Community Experience Distilled

Cassandra Data Modeling and Analysis

Design, build, and analyze your data intricately using Cassandra

C.Y. Kan

[PACKT] PUBLISHING open source* community experience distilled

Cassandra Data Modeling and Analysis

Design, build, and analyze your data intricately using Cassandra

C.Y. Kan

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Cassandra Data Modeling and Analysis

First published: December 2014

Production reference: 1171214

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

ISBN 978-1-78398-888-4

www.packtpub.com

Cover image by Suyog Gharat (yogiee@me.com)

Credits

Author
C.Y. Kan

Reviewers
Christopher Bailey
Swathi Kurunji
Robert McFrazier
Iuliia Proskurnia
Alexander Shvid
Mikhail Stepura

Commissioning Editor
Akram Hussain

Acquisition Editor
Owen Roberts

Content Development Editor
Manasi Pandire

Technical Editor
Vijin Boricha

Copy Editors
Deepa Nambiar
Vikrant Phadkay
Rashmi Sawant

Project Coordinator
Leena Purkait

Proofreaders
Maria Gould
Samantha Lyon

Indexer
Mariammal Chettiyar

Graphics
Abhinash Sahu

Production Coordinator
Arvindkumar Gupta

Cover Work
Arvindkumar Gupta

About the Author

C.Y. Kan is an expert in system integration and has over 20 years of IT experience, which includes 15 years of project management and an architect role in many large-scale international projects. He received a Bachelor's degree from the University of Hong Kong and later a Master's degree from the University of Technology, Sydney. He holds many professional qualifications such as PMP, PRINCE2 Practitioner, PMI-ACP, Scrum Master, CISSP, TOGAF9, and is a Certified SOA Architect.

Mr. Kan is an Assistant Vice President now working for PCCW Solutions Limited, Hong Kong. He has expertise and interests in software technologies and development methodologies, which includes Enterprise architecture, Service-oriented architecture, Java-related technologies, traditional and NoSQL database technologies, Cloud computing, Big Data, mobile application development, agile software development, and various kinds of project management methodologies.

Mr. Kan is often invited by Project Management Institute Hong Kong Chapter to teach courses on cloud computing, Big Data, service-oriented architecture, business process management, and agile software development. He is also the author of a video e-learning course, *Cassandra Administration*, *Packt Publishing*, which was published last year.

I would like to thank my family, Cally and Kobe, for all that they have done for me.

About the Reviewers

Christopher Bailey is a senior technical researcher at the University of Bristol. He has always been interested in technology and computing from an early age and followed his passion throughout undergraduate and postgraduate levels and obtained a PhD from the University of Southampton. Subsequently, he was drawn more to the application of academic theory in practice. He now works with academics and researchers to bring the latest advances in software engineering to play within an academic setting.

Throughout his career, Christopher has been developing tools and applications for a diverse range of projects, such as undergraduate e-learning platforms, pedagogical tools for lecturers, video repositories for humanities, enterprise-level application integrations, and location-aware crowdsourcing apps. More recently, he has been one of the lead developers on the national online survey tool BOS (`https://www.onlinesurveys.ac.uk`), which uses at its core, the performance throughput of Cassandra to help store and manage the large number of responses generated by the service.

Christopher lives in Bristol with his wife, and when he is not reviewing books, he can be found tinkering with his 3D printer, planning his next holiday, or immersed in his (unending) pursuit to learn Chinese.

Swathi Kurunji is a PhD student in the Computer Science department at the University of Massachusetts Lowell (UMass Lowell), USA. She has shown keen interest in database systems. Her PhD research involves Query optimization, Big Data Analysis, Data Warehouses, and cloud computing. She has shown excellence in her field of study through research publications in international conferences and journals. She has also received awards and scholarships at UMass Lowell for research and academics.

Swathi has a Master of Science degree in Computer Science from UMass Lowell and a Bachelor of Engineering degree in Information Science from KVGCE, India. During her studies at UMass Lowell, she has worked as a teaching assistant, where she helped professors in teaching classes and labs, designing projects, and grading exams.

Swathi has worked as a software development intern with IT companies such as EMC and SAP. At EMC, she gained experience on Apache Cassandra data modeling and performance analysis. At SAP, she gained experience on infrastructure/cluster management components of the Sybase-IQ product. She has worked with Wipro Technologies in India as a project engineer managing application servers.

She has a wide experience with database systems such as Apache Cassandra, Sybase-IQ, Oracle, MySQL, and MSAccess. Her interests include Software Design and Development, Big Data analysis, optimization of databases, and Cloud computing. Her Linkedin profile is `https://www.linkedin.com/pub/swathi-kurunji/49/578/30a/`.

I would like to thank my hubby and my family for all the support.

Robert E. McFrazier is an open source developer, manager, trainer, and architect. Having started with web development, he was able to progress in his career in multiple roles as a developer, trainer, build/release engineer, architect, and manager. He has been working primarily in PHP web development, but he also has experience in Java development, AWS cloud, message queues, Hadoop, Cassandra, and creating high volume SOAP/REST API services.

He has previously reviewed *Learning Cassandra for Administrators*, *Packt Publishing*.

Robert has worked for many software companies, including Nordstrom.com, InfoSpace, Clear, RealNetworks, Arise Virtual Solutions, T-Mobile, and Disney.

Iuliia Proskurnia is a PhD candidate at EDIC doctoral school at EPFL with specialization in distributed information systems. Currently, she is working on the Specific Domain Knowledge Base Construction. She was awarded a fellowship by the EPFL to conduct her doctoral research. She is a winner of the Google Anita Borg scholarship and Google Ambassador at KTH (2012-2013). She obtained a Master's diploma in Distributed computing (2013) from KTH (Stockholm, Sweden), UPC (Barcelona, Spain). During her master thesis, she designed and implemented a unique, real-time, low latency reliable, and strongly consistent distributed data store for stock exchange environment at NASDAQ OMX. Previously, she has obtained Master's and Bachelor's diploma in Computer Science with honors from National Technical University of Ukraine "KPI". Her master thesis was about fuzzy portfolio management in a priory uncertain condition. This period was productive for her in terms of publications and conference presentations. During her studies in Ukraine, she was awarded named scholarships several times.

Alexander Shvid is a Data Grid Architect with more than 10 years of software experience in Fortune 500 companies with focus on financial institutions. He has worked in the USA, Argentina, and Russia and has many architect and developer certifications, including those from Pivotal/Spring Source and Oracle. He is a regular speaker at user groups and conferences around the world, such as the Java One and Cassandra meet ups.

Alex works for PayPal in Silicon Valley, developing low-latency Big Data real-time solutions. His major specialization is in Big data and Fast data frameworks adoption for enterprise environments. He participated in an open source project, Spring Data Cassandra module, and developed Dell Crowbar automation barclamp for Cassandra. His recent projects in Fast data include integration of Gemfire from Pivotal as an event processing middleware solution and caching system for Gire (Buenos Aires, Argentina), Visa (Foster City, CA, USA), VMWare (Palo Alto, CA, USA) as well the Coherence from Oracle for Analog (Boston, MA, USA), RCI (Parsippany, NJ, USA) and custom data grid solution for Deutsche Bank (New York, NY, USA).

When he is not working, Alex can usually be found hiking with his wife along the Coastal Trail in San Francisco Bay Area.

Mikhail Stepura is software engineer with over 13 years of experience in developing software. He is passionate about programming, performance, scalability, and all technology-related things. He is always intrigued by new technology and enjoys learning new things. He enjoys contributing to open source projects in his spare time.

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: Bird's Eye View of Cassandra **7**

What is NoSQL? **8**

NoSQL Database types 12

Key/value pair store 12

Column-family store 13

Document-based repository 13

Graph database 14

What is Cassandra? **14**

Google BigTable 15

Amazon Dynamo 16

Cassandra's high-level architecture **17**

Partitioning 18

Replication 19

Snitch 20

Seed node 20

Gossip and Failure detection 21

Write path 21

Read path 23

Repair mechanism 24

Features of Cassandra **25**

Summary **26**

Chapter 2: Cassandra Data Modeling **27**

What is unique to the Cassandra data model? **28**

Map and SortedMap 29

Logical data structure 29

Column 30

Row 30

Column family 31

Keyspace 32
Super column and super column family 33
Collections 34
No foreign key 34
No join 34
No sequence 35
Counter 35
Time-To-Live 35
Secondary index 36
Modeling by query 36
Relational version 36
Cassandra version 38
Data modeling considerations 44
Data duplication 44
Sorting 44
Wide row 44
Bucketing 44
Valueless column 45
Time-series data 45
Cassandra Query Language 45
Summary 46
Chapter 3: CQL Data Types 47
Introduction to CQL 47
CQL statements 47
CQL command-line client – cqlsh 48
Native data types 49
Cassandra implementation 50
A not-so-long example 51
ASCII 53
Bigint 54
BLOB 54
Boolean 55
Decimal 55
Double 55
Float 55
Inet 56
Int 57
Text 57
Timestamp 59
Timeuuid 61
UUID 62

Varchar 62
Varint 62
Counter 63
Collections 64
Set 66
List 66
Map 67
User-defined type and tuple type 67
Summary 69
Chapter 4: Indexes 71
Primary index 71
Compound primary key and composite partition key 74
Time-series data 79
Partitioner 81
Murmur3Partitioner 81
RandomPartitioner 81
ByteOrderedPartitioner 82
Paging and token function 82
Secondary indexes 83
Multiple secondary indexes 85
Secondary index do's and don'ts 86
Summary 87
Chapter 5: First-cut Design and Implementation 89
Stock Screener Application 90
An introduction to financial analysis 90
Stock quote data 91
Initial data model 93
Processing flow 94
System design 96
The operating system 96
Java Runtime Environment 97
Java Native Access 97
Cassandra version 97
Programming language 98
Cassandra driver 99
The integrated development environment 99
The system overview 100
Code design and development 101
Data Feed Provider 101
Collecting stock quote 101

Transforming data 103
Storing data in Cassandra 104
Putting them all together 107
Stock Screener 109
Data Scoper 109
Time-series data 111
The screening rule 111
The Stock Screener engine 111
Test run 114
Summary 115
Chapter 6: Enhancing a Version 117
Evolving the data model 117
The enhancement approach 118
Watch List 120
Alert List 121
Adding the descriptive stock name 122
Queries on alerts 123
Enhancing the code 125
Data Mapper and Archiver 125
Stock Screener Engine 129
Queries on Alerts 133
Implementing system changes 137
Summary 138
Chapter 7: Deployment and Monitoring 139
Replication strategies 139
Data replication 140
SimpleStrategy 140
NetworkTopologyStrategy 141
Setting up the cluster for Stock Screener Application 143
System and network configuration 143
Global settings 144
Configuration procedure 145
Legacy data migration procedure 146
Deploying the Stock Screener Application 148
Monitoring 150
Nodetool 150
JMX and MBeans 151
The system log 153
Performance tuning 155
Java virtual machine 155
Caching 156
Partition key cache 156
Row cache 156

Monitoring cache 156
Enabling/disabling cache 157
Summary 158
Chapter 8: Final Thoughts 159
Supplementary information 159
Client drivers 159
Security 161
Authentication 161
Authorization 161
Inter-node encryption 161
Backup and restore 162
Useful websites 163
Apache Cassandra official site 163
PlanetCassandra 164
DataStax 165
Hadoop integration 167
Summary 168
Index 169

Preface

If you are asked about the top five hot topics in today's IT world, Big Data will be one of them. In fact, Big Data is a broad buzzword that encompasses a lot of components, and NoSQL database is an indispensable component of it.

Cassandra is one of the most popular NoSQL databases. Nowadays, it offers a vast number of technological advantages that enable you to break through the limits of a traditional relational database in a web-scale or cloud environment. However, designing a high-performance Cassandra data model is not an intuitive task, especially for those of you who have been so involved with relational data modeling for many years. It is too easy to adopt a suboptimal way to simply mimick a relational data model in Cassandra.

This book is about how to design a better model that leverages the superb scalability and flexibility that Cassandra provides. Cassandra data modeling will help you understand and learn the best practices to unleash the power of Cassandra to the greatest extent.

Starting with a quick introduction to Cassandra, we will guide you step-by-step from the fundamental data modeling approach, selecting data types, designing data model, and choosing suitable keys and indexes to a real-world application by applying these best practices.

Although the application is small and rudimentary, you will be involved in the full development life cycle. You will go through the design considerations of how to come up with a flexible and sustainable data model for a stock market technical-analysis application written in Python. Business changes continually and so does a data model. You will also learn the techniques of evolving a data model in order to address the new business requirements.

Running a web-scale Cassandra cluster requires many careful thoughts, such as evolving of the data model, performance tuning, and system monitoring, which are also supplemented at your fingertips.

The book is an invaluable tutorial for anyone who wants to adopt Cassandra in practical use.

What this book covers

Chapter 1, *Bird's Eye View of Cassandra*, explains what Big Data and NoSQL are and how they are related to Cassandra. Then, it introduces the important Cassandra architectural components and the features that come along with them. This chapter lays the fundamental knowledge, concepts, and capabilities that all the following chapters refer to.

Chapter 2, *Cassandra Data Modeling*, explains why data modeling using Cassandra is so different from the relational data modeling approach. The commonly used technique in making a Cassandra data model, modeling by query, is explained with ample examples in more detail.

Chapter 3, *CQL Data Types*, walks you through the Cassandra built-in data types, with many examples that compose of a data model. The internal physical storage of these data types is also provided in order to help you understand and visualize what is going on in Cassandra.

Chapter 4, *Indexes*, discusses the design of the primary and secondary indexes. Cassandra data model is based on the queries to be used in order to access them. Primary and secondary indexes are different from those with the same names in a relational model, which often causes confusion to many new Cassandra data modelers. This chapter clearly explains when and which of them can be used correctly.

Chapter 5, *First-cut Design and Implementation*, conducts the design of the data model for a simple stock market technical-analysis application. This chapter starts by defining the business problem to be solved, the functions to be provided, and proceeds to design the data model and the programs. By the end of this chapter, the technical-analysis application can run on Cassandra to provide real functions.

Chapter 6, *Enhancing a Version*, describes a number of enhancements on the technical-analysis application built in *Chapter 5*, *First-cut Design and Implementation*. The enhancements involve both data model and code changes. This chapter illustrates how Cassandra's data model can flexibly embrace the changes.

Chapter 7, *Deployment and Monitoring*, discusses several important considerations to migrate an application on Cassandra from a non-production environment to a production one. The considerations include replication strategy, data migration, system monitoring, and basic performance tuning. This chapter highlights these pertinent factors in a real-life production Cassandra cluster deployment.

Chapter 8, *Final Thoughts*, supplements additional topics in application development with Cassandra, such as client drivers that are available for different programming languages, built-in security features, backup, and restore. It also recommends a few useful websites for further information. Finally, a quick review of all the chapters wraps up the book.

What you need for this book

The readers are advised to go through Cassandra basics before starting the journey of developing a data model and an application. An excellent book to start with is *Learning Cassandra for Administrators*, *Vijay Parthasarathy*, *Packt Publishing*.

Though having prior knowledge of Cassandra is not mandatory, anybody with some background in any application design and implementation and relational data modeling experience will find it easy to relate to this book.

The book refers to the recent Cassandra 2.0.*x* Version. Some of the code examples refer to the features available in Cassandra Query Language 3 (CQL3).

In addition, as Python is used to develop the sample application, an elementary programming knowledge of Python and NumPy is sufficient for a smooth read of this book. The preferred Python Version is 2.7.*x*. A good book to familiarize yourself with Python and NumPy is *NumPy Cookbook*, *Ivan Idris*, *Packt Publishing*.

All the tools and packages required to get the sample up and running are freely available on the Internet. To get a hands-on experience of the sample application, a computer running on Ubuntu Linux is suggested.

Who this book is for

If you are interested in Cassandra and want to develop real-world analysis applications, then this book is perfect for you.

Using Cassandra as a web-scale or Cloud NoSQL database backend enables your architecture and application systems to be truly ready for Big Data. After reading this book, you will know how to handle and model the data to unleash the power of Cassandra.

Conventions

In this book, you will find a number of styles of text 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 SMA can be easily computed by the `rolling_mean()` function, as shown in `chapter05_007.py`."

A block of code is set as follows:

```
# -*- coding: utf-8 -*-
# program: chapter05_007.py

import pandas as pd

## function to compute a Simple Moving Average on a DataFrame
## d: DataFrame
## prd: period of SMA
## return a DataFrame with an additional column of SMA
def sma(d, prd):
    d['sma'] = pd.rolling_mean(d.close_price, prd)
    return d
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
CREATE TABLE stock_ticker (
symbol varchar references stock_symbol(symbol),
tick_date varchar,
open decimal,
high decimal,
```

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

```
ubtc01:~$ nodetool status
```

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: "The middle panel on the right-hand side is the **IPython console** that runs the code."

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.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: `https://www.packtpub.com/sites/default/files/downloads/8884OS_ImageBundle.pdf`.

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
Bird's Eye View of Cassandra

Imagine that we have turned back the clock to the 1990s and you an application architect. Whenever you were required to select a suitable database technology for your applications, what kind of database technology would you choose? I bet 95 percent (or more) of the time you would select relational databases.

Relational databases have been the most dominating data management solution since the 1970s. At that time, the application system was usually silo. The users of the application and their usage patterns were known and under control. The workload that had to be catered for by the relational database could be determined and estimated. Apart from the workload consideration, the data model can also be structured in normalized forms as recommended by the relational theory. Moreover, relational databases provide many benefits such as support of transactions, data consistency, and isolation. Relational databases just fit perfectly for the purposes. Therefore, it is not difficult to understand why the relational database has been so popular and why it is the de facto standard for persistent data stores in application development.

Nonetheless, with the proliferation of the Internet and the numerous web applications running on it, the control of the users and their usage patterns (hence the scale), the workload generated, and the flexibility of the data model were gone. Typical examples of these web applications were global e-commerce websites, social media sites, video community websites, and so on. They generated a tremendous amount of data in a very short period of time. It should also be noted that the data generated by these applications were not only structured, but also semi-structured and even unstructured. Since relational databases were the de facto standard at that time, developers and architects did not have many alternatives but were forced to tweak them to support these web applications, even though they knew that relational databases were suboptimal and had many limitations. It became apparent that a different kind of enabling technology should be found to break through the challenges.

We are in an era of information explosion, as a result of the ever-increasing amount of user-generated data and content on the Web and mobile applications. The generated data is not only large in volume and fast in velocity but it is also diversified in variety. Such rapidly growing data of different varieties is often termed as **Big Data**.

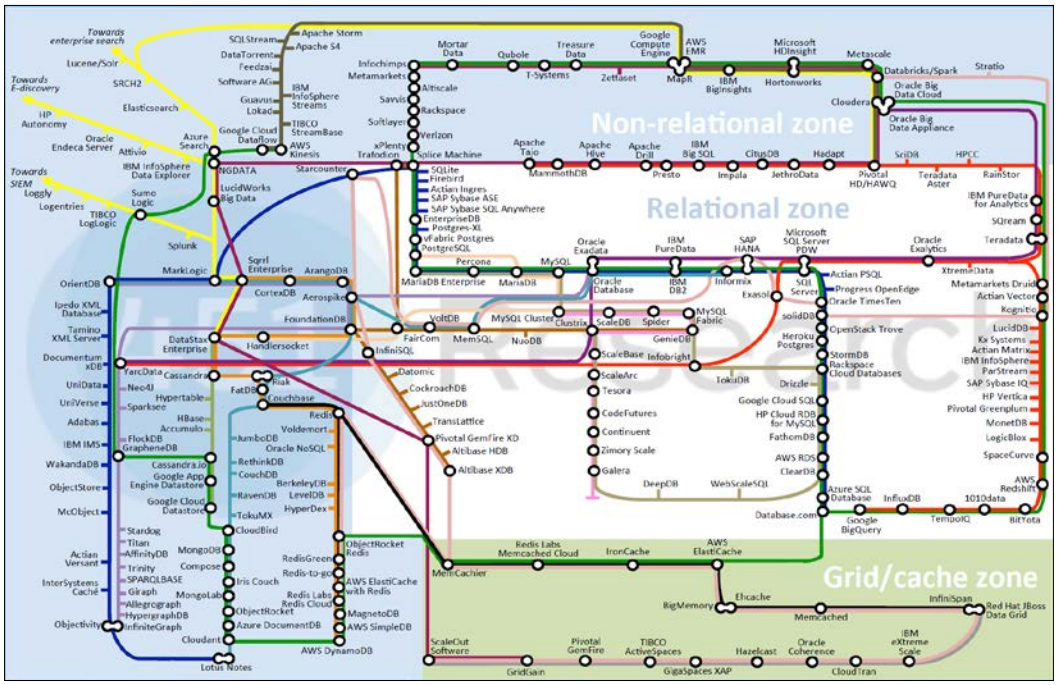
No one has a clear, formal definition of Big Data. People, however, unanimously agree that the most fundamental characteristics of Big Data are related to large volume, high velocity, and great variety. Big Data imposes real, new challenges to the information systems that have adopted traditional ways of handling data. These systems are not designed for web-scale and for being enhanced to do so, cost effectively. Due to this, you might find yourself asking whether or not we have any alternatives.

Challenges come with opportunities on the flip side. A new breed of data management products was born. The most recent answer to the question in the last paragraph is NoSQL.

What is NoSQL?

The need to tackle the Big Data challenges has led to the emergence of new data management technologies and techniques. Such technologies and techniques are rather different from the ubiquitous relational database technology that has been used for over 40 years. They are collectively known as **NoSQL**.

NoSQL is an umbrella term for the data stores that are not based on the relational data model. It encompasses a great variety of many different database technologies and products. As shown in the following figure, The **Data Platforms Landscape Map**, there are over 150 different database products that belong to the non-relational school as mentioned in `http://nosql-database.org/`. Cassandra is one of the most popular ones. Other popular NoSQL database products are, just to name a few, MongoDB, Riak, Redis, Neo4j, so on and so forth.



The Data Platforms Landscape Map (Source: 451 Research)

So, what kinds of benefits are provided by NoSQL? When compared to the relational database, NoSQL overcomes the weaknesses that the relational data model does not address well, which are as follows:

- Huge volume of structured, semi-structured, and unstructured data
- Flexible data model (schema) that is easy to change
- Scalability and performance for web-scale applications
- Lower cost
- Impedance mismatch between the relational data model and object-oriented programming
- Built-in replication
- Support for agile software development

Limitations of NoSQL Databases

Many NoSQL databases do not support transactions. They use replication extensively so that the data in the cluster might be momentarily inconsistent (although it is eventually consistent). In addition, the range queries are not available in NoSQL databases. Furthermore, a flexible schema might lead to problems with efficient searches.

The huge volume of structured, semi-structured, and unstructured data was mentioned earlier. What I want to dive deeper into here is that different NoSQL databases provide different solutions for each of them. The primary factor to be considered is the NoSQL database type, which will be introduced in the subsequent section.

All NoSQL databases provide a flexible data model that is easy to change and some might be even schemaless. In a relational database, the relational data model is called schema. You need to understand the data to be stored in a relational database, design the data model according to the relational database theory, and define the schema upfront in the relational database before you can actually store data inside it. It is a very structured approach for structured data. It is a prescriptive data modeling process. It is absolutely fine if the data model is stable, because there are not many changes required. But what if the data model keeps changing in the future and you do not know what needs to be changed? You cannot prescribe comprehensively in advance. It leads to many inevitable remedies; say, data patching for example, to change the schema.

Conversely, in NoSQL databases, you need not prescribe comprehensively. You only need to describe what is to be stored. You are not bound by the relational database theory. You are allowed to change the data model whenever necessary. The data model is schemaless and is a living object. It evolves as life goes on. It is a descriptive data modeling process.

Scalability and performance for web-scale applications refer to the ability of the system to be scaled, preferably horizontally, to support web-scale workloads without considerably deteriorating system performance. Relational databases can only be scaled out to form a cluster consisting of a very small number of nodes. It implies the rather low ceiling imposed on these web-scale applications using relational databases. In addition, changing the schema in a clustered relational database is a big task of high complexity. The processing power required to do this is so significant that the system performance cannot be unaffected. Most NoSQL databases were created to serve web-scale applications. They natively support horizontal scaling without very little degrade on the performance.

Now let us talk about money. Traditionally, most high-end relational databases are commercial products that demand their users to pay huge software license fees. Besides, to run these high-end relational databases, the underlying hardware servers are usually high-end as well. The result is that the hardware and software costs of running a powerful relational database are exceptionally large. In contrast, NoSQL databases are open source and community-driven in a majority, meaning that you need to pay the software license cost, which is an order of magnitude less than other databases. NoSQL databases are able to run on commodity machines that will lead to a possible churn, or crashes. Therefore, the machines are usually configured to be a cluster. High-end hardware servers are not needed and so the hardware cost is tremendously reduced. It should be noted that when NoSQL databases are put into production, some cost of the support is still required but it is definitely much less when compared to that of commercial products.

There exists a generation gap between the relational data model and object-oriented programming. The relational data model was the product of 1970s, whereas object-oriented programming became very popular in 1990s. The root cause, known as impedance mismatch, is an inherent difficulty of representing a record or a table in a relational data model with the object-oriented model. Although there are resolutions for this difficulty, most application developers still feel very frustrated to bring the two together.

Impedance Mismatch

Impedance mismatch is the difference between the relational model and the in-memory data structures that are usually encountered in object-oriented programming languages.

Built-in replication is a feature that most NoSQL databases provide to support high availability in a cluster of many nodes. It is usually automatic and transparent to the application developers. Such a feature is also available in relational databases, but the database administrators must struggle to configure, manage, and operate it by themselves.

Finally, relational databases do not support agile software development very well. Agile software development is iterative by nature. The software architecture and data model emerge and evolve as the project proceeds in order to deliver the product incrementally. Hence, it is conceivable that the need of changing the data model to meet the new requirements is inevitably frequent. Relational databases are structured and do not like changes. NoSQL can provide such flexibility for agile software development teams by virtue of its schemaless characteristic. Even better, NoSQL databases usually allow the changes to be implemented in real time without any downtime.

NoSQL Database types

Now you know the benefits of NoSQL databases, but the products that fall under the NoSQL databases umbrella are quite varied. How can you select the right one for yourself among so many NoSQL databases? The selection criteria of which NoSQL database fits your needs is really dependent on the use cases at hand. The most important factor to consider here is the NoSQL database type, which can be subdivided into four main categories:

- Key/value pair store
- Column-family store
- Document-based repository
- Graph database

The NoSQL database type dictates the data model that you can use. It is beneficial to understand each of them deeper.

Key/value pair store

Key/value pair is the simplest NoSQL database type. Key/value store is similar to the concept of Windows registry, or in Java or C#, a map, a hash, a key/value pair. Each data item is represented as an attribute name, also a key, together with its value. It is also the basic unit stored in the database. Examples of the NoSQL databases of key/value pair type are **Amazon Dynamo**, **Berkeley DB**, **Voldemort** and **Riak**.

Internally, key/value pairs are stored in a data structure called **hashmap**. Hashmap is popular because it provides very good performance on accessing data. The key of a key/value pair is unique and can be searched very quickly.

Key/value pair can be stored and distributed in the disk storage as well as in memory. When used in memory, it can be used as a cache, which depends on the caching algorithm, can considerably reduce disk I/O and hence boost up the performance significantly.

On the flip side, key/value pair has some drawbacks, such as lack of support of range queries, no way to operate on multiple keys simultaneously, and possible issues with load balancing.

Column-family store

A column in this context is not equal to a column in a relational table. In the NoSQL world, a column is a data structure that contains a key, value, and timestamp. Thus, it can be regarded as a combination of key/value pair and a timestamp. Examples are **Google BigTable**, **Apache Cassandra**, and **Apache HBase**. They provide optimized performance for queries over very large datasets.

Column-family store is basically a multi-dimensional map. It stores columns of data together as a row, which is associated with a row key. This contrasts with rows of data in a relational database. Column-family store does not need to store null columns, as in the case of a relational database and so it consumes much less disk space. Moreover, columns are not bound by a rigid schema and you are not required to define the schema upfront.

The key component of a column is usually called the primary key or the row key. Columns are stored in a sorted manner by the row key. All the data belonging to a row key is stored together. As such, read and write operations of the data can be confined to a local node, avoiding unnecessary inter-node network traffic in a cluster. This mechanism makes the data lookup and retrieval extremely efficient.

Obviously, a column-family store is not the best solution for systems that require ACID transactions and it lacks the support for aggregate queries provided by relational databases such as `SUM()`.

Document-based repository

Document-based repository is designed for documents or semi-structured data. The basic unit of a document-based repository associates each key, a primary identifier, with a complex data structure called a document. A document can contain many different key-value pairs, or key-array pairs, or even nested documents. Therefore, document-based repository does not adhere to a schema. Examples are **MongoDB** and **CouchDB**.

In practice, a document is usually a loosely structured set of key/value pairs in the form of **JavaScript Object Notation** (**JSON**). Document-based repository manages a document as a whole and avoids breaking up a document into fragments of key/value pairs. It also allows document properties to be associated with a document.

As a document database does not adhere to a fixed schema, the search performance is not guaranteed. There are generally two approaches to query a document database. The first is to use materialized views (such as CouchDB) that are prepared in advance. The second is to use indexes defined on the document values (such as MongoDB) that behave in the same way as a relational database index.

Graph database

Graph databases are designed for storing information about networks, such as a social network. A graph is used to represent the highly connected network that is composed of nodes and their relationships. The nodes and relationships can have individual properties. The prominent graph databases include **Neo4J** and **FlockDB**.

Owing to the unique characteristics of a graph, graph databases commonly provide APIs for rapid traversal of graphs.

Graph databases are particularly difficult to be scaled out with sharding because traversing a graph of the nodes on different machine does not provide a very good performance. It is also not a straightforward operation to update all or a subset of the nodes at the same time.

So far, you have grasped the fundamentals of the NoSQL family. Since this book concentrates on Apache Cassandra and its data model, you need to know what Cassandra is and have a basic understanding of what its architecture is, so that you can select and leverage the best available options when you are designing your NoSQL data model and application.

What is Cassandra?

Cassandra can be simply described in a single phrase: a massively scalable, highly available open source NoSQL database that is based on peer-to-peer architecture.

Cassandra is now 5 years old. It is an active open source project in the Apache Software Foundation and therefore it is known as Apache Cassandra as well. Cassandra can manage huge volume of structured, semi-structured, and unstructured data in a large distributed cluster across multiple data centers. It provides linear scalability, high performance, fault tolerance, and supports a very flexible data model.

Netflix and Cassandra

One very famous case study of Cassandra is Netflix's move to replace their Oracle SQL database to Cassandra running on cloud. As of March 2013, Netflix's Cassandra deployment consists of 50 clusters with over 750 nodes. For more information, please visit the case study at `http://www.datastax.com/wp-content/uploads/2011/09/CS-Netflix.pdf`.

In fact, many of the benefits that Cassandra provides are inherited from its two best-of-breed NoSQL parents, Google BigTable and Amazon Dynamo. Before we go into the details of Cassandra's architecture, let us walk through each of them first.

Google BigTable

Google BigTable is Google's core technology, particularly addressing data persistence and management on web-scale. It runs the data stores for many Google applications, such as Gmail, YouTube, and Google Analytics. It was designed to be a web-scale data store without sacrificing real-time responses. It has superb read and write performance, linear scalability, and continuous availability.

Google BigTable is a sparse, distributed, persistent, multidimensional sorted map. The map is indexed by a row key.

Despite the many benefits Google BigTable provides, the underlying design concept is really simple and elegant. It uses a persistent commitlog for every data write request that it receives and then writes the data into a memory store (acting as a cache). At regular intervals or when triggered by a particular event, the memory store is flushed to persistent disk storage by a background process. This persistent disk storage is called **Sorted String Table**, or **SSTable**. The SSTable is immutable meaning that once it has been written to a disk, it will never be changed again. The word *sorted* means that the data inside the SSTable is indexed and sorted and hence the data can be found very quickly. Since the write operation is log-based and memory-based, it does not involve any read operation, and therefore the write operation can be extremely fast. If a failure happens, the commitlog can be used to replay the sequence of the write operations to merge the data that persists in the SSTables.

Read operation is also very efficient by looking up the data in the memory store and the indexed SSTables, which are then merged to return the data.

All the above-mentioned Google BigTable brilliances do come with a price. Because Google BigTable is distributed in nature, it is constrained by the famous *CAP theorem*, stating the relationship among the three characteristics of a distributed system, namely Consistency, Availability, and Partition-tolerance. In a nutshell, Google BigTable prefers Consistency and Partition-tolerance to Availability.

The CAP theorem

CAP is an acronym of the three characteristics of a distributed system: Consistency, Availability, and Partition-tolerance. Consistency means that all the nodes in a cluster see the same data at any point in time. Availability means that every request that is received by a non-failing node in the cluster must result in a response. Partition-tolerance means that a node can still function when communication with other groups of nodes is lost. Originating from Eric A. Brewer, the theorem states that in a distributed system, only two out of the three characteristics can be attained at the most.

Google BigTable has trouble with Availability while keeping Consistency across partitioned nodes when failures happen in the cluster.

Amazon Dynamo

Amazon Dynamo is a proprietary key-value store developed by Amazon. It is designed for high performance, high availability, and continuous growth of data of huge volume. It is the distributed, highly available, fault-tolerant skeleton for Amazon. Dynamo is a peer-to-peer design meaning that each node is a peer and no one is a master who manages the data.

Dynamo uses data replication and auto-sharding across multiple nodes of the cluster. Imagine that a Dynamo cluster consists of many nodes. Every write operation in a node is replicated to two other nodes. Thus, there are three copies of data inside the cluster. If one of the nodes fails for whatever reason, there are still two copies of data that can be retrieved. Auto-sharding ensures that the data is partitioned across the cluster.

Auto-sharding

NoSQL database products usually support auto-sharding so that they can natively and automatically distribute data across the database cluster. Data and workload are automatically balanced across the nodes in the cluster. When a node fails for whatever reason, the failed node can be quickly and transparently replaced without service interruptions.

Dynamo focuses primarily on the high availability of a cluster and the most important idea is eventual consistency. While considering the CAP Theorem, Dynamo prefers Partition-tolerance and Availability to Consistency. Dynamo introduces a mechanism called **Eventual Consistency** to support consistency. Temporary inconsistency might occur in the cluster at a point in time, but eventually all the nodes will receive the latest consistent updates. Given a sufficiently long period of time without further changes, all the updates can be expected to propagate throughout the cluster and the replicas on all the nodes will be consistent eventually. In real life, an update takes only a fraction of a second to become eventually consistent. In other words, it is a trade-off between consistency and latency.

Eventual consistency

Eventual consistency is not inconsistency. It is a weaker form of consistency than the typical Atomic-Consistency-Isolation-Durability (ACID) type consistency is found in the relational databases. It implies that there can be short intervals of inconsistency among the replicated nodes during which the data gets updated among these nodes. In other words, the replicas are updated asynchronously.

Cassandra's high-level architecture

Cassandra runs on a peer-to-peer architecture which means that all nodes in the cluster have equal responsibilities except that some of them are seed nodes for other non-seed nodes to obtain information about the cluster during startup. Each node holds a partition of the database. Cassandra provides automatic data distribution and replication across all nodes in the cluster. Parameters are provided to customize the distribution and replication behaviors. Once configured, these operations are processed in the background and are fully transparent to the application developers.

Cassandra is a column-family store and provides great schemaless flexibility to application developers. It is designed to manage huge volume of data in a large cluster without a single point of failure. As multiple copies of the same data (replicas) are replicated in the cluster, whenever one node fails for whatever reason, the other replicas are still available. Replication can be configured to meet the different physical cluster settings, including data center and rack locations.

Any node in the cluster can accept read or write requests from a client. The node that is connected to a client with a request serves as the coordinator of that particular request. The coordinator determines which nodes are responsible for holding the data for the request and acts as a proxy between the client and the nodes.

Cassandra borrows the commitlog mechanism from Google BigTable to ensure data durability. Whenever a write data request is received by a node, it is written into the commitlog. The data that is being updated is then written to a memory structure, known as memtable. When the memtable is full, the data inside the memtable is flushed to a disk storage structure, SSTable. The writes are automatically partitioned by the row key and replicated to the other nodes holding the same partition.

Cassandra provides linear scalability, which means that the performance and capacity of the cluster is proportional to the number of nodes in it.

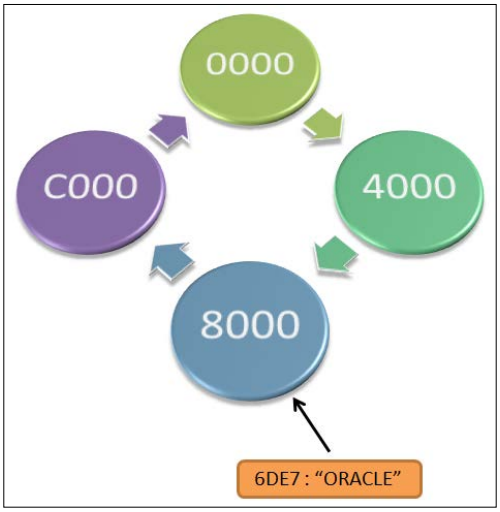
Partitioning

The ability to scale horizontally and incrementally is a Cassandra key design feature. To achieve this, Cassandra is required to dynamically partition the data over the set of nodes in the cluster.

A cluster is the outermost structure which is composed of nodes in Cassandra. It is also a container of keyspace. A keyspace in Cassandra is analogous to a schema in a relational database. Each Cassandra cluster has a system keyspace to keep system-wide metadata. It contains the replication settings which controls how the data is distributed and replicated in a cluster. Typically, one keyspace is assigned to one cluster but one cluster might contain more than one keyspace.

The smallest cluster in the theory contains a single node and a cluster of three or more nodes, which is much more practical. Each node holds a replica for the different range of data in partitions, and exchanges information across the cluster every second.

A client issues read or write requests to any node. The node that receives the request becomes a coordinator that acts as a proxy of the client to do the things as explained previously. Data is distributed across the cluster and the node addressing mechanism is called consistent hashing. Therefore, a cluster can be viewed as a ring of hash as each node in the cluster or the ring is assigned a single unique token so that each node is responsible for the data in the range from its assigned token to that of the previous node. For example, in the following figure, a cluster contains four nodes with unique tokens:



Cassandra's consistent hashing

Before Version 1.2, tokens were calculated and assigned manually and from Version 1.2 onwards, tokens can be generated automatically. Each row has a row key used by a partitioner to calculate its hash value. The hash value determines the node which stores the first replica of the row. The partitioner is just a hash function that is used for calculating a row key's hash value and it also affects how the data is distributed or balanced in the cluster. When a write occurs, the first replica of the row is always placed in the node with the key range of the token. For example, the hash value of a row key `ORACLE` is `6DE7` that falls in the range of 4,000 and 8,000 and so the row goes to the bottom node first. All the remaining replicas are distributed based on the replication strategy.

Consistent hashing

Consistent hashing allows each node in the cluster to independently determine which nodes are replicas for a given row key. It just involves hashing the row key, and then compares that hash value to the token of each node in the cluster. If the hash value falls in between a node's token, and the token of the previous node in the ring (tokens are assigned to nodes in a clockwise direction), that node is the replica for that row.

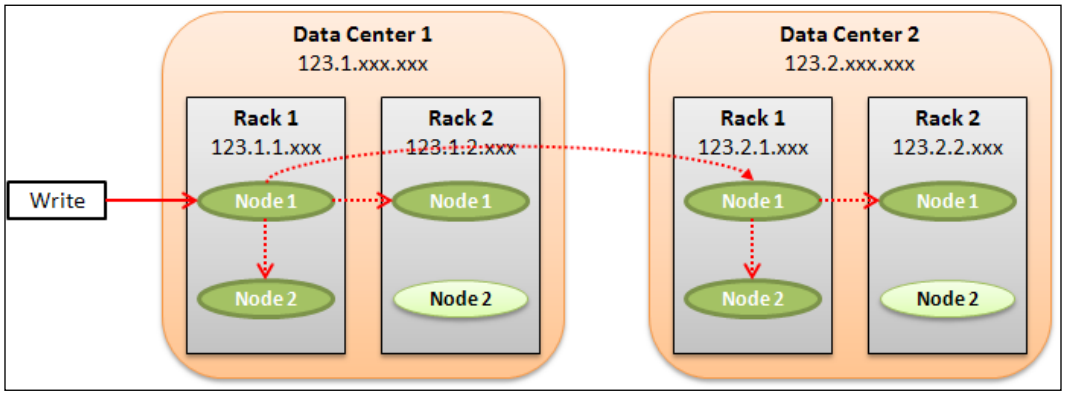
Replication

Cassandra uses replication to attain high availability and data durability. Each data is replicated at a number of nodes that are configured by a parameter called replication factor. The coordinator commands the replication of the data within its range. It replicates the data to the other nodes in the ring. Cassandra provides the client with various configurable options to see how the data is to be replicated, which is called replication strategy.

