unique user-defined name and can be created either on nodes or relationships.

Searching or retrieving the data from legacy indexes is also a manual process and can be done in two ways:

- `org.neo4j.graphdb.index.Index.get`: This will return the exact matches for a given key-value pair.
- `org.neo4j.graphdb.index.Index.query`: Exposes querying capabilities from the backend indexing APIs used for indexing the data. Neo4j leverages Lucene (<http://lucene.apache.org/>) and Lucene query syntax can be directly used for querying the database.

We will discuss in detail the various kinds of searches exposed by legacy indexes in the next chapter.

We can also use the autoindexing mechanism for enabling legacy indexing by enabling properties such as `node_auto_indexing`, `node_keys_indexable`, `relationship_auto_indexing`, and `relationship_keys_indexable` in `neo4j.properties`.

Maintaining or querying with legacy or manual indexes is a complicated process. Everything needs to be managed by the application itself, which is against the principles of databases where indexes should be automatically populated/updated and used once it is defined by the users. So, to improve the efficiency and ease of use, Neo4j 2.0 introduced a new mechanism for indexing and deprecated the old process of manual indexing, though it is still supported but only for backward compatibility for Neo4j applications which are developed and running on earlier versions of Neo4j (\leq Neo4j 1.9.9).

Using schema-level indexing

Neo4j 2.0 introduced indexes and constraints on labels which together define the schema for Neo4j. Cypher `CREATE` constructs can be used to define constraints on the attributes and indexes on labels. Creation of schema or defining schema is optional and can be deferred until you want to provide structure to your data and focus is on performance.

Defining schema/index improves the speed of searching and is leveraged implicitly during execution of the queries. Indexes are eventually available, that is, they are being populated asynchronously without impacting the user operations. The decision of whether indexes need to be used in a query will depend upon the state of the index. If an index is in the available state, then it will be used, otherwise not. All these complexities of indexes are hidden from developers and usually there is no need to specifically provide any hints in our queries. There are APIs that can tell you the status and availability of indexes and constraints in your database.

Further in this section, we will leverage our movie dataset that we created in the previous section and will create schema (indexes/constraints) using Cypher.

Creating schema with Cypher

Let's create some indexes and constraints using Cypher on our movie dataset:

- Create the index on the property `Name` for all the nodes that have the Label `Artist`:
`CREATE index on :Artist(Name);`
- Drop index which was created on `:Artist (Name)`:
`DROP index on :Artist(Name);`
- Create unique Constraint on `:Movies (Title)`:
`CREATE Constraint on (movies:Movie) ASSERT movies.Title IS UNIQUE;`
- Drop unique Constraint on `:Movies (Title)`:
`Drop Constraint on (movies:Movie) ASSERT movies.Title IS UNIQUE;`
- Query the status of all indexes and constraints within your database:
`Schema`

- Query the status of index `:Artist (Name)`

```
schema -l :Artist -p Name
```

In the preceding statement, `-l` will list all indexes on a given label. `-p` will list indexes on a given property and `-v` is to print errors for the indexes that are in the failed state.

- Forcing queries to use indexes:

```
MATCH (n:Artist)
USING INDEX n:Artist (Name)
WHERE n.Name = 'Sylvester Stallone' RETURN n;
```



Creating constraints also creates a unique index on the same column, so there is no need to explicitly create an index on unique columns.

Movie Demo with GraphGists

Gist (<https://help.github.com/articles/about-gists/>) is one of the wonderful features of GitHub, which provides a quick-and-easy way of sharing code, text, and files. It provides support for rendering a huge number of programming languages and facilitates rich syntax highlighting.

Neo4j leveraged the same thought process developed by Neo4j Gist <http://gist.neo4j.org/> for sharing and rendering the graphs.

In this section, we will re-use the same movie dataset and will see how Neo4j Gists can be leveraged to share our data with the community.

Let's follow the steps to create our Movie Demo and share it with the other users on Neo4j Gist:

1. Create a GitHub user ID and password and log in to <https://gist.github.com/>.
2. Create **ASCIIDoc**, then enter the text given in the following screenshot:

```

1  = Movie Demo
2  Sumit Gupta <sumit1001@yahoo.com>
3  v1.0, 16-Nov-2014
4  :neo4j-version: 2.1.5
5  :author: Sumit Gupta
6
7  == Domain
8  This is text kept for Domain text
9  == Setup
10
11 The sample data set which creates Actors, Movies and their relationships.
12
13 //hide
14 //setup
15 //output
16 [source,cypher]
17 ----
18 CREATE (movies1:Movies {Title : 'Rocky', Year : '1976'})
19 CREATE (movies2:Movies {Title : 'Rocky II', Year : '1979'})

```

Buttons: Add file, Create secret Gist, Create public Gist

- Next, just below the **[source, cypher]** statement, drop in your Cypher queries for creating the movie dataset. Your Cypher queries should be in between `----`, as shown in row 17 in the preceding screenshot. You can use the same queries that we used in the *Read-only Cypher queries* section to create our movie database.
- Now we will define a couple of read-only Cypher queries to fetch the data and show the results in the graph and table format, and click on **Create public Gist** and save our Movie Demo script.

```

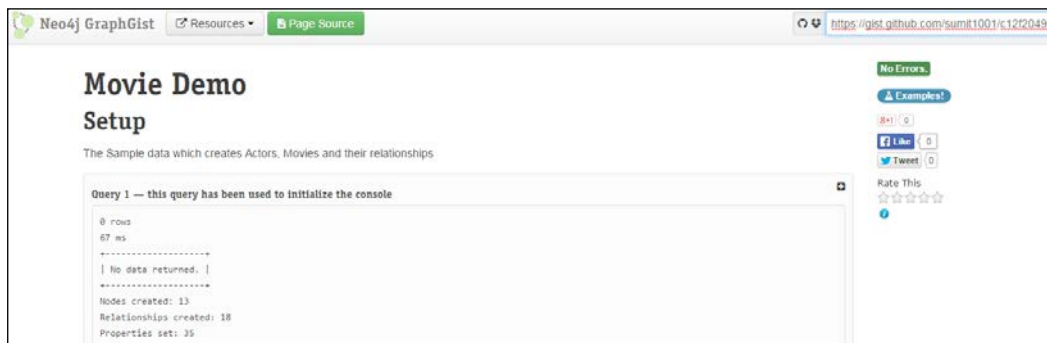
56  ----
57
58  === This will show a live Console where you can write and execute your queries!!!
59  //console
60
61  == Below will be our Queries for exploring Movie Dataset
62
63  == Get All Nodes
64  [source,cypher]
65  ----
66  MATCH (n) RETURN n;
67  ----
68  == This will show results in a table
69  //table
70
71  == This will show a Graph
72  //graph
73

```

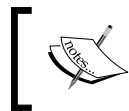
Buttons: Add file, Create secret Gist, Create public Gist

Visit <https://gist.github.com/sumit1001/c12f2049a6356a6bf664> and download the code of **ASCIIDoc** referred to in the preceding example.

5. Copy the URL as shown in your browser navigation bar.
6. Open a new browser window/tab and browse gist.neo4j.org.
7. Next in the top right-hand corner, just below the browser navigation bar, paste your GitHub Gist URL, which we copied in step 5, and click on *Enter*.



And we are done!!! Scroll down to see the Live Console, Graph, and Table.



Share your GitHub Gist URL with community members and they will be able to execute, contribute, and appreciate your work.

Summary

In this chapter, you have learned the basic concepts of data modeling in Neo4j and implementation of patterns and pattern matching in the Neo4j query language, that is, Cypher. We also talked about a new indexing feature, that is, schema over legacy indexing and, lastly, you learned a powerful way to share your work with the members of the Graph community.

In the next chapter, we will discuss Cypher queries for inserting data, creating/searching indexes, and also optimization techniques for writing performance-efficient queries.

4

Querying and Structuring Data

Reading and writing are two important aspects of any data store / database. Efficient writing of records to data structures is as important as reading records from these data structures. Reading and writing are closely linked to each other, where over engineering or focus on one process may lead to performance degradation of the other. It is important to keep a balance between performances gained by one process over the other and these are often categorized as architectural trade-offs.

Reads (using patterns) uncover the hidden patterns within data but writes in comparison to reads have a different set of challenges such as handling transactions, aligning to the principles of ACID – Atomicity, Consistency of records, Isolation, and Durability of data and transactions.

Essentially, in graph databases, reading and writing together define the *shape of data*, which is the important aspect of graph data structures.

In the last chapter, we covered the read aspects of Neo4j, and in this chapter we will talk and discuss about the write aspects of Neo4j.

This chapter will cover the following topics:

- Cypher write queries
- Writing data in legacy indexes and schema
- Unicity and other schema constraints
- Cypher optimizations

Cypher write queries

In this section, we will discuss Cypher write queries and the way they are formulated and applied to nodes, relationships, and paths using Cypher.

Working with nodes and relationships

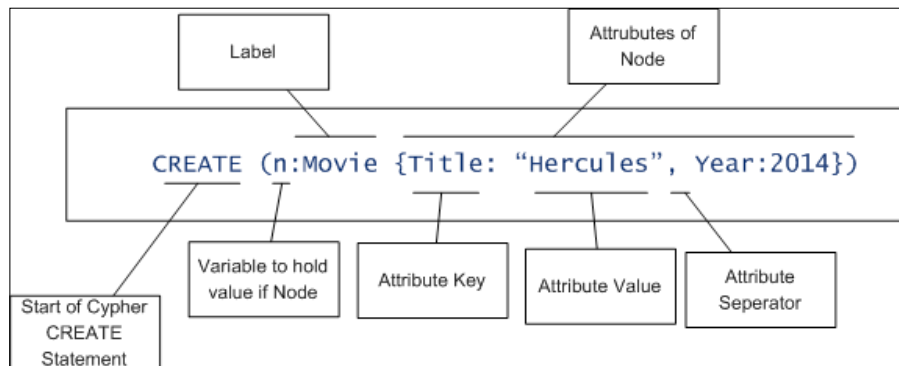
As we discussed in the earlier chapters, graph data structures essentially consist of two core elements: nodes and relationships. Nodes and relationships in Neo4j are created using the `CREATE` statement.

Let's see how `CREATE` can be used to create nodes and relationships:

- Creating a node with one or more labels and attributes:

```
CREATE (n:Movie {Title: "Hercules", Year:2014});
```

The preceding `CREATE` statement does many things. Let's understand the different parts of this statement:



We can also create the node with multiple labels and return some value to the invoking program:

```
CREATE (n:Movie:Hollywood {Title: "Hercules", Year:2014}) return n;
```

Execute the preceding CREATE statement on `<$NEO4J_HOME>/bin/neo4j-shell` and it will print the value of variable `n` on the console, as shown in the following screenshot:

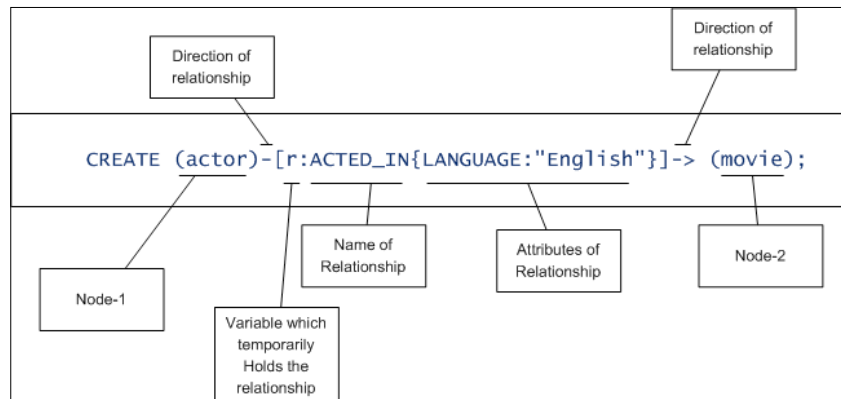
```
neo4j-sh (?)$ CREATE (n:Movie:Hollywood {Title: "Hercules", Year:2014}) return n;
+-----+
| n      |
+-----+
| Node[35]{Title:"Hercules",Year:2014} |
+-----+
1 row
Nodes created: 1
Properties set: 2
Labels added: 2
1232 ms
neo4j-sh (?)$
```

- Creating a relationship between two nodes with attributes:

Relationships are shown using a line with an arrow ahead such as `-->` or `<--`. It is also known as a directed relationship, as it shows the flow or direction of the relationship from one node to another; for example, let's create two nodes and create a relationship between them:

```
CREATE (actor:Actor {Name: "Russell Crowe", Age: 50})
CREATE (movie:Movie {Title: "NOAH", Year:2014})
CREATE (actor)-[r:ACTED_IN{Language:"English"}]-> (movie);
```

The first two CREATE statements in the preceding block of code create two distinct nodes and the last statement creates the relationship between the nodes. Similar to nodes, relationships can also have one or more attributes. The preceding statement needs to be executed as one single command on your Neo4j shell.



There is another way of writing cypher queries where we can combine multiple statements into one single statement. Extending the preceding example, we can combine all three `CREATE` statements into one single `CREATE` statement and add a `RETURN` in the end to see the actual value of nodes and relationships persisted in the Neo4j database:

```
CREATE (actor:Actor {Name: "Russell Crowe", Age: 50})-[:ACTED_
IN{Language:"English"}]->(movie:Movie {Title: "NOAH", Year:2014}) return
actor,r,movie;
```

We can also capture the pattern defined in the preceding statement in a variable and then return the variable to the invoking program:

```
CREATE p = (:Actor {Name: "Russell Crowe", Age: 50})-[:ACTED_
IN{LANGUAGE:"English"}]->(:Movie {Title: "NOAH", Year:2014})<-
[:DIRECTED{LANGUAGE:"English"}]-(:Actor {Name: "Darren Aronofsky", Age:
45})

return p;
```

Capturing patterns in a variable also helps in situations where all parts of patterns are not already in the scope of a single `CREATE` or `MATCH`. For example, in the preceding query, we are creating three nodes and two relationships in one go, which otherwise is not possible with a single `CREATE` statement.

Working with MERGE

`MERGE` is used to match a given element and return a value if it exists or create a new one if it does not exist. `MERGE` internally performs two sequential operations of first matching the given set of elements using `MATCH`, and if it does not exist, using `CREATE` to create the missing elements.

Another benefit of `MERGE` is that it avoids loading duplicate data. `MERGE` works on exact matches and follows the principle of *all or nothing*. It will either perform an exact match or create the complete pattern, but in any case, it will not be executed on partial elements of the provided pattern.

```
MERGE (actor:Actor {Name: "Russell Crowe", Age: 50}) return actor;
```

The preceding statement will first check the existence of the node with the given label and properties and will then return or create and return based on the results of the `MATCH` statement.

```
neo4j-sh (?)$ MERGE (actor:Actor {Name: "Russell Crowe", Age: 50}) return actor;
+-----+
| actor |
+-----+
| Node[29]{Name:"Russell Crowe",Age:50} |
+-----+
1 row
26 ms
neo4j-sh (?)$
```

`MERGE` is accompanied and followed with `ON CREATE` and `ON MATCH`. These allow us to perform additional changes to the nodes, properties, and relationships depending upon the result of the `MERGE` command:

```
MERGE (actor:Actor {Name: "Russell Crowe", Age: 50})
ON CREATE set actor.created_at = timestamp()
ON MATCH set actor.last_seen = timestamp()
return actor;
```

The preceding statement will first try to match the given node. If the match is successful, then it will ignore `ON CREATE` and execute the pattern defined in `ON MATCH` and will create another property by name of `last_seen` and associate it with the provided node, as shown in the following screenshot:

```
neo4j-sh (?)$ MERGE (actor:Actor {Name: "Russell Crowe", Age: 50})
> ON CREATE set actor.created_at = timestamp()
> ON MATCH set actor.last_seen = timestamp()
> return actor
> ;
+-----+
| actor |
+-----+
| Node[29]{Name:"Russell Crowe",Age:50,last_seen:1419586579763} |
+-----+
1 row
Properties set: 1
27 ms
neo4j-sh (?)$
```


MERGE can also be used with relationships where it checks whether a relationship exists, and if not, then it creates a new one:

```
MATCH (actor:Actor {Name: "Russell Crowe", Age: 50}), (movies:Movie
{Title: "NOAH", Year: 2014})
MERGE (actor)-[r:ACTED_IN{Language:"English"}]->(movies)
return actor,r,movies;
```

The preceding statement has three parts. In the first statement, it defines the MATCH clause and instructs to match the existence of the given nodes. In the second statement, it defines the MERGE statement and instructs either to create a new relationship, or if it matches, then return the existing one. Finally, in the last statement, we used `return actor,r,movies` and printed everything on the console.

Writing data in legacy indexing

We briefly touched upon the history of schema and legacy indexing in the last chapter under the *Schema and legacy indexing* section.

Let's quickly recap important aspects of schema and legacy indexing:

- Schema was introduced in Neo4j 2.0, where it provides the feature of creating indexes and constraints on the properties of nodes within a label scope
- Once schema is defined on a label, it will automatically be used by the Cypher query planner
- In legacy / manual indexing, users need to manually create/update/delete the indexes
- Searching legacy indexes is also a manual process where users need to provide the code for searching the indexes for faster retrieval
- Legacy indexes are still supported, but it is just to provide backward compatibility for Neo4j applications that are developed and are running on Neo4j 1.9.9 or earlier

Further in this chapter, we will see how writes and searches are performed in legacy indexes and schema.

Legacy indexes can be created and managed by using the Java APIs exposed by Neo4j. Let's see the usage and syntax of Java APIs for managing legacy indexes.

The following is a piece of Java code that creates two manual indexes each on nodes and relationships:

```
import org.neo4j.graphdb.*;
import org.neo4j.graphdb.factory.*;
import org.neo4j.graphdb.index.*;

public class CreateOrUpdateLegacyIndexes {

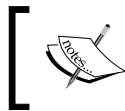
    public static void main(String[] args) {
        new CreateOrUpdateLegacyIndexes().addDataToNodeIndex();
    }

    public void addDataToNodeIndex() {
        String DBLocation = "$Neo4J_HOME/data/graph.db";
        // Get the reference of Graph Service
        GraphDatabaseService graphDb = new GraphDatabaseFactory()
            .newEmbeddedDatabase(DBLocation);
        //Start the Transaction.
        //try-with-resources is available only with Java 1.7 and above.
        try (Transaction tx = graphDb.beginTx()) {
            // Getting reference of Index Manager API
            IndexManager indexMgr = graphDb.index();
            // Creating New Index for Nodes and Relationship
            Index<Node> actorsIndex = indexMgr.forNodes("ArtistIndex");
            RelationshipIndex rolesIndex =
                indexMgr.forRelationships("RolesIndex");
            tx.success();
            tx.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

The preceding code will create two empty indexes, one called `ArtistIndex` for indexing nodes and the other called `RolesIndex`. Now, for indexing nodes and relationships, you need to manually add or remove each of the index entries by using methods provided in `org.neo4j.graphdb.index.Index` or `org.neo4j.graphdb.index.RelationshipIndex` API.

Let's continue the preceding example and add nodes to the `ArtistIndex`. Add the following code snippet just before `tx.success()` ;

```
// Creating a Node
Node node = graphDb.createNode();
// Adding Label and properties to Node
node.addLabel(DynamicLabel.label("Actors"));
node.setProperty("Name", "Ralph Macchio");
// Adding Node and its properties to the Index
actorsIndex.add(node, "Name", node.getProperty("Name"));
```



In order to understand the preceding code, follow the comments provided within the code. This needs to be followed for the other Java code snippets given in this chapter.

Next, extending the same example, let's populate an index for relationships and add the following code snippet just before `tx.success()` ;:

```
// Creating new node and Adding Label and properties
Node movieNode = graphDb.createNode();
movieNode.addLabel(DynamicLabel.label("Movie"));
movieNode.setProperty("Title", "The Karate Kid II");
// Define a relationship Type
DynamicRelationshipType actsIn = DynamicRelationshipType.withName(
    "ACTS_IN" );
//Create a relationship between Actor and Movie
//and define its property
Relationship role1 = node.createRelationshipTo( movieNode,
    actsIn);
role1.setProperty( "Role", "Daniel LaRusso" );
rolesIndex.add(role1, "Role",role1.getProperty( "Role" ) );
```

The next step is to search the legacy indexes. Legacy API provides the following kinds of searches:

- **Exact search:** This allows us to search for the exact key-value pair
- **Search for more than 1 key value pair:** This allows us to search with * or ?.
- **Full text:** Legacy index API extends Lucene APIs, so it also provides full text searches

Continuing with the same example, add the following code snippet just before `tx.success()` ; for searching nodes:

```
//Search the ActorsIndex by using the exact Match of Key and value
IndexHits<Node> hits = actorsIndex.get( "Name", "Ralph Macchio" );
Node searchNode = hits.next();
System.out.println("Node ID = "+searchNode.getId()+", Name = "+searchNode.getProperty("Name"));
//Search the ActorsIndex for more than one property
hits = actorsIndex.query("Name", "*a*");
Node searchNode1 = hits.next();
System.out.println("Node ID = "+searchNode1.getId()+", Name = "+searchNode1.getProperty("Name"));
```

And we are done!!!

For compiling and running the preceding code snippets, perform the following steps and execute it from IDE itself:

1. Stop your Neo4j server (if it is already running).
2. Initialize the Java variable `DBLocation` with the location of your Neo4j database as per your local filesystem, which is generally available at `<$NEO4J_HOME>/data/graph.db`.
3. Provide all JAR files from `<$NEO4J_HOME>/lib/` in the classpath.
4. Compile and execute the preceding code.

For deleting and removing indexes, you need to invoke the `delete()` and `remove()` methods of `org.neo4j.graphdb.index.Index`.



There is no operation for updating the index entries. First, the index entries need to be removed and added again to perform the update operations.

Neo4j also provides options to manage the behavior of the indexes. We can create a full text search index or case-insensitive search index:

