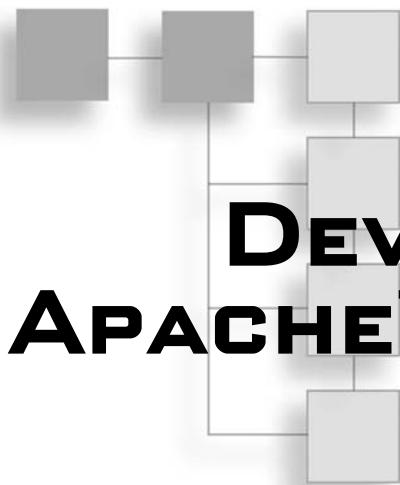


NoSQL Web Development with Apache Cassandra™

Deepak Vohra



NoSQL WEB DEVELOPMENT WITH APACHE™ CASSANDRA™

DEEPAK VOHRA

Cengage Learning PTR



Professional • Technical • Reference

NoSQL Web Development with Apache™ Cassandra™**Deepak Vohra****Publisher and General Manager, Cengage Learning PTR:** Stacy L. Hiquet**Associate Director of Marketing:**
Sarah Panella**Manager of Editorial Services:**
Heather Talbot**Product Manager:** Heather Hurley**Project Editor:** Kate Shoup**Technical Reviewer:** John Yearay**Copy Editor:** Kate Shoup**Interior Layout Tech:** MPS Limited**Cover Designer:** Mike Tanamachi**Proofreader:** Kelly Talbot Editing Services

© 2015 Cengage Learning PTR.

CENGAGE and CENGAGE LEARNING are registered trademarks of Cengage Learning, Inc., within the United States and certain other jurisdictions.

ALL RIGHTS RESERVED. No part of this work covered by the copyright herein may be reproduced, transmitted, stored, or used in any form or by any means graphic, electronic, or mechanical, including but not limited to photocopying, recording, scanning, digitizing, taping, Web distribution, information networks, or information storage and retrieval systems, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the publisher.

For product information and technology assistance, contact us at
Cengage Learning Customer & Sales Support, 1-800-354-9706.For permission to use material from this text or product, submit
all requests online at cengage.com/permissions.Further permissions questions can be emailed to
permissionrequest@cengage.com.

"Apache", "Apache Cassandra", and "Cassandra" are trademarks of the Apache Software Foundation. Used with permission. No endorsement by The Apache Software Foundation is implied by the use of these marks. Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners. Google and the Google logo are registered trademarks of Google, Inc., used with permission. "Eclipse" is a trademark of Eclipse Foundation, Inc. "DataStax" is a trademark of DataStax, Inc. Node.js is a registered trademark of Joyent, Inc. in the United States and other countries. This book is not formally related to or endorsed by the official Joyent Node.js project. MongoDB and Mongo are registered trademarks of MongoDB, Inc. Couchbase is a trademark of Couchbase, Inc. Windows is either registered trademark or trademark of Microsoft Corporation in the United States and/or other countries. All other trademarks are the property of their respective owners.

All images © Cengage Learning unless otherwise noted.

ISBN-13: 978-1-305-57676-6

ISBN-10: 1-305-57676-4

eISBN-10: 1-305-57677-2

Cengage Learning PTR

20 Channel Center Street

Boston, MA 02210

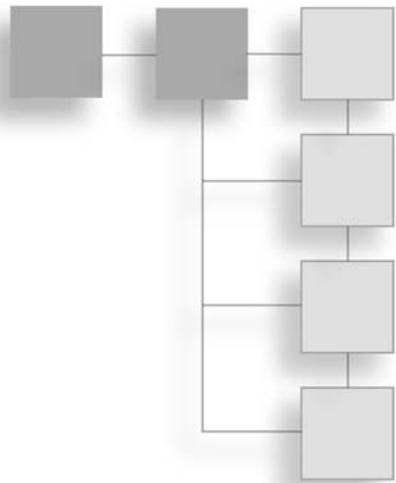
USA

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil, and Japan. Locate your local office at: international.cengage.com/region.

Cengage Learning products are represented in Canada by Nelson Education, Ltd.

For your lifelong learning solutions, visit cengageptr.com.

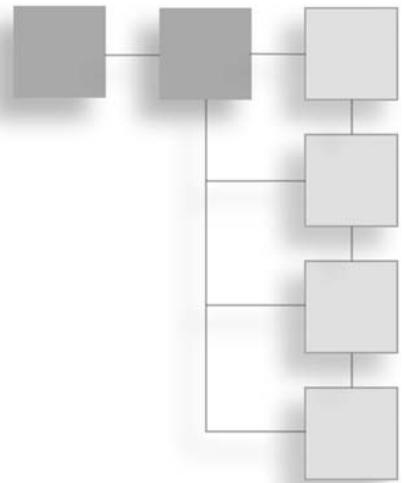
Visit our corporate website at cengage.com.



ABOUT THE AUTHOR

Deepak Vohra is a consultant and a principal member of the NuBean.com software company. Deepak is a Sun Certified Java Programmer and Web Component Developer, and he has worked in the fields of XML, Java programming, and Java EE for more than five years. Deepak is the co-author of *Pro XML Development with Java Technology* and was the technical reviewer for *WebLogic: The Definitive Guide*. Deepak was also the technical reviewer for *Ruby Programming for the Absolute Beginner*. Deepak is the author of *JDBC 4.0 and Oracle JDeveloper for J2EE Development*, *Processing XML Documents with Oracle JDeveloper 11g, EJB 3.0 Database Persistence with Oracle Fusion Middleware 11g*, *Java EE Development with Eclipse*, and *JavaServer Faces 2.0: Essential Guide for Developers*.

CONTENTS



Introduction

PART I JAVA CLIENTS

Chapter 1 Using Cassandra with Hector

Cassandra Storage Model

Overview of Hector Java Client

Setting the Environment

Creating a Java Project

Creating a Cassandra Cluster Object

Creating a Schema

Creating a Keyspace

Creating a Template

Adding Table Data

Adding a Single Column of Data in a Table

Adding Multiple Columns of Data in a Table

Retrieving Table Data

 Querying Single Column

 Querying Multiple Columns

 Querying with a Slice Query

 Querying with the MultigetSliceQuery

 Querying with a Range Slices Query

Updating Data

Deleting Table Data
 Deleting a Single Column
 Deleting Multiple Columns
The HectorClient Class
Summary

Chapter 2 **Querying Cassandra with CQL**

Overview of CQL
Setting the Environment
Creating a Java Project
Creating a Keyspace
Creating a Column Family
Using the INSERT Statement
Using the SELECT Statement
Creating a Secondary Index
Using the SELECT Statement with the WHERE Clause
Using the UPDATE Statement
Using the BATCH Statement
Using the DELETE Statement
Using the ALTER COLUMNFAMILY Statement
Dropping the Column Family
Dropping the Keyspace
The CQLClient Application
New Features in CQL 3
 Compound Primary Key
 Conditional Modifications
Summary