Replication strategy is the method of determining which nodes the replicas are placed in. It provides many options, such as rack-aware, rack-unaware, network-topology-aware, so on and so forth.

Snitch

A snitch determines which data centers and racks to go for in order to make Cassandra aware of the network topology for routing the requests efficiently. It affects how the replicas can be distributed while considering the physical setting of the data centers and racks. The node location can be determined by the rack and data center with reference to the node's IP address. An example of a cluster across two data centers is shown in the following figure, in order to illustrate the relationship among replication factor, replication strategy, and snitch in a better way:



Multiple data center cluster

Each data center has two racks and each rack contains two nodes respectively. The replication factor per data center is set to three here. With two data centers, there are six replicas in total. The node location that addresses the data center and rack locations are subject to the convention of IP address assignment of the nodes.

Seed node

Some nodes in a Cassandra cluster are designated as seed nodes for the others. They are configured to be the first nodes to start in the cluster. They also facilitate the bootstrapping process for the new nodes joining the cluster. When a new node comes online, it will talk to the seed node to obtain information about the other nodes in the cluster. The talking mechanism is called **gossip**. If a cluster is across multiple data centers, the best practice is to have more than one seed node per data center.

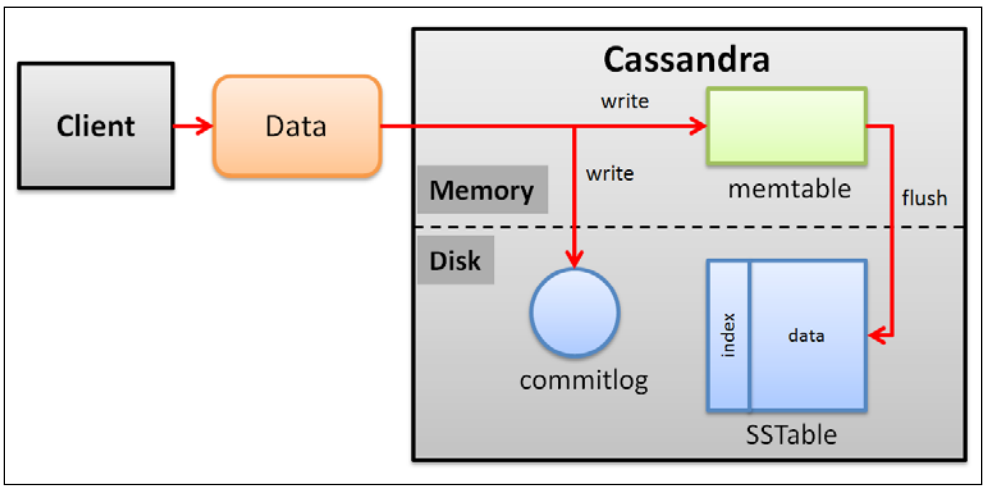
Gossip and Failure detection

Nodes need to communicate periodically (every second) to exchange state information (for example, dead or alive), about themselves and about other nodes they know about. Cassandra uses a gossip communication protocol to disseminate the state information, which is also known as epidemic protocol. It is a peer-to-peer communication protocol that provides a decentralized, periodic, and an automatic way for the nodes in the cluster to exchange the state information about themselves, and about other nodes they know about with up to three other nodes. Therefore, all nodes can quickly learn about all the other nodes in the cluster. Gossip information is also persisted locally by each node to allow fast restart.

Cassandra uses a very efficient algorithm, called *Phi Accrual Failure Detection Algorithm*, to detect the failure of a node. The idea of the algorithm is that the failure detection is not represented by a Boolean value stating whether a node is up or down. Instead, the algorithm outputs a value on the continuous suspicion level between dead and alive, on how confident it is that the node has failed. In a distributed environment, false negatives might happen due to the network performance, fluctuating workload, and other conditions. The algorithm takes all these factors into account and provides a probabilistic value. If a node has failed, the other nodes periodically try to gossip with it to see if it comes back online. A node can then determine locally from the gossip state and its history and adjust routes accordingly.

Write path

The following figure depicts the components and their sequence of executions that form a write path:



Cassandra write path

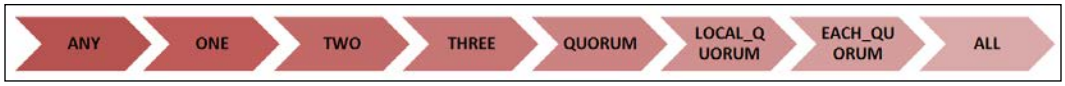
When a write occurs, the data will be immediately appended to the commitlog on the disk to ensure write durability. Then Cassandra stores the data in memtable, an in-memory store of hot and fresh data. When memtable is full, the memtable data will be flushed to a disk file, called SSTable, using sequential I/O and so random I/O is avoided. This is the reason why the write performance is so high. The commitlog is purged after the flush.

Due to the intentional adoption of sequential I/O, a row is typically stored across many SSTable files. Apart from its data, SSTable also has a primary index and a *bloom filter*. A primary index is a list of row keys and the start position of rows in the data file.

Bloom filter

Bloom filter is a sample subset of the primary index with very fast nondeterministic algorithms to check whether an element is a member of a set. It is used to boost the performance.

For write operations, Cassandra supports tunable consistency by various write consistency levels. The write consistency level is the number of replicas that acknowledge a successful write. It is tunable on a spectrum of write consistency levels, as shown in the following figure:



Cassandra write consistency levels

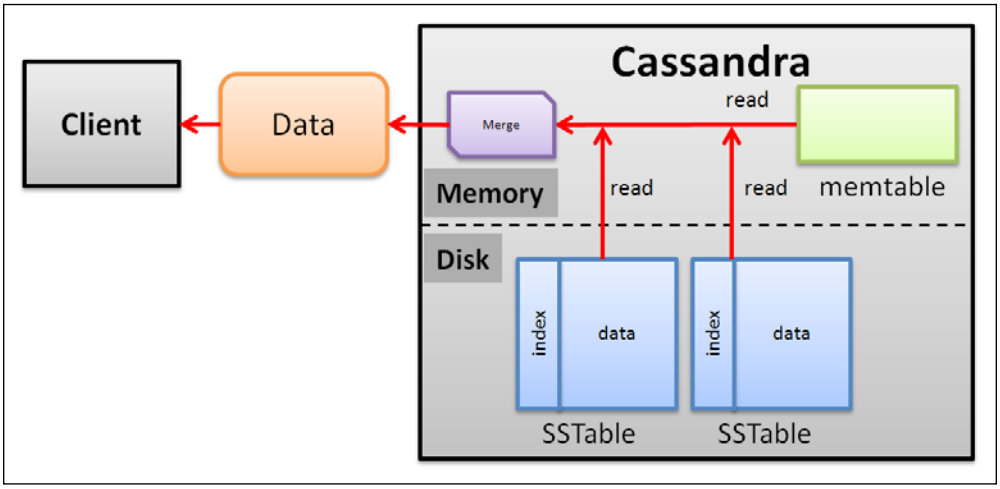
The following describes the terms in the figure:

- **ANY**: This is the lowest consistency (but highest availability)
- **ALL**: This is the highest consistency (but lowest availability)
- **ONE**: This gives at least one replica
- **TWO**: This gives at least two replicas
- **THREE**: This gives at least three replicas
- **QUORUM**: This ensures strong consistency by tolerating some level of failure, which is determined by *(replication_factor / 2) + 1* (rounded down to the nearest integer)
- **LOCAL_QUORUM**: This is for multi-data center and rack-aware without inter-data center traffic
- **EACH_QUORUM**: This is for multi-data center and rack-aware

The two extremes are the leftmost **ANY** which means weak consistency and the rightmost **ALL** means strong consistency. A consistency level of **THREE** is very common in practice. **QUORUM** can be chosen to be an optimum value, as calculated by the given formula. Here, the replication factor is the number of replicas of data on multiple nodes. Both **LOCAL QUORUM** and **EACH QUORUM** support multiple data centers and rack-aware write consistency with a slight difference as shown earlier.

Read path

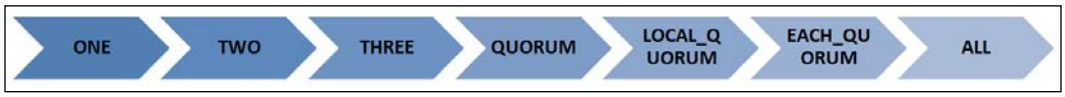
On the flip side, the following figure shows the components and their sequence of executions that form a read path:



Cassandra read path

When a read request comes in to a node, the data to be returned is merged from all the related SSTables and any unflushed memtables. Timestamps are used to determine which one is up-to-date. The merged value is also stored in a write-through row cache to improve the future read performance.

Similar to the write consistency levels, Cassandra also provides tunable read consistency levels, as shown in the following figure:



Cassandra read consistency levels

The following describes the terms in the figure:

- **ALL**: This is the highest consistency (but lowest availability)
- **ONE**: This gives at least one replica
- **TWO**: This gives at least two replicas
- **THREE**: This gives at least three replicas
- **QUORUM**: This ensures strong consistency by tolerating some level of failure, which is determined by *(replication_factor / 2) + 1* (rounded down to the nearest integer)
- **LOCAL_QUORUM**: This is for multi-data center and rack-aware without inter-data center traffic
- **EACH_QUORUM**: This is for multi-data center and rack-aware

Read consistency level is the number of replicas contacted for a successful, consistent read, almost identical to write consistency levels, except that **ANY** is not an option here.

Repair mechanism

There are three built-in repair mechanisms provided by Cassandra:

- Read repair
- Hinted handoff
- Anti-entropy node repair

During a read, the coordinator that is just the node connects and services the client, contacts a number of nodes as specified by the consistency level for data and the fastest replicas will return the data for a consistency check by in-memory comparison. As it is not a dedicated node, Cassandra lacks a single point of failure. It also checks all the remaining replicas in the background. If a replica is found to be inconsistent, the coordinator will issue an update to bring back the consistency. This mechanism is called **read repair**.

Hinted handoff aims at reducing the time to restore a failed node when rejoining the cluster. It ensures absolute write availability by sacrificing a bit of read consistency. If a replica is down at the time a write occurs, another healthy replica stores a hint. Even worse, if all the relevant replicas are down, the coordinator stores the hint locally. The hint basically contains the location of the failed replica, the affected row key, and the actual data that is being written. When a node responsible for the token range is up again, the hint will be handed off to resume the write. As such, the update cannot be read before a complete handoff, leading to inconsistent reads.

Another repair mechanism is called **anti-entropy** which is a replica synchronization mechanism to ensure up-to-date data on all nodes and is run by the administrators manually.

Features of Cassandra

In order to keep this chapter short, the following bullet list covers the great features provided by Cassandra:

- Written in Java and hence providing native Java support
- Blend of Google BigTable and Amazon Dynamo
- Flexible schemaless column-family data model
- Support for structured and unstructured data
- Decentralized, distributed peer-to-peer architecture
- Multi-data center and rack-aware data replication
- Location transparent
- Cloud enabled
- Fault-tolerant with no single point of failure
- An automatic and transparent failover
- Elastic, massively, and linearly scalable
- Online node addition or removal
- High Performance
- Built-in data compression
- Built-in caching layer
- Write-optimized
- Tunable consistency providing choices from very strong consistency to different levels of eventual consistency
- Provision of **Cassandra Query Language** (**CQL**), a SQL-like language imitating `INSERT`, `UPDATE`, `DELETE`, `SELECT` syntax of SQL
- Open source and community-driven

Summary

In this chapter, we have gone through a bit of history starting from the 1970s. We were in total control of the data models that were rather stable and the applications that were pretty simple. The relational databases were a perfect fit in the old days. With the emergence of object-oriented programming and the explosion of the web applications on the pervasive Internet, the nature of the data has been extended from structured to semi-structured and unstructured. Also, the application has become more complex. The relational databases could not be perfect again. The concept of Big Data was created to describe such challenges and NoSQL databases provide an alternative resolution to the relational databases.

NoSQL databases are of a wide variety. They provide some common benefits and can be classified by the NoSQL database type. Apache Cassandra is one of the NoSQL databases that is a blend of Google BigTable and Amazon Dynamo. The elegance of its architecture inherits from the DNA of these two parents.

In the next chapter, we will look at the flexible data model supported by Cassandra.

2
Cassandra Data Modeling

In this chapter, we will open the door to the world of Cassandra data modeling. We will briefly go through its building blocks, the main differences to the relational data model, and examples of constructing queries on a Cassandra data model.

Cassandra describes its data model components by using the terms that are inherited from the Google BigTable parent, for example, column family, column, row, and so on. Some of these terms also exist in a relational data model. They, however, have completely different meanings. It often confuses developers and administrators who have a background in the relational world. At first sight, the Cassandra data model is counterintuitive and very difficult to grasp and understand.

In the relational world, you model the data by creating entities and associating them with relationships according to the guidelines governed by the relational theories. It means that you can solely concentrate on the logical view or structure of the data without any considerations of how the application accesses and manipulates the data. The objective is to have a stable data model complying with the relational guidelines. The design of the application can be done separately. For instance, you can answer different queries by constructing different SQL statements, which is not of your concern during data modeling. In short, relational data modeling is process oriented, based on a clear separation of concerns.

On the contrary, in Cassandra, you reverse the above steps and always start from what you want to answer in the queries of the application. The queries exert a considerable amount of influence on the underlying data model. You also need to take the physical storage and the cluster topology into account. Therefore, the query and the data model are twins, as they were born together. Cassandra data modeling is result oriented based on a clear understanding of how a query works internally in Cassandra.

Owing to the unique architecture of Cassandra, many simple things in a relational database, such as sequence and sorting, cannot be presumed. They require your special handling in implementing the same. Furthermore, they are usually design decisions that you need to make upfront in the process of data modeling. Perhaps it is the cost of the trade-off for the attainment of superb scalability, performance, and fault tolerance.

To enjoy reading this book, you are advised to temporarily think in both relational and NoSQL ways. Although you may not become a friend of Cassandra, you will have an eye-opening experience in realizing the fact that there exists a different way of working in the world.

What is unique to the Cassandra data model?

If you want me to use just one sentence to describe Cassandra's data model, I will say it is a non-relational data model, period. It implies that you need to forget the way you do data modeling in a relational database.

You focus on modeling the data according to relational theories. However, in Cassandra and even in other NoSQL databases, you need to focus on the application in addition to the data itself. This means you need to think about how you will query the data in the application. It is a paradigm shift for those of you coming from the relational world. Examples are given in the subsequent sections to make sure that you understand why you cannot apply relational theories to model data in Cassandra.

Another important consideration in Cassandra data modeling is that you need to take the physical topology of a Cassandra cluster into account. In a relational database, the primary goal is to remove data duplication through normalization to have a single source of data. It makes a relational database ACID compliant very easily. The related storage space required is also optimized. Conversely, Cassandra is designed to work in a massive-scale, distributed environment in which ACID compliance is difficult to achieve, and replication is a must. You must be aware of such differences in the process of data modeling in Cassandra.

Map and SortedMap

In *Chapter 1, Bird's Eye View of Cassandra*, you learned that Cassandra's storage model is based on BigTable, a column-oriented store. A column-oriented store is a multidimensional map. Specifically, it is a data structure known as **Map**. An example of the declaration of map data structure is as follows:

```
Map<RowKey, SortedMap<ColumnKey, ColumnValue>>
```

The `Map` data structure gives efficient key lookup, and the sorted nature provides efficient scans. `RowKey` is a unique key and can hold a value. The inner `SortedMap` data structure allows a variable number of `ColumnKey` values. This is the trick that Cassandra uses to be schemaless and to allow the data model to evolve organically over time. It should be noted that each column has a client-supplied timestamp associated, but it can be ignored during data modeling. Cassandra uses the timestamp internally to resolve transaction conflicts.

In a relational database, column names can be only strings and be stored in the table metadata. In Cassandra, both `RowKey` and `ColumnKey` can be strings, long integers, Universal Unique IDs, or any kind of byte arrays. In addition, `ColumnKey` is stored in each column. You may opine that it wastes storage space to repeatedly store the `ColumnKey` values. However, it brings us a very powerful feature of Cassandra. `RowKey` and `ColumnKey` can store data themselves and not just in `ColumnValue`. We will not go too deep into this at the moment; we will revisit it in later chapters.

Universal Unique ID

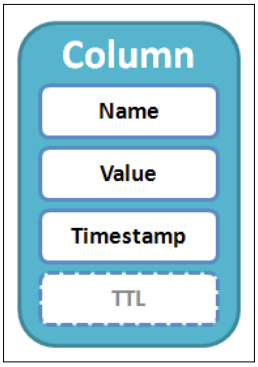
Universal Unique ID (**UUID**) is an **Internet Engineering Task Force** (**IETF**) standard, **Request for Comments** (**RFC**) 4122, with the intent of enabling distributed systems to uniquely identify information without significant central coordination. It is a 128-bit number represented by 32 lowercase hexadecimal digits, displayed in five groups separated by hyphens, for example: `0a317b38-53bf-4cad-a2c9-4c5b8e7806a2`

Logical data structure

There are a few logical building blocks to come up with a Cassandra data model. Each of them is introduced as follows.

Column

Column is the smallest data model element and storage unit in Cassandra. Though it also exists in a relational database, it is a different thing in Cassandra. As shown in the following figure, a column is a name-value pair with a timestamp and an optional **Time-To-Live** (**TTL**) value:

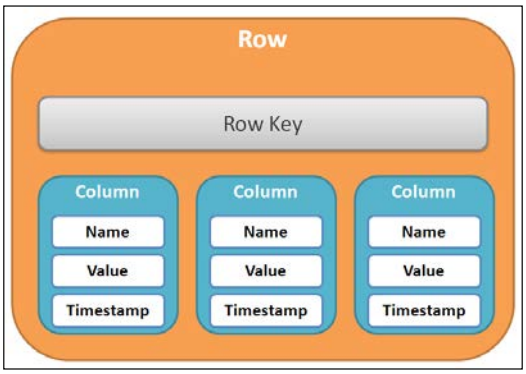


The elements of a column

The name and the value (`ColumnKey` and `ColumnValue` in `SortedMap` respectively) are byte arrays, and Cassandra provides a bunch of built-in data types that influence the sort order of the values. The timestamp here is for conflict resolution and is supplied by the client application during a write operation. Time-To-Live is an optional expiration value used to mark the column deleted after expiration. The column is then physically removed during compaction.

Row

One level up is a row, as depicted in the following figure. It is a set of orderable columns with a unique row key, also known as a primary key:



The structure of a row

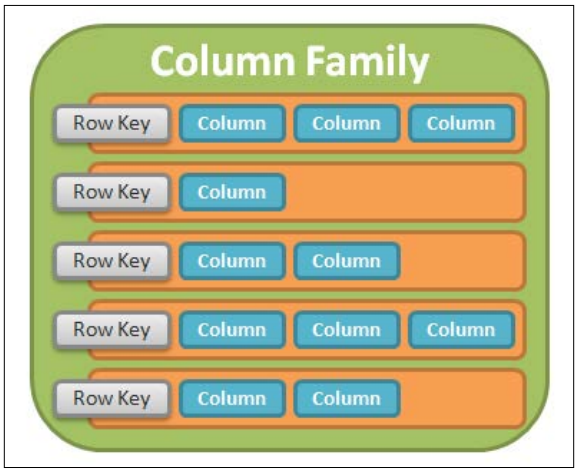
The row key can be any one of the same built-in data types as those for columns. What orderable means is that columns are stored in sorted order by their column names.

Sort order is extremely important because Cassandra cannot sort by value as we do in a relational database.

Different names in columns are possible in different rows. That is why Cassandra is both row oriented and column oriented. It should be remarked that there is no timestamp for rows. Moreover, a row cannot be split to store across two nodes in the cluster. It means that if a row exists on a node, the entire row exists on that node.

Column family

The next level up is a column family. As shown in the following figure, it is a container for a set of rows with a name:



The structure of a column family

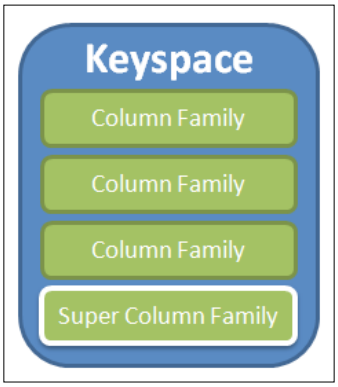
The row keys in a column family must be unique and are used to order rows. A column family is analogous to a table in a relational database, but you should not go too far with this idea. A column family provides greater flexibility by allowing different columns in different rows. Any column can be freely added to any column family at any time. Once again, it helps Cassandra be schemaless.

Columns in a column family are sorted by a comparator. The comparator determines how columns are sorted and ordered when Cassandra returns the columns in a query. It accepts long, byte and UTF8 for the data type of the column name, and the sort order in which columns are stored within a row.

Physically, column families are stored in individual files on a disk. Therefore, it is important to keep related columns in the same column family to save disk I/O and improve performance.

Keyspace

The outermost data model element is keyspace, as illustrated in the following figure:



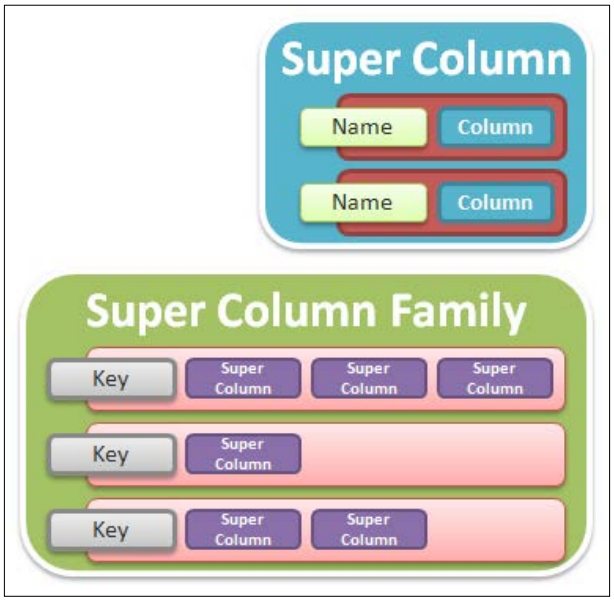
The structure of a keyspace

Keyspace is a set of column families and super column families, which will be introduced in the following section. It is analogous to a schema or database in the relational world. Each Cassandra instance has a system keyspace to keep system-wide metadata.

Keyspace contains replication settings controlling how data is distributed and replicated in the cluster. Very often, one cluster contains just one keyspace.

Super column and super column family

As shown in the following figure, a super column is a named map of columns and a super column family is just a collection of super columns:



The structure of a super column and a super column family

Super columns were popular in the earlier versions of Cassandra but are not recommended anymore since they are not supported by the Cassandra Query Language (CQL), a SQL-like language to manipulate and query Cassandra, and must be accessed by using the low-level Thrift API. A column family is enough in most cases.

Thrift

Thrift is a software framework for the development of scalable cross-language services. It combines a software stack with a code generation engine to build services that work efficiently and seamlessly with numerous programming languages. It is used as a **remote procedure call** (**RPC**) framework and was developed at Facebook Inc. It is now an open source project in the Apache Software Foundation.

There are other alternatives, for example, Protocol Buffers, Avro, MessagePack, JSON, and so on.

Collections

Cassandra allows collections, namely sets, lists, and maps, as parts of the data model. Collections are a complex type that can provide flexibility in querying.

Cassandra allows the following collections:

- **Sets**: These provide a way of keeping a unique set of values. It means that one can easily solve the problem of tracking unique values.
- **Lists**: These are suitable for maintaining the order of the values in the collection. Lists are ordered by the natural order of the type selected.
- **Maps**: These are similar to a store of key-value pairs. They are useful for storing table-like data within a single row. They can be a workaround of not having joins.

Here we only provided a brief introduction, and we will revisit the collections in subsequent chapters.

No foreign key

Foreign keys are used in a relational database to maintain referential integrity that defines the relationship between two tables. They are used to enforce relationships in a relational data model such that the data in different but related tables can be joined to answer a query. Cassandra does not have the concept of referential integrity and hence, joins are not allowed either.

No join

Foreign keys and joins are the product of normalization in a relational data model. Cassandra has neither foreign keys nor joins. Instead, it encourages and performs best when the data model is denormalized.

Indeed, denormalization is not completely disallowed in the relational world, for example, a data warehouse built on a relational database. In practice, denormalization is a solution to the problem of poor performance of highly complex relational queries involving a large number of table joins.

In Cassandra, denormalization is normal.

Foreign keys and joins can be avoided in Cassandra with proper data modeling.

No sequence

In a relational database, sequences are usually used to generate unique values for a surrogate key. Cassandra has no sequences because it is extremely difficult to implement in a peer-to-peer distributed system. There are however workarounds, which are as follows:

- Using part of the data to generate a unique key
- Using a UUID

In most cases, the best practice is to select the second workaround.

Counter

A counter column is a special column used to store a number that keeps counting values. Counting can be either increment or decrement and timestamp is not required.

The counter column should not be used to generate surrogate keys. It is just designed to hold a distributed counter appropriate for distributed counting. Also bear in mind that updating a counter is not idempotent.

Idempotent

Idempotent was originally a term in mathematics. But in computer science, idempotent is used more comprehensively to describe an operation that will produce the same results if executed once or multiple times.

Time-To-Live

Time-To-Live (**TTL**) is set on columns only. The unit is in seconds. When set on a column, it automatically counts down and will then be expired on the server side without any intervention of the client application.

Typical use cases are for the generation of security token and one-time token, automatic purging of outdated columns, and so on.

Secondary index

One important thing you need to remember is that the secondary index in Cassandra is not identical to that in a relational database. The secondary index in Cassandra can be created to query a column that is not a part of the primary key. A column family can have more than one secondary index. Behind the scenes, it is implemented as a separate hidden table which is maintained automatically by Cassandra's internal process.

The secondary index does not support collections and cannot be created on the primary key itself. The major difference between a primary key and a secondary index is that the former is a distributed index while the latter is a local index. The primary key is used to determine the node location and so, for a given row key, its node location can be found immediately. However, the secondary index is used just to index data on the local node, and it might not be possible to know immediately the locations of all matched rows without having examined all the nodes in the cluster. Hence, the performance is unpredictable.

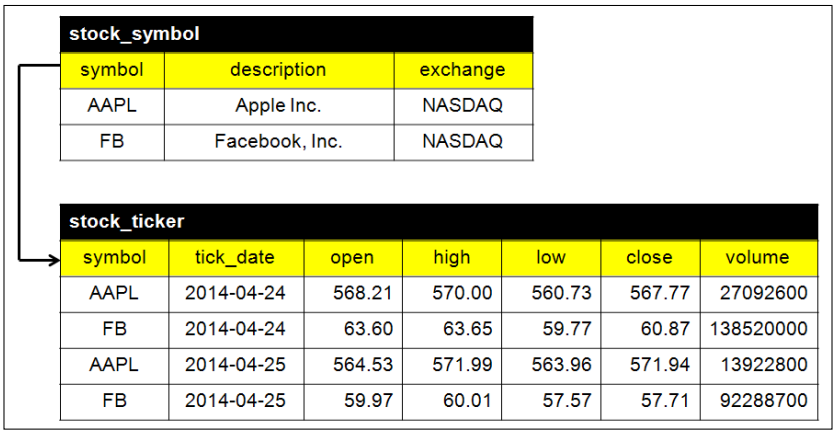
More information on secondary keys will be provided as we go through the later chapters.

Modeling by query

In the previous section, we gained a basic understanding of the differences between a relational database and Cassandra. The most important difference is that a relational database models data by relationships whereas Cassandra models data by query. Now let us start with a simple example to look into what modeling by query means.

Relational version

The following figure shows a simple relational data model of a stock quote application:



stock_symbol

symbol	description	exchange
AAPL	Apple Inc.	NASDAQ
FB	Facebook, Inc.	NASDAQ

stock_ticker

symbol	tick_date	open	high	low	close	volume
AAPL	2014-04-24	568.21	570.00	560.73	567.77	27092600
FB	2014-04-24	63.60	63.65	59.77	60.87	138520000
AAPL	2014-04-25	564.53	571.99	563.96	571.94	13922800
FB	2014-04-25	59.97	60.01	57.57	57.71	92288700

The relational data model of a stock quote application (Source: Yahoo! Finance)

The `stock_symbol` table is an entity representing the stock master information such as the symbol of a stock, the description of the stock, and the exchange that the stock is traded. The `stock_ticker` table is another entity storing the prices of `open`, `high`, `low`, `close`, and the transacted `volume` of a stock on a trading day. Obviously the two tables have a relationship based on the `symbol` column. It is a well-known one-to-many relationship.

The following is the **Data Definition Language** (**DDL**) of the two tables:

```
CREATE TABLE stock_symbol (
symbol varchar PRIMARY KEY,
description varchar,
exchange varchar
);

CREATE TABLE stock_ticker (
symbol varchar references stock_symbol(symbol),
tick_date varchar,
open decimal,
high decimal,
low decimal,
close decimal,
volume bigint,
PRIMARY KEY (symbol, tick_date)
);
```

Consider the following three cases: first, we want to list out all stocks and their description in all exchanges. The SQL query for this is very simple:

```
// Query A
SELECT symbol, description, exchange
FROM stock_symbol;
```

Second, if we want to know all the daily close prices and descriptions of the stocks listed in the `NASDAQ` exchange, we can write a SQL query as:

```
// Query B
SELECT stock_symbol.symbol, stock_symbol.description,
stock_ticker.tick_date, stock_ticker.close
FROM stock_symbol, stock_ticker
WHERE stock_symbol.symbol = stock_ticker.symbol
AND stock_symbol.exchange = ''NASDAQ'';
```

Furthermore, if we want to know all the day close prices and descriptions of the stocks listed in the `NASDAQ` exchange on April 24, 2014, we can use the following SQL query:

```
// Query C
SELECT stock_symbol.symbol, stock_symbol.description,
stock_ticker.tick_date, stock_ticker.open,
stock_ticker.high, stock_ticker.low, stock_ticker_close,
stock_ticker.volume
FROM stock_symbol, stock_ticker
WHERE stock_symbol.symbol = stock_ticker.symbol
AND stock_symbol.exchange = ''NASDAQ''
AND stock_ticker.tick_date = ''2014-04-24'';
```

By virtue of the relational data model, we can simply write different SQL queries to return different results with no changes to the underlying data model at all.

Cassandra version

Now let us turn to Cassandra. The DDL statements in the last section can be slightly modified to create column families, or tables, in Cassandra, which are as follows:

```
CREATE TABLE stock_symbol (
symbol varchar PRIMARY KEY,
description varchar,
exchange varchar
);

CREATE TABLE stock_ticker (
symbol varchar,
```

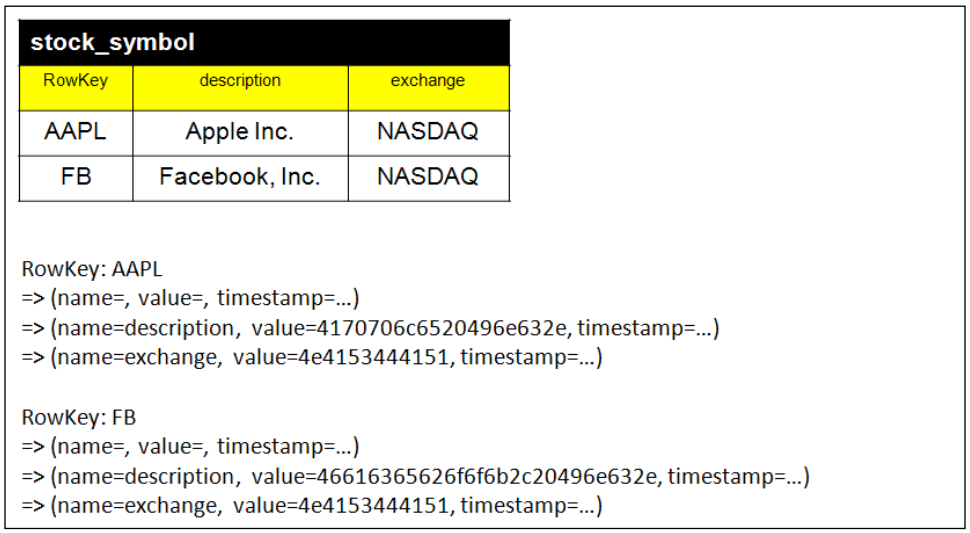
```
tick_date varchar,
open decimal,
high decimal,
low decimal,
close decimal,
volume bigint,
PRIMARY KEY (symbol, tick_date)
);
```

They seem to be correct at first sight.

As for `Query A`, we can query the Cassandra `stock_symbol` table exactly the same way:

```
// Query A
SELECT symbol, description, exchange
FROM stock_symbol;
```

The following figure depicts the logical and physical storage views of the `stock_symbol` table:



The Cassandra data model for Query A

The primary key of the `stock_symbol` table involves only one single column, `symbol`, which is also used as the row key and partition key of the column family. We can consider the `stock_symbol` table in terms of the SortedMap data structure mentioned in the previous section:

```
Map<RowKey, SortedMap<ColumnKey, ColumnValue>>
```

The assigned values are as follows:

```
RowKey=AAPL
ColumnKey=description
ColumnValue=Apple Inc.
ColumnKey=exchange
ColumnValue=NASDAQ
```

So far so good, right?

However, without foreign keys and joins, how can we obtain the same results for `Query B` and `Query C` in Cassandra? It indeed highlights that we need another way to do so. The short answer is to use denormalization.

For `Query B`, what we want is all the day close prices and descriptions of the stocks listed in the `NASDAQ` exchange. The columns involved are `symbol`, `description`, `tick_date`, `close`, and `exchange`. The first four columns are obvious, but why do we need the `exchange` column? The `exchange` column is necessary because it is used as a filter for the query. Another implication is that the `exchange` column is required to be the row key, or at least part of the row key.

Remember two rules:

1. A row key is regarded as a partition key to locate the nodes storing that row
2. A row cannot be split across two nodes

In a distributed system backed by Cassandra, we should minimize unnecessary network traffic as much as possible. In other words, the lesser the number of nodes the query needs to work with, the better the performance of the data model. We must cater to the cluster topology as well as the physical storage of the data model.

Therefore we should create a column family for `Query B` similar to the previous one:

```
// Query B
CREATE TABLE stock_ticker_by_exchange (
exchange varchar,
symbol varchar,
description varchar,
tick_date varchar,
close decimal,
PRIMARY KEY (exchange, symbol, tick_date)
);
```

The logical and physical storage views of `stock_ticker_by_exchange` are shown as follows:

stock_ticker_by_exchange

RowKey	AAPL:2014-04-24:	AAPL:2014-04-24:close	AAPL:2014-04-24:description	AAPL:2014-04-25:	AAPL:2014-04-25:close	AAPL:2014-04-25:description
NASDAQ	0	568.21	Apple Inc.	0	571.94	Apple Inc.
	FB:2014-04-24:	FB:2014-04-24:close	FB:2014-04-24:description	FB:2014-04-25:	FB:2014-04-25:close	FB:2014-04-25:description
	0	60.87	Facebook, Inc.	0	57.71	Facebook, Inc.

RowKey: NASDAQ
=> (name=AAPL:2014-04-24:, value=, timestamp=...)
=> (name=AAPL:2014-04-24:close, value=0000000200ddc9, timestamp=...)
=> (name=AAPL:2014-04-24:description, value=4170706c6520496e632e, timestamp=...)
=> (name=AAPL:2014-04-25:, value=, timestamp=...)
=> (name=AAPL:2014-04-25:close, value=0000000200df6a, timestamp=...)
=> (name=AAPL:2014-04-25:description, value=4170706c6520496e632e, timestamp=...)
=> (name=FB:2014-04-24:, value=, timestamp=...)
=> (name=FB:2014-04-24:close, value=0000000217c7, timestamp=...)
=> (name=FB:2014-04-24:description, value=46616365626f6f6b2c20496e632e, timestamp=...)
=> (name=FB:2014-04-25:, value=, timestamp=...)
=> (name=FB:2014-04-25:close, value=00000002168b, timestamp=...)
=> (name=FB:2014-04-25:description, value=46616365626f6f6b2c20496e632e, timestamp=...)

The Cassandra data model for Query B

The row key is the `exchange` column. However, this time, it is very strange that the column keys are no longer `symbol`, `tick_date`, `close`, and `description`. There are now 12 columns including `APPL:2014-04-24:`, `APPL:2014-04-24:close`, `APPL:2014-04-24:description`, `APPL:2014-04-25:`, `APPL:2014-04-25:close`, `APPL:2014-04-25:description`, `FB:2014-04-24:`, `FB:2014-04-24:close`, `FB:2014-04-24:description`, `FB:2014-04-25:`, `FB:2014-04-25:close`, and `FB:2014-04-25:description`, respectively. Most importantly, the column keys are now dynamic and are able to store data in just a single row. The row of this dynamic usage is called a wide row, in contrast to the row containing static columns of the `stock_symbol` table—termed as a skinny row.

Whether a column family stores a skinny row or a wide row depends on how the primary key is defined.

If the primary key contains only one column, the row is a skinny row.
If the primary key contains more than one column, it is called a compound primary key and the row is a wide row.

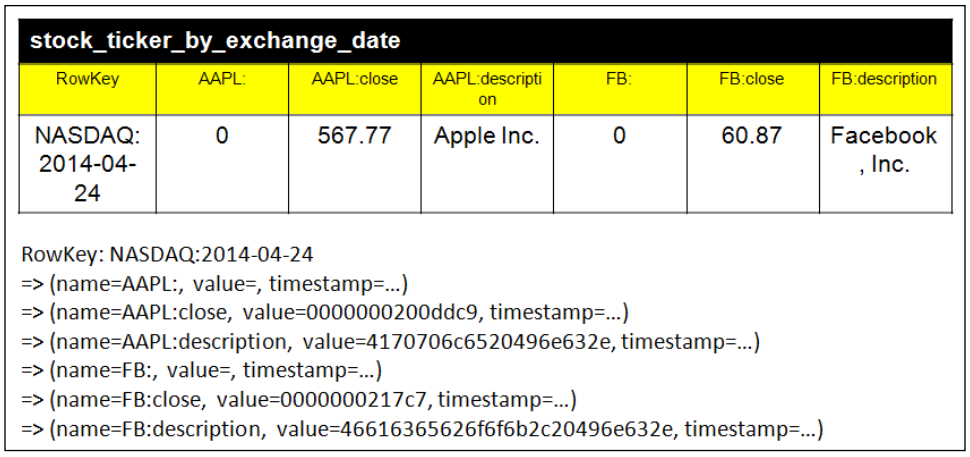
In either case, the first column in the primary key definition is the row key.

Finally, we come to `Query C`. Similarly, we make use of denormalization. `Query C` differs from `Query B` by an additional date filter on April 24, 2014. You might think of reusing the `stock_ticker_by_exchange` table for `Query C`. The answer is wrong. Why? The clue is the primary key which is composed of three columns, `exchange`, `symbol`, and `tick_date`, respectively. If you look carefully at the column keys of the `stock_ticker_by_exchange` table, you find that the column keys are dynamic as a result of the `symbol` and `tick_date` columns. Hence, is it possible for Cassandra to determine the column keys without knowing exactly which symbols you want? Negative.

A suitable column family for `Query C` should resemble the following code:

```
// Query C
CREATE TABLE stock_ticker_by_exchange_date (
exchange varchar,
symbol varchar,
description varchar,
tick_date varchar,
close decimal,
PRIMARY KEY ((exchange, tick_date), symbol)
);
```

This time you should be aware of the definition of the primary key. It is interesting that there is an additional pair of parentheses for the `exchange` and `tick_date` columns. Let's look at the logical and physical storage views of `stock_ticker_by_exchange_date`, as shown in the following figure:



stock_ticker_by_exchange_date

RowKey	AAPL:	AAPL:close	AAPL:description	FB:	FB:close	FB:description
NASDAQ:2014-04-24	0	567.77	Apple Inc.	0	60.87	Facebook, Inc.

RowKey: NASDAQ:2014-04-24
=> (name=AAPL:, value=, timestamp=...)
=> (name=AAPL:close, value=0000000200ddc9, timestamp=...)
=> (name=AAPL:description, value=4170706c6520496e632e, timestamp=...)
=> (name=FB:, value=, timestamp=...)
=> (name=FB:close, value=0000000217c7, timestamp=...)
=> (name=FB:description, value=46616365626f6f6b2c20496e632e, timestamp=...)