```
//Creating and initializing Full text Search Index
Index<Node> indexFullText = graphDb.index().forNodes(
    "full_text_search_index", MapUtil.stringMap( IndexManager.PROVIDER,
    "lucene", "type", "fulltext" ));
//Creating and initializing an exact search index, which is case insensitive.
Index<Node> indexCaseInsensitive = graphDb.index().forNodes(
    "exact-case-insensitive", MapUtil.stringMap( "type", "exact",
    "to_lower_case", "true" ));
```

Writing data in a schema

Neo4j 2.0 introduces the schema for managing the indexes and constraints. Legacy indexes are still supported but are deprecated and would be removed from the later versions of Neo4j. The rationale behind schema was to provide an API that leverages indexes as soon as they are defined by the users. The objective was to have an automatic process for inserting/ updating and searching the indexes without any manual intervention. At the time of writing this book, Neo4j Version 2.1.5 was released and APIs in schema were available to provide core features of indexes but still there were some features such as full-text searches case-insensitive searches, or spatial searches that are still in development and are due to be released in the upcoming versions of Neo4j.

Let's see the usage of various APIs in the schema package for managing indexes using Java and REST APIs.

Managing schema with Java API

In this section, you will learn about the various ways of performing CRUD operations over indexes provided by schema using the Java API.

The following Java code creates the index on the `Movie` Label:

```
GraphDatabaseService graphDb = new GraphDatabaseFactory()
    .newEmbeddedDatabase(DBLocation);
// Get the Schema from the Graph DB Service
Schema schema = graphDb.schema();
// Create Index on the provided Label and property
IndexDefinition indexDefinition =
    schema.indexFor(DynamicLabel.label("Movie")).on("Title").create();
```

The following Java code fetches all indexes from the Neo4j database and prints them on the console:

```
GraphDatabaseService graphDb = new GraphDatabaseFactory()
    .newEmbeddedDatabase(DBLocation);
// Get the Schema from the Graph DB Service
Schema schema = graphDb.schema();
// Iterate through all Indexes and Print the Labels and Properties
for (IndexDefinition inDef : schema.getIndexes()) {
    System.out.println("Label = " + inDef.getLabel().name()
        + ", Property Keys = " + inDef.getPropertyKeys());
}
```

The following Java code deletes all the indexes on the `Movie` label:

```
GraphDatabaseService graphDb = new GraphDatabaseFactory()
    .newEmbeddedDatabase(DBLocation);
// Get the Schema from the Graph DB Service
Schema schema = graphDb.schema();
// Iterate through all Indexes on a Label Movies and delete them
for (IndexDefinition inDef :
    schema.getIndexes(DynamicLabel.label("Movie"))) {
    inDef.drop();
}
```

For compiling and running each of the preceding code snippets, perform the following steps and execute it from the IDE itself. You should now see that indexes are being created/listed/dropped on the `Movies` label:

1. Stop your Neo4j server (if it is already running).
2. Provide all JAR files from `<$NEO4J_HOME>/lib/` in the classpath.
3. Apart from `org.neo4j.graphdb.*` and `org.neo4j.graphdb.factory.*` add or import `org.neo4j.graphdb.schema.*` as an additional package.
4. Wrap the code around the transaction in a try and catch block: `try (Transaction tx = graphDb.beginTx()) {}`. `try-with-resources` is available only with Java 1.7 and above.
5. Initialize the Java variable `DBLocation` with the location of your Neo4j database as per your local filesystem, which is generally available at `<$NEO4J_HOME>/data/graph.db`.



Updating / Inserting data is an automatic process, so there are no APIs that provide operations for updating or inserting data.

Managing schema with REST

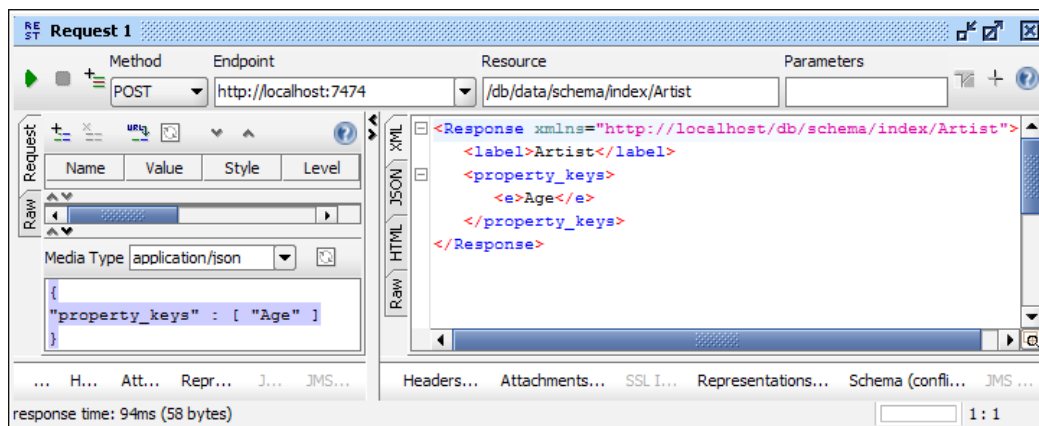
Let's create/retrieve/delete indexes using the REST API and use tools such as SoapUI or extensions to the Chrome / Firefox browser (advanced REST client or REST easy), which support testing of REST-based services / representation to execute the JSON request with the provided configuration.

Creating an index on Artists (Age) . Use the following configuration in your REST-based testing tool:

- The request method type is **POST**
- The request URL for creating an index is `http://<HOST>:<PORT>/db/data/schema/index/Artist`
- The headers are `Accept: application/json; charset=UTF-8` and `Content-Type: application/json`
- Put the following JSON request in the box provided for posting JSON request and execute it:

```
{"property_keys" : [ "Age" ]}
```

The preceding REST request will return the label and property on which the index is created — Artists (Age). The response of the preceding REST request will be similar to the results shown in the following screenshot:



To get all indexes on the Artist label, use the following configuration in your REST-based testing tool and execute the request:

- Request method type: **GET**
- Request URL for fetching index: `http://<HOST>:<PORT>/db/data/schema/index/Artist`
- Headers: `Accept: application/json; charset=UTF-8`

The preceding request will fetch all the indexes created on the Artist label.

To delete the index on the `Age` property, which is tagged with the `Artist` label, use the following configuration in your REST-based testing tool and execute the request:

- Request method type: **DELETE**
- Request URL for deleting index: `http://<HOST>:<PORT>/db/data/schema/index/Artist/Age`
- Headers: `Accept: application/json; charset=UTF-8` and `Content-Type: application/json`

The preceding request will delete the index on the `Age` property, which is tagged with the `Artist` label.

Unicity and other schema constraints

Neo4j 2.0 introduced operations for applying constraints on labels and properties. This not only helps in maintaining the data integrity but also provides shape to our data. The current version of Neo4j 2.1.5 only supports unique constraints, which ensure unicity or uniqueness of data residing in Neo4j.

Let's see the process of applying unique constraints in REST and Java.

Applying unicity constraints with REST

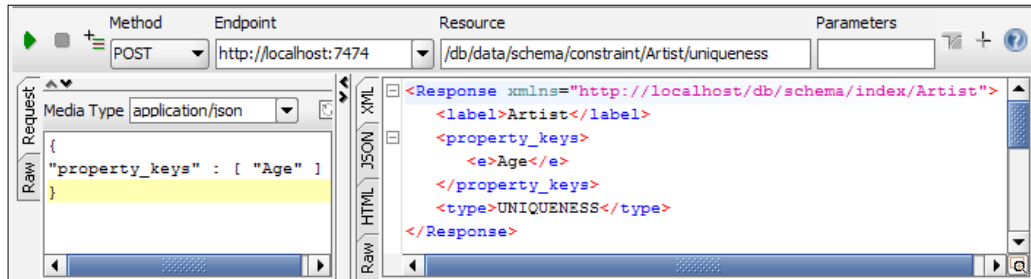
Let's create/retrieve/delete the unique constraints using REST API and use tools such as SoapUI or extensions to the Chrome / Firefox browser (advanced REST client or REST easy), which supports testing of REST-based services / representation to execute the JSON request with the provided configuration.

To create a constraint on `Artist (Age)`, use the following configuration in your REST-based testing tool:

- The request method type is **POST**
- The request URL for creating a constraint is `http://<HOST>:<PORT>/db/data/schema/constraint/Artist/uniqueness`
- The headers are `Accept: application/json; charset=UTF-8` and `Content-Type: application/json`
- Put the following JSON request in the box provided for posting the JSON request and execute it:

```
{"property_keys" : [ "Age" ] }
```

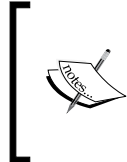

The response of the preceding REST request will be similar to the results shown in the following screenshot:



To get all constraints on the Artist label, use the following configuration in your REST-based testing tool and execute the request:

- The request method type is **GET**
- The request URL for fetching constraints on the Artist label is `http://<HOST>:<PORT>/db/data/schema/constraint/Artist`
- The headers are `Accept: application/json; charset=UTF-8`

The preceding request will fetch all the constraints created on the Artist label.



For deleting the constraint, you need to use DELETE as the request method type and append the property to the URL on which the index needs to be deleted, for example, `DELETE /db/data/schema/constraint/Artist/ uniqueness /Age` where Artist is the label and Age is the property.

Applying unicity constraints with Java

Let's create/retrieve/delete the unique constraints using the Java API.

The following Java code creates a unique constraint on the Artist label:

```
// Get Object of Graph Service
GraphDatabaseService graphDb = new GraphDatabaseFactory()
    .newEmbeddedDatabase(DBLocation);
// Open a Transaction
try (Transaction tx = graphDb.beginTx()) {
    //Get Schema from Graph Database Service
    Schema schema = graphDb.schema();
```

```
//Define Unique Constraint and Label and Property
ConstraintCreator creator =
schema.constraintFor(DynamicLabel.label("Artist")).assertPropertyI
sUnique("Age");
//Create the Constraint
creator.create();
tx.success();
tx.close();
} catch (Exception e) {e.printStackTrace();}
```

The following Java code lists all constraints on the Artist label:

```
// Get Object of Graph Service
GraphDatabaseService graphDb = new GraphDatabaseFactory()
.newEmbeddedDatabase(DBLocation);
// Open a Transaction
try (Transaction tx = graphDb.beginTx()) {
    Schema schema = graphDb.schema();
    for (ConstraintDefinition def :
        schema.getConstraints(DynamicLabel.label("Artist"))) {
        System.out.println("Label = Artist," + "Type of Constraint =
            "+ def.getConstraintType());
        System.out.print("ON Properties = ");
        for (String keys : def.getPropertyKeys()) {
            System.out.print(keys);
        }
        System.out.println();
    }
} catch (Exception e) {e.printStackTrace();}
```

Perform the following steps to compile and run the preceding code snippets from the IDE:

1. Stop your Neo4j server (if it is already running).
2. Provide all the JAR files from <\$NEO4J_HOME>/lib/ in the classpath.
3. Add or import org.neo4j.graphdb.schema.* as an additional package.
4. Wrap the code around the transaction in a try and catch block: try (Transaction tx = graphDb.beginTx()) {}. try-with-resources is available only with Java 1.7 and above.
5. Initialize the Java variable, DBLocation, with the location of your Neo4j database as per your local filesystem, which is generally available at <\$NEO4J_HOME>/data/graph.db.

The constraints can be deleted by using `ConstraintDefinition#drop`.



Creating the constraint automatically creates the index on the same column and will be deleted as soon as the constraint is dropped.

Cypher optimizations

Cypher query execution engine plans and selects the best possible path to execute a given query in the shortest possible time, but human (or rather developer) intervention is necessary for optimizing those aspects of the query that are dependent upon the understanding of the data and the domain.

Although there is no single universal method of achieving optimum performance, we do have some guidelines, which if followed and used appropriately, could give you the best results.

Let's discuss these guidelines and see how they can help us in achieving optimal performance for our Cypher queries:

- **Divide and conquer:** The fundamental law of performance / optimization is, "You need to measure first and then optimize," and it is true for Cypher too. We need to plan and strategize our efforts. So as a first step, divide your complete Cypher query into multiple pieces and see how each of the pieces perform with respect to the number of records and time. Next, start from the worst performing piece and try tuning it. In the majority of cases, half of the problem is solved just by knowing what to optimize; otherwise, it takes a good amount of time and effort just to uncover the areas that require tuning/optimization.



Append your Cypher queries with `profile` keywords to get the execution plan and statistics.

- **Plan for the worst:** Always tune your queries on the worst or minimum hardware and see the blazing fast speed on production boxes.

- **Execution engine and plans:** Cypher internally creates an execution plan for each and every Cypher query. Once the plan is created, it is cached and re-used in the future for the same set of Cypher queries. Enabling Cypher to re-use the cache plans saves a good amount of time and resources. There are two different ways that enable the Cypher to cache/re-use execution plans:
 - **Using parameters:** Replacing literals with parameters in your Cypher will enable the execution engine to re-use the cached query plans, though parameters of the query might be different. It is similar to the `PreparedStatement` in the `java.sql` package.
 - **Execution engine:** Re-use the object of `ExecutionEngine`. Though it is a small thing, if it is not taken care of, it can cause a considerable degradation of Cypher queries.
- **Patterns in the WHERE clause:** It is not that you cannot use it but it comes with a cost, which if taken care of, can considerably improve the time of the execution. Patterns in `WHERE` should be used only for the `not` conditions otherwise all other patterns can be used either in `Match` or we can combine two `Match` statements using the `WITH` clause. Always remember to use the `WHERE` condition only to filter the result set and-for all other cases such as selecting the results-use `MATCH`.
- **Indexing/Constraints:** Cypher uses indexes and constraints to scan the nodes and relationships. Depending upon your dataset and query, you should carefully analyze and see whether all the required indexes and constraints are `AVAILABLE` and `ONLINE`.
- **Data filtering:** Data should be filtered as early as possible with the least possible matches involved. It helps in scanning smaller result sets as we move forward and evaluate the rest of the query.
- **Data model:** In Neo4j, everything is based on the data model. A good understanding of the data model provides a good sense of structure of the graph, organization of nodes, relationships, highly / sparsely connected nodes, and so on. It is important to understand the data model for writing efficient queries to retrieve the data.

Optimization and tuning are very subjective and may vary from case to case, but all the guidelines mentioned earlier will help you to uncover and solve performance pitfalls for most of the scenarios. However, there is still room for the rarest of rare cases, and for all those, drop your queries at <http://stackoverflow.com/questions/tagged/neo4j> or <https://support.neo4j.com/>.

Summary

In this chapter, you have learned about the basic concepts and syntactical details of Cypher write queries for inserting data. We also talked about inserting data in schema and legacy indexing using Java and REST APIs. Finally, we also discussed standards for optimizing Cypher queries.

In the next chapter, we will discuss the integration and relationship of Neo4j and Java. We will also talk about Java APIs exposed by Neo4j and their applicability.

5

Neo4j from Java

One of the most important aspects of any software/application/database is its ability to integrate with the existing applications and the deployment models / options exposed and provided to the users.

Organizations in today's world already have some sort of software deployed and used by their internal or external users for managing their day-to-day operations and storing the data. Now, planning to deploy a new software/application would definitely bring up the question of flexibility provided by the new software and how well it can be integrated with the existing applications.

Here are a few such questions that organizations want to answer for any new software that is proposed to be deployed:

- Will I be able to integrate the solution with our home-grown applications? Does it support any out-of-the-box integrations?
- Do we want to make a smaller investment up-front or are we looking for an incremental return over the long - term? Do we need costly, dedicated, and high-end infrastructure now or in the near future?
- What about my organization's security concerns?
- Will my deployment choice support a growing business over time?

Neo4j supports different ways to build and deploy Graph Database and the fact that it is written in Java and Scala have exposed various options for testing, deployment, and configuration.

In this chapter, you will learn how to integrate Neo4j with the existing Java applications and it will cover the following points:

- Embedded versus REST
- Unit testing in Neo4j
- Java APIs
- Graph traversals

Embedded versus REST

Neo4j provides more than one deployment and configuration models to integrate with the existing applications. It exposes all the required Java APIs for deploying the server as an embedded application or a standalone / REST-based server.

Although both deployment models have their own pros and cons, the decision for production deployment model would depend upon the vision and the usage of the software.

Let's first understand each of the deployment models and then we will discuss the advantages/disadvantages and applicability of each of the models.

Embedding Neo4j in Java applications

Embedding Neo4j in a Java application should not be confused with the in-memory database. However, Neo4j exposes all the required APIs to deploy the server as an embedded application or configure it as in-memory database.

Execute the following steps to configure and run Neo4j as an embedded server within your existing applications:

1. Choose the appropriate Neo4j Edition – Community or Enterprise (refer to the *Licensing options* section in *Chapter 1, Installation and the First Query*).
2. You can either manually configure your project or use Maven to configure it. Here we will define the steps to configure Maven-based Java projects; but later in this section, we will also talk about the process to set up Java projects manually using an **Integrated Development Environment (IDE)** such as Eclipse – <https://eclipse.org/>, IntelliJ IDEA – <https://www.jetbrains.com/idea/>, NetBeans – <https://netbeans.org/>, and so on. Perform the following steps to create a Maven project:
 1. Download Maven 3.2.3 from <http://maven.apache.org/download.cgi>.
 2. Once the archive is downloaded, browse the directory on your filesystem and extract it.
 3. Define the system environment variable `M2_HOME=<location of extracted Archive file >`.
 4. Add `$M2_HOME/bin/` to your `$PATH` variable.

5. Next, open your console and browse the location where you want to create your new Maven project and execute:

```
mvn archetype:generate -DgroupId=neo4j.embedded.myserver
-DartifactId=MyNeo4jSamples -DarchetypeArtifactId=maven-
archetype-quickstart -DinteractiveMode=false
```

6. It will take some time as Maven will download the required dependencies to create your project. You will see a `MyNeo4jSamples` directory being created as soon as the project is created.
7. Edit the `MyNeo4jSamples/pom.xml` file and add the following piece of code under the dependencies section for defining the dependencies of the Neo4j libraries:

```
<dependencies>
  <dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j</artifactId>
    <version>2.1.5</version>
  </dependency>
</dependencies>
```

3. Next, create a Java class by the name `neo4j.embedded.myserver.EmbeddedServer.java` and add following piece of code:

```
public class EmbeddedServer {

    private static void registerShutdownHook(final
    GraphDatabaseService graphDb) {

        // Registers a shutdown hook for the Neo4j.
        Runtime.getRuntime().addShutdownHook(new Thread() {
            public void run() {
                //Shutdown the Database
                System.out.println("Server is shutting down");
                graphDb.shutdown();
            }
        });
    }

    public static void main(String[] args) {
        //Create a new Object of Graph Database
        GraphDatabaseService graphDb = new
        GraphDatabaseFactory().newEmbeddedDatabase("Location of
        Storing Neo4j Database Files");
        System.out.println("Server is up and Running");
        //Register a Shutdown Hook
        registerShutdownHook(graphDb);
    }
}
```


And we are done!!! Now we can use the instance of graph database by the name `graphDb` for creating nodes/relationships and properties.

To run the preceding code, perform the following steps:

1. Import the following Java packages in your `EmbeddedServer.java` program:
 - `org.neo4j.graphdb.GraphDatabaseService;`
 - `org.neo4j.graphdb.factory.GraphDatabaseFactory;`
2. Open the console and browse the location where you have created your Maven project.
3. Now, execute following command to compile and create a build:
`mvn clean compile install`
4. Next, to run your embedded server, execute the following Maven command in the console:

```
mvn exec:java -Dexec.mainClass="neo4j.embedded.myserver.  
EmbeddedServer" -Dexec.cleanupDaemonThreads=false
```

Let's understand the different sections of the preceding code:

```
GraphDatabaseService graphDb = new GraphDatabaseFactory()  
.newEmbeddedDatabase("Location of Storing Neo4j Database Files");
```

The preceding piece of code initializes the Neo4j database and stores the database files in the given directory, which should be the path of the local filesystem.