Chapter 3 **Using Cassandra with DataStax Java Driver**

Overview of DataStax Java Driver
Setting the Environment
Creating a Java Project
Creating a Connection
Overview of the Session Class
Creating a Keyspace
Creating a Table
Running the INSERT Statement
Running a SELECT Statement
Creating an Index

- >Selecting with SELECT and a WHERE Filter
- >Running an Async Query
- >Running a PreparedStatement Query
- >Running the UPDATE Statement
- >Running the DELETE Statement
- Running the BATCH Statement
- Dropping an Index
- Dropping a Table
- Dropping a Keyspace
- The CQLClient Application
- Summary

PART II SCRIPTING LANGUAGES

Chapter 4 Using Apache Cassandra with PHP

- An Overview of Phpcassa
- Setting the Environment
 - Installing PHP
 - Installing Phpcassa
- Creating a Keyspace
- Creating a Column Family and Connection Pool
- Adding Data
 - Adding Data in a Batch
 - Retrieving Data
 - Getting Selected Columns
 - Getting Columns from Multiple Rows
 - Getting Column Slices
 - Getting a Range of Rows and Columns
 - Updating Data
 - Deleting Data
 - Dropping the Keyspace and Column Family
- Summary

Chapter 5 Using a Ruby Client with Cassandra

- Setting the Environment
- Installing a Ruby Client with Cassandra
- Creating a Connection
- Creating a Keyspace
- Creating a Column Family
- Adding Data to a Table

- Adding Rows in Batch
- Retrieving Data from a Table
- Selecting a Single Row
- Selecting Multiple Rows
- Iterating over a Result Set
- Selecting a Range of Rows
- Using a Random Partitioner
- Using an Order-Preserving Partitioner
- Getting a Slice of Columns
- Updating Data in a Table
- Deleting Data in a Table
- Updating a Column Family
- Dropping a Keyspace
- Summary

Chapter 6 Using Node.js with Cassandra

- Overview of Node.js Driver for Cassandra CQL
 - The Client Class
 - The Connection Class
- Event-Driven Logging
- Mapping Data Types
- Setting the Environment
 - Creating a Keyspace and a Column Family
 - Installing Node.js
 - Installing Node.js driver for Apache Cassandra
- Creating a Connection with Cassandra
- Adding Data to a Table
- Retrieving Data from a Table
- Filtering the Query
- Querying with a Prepared Statement
- Streaming Query Rows
- Streaming a Field
- Streaming the Result
- Updating Data in Table
- Deleting a Column
- Deleting a Row
- Summary

PART III MIGRATION

Chapter 7 Migrating MongoDB to Cassandra

- Setting the Environment
- Creating a Java Project
- Creating a BSON Document in MongoDB
- Migrating the MongoDB Document to Cassandra
- Summary

Chapter 8 Migrating Couchbase to Cassandra

- Setting the Environment
- Creating a Java Project
- Creating a JSON Document in Couchbase
- Migrating the Couchbase Document to Cassandra
- Summary

PART IV JAVA EE

Chapter 9 Using Cassandra with Kundera

- Setting the Environment
- Creating a JPA Project in Eclipse
- Creating a JPA Entity Class
- Configuring JPA in Persistence.xml
- Creating a JPA Client Class
- Running JPA CRUD Operations
 - Creating a Catalog
 - Finding a Catalog Entry Using the Entity Class
 - Finding a Catalog Entry Using a JPA Query
 - Updating a Catalog Entry
 - Deleting a Catalog Entry
- Summary

Chapter 10 Using Spring Data with Cassandra

- Overview of the Spring Data Cassandra Project
- Setting the Environment
- Creating a Maven Project
- Configuring the Maven Project
- Configuring JavaConfig
- Creating a Model
- Using Spring Data with Cassandra with Template

Finding Out About the Cassandra Cluster

Running Cassandra CRUD Operations

- Save Operations

- Find Operations

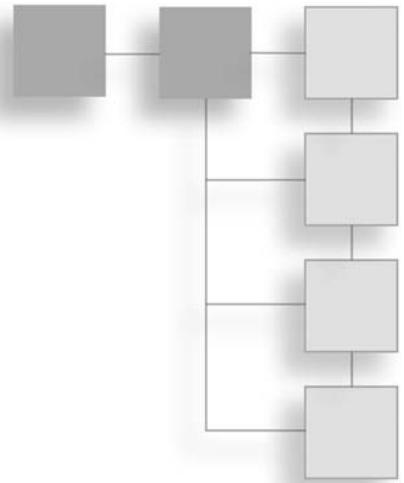
- Exists and Count Operations

- Update Operations

- Remove Operations

Summary

INTRODUCTION



Apache Cassandra is a wide-column data store. Cassandra is a non-relational database, also called a *NoSQL database*. Cassandra is based on a flexible schema. The top-level container of Cassandra is called a *keyspace*, which is equivalent to a schema in a relational database. The top-level data structure for storing data is called a *column family* or a *table*. A column family (or table) consists of columns, a *column* being the smallest increment of data. A Cassandra table is similar to a relational database table in that both have rows and columns. What makes Cassandra different is that the table structure is flexible—that is, not fixed, as in a relational database. Different rows may have different columns. While column metadata for the different columns in a table can be specified in a table definition, the actual data contained in a table is determined by the client application. The schema is flexible in that a row may not contain a particular column or any column at all. Or, a row may include columns not defined in the column's specification. Cassandra is ranked first among wide-column data stores.

Cassandra is a NoSQL data store based on the wide-column data model. NoSQL databases are increasingly replacing relational databases because of their inherent advantages of a flexible schema, ease of use, integration with Web applications, scalability, and integration with Apache Hadoop. Cassandra is ranked second among NoSQL databases. Cassandra is ranked 10th among all databases (relational or non-relational).

While several books on Cassandra administration are available, none are available that cover Cassandra development. This book is about Apache Cassandra Web development. It discusses all aspects of using Cassandra in applications. Java, PHP, Ruby, and

JavaScript are the most commonly used programming/scripting languages, and this book discusses using these languages to access Cassandra. This book also discusses migrating MongoDB server and Couchbase server, two other NoSQL databases, to Cassandra.

The objective of this book is to discuss how a Web developer would develop Web applications with Apache Cassandra. This book covers all aspects of application development, including the following:

- Setting the environment for an application
- Creating sample data
- Running a sample application

WHAT THIS Book COVERS

In Chapter 1, “Using Cassandra with Hector,” you learn how to use the Hector Java client to access Cassandra and create a CRUD (create, read, update, delete) application in the Eclipse IDE.

Chapter 2, “Querying Cassandra with CQL,” introduces the Cassandra Query Language (CQL), which is similar in syntax to SQL. It discusses the **INSERT**, **SELECT**, **UPDATE**, **WHERE**, **BATCH**, and **DELETE** clauses in CQL, with an example. Chapter 2 is based on CQL 2.

Chapter 3, “Using Cassandra with DataStax Java Driver,” discusses using CQL 3 with the Datastax Java client in the Eclipse IDE. In addition to CRUD, it discusses the support for running an **Async** query and a prepared statement query.