The Cassandra data model for Query C

You should pay attention to the number of column keys here. It is only six instead of 12 as in `stock_ticker_by_exchange` for `Query B`. The column keys are still dynamic according to the `symbol` column but the row key is now `NASDAQ:2014-04-24` instead of just `NASDAQ` in `Query B`. Do you remember the previously mentioned additional pair of parentheses? If you define a primary key in that way, you intend to use more than one column to be the row key and the partition key. It is called a composite partition key. For the time being, it is enough for you to know the terminology only. Further information will be given in later chapters.

Until now, you might have felt dizzy and uncomfortable, especially for those of you having so many years of expertise in the relational data model. I also found the Cassandra data model very difficult to comprehend at the first time. However, you should be aware of the subtle differences between a relational data model and Cassandra data model. You must also be very cautious of the query that you handle. A query is always the starting point of designing a Cassandra data model. As an analogy, a query is a question and the data model is the answer. You merely use the data model to answer the query. It is exactly what modeling by query means.

Data modeling considerations

Apart from modeling by query, we need to bear in mind a few important points when designing a Cassandra data model. We can also consider a few good patterns that will be introduced in this section.

Data duplication

Denormalization is an evil in a relational data model, but not in Cassandra. Indeed, it is a good and common practice. It is solely based on the fact that Cassandra does not use high-end disk storage subsystem. Cassandra loves commodity-grade hard drives, and hence disk space is cheap. Data duplication as a result of denormalization is by no means a problem anymore; Cassandra welcomes it.

Sorting

In a relational database, sorting can be easily controlled using the `ORDER BY` clause in a SQL query. Alternatively, a secondary index can be created to further speed up the sorting operations.

In Cassandra, however, sorting is by design because you must determine how to compare data for a column family at the time of its creation. The comparator of the column family dictates how the rows are ordered on reads. Additionally, columns are ordered by their column names, also by a comparator.

Wide row

It is common to use wide rows for ordering, grouping and efficient filtering. Besides, you can use skinny rows. All you have to consider is the number of columns the row contains.

It is worth noting that for a column family storing skinny rows, the column key is repeatedly stored in each column. Although it wastes some storage space, it is not a problem on inexpensive commodity hard disks.

Bucketing

Even though a wide row can accommodate up to 2 billion variable columns, it is still a hard limit that cannot prevent voluminous data from filling up a node. In order to break through the 2 billion column limit, we can use a workaround technique called bucketing to split the data across multiple nodes.

Bucketing requires the client application to generate a bucket ID, which is often a random number. By including the bucket ID into a composite partition key, you can break up and distribute segments of the data to different nodes. However, it should not be abused. Breaking up the data across multiple nodes causes reading operations to consume extra resources to merge and reorder data. Thus, it is expensive and not a favorable method, and therefore should only be a last resort.

Valueless column

Column keys can store values as shown in the *Modeling by query* section. There is no "Not Null" concept in Cassandra such that column values can store empty values without any problem. Simply storing data in column keys while leaving empty values in the column, known as a valueless column, is sometimes used purposely. It's a common practice with Cassandra.

One motivation for valueless columns is the sort-by-column-key feature of Cassandra. Nonetheless, there are some limitations and caveats. The maximum size of a column key is 64 KB, in contrast to 2 GB for a column value. Therefore, space in a column key is limited. Furthermore, using timestamp alone as a column key can result in timestamp collision.

Time-series data

What is time-series data? It is anything that varies on a temporal basis such as processor utilization, sensor data, clickstream, and stock ticker. The stock quote data model introduced earlier is one such example. Cassandra is a perfect fit for storing time-series data. Why? Because one row can hold as many as 2 billion variable columns. It is a single layout on disk, based on the storage model. Therefore, Cassandra can handle voluminous time-series data in a blazing fast fashion. TTL is another excellent feature to simplify data housekeeping.

In the second half of this book, a complete stock quote technical analysis application will be developed to further explain the details of using Cassandra to handle time-series data.

Cassandra Query Language

It is quite common for other authors to start introducing the Cassandra data model from CQL. I use a different approach in this chapter. I try to avoid diving too deep in CQL before we have a firm understanding of how Cassandra handles its physical storage.

The syntax of CQL is designed to be very similar to that of SQL. This intent is good for someone who is used to writing SQL statements in the relational world, to migrate to Cassandra. However, because of the high degree of similarity between CQL and SQL, it is even more difficult for us to throw away the relational mindsets if CQL is used to explain how to model data in Cassandra. It might cause more confusion in the end. I prefer the approach of a microscopic view of how the data model relates to the physical storage. By doing so, you can grasp the key points more quickly and understand the inner working mechanism more clearly. CQL is covered extensively in the next chapter.

Summary

In this chapter, we looked at the basics of a Cassandra data model and are now familiar with the column, row, column family, keyspace, counter, and other related terms. A comparison of the main differences between a relational data model and the Cassandra data model was also given to explain the concept of modeling by query that may seem shocking and counterintuitive at first sight. Then a few important considerations on data modeling and typical usage patterns were introduced. Finally, the reason why the introduction of CQL is deliberately postponed was expressed.

This chapter is only the first part on Cassandra data modeling. In the next chapter, we will continue the second part of the tour, Cassandra Query Language.

3
CQL Data Types

In this chapter, we will have an overview of Cassandra Query Language and take a detailed look into the wealthy set of data types supported by Cassandra. We will walk through the data types to study what their internal storage structure looks like. If you want to know how Cassandra implements them behind the scenes, the Java source code of Cassandra can be referenced. For those of you who have not installed and set up Cassandra, you can refer to *Chapter 5*, *First-cut Design and Implementation*, for a quick procedure.

Introduction to CQL

Cassandra introduced Cassandra Query Language (CQL) in release 0.8 as a SQL-like alternative to the traditional Thrift RPC API. As of the time of this writing, the latest CQL version is 3.1.7. I do not want to take you through all of its old versions and therefore, I will focus on version 3.1.7 only. It should be noted that CQL Version 3 is not backward compatible with CQL Version 2 and differs from it in many ways.

CQL statements

CQL Version 3 provides a model very similar to SQL. Conceptually, it uses a table to store data in rows of columns. It is composed of three main types of statements:

- **Data definition statements**: These are used to set and change how data is stored in Cassandra
- **Data manipulation statements**: These are used to create, delete, and modify data
- **Query statements**: These are used to look up data

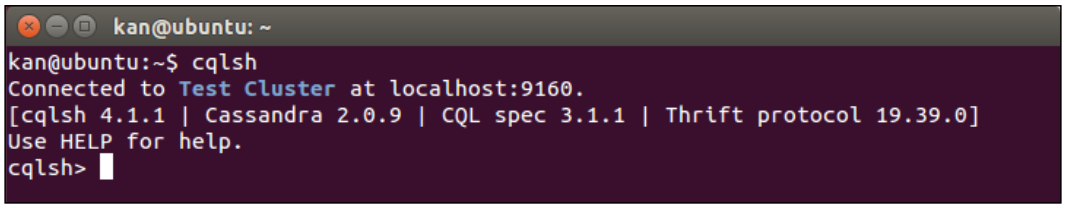
CQL is case insensitive, unless the word is enclosed in double quotation marks. It defines a list of keywords that have a fixed meaning for the language. It distinguishes between reserved and non-reserved keywords. **Reserved** keywords cannot be used as identifiers. They are truly reserved for the language. **Non-reserved** keywords only have a specific meaning in certain contexts but can be used as identifiers. The list of CQL keywords is shown in DataStax's documentation at `http://www.datastax.com/documentation/cql/3.1/cql/cql_reference/keywords_r.html`.

CQL command-line client – cqlsh

Cassandra bundles an interactive terminal supporting CQL, known as `cqlsh`. It is a Python-based command-line client used to run CQL commands. To start `cqlsh`, navigate to Cassandra's `bin` directory and type the following:

- On Linux, type `./cqlsh`
- On Windows, type `cqlsh.bat` or `python cqlsh`

As shown in the following figure, `cqlsh` shows the cluster name, Cassandra, CQL, and Thrift protocol versions on startup:

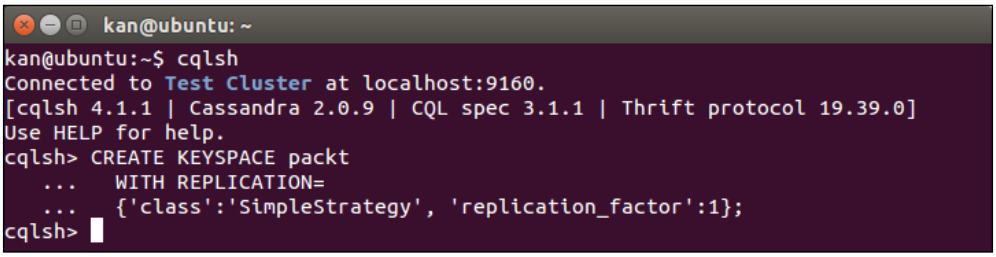


cqlsh connected to the Cassandra instance running on the local node

We can use `cqlsh` to connect to other nodes by appending the host (either hostname or IP address) and port as command-line parameters.

If we want to create a keyspace called `packt` using `SimpleStrategy` (which will be explained in *Chapter 6*, *Enhancing a Version*) as its replication strategy and setting the replication factor as one for a single-node Cassandra cluster, we can type the CQL statement, shown in the following screenshot, in `cqlsh`.

This utility will be used extensively in this book to demonstrate how to use CQL to define the Cassandra data model:



Create keyspace packt in cqlsh

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. 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.

Native data types

CQL Version 3 supports many basic data types for columns. It also supports collection types and all data types available to Cassandra. The following table lists the supported basic data types and their corresponding meanings:

Type	Description
ascii	ASCII character string
bigint	64-bit signed long
blob	Arbitrary bytes (no validation)
Boolean	`True` or `False`
counter	Counter column (64-bit signed value)
decimal	Variable-precision decimal
double	64-bit IEEE 754 floating point
float	32-bit IEEE 754 floating point
inet	An IP address that can be either 4 bytes long (IPv4) or 16 bytes long (IPv6) and should be inputted as a string
int	32-bit signed integer
text	UTF8 encoded string

Type	Description
timestamp	A timestamp in which string constants are allowed to input timestamps as dates
timeuuid	Type 1 UUID that is generally used as a "conflict-free" timestamp
uuid	Type 1 or type 4 UUID
varchar	UTF8-encoded string
varint	Arbitrary-precision integer

Table 1. CQL Version 3 basic data types

Cassandra implementation

If we look into the Cassandra's Java source code, the CQL Version 3 native data types are declared in an `enum` called `Native` in the `org.apache.cassandra.cql3.CQL3Type` interface, as shown in the following screenshot:

Cassandra source code declaring CQL Version 3 native data types

It is interesting to know that `TEXT` and `VARCHAR` are indeed both `UTF8Type`. The Java classes of `AsciiType`, `LongType`, `BytesType`, `DecimalType`, and so on are declared in the `org.apache.cassandra.db.marshal` package.

Cassandra source code is available on GitHub at `https://github.com/apache/cassandra`.

Knowing the Java implementation of the native data types allows us to have a deeper understanding of how Cassandra handles them. For example, Cassandra uses the `org.apache.cassandra.serializers.InetAddressSerializer` class and `java.net.InetAddress` class to handle the serialization/deserialization of the `INET` data type.

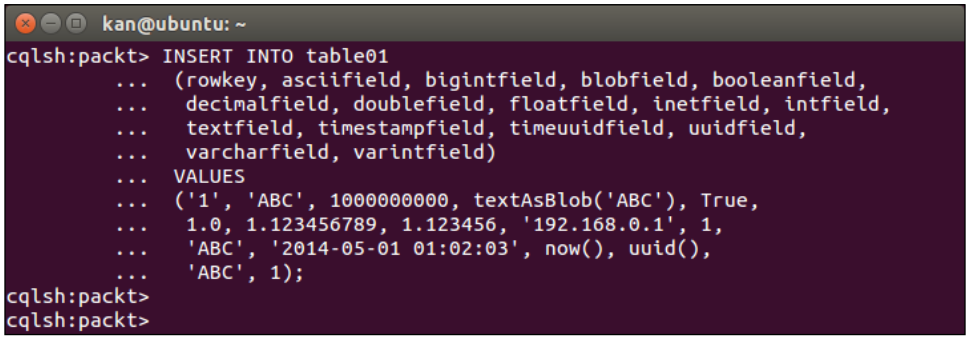
A not-so-long example

These native data types are used in CQL statements to specify the type of data to be stored in a column of a table. Now let us create an experimental table with columns of each native data type (except counter type since it requires a separate table), and then insert some data into it. We need to specify the keyspace, `packt` in this example, before creating the table called `table01`, as shown in the following screenshot:

```
kan@ubuntu: ~
kan@ubuntu:~$ cqlsh
Connected to Test Cluster at localhost:9160.
[cqlsh 4.1.1 | Cassandra 2.0.9 | CQL spec 3.1.1 | Thrift protocol 19.39.0]
Use HELP for help.
cqlsh> USE packt;
cqlsh:packt>
cqlsh:packt> CREATE TABLE table01 (
         ...  rowkey ascii,
         ...  asciifield ascii,
         ...  bigintfield bigint,
         ...  blobfield blob,
         ...  booleanfield boolean,
         ...  decimalfield decimal,
         ...  doublefield double,
         ...  floatfield float,
         ...  inetfield inet,
         ...  intfield int,
         ...  textfield text,
         ...  timestampfield timestamp,
         ...  timeuuidfield timeuuid,
         ...  uuidfield uuid,
         ...  varcharfield varchar,
         ...  varintfield varint,
         ...  PRIMARY KEY (rowkey)
         ... );
cqlsh:packt>
```

Create table01 to illustrate each native data type

We create the table using the default values, but, there are other options to configure the new table for optimizations, including compaction, compression, failure handling, and so on. The `PRIMARY KEY` clause, which is on only one column, could also be specified along with an attribute, that is, `rowkey ascii PRIMARY KEY`. Then insert a sample record into `table01`. We make it with an `INSERT` statement, as shown in the following screenshot:



Insert a sample record into table01

We now have data inside `table01`. We use `cqlsh` to query the table. For the sake of comparison, we also use another Cassandra command-line tool called Cassandra CLI to have a low-level view of the row. Let us open Cassandra CLI on a terminal.

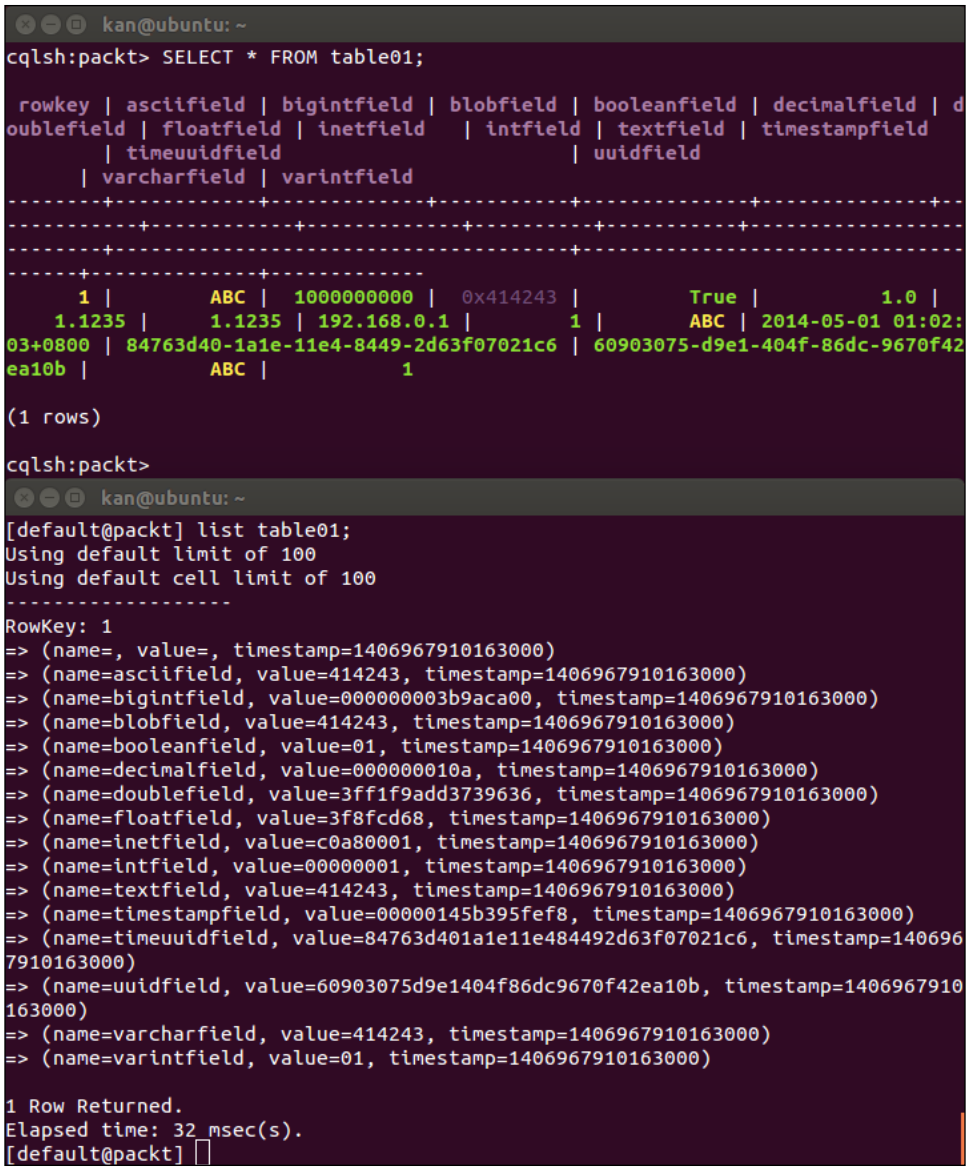
Cassandra CLI utility

Cassandra CLI is used to set storage configuration attributes on a per-keyspace or per-table basis. To start it up, you navigate to Cassandra bin directory and type the following:

- On Linux, `./cassandra-cli`
- On Windows, `cassandra.bat`

Note that it was announced to be deprecated in Cassandra 3.0 and `cqlsh` should be used instead.

The results of the `SELECT` statement in `cqlsh` and the `list` command in Cassandra CLI are shown in the following screenshot. We will then walk through each column one by one:



Comparison of the sample row in cqlsh and Cassandra CLI

ASCII

Internally, a data value `'ABC'` is stored as the byte values in hexadecimal representation of each individual character, `'A'`, `'B'`, and `'C'` as `0x41`, `0x42`, and `0x43` respectively.

Bigint

This one is simple; the hexadecimal representation of the number `1000000000` is `0x000000003b9aca00` of 64-bit length stored internally.

BLOB

A `BLOB` data type is used to store a large binary object. In our previous example, we inserted a text `'ABC'` as a `BLOB` into the `blobfield`. The internal representation is `414243`, which is just a stream of bytes in hexadecimal representation.

Obviously, a `BLOB` field can accept all kinds of data, and because of this flexibility it cannot have validation on its data value. For example, a data value `2` may be interpreted as either an integer `2` or a text `'2'`. Without knowing the interpretation we want, a `BLOB` field can impose a check on the data value.

Another interesting point of a `BLOB` field is that, as shown in the `SELECT` statement in the previous screenshot in `cqlsh`, the data value of `blobfield` returned is `0x414243` for `'ABC'` text. We know from the previous section that `0x41`, `0x42`, `0x43` are the byte values of `'A'`, `'B'`, and `'C'`, respectively. However, for a `BLOB` field, `cqlsh` prefixes its data value with `'0x'` to make it a so-called BLOB constant. A BLOB constant is a sequence of bytes in their hexadecimal values prefixed by `0[xX](hex)+` where `hex` is a hexadecimal character, such as `[0-9a-fA-F]`.

CQL also provides a number of `BLOB` conversion functions to convert native data types into a `BLOB` and vice versa. For every `<native-type>` (except `BLOB` for an obvious reason) supported by CQL, the `<native-type>AsBlob` function takes an argument of type `<native-type>` and returns it as a `BLOB`. Contrarily, the `blobAs<Native-type>` function reverses the conversion from a `BLOB` back to a `<native-type>`. As demonstrated in the `INSERT` statement, we have used `textAsBlob()` to convert a `text` data type into a `BLOB`.

BLOB constant

BLOB constants were introduced in CQL version 3.0.2 to allow users to input `BLOB` values. In older versions of CQL, inputting `BLOB` as string was supported for convenience. It is now deprecated and will be removed in a future version. It is still supported only to allow smoother transition to a `BLOB` constant. Updating the client code to switch to `BLOB` constants should be done as soon as possible.

Boolean

A `boolean` data type is also very intuitive. It is merely a single byte of either `0x00`, which means `False`, or `0x01`, which means `True`, in the internal storage.

Decimal

A `decimal` data type can store a variable-precision decimal, basically a `BigDecimal` data type in Java.

Double

The `double` data type is a double-precision 64-bit IEEE 754 floating point in its internal storage.

Float

The `float` data type is a single-precision 32-bit IEEE 754 floating point in its internal storage.

BigDecimal, double, or float?

The difference between `double` and `float` is obviously the length of precision in the floating point value. Both `double` and `float` use binary representation of decimal numbers with a radix which is in many cases an approximation, not an absolute value. `double` is a 64-bit value while `float` is an even shorter 32-bit value. Therefore, we can say that double is more precise than float. However, in both cases, there is still a possibility of loss of precision which can be very noticeable when working with either very big numbers or very small numbers.

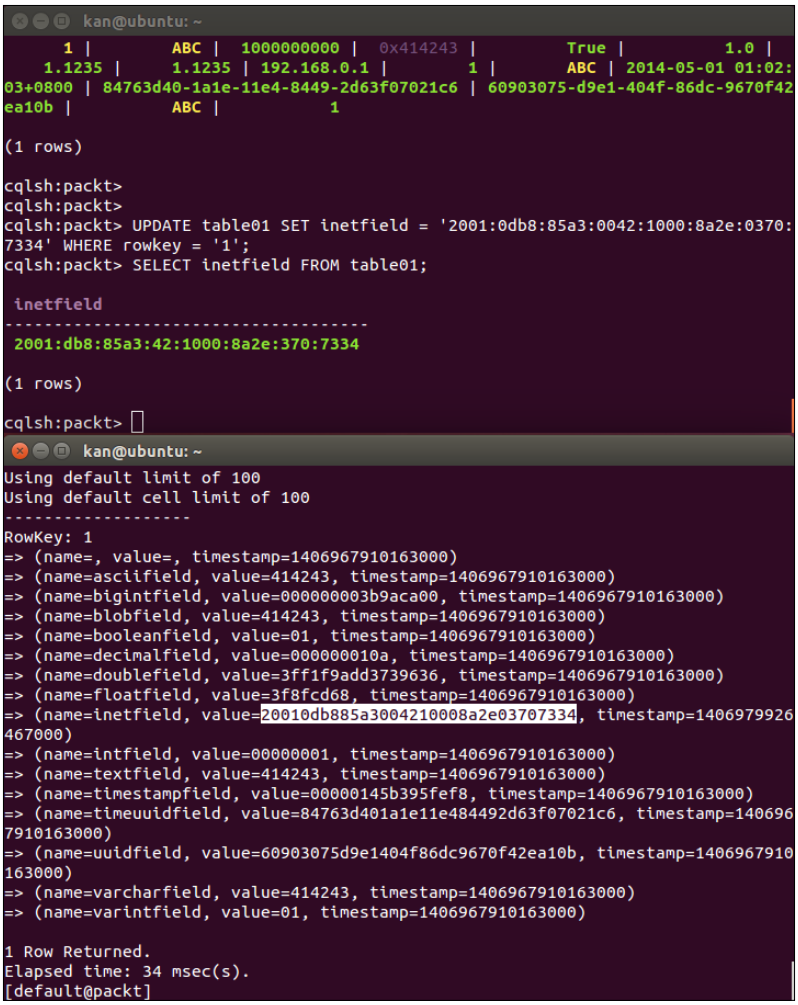
On the contrary, `BigDecimal` is devised to overcome this loss of precision discrepancy. It is an exact way of representing numbers. Its disadvantage is slower runtime performance.

Whenever you are dealing with money or precision is a must, `BigDecimal` is the best choice (or `decimal` in CQL native data types), otherwise `double` or `float` should be good enough.

Inet

The `inet` data type is designed for storing IP address values in **IP Version 4** (**IPv4**) and **IP Version 6** (**IPv6**) format. The IP address, `192.168.0.1`, in the example record is stored as four bytes internally; `192` is stored as `0xc0`, `168` as `0xa8`, `0` as `0x00`, and `1` as `0x01`, respectively. It should be noted that regardless of the IP address being stored is IPv4 or IPv6, the port number is *not* stored. We need another column to store it if required.

We can also store an IPv6 address value. The following `UPDATE` statement changes the `inetfield` to an IPv6 address `2001:0db8:85a3:0042:1000:8a2e:0370:7334`, as shown in the following screenshot:



Comparison of the sample row in cqlsh and Cassandra CLI in inetfield

Internet Protocol Version 6

Internet Protocol Version 6 (IPv6) is the latest version of the **Internet Protocol** (**IP**). It was developed by the IETF to deal with the long-anticipated problem of IPv4 address exhaustion.

IPv6 uses a 128-bit address whereas IPv4 uses 32-bit address. The two protocols are not designed to be interoperable, making the transition to IPv6 complicated.

IPv6 addresses are usually represented as eight groups of four hexadecimal digits separated by colons, such as `2001:0db8:85a3:0042:1000:8a2e:0370:7334`.

In `cqlsh`, the leading zeros of each group of four hexadecimal digits are removed. In Cassandra's internal storage, the IPv6 address value consumes 16 bytes.

Int

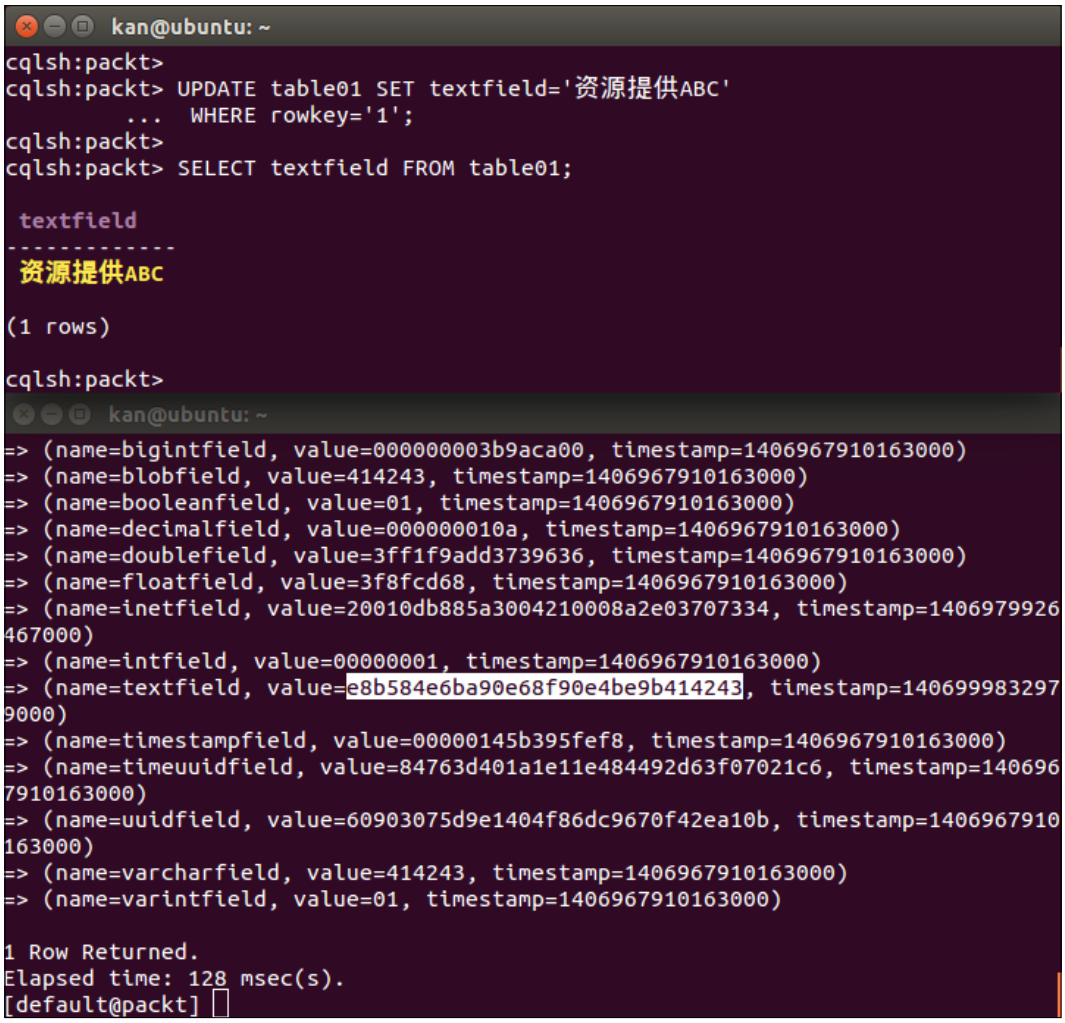
The `int` data type is a primitive 32-bit signed integer.

Text

The `text` data type is a UTF-8 encoded string accepting Unicode characters. As shown previously, the byte values of "`ABC`", `0x41`, `0x42`, and `0x43`, are stored internally. We can test the `text` field with non-ASCII characters by updating the `textfield` as shown in the following screenshot:

The `text` data type is a combination of non-ASCII and ASCII characters. The four non-ASCII characters are represented as their 3-byte UTF-8 values, `0xe8b584`, `0xe6ba90`, `0xe68f90`, and `0xe4be9b`.

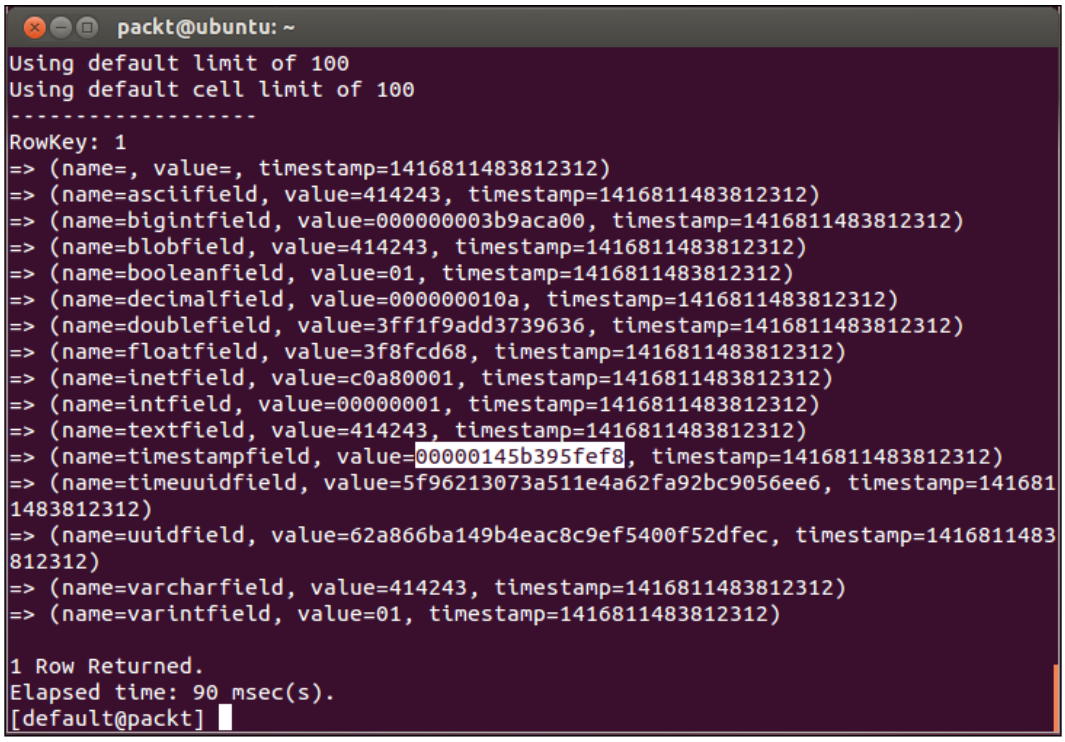
However, the ASCII characters are still stored as byte values, as shown in the screenshot:



Experiment of the textfield data type

Timestamp

The value of the `timestampfield` is encoded as a 64-bit signed integer representing a number of milliseconds since the standard base time known as the *epoch*: January 1, 1970, at 00:00:00 GMT. A `timestamp` data type can be entered as an integer for CQL input, or as a string literal in ISO 8601 formats. As shown in the following screenshot, the internal value of May 1, 2014, 16:02:03, in the +08:00 timezone is `0x00000145b6cdf878` or 1,398,931,323,000 milliseconds since the epoch:



Experiment of the timestamp data type

A `timestamp` data type contains a date portion and a time portion in which the time of the day can be omitted if only the value of the date is wanted. Cassandra will use 00:00:00 as the default for the omitted time of day.

ISO 8601

ISO 8601 is the international standard for representation of dates and times. Its full reference number is ISO 8601:1988 (E), and its title is "Data elements and interchange formats – Information interchange – Representation of dates and times."

ISO 8601 describes a large number of date/time formats depending on the desired level of granularity. The formats are as follows. Note that the "T" appears literally in the string to indicate the beginning of the time element.

- Year: YYYY (e.g. 1997)
- Year and month: YYYY-MM (e.g. 1997-07)
- Date: YYYY-MM-DD (e.g. 1997-07-16)
- Date plus hours and minutes: YYYY-MM-DDThh:mmTZD (e.g. 1997-07-16T19:20+01:00)
- Date plus hours, minutes and seconds: YYYY-MM-DDThh:mm:ssTZD (e.g. 1997-07-16T19:20:30+01:00)

- Date plus hours, minutes, seconds and a decimal fraction of a second: YYYY-MM-DDThh:mm:ss.sTZD (e.g. 1997-07-16T19:20:30.45+01:00)

Where:

- YYYY = four-digit year
- MM = two-digit month (01=January, etc.)
- DD = two-digit day of month (01 through 31)
- hh = two digits of hour (00 through 23) (am/pm NOT allowed)
- mm = two digits of minute (00 through 59)
- ss = two digits of second (00 through 59)
- s = one or more digits representing a decimal fraction of a second
- TZD = time zone designator (Z or +hh:mm or -hh:mm)

Times are expressed either in **Coordinated Universal Time** (**UTC**) with a special UTC designator "Z" or in local time together with a time zone offset in hours and minutes. A time zone offset of "+/-hh:mm" indicates the use of a local time zone which is "hh" hours and "mm" minutes ahead/behind of UTC.

If no time zone is specified, the time zone of the Cassandra coordinator node handing the write request is used. Therefore the best practice is to specify the time zone with the timestamp rather than relying on the time zone configured on the Cassandra nodes to avoid any ambiguities.

Timeuuid

A value of the `timeuuid` data type is a Type 1 UUID which includes the time of its generation and is sorted by timestamp. It is therefore ideal for use in applications requiring conflict-free timestamps. A valid `timeuuid` uses the time in 100 intervals since 00:00:00.00 UTC (60 bits), a clock sequence number for prevention of duplicates (14 bits), and the IEEE 801 MAC address (48 bits) to generate a unique identifier, for example, `74754ac0-e13f-11e3-a8a3-a92bc9056ee6`.

CQL v3 offers a number of functions to make the manipulation of `timeuuid` handy:

- **dateOf()**: This is used in a `SELECT` statement to extract the timestamp portion of a `timeuuid` column
- **now()**: This is used to generate a new unique `timeuuid`
- **minTimeuuid() and maxTimeuuid()**: These are used to return a result similar to a UUID given a conditional time component as its argument
- **unixTimestampOf()**: This is used in a `SELECT` statement to extract the timestamp portion as a raw 64-bit integer timestamp of a `timeuuid` column

The following figure uses `timeuuidfield` of `table01` to demonstrate the usage of these `timeuuid` functions:

```
kan@ubuntu: ~
cqlsh:packt>
cqlsh:packt> SELECT now(), timeuuidfield, dateOf(timeuuidfield),
         ...  unixTimestampOf(timeuuidfield),
         ...  minTimeuuid(dateOf(timeuuidfield)),
         ...  maxTimeuuid(dateOf(timeuuidfield))
         ...  FROM table01;

 now()                                | timeuuidfield                        | d
ateOf(timeuuidfield)     | unixTimestampOf(timeuuidfield) | minTimeuuid(dateOf(ti
meuuidfield))    | maxTimeuuid(dateOf(timeuuidfield))
--------------------------------------+--------------------------------------+--
----------------------+--------------------------------+----------------------
---------------+--------------------------------------
 f538c370-1a3e-11e4-8449-2d63f07021c6 | 84763d40-1a1e-11e4-8449-2d63f07021c6 | 2
014-08-02 16:25:10+0800 |                  1406967910164 | 84763d40-1a1e-11e4-80
80-808080808080 | 8476644f-1a1e-11e4-7f7f-7f7f7f7f7f7f

(1 rows)

cqlsh:packt>
```

Demonstration of timeuuid functions

Timestamp or Timeuuid?

Timestamp is suitable for storing date and time values. TimeUUID, however, is more suitable in those cases where a conflict free, unique timestamp is needed.

UUID

The `UUID` data type is usually used to avoid collisions in values. It is a 16-byte value that accepts a type 1 or type 4 UUID. CQL v3.1.6 or later versions provide a function called `uuid()` to easily generate random type 4 UUID values.

Type 1 or type 4 UUID?

Type 1 uses the MAC address of the computer that is generating the `UUID` data type and the number of 100-nanosecond intervals since the adoption of the Gregorian calendar, to generate UUIDs. Its uniqueness across computers is guaranteed if MAC addresses are not duplicated; however, given the speed of modern processors, successive invocations on the same machine of a naive implementation of a type 1 generator might produce the same `UUID`, negating the property of uniqueness.

Type 4 uses random or pseudorandom numbers. Therefore, it is the recommended type of `UUID` to be used.

Varchar

Basically `varchar` is identical to `text` as evident by the same `UTF8Type` in the source code.

Varint

A `varint` data type is used to store integers of arbitrary precision.

Counter

A `counter` data type is a special kind of column whose user-visible value is a 64-bit signed integer (though this is more complex internally) used to store a number that incrementally counts the occurrences of a particular event. When a new value is written to a given counter column, it is added to the previous value of the counter.

A counter is ideal for counting things quickly in a distributed environment which makes it invaluable for real time analytical tasks. The `counter` data type was introduced in Cassandra Version 0.8. Counter column tables must use `counter` data type. Counters can be stored in dedicated tables only, and you cannot create an index on a counter column.

Counter type don'ts

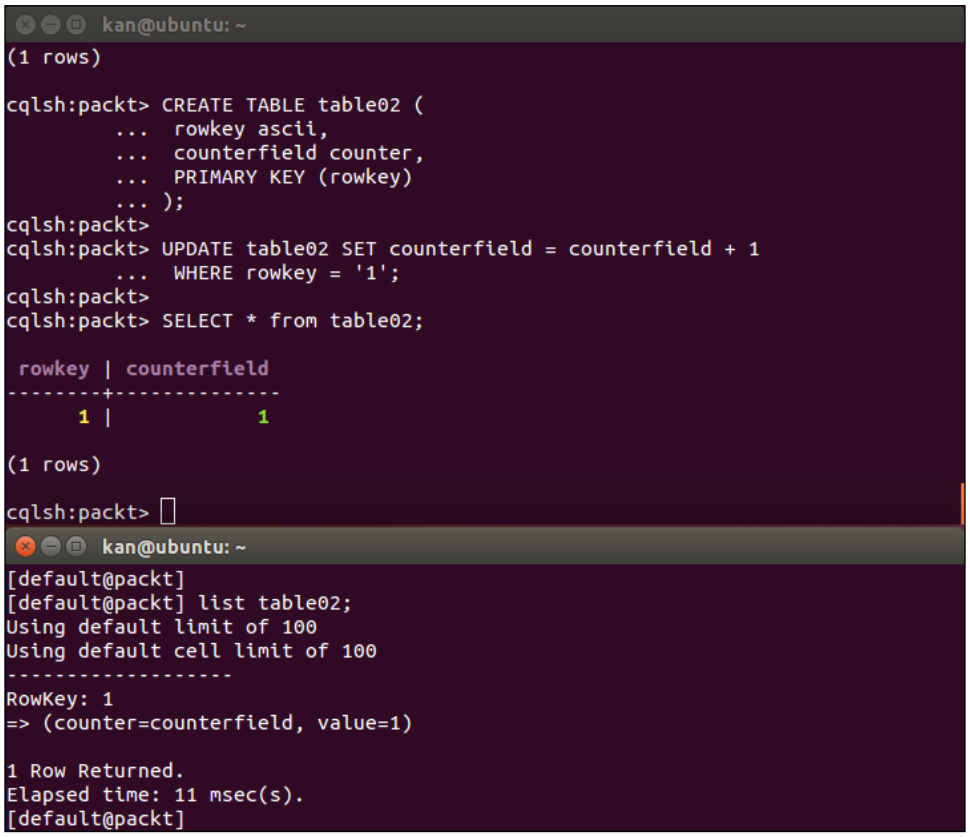
- Don't assign the `counter` data type to a column that serves as the primary key
- Don't use the `counter` data type in a table that contains anything other than `counter` data types and primary keys
- Don't use the `counter` data type to generate sequential numbers for surrogate keys; use the `timeuuid` data type instead

We use a `CREATE TABLE` statement to create a counter table. However, `INSERT` statements are not allowed on counter tables and so we must use an `UPDATE` statement to update the counter column as shown in the following screenshot.