```
//Register a Shutdown Hook  
registerShutdownHook(graphDb);
```

The preceding statement defines a function that registers a shutdown hook for the Neo4j database so that it shuts down nicely when the VM exits. It also ensures a clean shutdown even if you execute *Ctrl + C* on the running application.

We can also configure the various database configurations using the functions provided in `GraphDatabaseBuilder`. For example, the database creation code can be rewritten as follows:

```
GraphDatabaseFactory graphFactory = new GraphDatabaseFactory();  
GraphDatabaseBuilder graphBuilder =  
graphFactory.newEmbeddedDatabaseBuilder("Location of Storing Neo4j  
Database Files");  
graphBuilder.loadPropertiesFromFile("location of file containing  
the Neo4j database properties");  
GraphDatabaseService graphDb = graphBuilder.newGraphDatabase();
```



We can also configure the Neo4j database properties using the `GraphDatabaseBuilder.setConfig(...)` method. For example, `graphBuilder.setConfig(GraphDatabaseSettings.dump_configuration, "true");`. For the available configuration parameters, refer to the attributes defined in `org.neo4j.graphdb.factory.GraphDatabaseSettings.java` at <http://neo4j.com/docs/2.1.5/javadocs/org/neo4j/graphdb/factory/GraphDatabaseSettings.html>.

Before moving to the next section, let's also discuss the process of manually configuring the Java project. Perform the following steps to manually configure your Java project:

1. Download and open your favorite IDE (Eclipse, IntelliJ, NetBeans, and so on), create a new Java project, and name it as `MyNeo4jSamples`.
2. Add all the JAR files from `$NEO4J_HOME>/lib` as the dependency to your new Java project.
3. Add a new package in your Java project by the name `neo4j.embedded.myserver`.
4. Next, create a Java class `neo4j.embedded.myserver.EmbeddedServer.java` and add the same code that we discussed in step 3 for Maven-based projects.
5. Next, directly execute code from your IDE and the results will be the same as you see with your Maven-based Java projects.



Using appropriate plugins, you can create and execute Maven projects from the IDE itself. For example, in Eclipse, you can install and use M2E plugins. For more information, refer to <http://eclipse.org/m2e/>.

Neo4j as a REST-based application

Neo4j exposes various REST endpoints for performing CRUD and search operations over the Neo4j database. In order to work with these REST endpoints, we need to have a running Neo4j server and develop a REST client that can invoke these different services.

Let's understand and perform the following steps to deploy the Neo4j server and develop the REST client for invoking REST APIs exposed by the Neo4j server:

1. Open your console and execute `$NEO4J_HOME/bin/neo4j`. Wait for successful execution of the command and open your browser and type `http://localhost:7474/`. You will see the default Neo4j browser.



In case something goes wrong, then check for errors in `$NEO4J_HOME/data/messages.log`.

2. Let's extend our `MyNeo4jSamples` project, which we created in the previous section. Create Java REST clients, and then invoke the REST APIs exposed by the Neo4j server.
3. Open the `MyNeo4jSamples/pom.xml` file and add the following dependencies under the `<dependencies>` section:

```
<dependency>
  <groupId>com.sun.jersey</groupId>
  <artifactId>jersey-client</artifactId>
  <version>1.8</version>
</dependency>
<dependency>
  <groupId>org.json</groupId>
  <artifactId>json</artifactId>
  <version>20140107</version>
</dependency>
```

4. Add a new package called `neo4j.rest.client` and a Java class `MyRestClient.java`, and import the following packages:

```
import org.json.JSONObject;
import javax.ws.rs.core.MediaType;
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.ClientResponse;
import com.sun.jersey.api.client.WebResource;
```

5. Next we will define a new method in `MyRestClient.java` for adding a new node:

```
public class MyRestClient {
  public void createNode() {
    //Create a REST Client
    Client client = Client.create();
```

```

        //Define a resource (REST Endpoint) which needs to be
        //Invoked for creating a Node
        WebResource resource =
        client.resource("http://localhost:7474").path
        ("/db/data/node");
        //Define properties for the node.
        JSONObject node = new JSONObject();
        node.append("Name", "John");

        //Invoke the rest endpoint as JSON request
        ClientResponse res =
        resource.accept(MediaType.APPLICATION_JSON)
        .entity(node.toString())
        .post(ClientResponse.class);
        //Print the URI of the new Node
        System.out.println("URI of New Node = " +
        res.getLocation());
    }
}

```

6. Next, we will define another method in `MyRestClient.java`, just above the closing braces of the class (`}`), and add the following code to query the database using Cypher:

```

public void sendCypher() {
    //Create a REST Client
    Client client = Client.create();
    //Define a resource (REST Endpoint) which needs to be
    //Invoked
    //for executing Cypher Query
    WebResource resource =
    client.resource("http://localhost:7474").path
    ("/db/data/cypher");
    //Define JSON Object and Cypher Query
    JSONObject cypher = new JSONObject();
    cypher.accumulate("query", "match n return n limit 10");

    //Invoke the rest endpoint as JSON request
    ClientResponse res =
    resource.accept(MediaType.APPLICATION_JSON)
    .entity(cypher.toString())
    .post(ClientResponse.class);
    //Print the response received from the Server
    System.out.println(res.getEntity(String.class));
}

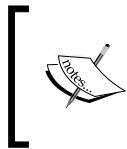
```

7. Now create an object of `MyRestClient` in your main method and invoke the `createNode()` and `sendCypher()` methods one by one.
8. Now execute the following command to compile and create a build:
9. To run your `MyRestClient`, execute the following Maven command in the console:

```
$M2_HOME/bin/mvn clean compile install  
  
$M2_HOME/bin/mvn exec:java -Dexec.mainClass="neo4j.rest.client.  
MyRestClient" -Dexec.cleanupDaemonThreads=false.
```

Both the methods `createNode()` and `sendCypher()` expose different endpoints for handling operations on nodes and query data with Cypher. The `createNode()` method will print the URI of the newly created node along with its ID `/db/data/node/{Node_ID}` and `sendCypher()` will print the results of the Cypher query on the console in the JSON format.

Similarly, there are other endpoints exposed by the server for performing operations on properties, relationships, labels, and so on.



For a complete list of REST endpoints, use your rest client and execute the URL <http://localhost:7474/db/data>. For more details, refer to <http://neo4j.com/docs/stable/rest-api.html>.

Which is best?

It depends!!! Yes, that's true, it all depends.

Both the options have their own advantages and the choice/applicability of the deployment model will largely depend upon your vision, use case, and the environment. Let's discuss each of the options, their advantages and disadvantages, and then we will talk about the best deployment model.

Let's start describing few of the advantages of deploying Neo4j as a REST-based server:

- It enables multiple applications (deployed in separate contexts) to perform CRUD operations over a single Neo4j database
- It provides scalability and High Availability (with the Enterprise Edition)
- It provides more than one language or client for performing CRUD operations
- It provides an administrator console and a browser to browse data

Wow!!! It's nice but that's not all. Let's discuss some of the disadvantages of the REST-based server:

- It takes a considerable amount of time to develop an application. Lots of code needs to be written for developing small-to-medium-sized applications.
- It does not provide performance (compared to embedded), as the request is over HTTP.

We have not concluded yet. Let's now talk about the embedded server, which has the following advantages:

- It is robust and well-defined and well-documented Java APIs are available for performing CRUD and traversals over the graph database
- It is easy to manage as it is only a single process that needs to be monitored and managed
- It provides maximum performance, as calls are local and not remote

It's nice too, but it also has its downside. Let's discuss some of the disadvantages of the embedded server:

- Neo4j database cannot be shared or used by multiple clients, which limits the scalability
- There is no admin console available
- It only supports JVM-based languages (such as Java or Scala)

Now, let's move ahead and discuss scenarios where we can think of leveraging either of the deployment models.

Scalability and High Availability are one of the critical **non-functional requirements (NFRs)** for most of the enterprise systems and that is sufficient reason/argument to deploy Neo4j as a REST-based server and then developing REST clients to perform CRUD and search operations. Also this is one of the most common forms of deployment models for the production environments as it provides scalable and highly available database systems for your enterprise applications.

But there are applications where quick and faster development is a priority or you do not want to have dedicated hardware or a separate process to be deployed, which may be too costly due to many obvious reasons, and in all those cases, Neo4j as an embedded server would be the best choice.

Considering the advantages/disadvantages and the scenarios, we would conclude that there is no single deployment model which works the best or is applicable for all scenarios. So based on the needs of your use case, decide upon the appropriate deployment model. It is good to have the requirements and vision of your application drive the selection of deployment models and not the other way round.

Unit testing in Neo4j

Unit testing is an important aspect of any development lifecycle. It not only involves the testing of the expected outcome, but at the same time, it should also test the unexpected conditions/scenarios and the behavior of the system.

In this fast-paced development environment where delivery models such as Agile—http://en.wikipedia.org/wiki/Agile_software_development—or Extreme Programming—http://en.wikipedia.org/wiki/Extreme_programming—focus on delivering the enhancements/features in the shortest possible time, a good set of unit test suites helps developers to ensure that adding new code does not break the integrity of the system, no matter how far the developers are familiar/aware of the rest of the code base. The goal of unit testing is to isolate each part of the program and show that the individual parts are correct, which is also used as a regression test suite later in the development cycle.

Even in the development models such as **Test-driven Development (TDD)**—http://en.wikipedia.org/wiki/Test-driven_development, the developer first writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally, refactors the new code to acceptable standards.

There are several benefits of unit testing, which we will not discuss but it would be worth mentioning a few of them:

- **Problem solving:** Identifying and solving problems early in the development cycle
- **Facilitates change:** Unit testing allows the programmer to refactor code at a later date
- **Simplifies integration:** It reduces uncertainty in the units themselves
- **Separation of concerns:** Interfaces are separated from implementation where each method is tested as a "unit" and the rest of the dependencies are injected by some other means such as mocks

Neo4j data models, as we discussed in *Chapter 3, Pattern Matching in Neo4j*, are evolving and developed over the period of time, so the code that supports these evolving models is also highly agile and is constantly changing. In these scenarios, unit testing becomes more important and every piece of code, either added or modified, should ensure that it does not break the other pieces of the system.

JUnit (provided by <http://junit.org>) is one of the most popular frameworks for writing Java-based unit test cases.

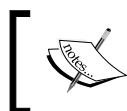
Let's perform the following steps to integrate JUnit framework, and enhance our Maven project, *MyNeo4jSamples*, and then add some basic unit test cases:

1. Open your *MyNeo4jSamples/pom.xml* file and add the following properties just below the `<packaging>` tag:

```
<properties>
  <java.version>1.7</java.version>
  <maven.compiler.source>1.7</maven.compiler.source>
  <maven.compiler.target>1.7</maven.compiler.target>
</properties>
```

2. Add the following dependencies in the `<dependency>` section:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.11</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.hamcrest</groupId>
  <artifactId>hamcrest-all</artifactId>
  <version>1.3</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.neo4j</groupId>
  <artifactId>neo4j-kernel</artifactId>
  <version>2.1.5</version>
  <type>test-jar</type>
  <scope>test</scope>
</dependency>
```



In case any of these dependencies already exist in your *pom.xml* file, then just update the version of the package defined in `<version>` tag.

3. Create a package `neo4j.tests` and a Java class `Neo4jTest.java` under `src/test/java` and add following code:

```
package neo4j.tests;

import org.junit.After;
import org.junit.Before;
import org.neo4j.graphdb.GraphDatabaseService;
import org.neo4j.test.TestGraphDatabaseFactory;

public class Neo4jTest {

    private GraphDatabaseService graphDb;

    @Before
    public void createTestDatabase() throws Exception{
        try {

            System.out.println("Set up Invoked");
            graphDb = new
                TestGraphDatabaseFactory().newImpermanentDatabase();

        } catch (Exception e) {
            System.err.println("Cannot setup the Test
                Database....check Logs for Stack Traces");
            throw e;
        }
    }

    @After
    public void cleanupTestDatabase() throws Exception {
        try {
            System.out.println("Clean up Invoked");
            graphDb.shutdown();
        } catch (Exception e) {
            System.err.println("Cannot cleanup Test
                Database....check Logs for Stack Traces");
            throw e;
        }
    }
}
```

In the preceding code, we have defined two methods `createTestDatabase` and `cleanupTestDatabase`, where the former creates a test database and the latter shuts down the database. These methods are automatically invoked by the JUnit framework and the annotations `@Before` and `@After` help JUnit framework to identify these setup and cleanup methods. Also, these methods are invoked before and after execution of each JUnit test case.

Neo4j provides a test database factory, which creates a test database with minimum configuration and helps developers to focus on writing unit test cases, rather than setting up the database. Here is that piece of code, which creates a test database `new TestGraphDatabaseFactory().newImpermanentDatabase()`;

4. Next, open your console and browse your project root, that is, `MyNeo4jSamples` and execute `$M2_HOME/bin/mvn test` for executing your test cases.
5. Your build should fail, because there are no unit tests to be executed. So now, let's add a unit test case by the name of `nodeCreationWithLabel` in `Neo4jTest.java` and add the following code:

```
@org.junit.Test
public void nodeCreationWithLabel() {
    //Open a Transaction
    try(Transaction transaction=graphDb.beginTx()){
        String testLabel="TestNode";
        //Create a Node with Label in the neo4j Database
        graphDb.createNode(DynamicLabel.label(testLabel));
        //Execute Cypher query and retrieve all Nodes from
        Neo4j Database
        ExecutionEngine engine = new ExecutionEngine(graphDb,
        StringLogger.SYSTEM);
        String query = "MATCH (n) return n";
        ExecutionResult result = engine.execute(query);
        Iterator <Object> objResult = result.columnAs("n");
        //Check that Database has only 1 Node and not more than
        1 node
        Assert.assertTrue(objResult.size()==1);
        while(objResult.hasNext()){
            Node cypherNode = (Node)objResult.next();
            //Check that Label matches with the same Label what
            was created initially
            Assert.assertTrue(cypherNode.hasLabel
            (DynamicLabel.label(testLabel)));
        }
    }
```

```
        transaction.success();
    }
}
```

Let's understand the preceding JUnit test:

```
graphDb.createNode(DynamicLabel.label(testLabel));
```

The preceding line creates a node in the test database with the provided label.

```
//Execute Cypher query and retrieve all Nodes from Neo4j Database
ExecutionEngine engine = new ExecutionEngine(graphDb,
StringLogger.SYSTEM);
String query = "MATCH (n) return n";
ExecutionResult result = engine.execute(query);
Iterator <Object> objResult = result.columnAs("n");
```

After creating node in the test database, the preceding code then executes the Cypher query to get the results from the same database.

```
Assert.assertTrue(objResult.size()==1);
```

Next, the preceding code checks whether the Cypher query has returned only one node. Not more and not less!

```
while(objResult.hasNext()){
    Node cypherNode = (Node)objResult.next();
    //Check that Label matches with the same Label what was created
    initially
    Assert.assertTrue(cypherNode.hasLabel(DynamicLabel.label
(testLabel)));
}
```

Further, we iterate through the results of the Cypher query and check whether the label of both the nodes is the same.

6. Now, let's run our JUnits once again and execute `$M2_HOME/bin/mvn test` from the root of your Java project, that is, `MyNeo4jSamples`, and this time you should see a successful build.

Easy and simple...isn't it?

Yes it is, but the only drawback is that there is a lot of boilerplate code such as iterating through the Cypher result set, which needs to be written for each and every JUnit.

Although we can create some helper methods, they also need to check everything such as nodes, labels, properties, paths, and many more conditions, which would be a time-consuming task.

Testing frameworks for Neo4j

In the previous section, we talked about Neo4j and the process of writing JUnit using the framework provided by <http://junit.org>.

We also discussed the shortcomings of JUnit because of the boilerplate code that needs to be written, even for a simple JUnit. It not only takes time but is also difficult to maintain.

To overcome all these shortcomings, frameworks such as GraphUnit—<http://graphaware.com/>—and AssertJ—<http://joel-costigliola.github.io/assertj/assertj-neo4j.html>—were evolved which provide various assertions to reduce unnecessary code (boilerplate code) and help developers to focus on writing effective and efficient unit tests.

Let's enhance our test suite, that is, `neo4j.tests.Neo4jTest.java` and see how we can leverage assertions provided by GraphUnit and AssertJ.

Perform the following steps to develop unit tests using GraphUnit:

1. Open your `MyNeo4jSamples/pom.xml` file and add the following dependencies within the `<dependencies>` section:

```
<dependency>
  <groupId>com.graphaware.neo4j</groupId>
  <artifactId>tests</artifactId>
  <version>2.1.5.25</version>
  <scope>test</scope>
</dependency>
```

2. Add another function by the name of `compareNeo4jDatabase()` and add the following code:

```
@org.junit.Test
public void compareNeo4jDatabase() {
    //Open a Transaction
    try(Transaction transaction=graphDb.beginTx()){
        String testLabel="TestNode";
        //Create a Node with Label in the neo4j Database
        graphDb.createNode(DynamicLabel.label(testLabel));
        transaction.success();
    }

    GraphUnit.assertSameGraph(graphDb,"create (n:TestNode)
    return n");
}
```

The first part of the preceding code, where we create a node in the test Neo4j database remains the same, but the latter part is replaced with a single assertion provided by GraphUnit, which checks whether the given graph is equivalent to the graph created by the provided cypher statement. GraphUnit also provides assertions for testing parts of graphs, that is, subgraphs. Refer to <http://graphaware.com/site/framework/latest/apidocs/com/graphaware/test/unit/GraphUnit.html> for available assertions with GraphUnit.

AssertJ also helps in a similar way. Perform the following steps to use assertions provided by AssertJ:

1. Open your `MyNeo4jSamples/pom.xml` file and add the following dependencies in the `<dependencies>` section:

```
<dependency>
  <groupId>org.assertj</groupId>
  <artifactId>assertj-neo4j</artifactId>
  <version>1.0.0</version>
  <scope>test</scope>
</dependency>
```
2. Add another function by the name of `compareProperties()` and add the following code:

```
@org.junit.Test
public void compareProperties () {
    //Open a Transaction
    try(Transaction transaction=graphDb.beginTx()){
        String testLabel="TestNode";
        String testKey="key";
        String testValue="value";
        //Create a Node with Label in the neo4j test Database
        Node node =
            graphDb.createNode(DynamicLabel.label(testLabel));
        node.setProperty(testKey, testValue);
        transaction.success();
    }

    //Check the Assertion
    assertThat(node).hasLabel(DynamicLabel.label
        (testLabel)).hasProperty(testKey, testValue);
}
```

The preceding code first creates a node with properties in the test Neo4j database and next it uses the assertion provided by AssertJ to check whether the node is successfully created with the provided labels and properties.

The following assertions are provided by AssertJ:

- Node assertions
- Path assertions
- Relationship assertions
- PropertyContainer assertions

In this section, you have discussed and learned the importance of unit testing while working with Neo4j and the various frameworks available for unit testing the code developed for creating and traversing Neo4j graphs.

Let's move ahead to the next section where we will talk about Java APIs exposed and available in Neo4j.

Java APIs

Neo4j exposes a good number of Java APIs that are packaged and categorized based on their usages. Let's see the different categories that are available and how they are organized within Neo4j:

- **Graph database:** The APIs within this category contain various classes for dealing with the basic operations of the graph database such as creating database / nodes / labels, and so on. The following is list of Java packages available with this category:

Java Package	Description
<code>org.neo4j.graphdb</code>	This contains the core APIs to work with graphs such as node / property / label creation
<code>org.neo4j.graphdb.config</code>	This package contains the classes used for modifying or setting the database configurations
<code>org.neo4j.graphdb.event</code>	This package contains the classes used for handling various events such as transaction management or database / kernel events
<code>org.neo4j.graphdb.factory</code>	This contains the factory classes for creating the database
<code>org.neo4j.graphdb.index</code>	This package contains the integrated API for managing legacy indexes on nodes and relationships
<code>org.neo4j.graphdb.schema</code>	This is a new package introduced for creating and managing schema (indexes and constraints) on graphs
<code>org.neo4j.graphdb.traversal</code>	This is a callback-based traversal API for graphs, which provides a choice between traversing breadth- or depth-first

- **Query language:** This contains the classes for executing Cypher queries from Java code. It contains two packages: `org.neo4j.cypher.export` and `org.neo4j.cypher.javacompat`.
- **Graph algorithms:** This defines Java packages and classes for defining and invoking various graph algorithms. It contains only one package, `org.neo4j.graphalgo`.
- **Management:** As the name suggests, it contains JMX APIs for monitoring the Neo4j database. It contains only one package, `org.neo4j.jmx`.



The Enterprise Edition of Neo4j comes with a new package, `org.neo4j.management`, which is used to provide advanced monitoring.

- **Tooling:** This provides the packages and classes for performing global operations over the graph database. It contains one package, `org.neo4j.tooling`.
- **Imports:** This contains the packages and classes for performing batch imports. It contains one package, `org.neo4j.unsafe.batchinsert`.
- **Helpers:** This contains the packages for providing helper classes such as common Java utilities or Iterator/Iterable utilities. It also contains only one package, `org.neo4j.helpers.collection`.
- **Graph matching:** This contains the packages and classes for pattern matching and filtering. It contains two packages: `org.neo4j.graphmatching` and `org.neo4j.graphmatching.filter`.



The Enterprise Edition provides one more package, `org.neo4j.backup`, for performing various backups such as online, cold, and so on, of Neo4j the database.

Graph traversals

Neo4j provides a callback API for traversing the graph based on certain rules provided by the users. Basically, users can define an approach to search a graph or subgraph, which depends upon certain rules/algorithms such as depth-first or breadth-first.

Let's understand a few concepts of traversing:

- **Path expanders:** This defines what to traverse in terms of relationship direction and type
- **Order of search:** The types of search operations are depth-first or breadth-first
- **Evaluator:** This decides whether to return, stop, or continue after a certain point in traversal
- **Starting nodes:** This is the point from where the traversal will begin
- **Uniqueness:** This visits the nodes (relationships and paths) only once

Let's continue our Movie dataset, which we created in *Chapter 3, Pattern Matching in Neo4j*, and create a Java-based traverser for traversing the graph:

```
import org.neo4j.cypher.ExecutionEngine;
import org.neo4j.cypher.ExecutionResult;
import org.neo4j.graphdb.GraphDatabaseService;
import org.neo4j.graphdb.Node;
import org.neo4j.graphdb.Path;
import org.neo4j.graphdb.RelationshipType;
import org.neo4j.graphdb.Transaction;
import org.neo4j.graphdb.factory.GraphDatabaseFactory;
import org.neo4j.graphdb.traversal.Evaluators;
import org.neo4j.graphdb.traversal.TraversalDescription;
import org.neo4j.graphdb.traversal.Traverser;
import org.neo4j.kernel.impl.util.StringLogger;

import scala.collection.Iterator;

public class Traversals {

    //This should contain the path of your Neo4j Database which is
    //generally found at <$NEO4J_HOME>/data/graph.db
    private static final String MOVIE_DB =
        "$NEO4J_HOME/data/graph.db";
    private GraphDatabaseService graphDb;

    public static void main(String[] args) {
        Traversals movies = new Traversals();
        movies.startTraversing();
    }
}
```



```
private void startTraversing() {
    // Initialize Graph Database
    graphDb = new
    GraphDatabaseFactory().newEmbeddedDatabase(MOVIE_DB);

    // Start a Transaction
    try (Transaction tx = graphDb.beginTx()) {

        // get the Traversal Descriptor from instance of GGraph DB
        TraversalDescription trvDesc =
        graphDb.traversalDescription();
        // Defining Traversals need to use Depth First Approach
        trvDesc = trvDesc.depthFirst();

        // Instructing to exclude the Start Position and include all
        others
        // while Traversing
        trvDesc =
        trvDesc.evaluator(Evaluators.excludeStartPosition());
        // Defines the depth of the Traversals. Higher the Integer,
        more
        // deep would be traversals.
        // Default value would be to traverse complete Tree
        trvDesc = trvDesc.evaluator(Evaluators.toDepth(3));

        // Get a Traverser from Descriptor
        Traverser traverser = trvDesc.traverse(getStartNode());

        // Let us get the Paths from Traverser and start iterating
        or moving
        // along the Path
        for (Path path : traverser) {
            //Print ID of Start Node
            System.out.println("Start Node ID = " +
            path.startNode().getId());
            //Print number of relationships between Start and End
            //Node
            System.out.println("No of Relationships = " +
            path.length());
            //Print ID of End Node
            System.out.println("End Node ID = " +
            path.endNode().getId());
        }
    }
}
```

```

    }

    private Node getStartNode() {
        try (Transaction tx = graphDb.beginTx()) {
            ExecutionEngine engine = new
            ExecutionEngine(graphDb, StringLogger.SYSTEM);
            ExecutionResult result = engine
            .execute("match (n)-[r]->() where n.Name=\"Sylvester
            Stallone\" return n as RootNode");
            Iterator<Object> iter = result.columnAs("RootNode");

            return (Node) iter.next();

        }
    }
}

```

Now let's understand the preceding code and discuss the important sections of the code. The code defines three methods:

- `public static void main(..)`: This main method is the entry point of the traversals and defines the flow of code
- `private Node getStartNode()`: This executes the Cypher query and gets the node which is used as a starting point for our traversals
- `private void startTraversing()`: This is the core part of the code that defines the Traversals, rules, and iterates over the paths

Let's deep dive and understand the logic of the `startTraversing()` method.

```
TraversalDescription trvDesc = graphDb.traversalDescription();
```

The preceding line of code fetches an instance of `TraversalDescription` from an instance of the graph database with all its default values. `TraversalDescription` defines the behavior of traversals and provides various methods for defining rules that are further used while traversing graphs. There are two types of traversals: `TraversalDescription` and `BidirectionalTraversalDescription`. Both traversals can be retrieved from the object of `graphDb`.

```

trvDesc = trvDesc.depthFirst();
trvDesc = trvDesc.evaluator(Evaluators.excludeStartPosition());
trvDesc = trvDesc.evaluator(Evaluators.toDepth(3));

```

The preceding statements define the three most common rules that are followed while doing the traversals:

- `depthFirst`: This is rule 1, where the traversals should be evaluating the depth of the tree. You can also define `breadthFirst` for evaluating the breadth of the tree.
- `excludeStartPosition`: This is rule 2 that excludes the starting node and does not evaluate any paths that are originating and ending from the starting point itself.
- `toDepth(3)`: This is rule 3, which defines the number of paths it should evaluate from the starting node. If nothing is specified, then it will evaluate all paths from the start node, that is, it will evaluate the complete graph.