In Chapter 4, “Using Apache Cassandra with PHP,” you learn to use a PHP library for Cassandra called phpcassa to connect to Cassandra from a PHP application. You will create a keyspace and column family and add data to the table with PHP. The chapter also discusses adding data in a batch. You will fetch data using the different PHP functions for getting data as multiple columns, column slices, and ranges of columns. The chapter also discusses updating and deleting Cassandra data in a PHP application.

In Chapter 5, “Using a Ruby Client with Cassandra,” you will use a Ruby client with Cassandra to create a keyspace and table (column family) and to add data to the table, including adding data in a batch. You will fetch data in various modes—in single row, in multiple rows, and in a range of rows. The chapter also introduces the ordered partitioner before discussion updating and deleting data and updating or dropping a column family and a keyspace.

Chapter 6, “Using Node.js with Cassandra,” discusses the Node.js driver for Cassandra for connecting to Cassandra and running CRUD operations. It discusses filtering a query and running a prepared query. It also discusses the support for streaming a field and streaming the complete result to a text file.

Chapter 7, “Migrating MongoDB to Cassandra,” discusses migrating a MongoDB document store to Cassandra. MongoDB is a BSON (binary JSON) based document model. First, you will create a document in MongoDB. Then you will use a MongoDB Java client to access MongoDB, fetch data, and migrate the data to Cassandra using the Hector Java client.

Chapter 8, “Migrating Couchbase to Cassandra,” discusses migrating data from Couchbase Server to Cassandra. Couchbase Server is based on the JSON document model. First, you will create a JSON document in Couchbase using the Couchbase Administration Console. Then you will access Couchbase Server using the Couchbase Java client and migrate Couchbase data to Cassandra using the Hector Java client.

Chapter 9, “Using Cassandra with Kundera,” introduces Kundera, a JPA 2.0-compliant object–data store mapping library for NoSQL data stores. In this chapter, you will create a JPA project, a JPA entity class, and a JPA client class in the Eclipse IDE. Then you will configure the object–data store mapping in the persistence.xml file. Finally, you will run CRUD operations on Cassandra using the JPA application.

In Chapter 10, “Using Spring Data with Cassandra,” you will use Apache Cassandra with the Spring Data project. You will create a Maven project in the Eclipse IDE to run CRUD operations on Cassandra with Spring Data.

WHAT YOU NEED FOR THIS Book

This book uses Apache Cassandra 2.04. You can download Apache Cassandra 2 from <http://cassandra.apache.org/download/>. The latest Cassandra version may be used instead of the version in this book. This book also uses the Eclipse IDE for Java EE developers, which you can download from <https://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/lunar>.

Apart from Apache Cassandra, which is used in all the chapters, and the Eclipse IDE, which is used for Java-based applications, the other software required is chapter specific. For example, for migrating Couchbase Server data, Couchbase Server is required.

This book uses the Windows operating system, but if you have Linux installed, this book may still be used (though the source code and samples have not been tested with Linux).

Slight modifications may be required with the Linux install. For example, the directory paths on Linux would be different from the Windows directory paths used in this book. You also need to install Java for Java-based chapters. Java SE 7 is used in this book.

Who This Book Is For

This book's target audience is NoSQL Web developers who want to learn about using Apache Cassandra as a data store. This book is suitable for professional NoSQL developers as well as beginners. This book is also suitable for an intermediate-level or advanced-level course in NoSQL Web development with Cassandra. The target audience is expected to have prior, albeit beginner-level to intermediate-level, knowledge about the languages (Java, PHP, Ruby, and JavaScript) and technologies (Node.js, JPA, Spring Data) used in this book. This book also requires some familiarity with the Eclipse IDE.

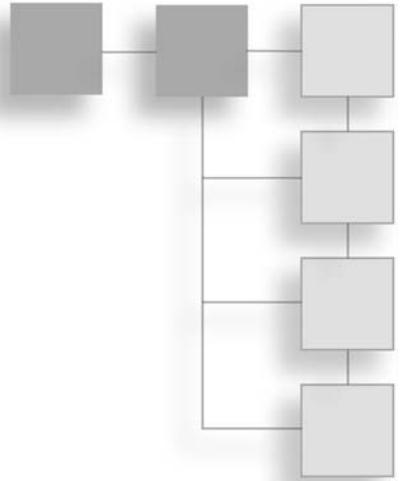
Companion Website Downloads

You may download the companion website files from www.cengageptr.com/downloads.

This page intentionally left blank

PART I

JAVA CLIENTS



This page intentionally left blank

CHAPTER 1



USING CASSANDRA WITH HECTOR

Hector is a Java client used to access Cassandra from a Java or Java EE application. Hector provides several features, which include the following:

- It's suitable for large-scale production systems.
- It offers support for object-oriented and object-relational mapping (ORM).
- It offers enhanced performance using connection pooling.
- It supports round-robin load balancing and client failover.
- It supports fault tolerance using replication of data to multiple nodes.
- It offers elasticity using automatic discovery of hosts.
- It supports automatic retry of downed hosts.
- It is designed for Cassandra's data model.
- It is scalable and highly available.
- It is durable, with no single points of failure.

This chapter discusses using the Hector Java client to access Cassandra in the Eclipse IDE. First, it discusses the Cassandra storage model.

CASSANDRA STORAGE MODEL

Cassandra is a NoSQL, highly available, distributed database based on a row/column structure. NoSQL implies that Cassandra is not a relational database system. Examples of relational database systems are MySQL server, Oracle database, and DB2 database. Relational databases store data in a table structure in rows and columns. A relational database is queried with Structured Query Language (SQL), while a NoSQL database such as Cassandra may be accessed using several different kinds of clients such as Java client, PHP client, and Ruby client, to name a few.

The top-level namespace in Cassandra is a keyspace. A *keyspace* is the equivalent of a database instance in a SQL relational database. An installation of Cassandra may have several keyspaces. The top-level data structure for data storage is a *column family*, which is a set of key-value pairs. A column family definition consists of columns, with one of the columns being the primary key column and the other columns being the data columns. A *column* is the smallest unit of data stored in Cassandra. It is associated with a name, a value and a timestamp.

One of the columns in a column family is the primary key, or row key. A *primary key* is identified with PRIMARY KEY in a column family definition. Some Cassandra APIs require the primary key column to be called KEY, which is the default name for the primary key column. Other Cassandra APIs do not have such a requirement. When an identifier other than KEY is used for the primary key column, a key alias for the primary key is set automatically. The only requirements to define a new column family are a column family name and a primary key and its associated type. The storage model used by Cassandra is shown in Figure 1.1.