Cassandra uses `counter` instead of `name` to indicate that the column is of a counter data type. The counter value is stored in the value of the column.

This is a very good article that explains the internals of how a counter works in a distributed environment `http://www.datastax.com/dev/blog/whats-new-in-cassandra-2-1-a-better-implementation-of-counters`.

The following screenshot shows that counter value is stored in the value of the column:



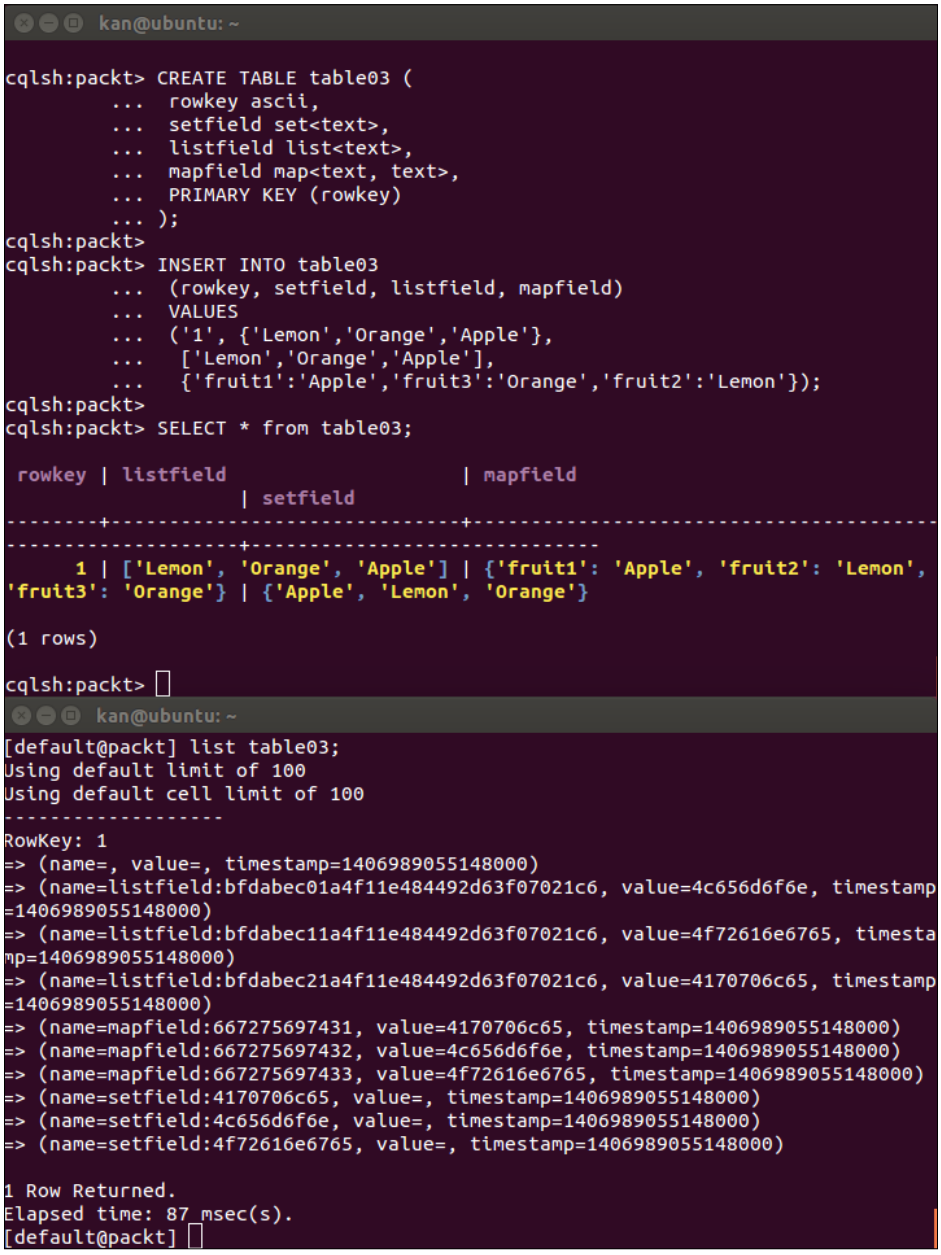
Experiment of the counter data type

Collections

Cassandra also supports collections in its data model to store a small amount of data. Collections are a complex type that can provide tremendous flexibility. Three collections are supported: Set, List, and Map. The type of data stored in each of these collections requires to be defined, for example, a set of timestamp is defined as `set<timestamp>`, a list of text is defined as `list<text>`, a map containing a text key and a text value is defined as `map<text, text>`, and so on. Also, only native data types can be used in collections.

Cassandra reads a collection in its entirety and the collection is not paged internally. The maximum number of items of a collection is 64K and the maximum size of an item is 64K.

To better demonstrate the CQL support on these collections, let us create a table in the `packt` keyspace with columns of each collection and insert some data into it, as shown in the following screenshot:



Experiment on collections

How to update or delete a collection?

CQL also supports updation and deletion of elements in a collection. You can refer to the relevant information in DataStax's documentation at `http://www.datastax.com/documentation/cql/3.1/cql/cql_using/use_collections_c.html`.

As in the case of native data types, let us walk through each collection below.

Set

CQL uses sets to keep a collection of unique elements. The benefit of a set is that Cassandra automatically keeps track of the uniqueness of the elements and we, as application developers, do not need to bother on it.

CQL uses curly braces ({}) to represent a set of values separated by commas. An empty set is simply `{}`. In the previous example, although we inserted the set as `{'Lemon', 'Orange', 'Apple'}`, the input order was not preserved. Why?

The reason is in the mechanism of how Cassandra stores the set. Internally, Cassandra stores each element of the set as a single column whose column name is the original column name suffixed by a colon and the element value. As shown previously, the ASCII values of `'Apple'`, `'Lemon'`, and `'Orange'` are `0x4170706c65`, `0x4c656d6f6e`, and `0x4f72616e6765`, respectively. So they are stored in three columns with column names, `setfield:4170706c65`, `setfield:4c656d6f6e`, and `setfield:4f72616e6765`. By the built-in order column-name-nature of Cassandra, the elements of a set are sorted automatically.

List

A list is ordered by the natural order of the type selected. Hence it is suitable when uniqueness is not required and maintaining order is required.

CQL uses square brackets (`[]`) to represent a list of values separated by commas. An empty list is `[]`. In contrast to a set, the input order of a list is preserved by Cassandra. Cassandra also stores each element of the list as a column. But this time, the columns have the same name composed of the original column name (`listfield` in our example), a colon, and a UUID generated at the time of update. The element value of the list is stored in the value of the column.

Map

A map in Cassandra is a dictionary-like data structure with keys and values. It is useful when you want to store table-like data within a single Cassandra row.

CQL also uses curly braces ({}) to represent a map of keys and values separated by commas. Each key-value pair is separated by a colon. An empty map is simply represented as {}. Conceivably, each key/value pair is stored in a column whose column name is composed of the original map column name followed by a colon and the key of that pair. The value of the pair is stored in the value of the column. Similar to a set, the map sorts its items automatically. As a result, a map can be imagined as a hybrid of a set and a list.

User-defined type and tuple type

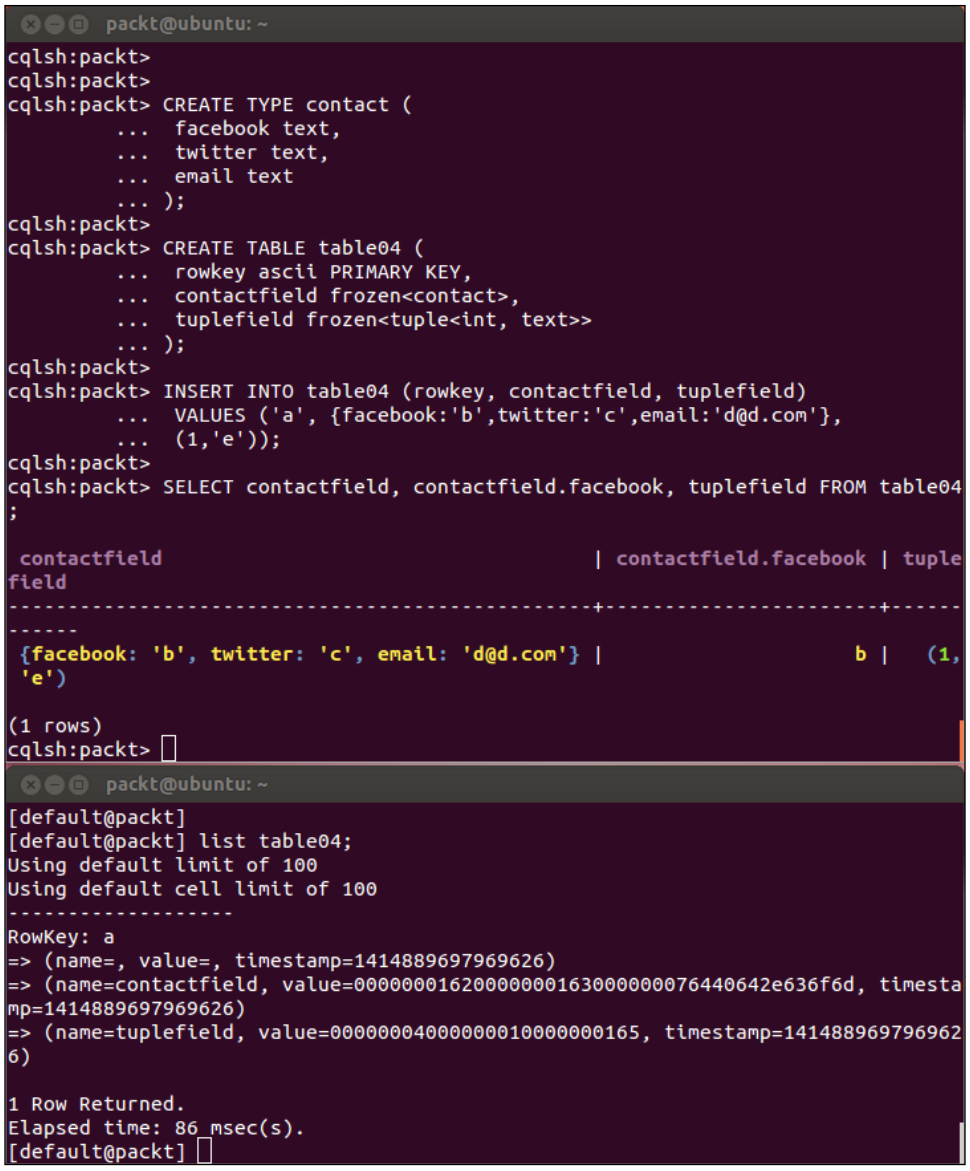
Cassandra 2.1 introduces support for **User-Defined Types** (**UDT**) and tuple types.

User-defined types are declared at the keyspace level. A user-defined type simplifies handling a group of related properties. We can define a group of related properties as a type and access them separately or as a single entity. We can map our UDTs to application entities. Another new type for CQL introduced by Cassandra 2.1 is the tuple type. A tuple is a fixed-length set of typed positional fields without labels.

We can use user-defined and tuple types in tables. However, to support future capabilities, a column definition of a user-defined or tuple type requires the `frozen` keyword. Cassandra serializes a frozen value having multiple components into a single value. This means we cannot update parts of a UDT value. The entire value must be overwritten. Cassandra treats the value of a frozen UDT like a `BLOB`.

We create a UDT called `contact` in the `packt` keyspace and use it to define `contactfield` in `table04`. Moreover, we have another column, `tuplefield`, to store a tuple in a row. Pay attention to the syntax of the `INSERT` statement for UDT and tuple. For UDT, we may use a dotted notation to retrieve a component of the UDT column, such as `contactfield.facebook` in our following example. As shown in `cassandra-cli`, `contactfield` is stored as a single value, `0000000162000000016300000076440642e636f6d`.

The value concatenates each UDT component in sequence with the format, a length of 4 bytes indicating the length of the component value and the component value itself. So, for `contactfield.facebook`, `0x00000001` is the length and `0x62` is the byte value of `'a'`. Cassandra applies the same treatment to a tuple:



Experiment of user-defined and tuple types

Further information can be found at DataStax's documentation, available at the following links: `http://www.datastax.com/documentation/cql/3.1/cql/cql_using/cqlUseUDT.html`

- `http://www.datastax.com/documentation/developer/python-driver/2.1/python-driver/reference/tupleTypes.html`

Summary

This chapter is the second part of Cassandra data modeling. We have learned the basics of Cassandra Query Language (CQL), which offers a SQL-like language to implement a Cassandra data model and operate the data inside. Then a very detailed walkthrough, with ample examples of native data types, more advanced collections, and new user-defined and tuple types, was provided to help you know how to select appropriate data types for your data models. The internal storage of each data type was also explained to let you know how Cassandra implements its data types.

In the next chapter, we will learn another important element of a Cassandra query—indexes.

4
Indexes

There is no doubt that Cassandra can store a gigantic volume of data effortlessly. However, if we cannot efficiently look for what we want in such a data abyss, it is meaningless. Cassandra provides very good support to search and retrieve the desired data by the primary index and secondary index.

In this chapter, we will look at how Cassandra uses the primary index and the secondary index to spotlight the data. After developing an understanding of them, we can then design a high-performance data model.

Primary index

Cassandra is a column-based database. Each row can have different number of columns. A cell is the placeholder of the value and the timestamp data is identified by a row and column. Each cell can store values that are less than 2 GB. The rows are grouped by partitions. The maximum number of cells per partition is limited to the condition that the number of rows times the number of columns is less than 2 billion. Each row is identified by a row key that determines which machine stores the row. In other words, the row key determines the node location of the row. A list of row keys of a table is known as a primary key. A primary index is just created on the primary key.

A primary key can be defined on a single column or multiple columns. In either case, the first component of a table's primary key is the partition key. Each node stores a data partition of the table and maintains its own primary key for the data that it manages. Therefore, each node knows what ranges of row key it can manage and the rows can then be located by scanning the row indexes only on the relevant replicas. The range of the primary keys that a node manages is determined by the partition key and a cluster-wide configuration parameter called partitioner. Cassandra provides three choices to partitioner that will be covered later in this chapter.

A primary key can be defined by the CQL keywords `PRIMARY KEY`, with the column(s) to be indexed. Imagine that we want to store the daily stock quotes into a Cassandra table called `dayquote01`. The `CREATE TABLE` statement creates a table with a simple primary key that involves only one column, as shown in the following screenshot:

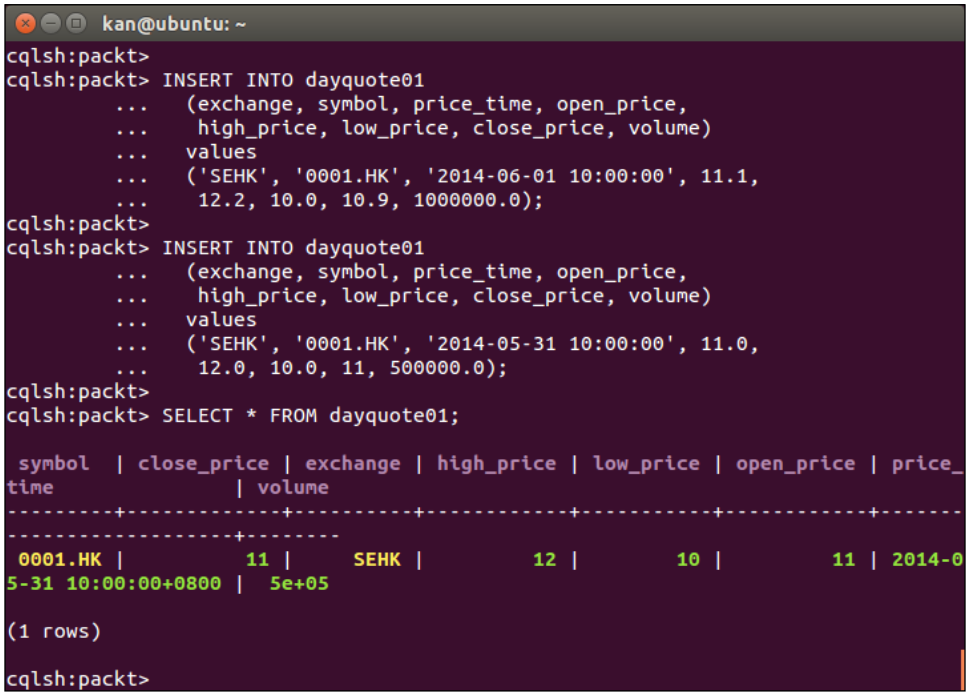
```
kan@ubuntu: ~
cqlsh:packt>
cqlsh:packt>
cqlsh:packt> CREATE TABLE dayquote01 (
         ...   symbol varchar PRIMARY KEY,
         ...   exchange varchar,
         ...   price_time timestamp,
         ...   open_price float,
         ...   high_price float,
         ...   low_price float,
         ...   close_price float,
         ...   volume double
         ... );
cqlsh:packt>
```

The `symbol` field is assigned the primary key of the `dayquote01` table. This means that all the rows of the same symbol are stored on the same node. Hence, this makes the retrieval of these rows very efficient.

Alternatively, the primary key can be defined by an explicit `PRIMARY KEY` clause, as shown in the following screenshot:

```
kan@ubuntu: ~
cqlsh:packt>
cqlsh:packt> CREATE TABLE dayquote02 (
         ...   symbol varchar,
         ...   exchange varchar,
         ...   price_time timestamp,
         ...   open_price float,
         ...   high_price float,
         ...   low_price float,
         ...   close_price float,
         ...   volume double,
         ...   PRIMARY KEY (symbol)
         ... );
cqlsh:packt>
```

Unlike relational databases, Cassandra does not enforce a unique constraint on the primary key, as there is no *primary key violation* in Cassandra. An `INSERT` statement using an existing row key is allowed. Therefore, in CQL, `INSERT` and `UPDATE` act in the same way, which is known as **UPSERT**. For example, we can insert two records into the table `dayquote01` with the same symbol and no primary key violation is alerted, as shown in the following screenshot:



The returned query result contains only one row, not two rows as expected. This is because the primary key is the symbol and the row in the latter `INSERT` statement overrode the record that was created by the former `INSERT` statement. There is no warning for a duplicate primary key. Cassandra simply and quietly updated the row. This silent UPSERT behavior might sometimes cause undesirable effects in the application logic.

Hence, it is very important for an application developer to handle duplicate primary key situations in the application logic. Do not rely on Cassandra to check the uniqueness for you.

In fact, the reason why Cassandra behaves like this becomes more clear when we know how the internal storage engine stores the row, as shown by Cassandra CLI in the following screenshot:

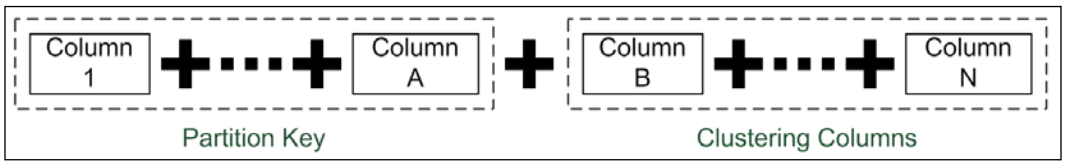
```
kan@ubuntu: ~
[default@packt]
[default@packt] list dayquote01;
Using default limit of 100
Using default cell limit of 100
-------------------
RowKey: 0001.HK
=> (name=, value=, timestamp=1407005232181000)
=> (name=close_price, value=41300000, timestamp=1407005232181000)
=> (name=exchange, value=5345484b, timestamp=1407005232181000)
=> (name=high_price, value=41400000, timestamp=1407005232181000)
=> (name=low_price, value=41200000, timestamp=1407005232181000)
=> (name=open_price, value=41300000, timestamp=1407005232181000)
=> (name=price_time, value=0000014650014900, timestamp=1407005232181000)
=> (name=volume, value=411e848000000000, timestamp=1407005232181000)

1 Row Returned.
Elapsed time: 23 msec(s).
[default@packt]
```

The row key is `0001.HK`. It is used to locate which node is used to store the row. Whenever we insert or update the row of the same row key, Cassandra blindly locates the row and modifies the columns accordingly, even though an `INSERT` statement has been used.

Although a single column primary key is not uncommon, a primary key composed of more than one column is much more practical.

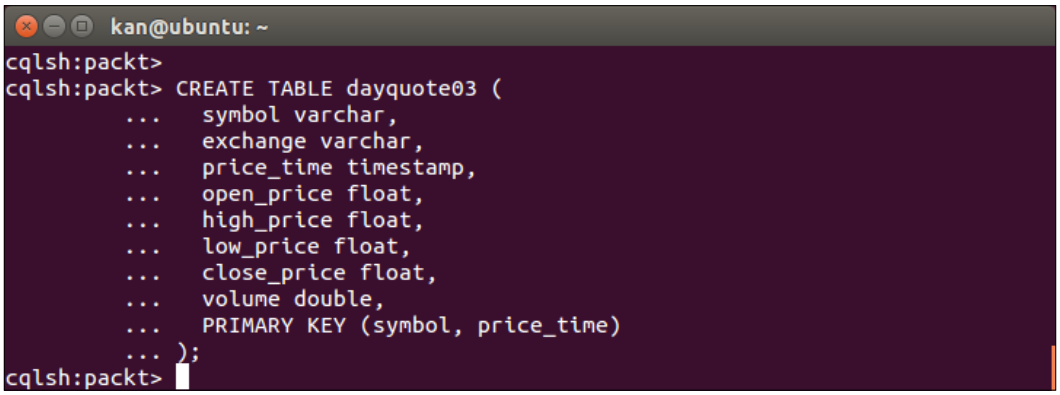
Compound primary key and composite partition key

A compound primary key is composed of more than one column. The order of the columns is important. The structure of a compound primary key is depicted in the following figure:



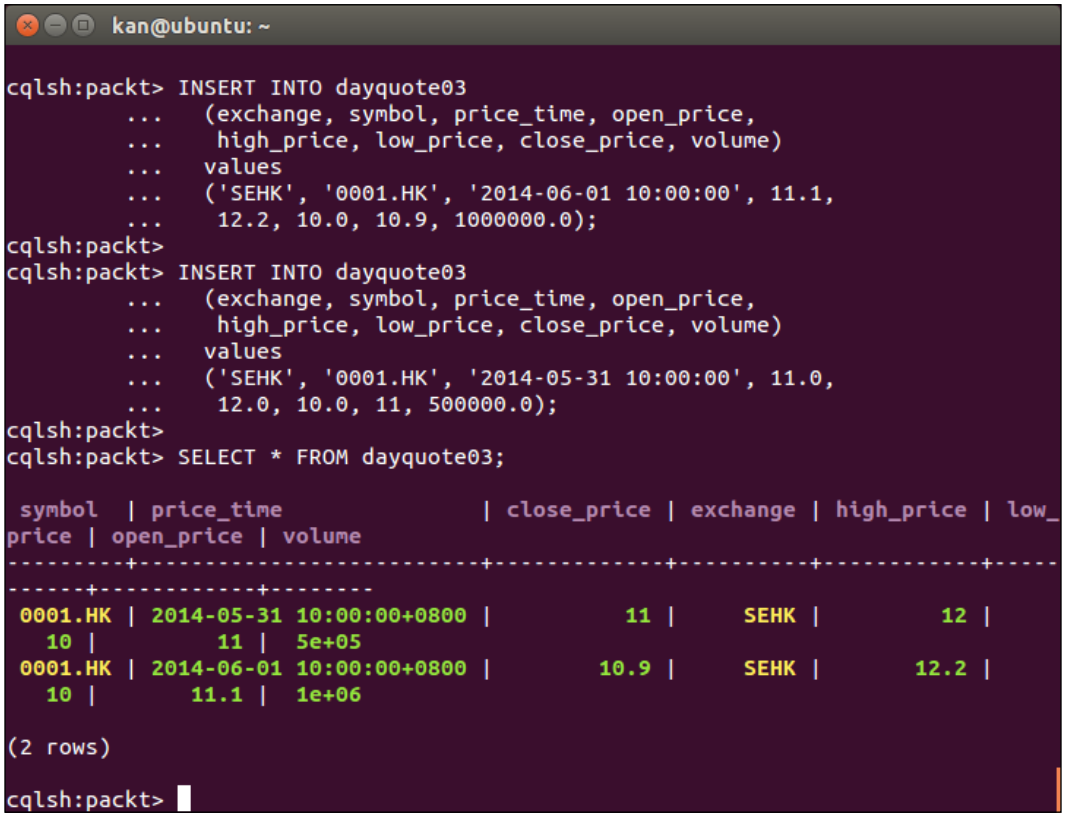
Columns 1 to A are used as the partition key for Cassandra to determine the node location for the partition. The remaining columns, columns B to N, are referred to as the clustering columns for the ordering of data. The clustering columns are used to locate a unique record in the data node. They are ordered, by default, and have the ability to use the `ORDER BY [DESC]` clause in the `SELECT` statements. Moreover, we can get the `MIN` or `MAX` values for clustering keys with the `LIMIT 1` clause. We also need to use the clustering columns for the predicates in a `WHERE` clause. We cannot leave out one when trying to build a query.

To define a compound primary key, an explicit `PRIMARY KEY` clause must be used in the `CREATE TABLE` or `ALTER TABLE` statements. We can define a compound primary key for the table `dayquote03`, as shown in the following screenshot:

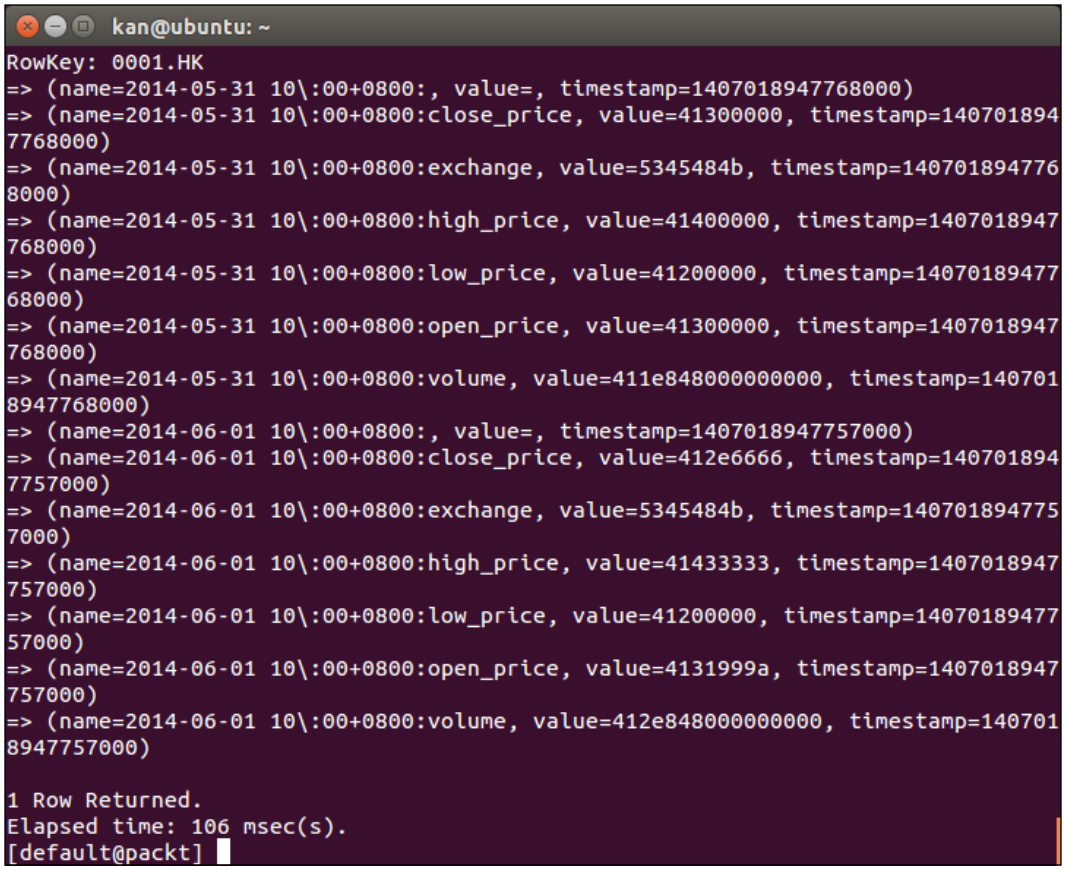


Because the first part of the primary key (that is `symbol`) is the same as that of the simple primary key, the partition key is the same as that in `dayquote01`. Therefore, the node location is the same regardless of whether the primary key is compound or not, as in this case.

So, what is difference between the simple primary key (`symbol`) and this compound one (`symbol, price_time`)? The additional field `price_time` instructs Cassandra to guarantee the clustering or ordering of the rows within the partition by the values of `price_time`. Thus, the compound primary key sorts the rows of the same symbol by `price_time`. We insert two records into the `dayquote03` table and select all the records to see the effect, as shown in the following screenshot:



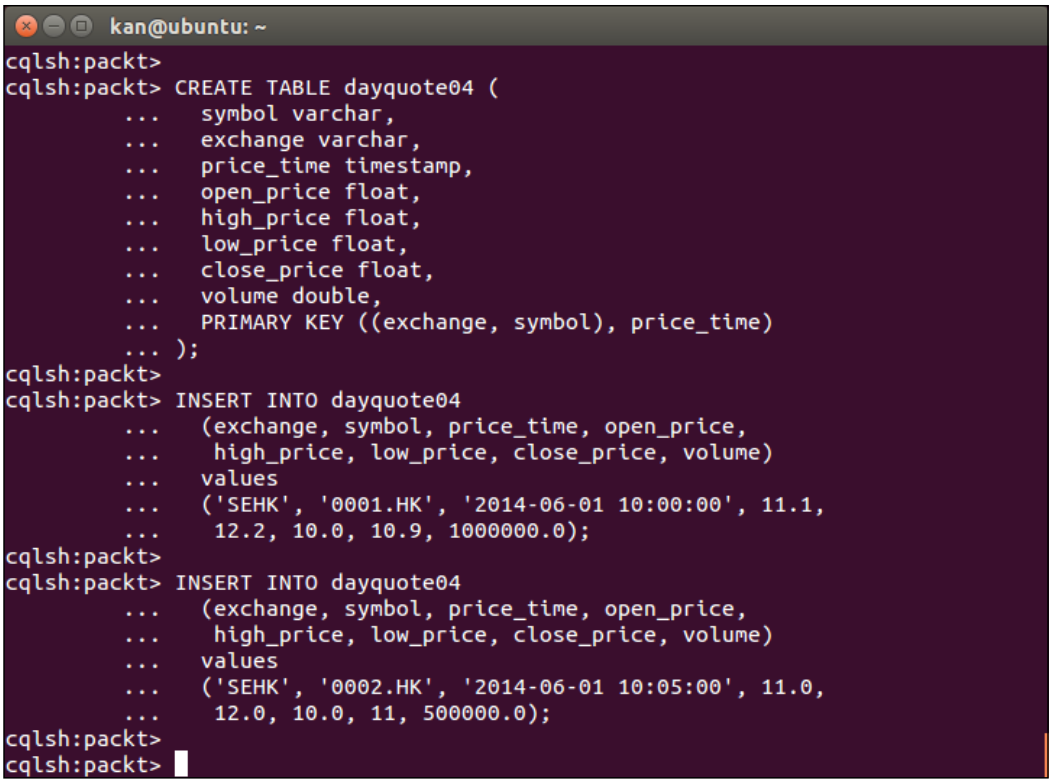
Two records are returned as expected (compared to only one record in `dayquote01`). Moreover, the ordering of the results is sorted by the values of `price_time`. The following screenshot shows the internal view of the rows in the `dayquote03` table:



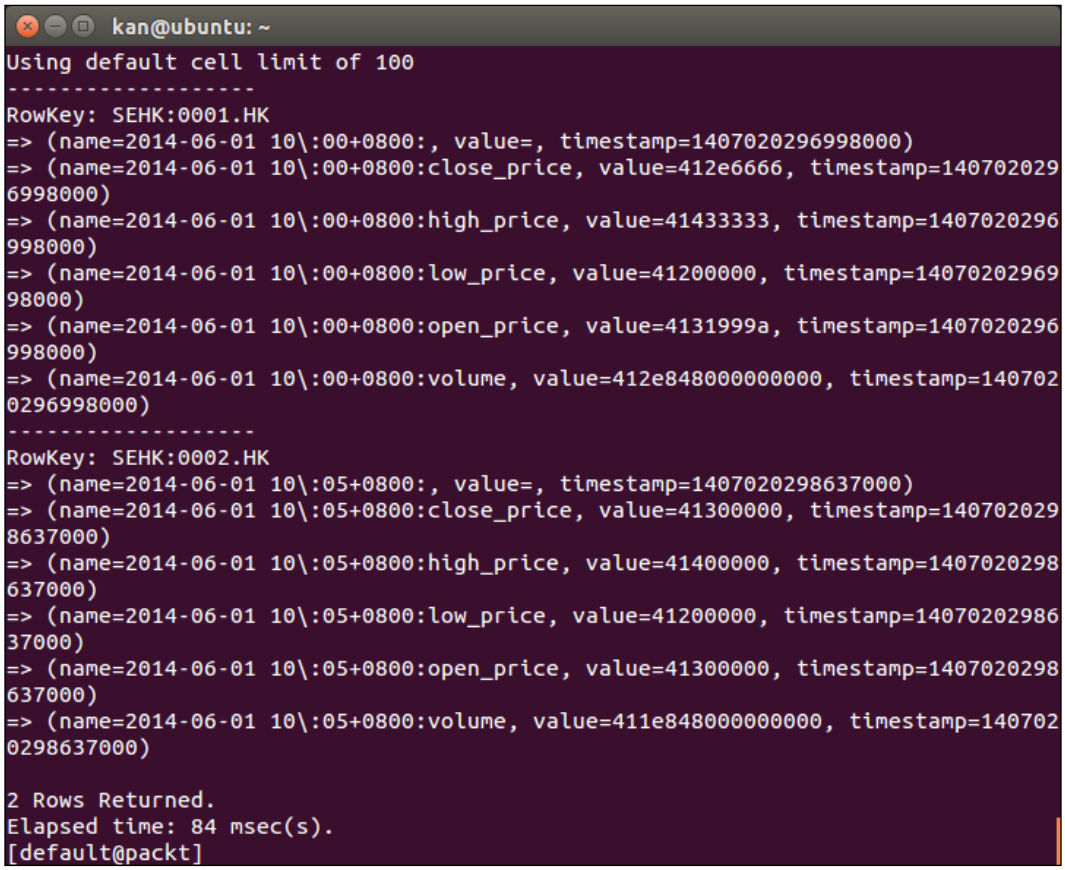
The row key is still the partition key, that is, `0001.HK`. However, Cassandra stores the two rows returned by the CQL `SELECT` statement, as one single internal row in its storage. The values of the clustering columns are used as a prefix to the columns that are not specified in the `PRIMARY KEY` clause. As Cassandra stores the internal columns in the sorting order of the column name, the rows returned by the CQL `SELECT` statement are sorted inherently. In a nutshell, on a physical node, when the rows for a partition key are stored in the order that is based on the clustering columns, the retrieval of rows is very efficient.

Now you know that the first part of a compound primary key is the partition key. If we need to keep on storing 3,000 daily quotes (around 10 years) for `0001.HK`, although the CQL `SELECT` statement returns 3,000 virtual rows, Cassandra requires to store these 3,000 virtual rows as one entire row on a node by the partition key. The size of the entire row gets bigger and bigger on a node as a result of storing more and more daily quotes. The row will quickly become gigantic over a period of time and will then pose a serious performance problem, as a result of an unbalanced cluster. The solution is a feature offered by Cassandra called composite partition key.

The composite partition key spreads the data over multiple nodes. It is defined by an extra set of parentheses in the `PRIMARY KEY` clause. Let us create another table `dayquote04` with a composite partition key in order to illustrate the effect. The columns `exchange` and `symbol` are now members of a composite partition key, whereas the column `price_time` is a clustering column. We insert the same two records of different symbols into `dayquote04`, as shown in the following screenshot:



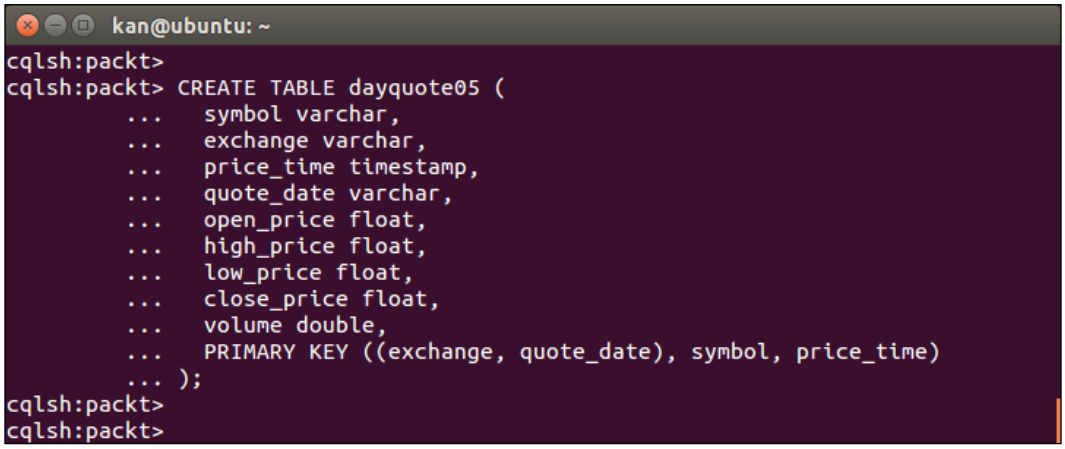
With reference to the following screenshot, two internal rows are returned with their row keys as `SEHK:0001.HK` and `SEHK:0002.HK`, respectively. Internally, Cassandra concatenates the columns in the composite partition key together as an internal row key. In short, the original row without a composite partition key is now split into two rows. As the row keys are now different from each other, the corresponding rows can be stored on different nodes. The value of the clustering column `price_time` is still used as a prefix in the internal column name to preserve the ordering of data:



Time-series data

Cassandra is very suitable for handling time-series type of data, such as web server logfiles, usage data, sensor data, SIP packets, and so on. The tables `dayquote01` to `dayquote04` in the previous sections are used to store the daily stock quotes is an example of the time-series data.

We have just seen in the last section that a composite partition key is a better way of not overwhelming the row. It limits the size of the rows on the basis of a symbol. However, this does partially solve the problem. The size of the row of a symbol still grows over a period of time. Do you have any other suggestion? We can define an artificial column, `quote_date`, in the table and set the composite partition key to `exchange` and `quote_date` instead, as shown in the following screenshot:



Now the composite partition key limits the size of the rows on a daily basis, and makes the rows more manageable. This way of doing is analogous to inserting the data into different buckets labeled by a particular date. Hence, it is given a name called the **date bucket pattern**. Partitioning by the date also makes table maintenance easier by allowing you to drop the partition of `quote_date`. One drawback of the date bucket pattern is that you always need to know the partition key in order to get the rows. So, in `dayquote05`, you cannot get the latest `quote_date` value using the `ORDER BY DESC` and `LIMIT 1` clauses.

The date bucket pattern gives an application developer a design option to attain a more balanced cluster, but how balanced a cluster is depends on a number of factors in which the most important one is the selection of the partitioner.

Partitioner

A partitioner is basically a hash function used to calculate the `TOKEN()` (the hash value) of a row key and so, it determines how data is distributed across the nodes in a cluster. Choosing a partitioner determines which node is used to place the first copy of data. Each row of data is uniquely identified by a partition key and is distributed across the cluster by the value of the `TOKEN()`. Cassandra provides the following three partitioners:

- `Murmur3Partitioner` (default since version 1.2)
- `RandomPartitioner` (default before version 1.2)
- `ByteOrderedPartitioner`

Murmur3Partitioner

`Murmur3Partitioner` provides faster hashing and improved performance than the partitioner `RandomPartitioner`. It is the default partitioning strategy and the right choice for new clusters in almost all cases. It uses the *MurmurHash* function that creates a 64-bit hash value of the partition key. The possible range of hash values is from -2^{63} to $+2^{63}$ -1. When using `Murmur3Partitioner`, you can page through all the rows using the `TOKEN()` function in a CQL `SELECT` statement.