You can also provide the uniqueness filter, which ensures uniqueness among the visited nodes/paths. For example, `trvDesc.uniqueness(Uniqueness.NODE_GLOBAL)` defines that a node should not be visited more than once during the traversing process.

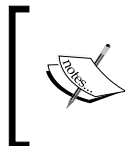
We can also filter on the relationships and its direction. For example, we can add the following piece of code in `Traversals.java` for applying filters on relationships.

Add an Enum in `Traversals.java`:

```
public enum RelTypes implements RelationshipType {  
    ACTED_IN, DIRECTED  
}
```

Then add the following code as the fourth rule for our traversal descriptor:

```
trvDesc.relationships(RelTypes.ACTED_IN, Direction.BOTH);
```



Refer to `org.neo4j.graphdb.traversal.Evaluators` at <http://neo4j.com/docs/2.1.5/javadocs/org/neo4j/graphdb/traversal/Evaluators.html> for understanding the various rules provided by Neo4j.

After defining the rules, we get the instance of traverser from the `TravesalDescription` by providing a starting node and then we start traversing the graph. The instances of traverser are lazy loading, so it's performant even when we are dealing with thousands of nodes, that is, they will not start traversing or scanning until we start iterating over it.

Apart from traversals, Neo4j also provides a graph algorithm factory `org.neo4j.graphalgo.GraphAlgoFactory` that exposes the following prebuilt graph algorithms:

- `GraphAlgoFactory.shortestPath(...)`: This returns the algorithm that finds the shortest path between two given nodes
- `GraphAlgoFactory.allPaths(...)`: This returns the algorithm that finds all the possible paths between two given nodes
- `GraphAlgoFactory.allSimplePaths(...)`: This returns the algorithm that finds all simple paths of a certain length between the given nodes
- `GraphAlgoFactory.dijkstra(...)`: This returns the algorithm that finds the path with the lowest cost between two given nodes
- `GraphAlgoFactory.aStar(...)`: This returns the algorithm that finds the cheapest path between two given nodes
- `GraphAlgoFactory.pathsWithLength(...)`: This returns the algorithm that finds all simple paths of a certain length between two given nodes

In this section, you learned about the various Java APIs and methods exposed by Neo4j for performing traversing/searching over Neo4j graphs.

Summary

In this chapter, you have learned and discussed various deployment models, their applicability, and various tools/ utilities/ frameworks available for performing unit testing in Neo4j. Lastly, we also talked about different Java APIs exposed by Neo4j and its traversal framework for searching and finding data in Neo4j graphs.

In the next chapter, we will discuss the integration of Spring and Neo4j.

6

Spring Data and Neo4j

Ease of development and availability of tools/technologies for **Rapid Application Development (RAD)** is one of the key factors in the adoption of new technologies.

There is always a need for greater productivity and reduced complexity in the area of software development and implementation so that value to the business / stakeholders can be delivered in the shortest possible time. No matter how familiar developers are with the low-level APIs, working with various tools for faster and quicker development is always recommended, so that the focus is on solving business problems and not on the technology.

This has been an underlying theme in a movement to change the way programmers approach for developing web or desktop applications using the Java platform.

Let's introduce Spring as a framework for RAD. Spring provides **Inversion of Control (IoC)** and a **DI-based (Dependency Injection)** simplified programming model that helps developers to focus on the business logic and leave the rest to the framework. It delivers enterprise services such as web services, JMS, database access, and so on by exposing a **POJO-based (Plain Old Java Object)** development, using template programming models, which is naturally reusable in a variety of runtime environments.

In the last chapter, we covered the Java APIs provided by Neo4j, and in this chapter we will discuss the best of both worlds, that is, Spring and Neo4j.

This chapter will cover the following topics:

- Spring Data philosophy
- First steps - Neo4jTemplate
- Spring Data repositories and entities
- Advanced mapping using AspectJ

Spring Data philosophy

One of the most critical functions of any enterprise application is to provide the components or a pattern for performing **Create, Read, Update, and Delete (CRUD)** operations over the enterprise data. Keeping aside the complexity involved in defining the data structures for storing the data, the data access itself has many pitfalls. Starting from managing connections, code for querying, unwrapping results and transforming them into Java objects, and last but not least, handling exceptions is really painful, and over a period of time, it becomes unmanageable and the code starts smelling. Developers are very careful in coding for each of these processes and eventually they end up spending significant time in handling these processes. As a result, there is less time to deal with or define business logic, or it may result in long development cycles.

Over and above, the complete code is very much specific to a database and you are in real trouble if a new data source is introduced or the existing one is replaced!!!

Spring comprehended the severity of the architectural problems with the traditional data access patterns and introduced an additional module *Spring Data access*.

One of the important goals of Spring is to provide a simple programming model that is based on object-oriented principles, where developers need to code for interfaces and not implementations.

Continuing the legacy of Spring, Spring Data access is also based on object-oriented principles and extends/leverages the POJO-based template programming model and provides an abstraction over the complex piece of data access code.

Spring Data access was developed for relational data structures also known as RDBMS, such as Oracle, MySQL, SQL Server, and so on, and provides different templates for using and leveraging various ORM frameworks, such as Hibernate, iBatis, EclipseLink, and so on.

Spring Data access abstracted the underlying implementation of various databases and ORM frameworks from developers and provided data in the form of objects that can be linked to each other by annotations or configurations. Now developers need to make the changes in **Object Oriented Data Model (OODM)**, which will be further persisted within the database by the Spring Framework. OODM is database independent and can be used or re-used with any type of RDBMS or DBMS.

Some of the notable features of Spring Data access are:

- **Session management:** Spring provides efficient, easy, and safe handling of various sessions/connections to the underlying databases. It transparently creates and binds a session, using either a declarative or programmatic approach.
- **Resource management:** Spring contexts easily handle the location and configuration of various ORM frameworks, raw JDBC connections, or session factories.
- **Integrated transaction management:** Spring provides an easy way to handle and declare transactions either declaratively or programmatically. In both cases, frameworks handle all transaction semantics and perform rollback or commit based on the outcome of the query execution.
- **Handling exception:** Spring wraps all database or ORM exceptions into standard exceptions. This allows developers to handle most persistence exceptions, which are non-recoverable, only in the appropriate layers, without annoying boilerplate catches / throws and exception declarations.
- **Vendor lock-in:** This is another benefit of Spring Data access where we can switch or change our underlying database at any point of time without worrying much about the datastore in use.
- **Ease of testing:** Spring's inversion of control makes it easy to create Mock Objects that can be easily configured and used for unit testing of individual pieces of code. This is another benefit when we code for Interfaces and not implementations.

But that was not enough!!!

Spring Data access was only for relational models but there are other non-relational models such as key-value stores, document stores, graph databases, and column stores, which still followed the conventional model of coding.

The rising popularity of NoSQL databases for handling non-relational datasets led developers again into the same problems where they were dealing with the low-level APIs of non-relational data stores, managing connections, code for querying, unwrapping results and transforming them into Java objects, and so on.

Spring again came forward and re-designed the complete data access layer for relational, non-relational data, cloud data, MapReduce, and so on, into a single project called *Spring Data* — <http://projects.spring.io/spring-data/> — for providing a consistent approach to data access.

The objective of Spring Data was to provide a consistent programming model for accessing the data from various type of data stores, irrespective of the format and type of underlying data, which can be in any of these forms: key-value, document, graph, or column.

Spring Data is also referred to as the umbrella project, which contains many subprojects that are specific to a given database. All these subprojects are developed by working together with many of the companies and developers that are behind these new technologies.

Spring Data Neo4j is based on similar principles and is a subproject of Spring Data that enables POJO-based development for the Neo4j graph database by leveraging Spring's familiar template programming model.

Spring Data Neo4j offers annotation-based configuration of entities and then maps them to nodes and relationships within the Neo4j database. It also provides advanced repositories that are built on the Spring repository infrastructure — <http://docs.spring.io/spring-data/data-commons/docs/current/reference/html/#repositories>.

Spring Data Neo4j repositories support annotated and named queries for the Neo4j Cypher Query Language and come with typed repository implementations that provide methods for locating node and relationship entities.

There are several types of basic repository interfaces and implementations as follows:

- `CRUDRepository`: This interface/implementation is the base class that provides basic CRUD operations over Nodes and Relationships
- `IndexRepository` / `NamedIndexRepository`: This interface/implementation leverages Neo4J Indexing APIs to perform searching or scanning of indexes
- `CypherDslRepository`: This interface/implementation provides wrapper and integration with Neo4j APIs for executing Cypher Queries
- `TraversalRepository`: This interface/implementation provides wrapper and integration with the Neo4j traversal API, and also provides convenience methods for searching and traversals
- `RelationshipOperationsRepository`: This interface/implementation performs CRUD operations over relationships
- `SpatialRepository`: This interface/implementation is a special type of repository that provides geographic searches
- `SchemaIndexRepository`: This interface/implementation deals with Neo4j Schema APIs, which was introduced with Neo4j 2.0 and above

Apart from repositories, there are a few more notable features provided by Spring Data Neo4j:

- Support for property graphs
- Object graph mapping via annotated POJO entities
- Advanced mapping mode via AspectJ
- Neo4jTemplate with convenient APIs for performing common operations with exception translation and optional transaction management
- Support for Cypher and Gremlin Query Languages
- Cross-store support for partial JPA – Graph entities
- Support for transparently accessing the Neo4j server via its REST API
- Support for running as extensions in the Neo4j server

In this section, we have discussed the objective and philosophy of Spring Data and various features of Spring Data Neo4j, which helps in quick and faster development.

Let's move forward and learn the basic and advanced usage of Spring Data Neo4j in the upcoming sections.

First step – Neo4jTemplate

Neo4jTemplate, also known as `org.springframework.data.neo4j.support.Neo4jTemplate`, is one of the convenience classes that provide methods for performing CRUD operations over the Neo4j database. It provides functionality that is analogous to `org.neo4j.graphdb.GraphDatabaseService` provided by Neo4j. Apart from CRUD operations, it also provides various methods for searching and querying the Neo4j database.

Let's perform the following steps and set up a Maven-based Spring Data Neo4j project and then use Neo4jTemplate to perform CRUD and search operations:

1. Download Maven 3.2.3 from <http://maven.apache.org/download.cgi>.
2. Once the archive is downloaded, browse the directory on your filesystem and extract it.
3. Define the system environment variable `M2_HOME=<location of extracted Archive file >`.
4. Add `$M2_HOME/bin/` to your `$PATH` variable.

5. Next, open your console and browse the location where you want to create your new maven project and execute the following command:

```
mvn archetype:generate -DgroupId=org.neo4j.spring.samples
-DArtifactId=Spring-Neo4j -DarchetypeArtifactId=maven-archetype
-quickstart -DinteractiveMode=false
```



Using appropriate plugins, you can create maven projects from the IDE itself. For example, in Eclipse, you can install and use M2E plugin
–<http://eclipse.org/m2e/>.

6. It will take some time as maven will download the required dependencies to create your project. You will see a directory `Spring-Neo4j` being created as soon as the command is successfully completed.
7. Next, open your `Spring-Neo4j\pom.xml` file and add the following properties just below the `<name>Spring-Neo4j </name>` tag. These properties hold the version of various dependencies that will be used within the project:

```
<properties>
  <!-- Generic properties -->
  <java.version>1.7</java.version>
  <project.build.sourceEncoding>UTF-8
</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8
</project.reporting.outputEncoding>
  <!-- Spring -->
  <spring-framework.version>4.1.2.RELEASE</spring-
framework.version>
  <spring-data.version>3.2.1.RELEASE</spring-data.version>

  <!-- Hibernate / JPA -->
  <hibernate.version>4.2.0.Final</hibernate.version>
  <hibernate-jpa.version>1.0.0.Final</hibernate-
jpa.version>
  <!-- Logging -->
  <logback.version>1.0.13</logback.version>
  <slf4j.version>1.7.5</slf4j.version>
  <!-- Test -->
  <junit.version>4.11</junit.version>
  <!-- Neo4j version -->
  <neo4j.version>2.1.5</neo4j.version>
</properties>
```

8. Change the version of JUnit to `${junit.version}`.
9. Next, within the `<dependencies>` section, add the following Spring dependencies:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-core</artifactId>
  <version>${spring-framework.version}</version>
</dependency>
```

Repeat the same process and change the value of `<artifactId>` and define the dependency for `spring-context`, `spring-aspects`, `spring-context-support`, and `spring-tx`.

10. Now, let's add the following dependencies for Spring Data Neo4j and Neo4j libraries:

```
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-neo4j</artifactId>
  <version>${spring-data.version}</version>
</dependency>
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-neo4j-aspects</artifactId>
  <version>${spring-data.version}</version>
</dependency>
<dependency>
  <groupId>org.neo4j</groupId>
  <artifactId>neo4j</artifactId>
  <version>${neo4j.version}</version>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-validator</artifactId>
  <version>${hibernate.version}</version>
</dependency>
<dependency>
  <groupId>org.hibernate.javax.persistence</groupId>
  <artifactId>hibernate-jpa-2.0-api</artifactId>
  <version>${hibernate-jpa.version}</version>
  <optional>true</optional>
  <scope>compile</scope>
</dependency>
<dependency>
```

```
<groupId>javax.cache</groupId>
<artifactId>cache-api</artifactId>
<version>1.0.0</version>
</dependency>
```

11. Let's also add dependencies for logging APIs, which are used by Spring under the `<dependencies>` section of `pom.xml`:

```
<!-- Logging with SLF4J & LogBack -->
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  <version>${slf4j.version}</version>
  <scope>compile</scope>
</dependency>
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
  <version>${logback.version}</version>
  <scope>runtime</scope>
</dependency>
```

Our Maven-based Spring Data Neo4j project is ready with all its dependencies. Now we need to create and configure our domain model, which will contain Movies, Actors, and their associations, similar to what we created in *Chapter 3, Pattern Matching in Neo4j*.

Let's perform the following steps to create and configure the domain model using Spring Data Neo4j. All Java source files should be stored at `Spring-Neo4j\src\main\java`.

1. Create an `org.neo4j.spring.samples.domain.Artist.java` class with the following attributes:

```
package org.neo4j.spring.samples.domain;

import org.springframework.data.neo4j.annotation.*;

public class Artist {

    private Long graphId;
    private String id;
    private String name;
    private int year_Of_Birth;
}
```

2. Define the getters and setters for each of the properties defined in `Artist.java` and you get a POJO.
3. Now, let's annotate this class with `@NodeEntity`. This is to indicate and inform the Spring framework that this POJO (`Artist.java`) is an entity of a node, which will be persisted in the Neo4j database.
4. Next annotate the property `graphId` with `@GraphId`. This is to indicate that this field will be used to store the unique node IDs generated by Neo4j.
5. There is another property, `id`, which we have created to store a user-generated unique ID. This is to make sure that we do not depend upon `GraphId` for uniqueness of our nodes. Let's define a unique index on the `id` property by annotating it with `@Indexed(unique=true)`. This is similar to creating a unique index on a node property in the Neo4j database.

The rest of the attributes, that is, `name` and `year_Of_Birth`, would be used to store the name of the artist and his year of birth. The updated `Artist.java` class would look like this:

```
package org.neo4j.spring.samples.domain;

import org.springframework.data.neo4j.annotation.*;

@NodeEntity
public class Artist {
    @GraphId
    private Long graphId;

    @Indexed ( unique=true )
    private String id;

    private String [] workedAs;
    private String name;
    private int year_Of_Birth;
    ...
    //Define getters and setters for all properties.
}
```

6. Next, create a `org.neo4j.spring.samples.domain.Movie.java` class and define the following properties and annotation:

```
package org.neo4j.spring.samples.domain;

import org.springframework.data.neo4j.annotation.*;

@NodeEntity
```

```
public class Movie {
    @GraphId
    private Long graphId;

    @Indexed ( unique=true )
    private String id;
    private String title;
    private int year;
    ...
    //Define getters and setters for all properties.
}
```

7. Now we will define another POJO that will define the relationship between Artist and Movie. Let's create `org.neo4j.spring.samples.domain.Role.java` as follows:

```
package org.neo4j.spring.samples.domain;

import org.springframework.data.neo4j.annotation.*;

@RelationshipEntity(type="ACTED_IN")
public class Role {

    @GraphId
    private Long graphId;

    @StartNode
    private Artist artist;
    @EndNode
    private Movie movie;

    private String role_name;
    //Default No Argument Constructor
    public Role(){}

    //Argument constructor to set the properties
    public Role(Artist artist, Movie movie, String
    role_name){
        //Invoke setters methods of artist, movie and role_name
    }
    ...
    //Define getters and setters for all properties.
}
```

In the preceding code, we have annotated the relationship class with `@RelationshipEntity` and the type of relationship will be `ACTED_IN`, which indicates that this bean is defining the relationship between two nodes, namely Artist and Movies. We have also specified attributes for Artist and Movie and annotated them with `@StartNode` and `@EndNode`, which defines the `START` and `END` of the relationship between these two nodes.

Until now, we have created two nodes and one relationship class, which defines the relationship between the nodes. Now it is time to configure these domain entities so that the framework can identify and process these entities. So let's define the Spring configuration file `Spring-Neo4j/src/main/resources/Application-Config.xml`.

1. Define the following schemas in your `Application-Config.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:neo4j="http://www.springframework.org/schema/data/neo4j"
xmlns:repository="http://www.springframework.org/schema/data/repository"
xmlns:tx="http://www.springframework.org/schema/tx"
xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-
3.2.xsd
http://www.springframework.org/schema/data/neo4j
http://www.springframework.org/schema/data/neo4j/spring-
neo4j-2.2.xsd
http://www.springframework.org/schema/data/repository
http://www.springframework.org/schema/data/repository/sprin
g-repository-1.7.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-
context-3.1.xsd
">
</beans>
```


2. The preceding XML configuration defines the schema for Spring context, Neo4j, and Spring transactions. Next, between the `<beans></beans>` tags define the following:

```
<!-- This is declared to activate annotations in beans
already registered in the application context (no matter if
they were defined with XML or by package scanning). This
instructs the framework that the classes are using
annotations and Spring should automatically wire values
into properties, methods, and constructors.
-->
<context:annotation-config/>

<!-- enable the configuration of transactional behavior
based on annotations -->

<tx:annotation-driven mode="proxy"/>

<!-- Neo4j Specific configuration, where first parameter is
the directory which defines the location of the database
and second parameter is the directory/ package which will
contain the domain classes.
-->

<neo4j:config storeDirectory="data/spring.data" base-
package="org.neo4j.spring.samples"/>
```



Follow the comments provided within the preceding configuration file to understand the purpose and role of various tags.

3. Next we need a class that can populate our domain objects and persist in the database. Let's create `org.neo4j.spring.samples.PersistNeo4JData` and configure it as a bean with the `Application-Config.xml` file. Edit your `Application-Config.xml` file and add the following code between the `<beans></beans>` tags:

```
<bean id = "persistNeo4JData"
class="org.neo4j.spring.samples.PersistNeo4JData"></bean>
```

4. Now, let's write some persistence code in the `PersistNeo4JData.java` file as shown in the following code:

```
package org.neo4j.spring.samples;

import org.neo4j.spring.samples.domain.*;
```

```
import org.springframework.beans.factory.annotation.*;
import org.springframework.data.neo4j.support.*;
import org.springframework.transaction.annotation.*;

public class PersistNeo4JData {

    @Autowired
    private Neo4jTemplate neo4jTemplate;

    @Transactional
    public void addData() {
        // Create Movie
        Movie rocky = new Movie();
        rocky.setTitle("Rocky");
        rocky.setYear(1976);
        rocky.setId(rocky.getTitle() + "-"
            +String.valueOf(rocky.getYear()));

        // Create Artists
        Artist artist = new Artist();
        artist.setName("Sylvester Stallone");
        artist.setWorkedAs(new String[] { "Actor", "Director" });
        artist.setYear_Of_Birth(1946);
        artist.setId(artist.getName() + "-"
            +artist.getYear_Of_Birth());

        // create Relationship
        Role role = new Role(artist, rocky, "Rocky Balboa");

        neo4jTemplate.save(artist);
        neo4jTemplate.save(rocky);
        neo4jTemplate.save(role);

    }
}
```

In the preceding code, we have defined the `addData()` method, which creates the object of `Movie` and `Artist`, provides appropriate values to its properties, and then defines an Object of `Role`, which further defines the relationship between `Artist` and `Movie`. The `addData()` method is annotated with `@Transactional`, which will inform the Spring framework to start a new transaction while executing this method.

Easy and clean isn't it?

We have only written the business logic; all the boilerplate code such as wrapping your Neo4j code within a transaction will be handled by the framework.