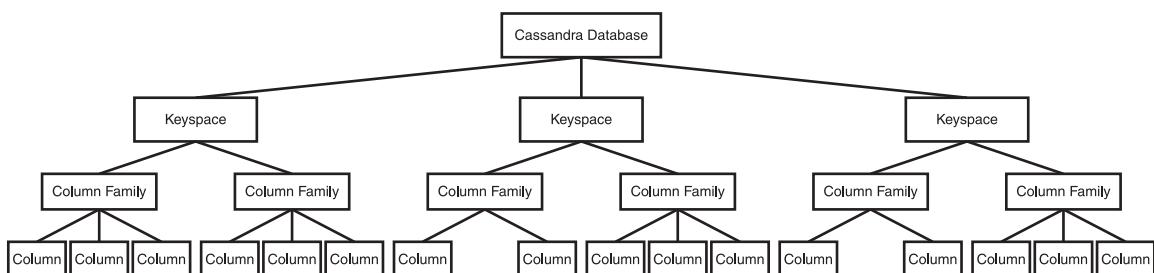


Figure 1.1
Cassandra storage model.

As of Cassandra Query Language (CQL) 3, which is similar to SQL, a column family is also called a table. A key-value pair in a table is also called a record. Column values that

have the same primary key comprise a row, which makes a column family a container of rows, as shown in Figure 1.2. A key-value pair in a column family is the primary key and the row of data (value) associated with a primary key.

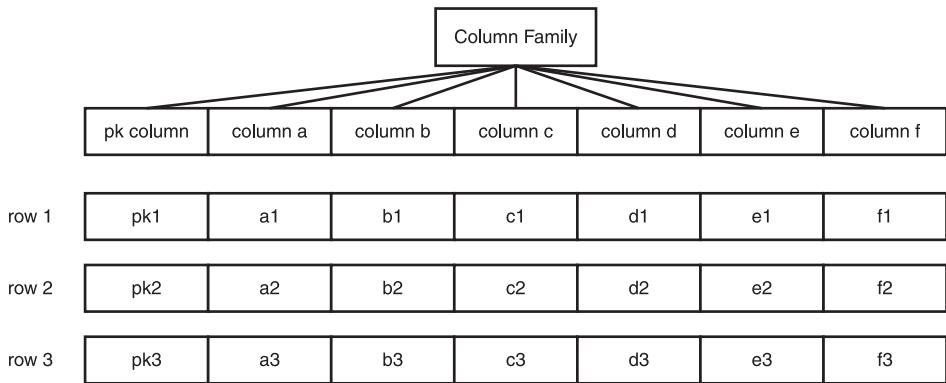


Figure 1.2
Column family as a container of rows.

The primary key must be associated with a data type. Each column may optionally be associated with a data type, which is used during the serialization and de-serialization of data. The different data types supported by the row KEY values and the data columns values are called the *CQL data types*. In fact, a data type may also be associated with a column name, not just the column values. The different data types supported by CQL are discussed in Table 1.1.

Table 1.1 CQL Data Types

CQL Data Type	Description
ascii	A US-ASCII character string.
bigint	A 64-bit signed long integer.
blob	Arbitrary bytes in hexadecimal form.
boolean	A value of true or false.
counter	Used to store a counter value. The counter type is unique in that it should not be assigned to a primary key column and should be used only in a table with counters and the primary key column. Counters are a special kind of columns used to store and count. Counters are stored in dedicated tables.

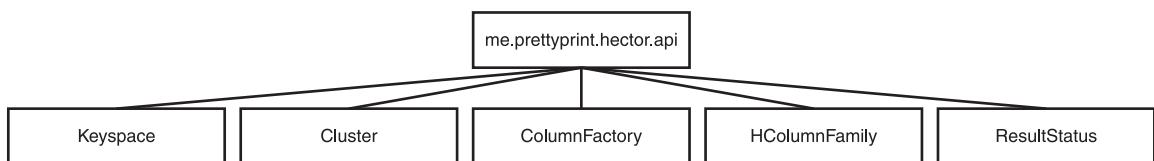
(Continued)

Table 1.1 CQL Data Types (*Continued*)

CQL Data Type	Description
decimal	A variable-precision decimal.
double	A 64-bit IEEE-754 floating point number.
float	A 32-bit IEEE-754 floating point number.
inet	An IPv4 or IPv6 address string.
int	A 32-bit signed integer.
list	A collection of one or more ordered elements.
map	A JSON-style array of literals.
set	A collection of one or more elements.
text	A UTF-8 encoded string.
timestamp	The date and time in epoch time, encoded as an 8-byte string. The epoch time is the number of seconds since January 1, 1970 midnight UTC/GMT (1/1/1970 00:00:00 UTC), not including leap seconds.
timeuuid	A type 1 UUID only.
uuid	A UUID.
varchar	A UTF-8 encoded string.
varint	An arbitrary precision integer.

OVERVIEW OF HECTOR JAVA CLIENT

This section discusses the different packages and classes in the Hector Java client API. The entry points of the Hector API are defined in the `me.prettyprint.hector.api` package, which is illustrated in Figure 1.3.

**Figure 1.3**

Entry points of the Hector API.

The main interfaces in the `me.prettyprint.hector.api` package are discussed in Table 1.2.

Table 1.2 Main Interfaces in the `me.prettyprint.hector.api` Package

Interface	Description
Keyspace	Defines a keyspace
Cluster	Defines a Cassandra cluster of hosts
ColumnFactory	A factory to create columns
HColumnFamily<K, N>	Defines a column family
ResultStatus	Used to track the Cassandra host used for the execution of an operation and the time taken to execute the operation

The serializers used to convert between bytes and different data types are defined in the `me.prettyprint.cassandra.serializers` package, which is illustrated in Figure 1.4.

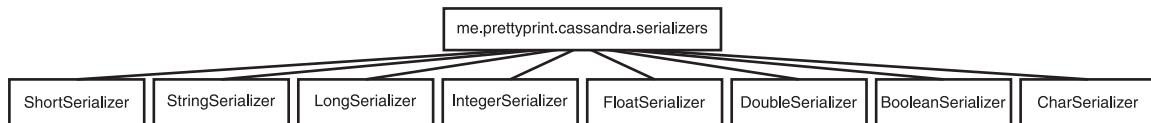


Figure 1.4
Serializers.

The main classes in the `me.prettyprint.cassandra.serializers` package are discussed in Table 1.3.

Table 1.3 Main Classes in the `me.prettyprint.cassandra.serializers` Package

Class	Description
ShortSerializer	A serializer used to convert bytes to and from a short value
StringSerializer	A serializer used to convert bytes to and from a string value using UTF-8 encoding
LongSerializer	A serializer used to convert bytes to and from a long value
IntegerSerializer	A serializer used to convert bytes to and from an integer value

(Continued)

**Table 1.3 Main Classes in the `me.prettyprint.cassandra.serializers` Package
(Continued)**

Class	Description
FloatSerializer	A serializer used to convert bytes to and from a float value
DoubleSerializer	A serializer used to convert bytes to and from a double value
BooleanSerializer	A serializer used to convert bytes to and from a boolean value
CharSerializer	A serializer used to convert bytes to and from a character value

The service interfaces and classes are defined in the `me.prettyprint.cassandra.service` package, which is illustrated in Figure 1.5.