RandomPartitioner

`RandomPartitioner` was the default partitioner prior to Cassandra Version 1.2. It distributes data evenly across the nodes using an *MD5* hash value of the row key. The possible range of hash values is from 0 to 2^{127} -1. The MD5 hash function is slow in performance, that is why Cassandra has moved to Murmur3 hashes. When using `RandomPartitioner`, you can page through all rows using the `TOKEN()` function in a CQL `SELECT` statement.

ByteOrderedPartitioner

`ByteOrderedPartitioner`, as its name suggests, is used for ordered partitioning. This partitioner orders rows lexically by key bytes. Tokens are calculated by looking at the actual values of the partition key data and using a hexadecimal representation of the leading character(s) in a key. For example, if you wanted to partition rows alphabetically, you can assign a B `TOKEN()` using its hexadecimal representation of `0x42`.

Using `ByteOrderedPartitioner` allows ordered scans by a primary key as though you were moving a cursor through a traditional index in a relational table. This type of range scan query is not possible using `RandomPartitioner` because the keys are stored in the order of their MD5 hash, and not in the sequential order of the keys.

Apparently, performing range-scan on rows sounds like a desirable feature of `ByteOrderedPartitioner`. There are ways to achieve the same functionality using secondary indexes. Conversely, using `ByteOrderedPartitioner` is not recommended for the following reasons:

- **Difficult load balancing**: More administrative overhead is required to load balance the cluster. `ByteOrderedPartitioner` requires administrators to manually calculate partition ranges based on their estimates of the partition key distribution.
- **Sequential writes can cause hot spots**: If the application tends to write or update a sequential block of rows at a time, the writes will not be distributed across the cluster. They all go to one node. This is frequently a problem for applications dealing with timestamped data.
- **Uneven load balancing for multiple tables**: If the application has multiple tables, chances are that these tables have different row keys and different distributions of data. An ordered partitioner that is balanced for one table can cause hot spots and uneven distribution for another table in the same cluster.

Paging and token function

When using the `RandomPartitioner` or `Murmur3Partitioner`, the rows are ordered by the hash of their value. Hence, the order of the rows is not meaningful. Using CQL, the rows can still be paged through even when using `RandomPartitioner` or `Murmur3Partitioner` using the `TOKEN()` function, as shown in the following screenshot:

```
kan@ubuntu: ~
cqlsh:packt>
cqlsh:packt> SELECT symbol FROM dayquote04
         ...  WHERE TOKEN(exchange,symbol) < TOKEN('SEHK','0002.HK');

 symbol
---------
 0001.HK

(1 rows)

cqlsh:packt>
```

`ByteOrderedPartitioner` arranges tokens in the same way as key values, while `RandomPartitioner` and `Murmur3Partitioner` distribute tokens in a completely unordered manner. The `TOKEN()` function makes it possible to page through the unordered partitioner results. It actually queries results directly using tokens.

Secondary indexes

As Cassandra only allows each table to have one primary key, it supports secondary index on columns other than those in the primary key. The benefit is a fast, efficient lookup of data matching the indexed columns in the `WHERE` clause. Each table can have more than one secondary index. Cassandra uses secondary indexes to find the rows that are not using the row key. Behind the scenes, the secondary index is implemented as a separate, hidden table that is maintained automatically by the internal process of Cassandra. As with relational databases, keeping secondary indexes up to date is not free, so unnecessary indexes should be avoided.

The major difference between a primary index and a secondary index is that the primary index is a distributed index used to locate the node that stores the row key, whereas the secondary index is a local index just to index the data on the local node.

Therefore, the secondary index will not be able to know immediately the locations of all matched rows without having examined all the nodes in the cluster. This makes the performance of the secondary index unpredictable.

The secondary index is the most efficient when using equality predicates. This is indeed a limitation that must have at least one equality predicate clause to hopefully limit the set of rows that need to be read into memory.

In addition, the secondary index cannot be created on the primary key itself.

Caveat!

Secondary indexes in Cassandra are *NOT* equivalent to those in the traditional RDBMS. They are not akin to a B-tree index in RDBMS. They are mostly like a hash. So, the range queries do not work on secondary indexes in Cassandra, only equality queries work on secondary indexes.

We can use the CQL `CREATE INDEX` statement to create an index on a column after we define a table. For example, we might want to add a column `sector` to indicate the sector that the stock belongs to, as shown in the following screenshot:

```
kan@ubuntu: ~
cqlsh:packt>
cqlsh:packt> CREATE TABLE dayquote06 (
         ...   symbol varchar,
         ...   exchange varchar,
         ...   sector varchar,
         ...   price_time timestamp,
         ...   quote_date varchar,
         ...   open_price float,
         ...   high_price float,
         ...   low_price float,
         ...   close_price float,
         ...   volume double,
         ...   PRIMARY KEY ((exchange, quote_date), symbol, price_time)
         ... );
cqlsh:packt>
cqlsh:packt> INSERT INTO dayquote06
         ...   (exchange, symbol, sector, price_time, open_price,
         ...    high_price, low_price, close_price, volume, quote_date)
         ...   values
         ...   ('SEHK', '0001.HK', 'Properties', '2014-06-01 10:00:00', 11.1,
         ...    12.2, 10.0, 10.9, 1000000.0, '20140601');
cqlsh:packt>
cqlsh:packt> INSERT INTO dayquote06
         ...   (exchange, symbol, sector, price_time, open_price,
         ...    high_price, low_price, close_price, volume, quote_date)
         ...   values
         ...   ('SEHK', '0002.HK', 'Utilities', '2014-06-01 10:05:00', 11.0,
         ...    12.0, 10.0, 11, 500000.0, '20140601');
cqlsh:packt> _
```

If we want to search `dayquote06` for symbols that belong to `Properties`, we might run the command, as shown in the following screenshot:

```
kan@ubuntu: ~
cqlsh:packt>
cqlsh:packt> SELECT symbol FROM dayquote06 WHERE sector = 'Properties';
Bad Request: No indexed columns present in by-columns clause with Equal operator
cqlsh:packt>
```

As `sector` is not in the primary key, we cannot query Cassandra directly by `sector`. Instead, we can create a secondary index on the column `sector` to make this possible, as shown in the following screenshot:

```
kan@ubuntu: ~
cqlsh:packt>
cqlsh:packt> CREATE INDEX dayquote06_sector_idx ON dayquote06 (sector);
cqlsh:packt>
cqlsh:packt> SELECT symbol FROM dayquote06 WHERE sector = 'Properties';

 symbol
---------
 0001.HK

(1 rows)

cqlsh:packt>
```

The index name `dayquote06_sector_idx` is optional, but must be unique within the keyspace. Cassandra assigns a name such as `dayquote06_idx` if you do not provide a name. We can now query Cassandra for daily stock quotes by `sector`.

You can see that the columns in the primary key are not present in the `WHERE` predicate clause in the previous screenshot and Cassandra uses the secondary index to look for the rows matching the selection condition.

Multiple secondary indexes

Cassandra supports multiple secondary indexes on a table. The `WHERE` clause is executed if at least one column is involved in a secondary index. Thus, we can use multiple conditions in the `WHERE` clause to filter the results. When multiple occurrences of data match a condition in the `WHERE` predicate clause, Cassandra selects the least frequent occurrence of a condition to process first so as to have a better query efficiency.

When a potentially expensive query is attempted, such as a range query, Cassandra requires the `ALLOW FILTERING` clause, which can apply additional filters to the result set for values of other non-indexed columns. It works very slowly because it scans all rows in all nodes. The `ALLOW FILTERING` clause is used to explicitly direct Cassandra to execute that potentially expensive query on any `WHERE` clause without creating secondary indexes, despite unpredictability of the performance.

Secondary index do's and don'ts

The secondary index is best on a table that has many rows that contain fewer unique values, that is low cardinality in the relational database terminologies, which is counterintuitive to the relational people. The more unique values that exist in a particular column, the more overhead you will have to query and maintain the index. Hence, it is not suitable for querying a huge volume of records for a small number of results.

Do index the columns with values that have low cardinality. Cassandra stores secondary indexes only for local rows in the data node as a hash-multimap or as bitmap indexes, you can refer to it at `https://issues.apache.org/jira/browse/CASSANDRA-1472`.

Secondary indexes should be avoided in the following situations:

- On high-cardinality columns for a small number of results out of a huge volume of rows

  An index on a high-cardinality column will incur many seeks for very few results. For columns containing unique values, using an index for convenience is fine from a performance perspective, as long as the query volume to the indexed column family is moderate and not under constant load.

- In tables that use a counter column
- On a frequently updated or deleted column

  Cassandra stores tombstones (a marker in a row that indicates that a column was deleted. During compaction, marked columns are deleted in the index (a hidden table) until the tombstone limit reaches 100 K cells. After exceeding this limit, the query that uses the indexed value will fail.

- To look for a row in a large partition

  A query on an indexed column in a large cluster typically requires collating responses from multiple data partitions. The query response slows down as more machines get added to the cluster.

Important points to take note of

- Don't index on high-cardinality columns
- Don't use index in tables having a counter column
- Don't index on a frequently updated or deleted column
- Don't abuse the index to look for a row in a large partition

Summary

We have learned about the primary and secondary indexes in this chapter. Related topics such as compound primary key, composite partition key, and partitioner are also introduced. With the help of the explanation of the internal storage and inner working mechanisms of Cassandra, you should now be able to state the difference between the primary index and the secondary index, as well as use them properly in your data model.

In the next chapter, we will start building the first version of the technical analysis application using Cassandra and Python. A quick installation and setup guide on how to connect Python to Cassandra and collect market data will also be provided.

5
First-cut Design and Implementation

Riding on the ingredients of a Cassandra data model that were explained in the previous chapters, now it is time to put them into a working application. We will begin defining what we really want to store and inquire in the data model, setting up the environment, writing the program code, and finally testing the application.

The application to be built is a Stock Screener Application, which stores the historical stock quotes in a Cassandra database for technical analysis. The application collects the stock quote data from a free source on the Internet and then applies some technical analysis indicators to find out the buy and sell reference signals. A brief and quick introduction of technical analysis is given in order to enable you to easily understand what the application does. Although it is oversimplified in architecture and not complete in features, it does provide a good foundation for further improvement on more advanced features to be made by you.

Disclaimer

It should be assumed that the methods, techniques, or indicators discussed in this book will be profitable and will not result in losses. There is no assurance that the strategies and methods presented will be successful, or that you will become a profitable trader. The past performance and results of any trading system or trading methodology are not necessarily indicative of future results. You should not trade with money that you cannot afford to lose. The examples discussed and presented in this book are for educational purposes only. These are not solicitations of any order to buy or sell. I assume no responsibility for your trading results. No representation is being made that any account will, or is likely to, achieve profits or losses similar to those discussed in this book. There is a very high degree of risk in trading. You are encouraged to consult a certified financial advisor before making any investment or trading decisions.

Stock Screener Application

In this section, we will learn some background information of the sample application. Then, we will discuss the data source, the initial data model, and the high-level processing logic of the application.

An introduction to financial analysis

A stock screener is a utility program that uses a certain set of criteria to screen a large number of stocks that match your preferences. It is akin to a search engine on stocks but not on websites. The screening criteria might be based on fundamental and/or technical analysis methods.

Firstly, let us look at what fundamental analysis is.

Fundamental analysis

Fundamental analysis involves analyzing a company's historical and present financial statements and health, its management and competitive advantages, and its competitors and markets, in order to assess and calculate the value of a company stock and predict its probable price evolution. The goal is to make financial forecasts and find out the undervalued stock (stock that is cheap, in other words) for buy-and-hold.

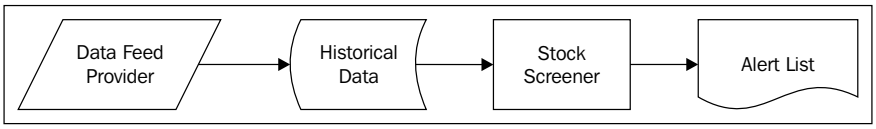
In contrast, technical analysis is a totally different approach.

Technical analysis

Technical analysis is a stock analysis methodology used to forecast the direction of prices through the study of past market data, primarily price and volume. The fundamental principle of technical analysis is that the market price reflects all the relevant information, so the analysis looks at the history of the trading patterns rather than external drivers such as economic, fundamental, and news events.

In this book, technical analysis is solely used for the Stock Screener Application. As technical analysis focuses on price actions, the Stock Screener Application requires stock price data as its input and then it applies technical analysis techniques to determine whether the stock fulfills the buy or sell conditions. Whenever such a condition is fulfilled, we can say that a trading signal is triggered.

The conceptual design of the Stock Screener Application is shown in the following figure:



We will go through the preceding figure from the left to the right. **Data Feed Provider** is the source of stock quote data that is collected from a free Data Feed Provider on the Internet, such as Yahoo! Finance. It should be noted that Yahoo! Finance provides free-of-charge **end-of-day** (**EOD**) stock quote data, thus providing the daily stock quote. If you want the **Stock Screener** to produce intraday signals, you need to look for other Data Feed Providers who typically have a wide range of paid service offers available. **Historical Data** is a repository to store the historical stock quote data. **Stock Screener** is the application to be developed in this chapter. Lastly, **Alert List** is a list of trading signals found by the **Stock Screener**.

Before we proceed to the high-level design of the **Stock Screener**, I would like to highlight the reasons of establishing the **Historical Data** repository. There are three major reasons. First, it can save tremendous network bandwidth from repeatedly downloading historical stock quote data from the Data Feed Provider (actually, Yahoo! Finance provides as many as 10 years of historical price data.) Second, it serves as a canonical data model so that the **Stock Screener** does not need to cater for the different data formats of different Data Feed Providers. Finally, the **Stock Screener** can still perform technical analysis on the historical data even though it is disconnected from the Internet.

Stock quote data

Technical analysis only focuses on price action. So what is price action? Price action is simply the movement of a stock's price. It is encompassed in technical and chart pattern analysis in an attempt to discover the order in the seemingly random movement of price.

On a single day, the price action of a stock can be summarized by four important prices:

- **Open price**: This is the starting price for that day
- **High price**: This is the highest price for that day
- **Low price**: This is the lowest price for that day
- **Close price**: This is the closing price for that day

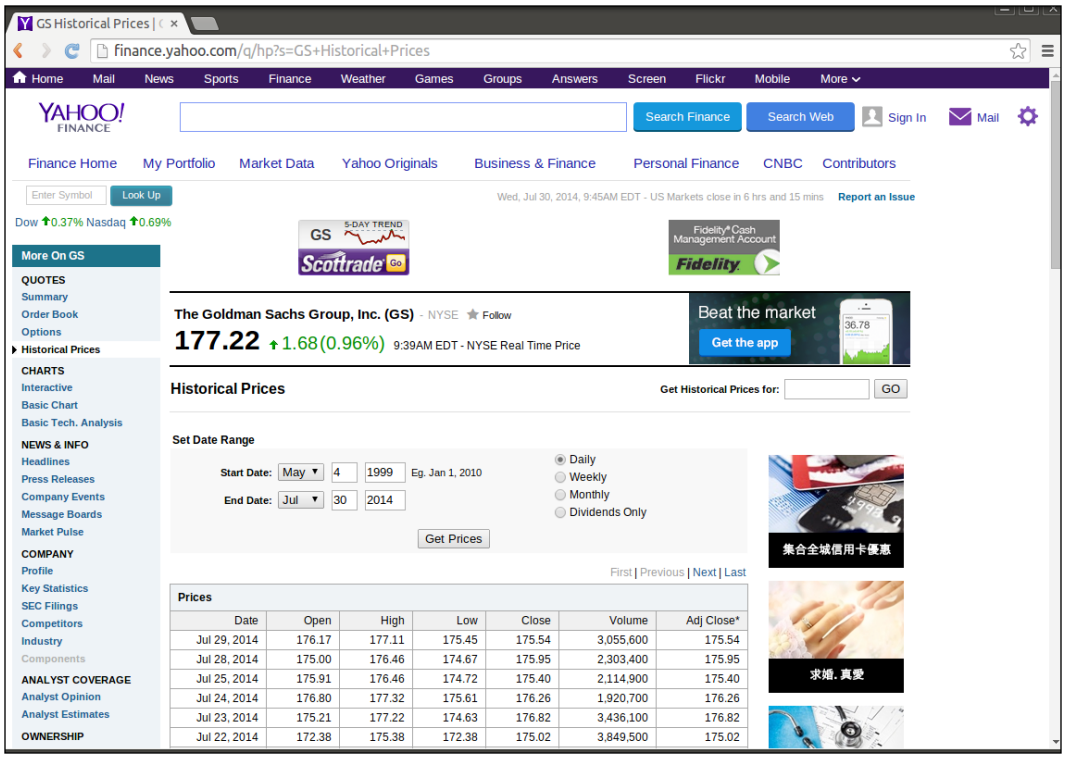
These four prices are usually abbreviated as OHLC. In addition to OHLC, another measure of how much of a given stock has been traded in a given period of time is known as Volume. For a complete trading day, the volume is called daily volume.

Only five attributes such as **open price**, **high price**, **low price**, **close price**, and **volume** (**OHLCV**), provide all the necessary and sufficient data for technical analysis of stock. Now we know the input for technical analysis, but how do we get them?

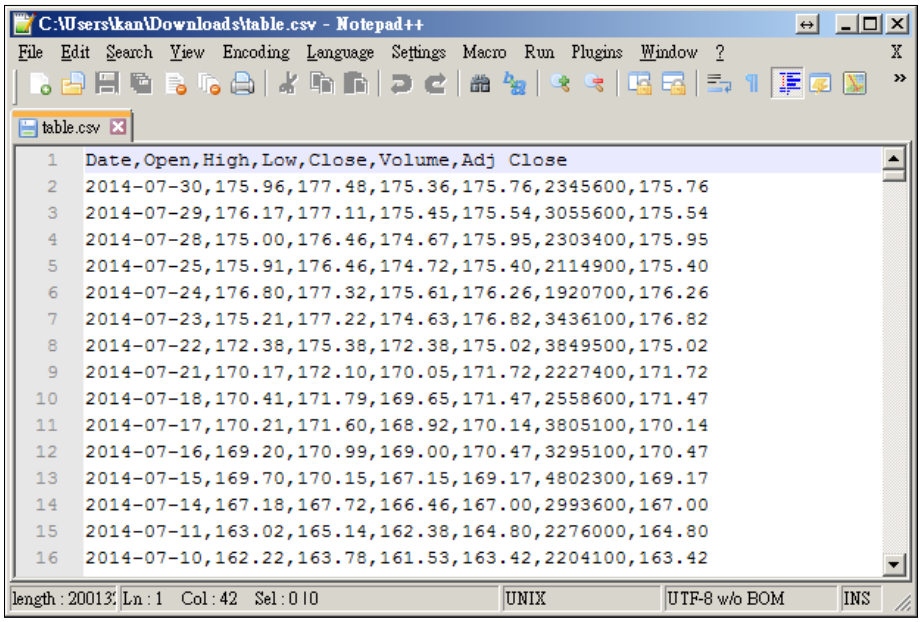
Many websites provide free-of-charge stock quote data that are very easy to obtain, and are especially suitable for amateur or retail traders. The following websites are just a few of them listed for your reference:

- Yahoo! Finance: `http://finance.yahoo.com`
- Google Finance: `https://www.google.com/finance`
- EODData: `http://eoddata.com`

However, there is a caveat that stock quote data might have errors, for example, incorrect high and low prices. In this book, I selected Yahoo! Finance as the prime Data Feed Provider. The following screenshot is a sample of the historical prices of a stock called *GS*:



As you scroll to the bottom of the web page, you will see a link *Download to Spreadsheet*. When you click on this link, the historical stock quote data can be downloaded as a **Comma Separated Values** (**CSV**) file. An excerpt of the CSV file is shown in the following screenshot:



Of course, we can manually download the historical stock quote data from the website. Nonetheless, it becomes impractical when we want to download the data of many different stocks on a daily basis. Thus, we will develop a program to automatically collect the data feed.

Initial data model

We now know that a single daily price action consists of a stock symbol, trading date, open price, high price, low price, close price, and volume. Obviously, a sequence of price action measured typically at successive trading days is of a time-series nature and Cassandra is very suitable for storing time-series type data.

As mentioned previously, it is beneficial to store the collected stock quote data locally in a repository. Therefore, we will implement the repository as a table in a Cassandra database.

We can use CQL to define a table called `quote` to store the historical prices:

```
// table to store historical stock quote data
CREATE TABLE quote (
  symbol varchar, // stock symbol
  price_time timestamp, // timestamp of quote
  open_price float, // open price
  high_price float, // high price
  low_price float, // low price
  close_price float, // close price
  volume double, // volume
  PRIMARY KEY (symbol, price_time) // primary key
);
```

The column data types and names are self-explanatory.

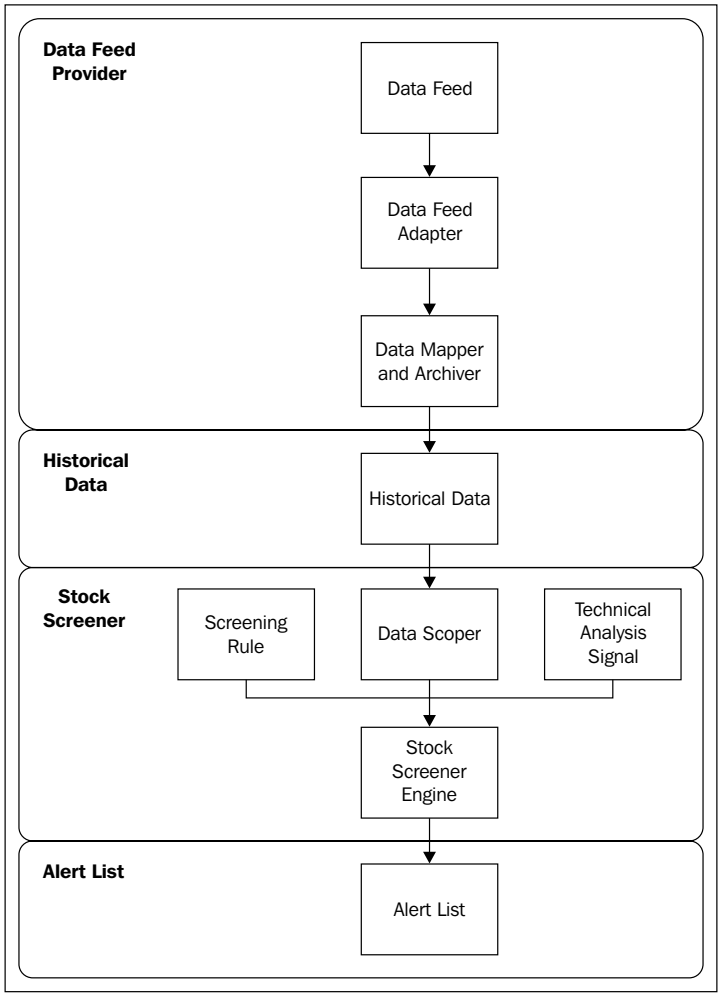
One useful technique of designing a Cassandra data model is to imagine the visual representation of the internal storage of a row. The following figure is such an example:

RowKey	2014-07-30 00:00:	2014-07-30 00:00:close_price	2014-07-30 00:00:high_price	2014-07-30 00:00:low_price	2014-07-30 00:00:open_price	2014-07-30 00:00:volume	2014-07-31 00:00:	...
GS		175.76	177.48	175.36	175.96	2345600		

Based on the design of the primary key, the row key is `symbol` and the clustering column is `price_time`. It is expected that a row will become a wide row, as more historical stock quote data gets added to it. Without the internal storage picture, this might not be easy to spot in the initial data model design stage. For the time being, we just take note of the potential wide row problem and leave it as is (one possible solution is the date bucket pattern).

Processing flow

The following figure shows the processing flow of the **Stock Screener**, which elaborates the conceptual design with a more detailed sequence of steps. Each of the building blocks is explained starting first from the top, as shown in the following screenshot:



Data Feed Provider consists of **Data Feed**, **Data Feed Adapter**, and **Data Mapper and Archiver**. Yahoo! Finance is chosen as the data feed. **Data Feed Adapter** is used to deal with the different connectivity and interfacing methods if we switch to other Data Feed Providers. **Data Mapper and Archiver** caters for the different stock quote data formats and standardizes them to the corresponding columns of the `quote` table.

The `quote` table is the **Historical Data** repository and has been explained previously.

We now turn our focus to the core **Stock Screener**. The heart of the **Stock Screener** is the **Stock Screener Engine** that uses the **Screening Rule** on the **Historical Data**, which is filtered by the **Data Scoper**. The **Screen Rule** is used by one or more **Technical Analysis Signals** so that the **Stock Screener Engine** produces an alert if the conditions of the **Technical Analysis Signals** are met.

The alert generated by the **Stock Screener Engine** is presented in the form of an **Alert List**, which can be kept as records or distributed through other means.

Basically, the **Data Feed Provider** and the **Stock Screener** need not run in the same process. They work in an asynchronous mode. This means that the **Data Feed Provider** can collect, map, and archive the historical stock quote data into the **Historical Data** repository, whereas the **Stock Screener** can analyze and produce alerts independently.

We have come up with a high-level design of the application, the next thing to do is conceivably see how it can be implemented.

System design

In this section, we will select the appropriate software for various system components.

The operating system

When considering the implementation, the first fundamental choice is the operating system. The single most important constraint is that it must be supported by Cassandra. For this book, I have selected Ubuntu 14.04 LTS 64-bit Version, which can be obtained at the official Ubuntu website, `http://www.ubuntu.com/`. You should be able to painlessly set up your Linux box by following the verbose installation instructions.

However, it is entirely up to you to use any other operating systems, supported by Cassandra, such as Microsoft Windows and Mac OS X. Please follow the respective operating system installation instructions to set up your machine. I have already considered the portability of the Stock Screener. As you will see in the subsequent sections, the Stock Screener Application is designed and developed in order to be compatible with a great number of operating systems.

Java Runtime Environment

As Cassandra is Java-based, a **Java Runtime Environment** (**JRE**) is required as a prerequisite. I have used Oracle Java SE Runtime Environment 7 64-bit Version 1.7.0_65. It is provided at the following URL:`http://www.oracle.com/technetwork/java/javase/downloads/jre7-downloads-1880261.html`.

Of course, I have downloaded the Linux x64 binary and followed the instructions at `http://www.datastax.com/documentation/cassandra/2.0/cassandra/install/installJreDeb.html` to properly set up the JRE.

At the time of writing, Java SE has been updated to Version 8. However, I have not tested JRE 8 and DataStax recommends JRE 7 for Cassandra 2.0 too. Therefore, I will stick to JRE 7 in this book.

Java Native Access

If you want to deploy Cassandra in production use on Linux platforms, **Java Native Access** (**JNA**) is required to improve Cassandra's memory usage. When installed and configured, Linux does not swap the **Java virtual machine** (**JVM**), and thus avoids any performance related issues. This is recommended as a best practice even when Cassandra, which is to be installed, is for non-production use.

To install JNA on Ubuntu, simply use Aptitude Package Manager with the following command in a terminal:

```
$ sudo apt-get install libjna-java
```

Cassandra version

I used Cassandra Version 2.0.9, which is distributed by DataStax Community, on Debian or Ubuntu. The installation steps are well documented at `http://www.datastax.com/documentation/getting_started/doc/getting_started/gettingStartedDeb_t.html`.

The installation process typically takes several minutes depending on your Internet bandwidth and the performance of your machine.

DataStax

DataStax is a computer software company based in Santa Clara, California which offers commercial enterprise grade for Apache Cassandra in its DataStax Enterprise product. It also provides tremendous support for the Apache Cassandra community.

Programming language

It is now time to turn our attention to the programming language for the implementation of the Stock Screener Application. For this book, I have chosen Python. Python is a high-level programming language designed for speed of development. It is open source, free, and cross-platform. It possesses a wealthy set of libraries for almost every popular algorithm you can imagine.

You need not be afraid of learning Python if you are not familiar with it. Python is designed such that it is very easy to learn when compared to other programming languages such as C++. Coding a Python program is pretty much like writing pseudocode that improves the speed of development.

In addition, there are many renowned Python libraries used for data analysis, for example, NumPy, SciPy, pandas, scikit-learn, and matplotlib. You can make use of them to quickly build a full-blown application with all the bells and whistles. For the Stock Screener Application, you will use NumPy and pandas extensively.

When it comes to high performance, Python can also utilize Cython, which is an optimizing static complier for Python programs to run as fast as native C or C++ programs.

The latest major version of Python is Python 3. However, there are still many programs running that are written in Python 2. This is caused by the breaking backward compatibility of Python 3 that makes the migration of so many libraries written in Python 2 to Python 3, a very long way to go. Hence, the coexistence of Python 2 and Python 3 is expected for quite a long time in future. For this book, Python 2.7.x is used.

The following steps are used to install Python 2.7 in Ubuntu using a terminal:

```
$ sudo apt-get -y update
$ sudo apt-get -y upgrade
$ sudo apt-get install python-pip python-dev \
$ python2.7-dev build-essential
```

Once the installation is complete, type the following command:

```
$ python --version
```

You should see the version string returned by Python, which tells you that the installation has been successful.

One problem that many Python beginners face is the cumbersome installation of the various library packages. To rectify this problem, I suggest that the reader downloads the Anaconda distribution. Anaconda is completely free and includes almost 200 of the most popular Python packages for Science, Mathematics, engineering, and data analysis. Although it is rather bulky in size, it frees you from the Python package hustle. Anaconda can be downloaded at `http://continuum.io/downloads`, where you can select the appropriate versions of Python and the operating system. It is straightforward to install Anaconda by following the installation instructions, so I will not detail the steps here.

Cassandra driver

The last item of the system environment is the driver software for Python to connect to a Cassandra database. In fact, there are several choices out there, for example, pycassa, Cassandra driver, and Thrift. I have chosen Python Driver 2.0 for Apache Cassandra distributed by DataStax. It exclusively supports CQL 3 and Cassandra's new binary protocol, which was introduced in Version 1.2. More detailed information can be found at `http://www.datastax.com/documentation/developer/python-driver/2.0/common/drivers/introduction/introArchOverview_c.html`.

The driver can be easily installed with pip in a Ubuntu terminal:

```
$ pip install cassandra-driver
```

pip

pip is a command-line package management system used to install and manage Python library packages. Its project page can be found at Github, `https://github.com/pypa/pip`.

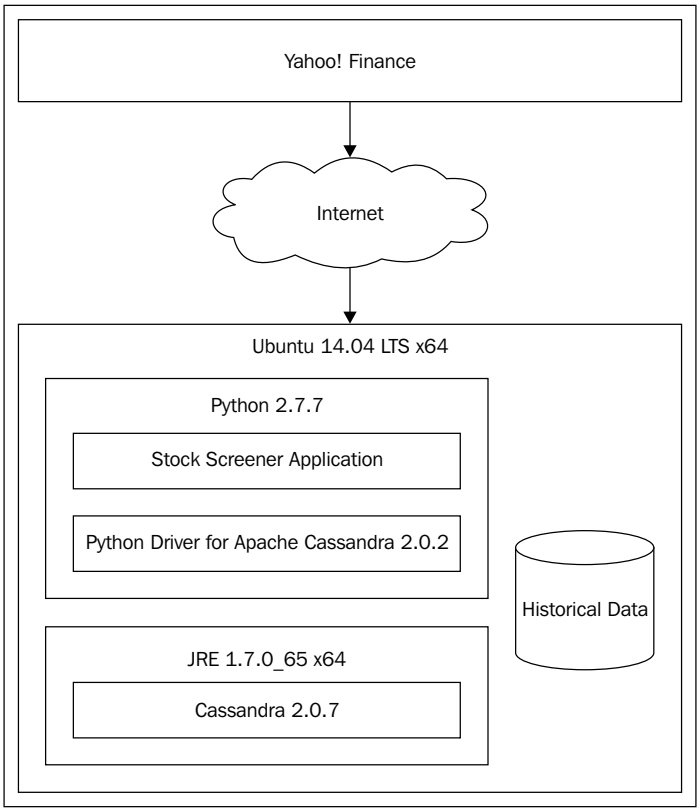
The integrated development environment

Spyder is an open source, cross-platform **integrated development environment** (**IDE**), usually used for scientific programming in Python. It is automatically installed by Anaconda and integrates NumPy, SciPy, matplotlib, IPython, and other open source software. It is also my favorite Python development environment.

There are many other good and popular Python IDEs, such as IPython and Eclipse. The code in this book is friendly to these IDEs.

The system overview

Alright, we have gone through the major system components of the Stock Screener Application and decided their implementation. The following figure depicts the system overview for the implementation of the application:



It is worth noting that the system will be developed on a single Ubuntu machine first and then on a single node Cassandra cluster (In *Chapter 7*, *Deployment and Monitoring*, we will expand the cluster to a two-node cluster). It serves as a limit to the superb clustering capabilities of Cassandra. However, from the software development perspective, the most important thing is to completely realize the required functionalities rather than splitting the significant efforts on the system or infrastructure components, which are of second priority.

Code design and development

We are now entering the development stage. I will walk you through the coding of the application building blocks step-by-step. Logically, two core modules will be built, namely, Data Feed Provider and Stock Screener. First, we will build the Data Feed Provider.

Data Feed Provider

The Data Feed Provider achieves the following three tasks:

1. Collecting the historical stock quote data from Yahoo! Finance.
2. Transforming the received data into a standardized format.
3. Saving the standardized data into the Cassandra database.

Python has a well-known data analysis library called pandas. It is an open source library providing high-performance, easy-to-use data structures, and data analysis tools, especially, for time-series type of data. You can go to `http://pandas.pydata.org/` for more details.

Collecting stock quote

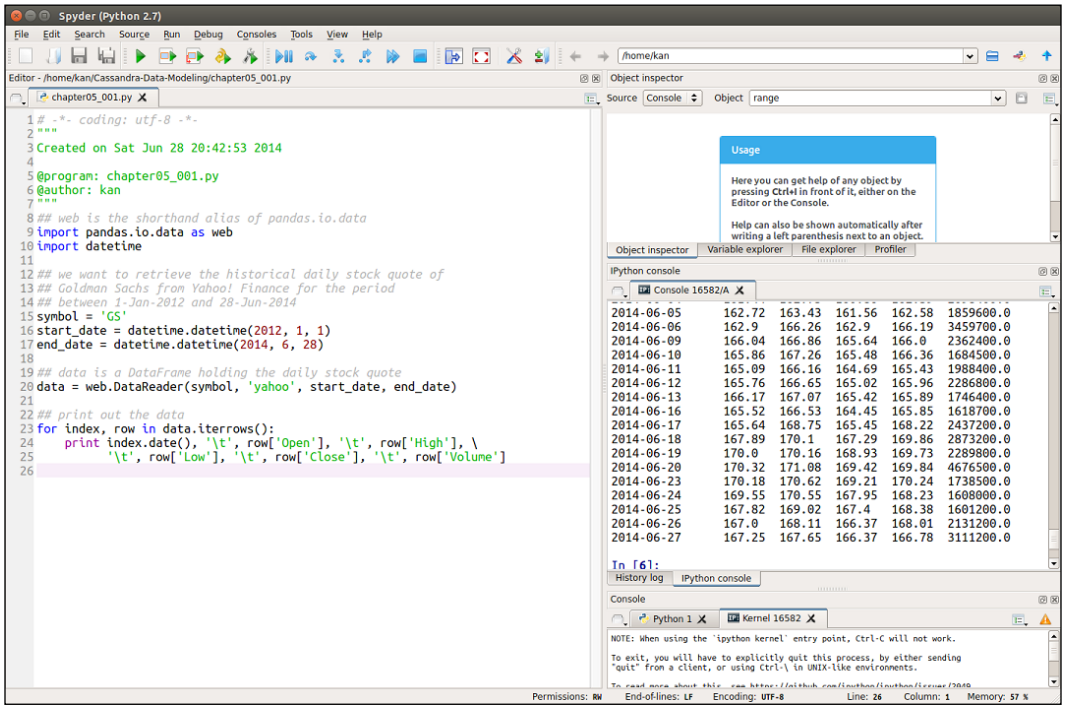
pandas offers a `DataReader` function in its `pandas.io.data` package. `DataReader` extracts financial data from various Internet sources into a data structure known as `DataFrame`. Yahoo! Finance is one of the supported Internet sources, making the collection of the historical stock quote data a piece of cake. Refer to the following Python code, `chapter05_001.py`:

```
# -*- coding: utf-8 -*-
# program: chapter05_001.py

## web is the shorthand alias of pandas.io.data
import pandas.io.data as web
import datetime

## we want to retrieve the historical daily stock quote of
## Goldman Sachs from Yahoo! Finance for the period
## between 1-Jan-2012 and 28-Jun-2014
symbol = 'GS'
start_date = datetime.datetime(2012, 1, 1)
end_date = datetime.datetime(2014, 6, 28)

## data is a DataFrame holding the daily stock quote
```

```
data = web.DataReader(symbol, 'yahoo', start_date, end_date)

## use a for-loop to print out the data
for index, row in data.iterrows():
    print index.date(), '\t', row['Open'], '\t', row['High'], \
          '\t', row['Low'], '\t', row['Close'], '\t',
row['Volume']
```

A brief explanation is required. pandas offers a very handy data structure called `DataFrame`, which is a two-dimensional labeled data structure with columns of potentially different types. You can think of it as a spreadsheet or SQL table. It is generally the most commonly used pandas object.

The following is a screenshot demonstrating the use of Spyder to write and test `chapter05_001.py` code:



The left-hand side of the Spyder IDE is the place where you write Python code. The middle panel on the right-hand side is the **IPython console** that runs the code.

Transforming data

Along with the data in the `DataFrame`, you can optionally pass index (row labels) and columns (column labels). The row and column labels can be accessed respectively, by accessing the index and columns attributes. For example, you can revisit the screenshot of `table.csv` and see that the column names returned by Yahoo! Finance are **Date**, **Open**, **High**, **Low**, **Close**, **Volume**, and **Adj Close**, respectively. `DataReader` uses **Date** as the index of the returned `DataFrame`. The remaining column names become the column labels of the `DataFrame`.

The last for-loop in `chapter05_001.py` is also worth some remarks. `DataFrame` has a function, `iterrows()`, for iterating over its rows as (index, columns) pairs. Therefore, the for-loop uses `iterrows()` to iterate the daily stock quotes and we simply print out the index (that is converted to a string by the `date()` function), and the **Open**, **High**, **Low**, **Close**, **Volume** columns by passing the respective column labels to the row. **Adj Close** is a close price with adjustments of stock split, merge, and dividend. We do not use this, as we want to focus on pure prices.