The interesting part is the declaration of:

```
@Autowired
private Neo4jTemplate neo4jTemplate;
```

Neo4jTemplate, as we explained in the beginning of this section, is the convenience class that provides methods to perform CRUD operations over the Neo4j database. @Autowired instructs the Spring Framework to automatically create (or provide reference of the already created Object of Class Neo4jTemplate) the Object of Neo4jTemplate and assign its reference to the given variable.

Next, using Neo4jTemplate, we first persist Artist, then Movie, and finally Role, which defines the association between Artist and Movie:

```
neo4jTemplate.save(artist);
neo4jTemplate.save(rocky);
neo4jTemplate.save(role);
```

5. Lastly, we need a main class that can bring up our Spring framework and invoke addData(). Let's define org.neo4j.spring.samples.MainClass.java and add following code:

```
package org.neo4j.spring.samples;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.
ClassPathXmlApplicationContext;

public class MainClass {

String[] configs = { "classpath:Application-Config.xml" };

public static void main(String[] args) {
    MainClass mainClass = new MainClass();
    mainClass.setup();
}

public void setup(){
    //Read the Spring configuration file and bring up the
    Spring framework, so that all required beans can be
    initialized.
    ApplicationContext context = new
    ClassPathXmlApplicationContext(configs);
    System.out.println("Spring Context Created");
}
```

```

//Get the Object of PersistNeo4JData from the Spring
Context.
PersistNeo4JData data = (PersistNeo4JData)
context.getBean("persistNeo4JData");
//Invoke addData() method to persist our neo4j nodes and
relationships.
data.addData();

}
}

```

And we are done !!! Now, in the next step, let's execute our code.

6. Open your console, browse the root directory of your project, that is, Spring-Neo4j, and execute the following Maven commands:

```

mvn install

mvn exec:java -Dexec.cleanupDaemonThreads=false -Dexec.
mainClass="org.neo4j.spring.samples.MainClass"

```



Suppress DEBUG logs generated by the Spring framework by adding a new file — Spring-Neo4j\src\main\resources\logback.xml. The content of this new file should be similar to <https://github.com/qos-ch/logback/blob/master/logback-examples/src/main/java/chapters/configuration/sample0.xml>. Now change <root level="debug"> to <root level="warn">.

Wait for the successful execution and completion of the preceding Maven commands and you will see that the data is persisted in your Neo4j database. To verify whether we have successfully persisted the data, open your console and execute the following command to open the Neo4j shell — \$NEO4J_HOME/bin/neo4j-shell -path <path of Database file where we created the initial data>—and execute the following Cypher query:

```

Match (n) -[r]->(n1) return n.name as Artist,type(r) as Relationship,
n1.title as Movie;

```

The output is shown in the following screenshot:

```

neo4j-sh (?)$ Match (n)-[r]->(n1) return n.name as Artist,type(r) as Relationship, n1.title as Movie;
+-----+-----+-----+
| Artist | Relationship | Movie |
+-----+-----+-----+
| "Sylvester Stallone" | "ACTED_IN" | "Rocky" |
+-----+-----+-----+
1 row
2158 ms
neo4j-sh (?)$

```

We can also verify the same by enhancing our Spring Data Neo4j code and adding some methods for searching the Neo4j database.

Let's add a new method `searchData()` in `PersistNeo4JData` and add the following code in this new method:

```
@Transactional
public void searchData() {
    System.out.println("Retrieving Roles from Database");
    Result<Role> result =
neo4jTemplate.findAll(org.neo4j.spring.samples.domain.Role.class);
    while (result.iterator().hasNext()) {
        Role role = result.iterator().next();
        System.out.println(role.toString());
    }
    System.out.println("Artists and Roles retrieved from Database");
}
```

Next, import `org.springframework.data.neo4j.conversion.*` as a new package in `PersistNeo4JData` and invoke the `searchData()` method from `MainClass.java`.



In order to see some meaningful values on the console, add a `toString()` method in `Role.java`, which returns a formatted string containing the values of the attributes.

The preceding code will fetch the value of the attributes defined directly in `Role.java`, but it will not fetch the values of the referenced objects such as `Artist` and `Movie`.

In order to fetch the values of the referenced objects, we need to define the `@Fetch` annotation for the referenced object, so that Spring can eagerly load the values of the referenced objects. By default, relationships do not fetch the values unless we ask them to do so. This is also beneficial in scenarios where we have large graphs and do not want to load everything in memory.

For example, `Role.java` will be modified to eagerly load the object's data—`Artist` and `Movie`—as shown in the following code:

```
package org.neo4j.spring.samples.domain;

import org.springframework.data.neo4j.annotation.*;

@RelationshipEntity(type="ACTED_IN")
public class Role {
```

```

    @StartNode @Fetch
    private Artist artist;
    @EndNode @Fetch
    private Movie movie;
    private String roleName;
    ...
    //Define getters and setters for all properties.
}

```

There are two more important annotations which can be used with our domain model / entities:

- **@RelatedTo:** This annotation is used for the fields to define the type of relationship within the node entity. For example, we can define a collection in `Artist.java` and define the relationship with the `Movie` object there itself:

```

@RelatedTo(type="ACTED_IN", direction=Direction.OUTGOING)
private Set<Movie> movies;

```

- **@RelatedToVia:** This annotation is used for the fields to define the type of relationship within the node entity. It provides the read-only iterator for iterating over the related entities fetched while reading data from the database. For example, we can define a collection in `Artist.java` and define the relationship with the `Movie` object, so that when we read the `Artist` object, we also read the related `Movies`:

```

@RelatedToVia(type="ACTED_IN", direction=Direction.OUTGOING)
private Iterable<Role> roles;

```

It can be applied only to the classes that are annotated with `@RelationshipEntity`.

`Neo4jTemplate` defines various convenience methods for performing CRUD and search operations. Refer to <http://docs.spring.io/autorepo/docs/spring-data-neo4j/3.2.x/api/org/springframework/data/neo4j/support/Neo4jTemplate.html> for complete details about the other methods exposed by `Neo4jTemplate`.

Spring Data repositories and entities

The objective of Spring Data is to provide an abstract layer of interfaces to developers and hide all complexity of accessing data from the underlying data store. Spring Data repositories were introduced for the same cause and it exposes a generic interface `org.springframework.data.repository.Repository` which is extended and implemented by the domain-specific repositories.

Spring Data Neo4j was designed and developed to extend the same concept and provides a generic repository interface `org.springframework.data.neo4j.repository.GraphRepository`, which implements `CRUDRepository`, `IndexRepository`, `SchemaIndexRepository`, and `TraversalRepository`.

Developers can extend the generic repository interface and can define repositories specific to their domain models, which can contain only domain-specific methods or operations and all generic and common methods for performing CRUD operations or traversing or indexing will be injected/provided by the framework.

Let's extend our Spring Data Neo4j example that we created in the previous section and create repositories for the search operation:

1. Create a new interface `org.neo4j.spring.samples.repositories`.

MovieRepository that extends `GraphRepository`:

```
package org.neo4j.spring.samples.repositories;
```

```
import org.neo4j.spring.samples.domain.Movie;
import org.springframework.data.neo4j.repository.GraphRepository;
import org.springframework.stereotype.Repository;
```

```
@Repository
```

```
public interface MovieRepository extends GraphRepository<Movie>{
}
```

We have annotated our repository with `@Repository` so that the Spring framework can consider it as a Spring managed repository and inject all the required implementations of common CRUD operations, traversing and indexing.

2. Now, let's modify our `Application-Config.xml` file and add the following piece of configuration anywhere between the `<beans></beans>` tags:

```
<neo4j:repositories base-package="org.neo4j.spring.samples.
repositories"/>
```

The preceding configuration provides the framework with the location and package of user-defined repositories. Spring framework will further scan the provided package, that is, `org.neo4j.spring.samples.repositories` and search for all interfaces that are marked with `@Repository` and will inject the implementation required for performing CRUD and other search operations.

3. Next, let's enhance `PersistNeo4JData` and define the `getAllMovies()` method and add `@Autowired MovieRepository movieRepo;` as an instance variable. `@Autowired` will instruct the Spring framework to automatically create the object of implementation provided for the `MovieRepository` interface—or its parent interface, that is, `GraphRepository`—and assign its reference to the given variable.
4. Import `java.util.Iterator` as an additional package in `PersistNeo4JData`.
5. Let's see the implementation of the `getAllMovies()` method that is very simple and straightforward:

```
@Transactional
public void getAllMovies() {

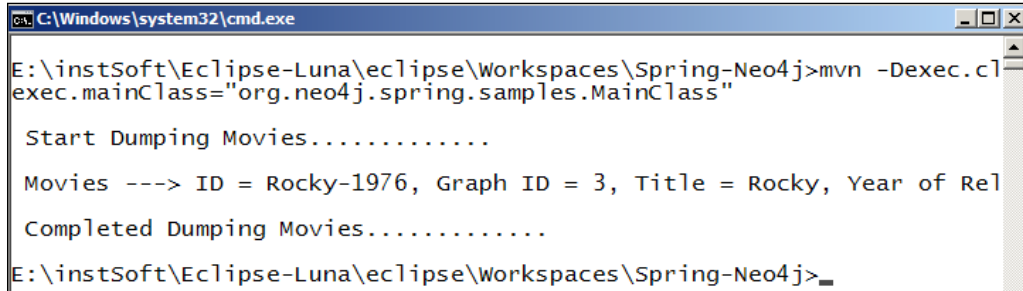
    Iterator<Movie> iterator = movieRepo.findAll().iterator();
    System.out.println("\n Start Dumping
    Movies.....\n");
    while (iterator.hasNext()) {
        Movie movie = iterator.next();
        System.out.println(" Movies ---> ID = " + movie.getId() +
            ", Graph ID = " + movie.getGraphId() + ", Title = " +
            movie.getTitle() + ", Year of Release = " +
            movie.getYear());
        System.out.println("\n Completed Dumping of
        Movies.....\n");
    }
}
```

In the preceding method, we are fetching all movies from our Neo4j database and then printing it on the console. The implementation of the `findAll()` method will be injected by Spring and added to our `MovieRepository`. For other available methods, refer to <http://docs.spring.io/autorepo/docs/spring-data-neo4j/3.1.0.RC1/api/org/springframework/data/neo4j/repository/CRUDRepository.html>.

6. Next, invoke this new method from `MainClass.java`; open your console and execute the following Maven command:

```
$M2_HOME/bin/mvn -Dexec.cleanupDaemonThreads=false -q exec:java -D
exec.mainClass="org.neo4j.spring.samples.MainClass"
```


The output after execution of the preceding command is shown in the following screenshot:



```
C:\Windows\system32\cmd.exe
E:\instSoft\Eclipse-Luna\eclipse\Workspaces\Spring-Neo4j>mvn -Dexec.c
exec.mainClass="org.neo4j.spring.samples.MainClass"

Start Dumping Movies.....
Movies ---> ID = Rocky-1976, Graph ID = 3, Title = Rocky, Year of Rel
Completed Dumping Movies.....
E:\instSoft\Eclipse-Luna\eclipse\Workspaces\Spring-Neo4j>
```

We can also declare some custom methods that are based on the results of custom Cypher queries, which we might have written to traverse our domain model.

For example, we can declare the following method in our `MovieRepository.java` class and then replace `movieRepo.findAll()` with `movieRepo.getAllMovies()` in `PersistNeo4JData.java`:

```
@Query("match (n) return n;")
public Iterable<Movie> getAllMovies();
```

We have annotated the preceding method with the `@Query` annotation, which will instruct the framework to execute the given query and provide the results back to users via `Iterable<Movie>`. So whenever the user invokes `getAllMovies()`, the framework will execute the given Cypher query and provide the results.

The results will be the same, but the point here is when we did not provide any implementation of either of the methods, then who provided them? The answer is, Spring Data does that for you. All the boilerplate code has been automatically injected into your repositories and you just need to focus on your domain model and queries.

In this section, we have explored the Spring Data repositories and how they can be used to avoid all boilerplate code; thus, developers need to focus only on the business logic and the domain-specific queries, which not only results in faster development cycles but also provides a much cleaner and readable code.

Let's move further and discuss advanced mapping of entities with AspectJ.

Advanced mapping mode – AspectJ

In the previous sections, we have used simple mapping mode for performing CRUD and search operations over entities. Spring Data also provides "Advanced mapping mode" with AspectJ for modeling and working with Spring Neo4j entities.

Spring Data Neo4j heavily depends on AspectJ and especially the advanced mapping mode, which is nothing more than the dynamic instrumentation of Java classes provided by AspectJ.

AspectJ is an API that provides implementation of aspect-oriented programming in Java language. **Aspect-oriented programming (AOP)** aims to increase the modularity and re-usability by separation of cross-cutting concerns.

It helps developers to isolate the auxiliary and cross-cutting concerns such as security, logging, and so on from the business logic and helps components to focus on core concerns.

AOP defines the following main concepts:

- **Cross-cutting concerns:** Common code or secondary requirements / needs for various components within the system such as security, logging, and so on. These secondary requirements remain the same for all components within a system.
- **Advice:** An advice is a code which is developed to implement the secondary requirement.
- **Pointcut:** This is the place where we want to apply the advices.
- **Aspect:** A combination of the pointcut and the advice is termed as an aspect.

An application developer could find difficulty in implementing the pointcut language, but developers working with Spring or Spring Data do not have to worry about that. Spring does all heavy loading and provides implementation of various cross-cutting concerns and further exposes it to developers via annotations.

Spring Data Neo4j leverages the concept of aspect-oriented programming and implements advices and exposes these advices to users via annotations.

For example, Spring leverages AspectJ and scans the Java packages and classes, and as soon as it encounters annotations such as `@NodeEntity`, it introduces a new interface called `org.springframework.data.neo4j.aspects.core.NodeBacked` or `org.springframework.data.neo4j.aspects.core.RelationshipBacked` for `@RelationshipEntity`.

Further, these new interfaces are implemented by the aspects provided by Spring. For example, `NodeBacked` is implemented by `Neo4jNodeBacking` and `RelationshipBacked` is implemented by `Neo4jRelationshipBacking`.

You can find all these aspects under `org.springframework.data.neo4j.aspects` and the source can be downloaded from GitHub at <https://github.com/spring-projects/spring-data-neo4j/tree/master/spring-data-neo4j-aspects/src>.

All these aspects are injected as dependencies to the backing interfaces, and in some cases, calls to methods are intercepted and delegated to the specific implementations of the graph databases. Spring also introduces new methods and properties to the annotated classes.

Perform the following steps to enable advanced mapping mode in your Spring project:

1. Edit your `Spring-Neo4j/pom.xml` and add the following Maven plugin to include the AspectJ compilation just before the `</project>` tag:

```
<plugins>
  <build><plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>aspectj-maven-plugin</artifactId>
    <version>1.4</version>
  </build>
  <configuration>
    <outxml>true</outxml>
    <aspectLibraries>
      <aspectLibrary>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aspects</artifactId>
      </aspectLibrary>
      <aspectLibrary>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-neo4j-aspects</artifactId>
      </aspectLibrary>
    </aspectLibraries>
    <source>1.7</source>
    <target>1.7</target>
  </configuration>
  <executions><execution>
    <goals><goal>compile</goal>
    <goal>test-compile</goal>
  </goals> </execution> </executions>
  <dependencies> <dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>${aspectj.version}</version>
  </dependencies>
</plugins>
```

```

</dependency> <dependency>
<groupId>org.aspectj</groupId>
<artifactId>aspectjtools</artifactId>
<version>${aspectj.version}</version>
</dependency> </dependencies>
</plugin> </plugins>
</build>

```

2. Next, open your `PersistNeo4JData.java` and replace `neo4jTemplate.save(artist);` with `((NodeBacked) artist).persist();` and also import `org.springframework.data.neo4j.aspects.core.NodeBacked`. You can also replace the calls to the `Role` object with `RelationshipBacked` in the same manner as we did for nodes.

The modifications made in the preceding steps assume that your project and entities are already compiled and enhanced with the AspectJ compiler. So in the next step, recompile your project by executing the following Maven command:

```
mvn clean install
```

And we are done!!! You can execute your project as you were doing it earlier and there should be no changes in the output.

All your domain entities (.class files) such as `Artist`, `Movie`, and `Role` are enhanced and modified to use the advices provided by the Spring Neo4j aspects.

Now you can also use the methods provided by the `NodeBacked` interface at <http://docs.spring.io/spring-data/neo4j/docs/current/api/org/springframework/data/neo4j/aspects/core/class-use/NodeBacked.html>.

AspectJ performs the byte code instrumentation in three different modes:

- **Compile-time weaving:** Instrumentation is performed at compile time itself using **AspectJ compiler (AJC)**
- **Load-time weaving:** Instrumentation is performed as the classes are loaded in memory
- **Run-time weaving:** Instrumentation is performed when the objects are created in memory

As of now, Spring Data only supports compile-time weaving, which means all classes should be compiled with the AJC.

In this section, we explored the advanced mapping mode of Spring Data Neo4j entities using AspectJ. We also enhanced our examples and enabled the aspects in our examples by introducing AJC in our build process and further used the modified libraries / classes in our example.

Summary

In this chapter, you learned about the role of Spring / Spring Data as a framework for quick and fast development of Neo4j-based applications. You learned about the philosophy and evolution of Spring Data and also implemented examples using Spring Data and Neo4j for simple mapping and advanced mapping.

In the next chapter, we will discuss Neo4j architecture and deployment. We will also talk about monitoring options available for Neo4j deployments.

7

Neo4j Deployment

Designing a scalable and distributed software deployment architecture is another challenge for developers/architects. Development teams are constantly striving to deploy software in such a way that various enterprise concerns such as maintenance, backups, restores, and disaster recovery are easier to perform and flexible enough to scale and accommodate future needs.

However, application software does provide various deployment options, but which one to use and how they are used, will largely depend upon the end users. For example, you may have more reads being performed as compared to writes or vice versa. So your deployment architecture would need to support and be optimized either for a high volume of reads or high volume of writes, or both.

Scalability is another aspect that is closely linked to the deployments. In simple terms, it defines a feature of a software where it can support x number of user requests by adding more nodes in a cluster (horizontal scalability or scale out) or by upgrading the existing hardware by adding more processing power CPUs or memory – RAM (vertical scalability). Now, which one to use, without impacting the SLAs, is again a challenging question that will require pushing the software to its limits, so that you can understand its behavior in extreme / rare circumstances and then define an appropriate strategy / plan for your production deployment. We should also remember that the production deployments are evolving and will change over time due to many reasons such as new versions of software, innovation in hardware, changes in user behavior, and so on.

In this chapter, we will discuss the Neo4j deployment scenarios and also talk about recommended setup and monitoring. This chapter will cover the following topics:

- Neo4j architecture and advanced settings
- Cluster mode – principles and recommended setup
- Monitoring

Neo4j architecture and advanced settings

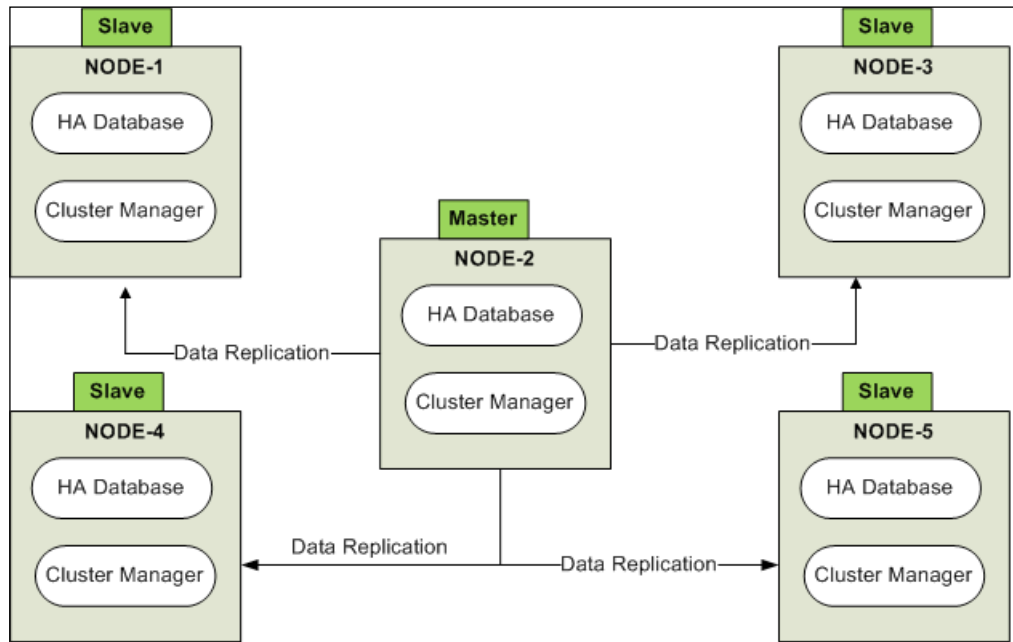
Neo4j architecture, which is also known as Neo4j HA architecture, provides a robust and scalable architecture that fits best towards the needs of enterprises. It mainly provides the following features:

- High Availability (HA) and linear scalability
- Fault tolerance
- Data replication and data locality
- Backup and recovery

Let's discuss the preceding features provided by Neo4j HA architecture.

High Availability and linear scalability

Neo4j HA architecture provides clustering of Neo4j servers and implements a master-slave architecture.



As shown in the preceding diagram, each Neo4j server instance has two parts: one is the Neo4j HA database and the other is the cluster manager. Neo4j HA database is responsible for storing and retrieving the data, and cluster manager is responsible for ensuring a HA and fault-tolerant cluster.

Neo4j HA database directly communicates with the other instances for data replication with the help of cluster manager.

The following are the features provided by cluster manager:

- Maintaining and tracking of live and dead nodes
- Enabling data replication from the master node by polling at regular intervals
- Electing the master node
- Exposing system health / monitoring via JMX beans

Neo4j architecture is a self-driven and independent architecture and requires minimum human intervention or management.

All the nodes in a cluster are self-sufficient and any node, either slave or master, can be used to read data from the database. Writing is an exception where all writes have to be persisted or relayed through the master node. Although you can still connect to the slave nodes and relay your write requests, before returning success, the slave nodes will direct all requests to the master node, and once the master node confirms the success, only then will the client receive a success response. So, it is always advisable to relay all writes to the master node, and reads can be performed from any node.

Neo4j also provides linear scalability where we can add nodes to the existing cluster and data is asynchronously replicated to the new nodes.