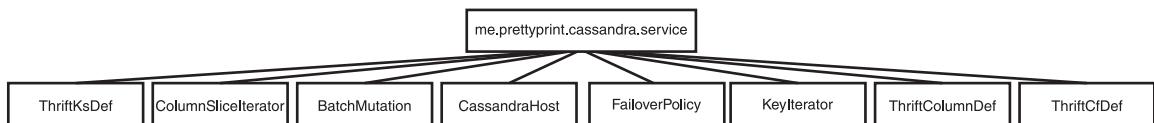


Figure 1.5
Service interfaces.

The main classes in the `me.prettyprint.cassandra.service` package are discussed in Table 1.4.

Table 1.4 Main Classes in the `me.prettyprint.cassandra.service` Package

Class	Description
ThriftKsDef	Defines a keyspace, including its name, strategy class, and replication factor
ColumnSliceIterator	An iterator for a column slice (Column slices are discussed in a later section.)
BatchMutation<K>	Encapsulates a set of insertions and deletions, but not updates
CassandraHost	Encapsulates information required to connect to a Cassandra host including pool configuration parameters

<code>FailoverPolicy</code>	The client policy used if a call to a Cassandra host fails
<code>KeyIterator<K></code>	An iterator over each key in a column family
<code>ThriftColumnDef</code>	Defines a column
<code>ThriftCfDef</code>	Defines a column family

The bean interfaces used to encapsulate columns, column slices, and rows are specified in the `me.prettyprint.hector.api.beans` package, which is illustrated in Figure 1.6.

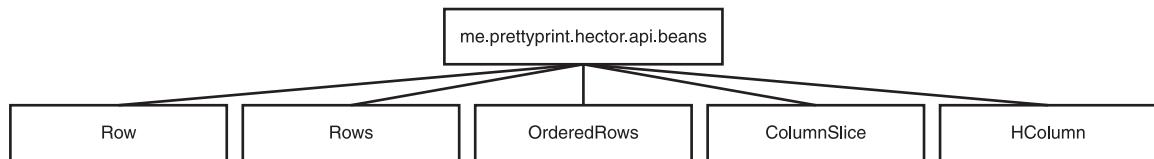


Figure 1.6
Bean interfaces.

The main interfaces in the `me.prettyprint.hector.api.beans` package are discussed in Table 1.5.

Table 1.5 Main Interfaces in the `me.prettyprint.hector.api.beans` Package

Interface	Description
<code>Row<K, N, V></code>	A tuple consisting of a key and a column slice
<code>Rows<K, N, V></code>	A set of rows returned by a multi-get query and consisting of multiple rows
<code>OrderedRows<K, N, V></code>	A set of ordered rows returned by a multi-get query and consisting of multiple rows
<code>ColumnSlice<N, V></code>	Encapsulates a set of columns
<code>HColumn<N, V></code>	Defines a column

The data definition language operations supported by Hector are specified in the `me.prettyprint.hector.api.ddl` package, which is illustrated in Figure 1.7. The package is used for adding and removing new keyspaces and column families, and for defining indices.

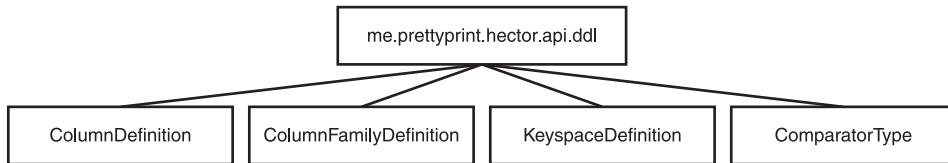


Figure 1.7
DDL classes and interfaces.

The main interfaces and classes in the `me.prettyprint.hector.api.ddl` package are discussed in Table 1.6. DDL operations are performed serially. Concurrent DDL operations are not supported.

Table 1.6 Main Interfaces in the `me.prettyprint.hector.api.ddl` Package

Class or Interface	Description
ColumnDefinition	Provides methods for getting the index name and index type
ColumnFamilyDefinition	Provides methods to perform operations such as setting the keyspace name, column definition, and column type
KeyspaceDefinition	Provides methods for getting and adding to the collection of column family definitions and strategy options associated with a keyspace
ComparatorType	The comparison class used to compare different CQL data types

The exceptions that a Hector client application could throw are specified in the `me.prettyprint.hector.api.exceptions` package, which is illustrated in Figure 1.8.

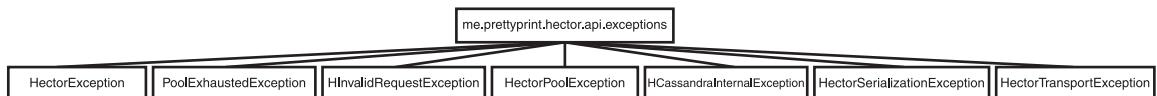


Figure 1.8
Exceptions.

The main exception classes are discussed in Table 1.7.

Table 1.7 Main Classes in the `me.prettyprint.hector.api.exceptions` Package

Class	Description
HectorException	Base exception class for all Hector-related exceptions.
PoolExhaustedException/ HPoolExhaustedException	Thrown if the client pool is exhausted. HPoolExhaustedException since 1.01 version of the API.
HInvalidRequestException	Thrown if the request is invalid.
HectorPoolException	The exception thrown while getting or returning an object to a pool.
HCassandraInternalException	An internal exception thrown by Cassandra.
HectorSerializationException	A serialization exception that could get thrown during the serialization or deserialization of bytes.
HectorTransportException	A Hector transport exception.

The `me.prettyprint.hector.api.factory` package, which is illustrated in Figure 1.9, contains only the `HFactory` class, which is a convenience class with static methods to create keyspaces, column definitions, mutators, columns, and queries, to list a few.

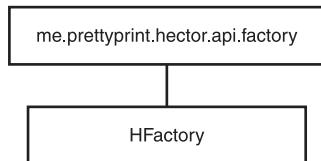


Figure 1.9
Factory Class.

The `me.prettyprint.hector.api.mutation` package contains classes for mutations (insertions, deletions, and such), and is illustrated in Figure 1.10.

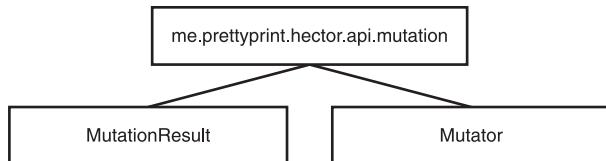


Figure 1.10
Mutation Classes.

The `me.prettyprint.hector.api.mutation` package contains only two classes, which are discussed in Table 1.8.

Table 1.8 Classes in the `me.prettyprint.hector.api.mutation` Package

Class	Description
<code>MutationResult<K></code>	Encapsulates the result from a mutation
<code>Mutator</code>	Used to insert or delete values from a cluster

The different types of queries supported by Hector are defined in the `me.prettyprint.hector.api.query` package interfaces, as illustrated in Figure 1.11.