Please be aware that stock quote data from the different sources might have different formats and, needless to say, different column names. Therefore, we need to take care of such a subtle difference, when mapping them to our standardized data model. `DataFrame` provides a very handy way to retrieve the data by column names and a few useful functions to manipulate the index and columns. We can make use of them to standardize the data format, as shown in `chapter05_002.py`:

```
# -*- coding: utf-8 -*-
# program: chapter05_002.py

## web is the shorthand alias of pandas.io.data
import pandas.io.data as web
import datetime

## we want to retrieve the historical daily stock quote of
## Goldman Sachs from Yahoo! Finance for the period
## between 1-Jan-2012 and 28-Jun-2014
symbol = 'GS'
start_date = datetime.datetime(2012, 1, 1)
end_date = datetime.datetime(2014, 6, 28)

## data is a DataFrame holding the daily stock quote
data = web.DataReader(symbol, 'yahoo', start_date, end_date)

## standardize the column names
```

```
## rename index column to price_date to match the Cassandra table
data.index.names=['price_date']

## drop extra column 'Adj Close'
data = data.drop(['Adj Close'], axis=1)

## rename the columns to match the respective columns in Cassandra
data = data.rename(columns={'Open':'open_price', \
                            'High':'high_price', \
                            'Low':'low_price', \
                            'Close':'close_price', \
                            'Volume':'volume'})

## use a for-loop to print out the transformed data
for index, row in data.iterrows():
    print index.date(), '\t', row['open_price'], '\t', \
                              row['high_price'], '\t', \
                              row['low_price'], '\t', \
                              row['close_price'], '\t', \
                              row['volume']
```

Storing data in Cassandra

Before storing the retrieved data in Cassandra, we need to create the keyspace and table in the Cassandra database. We will create a keyspace called `packtcdma` and a table called `quote` in `chapter05_003.py` to hold the Historical Data, as shown in the following code:

```
# -*- coding: utf-8 -*-
# program: chapter05_003.py

## import Cassandra driver library
from cassandra.cluster import Cluster

## create Cassandra instance
cluster = Cluster()

## establish Cassandra connection, using local default
session = cluster.connect()

## create keyspace packtcdma if not exists
## currently it runs on a single-node cluster
session.execute("CREATE KEYSPACE IF NOT EXISTS packtcdma " + \
                "WITH replication" + \
```

```
                "={'class':'SimpleStrategy', " + \
                "'replication_factor':1}")

## use packtcdma keyspace
session.set_keyspace('packtcdma')

## execute CQL statement to create quote table if not exists
session.execute('CREATE TABLE IF NOT EXISTS quote (' + \
                'symbol varchar,' + \
                'price_time timestamp,' + \
                'open_price float,' + \
                'high_price float,' + \
                'low_price float,' + \
                'close_price float,' + \
                'volume double,' + \
                'PRIMARY KEY (symbol, price_time))')

## close Cassandra connection
cluster.shutdown()
```

The comments of the code are sufficient to explain what it is doing. Now, we have the Historical Data repository ready and what follows is to store the received data into it. This is exactly the purpose of `chapter05_004.py` in which a Python function is created to insert the data, as shown in the following code:

```
# -*- coding: utf-8 -*-
# program: chapter05_004.py

## import Cassandra driver library
from cassandra.cluster import Cluster
from decimal import Decimal

## function to insert historical data into table quote
## ss: Cassandra session
## sym: stock symbol
## d: standardized DataFrame containing historical data
def insert_quote(ss, sym, d):
    ## CQL to insert data, ? is the placeholder for parameters
    insert_cql = 'INSERT INTO quote (' + \
                 'symbol, price_time, open_price, high_price,' + \
                 'low_price, close_price, volume' + \
                 ') VALUES (' + \
                 '?, ?, ?, ?, ?, ?, ?' + \
                 ')'
    ## prepare the insert CQL as it will run repeatedly
```

```
insert_stmt = ss.prepare(insert_cql)

## set decimal places to 4 digits
getcontext().prec = 4

## loop thru the DataFrame and insert records
for index, row in d.iterrows():
    ss.execute(insert_stmt, \
                [sym, index, \
                Decimal(row['open_price']), \
                Decimal(row['high_price']), \
                Decimal(row['low_price']), \
                Decimal(row['close_price']), \
                Decimal(row['volume']) \
                ])
```

Although `chapter05_004.py` contains less than ten lines of code, it is rather complicated and needs some explanation.

We can create a function in Python using the keyword `def`. This must be followed by the function name and the parenthesized list of formal parameters. The code that form the body of the function starts in the next line, indented by a tab. Thus, in `chapter05_004.py`, the function name is `insert_quote()` with three parameters, namely, `ss`, `sym`, and `d`.

Indentation in Python

In Python, leading whitespace (spaces and tabs) at the beginning of a logical line is used to compute the indentation level of the line, which in turn is used to determine the grouping of statements. Be very careful of this. Most of the Python IDE has features to check against the indentations. The article on the myths about indentation of Python is worth reading, which is available at `http://www.secnetix.de/olli/Python/block_indentation.hawk`.

The second interesting thing is the `prepare()` function. It is used to prepare CQL statements that are parsed by Cassandra and then saved for later use. When the driver uses a prepared statement, it only needs to send the values of parameters to bind. This lowers network traffic and CPU utilization as a result of the avoidance of re-parsing the statement each time.

The placeholders for prepared statements are `?` characters so that the parameters are passed in sequence. This method is called positional parameter passing.

The last segment of code is a for-loop that iterates through the `DataFrame` and inserts each row into the quote table. We also use the `Decimal()` function to cast the string into numeric value.

Putting them all together

All pieces of Python code can be combined to make the Data Feed Provider. To make the code cleaner, the code fragment for the collection of stock quote is encapsulated in a function called `collect_data()` and that for data transformation in `transform_yahoo()` function. The complete program, `chapter05_005.py`, is listed as follows:.

```
# -*- coding: utf-8 -*-
# program: chapter05_005.py

## import Cassandra driver library
from cassandra.cluster import Cluster
from decimal import Decimal

## web is the shorthand alias of pandas.io.data
import pandas.io.data as web
import datetime

## function to insert historical data into table quote
## ss: Cassandra session
## sym: stock symbol
## d: standardized DataFrame containing historical data
def insert_quote(ss, sym, d):
    ## CQL to insert data, ? is the placeholder for parameters
    insert_cql = "INSERT INTO quote (" + \
                 "symbol, price_time, open_price, high_price," + \
                 "low_price, close_price, volume" + \
                 ") VALUES (" + \
                 "?, ?, ?, ?, ?, ?, ?" + \
                 ")"
    ## prepare the insert CQL as it will run repeatedly
    insert_stmt = ss.prepare(insert_cql)

    ## set decimal places to 4 digits
    getcontext().prec = 4

    ## loop thru the DataFrame and insert records
    for index, row in d.iterrows():
        ss.execute(insert_stmt, \
                   [sym, index, \
```

```
                    Decimal(row['open_price']), \
                    Decimal(row['high_price']), \
                    Decimal(row['low_price']), \
                    Decimal(row['close_price']), \
                    Decimal(row['volume']) \
                    ])

## retrieve the historical daily stock quote from Yahoo! Finance
## Parameters
## sym: stock symbol
## sd: start date
## ed: end date
def collect_data(sym, sd, ed):
    ## data is a DataFrame holding the daily stock quote
    data = web.DataReader(sym, 'yahoo', sd, ed)
    return data

## transform received data into standardized format
## Parameter
## d: DataFrame containing Yahoo! Finance stock quote
def transform_yahoo(d):
    ## drop extra column 'Adj Close'
    d1 = d.drop(['Adj Close'], axis=1)

    ## standardize the column names
    ## rename index column to price_date
    d1.index.names=['price_date']

    ## rename the columns to match the respective columns
    d1 = d1.rename(columns={'Open':'open_price', \
                            'High':'high_price', \
                            'Low':'low_price', \
                            'Close':'close_price', \
                            'Volume':'volume'})
    return d1

## create Cassandra instance
cluster = Cluster()

## establish Cassandra connection, using local default
session = cluster.connect('packtcdma')

symbol = 'GS'
start_date = datetime.datetime(2012, 1, 1)
```

```
end_date = datetime.datetime(2014, 6, 28)

## collect data
data = collect_data(symbol, start_date, end_date)

## transform Yahoo! Finance data
data = transform_yahoo(data)

## insert historical data
insert_quote(session, symbol, data)

## close Cassandra connection
cluster.shutdown()
```

Stock Screener

The Stock Screener retrieves historical data from the Cassandra database and applies technical analysis techniques to produce alerts. It has four components:

1. Retrieve historical data over a specified period
2. Program a technical analysis indicator for time-series data
3. Apply the screening rule to the historical data
4. Produce alert signals

Data Scoper

To utilize technical analysis techniques, a sufficient optimal number of stock quote data is required for calculation. We do not need to use all the stored data, and therefore a subset of data should be retrieved for processing. The following code, `chapte05_006.py`, retrieves the historical data from the table quote within a specified period:

```
# -*- coding: utf-8 -*-
# program: chapter05_006.py

import pandas as pd
import numpy as np

## function to insert historical data into table quote
## ss: Cassandra session
## sym: stock symbol
## sd: start date
## ed: end date
```

```
## return a DataFrame of stock quote
def retrieve_data(ss, sym, sd, ed):
    ## CQL to select data, ? is the placeholder for parameters
    select_cql = "SELECT * FROM quote WHERE symbol=? " + \
                 "AND price_time >= ? AND price_time <= ?"

    ## prepare select CQL
    select_stmt = ss.prepare(select_cql)

    ## execute the select CQL
    result = ss.execute(select_stmt, [sym, sd, ed])

    ## initialize an index array
    idx = np.asarray([])

    ## initialize an array for columns
    cols = np.asarray([])

    ## loop thru the query resultset to make up the DataFrame
    for r in result:
        idx = np.append(idx, [r.price_time])
        cols = np.append(cols, [r.open_price, r.high_price, \
                        r.low_price, r.close_price, r.volume])

    ## reshape the 1-D array into a 2-D array for each day
    cols = cols.reshape(idx.shape[0], 5)

    ## convert the arrays into a pandas DataFrame
    df = pd.DataFrame(cols, index=idx, \
                      columns=['close_price', 'high_price', \
                      'low_price', 'close_price', 'volume'])
    return df
```

The first portion of the function should be easy to understand. It executes a `select_cql` query for a particular stock symbol over a specified date period. The clustering column, `price_time`, makes range query possible here. The query result set is returned and used to fill two NumPy arrays, `idx` for index, and `cols` for columns. The `cols` array is then reshaped as a two-dimensional array with rows of prices and volume for each day. Finally, both `idx` and `cols` arrays are used to create a `DataFrame` to return `df`.

Time-series data

As a simple illustration, we use a 10-day **Simple Moving Average** (**SMA**) as the technical analysis signal for stock screening. pandas provides a rich set of functions to work with time-series data. The SMA can be easily computed by the `rolling_mean()` function, as shown in `chapter05_007.py`:

```
# -*- coding: utf-8 -*-
# program: chapter05_007.py

import pandas as pd

## function to compute a Simple Moving Average on a DataFrame
## d: DataFrame
## prd: period of SMA
## return a DataFrame with an additional column of SMA
def sma(d, prd):
    d['sma'] = pd.rolling_mean(d.close_price, prd)
    return d
```

The screening rule

When SMA is calculated, we can apply a screening rule in order to look for trading signals. A very simple rule is adopted: a buy-and-hold signal is generated whenever a trading day whose close price is higher than 10-day SMA. In Python, it is just a one liner by virtue of pandas power. Amazing! Here is an example:

```
# -*- coding: utf-8 -*-
# program: chapter05_008.py

## function to apply screening rule to generate buy signals
## screening rule, Close > 10-Day SMA
## d: DataFrame
## return a DataFrame containing buy signals
def signal_close_higher_than_sma10(d):
    return d[d.close_price > d.sma]
```

The Stock Screener engine

Until now, we coded the components of the Stock Screener. We now combine them together to generate the Alert List, as shown in the following code:

```
# -*- coding: utf-8 -*-
# program: chapter05_009.py

## import Cassandra driver library
```

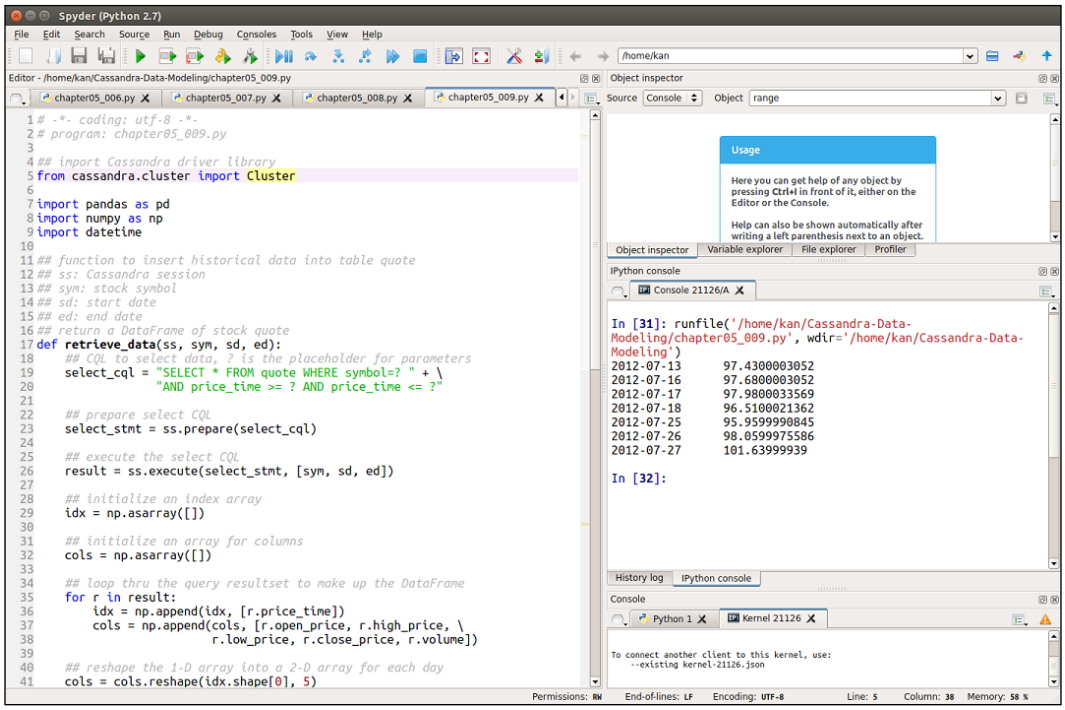
```
from cassandra.cluster import Cluster

import pandas as pd
import numpy as np
import datetime

## function to insert historical data into table quote
## ss: Cassandra session
## sym: stock symbol
## sd: start date
## ed: end date
## return a DataFrame of stock quote
def retrieve_data(ss, sym, sd, ed):
    ## CQL to select data, ? is the placeholder for parameters
    select_cql = "SELECT * FROM quote WHERE symbol=? " + \
                 "AND price_time >= ? AND price_time <= ?"

    ## prepare select CQL
    select_stmt = ss.prepare(select_cql)

    ## execute the select CQL
    result = ss.execute(select_stmt, [sym, sd, ed])

    ## initialize an index array
    idx = np.asarray([])

    ## initialize an array for columns
    cols = np.asarray([])

    ## loop thru the query resultset to make up the DataFrame
    for r in result:
        idx = np.append(idx, [r.price_time])
        cols = np.append(cols, [r.open_price, r.high_price, \
                        r.low_price, r.close_price, r.volume])

    ## reshape the 1-D array into a 2-D array for each day
    cols = cols.reshape(idx.shape[0], 5)

    ## convert the arrays into a pandas DataFrame
    df = pd.DataFrame(cols, index=idx, \
                      columns=['open_price', 'high_price', \
                      'low_price', 'close_price', 'volume'])
```

```
    return df

## function to compute a Simple Moving Average on a DataFrame
## d: DataFrame
## prd: period of SMA
## return a DataFrame with an additional column of SMA
def sma(d, prd):
    d['sma'] = pd.rolling_mean(d.close_price, prd)
    return d

## function to apply screening rule to generate buy signals
## screening rule, Close > 10-Day SMA
## d: DataFrame
## return a DataFrame containing buy signals
def signal_close_higher_than_sma10(d):
    return d[d.close_price > d.sma]

## create Cassandra instance
cluster = Cluster()

## establish Cassandra connection, using local default
session = cluster.connect('packtcdma')
## scan buy-and-hold signals for GS over 1 month since 28-Jun-2012
symbol = 'GS'
start_date = datetime.datetime(2012, 6, 28)
end_date = datetime.datetime(2012, 7, 28)

## retrieve data
data = retrieve_data(session, symbol, start_date, end_date)

## close Cassandra connection
cluster.shutdown()

## compute 10-Day SMA
data = sma(data, 10)

## generate the buy-and-hold signals
alerts = signal_close_higher_than_sma10(data)

## print out the alert list
for index, r in alerts.iterrows():
    print index.date(), '\t', r['close_price']
```

Test run

An End-to-End Test consists of two parts. First, we test and verify `chapter05_005.py`, which is the complete Data Feed Provider module. Then run `chapter05_005.py` in Spyder. Historical stock quote data should be stored in the Cassandra database. Then run and verify the Stock Screener module, `chapter05_009.py`, also in Spyder.

A sample screen of the test run is shown in the following screenshot. The Alert List should have seven buy-and-hold trading signals:



Summary

This chapter was rather jam-packed. We designed a simple stock screening application that collects stock quote data from Yahoo! Finance, which uses Cassandra as its repository. The system environment of the application was also introduced with brief setup instructions. Then we developed the application in Python with a step-by-step explanation. Despite of using one Cassandra table, the basic row manipulation logic has been demonstrated.

In the next chapter, we will continue enhancing the Stock Screener Application to collect stock quote data of a bunch of stocks and optimize the application with several refinements.

6
Enhancing a Version

Traditionally, changes are usually not welcomed and are avoided as much as possible by a relational database developer. However, business changes every day, especially in the present fast-paced era. The delayed response to business changes of a system using a relational database deteriorates the agility and even threatens the survival of the enterprise. With the advancement of NoSQL and other related technologies, we now have alternatives to embrace such business changes.

By continuing with the enhancements of the Stock Screener Application developed in *Chapter 5*, *First-cut Design and Implementation*, the techniques of how to evolve an existing Cassandra data model will be explained in detail. Meanwhile, the techniques of modeling by query will be demonstrated as well. The source code of the Stock Screener Application will then be modified accordingly. By the end of this chapter, a complete technical analysis application on stocks will be developed. You can use it as a foundation to quickly develop your own.

Evolving the data model

The Stock Screener Application created in *Chapter 5*, *First-cut Design and Implementation*, is good enough to retrieve and analyze a single stock at one time. However, scanning just a single stock looks very limited in practical use. A slight improvement can be made here; it can handle a bunch of stocks instead of one. This bunch of stocks will be stored as Watch List in the Cassandra database.

Accordingly, the Stock Screener Application will be modified to analyze the stocks in the Watch List, and therefore it will produce alerts for each of the stocks being watched based on the same screening rule.

For the produced alerts, saving them in Cassandra will be beneficial for backtesting trading strategies and continuous improvement of the Stock Screener Application. They can be reviewed from time to time without having to review them on the fly.

Backtesting is a jargon used to refer to testing a trading strategy, investment strategy, or a predictive model using existing historical data. It is also a special type of cross-validation applied to time series data.

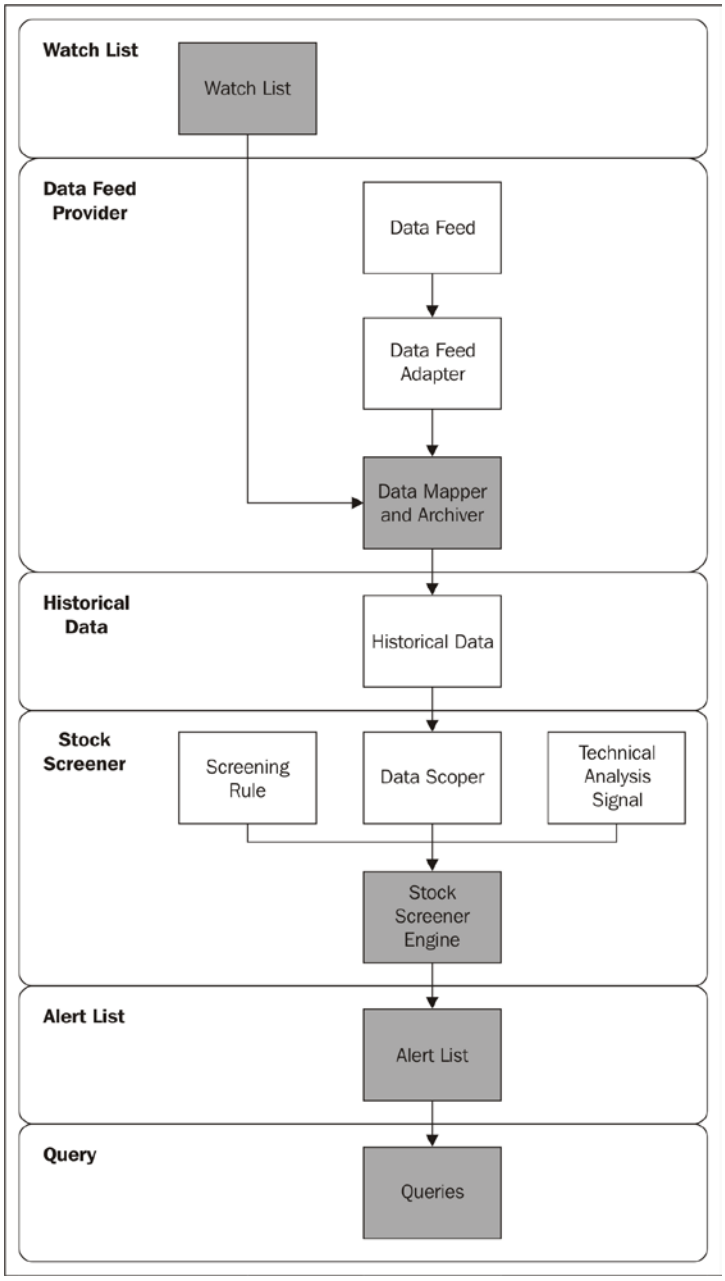
In addition, when the number of the stocks in the Watch List grows to a few hundred, it will be difficult for a user of the Stock Screener Application to recall what the stocks are by simply referring to their stock codes. Hence, it would be nice to have the name of the stocks added to the produced alerts to make them more descriptive and user-friendly.

Finally, we might have an interest in finding out how many alerts were generated on a particular stock over a specified period of time and how many alerts were generated on a particular date. We will use CQL to write queries to answer these two questions. By doing so, the modeling by query technique can be demonstrated.

The enhancement approach

The enhancement approach consists of four change requests in total. First, we will conduct changes in the data model and then the code will be enhanced to provide the new features. Afterwards, we will test run the enhanced Stock Screener Application again. The parts of the Stock Screener Application that require modifications are highlighted in the following figure.

It is remarkable that two new components are added to the Stock Screener Application. The first component, **Watch List**, governs **Data Mapper and Archiver** to collect stock quote data of those stocks in the Watch List from Yahoo! Finance. The second component is **Query**. It provides two Queries on **Alert List** for backtesting purposes:



Watch List

Watch List is a very simple table that merely stores the stock code of its constituents. It is rather intuitive for a relational database developer to define the stock code as the primary key, isn't it? Nevertheless, remember that in Cassandra, the primary key is used to determine the node that stores the row. As Watch List is expected to not be a very long list, it would be more appropriate to put all of its rows on the same node for faster retrieval. But how can we do that?

We can create an additional column, say `watch_list_code`, for this particular purpose. The new table is called `watchlist` and will be created in the `packtcdma` keyspace. The CQL statement is shown in `chapter06_001.py`:

```
# -*- coding: utf-8 -*-
# program: chapter06_001.py

## import Cassandra driver library
from cassandra.cluster import Cluster

## function to create watchlist
def create_watchlist(ss):
    ## create watchlist table if not exists
    ss.execute('CREATE TABLE IF NOT EXISTS watchlist (' + \
               'watch_list_code varchar,' + \
               'symbol varchar,' + \
               'PRIMARY KEY (watch_list_code, symbol))')

    ## insert AAPL, AMZN, and GS into watchlist
    ss.execute("INSERT INTO watchlist (watch_list_code, " + \
               "symbol) VALUES ('WS01', 'AAPL')")
    ss.execute("INSERT INTO watchlist (watch_list_code, " + \
               "symbol) VALUES ('WS01', 'AMZN')")
    ss.execute("INSERT INTO watchlist (watch_list_code, " + \
               "symbol) VALUES ('WS01', 'GS')")

## create Cassandra instance
cluster = Cluster()

## establish Cassandra connection, using local default
session = cluster.connect()

## use packtcdma keyspace
session.set_keyspace('packtcdma')

## create watchlist table
```

```
create_watchlist(session)

## close Cassandra connection
cluster.shutdown()
```

The `create_watchlist` function creates the table. Note that the `watchlist` table has a compound primary key made of `watch_list_code` and `symbol`. A Watch List called `WS01` is also created, which contains three stocks, `AAPL`, `AMZN`, and `GS`.

Alert List

In *Chapter 5*, *First-cut Design and Implementation*, **Alert List** is very rudimentary. It is produced by a Python program and enumerates the date when the close price was above its 10-day SMA, that is, the signal and the close price at that time. Note that there were no stock code and stock name.

We will create a table called `alertlist` to store the alerts with the code and name of the stock. The inclusion of the stock name is to meet the requirement of making the Stock Screener Application more user-friendly. Also, remember that joins are not allowed and denormalization is really the best practice in Cassandra. This means that we do not mind repeatedly storing (duplicating) the stock name in the tables that will be queried. A rule of thumb is *one table for one query*; as simple as that.

The `alertlist` table is created by the CQL statement, as shown in `chapter06_002.py`:

```
# -*- coding: utf-8 -*-
# program: chapter06_002.py

## import Cassandra driver library
from cassandra.cluster import Cluster

## function to create alertlist
def create_alertlist(ss):
    ## execute CQL statement to create alertlist table if not exists
    ss.execute('CREATE TABLE IF NOT EXISTS alertlist (' + \
               'symbol varchar,' + \
               'price_time timestamp,' + \
               'stock_name varchar,' + \
               'signal_price float,' + \
               'PRIMARY KEY (symbol, price_time))')

## create Cassandra instance
cluster = Cluster()

## establish Cassandra connection, using local default
```

```
session = cluster.connect()

## use packtcdma keyspace
session.set_keyspace('packtcdma')

## create alertlist table
create_alertlist(session)

## close Cassandra connection
cluster.shutdown()
```

The primary key is also a compound primary key that consists of `symbol` and `price_time`.

Adding the descriptive stock name

Until now, the `packtcdma` keyspace has three tables, which are `alertlist`, `quote`, and `watchlist`. To add the descriptive stock name, one can think of only adding a column of stock name to `alertlist` only. As seen in the previous section, this has been done. So, do we need to add a column for `quote` and `watchlist`?

It is, in fact, a design decision that depends on whether these two tables will be serving user queries. What a user query means is that the table will be used to retrieve rows for a query raised by a user. If a user wants to know the close price of Apple Inc. on June 30, 2014, it is a user query. On the other hand, if the Stock Screener Application uses a query to retrieve rows for its internal processing, it is not a user query. Therefore, if we want `quote` and `watchlist` to return rows for user queries, they need the stock name column; otherwise, they do not need it.

The `watchlist` table is only for internal use by the current design, and so it need not have the stock name column. Of course, if in future, the Stock Screener Application allows a user to maintain Watch List, the stock name should also be added to the `watchlist` table.

However, for `quote`, it is a bit tricky. As the stock name should be retrieved from the Data Feed Provider, which is Yahoo! Finance in our case, the most suitable time to get it is when the corresponding stock quote data is retrieved. Hence, a new column called `stock_name` is added to `quote`, as shown in `chapter06_003.py`:

```
# -*- coding: utf-8 -*-
# program: chapter06_003.py

## import Cassandra driver library
```

```
from cassandra.cluster import Cluster

## function to add stock_name column
def add_stockname_to_quote(ss):
    ## add stock_name to quote
    ss.execute('ALTER TABLE quote ' + \
               'ADD stock_name varchar')

## create Cassandra instance
cluster = Cluster()

## establish Cassandra connection, using local default
session = cluster.connect()

## use packtcdma keyspace
session.set_keyspace('packtcdma')

## add stock_name column
add_stockname_to_quote(session)

## close Cassandra connection
cluster.shutdown()
```

It is quite self-explanatory. Here, we use the `ALTER TABLE` statement to add the `stock_name` column of the `varchar` data type to `quote`.

Queries on alerts

As mentioned previously, we are interested in two questions:

- How many alerts were generated on a stock over a specified period of time?
- How many alerts were generated on a particular date?

For the first question, `alertlist` is sufficient to provide an answer. However, `alertlist` cannot answer the second question because its primary key is composed of `symbol` and `price_time`. We need to create another table specifically for that question. This is an example of modeling by query.

Basically, the structure of the new table for the second question should resemble the structure of `alertlist`. We give that table a name, `alert_by_date`, and create it as shown in `chapter06_004.py`:

```
# -*- coding: utf-8 -*-
# program: chapter06_004.py

## import Cassandra driver library
from cassandra.cluster import Cluster

## function to create alert_by_date table
def create_alertbydate(ss):
    ## create alert_by_date table if not exists
    ss.execute('CREATE TABLE IF NOT EXISTS alert_by_date (' + \
               'symbol varchar,' + \
               'price_time timestamp,' + \
               'stock_name varchar,' + \
               'signal_price float,' + \
               'PRIMARY KEY (price_time, symbol))')

## create Cassandra instance
cluster = Cluster()

## establish Cassandra connection, using local default
session = cluster.connect()

## use packtcdma keyspace
session.set_keyspace('packtcdma')

## create alert_by_date table
create_alertbydate(session)

## close Cassandra connection
cluster.shutdown()
```

When compared to `alertlist` in `chapter06_002.py`, `alert_by_date` only swaps the order of the columns in the compound primary key. One might think that a secondary index can be created on `alertlist` to achieve the same effect. Nonetheless, in Cassandra, a secondary index cannot be created on columns that are already engaged in the primary key. Always be aware of this constraint.

We now finish the modifications on the data model. It is time for us to enhance the application logic in the next section.

Enhancing the code

Regarding the new requirements to be incorporated into the Stock Screener Application, Watch List is created, and we will continue to implement the code for the remaining changes in this section.

Data Mapper and Archiver

Data Mapper and Archiver are components of the Data Feed Provider module, and its source code file is `chapter05_005.py`. Most of the source code can be left intact; we only need to add code to:

1. Load Watch List for a Watch List code and retrieve data feed based on that
2. Retrieve the name of the stocks and store it in the quote table

The modified source code is shown in `chapter06_005.py`:

```
# -*- coding: utf-8 -*-
# program: chapter06_005.py

## import Cassandra driver library
from cassandra.cluster import Cluster
from decimal import *

## web is the shorthand alias of pandas.io.data
import pandas.io.data as web
import datetime

## import BeautifulSoup and requests
from bs4 import BeautifulSoup
import requests

## function to insert historical data into table quote
## ss: Cassandra session
## sym: stock symbol
## d: standardized DataFrame containing historical data
## sn: stock name
def insert_quote(ss, sym, d, sn):
    ## CQL to insert data, ? is the placeholder for parameters
    insert_cql = "INSERT INTO quote (" + \
                 "symbol, price_time, open_price, high_price," + \
                 "low_price, close_price, volume, stock_name" + \
                 ") VALUES (?, ?, ?, ?, ?, ?, ?, ?)"
    ## prepare the insert CQL as it will run repeatedly
```

```
        insert_stmt = ss.prepare(insert_cql)

        ## set decimal places to 4 digits
        getcontext().prec = 4

        ## loop thru the DataFrame and insert records
        for index, row in d.iterrows():
            ss.execute(insert_stmt, \
                       [sym, index, \
                       Decimal(row['open_price']), \
                       Decimal(row['high_price']), \
                       Decimal(row['low_price']), \
                       Decimal(row['close_price']), \
                       Decimal(row['volume']), \
                       sn])
```

Here, we changed the `INSERT` statement to store the stock name into `quote` in the `insert_quote` function. We then add a function called `load_watchlist`:

```
    ## retrieve the historical daily stock quote from Yahoo! Finance
    ## Parameters
    ## sym: stock symbol
    ## sd: start date
    ## ed: end date
    def collect_data(sym, sd, ed):
        ## data is a DataFrame holding the daily stock quote
        data = web.DataReader(sym, 'yahoo', sd, ed)
        return data

    ## transform received data into standardized format
    ## Parameter
    ## d: DataFrame containing Yahoo! Finance stock quote
    def transform_yahoo(d):
        ## drop extra column 'Adj Close'
        d1 = d.drop(['Adj Close'], axis=1)

        ## standardize the column names
        ## rename index column to price_date
        d1.index.names=['price_date']

        ## rename the columns to match the respective columns
        d1 = d1.rename(columns={'Open':'open_price', \
                                'High':'high_price', \
                                'Low':'low_price', \
                                'Close':'close_price', \
```

```
                                'Volume':'volume'})
    return d1

## function to retrieve watchlist
## ss: Cassandra session
## ws: watchlist code
def load_watchlist(ss, ws):
    ## CQL to select data, ? is the placeholder for parameters
    select_cql = "SELECT symbol FROM watchlist " + \
                 "WHERE watch_list_code=?"

    ## prepare select CQL
    select_stmt = ss.prepare(select_cql)

    ## execute the select CQL
    result = ss.execute(select_stmt, [ws])

    ## initialize the stock array
    stw = []

    ## loop thru the query resultset to make up the DataFrame
    for r in result:
        stw.append(r.symbol)

    return stw
```

Here, the new function, `load_watchlist`, submits a `SELECT` query on `watch_list` to retrieve the stocks to be watched of a particular Watch List code; it then returns a list of `symbol`:

```
## function to retrieve stock name from Yahoo!Finance
## sym: stock symbol
def get_stock_name(sym):
  url = 'http://finance.yahoo.com/q/hp?s=' + sym + \
  '+Historical+Prices'
  r = requests.get(url)
  soup = BeautifulSoup(r.text)
  data = soup.findAll('h2')
  return data[2].text

def testcase001():
    ## create Cassandra instance
    cluster = Cluster()

    ## establish Cassandra connection, using local default
```

```
        session = cluster.connect('packtcdma')

        start_date = datetime.datetime(2012, 1, 1)
        end_date = datetime.datetime(2014, 6, 28)

        ## load the watchlist
        stocks_watched = load_watchlist(session, "WS01")

        ## iterate the watchlist
        for symbol in stocks_watched:
            ## get stock name
            stock_name = get_stock_name(symbol)

            ## collect data
            data = collect_data(symbol, start_date, end_date)

            ## transform Yahoo! Finance data
            data = transform_yahoo(data)

            ## insert historical data
            insert_quote(session, symbol, data, stock_name)

        ## close Cassandra connection
        cluster.shutdown()

    testcase001()
```

The change here is a new function named `get_stock_name`, which sends a web service request to Yahoo! Finance and extracts the name of the stock from the returned HTML page. We use a Python package called `BeautifulSoup` to make the extraction of an element from a HTML page very convenient. The `get_stock_name` function then returns the stock name.

`BeautifulSoup` is a library designed for quick turnaround projects such as screen scraping. It primarily parses any text given to it and finds anything wanted through the tree traversal of the parsed text. More information can be found at `http://www.crummy.com/software/BeautifulSoup/`.

A `for` loop is used to iterate through the Watch List to retrieve the stock name and the stock quote data. In addition, as we need to store the stock name in the `quote` table, the `insert_quote` function accepts the stock name as a new parameter and requires a little modification on the `INSERT` statement and the `for` loop accordingly.

That is all about the changes on Data Mapper and Archiver.

Stock Screener Engine

We will use the source code of Stock Screener Engine in *Chapter 5, First-cut Design and Implementation* to include the enhancements; to do so, we will perform the following:

1. Similar to Data Mapper and Archiver, we will load Watch List for a Watch List code and scan for alerts on each stock.
2. Retrieve stock quote data with the stock name column from the quote table.
3. Save the alerts into `alertlist`.

The modified source code is shown in `chapter06_006.py`:

```
# -*- coding: utf-8 -*-
# program: chapter06_006.py

## import Cassandra driver library
from cassandra.cluster import Cluster

import pandas as pd
import numpy as np
import datetime

## import Cassandra BatchStatement library
from cassandra.query import BatchStatement
from decimal import *

## function to insert historical data into table quote
## ss: Cassandra session
## sym: stock symbol
## sd: start date
## ed: end date
## return a DataFrame of stock quote
def retrieve_data(ss, sym, sd, ed):
    ## CQL to select data, ? is the placeholder for parameters
    select_cql = "SELECT * FROM quote WHERE symbol=? " + \
```

```
                "AND price_time >= ? AND price_time <= ?"

    ## prepare select CQL
    select_stmt = ss.prepare(select_cql)

    ## execute the select CQL
    result = ss.execute(select_stmt, [sym, sd, ed])

    ## initialize an index array
    idx = np.asarray([])

    ## initialize an array for columns
    cols = np.asarray([])

    ## loop thru the query resultset to make up the DataFrame
    for r in result:
        idx = np.append(idx, [r.price_time])
        cols = np.append(cols, [r.open_price, r.high_price, \
                        r.low_price, r.close_price, \
                        r.volume, r.stock_name])

    ## reshape the 1-D array into a 2-D array for each day
    cols = cols.reshape(idx.shape[0], 6)

    ## convert the arrays into a pandas DataFrame
    df = pd.DataFrame(cols, index=idx, \
                      columns=['open_price', 'high_price', \
                      'low_price', 'close_price', \
                      'volume', 'stock_name'])
    return df
```

As we have included the stock name in the query resultset, we need to modify the `SELECT` statement in the `retrieve_data` function:

```
## function to compute a Simple Moving Average on a DataFrame
## d: DataFrame
## prd: period of SMA
## return a DataFrame with an additional column of SMA
def sma(d, prd):
    d['sma'] = pd.rolling_mean(d.close_price, prd)
    return d

## function to apply screening rule to generate buy signals
## screening rule, Close > 10-Day SMA
## d: DataFrame
```

```
## return a DataFrame containing buy signals
def signal_close_higher_than_sma10(d):
    return d[d.close_price > d.sma]

## function to retrieve watchlist
## ss: Cassandra session
## ws: watchlist code
def load_watchlist(ss, ws):
    ## CQL to select data, ? is the placeholder for parameters
    select_cql = "SELECT symbol FROM watchlist " + \
                 "WHERE watch_list_code=?"

    ## prepare select CQL
    select_stmt = ss.prepare(select_cql)

    ## execute the select CQL
    result = ss.execute(select_stmt, [ws])

    ## initialize the stock array
    stw = []

    ## loop thru the query resultset to make up the DataFrame
    for r in result:
        stw.append(r.symbol)

    return stw

## function to insert historical data into table quote
## ss: Cassandra session
## sym: stock symbol
## d: standardized DataFrame containing historical data
## sn: stock name
def insert_alert(ss, sym, sd, cp, sn):
    ## CQL to insert data, ? is the placeholder for parameters
    insert_cql1 = "INSERT INTO alertlist (" + \
                  "symbol, price_time, signal_price, stock_name" +\
                  ") VALUES (?, ?, ?, ?)"

    ## CQL to insert data, ? is the placeholder for parameters
    insert_cql2 = "INSERT INTO alert_by_date (" + \
                  "symbol, price_time, signal_price, stock_name" +\
                  ") VALUES (?, ?, ?, ?)"

    ## prepare the insert CQL as it will run repeatedly
```

```
        insert_stmt1 = ss.prepare(insert_cql1)
        insert_stmt2 = ss.prepare(insert_cql2)

        ## set decimal places to 4 digits
        getcontext().prec = 4

        ## begin a batch
        batch = BatchStatement()

        ## add insert statements into the batch
        batch.add(insert_stmt1, [sym, sd, cp, sn])
        batch.add(insert_stmt2, [sym, sd, cp, sn])

        ## execute the batch
        ss.execute(batch)

    def testcase002():
        ## create Cassandra instance
        cluster = Cluster()

        ## establish Cassandra connection, using local default
        session = cluster.connect('packtcdma')

        start_date = datetime.datetime(2012, 6, 28)
        end_date = datetime.datetime(2012, 7, 28)

        ## load the watch list
        stocks_watched = load_watchlist(session, "WS01")

        for symbol in stocks_watched:
            ## retrieve data
            data = retrieve_data(session, symbol, start_date, end_date)

            ## compute 10-Day SMA
            data = sma(data, 10)

            ## generate the buy-and-hold signals
            alerts = signal_close_higher_than_sma10(data)

            ## save the alert list
            for index, r in alerts.iterrows():
                insert_alert(session, symbol, index, \
                             Decimal(r['close_price']), \
```

```
                                r['stock_name'])

        ## close Cassandra connection
        cluster.shutdown()

testcase002()
```

At the bottom of `chapter06_006.py`, the `for` loop is responsible for iterating `watchlist` loaded by the new `load_watchlist` function, which is the same function as in `chapter06_005.py` and does not require further explanation. Another `for` loop inside saves the scanned alerts into `alertlist` by calling the new `insert_alert` function.

Before explaining the `insert_alert` function, let us jump to the `retrieve_data` function at the top. The `retrieve_data` function is modified to return the name of the stock as well and hence the `cols` variable now contains six columns. Scroll down a bit to `insert_alert`.

The `insert_alert` function, as its name suggests, saves the alert into `alertlist` and `alert_by_date`. It has two `INSERT` statements for these two tables, respectively. The `INSERT` statements are almost identical except for the name of the table. Obviously, they are repeated, and this is what denormalization means. We also apply a new feature of Cassandra 2.0 here, known as *batch*. A batch combines multiple **data modification language** (**DML**) statements into a single logical, atomic operation. The Cassandra Python driver from DataStax supports this feature by the `BatchStatement` package. We create a batch by calling the `BatchStatement()` function, then add the prepared `INSERT` statements into the batch, and finally execute it. If either `INSERT` statement comes across an error during commit, all DML statements in the batch will not be executed. Therefore, it is analogous to a transaction in a relational database.

Queries on Alerts

The last modification to the Stock Screener Application is the enquiry functions on alerts that are useful for backtesting and performance measurement. We write two queries to answer the two questions, which are as follows:

- How many alerts were generated on a stock over a specified period of time?
- How many alerts were generated on a particular date?