To enable HA / cluster mode, modify the value of `org.neo4j.server.database.mode` as HA in `<$NEO4J_HOME>/conf/neo4j-server.properties`.

Fault tolerance

Neo4j provides the custom implementation of multi-paxos paradigm at http://en.wikipedia.org/wiki/Paxos_%28computer_science%29#Multi-Paxos, which provides the following features:

- Cluster management keeps a track of the leaving or joining nodes. It checks the heartbeat of the other participating nodes in a cluster and keeps a track of the last sync and its availability.
- Message broadcasting and replication.
- Electing and choosing master nodes.
- Assisting Neo4j HA database in transaction propagation and data replication.



You can also read more about Paxos from <http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf>.

Although reads can be served even with a single node, when it comes to writes, it is essential to have a Quorum of nodes available in a cluster.

Whenever a Neo4j database instance becomes unavailable, due to any reason such as hardware or network failure, the other database instances in the cluster will detect that and mark it as temporarily failed.

Once the temporarily failed instance is available and ready to serve user requests, it is automatically synced with the other nodes in the cluster.

If the master goes down, then another (best-suited) member will be elected and have its role switched from slave to master after a quorum has been reached within the cluster. Writes are blocked during the election process of the master node.

For all those cases where Quorum is not available and we still want to elect the master, Neo4j provides arbiter nodes that can be deployed to achieve the Quorum. Arbiter nodes do not contain the data and neither serve the read or write requests. They are used only in the election of the master node with the single purpose of breaking ties.

Arbiter instances are configured in the same way as Neo4j HA members are configured in the `neo4j.properties` file under `conf/` and the following command is used to start an arbiter instance:

```
<$NEO4J_HOME>/bin/neo4j-arbiter start
```

In cases where the new master performs some changes to the data, before the old master recovers, there will be two branches of the database after the point where the old master became unavailable. The old master will move away from its database (its branch), will download a full copy from the new master, would be marked as available, and will also be added as a slave node in the cluster.

Data replication and data locality

Neo4j HA architecture asynchronously replicates the data to other nodes in a cluster. All write operations are first performed by the master node and then the slave nodes are synchronized or they poll the new data from the last checkpoint from the master node. The behavior of data replication is driven by the following properties defined at `<$NEO4J_HOME>/conf/neo4j.properties`:

- `ha.pull_interval`: This is the interval at which slaves will pull updates from the master. The unit is in seconds.
- `ha.tx_push_factor`: This is the amount of slaves node the master will try to push a transaction before returning success to the client. We can also set this to 0, which will switch off the synchronous data writes to slave node and would eventually increase the write performance, but would also increase the risk of data loss, as the master would be the only node containing the transaction.
- `ha.tx_push_strategy`: It should be either fixed or round robin. This means the priority of nodes that will be selected to push the events. In case of fixed, the priority is decided based on the value of `ha.server_id`, which is further based on the principle of highest first.

All write transactions on a slave will be first synchronized with the master. When the transaction commits, it will be first committed on the master, and if successful, then it will be committed on the slave. To ensure consistency, the slave has to be updated and synchronized with the master before performing a write operation. This is built into the communication protocol between the slave and the master so that updates are applied automatically to a slave node communicating with its master node.

Neo4j provides full data replication on each node so that each node is self-sufficient to serve read/write requests. It also helps in achieving low latency. In order to serve the global audience, additional Neo4j servers can be configured as read-only slave servers and these servers can be placed near the customer (maybe geographically). These slave read-only servers are synced up with the master in real time and all the local read requests are directed and served by these read-only slave servers, which provide data locality for our client applications.

Backup and recovery

Backup and recovery is another challenge for distributed systems and Neo4j HA architecture provides various tools and utilities for performing online backup and recovery, which is in sync with the enterprise's operational needs.

Neo4j provides `<$NEO4J_HOME>/bin/neo4j-backup` as a command-line utility for performing the Full and Incremental / hot backups.

The following steps need to be performed to enable backups:

1. Enabling online backup: `online_backup_enabled` should be enabled in `<$NEO4J_HOME>/conf/neo4j.properties`.

2. Full backup: Create a blank directory on the machine where you want to take the full backup and run the backup tool `<$NEO4J_HOME>/bin/neo4j-backup -host <IP-ADDRESS> -port <PORT#> -to <DIR location on remote server>`.
3. Incremental backup: Run the same command that we used to take full backup and `neo4j-backup` will only copy the updates from the last backup. Incremental backups can fail in case the provided directory does not have valid backup or previous backup is from the older version of the Neo4j database.
4. Recovering database from backup: Modify the `org.neo4j.server.database.location` property in `<$NEO4J_HOME>/conf/neo4j-server.properties` and provide the location of directory where backup is stored and restart your Neo4j server.



Online backup can also be performed programmatically using Java by using `org.neo4j.backup.OnlineBackup.java`.

Advanced settings

Let's discuss the advanced settings exposed by Neo4j, which should be considered for any production deployment:

<\$NEO4J_HOME>/conf/neo4j-server.properties - Neo4j server configuration file		
Parameter	Default value	Description
<code>org.neo4j.server.webserver.address</code>	0.0.0.0	Client accept pattern for the web server. By default it accepts connection only from local boxes. Define the IP address of the box for accepting remote connections.
<code>org.neo4j.server.webserver.maxthreads</code>	200	This controls the concurrent request handled by the web server.
<code>org.neo4j.server.transaction.timeout</code>	60	Timeouts for orphaned transactions.

<\$NEO4J_HOME>/conf/neo4j.properties - low-level configurations for Neo4j database		
Read_only	False	This is the Boolean value for enabling READ mode or WRITE mode. By default it is in WRITE mode.
Cache_type	soft	<p>This defines the type of cache used to store the nodes and relationships. The following are the types of cache:</p> <ul style="list-style-type: none"> • none: Do not use any cache. • soft: LRU cache using soft references. Used when the application load isn't very high. • weak: LRU cache using weak references. Used when the application is under heavy load with lots of reads and traversals. • strong: Will hold on to all data that gets loaded and never release it. Use it when your graph is small enough to fit in memory. • hpc: High Performance Cache, is only available with the Enterprise version of Neo4j.
dump_configuration	False	This logs all configuration parameters at the time of server start up.
query_cache_size	100	This is the number of Cypher query execution plans to be cached.
keep_logical_logs	7 Days	This is the logical transaction logs to back up the database. It is used to specify the threshold to prune logical logs.
logical_log_rotation_threshold	25M	This is the size of the file that will be used to autorotate the logical logfile after a certain threshold. Value of 0 means there is no rotation based on the size of the file.

Apart from the properties mentioned in the preceding table, we can also configure and tune the JVM by defining the configurations such as type of GC, GC Logs, Max and Min memory, and so on in `<$NEO4J_HOME>/conf/neo4j-wrapper.conf`.

To summarize, Neo4j HA architecture meets the enterprise needs in a true sense where it provides the highest degree of fault tolerance, can operate even with a single machine, and can expose various configurations, which help in tuning our database to produce maximum throughput.

Neo4j cluster – principles and recommended setup

Neo4j HA cluster can be setup/tuned and optimized for accommodating various kinds of requirements such as fault tolerance, load (read versus write) availability, and so on.

The bad news is that there is no single setup that can meet all these needs, but the good news is that we do have a solution to meet the requirements individually.

Let's discuss different aspects/principles of the Neo4j cluster and its recommended setup for various scenarios.

Scaling write throughputs

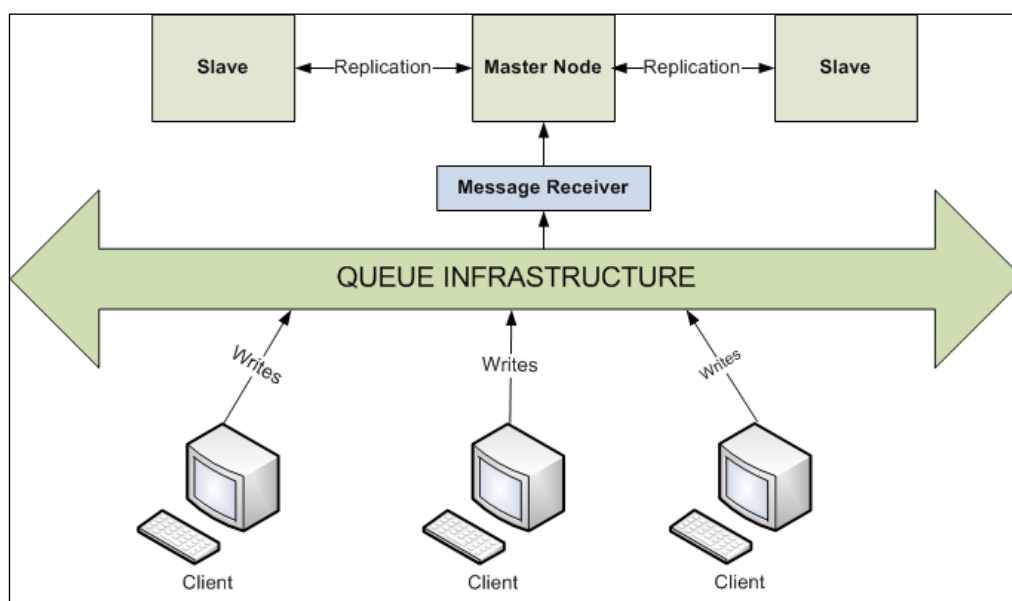
As we discussed earlier, Neo4j provides a master-slave architecture, where all writes are served by the master and then eventually they are replicated to the slave nodes in an asynchronous mode. Although we can perform writes on slave nodes too, slave nodes eventually will contact the master node before returning the success response to the master node.

It is always advisable that all writes are directly written to the master nodes for better performance, though it also means that write throughput is limited to a single machine. Despite this limitation of a single machine, write throughput can still be very high by introducing the upcoming strategies.

Introducing queues for write operations

We need to realize that the master node plays a very important role in the cluster setup, and at any point of time, we may not expose our master node directly to our clients for any kind of operations, be it read or write.

Especially in cases where we need high throughput from our write operations, it is recommended to introduce a queuing solution so that the cluster can service a steady and manageable stream of write operations. It will also rescue us from losing any write transactions, which may occur in extreme situations where our master node abruptly goes down.



Batch writes

We talked about batch writes and the optimization techniques in *Chapter 2, Ready for Take Off*.

We should remember that Neo4j provides ACID properties and every request to Neo4j is wrapped around a transaction, which also induces some overhead. Planning your writes and submitting it to the server in batches using a low-level API like `BatchInserter` would help in directly interacting or inserting data into Neo4j and avoiding all overhead-like transactions. It would definitely help in achieving higher throughput.

Vertical scaling

Vertical scaling or upgrading the hardware of the master node is another option where we can scale our write operations. Neo4j cluster does support SSD drives such as Fusion-io, which does provide exceptional high write performance.

Tuning Neo4j caches

Neo4j provides two kinds of caches: File Buffer Cache and Object Cache.

File Buffer Cache is specifically introduced to improve write performance. It enables and persists all write operations on cache, deferring durable writes until the logical log is rotated. It is advised to tune and optimize both caches as per your available hardware and memory.

Scaling read throughputs

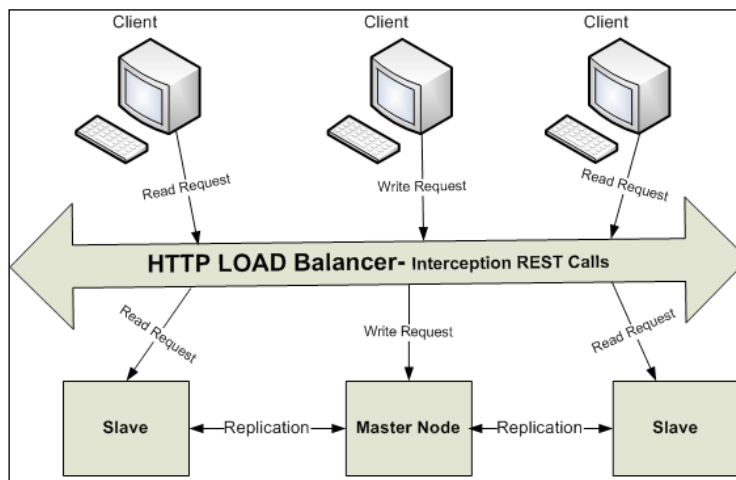
Unlike the write operation where only the master node is eligible for writing, reads can be performed locally and can be served by any node in the cluster. Read operations in HA Neo4j cluster increase linearly where any node in a cluster can serve the read request and new nodes can be added at any point of time.

Let's discuss strategies, which if implemented, can provide desirable throughputs for read operations.

Load balancer

Neo4j HA cluster does not provide any load balancing that can distribute the load over the nodes in a cluster, but we can introduce a software or hardware load balancer for distributing the load equally over the various nodes in the cluster.

For example, we can introduce a software load balancer such as HA proxy—<http://www.haproxy.org/> or Apache Proxy—http://httpd.apache.org/docs/2.2/mod/mod_proxy.html, which would intercept REST calls, and based on routing rules, can delegate the request to the appropriate node in a cluster.



Apart from distributing the load across the cluster, the following are the other benefits of introducing a load balancer to our Neo4j cluster:

- The client does not have to be aware of the location and address of the physical nodes of the cluster
- The nodes can be removed or added at any point of time, while the customer can still perform the reads
- There is flexibility of re-routing the request based on the type of request, customer, or URL
- It can implement cache-based sharding where specific nodes can be determined to serve specific needs of the customer

Cache-based sharding

Caching and sharding are two important strategies for any production system. Neo4j provides a high-performance cache with the Enterprise Edition, which can be leveraged for fast lookups. It enables the flexibility to provide specific memory sizes for nodes.



Enable high-performance cache by modifying `cache_type=hpc` in `<$NEO4J_HOME>/conf/neo4j.properties`.

In the Neo4j world, we consider large data, large enough that it cannot fit into the provided memory (RAM), so the next option would be to introduce shards and distribute these shards to individual nodes.

Sharding data and then caching it on individual nodes is a reasonable and scalable solution, but it is difficult to shard the graphs with a traditional sharding approach and it may not scale for real-time transactions too. That's the reason there is no utility/API provided in Neo4j to shard the data.

So what's next???

The answer is cache-based sharding.

In cache-based sharding, all nodes in a cluster contain the full data, but we partition the type of requests served by each database instance to increase the likelihood of hitting a warm cache for a given request. Warm caches in Neo4j are ridiculously high in performance, especially the HPC – high-performance cache.

In short, we would recommend a routing strategy that routes the user read requests in such a manner that they are always served by a specific set of nodes in a cluster.

The strategy could be based on the domain or may be based on the specific type of query or characteristics of data. We could also use sticky sessions where the first and subsequent request is served by the same node.

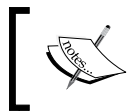
In any case, we need to ensure that majority of READ requests are served by warm cache and not by the disk.

Monitoring

We discussed the Neo4j browser in *Chapter1, Installation and the First Query*.

Neo4j browser exposes the basic configuration of our server and database, but that is not enough for the enterprise class systems, where we need detail monitoring, statistics, and options to modify certain configurations at runtime without restarting the server.

Neo4j exposes JMX beans for advanced level of monitoring and management, which not only exposes the overall health of our Neo4j server and database, but also provides certain operations that can be invoked over live Neo4j instances and that too without restarting the server. Most of the monitoring options exposed through JMX beans are only available with the Enterprise version of Neo4j.



For more information on JMX, refer to <http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html>.

Java provides JConsole — <http://docs.oracle.com/javase/7/docs/technotes/guides/management/jconsole.html> — which is packaged with standard JDK 7 distribution for viewing/modifying/invoking the attributes or operations exposed by JMX beans. JConsole is also leveraged for viewing the overall health of our system where it exposes the various memory statistics such as heap/non-heap, threads JVM configurations, classes loaded in VM, and active threads along with their current state.

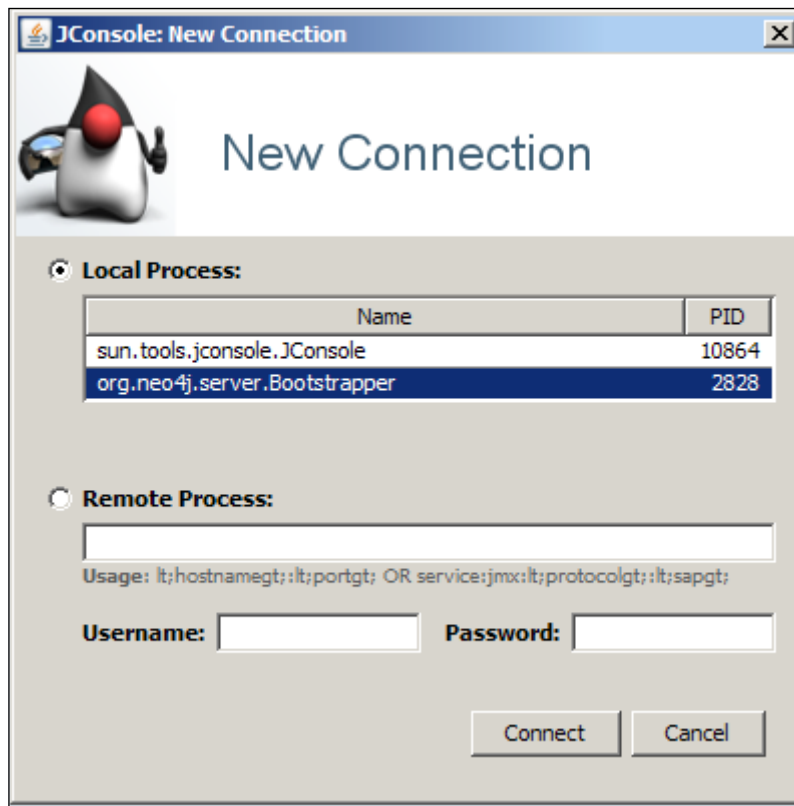
Let's move forward and discuss the configurations required for enabling JMX beans via JConsole for viewing the various monitoring attributes exposed by Neo4j in local and remote mode.

JConsole in local mode

JMX provides the option of directly connecting to the process ID of Neo4j instance running on the local machine and viewing the various JMX beans exposed by Neo4j. Perform the following steps to view JMX beans and JConsole in local mode in Linux OS.

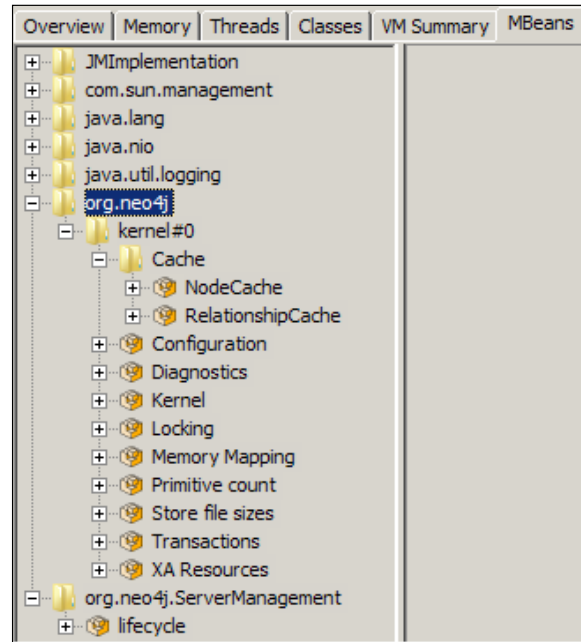
For Windows, change the forward slash / to a backward slash \, and the rest of the steps remain the same:

1. Start your Neo4j server by executing `<$NEO4J_HOME>/bin/neo4j start`.
2. Open your console and execute `<$JAVA_HOME>/bin/jconsole` on the same machine that is hosting your Neo4j server.
3. Select the **Local Process** option and then select **org.neo4j.server.Bootstrapper** and click on **Connect**.



And we are done!!! You will be able to see the JConsole UI that displays the health of your system, such as memory utilization, details about JVM, and so on.

4. Click on the last tab **MBeans** and you will see two JMX Beans: **org.neo4j** and **org.neo4j.ServerManagement**.



JConsole in remote mode

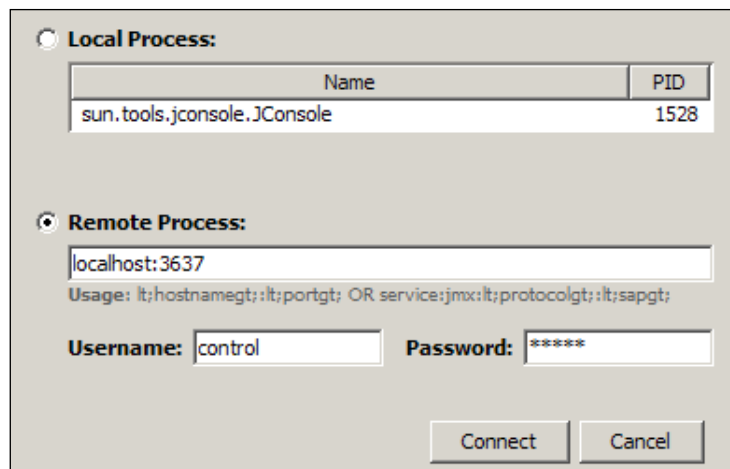
JMX also provides the option of connecting to the remote Java process or Neo4j instance and viewing the various JMX beans. Perform the following steps to view JMX beans and JConsole in remote mode in Linux OS. For windows, change the forward slash / to a backward slash \, and rest of the steps remain the same:

1. Open `<$NEO4J_HOME>/conf/neo4j-wrapper.conf` and enable the following properties:

```
wrapper.java.additional=-
|Dcom.sun.management.jmxremote.port=3637
wrapper.java.additional=-
Dcom.sun.management.jmxremote.authenticate=true
wrapper.java.additional=-
Dcom.sun.management.jmxremote.ssl=false
wrapper.java.additional=-
Dcom.sun.management.jmxremote.password.file=conf/jmx.passwo
rd
wrapper.java.additional=-
Dcom.sun.management.jmxremote.access.file=conf/jmx.access
```

All the preceding properties define the configuration of JMX beans such as communication port, username and password files, and so on.