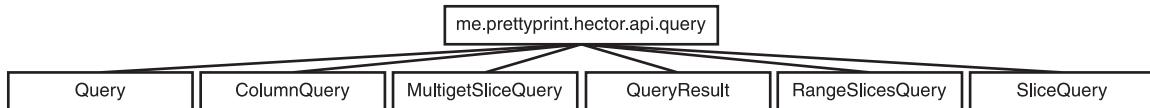


Figure 1.11

Queries.

The main interfaces in the `me.prettyprint.hector.api.query` package are discussed in Table 1.9.

Table 1.9 Main Interfaces in the `me.prettyprint.hector.api.query` Package

Interface	Description
<code>Query<T></code>	The base interface for all Hector queries
<code>ColumnQuery<K,N,V></code>	Used for querying a single and standard column
<code>MultigetSliceQuery<K,N,V></code>	Used for making a multi-get query for a slice of columns
<code>QueryResult<T></code>	The return type for the result of a query
<code>RangeSlicesQuery<K,N,V></code>	A query for a range of column slices
<code>SliceQuery<K,N,V></code>	A query for a slice of columns

Some of the fields, such as keyspace, column family name, key serializer, and column family serializer, are used in every Hector client operation and have to be passed in for every operation separately. The `me.prettyprint.cassandra.service.template` package provides class and interface types to create templates for Hector operations—templates that may be used repeatedly without having to pass in the fields for each operation separately. The `me.prettyprint.cassandra.service.template` package class and interface types are illustrated in Figure 1.12.

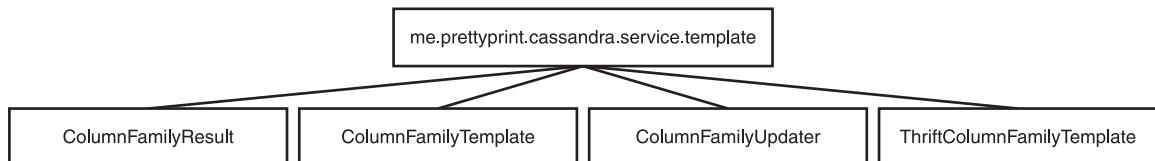


Figure 1.12

Templates.

The class and interfaces in the `me.prettyprint.cassandra.service.template` package are discussed in Table 1.10.

Table 1.10 Class and Interfaces in the `me.prettyprint.cassandra.service.template` Package

Interface	Description
<code>ColumnFamilyResult</code>	A common interface to access the result of a query.
<code>ColumnFamilyTemplate</code>	Defines a template for specifying fields common to all column family operations.
<code>ColumnFamilyUpdater</code>	A common interface for updating a row.
<code>ThriftColumnFamilyTemplate</code>	Thrift-specific implementation of <code>ColumnFamilyTemplate</code> . (Thrift is a service interface definition language. Using it, RPC clients and servers can be built to communicate seamlessly across programming languages.)

In the next section, you will set the environment to access Cassandra from the Hector Java client.

SETTING THE ENVIRONMENT

To set the environment, you must download the following software:

- Apache Cassandra apache-cassandra-2.0.4-bin.tar.gz or a later version from <http://cassandra.apache.org/download/>.
- Hector Java client hector-core-1.1-4.jar or a later version from <http://repo2.maven.org/maven2/org/hectorclient/hector-core/1.1-4/>.
- Eclipse IDE for Java EE developers from <https://eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/kepler>.
- Apache Commons Lang 2.6 from http://commons.apache.org/proper/commons-lang/download_lang.cgi.
- Java SE 6 or later, preferably Java SE 7 or Java SE 8. Java SE 7 is used in this chapter.

Then follow these steps:

1. Install the Eclipse IDE.
2. Extract the Apache Cassandra TAR file to a directory (for example, C:\Cassandra\apache-cassandra-2.0.4).
3. Add the bin folder, C:\Cassandra\apache-cassandra-2.0.4\bin, to the PATH environment variable.
4. Start Apache Cassandra server with the following command:
`cassandra -f`

The Cassandra server starts and begins listening for CQL clients on localhost:9042. Cassandra also listens for Thrift clients on localhost:9160, as shown in Figure 1.13.

```

C:\ Command Prompt - cassandra -f
INFO 16:21:39,410 Node localhost/127.0.0.1 state jump to normal
INFO 16:21:39,427 Startup completed! Now serving reads.
INFO 16:21:39,567 Starting listening for CQL clients on localhost/127.0.0.1:9042...
INFO 16:21:39,645 Using TFramedTransport with a max frame size of 15728640 byte
s.
INFO 16:21:39,710 Binding thrift service to localhost/127.0.0.1:9160
INFO 16:21:39,724 Using synchronous/threadpool thrift server on localhost : 9160
INFO 16:21:39,728 Listening for thrift clients...
INFO 16:21:40,234 Compacted 4 sstables to [var\lib\cassandra\data\system\local
\system-local-jb-9.l. 6.194 bytes to 5,672 (<91% of original) in 1,334ms = 0.00
4055MB/s. 4 total partitions merged to 1. Partition merge counts were {4:1, }

```

Figure 1.13

Starting the Cassandra server.

Source: Microsoft Corporation.

CREATING A JAVA PROJECT

In this section, you will develop a Java project in Eclipse to use the Hector Java client with Cassandra. Follow these steps:

1. Choose File > New > Other in the Eclipse IDE.
2. In the New window, select the Java Project wizard as shown in Figure 1.14. Then click Next.

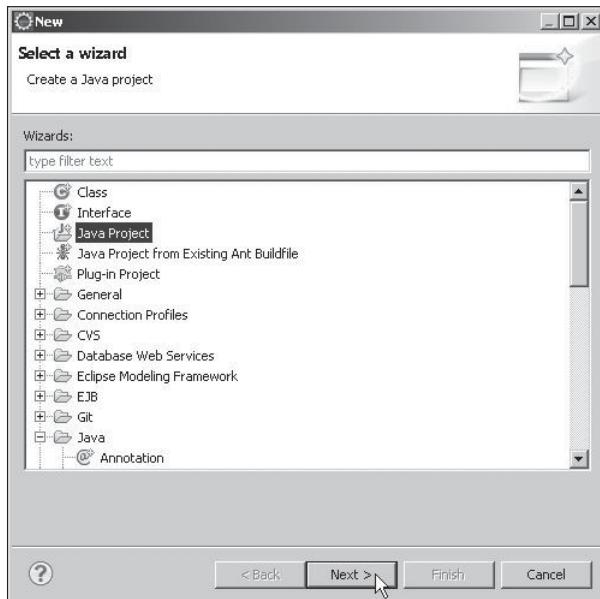


Figure 1.14
Selecting the Java Project wizard.
Source: Eclipse Foundation.

3. In the Create a Java Project screen, specify a project name (Hector) and a directory location for the Java project and click Next. (See Figure 1.15.)