As we have used denormalization on the data model, it is very easy to execute. For the first query, see `chapter06_007.py`:

```
# -*- coding: utf-8 -*-
# program: chapter06_007.py

## import Cassandra driver library
from cassandra.cluster import Cluster

import pandas as pd
import numpy as np
import datetime

## execute CQL statement to retrieve rows of
## How many alerts were generated on a particular stock over
## a specified period of time?
def alert_over_daterange(ss, sym, sd, ed):
    ## CQL to select data, ? is the placeholder for parameters
    select_cql = "SELECT * FROM alertlist WHERE symbol=? " + \
                 "AND price_time >= ? AND price_time <= ?"

    ## prepare select CQL
    select_stmt = ss.prepare(select_cql)

    ## execute the select CQL
    result = ss.execute(select_stmt, [sym, sd, ed])

     ## initialize an index array
    idx = np.asarray([])

    ## initialize an array for columns
    cols = np.asarray([])

    ## loop thru the query resultset to make up the DataFrame
    for r in result:
        idx = np.append(idx, [r.price_time])
        cols = np.append(cols, [r.symbol, r.stock_name, \
                        r.signal_price])

    ## reshape the 1-D array into a 2-D array for each day
    cols = cols.reshape(idx.shape[0], 3)

    ## convert the arrays into a pandas DataFrame
    df = pd.DataFrame(cols, index=idx, \
```

```
                        columns=['symbol', 'stock_name', \
                        'signal_price'])
    return df

def testcase001():
    ## create Cassandra instance
    cluster = Cluster()

    ## establish Cassandra connection, using local default
    session = cluster.connect()

    ## use packtcdma keyspace
    session.set_keyspace('packtcdma')

    ## scan buy-and-hold signals for GS
    ## over 1 month since 28-Jun-2012
    symbol = 'GS'
    start_date = datetime.datetime(2012, 6, 28)
    end_date = datetime.datetime(2012, 7, 28)

    ## retrieve alerts
    alerts = alert_over_daterange(session, symbol, \
                                  start_date, end_date)

    for index, r in alerts.iterrows():
        print index.date(), '\t', \
            r['symbol'], '\t', \
            r['stock_name'], '\t', \
            r['signal_price']

    ## close Cassandra connection
    cluster.shutdown()

testcase001()
```

A function named `alert_over_daterange` is defined to retrieve the rows relevant to the first question. Then it transforms the CQL resultset to a pandas DataFrame.

Then we can come up with a query for the second question with reference to the same logic in `chapter06_007.py`. The source code is shown in `chapter06_008.py`:

```
# -*- coding: utf-8 -*-
# program: chapter06_008.py

## import Cassandra driver library
```

```
from cassandra.cluster import Cluster

import pandas as pd
import numpy as np
import datetime

## execute CQL statement to retrieve rows of
## How many alerts were generated on a particular stock over
## a specified period of time?
def alert_on_date(ss, dd):
    ## CQL to select data, ? is the placeholder for parameters
    select_cql = "SELECT * FROM alert_by_date WHERE " + \
                 "price_time=?"

    ## prepare select CQL
    select_stmt = ss.prepare(select_cql)

    ## execute the select CQL
    result = ss.execute(select_stmt, [dd])

     ## initialize an index array
    idx = np.asarray([])

    ## initialize an array for columns
    cols = np.asarray([])

    ## loop thru the query resultset to make up the DataFrame
    for r in result:
        idx = np.append(idx, [r.symbol])
        cols = np.append(cols, [r.stock_name, r.price_time, \
                        r.signal_price])

    ## reshape the 1-D array into a 2-D array for each day
    cols = cols.reshape(idx.shape[0], 3)

    ## convert the arrays into a pandas DataFrame
    df = pd.DataFrame(cols, index=idx, \
                      columns=['stock_name', 'price_time', \
                      'signal_price'])
    return df

def testcase001():
    ## create Cassandra instance
```

```
    cluster = Cluster()

    ## establish Cassandra connection, using local default
    session = cluster.connect()

    ## use packtcdma keyspace
    session.set_keyspace('packtcdma')

    ## scan buy-and-hold signals for GS
    over 1 month since 28-Jun-2012
    on_date = datetime.datetime(2012, 7, 13)

    ## retrieve alerts
    alerts = alert_on_date(session, on_date)

    ## print out alerts
    for index, r in alerts.iterrows():
        print index, '\t', \
              r['stock_name'], '\t', \
              r['signal_price']

    ## close Cassandra connection
    cluster.shutdown()

testcase001()
```

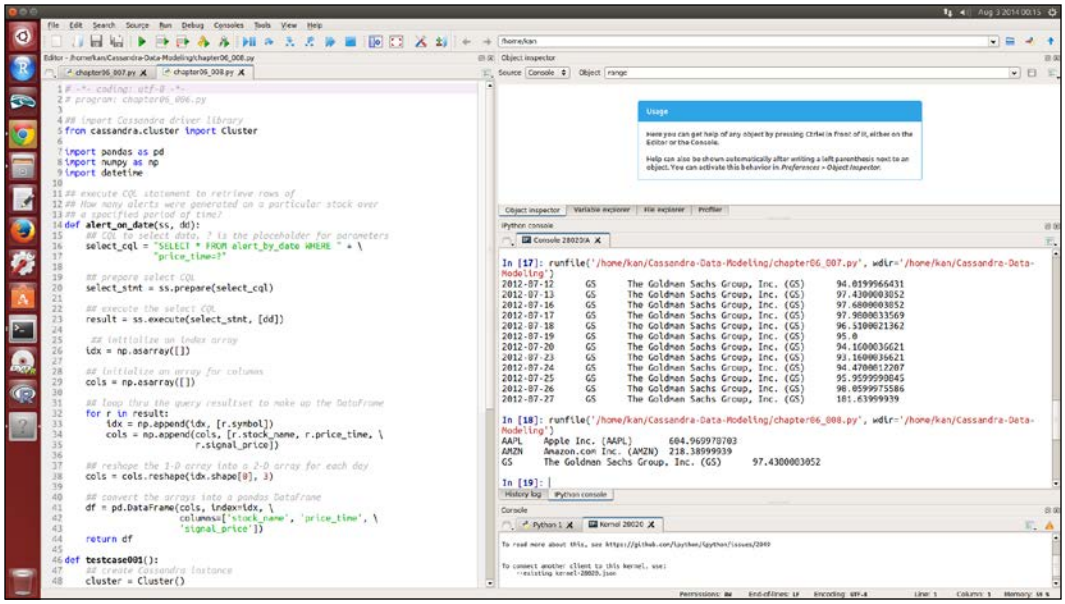
Once again, denormalization is a friend of Cassandra. It does not require a foreign key, referential integrity, or table join.

Implementing system changes

We can now implement the changes to the system one-by-one:

1. First we run `chapter06_001.py` through to `chapter06_004.py` in sequence to make changes to the data model.
2. Then we execute `chapter06_005.py` to retrieve stock quote data for the Watch List. It is worth mentioning that UPSERT is a very nice feature of Cassandra. We do not encounter a duplicate primary key while we insert the same row into a table. It simply updates the row if the row already exists or inserts the row otherwise. It makes the data manipulation logic neat and clean.
3. Further, we run `chatper06_006.py` to store the alerts by scanning over the stock quote data of each stock in the Watch List.

4. Finally, we execute `chapter06_007.py` and `chapter06_008.py` to enquire `alertlist` and `alert_by_date`, respectively. Their sample test results are shown in the following figure:



Summary

This chapter extends the Stock Screener Application by a number of enhancements. We made changes to the data model to demonstrate the modeling by query techniques and how denormalization can help us achieve a high-performance application. We also tried the batch feature provided by Cassandra 2.0.

Note that the source code in this chapter is not housekept and can be refactored somehow. However, because of the limit on the number of pages, it is left as an exercise for the reader.

The Stock Screener Application is now running on a single node cluster.

In the next chapter, we will delve into the considerations and procedures of expanding it to a larger cluster, which is quite common in real-life production systems.

7
Deployment and Monitoring

We have explored the development of the Stock Screener Application in previous chapters; it is now time to consider how to deploy it in the production environment. In this chapter, we will discuss the most important aspects of deploying a Cassandra database in production. These aspects include the selection of an appropriate combination of replication strategy, snitch, and replication factor to make up a fault-tolerant, highly available cluster. Then we will demonstrate the procedure to migrate our Cassandra development database of the Stock Screener Application to a production database. However, cluster maintenance is beyond the scope of this book.

Moreover, a live production system that continuously operates certainly requires monitoring of its health status. We will cover the basic tools and techniques of monitoring a Cassandra cluster, including the nodetool utility, JMX and MBeans, and system log.

Finally, we will explore ways of boosting the performance of Cassandra other than using the defaults. Actually, performance tuning can be made at several levels, from the lowest hardware and system configuration to the highest application coding techniques. We will focus on the **Java Virtual Machine** (**JVM**) level, because Cassandra heavily relies on its underlying performance. In addition, we will touch on how to tune caches for a table.

Replication strategies

This section is about the data replication configuration of a Cassandra cluster. It will cover replication strategies, snitches, and the configuration of the cluster for the Stock Screener Application.

Data replication

Cassandra, by design, can work in a huge cluster across multiple data centers all over the globe. In such a distributed environment, network bandwidth and latency must be critically considered in the architecture, and careful planning in advance is required, otherwise it would lead to catastrophic consequences. The most obvious issue is the time clock synchronization—the genuine means of resolving transaction conflicts that can threaten data integrity in the whole cluster. Cassandra relies on the underlying operating system platform to provide the time clock synchronization service. Furthermore, a node is highly likely to fail at some time and the cluster must be resilient to this typical node failure. These issues have to be thoroughly considered at the architecture level.

Cassandra adopts data replication to tackle these issues, based on the idea of using space in exchange of time. It simply consumes more storage space to make data replicas so as to minimize the complexities in resolving the previously mentioned issues in a cluster.

Data replication is configured by the so-called replication factor in a **keyspace**. The replication factor refers to the total number of copies of each row across the cluster. So a replication factor of `1` (as seen in the examples in previous chapters) means that there is only one copy of each row on one node. For a replication factor of `2`, two copies of each row are on two different nodes. Typically, a replication factor of `3` is sufficient in most production scenarios.

All data replicas are equally important. There are neither master nor slave replicas. So data replication does not have scalability issues. The replication factor can be increased as more nodes are added. However, the replication factor should not be set to exceed the number of nodes in the cluster.

Another unique feature of Cassandra is its awareness of the physical location of nodes in a cluster and their proximity to each other. Cassandra can be configured to know the layout of the data centers and racks by a correct IP address assignment scheme. This setting is known as replication strategy and Cassandra provides two choices for us: `SimpleStrategy` and `NetworkTopologyStrategy`.

SimpleStrategy

`SimpleStrategy` is used on a single machine or on a cluster in a single data center. It places the first replica on a node determined by the partitioner, and then the additional replicas are placed on the next nodes in a clockwise fashion without considering the data center and rack locations. Even though this is the default replication strategy when creating a keyspace, if we ever intend to have more than one data center, we should use `NetworkTopologyStrategy` instead.

NetworkTopologyStrategy

`NetworkTopologyStrategy` becomes aware of the locations of data centers and racks by understanding the IP addresses of the nodes in the cluster. It places replicas in the same data center by the clockwise mechanism until the first node in another rack is reached. It attempts to place replicas on different racks because the nodes in the same rack often fail at the same time due to power, network issues, air conditioning, and so on.

As mentioned, Cassandra knows the physical location from the IP addresses of the nodes. The mapping of the IP addresses to the data centers and racks is referred to as a **snitch**. Simply put, a snitch determines which data centers and racks the nodes belong to. It optimizes read operations by providing information about the network topology to Cassandra such that read requests can be routed efficiently. It also affects how replicas can be distributed in consideration of the physical location of data centers and racks.

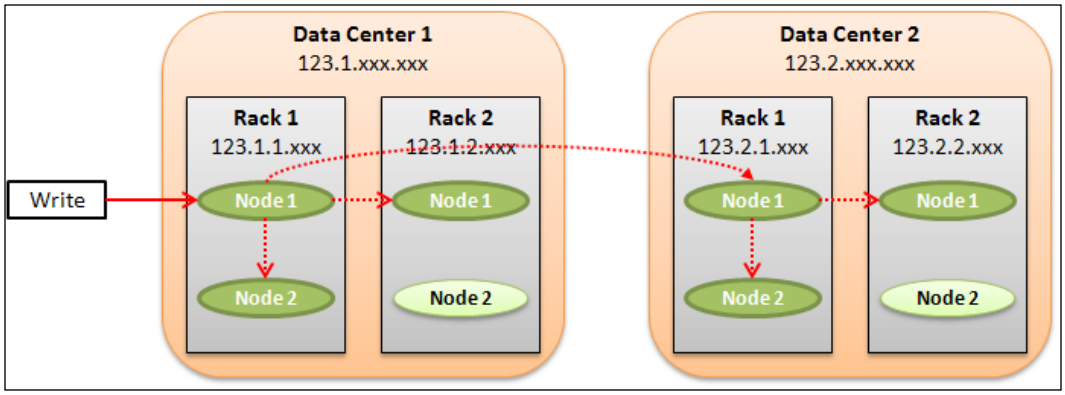
There are many types of snitches available for different scenarios and each comes with its pros and cons. They are briefly described as follows:

- `SimpleSnitch`: This is used for single data center deployments only
- `DynamicSnitch`: This monitors the performance of read operations from different replicas, and chooses the best replica based on historical performance
- `RackInferringSnitch`: This determines the location of the nodes by data center and rack corresponding to the IP addresses
- `PropertyFileSntich`: This determines the locations of the nodes by data center and rack
- `GossipingPropertyFileSnitch`: This automatically updates all nodes using gossip when adding new nodes
- `EC2Snitch`: This is used with Amazon EC2 in a single region
- `EC2MultiRegionSnitch`: This is used with Amazon EC2 in multiple regions
- `GoogleCloudSnitch`: This is used with Google Cloud Platform across one or more regions
- `CloudstackSnitch`: This is used for Apache Cloudstack environments

Snitch Architecture

For more detailed information, please refer to the documentation made by DataStax at `http://www.datastax.com/documentation/cassandra/2.1/cassandra/architecture/architectureSnitchesAbout_c.html`.

The following figure illustrates an example of a cluster of eight nodes in four racks across two data centers using `RackInferringSnitch` and a replication factor of three per data center:



All nodes in the cluster must use the same snitch setting.

Let us look at the IP address assignment in **Data Center 1** first. The IP addresses are grouped and assigned in a top-down fashion. All the nodes in **Data Center 1** are in the same **123.1.0.0** subnet. For those nodes in **Rack 1**, they are in the same **123.1.1.0** subnet. Hence, **Node 1** in **Rack 1** is assigned an IP address of **123.1.1.1** and **Node 2** in **Rack 1** is **123.1.1.2**. The same rule applies to **Rack 2** such that the IP addresses of **Node 1** and **Node 2** in **Rack 2** are **123.1.2.1** and **123.1.2.2**, respectively. For **Data Center 2**, we just change the subnet of the data center to **123.2.0.0** and the racks and nodes in **Data Center 2** are then changed similarly.

The `RackInferringSnitch` deserves a more detailed explanation. It assumes that the network topology is known by properly assigned IP addresses based on the following rule:

IP address = <*arbitrary octet*>.<*data center octet*>.<*rack octet*>.<*node octet*>

The formula for IP address assignment is shown in the previous paragraph. With this very structured assignment of IP addresses, Cassandra can understand the physical location of all nodes in the cluster.

Another thing that we need to understand is the replication factor of the three replicas that are shown in the previous figure. For a cluster with `NetworkToplogyStrategy`, the replication factor is set on a per data center basis. So in our example, three replicas are placed in **Data Center 1** as illustrated by the dotted arrows in the previous diagram. **Data Center 2** is another data center that must have three replicas. Hence, there are six replicas in total across the cluster.

We will not go through every combination of the replication factor, snitch and replication strategy here, but we should now understand the foundation of how Cassandra makes use of them to flexibly deal with different cluster scenarios in real-life production.

Setting up the cluster for Stock Screener Application

Let us return to the Stock Screener Application. The cluster it runs on in *Chapter 6*, *Enhancing a Version*, is a single-node cluster. In this section, we will set up a cluster of two nodes that can be used in small-scale production. We will also migrate the existing data in the development database to the new fresh production cluster. It should be noted that for quorum reads/writes, it's usually best practice to use an odd number of nodes.

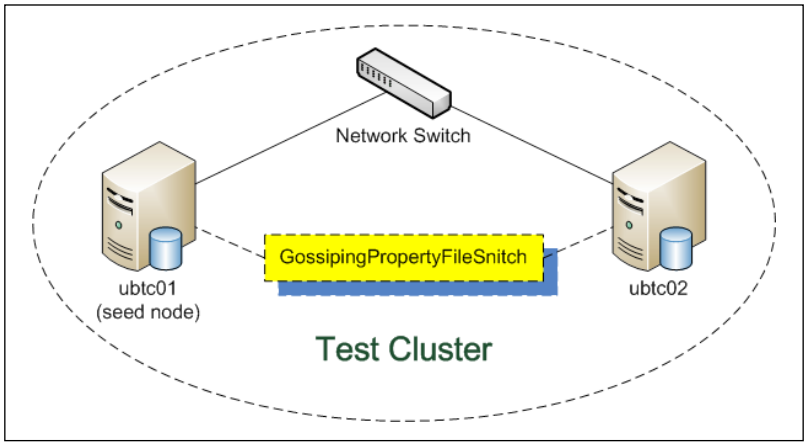
System and network configuration

The steps of installation and setup of the operating system and network configuration are assumed to be done. Moreover, both nodes should have Cassandra freshly installed. The system configuration of the two nodes is identical and shown as follows:

- OS: Ubuntu 12.04 LTS 64-bit
- Processor: Intel Core i7-4771 CPU @3.50GHz x 2
- Memory: 2 GB
- Disk: 20 GB

Global settings

The cluster is named **Test Cluster**, in which both the **ubtc01** and **ubtc02** nodes are in the same rack, `RACK1`, and in the same data center, `NY1`. The logical architecture of the cluster to be set up is depicted in the following diagram:



In order to configure a Cassandra cluster, we need to modify a few properties in the main configuration file, `cassandra.yaml`, for Cassandra. Depending on the installation method of Cassandra, `cassandra.yaml` is located in different directories:

- Package installation: `/etc/cassandra/`
- Tarball installation: `<install_location>/conf/`

The first thing to do is to set the properties in `cassandra.yaml` for each node. As the system configuration of both nodes is the same, the following modification on `cassandra.yaml` settings is identical to them:

```
-seeds: ubtc01
listen_address:
rpc_address: 0.0.0.0
endpoint_snitch: GossipingPropertyFileSnitch
auto_bootstrap: false
```

The reason for using `GossipingPropertyFileSnitch` is that we want the Cassandra cluster to automatically update all nodes with the gossip protocol when adding a new node.

Apart from `cassandra.yaml`, we also need to modify the data center and rack properties in `cassandra-rackdc.properties` in the same location as `cassandra.yaml`. In our case, the data center is `NY1` and the rack is `RACK1`, as shown in the following code:

```
dc=NY1
rack=RACK1
```

Configuration procedure

The configuration procedure of the cluster (refer to the following bash shell scripts: `setup_ubtc01.sh` and `setup_ubtc02.sh`) is enumerated as follows:

1. Stop Cassandra service:

   ```
   ubtc01:~$ sudo service cassandra stop
   ubtc02:~$ sudo service cassandra stop
   ```

2. Remove the system keyspace:

   ```
   ubtc01:~$ sudo rm -rf /var/lib/cassandra/data/system/*
   ubtc02:~$ sudo rm -rf /var/lib/cassandra/data/system/*
   ```

3. Modify `cassandra.yaml` and `cassandra-rackdc.properties` in both nodes based on the global settings as specified in the previous section
4. Start the seed node `ubtc01` first:

   ```
   ubtc01:~$ sudo service cassandra start
   ```

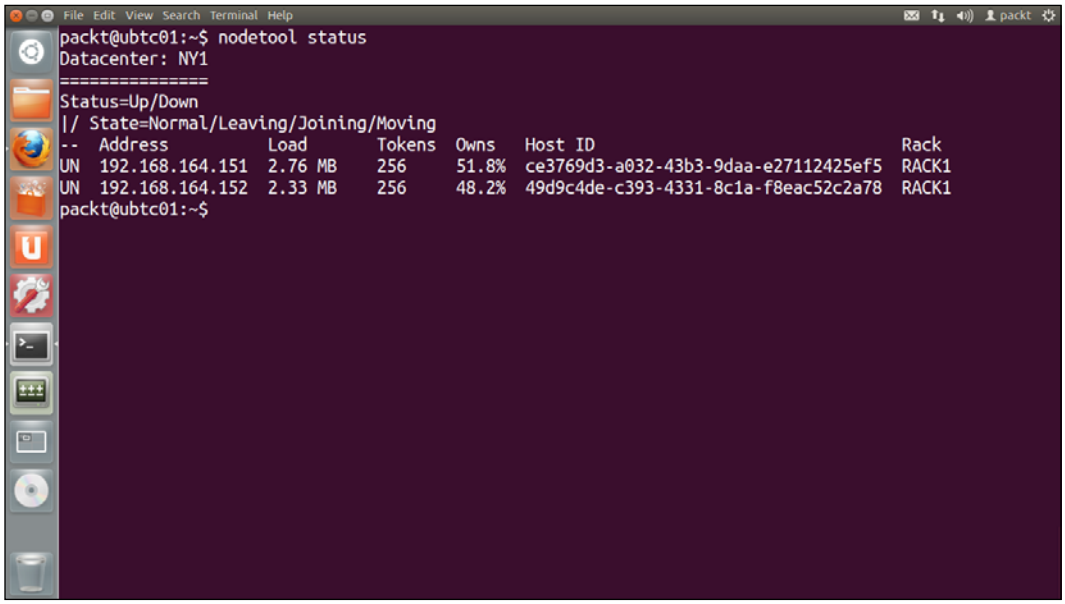
5. Then start `ubtc02`:

   ```
   ubtc02:~$ sudo service cassandra start
   ```

6. Wait for a minute and check if `ubtc01` and `ubtc02` are both up and running:

   ```
   ubtc01:~$ nodetool status
   ```

A successful result of setting up the cluster should resemble something similar to the following screenshot, showing that both nodes are up and running:



Legacy data migration procedure

We now have the cluster ready but it is empty. We can simply rerun the Stock Screener Application to download and fill in the production database again. Alternatively, we can migrate the historical prices collected in the development single-node cluster to this production cluster. In the case of the latter approach, the following procedure can help us ease the data migration task:

1. Take a snapshot of the `packcdma` keyspace in the development database (ubuntu is the hostname of the development machine):

   ```
   ubuntu:~$ nodetool snapshot packtcdma
   ```

2. Record the snapshot directory, in this example, **1412082842986**
3. To play it safe, copy all SSTables under the snapshot directory to a temporary location, say `~/temp/`:

   ```
   ubuntu:~$ mkdir ~/temp/
   ubuntu:~$ mkdir ~/temp/packtcdma/
   ubuntu:~$ mkdir ~/temp/packtcdma/alert_by_date/
   ubuntu:~$ mkdir ~/temp/packtcdma/alertlist/
   ubuntu:~$ mkdir ~/temp/packtcdma/quote/
   ```

```
ubuntu:~$ mkdir ~/temp/packtcdma/watchlist/

ubuntu:~$ sudo cp -p /var/lib/cassandra/data/packtcdma/alert_by_
date/snapshots/1412082842986/* ~/temp/packtcdma/alert_by_date/

ubuntu:~$ sudo cp -p /var/lib/cassandra/data/packtcdma/alertlist/
snapshots/1412082842986/* ~/temp/packtcdma/alertlist/

ubuntu:~$ sudo cp -p /var/lib/cassandra/data/packtcdma/quote/
snapshots/1412082842986/* ~/temp/packtcdma/quote/

ubuntu:~$ sudo cp -p /var/lib/cassandra/data/packtcdma/watchlist/
snapshots/1412082842986/* ~/temp/packtcdma/watchlist/
```

4. Open cqlsh to connect to `ubtc01` and create a keyspace with the appropriate replication strategy in the production cluster:

```
ubuntu:~$ cqlsh ubtc01

cqlsh> CREATE KEYSPACE packtcdma WITH replication = {'class':
'NetworkTopologyStrategy',  'NY1': '2'};
```

5. Create the `alert_by_date`, `alertlist`, `quote`, and `watchlist` tables:

```
cqlsh> CREATE TABLE packtcdma.alert_by_date (
  price_time timestamp,
  symbol varchar,
  signal_price float,
  stock_name varchar,
  PRIMARY KEY (price_time, symbol));
cqlsh> CREATE TABLE packtcdma.alertlist (
  symbol varchar,
  price_time timestamp,
  signal_price float,
  stock_name varchar,
  PRIMARY KEY (symbol, price_time));
cqlsh> CREATE TABLE packtcdma.quote (
  symbol varchar,
  price_time timestamp,
  close_price float,
  high_price float,
  low_price float,
  open_price float,
  stock_name varchar,
  volume double,
  PRIMARY KEY (symbol, price_time));
```

```
cqlsh> CREATE TABLE packtcdma.watchlist (
  watch_list_code varchar,
  symbol varchar,
  PRIMARY KEY (watch_list_code, symbol));
```

6. Load the SSTables back to the production cluster using the `sstableloader` utility:

```
ubuntu:~$ cd ~/temp
ubuntu:~/temp$ sstableloader -d ubtc01 packtcdma/alert_by_date
ubuntu:~/temp$ sstableloader -d ubtc01 packtcdma/alertlist
ubuntu:~/temp$ sstableloader -d ubtc01 packtcdma/quote
ubuntu:~/temp$ sstableloader -d ubtc01 packtcdma/watchlist
```

7. Check the legacy data in the production database on `ubtc02`:

```
cqlsh> select * from packtcdma.alert_by_date;
cqlsh> select * from packtcdma.alertlist;
cqlsh> select * from packtcdma.quote;
cqlsh> select * from packtcdma.watchlist;
```

Although the previous steps look complicated, it is not difficult to understand what they want to achieve. It should be noted that we have set the replication factor per data center as `2` to provide data redundancy on both nodes, as shown in the `CREATE KEYSPACE` statement. The replication factor can be changed in future if needed.

Deploying the Stock Screener Application

As we have set up the production cluster and moved the legacy data into it, it is time to deploy the Stock Screener Application. The only thing needed to modify is the code to establish Cassandra connection to the production cluster. This is very easy to do with Python. The code in `chapter06_006.py` is modified to work with the production cluster as `chapter07_001.py`. A new test case named `testcase003()` is created to replace `testcase002()`. To save pages, the complete source code of `chapter07_001.py` is not shown here; only the `testcase003()` function is depicted as follows:

```
# -*- coding: utf-8 -*-
# program: chapter07_001.py

# other functions are not shown for brevity

def testcase003():
    ## create Cassandra instance with multiple nodes
```

```
        cluster = Cluster(['ubtc01', 'ubtc02'])

        ## establish Cassandra connection, using local default
        session = cluster.connect('packtcdma')

        start_date = datetime.datetime(2012, 6, 28)
        end_date = datetime.datetime(2013, 9, 28)

        ## load the watch list
        stocks_watched = load_watchlist(session, "WS01")

        for symbol in stocks_watched:
            ## retrieve data
            data = retrieve_data(session, symbol, start_date, end_date)

            ## compute 10-Day SMA
            data = sma(data, 10)

            ## generate the buy-and-hold signals
            alerts = signal_close_higher_than_sma10(data)

            ## save the alert list
            for index, r in alerts.iterrows():
                insert_alert(session, symbol, index, \
                             Decimal(r['close_price']), \
                             r['stock_name'])

        ## close Cassandra connection
        cluster.shutdown()

    testcase003()
```

The cluster connection code right at the beginning of the `testcase003()` function is passed with an array of the nodes to be connected (`ubtc01` and `ubtc02`). Here we adopted the default `RoundRobinPolicy` as the connection load balancing policy. It is used to decide how to distribute requests among all possible coordinator nodes in the cluster. There are many other options which are described in the driver API documentation.

Cassandra Driver 2.1 Documentation

For the complete API documentation of the Python Driver 2.1 for Apache Cassandra, you can refer to `http://datastax.github.io/python-driver/api/index.html`.

Monitoring

As the application system goes live, we need to monitor its health day-by-day. Cassandra provides a number of tools for this purpose. We will introduce some of them with pragmatic recommendations. It is remarkable that each operating system also provides a bunch of tools and utilities for monitoring, for example, `top`, `df`, `du` on Linux and Task Manager on Windows. However, they are beyond the scope of this book.

Nodetool

The nodetool utility should not be new to us. It is a command-line interface used to monitor Cassandra and perform routine database operations. It includes the most important metrics for tables, server, and compaction statistics, and other useful commands for administration.

Here are the most commonly used `nodetool` options:

- `status`: This provides a concise summary of the cluster, such as the state, load, and IDs
- `netstats`: This gives the network information for a node, focusing on read repair operations
- `info`: This gives valuable node information including token, on disk load, uptime, Java heap memory usage, key cache, and row cache
- `tpstats`: This provides statistics about the number of active, pending, and completed tasks for each stage of Cassandra operations by thread pool
- `cfstats`: This gets the statistics about one or more tables, such as read-and-write counts and latencies, metrics about SSTable, memtable, bloom filter, and compaction.

A detailed documentation of nodetool can be referred to at `http://www.datastax.com/documentation/cassandra/2.0/cassandra/tools/toolsNodetool_r.html`.

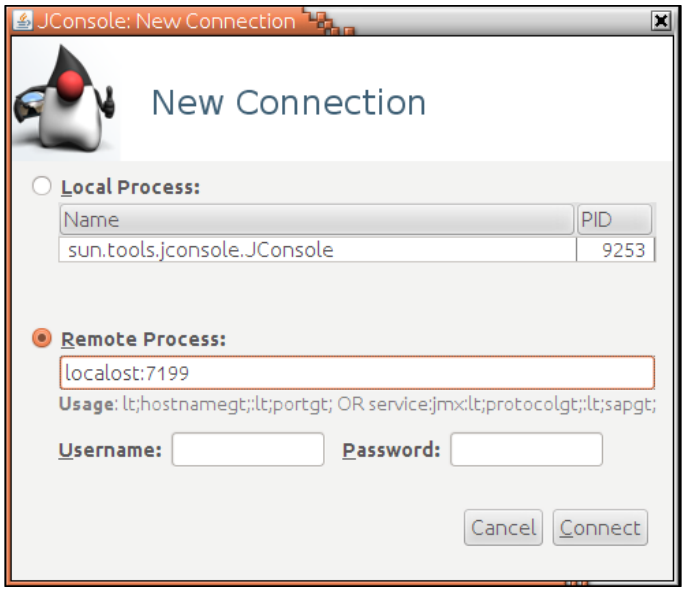
JMX and MBeans

Cassandra is written in the Java language and so it natively supports **Java Management Extensions** (**JMX**). We may use JConsole, a JMX-compliant tool, to monitor Cassandra.

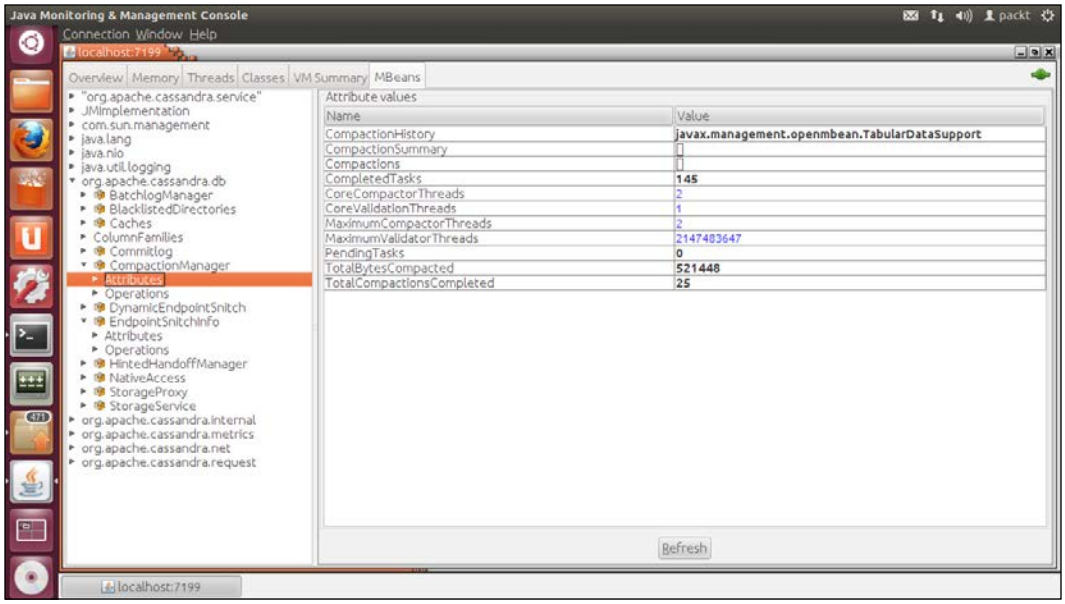
JConsole

JConsole is included with Sun JDK 5.0 and higher versions. However, it consumes a significant amount of system resources. It is recommended that you run it on a remote machine rather than on the same host as a Cassandra node.

We can launch JConsole by typing `jconsole` in a terminal. Assuming that we want to monitor the local node, when the **New Connection** dialog box pops up, we type `localhost:7199` (`7199` is the port number of JMX) in the **Remote Process** textbox, as depicted in the following screenshot:



After having connected to the local Cassandra instance, we will see a well-organized GUI showing six separate tabs placed horizontally on the top, as seen in the following screenshot:



The tabs of the GUI are explained as follows:

- **Overview**: This displays overview information about the JVM and monitored values
- **Memory**: This displays information about heap and non-heap memory usage and garbage collection metrics
- **Threads**: This displays information about thread use
- **Classes**: This displays information about class loading
- **VM Summary**: This displays information about the JVM
- **MBeans**: This displays information about specific Cassandra metrics and operations

Furthermore, Cassandra provides five MBeans for JConsole. They are briefly introduced as follows:

- `org.apache.cassandra.db`: This includes caching, table metrics, and compaction
- `org.apache.cassandra.internal`: These are internal server operations such as gossip and hinted handoff

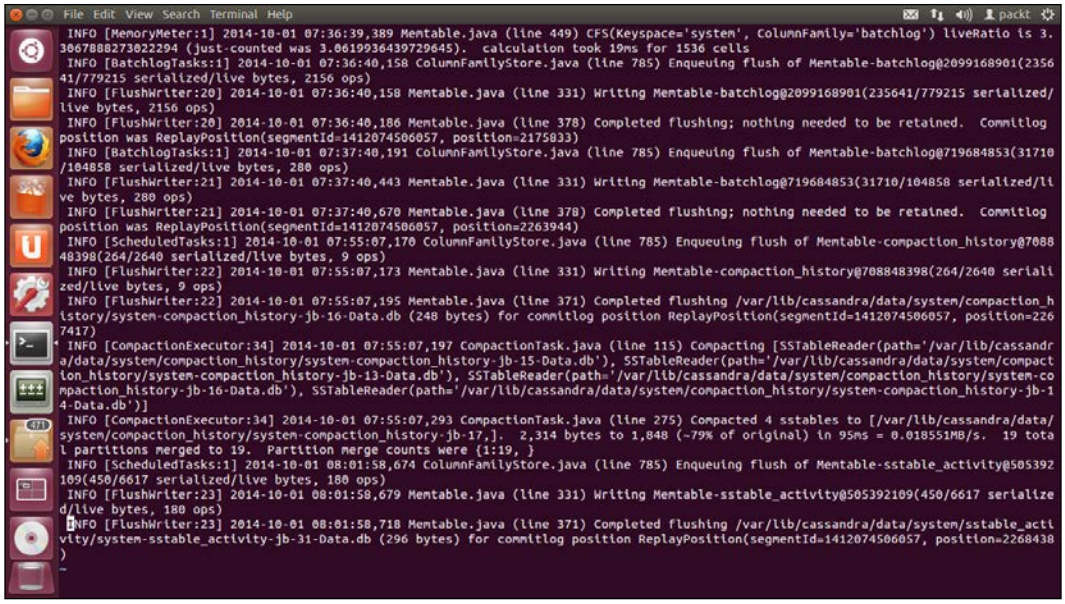
- `org.apache.cassandra.metrics`: These are various metrics of the Cassandra instance such as cache and compaction
- `org.apache.cassandra.net`: This has Inter-node communication including FailureDetector, MessagingService and StreamingService
- `org.apache.cassandra.request`: These include tasks related to read, write, and replication operations

MBeans

An **Managed Bean** (**MBean**) is a Java object that represents a manageable resource such as an application, a service, a component, or a device running in the JVM. It can be used to collect statistics on concerns such as performance, resource usage, or problems, for getting and setting application configurations or properties, and notifying events like faults or state changes.

The system log

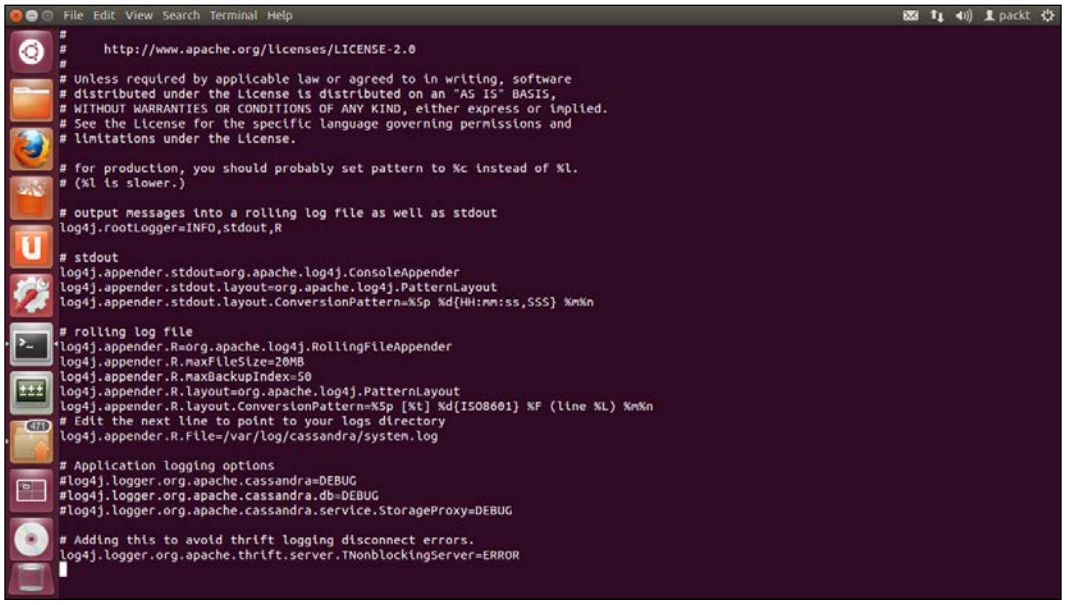
The most rudimentary, yet the most powerful, monitoring tool is Cassandra's system log. The default location of the system log is named `system.log` under `/var/log/cassandra/`. It is simply a text file and can be viewed or edited by any text editor. The following screenshot shows an extract of `system.log`:



This piece of log looks long and weird. However, if you are a Java developer and you are familiar with the standard log library, Log4j, it is pretty straightforward. The beauty of Log4j is the provision of different log levels for us to control the granularity of the log statements to be recorded in `system.log`. As shown in the previous figure, the first word of each line is `INFO`, meaning that the log statement is a piece of information. Other log level choices include `FATAL`, `ERROR`, `WARN`, `DEBUG`, and `TRACE`, from the least verbose to the most verbose.

The system log is very valuable in troubleshooting problems as well. We may increase the log level to `DEBUG` or `TRACE` for troubleshooting. However, running a production Cassandra cluster in the `DEBUG` or `TRACE` mode will degrade its performance significantly. We must use them with great care.