2. Next we need to modify the username/password for connecting the JMX server remotely. Open `<$NEO4J_HOME>/conf/jmx.access` and uncomment the line `control readwrite`. Uncommenting this line will enable the admin role for the JMX beans and you can modify and invoke the operations exposed by Neo4j JMX beans.
3. Now we will add the username and password in `jmx.password`. Open `<$NEO4J_HOME>/conf/jmx.password` and at the end of file enter `control <space><password>` like `control Sumit`, where the first word is the username and the second word is the password. The username should match with the entry made in the `jmx.access` file.
4. We also need to make sure that the permissions for `conf/jmx.password` are `0600` in Linux. Open the console and execute `sudo chmod 0600 conf/jmx.password` in Linux, and For windows, follow instructions given at <http://docs.oracle.com/javase/7/docs/technotes/guides/management/security-windows.html>.
5. In your Linux console, execute `<$JAVA_HOME>/bin/jconsole` and select **Remote Process**, and in the textbox enter `localhost:3637`, username as `control`, and password as `Sumit`.



6. And we are done!!! Click on **Connect** and you will be able to see the UI of JConsole exposing the health statistics of your system along with the MBeans exposed by Neo4j.

Summary

In this chapter, we discussed the Neo4j architecture and its various components which converged and provided a scalable and HA architecture. We also talked about various principles of the Neo4j cluster and also provided recommendations for various production scenarios. Lastly, we also discussed about the monitoring options available to the Neo4j users.

In the next chapter, we will discuss the Neo4j security and extensions.

8

Neo4j Security and Extension

Enterprises in today's world have various security requirements where they not only restrict access to data but also need to comply with various security norms published by the government.

For example, the companies in the healthcare domain need to abide by HIPAA (The Health Insurance Portability and Accountability Act) security rule, which defines the national standards for securing the data pertaining to health information of patients.

To meet these compliance levels and to protect confidential data from unauthorized access, it is important that enterprise solutions provide flexibility to apply standard security measures and various hooks for defining the customized security rules.

This also introduces another important aspect of a software solution – extensibility of a proposed solution where it defines approaches to extensibility and provides flexibility to users to insert their own program routines.

Every business is different, with its own needs and path of growth. As the business grows, we may need to extend or add functionality to the existing system for providing ease to the users or may be comply with new security measures introduced by the government (such as HIPAA), and for all these reasons, it is important that the proposed solution provides the flexibility to introduce new APIs/functionalities.

Unlike security, extensibility may not be an explicit requirement, but for enterprises, it continues to remain as one of the evaluation criteria for any software solution.

In this chapter, we will discuss the security and extension provided by the Neo4j server, which not only provides measures to protect from unauthorized access but also provides flexibility to implant customized routines and extend the functionality of the Neo4j server.

This chapter will cover the following topics:

- Neo4j security
- API extension

Neo4j security

Neo4j does not enforce any security at data levels but provides various means to protect unauthorized access to our Neo4j deployment.

Let's discuss the security measures and custom hooks provided by Neo4j for protecting the Neo4j deployment.

Securing access to Neo4j deployment

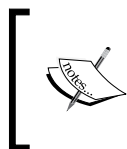
The first step in any enterprise security is to restrict direct access to the production servers and allow only secure communication.

Neo4j is bundled with a web server that provides the access to the Neo4j browser. We can modify the web server parameters and control access to the Neo4j browser.

Let's discuss the various web server configurations defined in `<$NEO4J_HOME>/conf/neo4j-server.properties` and their role in securing the access of the Neo4j browser:

- `org.neo4j.server.webserver.address=0.0.0.0`: This property defines the IP address on which the web server will accept the incoming connections. The default value is 0.0.0.0, which means it will accept connections only from the local box hosting the Neo4j server.
- `org.neo4j.server.webserver.https.enabled=true`: This is a Boolean parameter used to enable or disable the support for HTTPS. We should set it to true to enable and accept HTTPS connections.
- `org.neo4j.server.webserver.https.port=7473`: This parameter defines the port for accepting the HTTPS request. The default value is 7473.
- `org.neo4j.server.webserver.port=7474`: This is the default HTTP port for unsecured communication with the web server or the Neo4j browser. We should disable this property by appending # and stop all unsecured communication.

- `org.neo4j.server.webserver.https.cert.location=conf/ssl/snakeoil.cert`, `org.neo4j.server.webserver.https.key.location=conf/ssl/snakeoil.key`, and `org.neo4j.server.webserver.https.keystore.location=data/keystore`: These properties define the location of the certificate, key, and Keystore, used in secured communication on the HTTPS protocol. We should change the default values and define our own certificate, key, and Keystore. Refer to the link <https://docs.oracle.com/cd/E19798-01/821-1751/ghlgv/index.html> to generate the certificate, key, and Keystore.



For sensitive deployments/applications, it is recommended to buy and install certificates from a trusted **certificate authority (CA)** such as Verisign <https://www.verisigninc.com/>, Thawte <http://www.thawte.com/>, and so on.

Restricting access to Neo4j server / cluster with the proxy server

Neo4j provides various security measures for secure communication, but at the same time, it is recommended and sensible to place a proxy server such as Apache in front of our Neo4j servers. Proxy servers will act as a gateway to the outside world for accessing our Neo4j server. The following are the advantages of having proxy servers:

- It controls access to the Neo4j server to specific IP addresses, URL patterns, and IP ranges. For example, it can allow access to the Neo4j browser only from specific IP addresses or a range of IP addresses. Refer to the following link to configure Apache as a proxy server: http://httpd.apache.org/docs/2.2/mod/mod_proxy.html.
- It hides all sensitive IP addresses and URLs from the outside world by using URL rewriting. Refer to the following link to configure and enable URL rewriting in Apache: http://httpd.apache.org/docs/2.2/mod/mod_rewrite.html.
- It can also act as a load balancer for Neo4j HA cluster. Refer to the following link to configure and enable load balancing in Apache: http://httpd.apache.org/docs/2.2/mod/mod_proxy_balancer.html.

Feature deactivation

Neo4j provides flexibility to add and execute arbitrary pieces of code through unmanaged extensions (see the next section for details about unmanaged extensions), which can produce severe security threat to your infrastructure. It is advisable to think twice before you decide to move ahead with unmanaged extensions and also make sure that all offending plugins or extensions are removed from the server classpath. Alternatively, we can also secure the access to these custom plugins/extensions through proxies or authorization rules.

Neo4j is implemented in Java, so we can also introduce and implement security manager from <http://docs.oracle.com/javase/7/docs/technotes/guides/security/index.html> to secure the different parts of the code base.

Fine-grained authorization

Security requirements may vary from organization to organization and to make it more complex, there are instances where the country laws, local authorities, or various government policies govern the security requirements. It is difficult (if not impossible) to provide a single framework that meets all security requirements of various organizations. Therefore, for all those instances where standard security measures are not enough, Neo4j provides the flexibility to implement and inject custom security rules / policies to secure various REST endpoints. It exposes the `org.neo4j.server.rest.security.SecurityRule` interface for implementing custom security policies / rules; furthermore, it can be configured in `<$NEO4J_HOME>/conf/neo4j-server.properties`.

Let's extend our *Spring-Neo4j* example, which we created in *Chapter 6, Spring Data and Neo4j* and perform the following steps to implement and configure a custom security rule to deny access to all HTTP REST endpoints, which creates or updates a node or relationship:

1. Open `Spring-Neo4j/pom.xml` and add the following dependency within the `<dependencies>` and `</dependencies>` tags:

```
<dependency>
  <groupId>org.neo4j.app</groupId>
  <artifactId>neo4j-server</artifactId>
  <version>${neo4j.version}</version>
</dependency>
```
2. Create a new package `org.neo4j.custom.security.rules` and create a new class `DenyCreateRequestSecurityRule.java` within this package.

3. Add the following source code within `DenyCreateRequestSecurityRule`:

```
package org.neo4j.custom.security.rules;

import javax.servlet.http.HttpServletRequest;
import org.neo4j.server.rest.security.SecurityFilter;
import org.neo4j.server.rest.security.SecurityRule;

public class DenyCreateRequestSecurityRule implements
SecurityRule{

    //Read about RFC - 2617 for more information about REALM
    -
    //http://tools.ietf.org/html/rfc2617RFC 2617
    public static final String REALM = "WallyWorld";

    @Override
    public boolean isAuthorized(HttpServletRequest request) {
        //Deny all other Type of Request except GET
        if(request.getMethod().equalsIgnoreCase("GET")){
            return true;
        }
        return false;
    }

    @Override
    public String forUriPath() {

        //This security rule will apply only on the URL which
        starts /db/data/
        return "/db/data/*";
    }

    @Override
    public String wwwAuthenticateHeader() {
        //Read about RFC - 2617 for more information about REALM
        -
        //http://tools.ietf.org/html/rfc2617RFC 2617
        return SecurityFilter.basicAuthenticationResponse(REALM);
    }
}
```

The preceding rule implements `SecurityRule` and defines three methods:

- `wwwAuthenticateHeader()`: This method defines the security REALM and adds it to the security filter used by Neo4j for basic authentication.
- `forUriPath()`: This method defines the URL on which this security needs to be applied or evaluated.
- `isAuthorized()`: This is the main method that defines the custom logic for denying or approving the access to the specific URI requested by the user. For example, we have allowed only GET requests and denied access to all other types of HTTP requests.

4. Open your console, browse the root directory of your project, that is, `Spring-Neo4j`, and execute the following Maven command:

```
$M2_HOME/bin/mvn clean install
```

5. Now open `<$Spring-Neo4j>/target/` and place the project's JAR file in the server's classpath or simply copy to `<$NEO4J_HOME>/lib/`.
6. Next, open `<$NEO4J_HOME>/conf/neo4j-server.properties` and add the following property to configure the security rule:

```
org.neo4j.server.rest.security_rules=org.neo4j.custom.security.rules.DenyCreateRequestSecurityRule
```



We can define more than one rule by separating them with a comma (,).

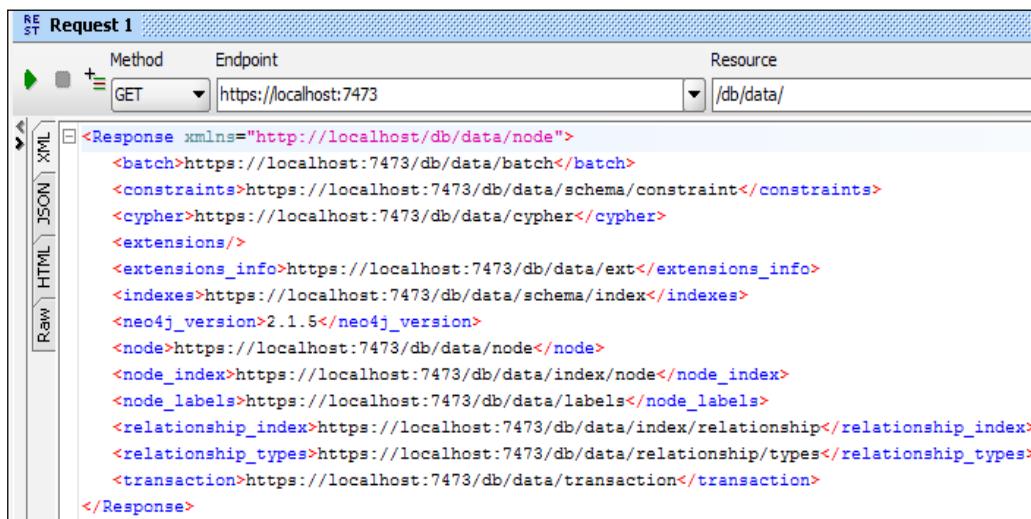
7. Next, start your Neo4j server by executing `<$NEO4J_HOME>/bin/neo4j start`.

And we are done!!! Our first security rule is deployed and now each time a user executes an HTTP or REST request, our security rule will be evaluated, and based on the type of request, access to the server will be denied or approved.

8. Next, execute any REST request from SoapUI and the security rule will deny access to the application or Neo4j server for all other types of requests except GET.

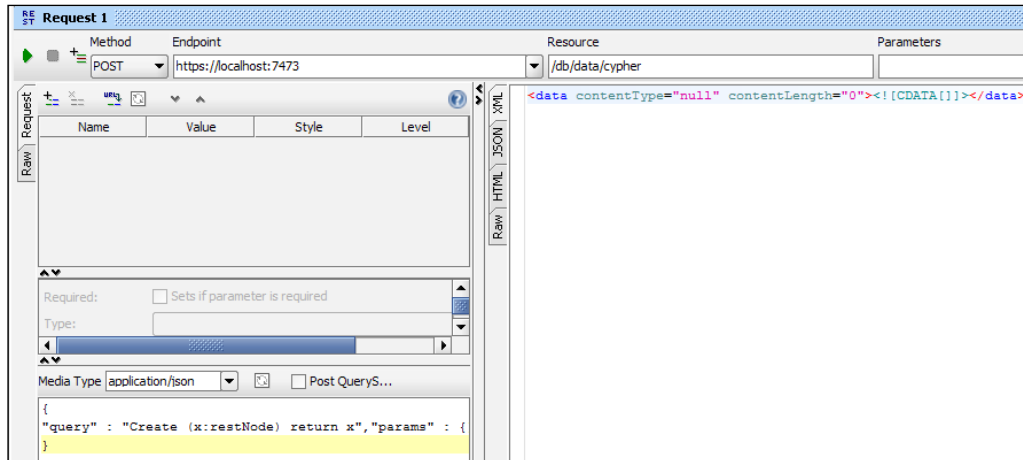
9. Open SoapUI or any other tool for executing a GET REST request with the following configurations:
 - Request method type: **GET**
 - Request URL: `https://localhost:7473/db/data/` or `http://localhost:7474/db/data/`
 - Request headers: Accept: application/json; charset=UTF-8 and Content-Type: application/json

Execute the REST request and you will see results as a JSON response, as shown in the following screenshot:



10. Let's execute a POST request and see the results. Open SoapUI or any other tool for executing a POST REST request with the following configurations:
 - Request method type: **POST**
 - Request URL: `https://localhost:7473/db/data/cypher`
 - Request headers: Accept: application/json; charset=UTF-8 and Content-Type: application/json
 - JSON request: `{"query" : "Create (x:restNode) return x", "params" : {}}`

11. Execute the REST request and you will not get any response as the request is denied by our custom security rule, as shown in the following screenshot:



API extensions – server plugins and unmanaged extensions

Neo4j provides flexibility to extend and add new functionality to the Neo4j server via custom server plugins and extensions. Let's discuss each of these features and see the process for implementing / deploying plugins and extensions:

- **Server plugins:** Server plugins are user-defined code or program routines that extend the capabilities of the database, nodes, or relationships. These plugins are then advertised as new REST endpoints to the end users.
- **Unmanaged extensions:** Neo4j also provides flexibility to develop and deploy unmanaged extensions, which provide full control over the exposed APIs. Unmanaged extensions are a way of deploying arbitrary JAX-RS code into the Neo4j server. If not used carefully, then it can be disastrous to an extent where a poorly written code can bring down the whole server.

Let's see sample code and examples for developing and deploying server plugins and unmanaged extensions.

Server plugins

Server plugins provide an architectural pattern exposed by Neo4j for extending its functionality and exposing the new REST endpoints for its consumers.

These new REST endpoints are automatically exposed and advertised in the representations and clients can discover the extension implemented by the plugins by performing a GET on the default database URI: `http://localhost:7474/db/data/` or `https://localhost:7473/db/data/`.

Let's extend our `Spring-Neo4j` project and perform the following steps to implement a server plugin for traversing the graph from a given node on the movie dataset that we created in *Chapter 3, Pattern Matching in Neo4j*, under the *Read-only Cypher queries* section:

1. Remove the project's JAR file from `<$NEO4J_HOME>/lib/`, which we copied during our previous example stated in the *Fine-grained authorization* section.
2. Disable all custom security rules (if any). Open `<$NEO4J_HOME>/conf/neo4j-server.properties` and comment the entry, which starts with `org.neo4j.server.rest.security_rules`.
3. Create a new package, `org.neo4j.custom.server.plugins`, and define a new Java class `GraphTraversal.java` under this new package.
4. Add the following code in `GraphTraversal.java` and follow the comments provided in the code to understand the implementation logic:

```
package org.neo4j.custom.server.plugins;

import java.util.ArrayList;
import java.util.List;
import org.neo4j.cypher.*;
import org.neo4j.graphdb.*;
import org.neo4j.graphdb.factory.Description;
import org.neo4j.graphdb.traversal.*;
import org.neo4j.kernel.impl.util.StringLogger;
import org.neo4j.server.plugins.*
import scala.collection.Iterator;

@Description("An extension to the Neo4j Server for
Traversing graph till 3 levels in all directions from a
given Node")
```

```
public class GraphTraversal extends ServerPlugin {
    @Name("graph_traversal")
    @org.neo4j.server.plugins.Description("Traversing Graphs
    from a given Node")
    @PluginTarget(GraphDatabaseService.class)
    public Iterable<Path> executeTraversal(@Source
    GraphDatabaseService graphDb,
    @org.neo4j.server.plugins.Description("Value of 'Name'
    property, considered as RootNode for searching ")
    @Parameter(name = "name") String name) {
        return startTraversing(graphDb, name);
    }

    public enum RelTypes implements RelationshipType {
        ACTED_IN, DIRECTED
    }

    private List<Path> startTraversing(GraphDatabaseService
    graphDb, String name) {
        List<Path> allPaths = new ArrayList<Path>();
        // Start a Transaction
        try (Transaction tx = graphDb.beginTx()) {
            // get the Traversal Descriptor from instance of
            Graph DB
            TraversalDescription trvDesc =
            graphDb.traversalDescription();
            // Defining Traversals needs to use Depth First
            Approach
            trvDesc = trvDesc.depthFirst();
            // Instructing to exclude the Start Position and
            include all
            // other Nodes while Traversing
            trvDesc =
            trvDesc.evaluator(Evaluators.excludeStartPosition());
            // Defines the depth of the Traversals. Higher the
            Integer, more
            // deep would be traversals. Default value would be to
            traverse //complete Tree
            trvDesc = trvDesc.evaluator(Evaluators.toDepth(3));
            //Define Uniqueness in visiting Nodes while Traversing
            trvDesc = trvDesc.uniqueness(Uniqueness.NODE_GLOBAL);
```

```

//Traverse only specific type of relationship
trvDesc = trvDesc.relationships(RelTypes.ACTED_IN,
Direction.BOTH);
// Get a Traverser from Descriptor
Traverser traverser =
trvDesc.traverse(getStartNode(graphDb,name));

// Let us get the Paths from Traverser and start
iterating or
//moving along the Path
for (Path path : traverser) {
    //Add Paths
    allPaths.add(path);
}
//Return Paths
return allPaths;
}}

/**
 * Get a Root Node as a Start Node for Traversal.
 * @param graphDb
 * @param name
 * @return Root Node
 */
private Node getStartNode(GraphDatabaseService graphDb,
String name) {
    try (Transaction tx = graphDb.beginTx()) {
        ExecutionEngine engine = new ExecutionEngine(graphDb,
StringLogger.SYSTEM);
        String cypherQuery = "match (n)-[r]->() where
n.Name='"+name+"'" return n as RootNode";
        ExecutionResult result = engine.execute(cypherQuery);
        Iterator<Object> iter = result.columnAs("RootNode");
        //Check to see that we do not have empty Iterator
        if(iter==null || iter.isEmpty()){
            return null;
        }
        return (Node) iter.next();
    }
}
}

```


The preceding code extends `ServerPlugin`, which manages the lifecycle of our custom plugin and adds it as an extension to the already exposed REST endpoints.

There are a few important points that we need to remember or implement while creating the server plugin:

- We need to ensure that our plugin produces or returns an instance or list of node, relationship, or path, or an instance of `org.neo4j.server.rest.repr.Representation` that can be iterated. For example, the preceding code returns `Iterable<Path>`.
 - It specifies request parameters using the `@Parameter` annotation. Similar to what we used in the preceding code `@Parameter (name = "name")`. You can skip this annotation if your service does not need any user inputs.
 - Specify the point of extension as we specified in the preceding code using the `@Name` annotation.
 - Specify the discovery point type in the `@PluginTarget` annotation and the `@Source` parameter, which are of the same type.
 - Needless to mention, we also need to define the application logic, which in the preceding code is defined in the `startTraversing` method.
5. Add the `META-INF` and `services` directories under `<$Spring-Neo4j>/src/main/resources`.
 6. Add a new file `org.neo4j.server.plugins.ServerPlugin` under `<$Spring-Neo4j>/src/main/resources/META-INF/services`. The content of this file should be the fully qualified name of your plugin: `org.neo4j.custom.server.plugins.GraphTraversal`.

We can add more plugins but every plugin needs to be defined on a new line.

7. Open the console, browse `<$Spring-Neo4j>`, and execute the following command to compile the code:

```
<$M2_HOME>/bin/mvn clean install
```
8. Copy the project's JAR file from `<$Spring-Neo4j>/target/` and place it in `<$NEO4J_HOME>/plugins/`.
9. Stop your Neo4j server by pressing `Ctrl + C` (in case it is already running) and execute `<$NEO4J_HOME>/bin/neo4j start` and your Neo4j server will be up and running with your custom plugin.