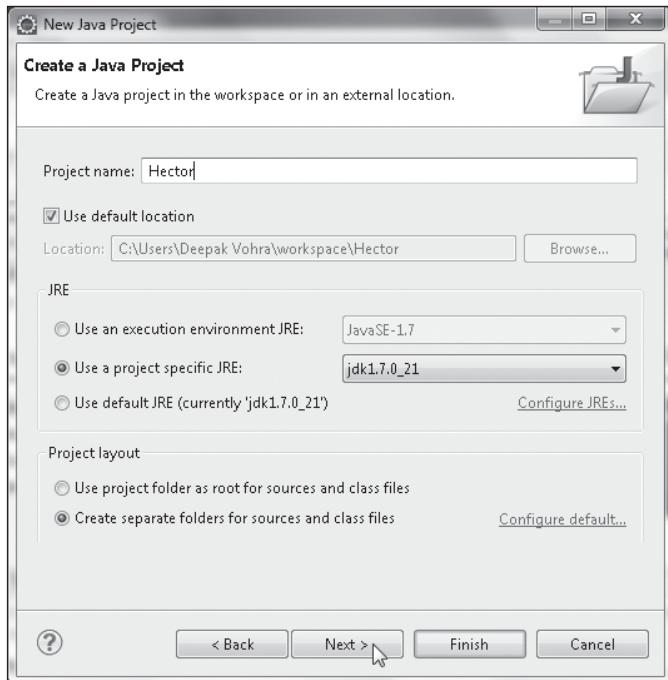
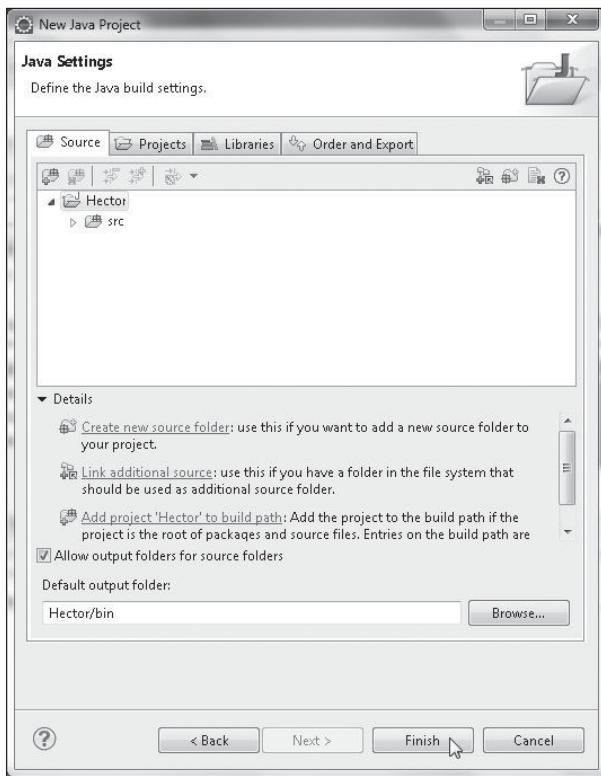


Figure 1.15
Configuring a new Java project.

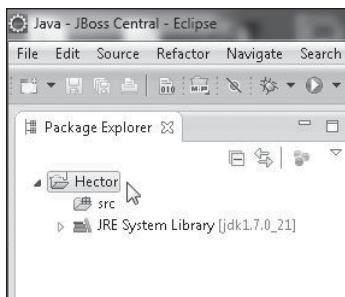
Source: Eclipse Foundation.

4. In the Java Settings dialog box, select the default settings and click Finish, as shown in Figure 1.16. A Java project is created and is added to the Package Explorer, as shown in Figure 1.17.

**Figure 1.16**

Configuring Java settings.

Source: Eclipse Foundation.

**Figure 1.17**

The new Java project.

Source: Eclipse Foundation.

5. Add a Java client class to access Cassandra using Hector. To do so, again choose File > New > Other. This time, however, choose Java > Class in the New window. Then click Next. (See Figure 1.18.)

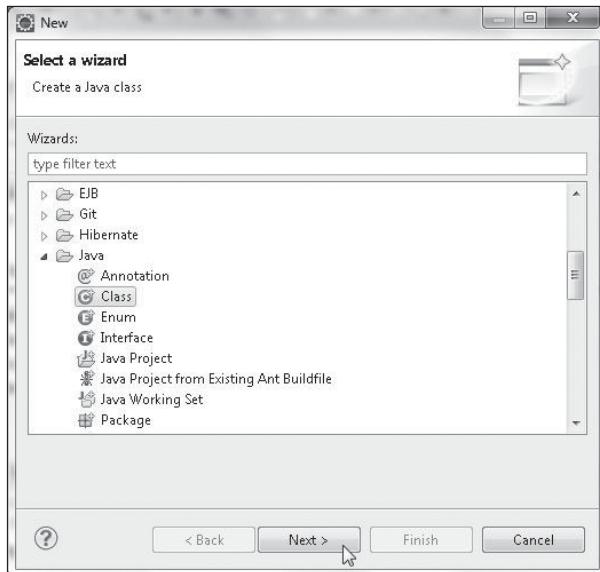


Figure 1.18
Selecting the Java Class wizard.

Source: Eclipse Foundation.

6. In the New Java Class wizard, select a source folder, specify a package (hector), enter a class name (HectorClient), and click Finish, as shown in Figure 1.19. A Java class HectorClient is created, as shown in the Package Explorer in Figure 1.20.

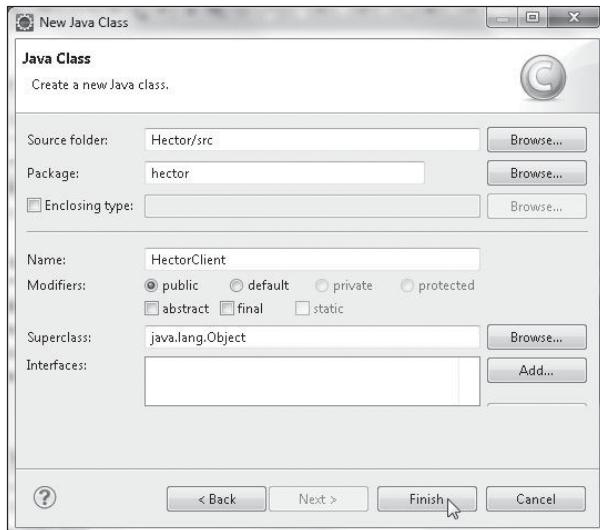
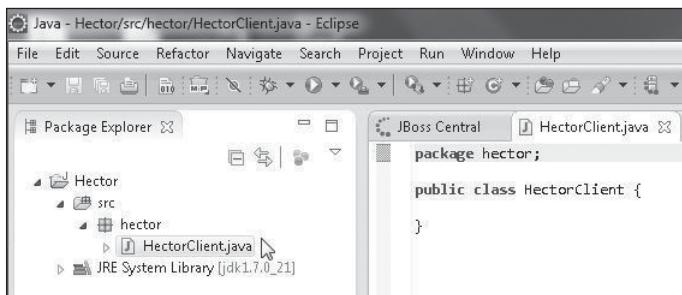


Figure 1.19
Configuring a new Java class.
Source: Eclipse Foundation.

**Figure 1.20**

The new Java class.

Source: Eclipse Foundation.