We can change the standard log level in Cassandra by adjusting the `log4j.rootLogger` property in `log4j-server.properties` in the Cassandra configuration directory. The following screenshot shows the content of `log4j-server.properties` in ubtc02:



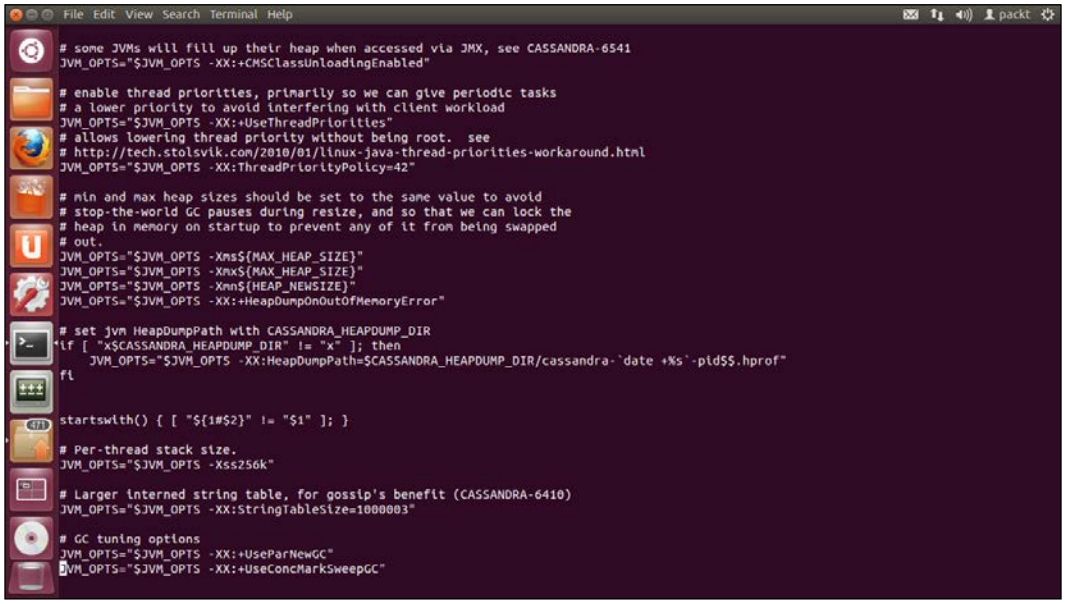
It is important to mention that `system.log` and `log4j-server.properties` are only responsible for a single node. For a cluster of two nodes, we will have two `system.log` and two `log4j-server.properties` on the respective nodes.

Performance tuning

Performance tuning is a large and complex topic that in itself can be a whole course. We can only scratch the surface of it in this short section. Similar to monitoring in the last section, operating system-specific performance tuning techniques are beyond the scope of this book.

Java virtual machine

Based on the information given by the monitoring tools and the system log, we can discover opportunities for performance tuning. The first things we usually watch are the Java heap memory and garbage collection. JVM's configuration settings are controlled in the environment settings file for Cassandra, `cassandra-env.sh`, located in `/etc/cassandra/`. An example is shown in the following screenshot:



Basically, it already has the boilerplate options calculated to be optimized for the host system. It is also accompanied with explanation for us to tweak specific JVM parameters and the startup options of a Cassandra instance when we experience real issues; otherwise, these boilerplate options should not be altered.

A detailed documentation on how to tune JVM for Cassandra can be found at `http://www.datastax.com/documentation/cassandra/2.0/cassandra/operations/ops_tune_jvm_c.html`.

Caching

Another area we should pay attention to is caching. Cassandra includes integrated caching and distributes cache data around the cluster. For a cache specific to a table, we will focus on the partition key cache and the row cache.

Partition key cache

The partition key cache, or key cache for short, is a cache of the partition index for a table. Using the key cache saves processor time and memory. However, enabling just the key cache makes the disk activity actually read the requested data rows.

Row cache

The row cache is similar to a traditional cache. When a row is accessed, the entire row is pulled into memory, merging from multiple SSTables when required, and cached. This prevents Cassandra from retrieving that row using disk I/O again, which can tremendously improve read performance.

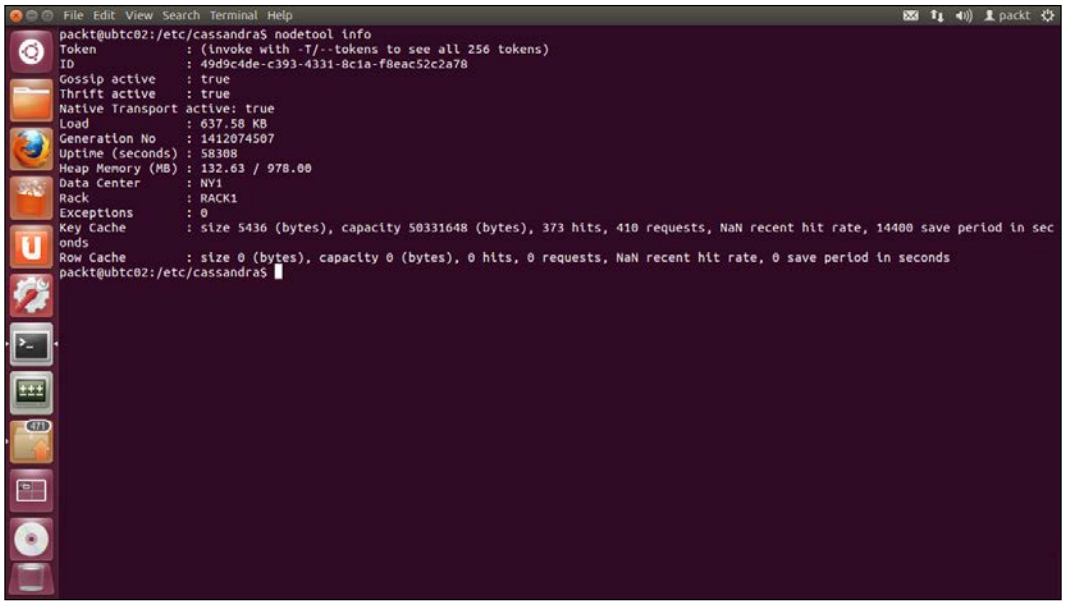
When both row cache and partition key cache are configured, the row cache returns results whenever possible. In the event of a row cache miss, the partition key cache might still provide a hit that makes the disk seek much more efficient.

However, there is one caveat. Cassandra caches all the rows of a partition when reading that partition. So if the partition is large or only a small portion of the partition is read every time, the row cache might not be beneficial. It is very easy to be misused and consequently the JVM will be exhausted, causing Cassandra to fail. That is why the row cache is disabled by default.

We usually enable either the key or row cache for a table, not both at the same time.

Monitoring cache

Either the `nodetool info` command or JMX MBeans can provide assistance in monitoring cache. We should make changes to cache options in small, incremental adjustments, and then monitor the effects of each change using the nodetool utility. The last two lines of output of the `nodetool info` command, as seen in the following figure, contain the `Row Cache` and `Key Cache` metrics of `ubtc02`:



In the event of high memory consumption, we can consider tuning data caches.

Enabling/disabling cache

We use the CQL to enable or disable caching by altering the cache property of a table. For instance, we use the `ALTER TABLE` statement to enable the row cache for `watchlist`:

```
ALTER TABLE watchlist WITH caching=''ROWS_ONLY'';
```

Other available table caching options include `ALL`, `KEYS_ONLY` and `NONE`. They are quite self-explanatory and we do not go through each of them here.

Further information about data caching can be found at `http://www.datastax.com/documentation/cassandra/2.0/cassandra/operations/ops_configuring_caches_c.html`.

Summary

This chapter highlights the most important aspects of deploying a Cassandra cluster into the production environment. Cassandra can be taught to understand the physical location of the nodes in the cluster in order to intelligently manage its availability, scalability and performance. We deployed the Stock Screener Application to the production environment, though the scale is small. It is also valuable for us to learn how to migrate legacy data from a non-production environment.

We then learned the basics of monitoring and performance tuning which are a must for a live running system. If you have experience in deploying other database and system, you may well appreciate the neatness and simplicity of Cassandra.

In the next chapter, we will have a look at the supplementary information pertinent to application design and development. We will also summarize of the essence of each chapter.

8
Final Thoughts

In the previous chapters, we went through a quick journey of developing a technical analysis application in Python using Cassandra. We started from the theoretical basic knowledge and proceeded step-by-step to design and develop a running application. Even though you are a novice in computer programming, you should have no trouble reading the chapters in a sequence.

We now come to the final chapter of this book. We will take a look at the supplementary information pertinent to application design and development. Then we will quickly review the basics of each chapter in order to wrap up this book.

Supplementary information

Here, we will take a glance at the supplementary information on client drivers, security features, backup and restore.

Client drivers

A driver eases the burden of an application developer to deal with the repetitive, low-level nitty-gritty of communicating with the underlying database. The application developer can then focus on her/his efforts in writing business logic.

As Cassandra is growing popular, drivers are developed for the contemporary programming languages. This greatly simplifies the workload of an application developer, who was used to the clumsy Thrift API.

Drivers for different languages

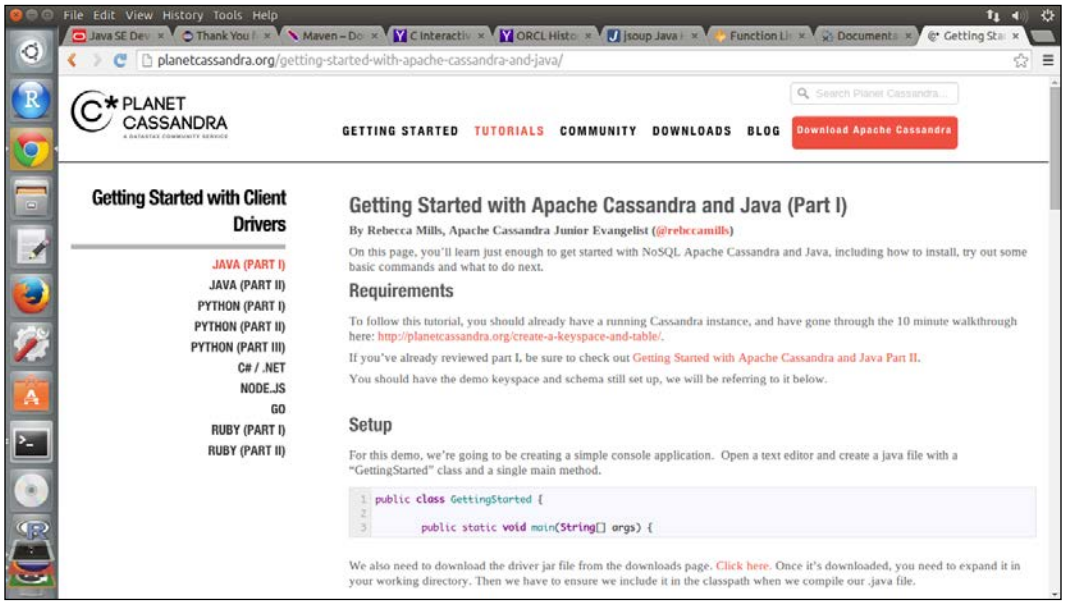
A list of the commonly used Cassandra drivers and their corresponding supported programming languages can be seen at PlanetCassandra `http://planetcassandra.org/client-drivers-tools/`.

A great number of Cassandra drivers available nowadays is still growing and many of them are open source. If you really want production support, then DataStax is worth your consideration.

A few comments on the selection of a driver are provided as follows:

- First, the programming language to be supported is the single most important constraint. The communication protocol then comes into play. Thrift API is long-lived, yet rather difficult to use. Unless you need to support an application working with an older version of a Cassandra cluster, a driver providing CQL support is highly recommended and it will align the data modeling techniques that are introduced in this book. The implementation of the data model will be much easier as well.
- Another selection factor is the additional features that the driver offers, for example, node failure detection, node failover, automatic load balancing, and performance.

PlanetCassandra also provides crispy tutorials on how to get started with the client drivers, as shown in the following screenshot:



Security

Security is a broad and complex topic. From the perspective of application development, authentication, authorization and inter-node encryption are the most fundamental security measures to safeguard a production application.

Authentication

In Cassandra, authentication is based on an internally controlled login username and password. The login username and password are stored in the `system_auth.credentials` table. The internal authentication is disabled, by default. We can configure Cassandra in order to enable it by modifying `cassandra.yaml`. We also need to increase the replication factor of the `system_auth` keyspace, as the `system_auth` keyspace is no different and might fail as well!

Once the internal authentication is enabled, we can use the superuser account and CQL statements such as `CREATE USER`, `ALTER USER`, `DROP USER`, and `LIST USERS` to create and manage user authentication.

Authorization

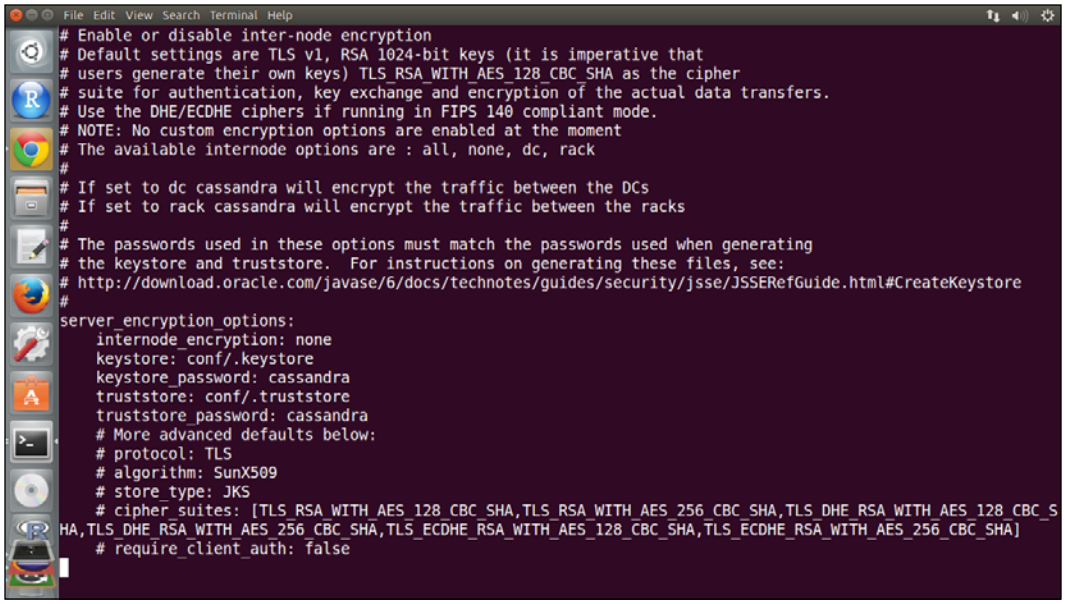
Similarly, Cassandra provides internal authorization to work hand-in-hand with internal authentication. It borrows the practices from the traditional database `GRANT`/`REVOKE` paradigm in order to manage permissions on the schema objects.

The tables in the `system` keyspace are granted read permission, by default, to every authenticated user. For user-created keyspace and the objects inside, we can also use CQL statements, namely, `GRANT`, `REVOKE`, and `LIST PERMISSIONS` to manage the object permission.

Inter-node encryption

Cassandra provides inter-node encryption that protects data transferred among nodes, including gossip communication, in a cluster using **Secure Sockets Layer** (**SSL**). All nodes must have all the relevant SSL certificates on all nodes. The encryption can be applied to the traffic between all nodes, data centers, or racks.

We must set the `server_encryption_options` in the `cassandra.yaml` file on each node in order to enable the inter-node encryption option and the configuration settings of the `keystore` and `truststore` files, as shown in the following screenshot:



Backup and restore

Backup is an interesting topic in a large distributed system such as Cassandra. It is very likely that the data volume will be gigantic and the number of nodes will be large. Making a consistent backup of the whole cluster can be very tricky.

In my opinion, backup is optional in Cassandra, in contrast to the must-have regular backups of a traditional database. The need of backing up a Cassandra cluster really depends on the chosen deployment strategies. For example, if the nodes of a cluster are distributed in geographically dispersed areas like New York, Tokyo, and London, and the replication factor is set to three or above, it might be beneficial to explicitly have an external backup of the data in the cluster. This example cluster has built-in resilience and each piece of data has a number of replicas serving as a backup of itself. The chance of the simultaneous failure of all the three geographical areas is rather low.

Of course, you might still make regular backups of the cluster if you need to comply with policies, regulations, and so on. Maybe you want to have the so-called point-in-time recovery available for the system. In these cases, a managed backup is a must. However, this will definitely complicate the whole system architecture.

All in all, this is a design decision for your implementation.

Useful websites

Here are some of the useful websites for us to get up-to-date Cassandra information.

Apache Cassandra official site

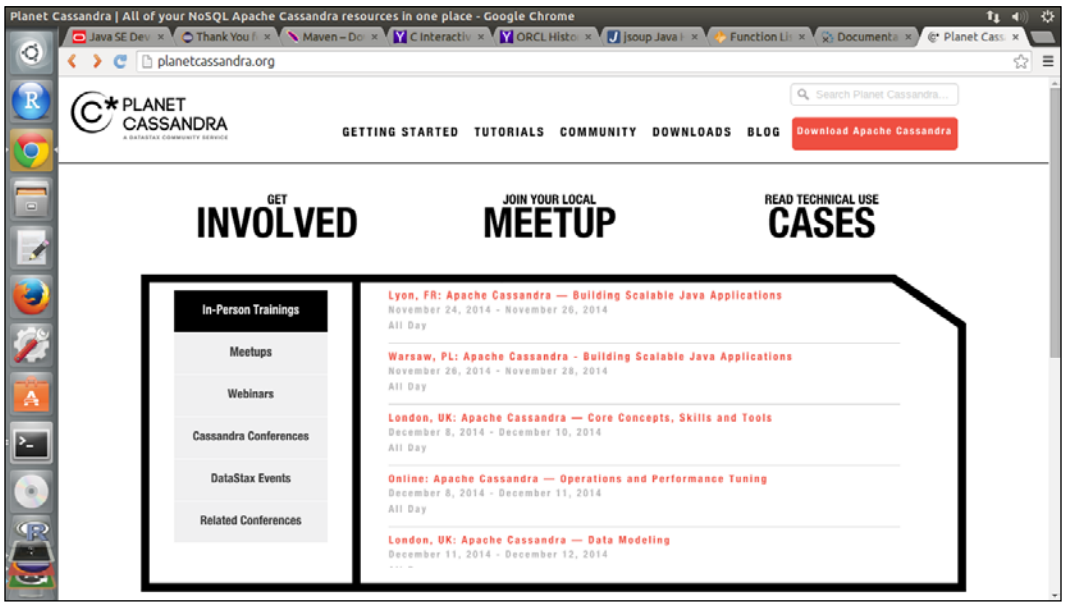
The official Cassandra website, `http://cassandra.apache.org/`, is always the first place to go for any information. The latest released version information can be found on its home page. You might get the source code there if you want to dive deep into the heart of Cassandra or if you want to install a Cassandra instance back to square one by building from the source code.

Just akin to other projects under the Apache Software Foundation, you are welcome to contribute to the community. You can also find out how to join this enthusiastic team of developers in order to improve such a great NoSQL database.

You can also find a link to another website called PlanetCassandra, which is worth a separate introduction.

PlanetCassandra

PlanetCassandra, `http://planetcassandra.org/`, is a community service website supported by DataStax, a commercial company, that provides production-ready Apache Cassandra products and services:



This website deals more with the collaboration aspects of the Cassandra community. We can look for meetups, involvements, webinars, conferences and events, and even educational training courses there. The most valuable section of the website is the *Apache Cassandra Use Cases* that is a repository of the companies who run their applications on Apache Cassandra and enjoy the real benefits from it.

The repository is categorized by several dimensions, namely, **Product Catalog/Playlist**, **Recommendation/Personalization**, **Fraud Detection**, **Messaging**, **IOT/Sensor Data**, and **Undefined**. Each entry of the repository has a name and a brief introduction of the company of the use case, and how it uses Cassandra to drive the business. You certainly can learn and generate some ideas by learning from the use cases.

A must-read is the Netflix case study. The use case is a personalization system that understands each person's unique habits and preferences and bring to light products and items that a user might be unaware of and not looking for. The challenges were to acquire affordable capacity in order to store and process immense amounts of data, to address a single point of failure with Oracle's legacy relational architecture, and to achieve business agility for international expansion. Netflix used a commercial version of Cassandra that delivers 100 percent uptime and cost-effective scale across multiple data centers. The results are stunning, which are as follows:

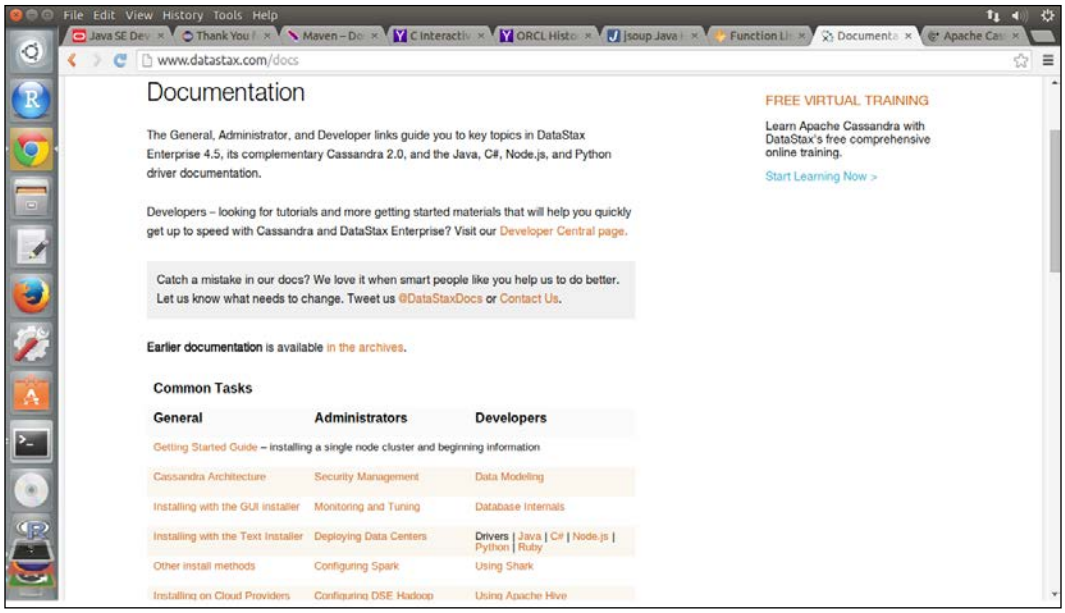
- First, the throughput of the system is more than 10 million transactions per second
- Second, the creation and management of the new data clusters across various regions is nearly effortless
- Lastly, customer viewing and log data can be captured in the finest detail in Cassandra

It is highly recommended that you read this, especially for those of you who are considering to migrate from a relational database to Cassandra.

DataStax

The Cassandra version used in this book is an open source one that can be obtained freely on the Internet. It is good enough for most systems. However, many companies still look for enterprise grade products built on Cassandra and the related support, training, and consultancy services. DataStax, `http://www.datastax.com/`, is one of them.

DataStax serves to compile the most comprehensive Cassandra documentation, as shown in the following screenshot. The documentation is freely available on its website. It also develops and provides support to the client drivers for Java, C#, Python, and so on:



DataStax offers an enterprise version of Apache Cassandra, known as DataStax Enterprise, with enhanced features such as advanced security and management tools that simplify the day-to-day system management of a Cassandra cluster.

DataStax Enterprise includes a powerful enterprise system management tool, OpsCenter, to allow administrators to easily grasp the status and performance of the system through a dashboard. It monitors the cluster and triggers alerts or notifications of changes in the cluster. Backup and restore operations are greatly streamlined as well.

DataStax Enterprise also extends Cassandra to support Apache Hadoop and Solr, as an integrated enterprise platform.

Hadoop integration

Cassandra integrated with Hadoop can be a powerful platform for Big Data Analytics. Cassandra has been able to directly integrate with Hadoop since its Version 0.6. It began with MapReduce support. Since then, the support has matured significantly and now includes native support for Pig and Hive. Cassandra's Hadoop support implements the same interface as **Hadoop Distributed File System** (**HDFS**) in order to achieve input data locality.

Cassandra provides the `ColumnFamilyInputFormat` and `ColumnFamilyOutputFormat` classes for direct integration with Hadoop from MapReduce programs. It involves data being read directly from Cassandra column families in MapReduce mappers and does include data movement.

Setup and configuration involves overlaying a Hadoop cluster on Cassandra nodes, configuring a separate server for the Hadoop JobTracker, and installing a Hadoop TaskTracker and DataNode on each Cassandra node.

Setup and configuration procedures

The detailed procedures of integrating Cassandra with Hadoop can be found at:

- `http://www.datastax.com/documentation/cassandra/2.1/cassandra/configuration/configHadoop.html`
- `http://wiki.apache.org/cassandra/HadoopSupport`

The nodes in the Cassandra data center can draw from data in the HDFS DataNode as well as from Cassandra. The JobTracker receives the MapReduce input from the client application. It then sends a MapReduce job request to the TaskTrackers and optional clients, for example, MapReduce and Pig. The data is written to Cassandra and the results are sent back to the client.

DataStax has also created a simple way to use Hadoop with Cassandra and built it into the enterprise version.

Summary

We started from *Chapter 1*, *Bird's Eye View of Cassandra*, to review the basics of Cassandra. We then touched on the important facets of data modeling in Cassandra, such as Modeling by query technique, the wealthy set of data types, and the indexes. These techniques and knowledge are put together into an example data analysis application in the stock trading domain called Stock Screener Application. We walked through and explained every single detail of the application, although at a fast pace. We also illustrated how to enhance the first-cut version with changes on the data model and coding to demonstrate the great flexibility provided by Cassandra. We then turned to plan and migrate the enhanced system to a production-ready cluster with considerations on the replication strategy, snitch, replication factor, the basic monitoring and performance tuning tools.

I really enjoyed writing this book for you and I sincerely hope that it is as useful for you when it comes to rapidly using Cassandra in real-world projects, as it has been fun for me. Your comments are always welcomed and you can contact me through Packt Publishing.

As Sir Winston Churchill said:

> "*Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.*"

Index

A

Alert List 91, 96, 121
ALLOW FILTERING clause 86
Amazon Dynamo 12, 16
Anaconda
  about 99
  URL 99
Apache Cassandra 13
Apache HBase 13
ASCII data type 53
Atomic-Consistency-Isolation-Durability (ACID) 17
authentication 161
authorization 161
auto-sharding 16

B

backup 162
BeautifulSoup 128
benefits, NoSQL 9
Berkeley DB 12
Big Data 8
BigDecimal data type 55
bigint data type 54
BLOB constant 54
BLOB data type 54
bloom filter 22
boolean data type 55
bucketing 44, 45
ByteOrderedPartitioner
  about 82
  limitations 82

C

caching
  about 156
  cache, disabling 157
  cache, enabling 157
  cache, monitoring 156, 157
  partition key cache 156
  row cache 156
CAP theorem 15
Cassandra
  about 14
  data, storing 104-106
  features 25
  high-level architecture 17-24
  implementation 50
  URL, for official website 163
  URL, for source code 51
Cassandra CLI 52
Cassandra data model
  collections 34
  consideration 28
  counter column 35
  logical data structure 29
  Map data structure 29
  no foreign key 34
  no join 34
  no sequence 35
  secondary index 36
  SortedMap data structure 29
  Time-To-Live (TTL) 35
  unique architecture 28
Cassandra driver 99
Cassandra Query Language. *See* **CQL**

Cassandra read consistency levels
ALL 24
EACH_QUORUM 24
LOCAL_QUORUM 24
ONE 24
QUORUM 24
THREE 24
TWO 24
Cassandra Version 2.0.9 97
Cassandra write consistency levels
ALL 22
ANY 22
EACH_QUORUM 22
LOCAL_QUORUM 22
ONE 22
QUORUM 22
THREE 22
TWO 22
client drivers
about 159
selecting 160
CloudstackSnitch 141
cluster 18
cluster, setting up for Stock Screener Application
about 143
configuration procedure 145, 146
global settings 144
legacy data migration procedure 146-148
system and network configuration 143
code design, Stock Screener Application
about 101
Data Feed Provider 101
Stock Screener 109
code enhancement, Stock Screener Application
about 125
Data Mapper and Archiver 125-128
alerts, querying 133-137
Stock Screener Engine 129-133
collections
about 34, 64-66
list 34, 66
map 34, 67
reference link 66
set 34, 66
column 30
column family
about 31
structure 31, 32
column-family store, NoSQL database 13
Comma Separated Values (CSV) 93
composite partition key
about 74-78
time-series data 79, 80
compound primary key 74-78
consistent hashing 19
Coordinated Universal Time (UTC) 60
CouchDB 13
counter column 35
counter data type 63
CQL
about 25, 45, 47
data types 49
CQL command-line client 48, 49
CQL keywords
reference link 48
cqlsh 48, 49
CQL statements
about 47, 48
data definition statements 47, 48
data manipulation statements 47
query statements 47

D

data caching
reference link 157
Data Definition Language (DDL) 37
data duplication 44
Data Feed Provider
about 91
Data Feed 95
Data Feed Adapter 95
Data Mapper and Archiver 95
data, storing in Cassandra 104-106
data, transforming 103
stock quote, collecting 101, 102
summarizing 107
tasks 101
data manipulation statements 47
Data Mapper and Archiver 95, 119
data modeling, by query
about 36

Cassandra version 38-43
relational version 36-38
data modeling considerations
about 44
bucketing 44, 45
data duplication 44
sorting 44
time-series data 45
valueless column 45
wide row 44
data model, Stock Screener Application
enhancement approach 118
evolving 117, 118
data modification language (DML) 133
Data Platforms Landscape Map 8
data replication 140
Data Scoper 96, 109, 110
DataStax
about 97, 165, 166
URL 165
data types, CQL
ASCII 49, 53
bigint 49, 54
BLOB 49, 54
boolean 49, 55
counter 49, 63
decimal 49, 55
double 49, 55
example 51, 52
float 49, 55
inet 49, 56, 57
int 49, 57
text 49, 57
timestamp 50, 59, 60
timeuuid 50, 61
UUID 50, 62
varchar 50, 62
varint 50, 62
date bucket pattern 80
dateOf() function 61
decimal data type 55
denormalization 34, 44
document-based repository, NoSQL database 13
double data type 55
DynamicSnitch 141

E

EC2MultiRegionSnitch 141
EC2Snitch 141
end-of-day (EOD) 91
End-to-End Test run 114
enhancement approach, Stock Screener Application
about 118
Alert List 121, 122
descriptive stock name, adding 122, 123
alerts, querying 123, 124
Watch List 120, 121
EODData
URL 92
epidemic protocol 21
Eventual Consistency 16

F

Failure detection 21
financial analysis 90
float data type 55
FlockDB 14
fundamental analysis 90

G

Google BigTable 13, 15
GoogleCloudSnitch 141
Google Finance
URL 92
gossip 20
GossipingPropertyFileSnitch 141
graph database, NoSQL database 14

H

Hadoop Distributed File System (HDFS) 167
Hadoop integration
about 167
URL, for procedure 167
hashmap 12
high-level architecture, Cassandra
Failure detection 21
gossip 21

partitioning 18, 19
read path 23
repair mechanism 24
replication 19
seed node 20
snitch 20
write path 21, 23
Historical Data 91

I

idempotent 35
impedance mismatch 11
implementation, Cassandra 50
inet data type 56, 57
initial data model 93, 94
int data type 57
integrated development environment (IDE) 99
Internet Engineering Task Force (IETF) 29
Internet Protocol (IP) 57
inter-node encryption 161
IP Version 4 (IPv4) 56
IP Version 6 (IPv6) 56, 57
ISO 8601 60

J

Java Native Access (JNA) 97
Java Runtime Environment (JRE)
about 97
URL 97
JavaScript Object Notation (JSON) 13
Java Virtual Machine (JVM)
about 97, 139, 155
configuration settings 155
installing 97
reference link, for documentation 155
JConsole
about 151
launching 151
MBeans 152
tabs 152
JMX 151

K

keyspace 18, 32, 140
key/value pair store, NoSQL database 12

L

list 34, 66
logical data structure
about 29
column 30
column family 31, 32
keyspace 32
row 30, 31
super column 33
super column family 33

M

Managed Bean. *See* **MBeans**
map 34, 67
Map data structure 29
maxTimeuuid() function 61
MBeans 151-153
MBeans, for JConsole
org.apache.cassandra.db 152
org.apache.cassandra.internal 152
org.apache.cassandra.metrics 153
org.apache.cassandra.net 153
org.apache.cassandra.request 153
minTimeuuid() function 61
MongoDB 13
monitoring 150
multiple secondary indexes 85
Murmur3Partitioner 81

N

Neo4J 14
Netflix
about 14
URL, for case study 14
NetworkTopologyStrategy 141
nodetool
about 150
reference link, for documentation 150
nodetool, options
cfstats 150
info 150
netstats 150
status 150

tpstats 150
non-reserved keywords 48
NoSQL
about 8
benefits 9
URL 8
NoSQL databases
limitations 10
overview 10, 11
types 12
NoSQL databases, types
column-family store 13
document-based repository 13
graph database 14
key/value pair store 12
now() function 61

O

open price, high price, low price, close price and volume (OHLCV) 92
operating system 96

P

paging 82
pandas
about 101
URL 101
partitioner
about 81
ByteOrderedPartitioner 82
Murmur3Partitioner 81
paging 82
RandomPartitioner 81
TOKEN() function 82
partitioning 18, 19
partition key cache 156
performance tuning
about 155
caching 156
Java Virtual Machine (JVM) 155
Phi Accrual Failure Detection Algorithm 21
pip
about 99
URL 99
PlanetCassandra
about 164, 165
URL 159, 164
price action, stock
close price 91
high price 91
low price 91
open price 91
primary index
about 71-74
differentiating, with secondary index 83
primary key 71
processing flow, Stock Screener Application 94-96
programming language 98
PropertyFileSntich 141
Python 2.7
installing, in Ubuntu 98
Python Driver 2.0
URL 99
Python Driver 2.1
reference link, for API documentation 149

Q

Query 119
query statements 47

R

RackInferringSnitch
about 141
using 142
RandomPartitioner 81
read path 23
relational database 7
remote procedure call (RPC) 33
repair mechanisms
anti-entropy 25
hinted handoff 24
read repair 24
replication strategies
about 19, 139
data replication 140
NetworkTopologyStrategy 141
SimpleStrategy 140
Request for Comments (RFC) 29
reserved keywords 48
restore 162
Riak 12

row 30, 31
row cache 156

S

screening rule 96, 111
Screen Rule 96
secondary index
about 36, 83-85
caveats 84
differentiating, with primary index 83
don'ts 86, 87
do's 86, 87
multiple secondary indexes 85
reference link 86
Secure Sockets Layer (SSL) 161
security
about 161
authentication 161
authorization 161
inter-node encryption 161
seed node 20
set 34, 66
Simple Moving Average (SMA) 111
SimpleSnitch 141
SimpleStrategy 140
snitch
about 20, 141
CloudstackSnitch 141
DynamicSnitch 141
EC2MultiRegionSnitch 141
EC2Snitch 141
GoogleCloudSnitch 141
GossipingPropertyFileSnitch 141
PropertyFileSntich 141
RackInferringSnitch 141
SimpleSnitch 141
SortedMap data structure 29
Sorted String Table (SSTable) 15
sorting 44
sort order 31
stock quote data
about 91-93
collecting 101, 102
Stock Screener Application
about 90, 91, 109
cluster, setting up 143
code design 101
components 109
Data Scoper 109, 110
deploying 148, 149
development 101
engine 111
enhancing 117
financial analysis 90-93
initial data model 93, 94
overview 89
processing flow 94
screening rule 111
stock quote data 91-93
time-series data 111
Stock Screener Application, enhancements
code, enhancing 125
data model, evolving 117, 118
system changes, implementing 137, 138
Stock Screener engine 96, 111
super column 33
super column family 33
system design
about 96
Cassandra driver 99
Cassandra Version 2.0.9 97
IDE 99
Java Native Access (JNA) 97
Java Runtime Environment (JRE) 97
operating system 96
programming language 98, 99
system overview 100
system log 153, 154

T

technical analysis 90
Technical Analysis Signals 96
Test Cluster 144
text data type 57
Thrift 33
time-series data 45, 79, 80
timestamp data type
about 59, 60
versus timeuuid data type 62
Time-To-Live (TTL) 30, 35

timeuuid data type 61
TOKEN() function 82
tools, for monitoring
about 150
JMX 151
MBeans 151
nodetool 150
system log 153, 154
tuple type 67-69
type 1 UUID
versus type 4 UUID 62

U

ubtc01 node 144
ubtc02 node 144
Ubuntu
URL 96
Universal Unique ID (UUID) 29
unixTimestampOf() function 61
UPSERT 73
user-defined types (UDT) 67-69
UUID data type 62

V

valueless column 45
varchar data type 62
varint data type 62

W

Watch List 119, 120
websites, for Cassandra information
about 163
Apache Cassandra official site 163
DataStax 165, 166
Hadoop integration 167
PlanetCassandra 164, 165
WHERE clause 85
wide row 44
write path 21

Y

Yahoo! Finance
URL 92

Thank you for buying

Cassandra Data Modeling and Analysis

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

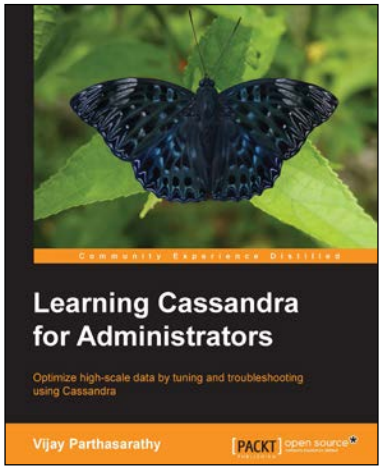
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced 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.

[PACKT] open source
community experience distilled
PUBLISHING

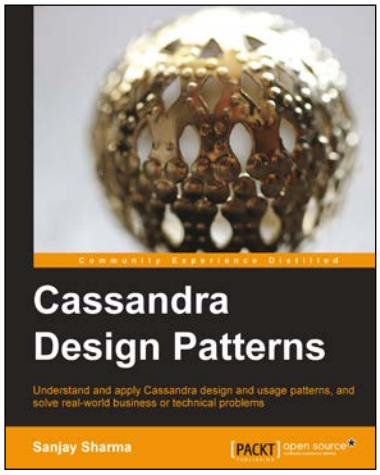


Learning Cassandra for Administrators

ISBN: 978-1-78216-817-1 Paperback: 120 pages

Optimize high-scale data by tuning and troubleshooting using Cassandra

1. Install and set up a multi datacenter Cassandra.
2. Troubleshoot and tune Cassandra.
3. Covers CAP tradeoffs, physical/hardware limitations, and helps you understand the magic.
4. Tune your kernel, JVM, to maximize the performance.



Cassandra Design Patterns

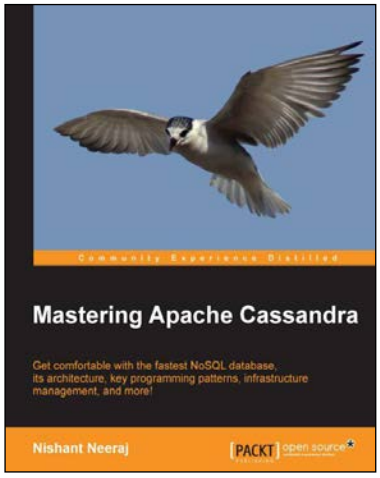
ISBN: 978-1-78328-880-9 Paperback: 88 pages

Understand and apply Cassandra design and usage patterns, and solve real-world business or technical problems

1. Learn how to identify real-world use cases that Cassandra solves easily, in order to use it effectively.
2. Identify and apply usage and design patterns to solve specific business and technical problems including technologies that work in tandem with Cassandra.
3. A hands-on guide that will show you the strengths of the technology and help you apply Cassandra design patterns to data models.

Please check **www.PacktPub.com** for information on our titles

[PACKT] open source
community experience distilled
PUBLISHING

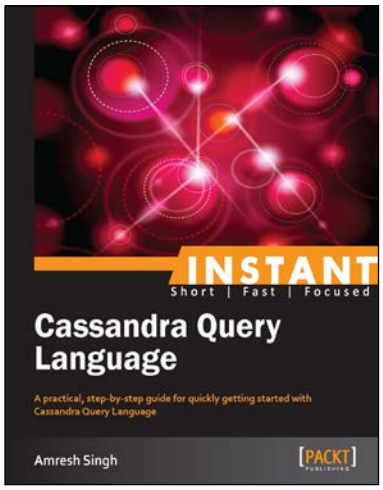


Mastering Apache Cassandra

ISBN: 978-1-78216-268-1 Paperback: 340 pages

Get comfortable with the fastest NoSQL database, its architecture, key programming patterns, infrastructure management, and more!

1. Complete coverage of all aspects of Cassandra.
2. Discusses prominent patterns, pros and cons, and use cases.
3. Contains briefs on integration with other software.



Instant Cassandra Query Language

ISBN: 978-1-78328-271-5 Paperback: 54 pages

A practical, step-by-step guide for quickly getting started with Cassandra Query Language

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.
2. Covers the most frequently used constructs using practical examples.
3. Dive deeper into CQL, TTL, batch operations, and more.
4. Learn how to shed Thrift and adopt a CQL-based binary protocol.