10. Open SoapUI or any other tool for executing a GET REST request with the following configurations:
 - Request method type: **GET**
 - Request URL: `https://localhost:7473/db/data/` or `http://localhost:7474/db/data/`
 - Request headers: `Accept: application/json; charset=UTF-8` and `Content-Type: application/json`
11. Execute the REST request and you will see that your plugin is listed as a new REST endpoint under `<extensions>`, as shown in the following screenshot:



```

GET https://localhost:7473/db/data/
<Response xmlns="http://localhost/db/schema/index/Artist">
  <batch>https://localhost:7473/db/data/batch</batch>
  <constraints>https://localhost:7473/db/data/schema/constraint</constraints>
  <cypher>https://localhost:7473/db/data/cypher</cypher>
  <extensions>
    <GraphTraversal>
      <graph_traversal>https://localhost:7473/db/data/ext/GraphTraversal/graphdb/graph_traversal</graph_traversal>
    </GraphTraversal>
  </extensions>
  <extensions_info>https://localhost:7473/db/data/ext</extensions_info>
  <indexes>https://localhost:7473/db/data/schema/index</indexes>
  <neo4j_version>2.1.5</neo4j_version>
  <node>https://localhost:7473/db/data/node</node>
  <node_index>https://localhost:7473/db/data/index/node</node_index>
  <node_labels>https://localhost:7473/db/data/labels</node_labels>
  <relationship_index>https://localhost:7473/db/data/index/relationship</relationship_index>
  <relationship_types>https://localhost:7473/db/data/relationship/types</relationship_types>
  <transaction>https://localhost:7473/db/data/transaction</transaction>
</Response>

```

12. Open your `<$NEO4J_HOME>/bin/neo4j-shell` and execute the following cypher statements to clean up your database:


```

match (x) - [r] - () delete r;
match (x) delete x;

```
13. Next, execute the Cypher queries given in *Chapter 3, Pattern Matching in Neo4j*, under the *Read-only Cypher queries* section for creating a sample movie dataset.
14. Open SoapUI or any other tool for executing a POST REST request with the following configurations:
 - Request method type: **POST**
 - Request URL: `https://localhost:7473//db/data/ext/GraphTraversal/graphdb/graph_traversal`

- Request headers: Accept: application/json; charset=UTF-8 and Content-Type: application/json
- JSON as requests parameter: {"name" : "Sylvester Stallone"}

15. Execute the REST request and you will see the results as a JSON response.

Unmanaged extensions

Unmanaged extensions provide fine-grained control over server-side code. It provides flexibility to deploy an arbitrary piece of JAX-RS code into your Neo4j server and then further exposes it as a new REST endpoint. We need to be careful while developing and deploying unmanaged extensions, as it can directly impact the performance of the server, in case of poorly written code.

Let's extend our Spring-Neo4j project and perform the following steps to implement unmanaged extensions to get all the node IDs from the Neo4j database:

1. Create a new package `org.neo4j.custom.server.extension.unmanaged` and define a new Java class `GetAllNodes.java` under this new package.
2. Add the following code in `GetAllNodes.java` and follow the comments provided in the code to understand the implementation logic:

```
package org.neo4j.custom.server.extension.unmanagaed;
```

```
import java.nio.charset.Charset;
import java.util.HashMap;
import java.util.Map;
import javax.ws.rs.*;
import javax.ws.rs.core.*;
import javax.ws.rs.core.Response.*;
```

```
import org.neo4j.graphdb.*;
import org.neo4j.shell.util.json.JSONObject;
import org.neo4j.tooling.GlobalGraphOperations;
```

```
//Context Path for exposing it as REST endpoint
@Path("/getAllNodes")
public class GetAllNodes {
```

```
    private final GraphDatabaseService graphDb;
```

```

//Injected by the Neo4j Server
public GetAllNodes(@Context GraphDatabaseService graphDb)
{
    this.graphDb = graphDb;
}

//Implementation Logic for exposing this as a GET
response
@GET
@Produces(MediaType.APPLICATION_JSON)
public Response nodeIDs() {
    Map<String, String> nodeIDs = new
    HashMap<String,String>();

    try (Transaction tx = graphDb.beginTx()) {
        //Get all Node ID's and put them into the MAP.
        //Key of Map needs to be appended by Node_ID, so that
        every entry is Unique
        for (Node node :
        GlobalGraphOperations.at(graphDb).getAllNodes()) {
            nodeIDs.put("Node_ID_"+Long.toString(node.getId()),
            Long.toString(node.getId()));
        }
        tx.success();
    }
    //Converting the Map Object into JSON String
    JSONObject jsonObj = new JSONObject(nodeIDs);

    //Returning a Success Status, along with the JSON
    response.
    return Response.status(Status.OK)
    .entity(jsonObj.toString().getBytes(Charset
    .forName("UTF-8"))).build();
}
}

```

The preceding code uses the `@Context` annotation to inject the dependency of `GraphDatabaseService`, which fetches all the nodes from the Neo4j database and wraps them into a JSON object. Finally, the code transforms JSON into string and returns it back to the user as a response.

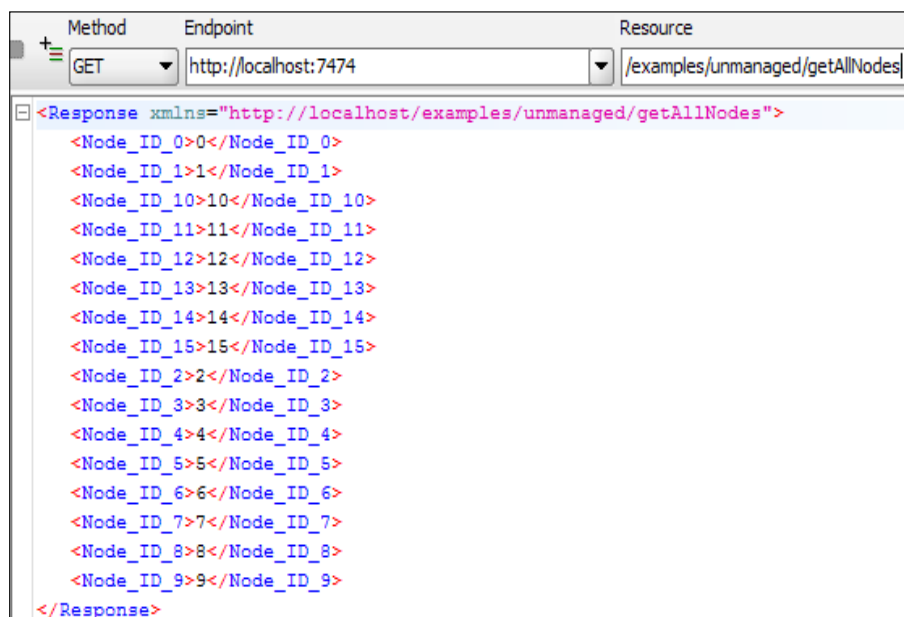
3. Open the console, browse <\$Spring-Neo4j>, and execute the following command to compile the code:

```
<$M2_HOME>/bin/mvn clean install
```
4. You will see the project's JAR file in <\$Spring-Neo4j>/target/. Copy the JAR file and place it in <\$NEO4J_HOME>/plugins/.
5. Open <\$NEO4J_HOME>/conf/neo4j-server.properties and add the following configuration:

```
org.neo4j.server.thirdparty_jaxrs_classes=org.neo4j.custom.server.  
extension.unmanaged=/examples/unmanaged
```

The preceding configuration defines the package name, which contains your unmanaged extensions (org.neo4j.custom.server.extension.unmanaged) and the extension point (/examples/unmanaged), which will be exposed as a new REST endpoint (/examples/unmanaged/getAllNodes).
6. Stop your Neo4j server by pressing *Ctrl* + *C* (in case it is already running) and execute <\$NEO4J_HOME>/bin/neo4j start. Now your Neo4j server will be up and running with your unmanaged extension.
7. Open SoapUI or any other tool and execute the GET REST request with the following configurations:
 - Request method type: **GET**
 - Request URL: https://localhost:7473/examples/unmanaged/getAllNodes or http://localhost:7474/examples/unmanaged/getAllNodes
 - Request headers: Accept: application/json; charset=UTF-8 and Content-Type: application/json

8. Execute the REST request and you will see that the request is successful and returns a JSON response containing the node IDs, as shown in the following screenshot:



```
<Response xmlns="http://localhost/examples/unmanaged/getAllNodes">
  <Node_ID_0>0</Node_ID_0>
  <Node_ID_1>1</Node_ID_1>
  <Node_ID_10>10</Node_ID_10>
  <Node_ID_11>11</Node_ID_11>
  <Node_ID_12>12</Node_ID_12>
  <Node_ID_13>13</Node_ID_13>
  <Node_ID_14>14</Node_ID_14>
  <Node_ID_15>15</Node_ID_15>
  <Node_ID_2>2</Node_ID_2>
  <Node_ID_3>3</Node_ID_3>
  <Node_ID_4>4</Node_ID_4>
  <Node_ID_5>5</Node_ID_5>
  <Node_ID_6>6</Node_ID_6>
  <Node_ID_7>7</Node_ID_7>
  <Node_ID_8>8</Node_ID_8>
  <Node_ID_9>9</Node_ID_9>
</Response>
```

Summary

In this chapter, we discussed various security aspects of the Neo4j server. We talked about the threats and procedures/methods to secure our Neo4j deployment. We also talked about the flexibility provided by Neo4j for extending its functionality by implementing plugins and extensions.

Index

Symbols

<\$NEO4J_HOME>/bin/neo4j utility

about 24

neo4j console 24

neo4j start 25

neo4j status 25

neo4j stop 25

<\$NEO4J_HOME>/bin/neo4j-installer

utility

about 25

neo4j-installer install 25

neo4j-installer remove 25

neo4j-installer start 25

neo4j-installer status 25

neo4j-installer stop 25

neo4j-installer usage 25

<\$NEO4J_HOME>/bin/neo4j-shell utility

about 25

neo4j-shell -c <COMMAND> 25

neo4j-shell -config - <CONFIG> 26

neo4j-shell -file <FILE> 26

neo4j-shell -host <HOST> 26

neo4j-shell -name <NAME> 26

neo4j-shell -path <PATH> 25

neo4j-shell -pid <PID> 25

neo4j-shell -port <PORT> 26

neo4j-shell -readonly 25

@RelatedTo annotation 137

@RelatedToVia annotation 137

A

Agile data modeling 56-58

Apache Proxy

URL 154

API extensions

server plugins 168

unmanaged extensions 168

AspectJ

used, for mapping mode 141-143

AspectJ modes

compile-time weaving 143

load-time weaving 143

run-time weaving 143

Aspect-oriented programming (AOP)

about 141

defining 141

AssertJ

URL 111

B

backups

enabling 149

batch import processes

CSV importer 39

Excel 42, 43

HTTP batch imports 43-45

Java API 45

batch indexing 48-50

BatchInserter 45, 48

BatchInserterIndex

demonstrating 50

batch inserter / indexer

considerations, for tuning memory

and JVM 53

Business Intelligence (BI)

about 33

URL 33

C

cache-based sharding 155

certificate authority (CA) 163

cluster manager

features 147

Community Edition 10

CREATE command

LOAD CSV command, using with 40, 41

Create, Read, Update, and Delete (CRUD) 122

CSV importer

used, for importing data in batches 39-42

Cypher

about 55

used, for creating schema 75

Cypher optimizations, guidelines

data filtering 95

data model 95

divide and conquer 94

execution engine and plans 95

indexing/constraints 95

WHERE clause, patterns 95

Cypher query

interactive console (Neo4j shell) 27

Java code API 29, 30

Neo4j browser 31

Neo4j database, embedding 29, 30

REST APIs, defining 28

running 27

Cypher write queries

about 80

MERGE, working with 82

nodes, working with 80

relationships, working with 80

D

data

writing, in legacy indexing 84-87

writing, in schema 88

Dependency Injection (DI) 121

deployment options, Neo4j

about 13

Neo4j Community Edition on

Windows/Linux 14

Neo4j Enterprise Edition on

Windows/Linux 14

developers

tools 24-27

utilities 24-27

E

Eclipse

URL 98

embedded, versus REST

about 98

Neo4j, as REST-based application 101-104

Neo4j, embedding in Java

applications 98-100

Enterprise Edition

about 11

enterprise subscriptions 12

features 11

personal license plan 11

start up program 12

URL 11

enterprise subscriptions 12

Excel

used, for importing data in batches 42, 43

F

File Buffer Cache 154

fine-grained authorization 164-168

frameworks

testing, for Neo4j 111, 112

G

General Public License (GPL) Version 3

URL 10

Gist

URL 76

graph data structure

URL 56

GraphGists

used, for creating Movie Demo 76-78

graph traversals

about 114-118

evaluator 115

order of search 115

path expanders 115

starting nodes 115

uniqueness 115

GraphUnit

URL 111

GREMLIN

URL 59

H

HA proxy

URL 154

HTTP batch imports 43-45

I

indexing techniques

legacy indexes 74

schema level indexing 75

IndexManager API

URL 74

IntelliJ IDEA

URL 98

interactive console (Neo4j shell) 27

Inversion of Control (IoC) 121

J

Java

used, for applying unicity constraints 92-94

Java API

about 113, 114

BatchInsertter 45, 48

BatchInsertterIndex 48-50

graph algorithms 114

graph database 113

graph matching 114

helpers 114

imports 114

management 114

packages 113

query language 114

tooling 114

URL 45

used, for importing data in batches 45

used, for managing schema 88, 89

Java code API 29, 30

JConsole

in local mode 156, 157

in remote mode 158, 159

URL 156

JMX

URL 156

JUnit

URL 107

JVM

tuning 52

L

labels 57

legacy indexing

about 74

data, writing 84-87

using 74

licensing options, Neo4j

applicability 12

Community Edition 10

Enterprise Edition 11

Linux service

Neo4j, installing as 20

Linux tar/ standalone application

Neo4j, installing as 19, 20

Linux/Unix

Neo4j cluster, configuring on 23, 24

load balancer

benefits 155

LOAD CSV command

about 52, 53

using, with CREATE command 40, 41

using, with MERGE command 41, 42

Lucene

URL 74

M

M2E plugin

URL 126

mapping mode

AspectJ, used for 141-143

MATCH clause

nodes, working with 68

relationships, working with 69, 70

working with 68

Maven 3.2.3

URL 98, 125

MERGE command

LOAD CSV command, using with 41, 42

working with 82-84

Metaweb Query Language (MQL)

URL 59

methods, SecurityRule

forUriPath() 166

isAuthorized() 166

wwwAuthenticateHeader() 166

monitoring

about 156

JConsole, in local mode 156, 157

JConsole, in remote mode 158, 159

movie dataset

creating 65-67

Movie Demo

creating, GraphGists used 76-78

N

Neo4j

about 9, 34

advanced settings 146

agile data modeling 56-58

architecture 146

embedding, in Java applications 98-100

features 146, 150-152

frameworks, testing for 111, 112

integrating, with QlikView 34-39

licensing options 10

pattern matching 55

security 162

Neo4j, as REST-based application

about 101-104

deploying, advantages 104, 105

selecting 104

Neo4j browser 31

Neo4j cluster

configuring, on Linux/Unix 23, 24

configuring, on Windows 21, 23

principles 152

read throughputs, scaling 154

recommended setup 152

write throughputs, scaling 152

Neo4j Community Edition, on Linux/Unix

installing 18, 19

installing, as Linux service 20

installing, as Linux tar / standalone
application 19, 20

Neo4j Community Edition, on Windows

installing 14

installing, as Windows archive / standalone
application 18

installing, as Windows service 15-17

Neo4j database

embedding 29, 30

Neo4j deployment

access, securing to 162, 163

Neo4j Enterprise Edition

installing 21

Neo4j, features

advanced settings 150-152

backup and recovery 149

data locality 148

data replication 148

fault tolerance 147

High Availability 146

linear scalability 146, 147

Neo4j Gist

URL 76

Neo4j JDBC 34

Neo4j JDBC driver

URL, for downloading 34

Neo4j security

access, restricting to Neo4j cluster
with proxy server 163

access, restricting to Neo4j server with
proxy server 163

access, securing to Neo4j

deployment 162, 163

features, deactivating 164

fine-grained authorization 164-168

Neo4jTemplate 125-137

Neo4j web interfaces

Neo4j browser 27

Neo4j WebAdmin 27

NetBeans

URL 98

nodes

about 56

working with 80-82

non-functional requirements (NFRs) 105

O

Object Oriented Data Model (OODM) 122

ODBC-JDBC Bridge, EasySoft

URL 39

optimizations 50

OPTIONAL MATCH clause

working with 71

Oracle Java 7

URL 14, 19

P

parameters, for memory tuning

array_block_size 51

label_block_size 51

mapped_memory_page_size 51

neostore.nodestore.db.mapped_memory 51

neostore.propertystore.db.arrays.

mapped_memory 51

neostore.propertystore.db.

mapped_memory 51

neostore.propertystore.db.strings.

mapped_memory 51

neostore.relationshipstore.db.

mapped_memory 51

node_auto_indexing 51

string_block_size 51

pattern matching 59

patterns

about 59

expressions 62

for labels 61

for multiple nodes 60

for nodes 60

for paths 61

for properties 62

for relationships 61

usage 63, 64

Paxos

URL 148

performance tuning 50

personal license plan 11

prebuilt graph algorithms 119

Q

QlikView

about 34

Neo4j, integrating with 34-39

QlikView JDBC Connector

URL, for downloading 34

R

Rapid Application Development (RAD) 121

read-only Cypher queries

about 65

MATCH clause, working with 68

OPTIONAL MATCH clause,

working with 70

RETURN clause, working with 73

sample dataset, creating 65-67

START clause, working with 71

WHERE clause, working with 72

read throughputs

cache-based sharding 155

load balancer 154, 155

scaling 154

ready-to-use database

creating 39

relationships

about 57

bi-directional 61

unidirectional 61

working with 80-82

repository interfaces, Spring Data

CRUDRepository 124

CypherDslRepository 124

IndexRepository 124

NamedIndexRepository 124

RelationshipOperationsRepository 124

SchemaIndexRepository 124

SpatialRepository 124

TraversalRepository 124

REST

used, for applying unicity constraints 91, 92

used, for managing schema 89, 90

versus embedded 98

RETURN clause

working with 73

S

schema

- creating, Cypher used 75
- data, writing 88
- managing, with Java API 88, 89
- managing, with REST 89-91

schema level indexing

- about 74
- using 75

security manager

- URL 164

server plugins 168-173

SPARQL

- URL 59

Spring Data

- about 122-124
- entities 137-140
- features 123
- repositories 137-140
- URL 123

Spring Data Neo4j

- features 125
- Neo4jTemplate 125-137

Spring repository infrastructure

- URL 124

START clause

- working with 71

start up program 12

system hardware requirements, Neo4j 12, 13

T

Traversals

- rules 118

U

unicity constraints

- applying, with Java 92-94
- applying, with REST 91, 92

unit testing, in Neo4j

- about 106-110
- benefits 106
- frameworks, testing for Neo4j 111, 112

unmanaged extensions 168-177

W

WHERE clause

- working with 72

Windows

- Neo4j cluster, configuring on 21-23
- Neo4j, integrating with QlikView 34-39

Windows archive / standalone application

- Neo4j, installing as 18

Windows service

- Neo4j, installing as 15-17

write operations

- queues, used for 152

write throughputs

- batch writes 153
- Neo4j caches, tuning 154
- queues, for write operations 152
- scaling 152
- vertical scaling 153



Thank you for buying Neo4j Essentials

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

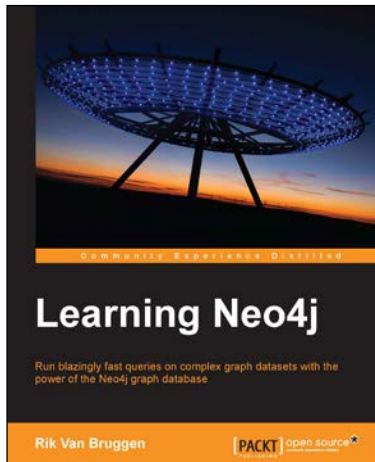
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



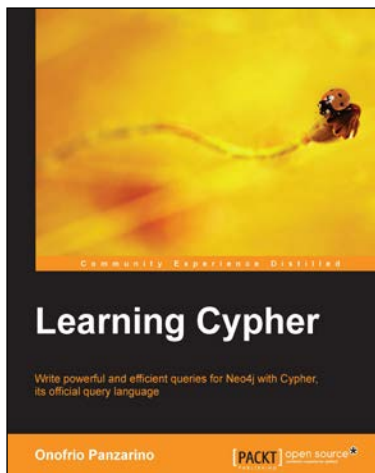
Learning Neo4j

ISBN: 978-1-84951-716-4

Paperback: 222 pages

Run blazingly fast queries on complex graph datasets with the power of the Neo4j graph database

1. Get acquainted with graph database systems and apply them in real-world use cases.
2. Get started with Neo4j, a unique NOSQL database system that focuses on tackling data complexity.
3. A practical guide filled with sample queries, installation procedures, and useful pointers to other information sources.



Learning Cypher

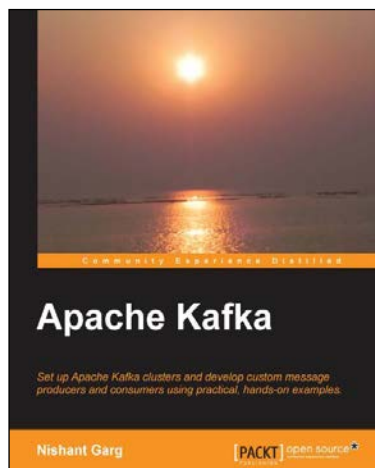
ISBN: 978-1-78328-775-8

Paperback: 162 pages

Write powerful and efficient queries for Neo4j with Cypher, its official query language

1. Improve performance and robustness when you create, query, and maintain your graph database.
2. Save time by writing powerful queries using pattern matching.
3. Step-by-step instructions and practical examples to help you create a Neo4j graph database using Cypher.

Please check www.PacktPub.com for information on our titles



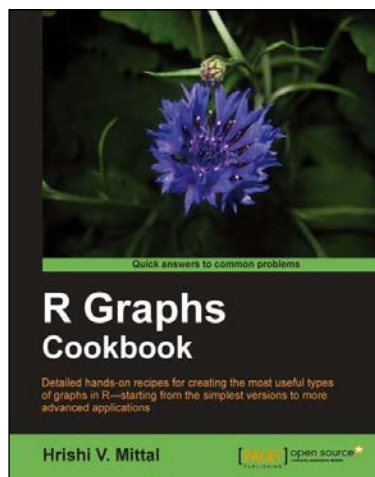
Apache Kafka

ISBN: 978-1-78216-793-8

Paperback: 88 pages

Set up Apache Kafka clusters and develop custom message producers and consumers using practical, hands-on examples.

1. Write custom producers and consumers with message partition techniques.
2. Integrate Kafka with Apache Hadoop and Storm for use cases such as processing streaming data.
3. Provide an overview of Kafka tools and other contributions that work with Kafka in areas such as logging, packaging, and so on.



R Graphs Cookbook

ISBN: 978-1-84951-306-7

Paperback: 272 pages

Detailed hands-on recipes for creating the most useful types of graphs in R—starting from the simplest versions to more advanced applications

1. Learn to draw any type of graph or visual data representation in R.
2. Filled with practical tips and techniques for creating any type of graph you need; not just theoretical explanations.
3. All examples are accompanied with the corresponding graph images, so you know what the results look like.

Please check www.PacktPub.com for information on our titles