7. To be able to access Cassandra from the Java application using Hector, you need to add some JAR files to the Java build path of the application. To begin, right-click the Hector project node in the Package Explorer and select Properties.
8. In the Properties window, select the Java Build Path node. Then select Libraries and click Add External JARs to add external JAR files. Add the JAR files listed in Table 1.11.

Table 1.11 JAR Files

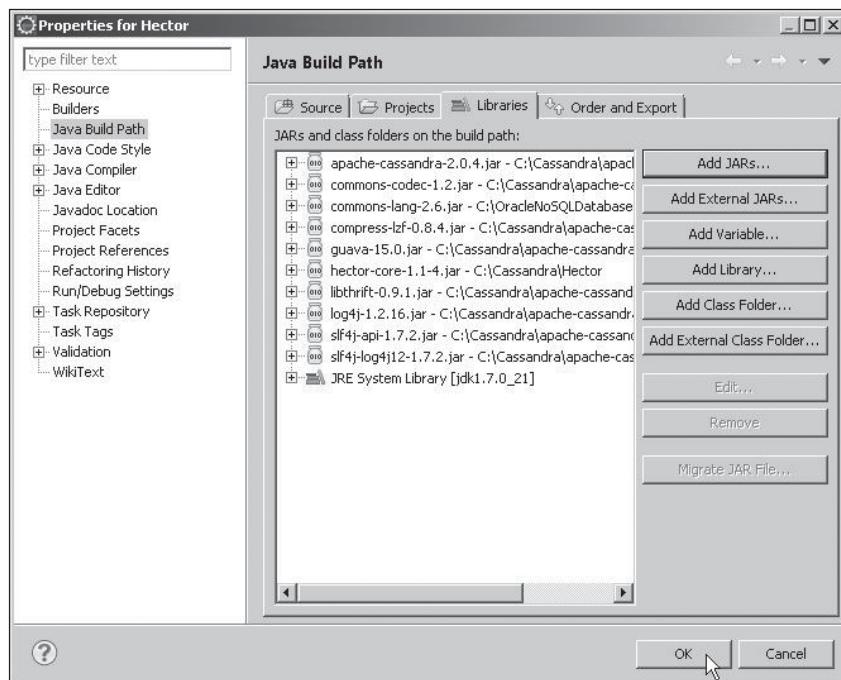
JAR File	Description
apache-cassandra-2.0.4.jar	The Apache Cassandra API
commons-codec-1.2.jar	The Apache Commons Codec file, which provides implementations of common encoders and decoders
commons-lang-2.6.jar	The Apache Commons Lang file, which provides methods for manipulating core classes of the other Java libraries
compress-lzf-0.8.4.jar	The Compression Codec for LZF encoding, which provides encoding/decoding with reasonable compression
guava-15.0.jar	Google's core libraries used in Java projects: collections, caching, primitives support, concurrency, commons annotations, string processing, and I/O to list a few
hector-core-1.1-4.jar	The Hector client API
libthrift-0.9.1.jar	The software framework for scalable cross-language service development

(Continued)

Table 1.11 JAR Files (*Continued*)

JAR File	Description
log4j-1.2.16.jar	A logging library for Java
slf4j-api-1.7.2.jar	The Simple Logging Framework for Java (slf4j), which provides abstraction for various logging frameworks
slf4j-log4j12-1.7.2.jar	Provides the slf4j-log4j binding

9. The external JAR files required for accessing Cassandra from a Hector Java client application are shown in the Eclipse IDE Properties wizard. Click OK after adding the required JAR files, as shown in Figure 1.21.

**Figure 1.21**

Adding JAR files to the Java build path.

Source: Eclipse Foundation.

CREATING A CASSANDRA Cluster OBJECT

The `me.prettyprint.hector.api.Cluster` interface defines a cluster of Cassandra hosts. To be able to access a Cassandra cluster, you must first create a `Cluster` instance for a Cassandra cluster. The `HFactory` class provides several static methods to get or create a `Cluster` instance, as listed in Table 1.12.

Table 1.12 HFactory Class Methods to Create or Get a Cluster

Method	Description
<code>createCluster(String clusterName, CassandraHostConfigurator cassandraHostConfigurator)</code>	Creates a <code>Cluster</code> instance with the given cluster name and configurator if none by the name already exists.
<code>createCluster(String clusterName CassandraHostConfigurator cassandraHostConfigurator, Map<String, String> credentials)</code>	Creates a <code>Cluster</code> instance with the given cluster name, configurator, and credentials if none by the name already exists.
<code>getCluster(String clusterName)</code>	Gets a <code>Cluster</code> instance by the given name.
<code>getOrCreateCluster(String clusterName, CassandraHostConfigurator cassandraHostConfigurator)</code>	Gets or creates a <code>Cluster</code> instance with the specified name and configurator. Gets from the cache if one already exists.
<code>getOrCreateCluster(String clusterName, String hostIp)</code>	Gets or creates a <code>Cluster</code> instance with the specified name and host IP address, which should be in <code>hostname:port</code> format. The <code>hostIp</code> argument may also be provided in <code>ipaddress:port</code> format, but the <code>hostname:port</code> format is preferred. Gets from the cache if one already exists.

In the `HectorClient` class, create a `Cluster` instance using the `getOrCreateCluster(String clusterName, String hostIp)` method as follows:

```
Cluster cluster = HFactory.getOrCreateCluster("hector-cluster", "localhost:9160");
```

Alternatively, you may create a `Cluster` instance as follows:

```
String clusterName = " hector-cluster";  
String host = "localhost:9160";
```

```
Cluster cluster = HFactory.getOrCreateCluster(clusterName, new
CassandraHostConfigurator(host));
```

You'll add a method `createSchema()` to create a column family definition in the next section. You are not expected to build the `HectorClient` class from code snippets. Instead, copy the listing at the end of the discussion.

CREATING A SCHEMA

A schema consists of a column family definition and a keyspace definition. The `HFactory` class provides several static methods to create a column family definition, as listed in Table 1.13.

Table 1.13 HFactory Class Methods to Create a Column Family Definition

Method	Description
<code>createColumnFamilyDefinition(String keyspace, String cfName)</code>	Creates a column family by the specified keyspace name and column family name. Returns a <code>ColumnFamilyDefinition</code> instance.
<code>createColumnFamilyDefinition(String keyspace, String cfName, ComparatorType comparatorType)</code>	Creates a column family by the specified keyspace name and column family name and comparator. Returns a <code>ColumnFamilyDefinition</code> instance.
<code>createColumnFamilyDefinition(String keyspace, String cfName, ComparatorType comparatorType, List<ColumnDefinition> columnMetadata)</code>	Creates a column family by the specified keyspace name, column family name, comparator, and list of column family definitions. Returns a <code>ColumnFamilyDefinition</code> instance.

The `HFactory` class also provides the methods discussed in Table 1.14 to create a keyspace definition.

Table 1.14 HFactory Class Methods to Create a Keyspace Definition

Method	Description
createKeyspaceDefinition(String keyspace)	Creates a KeyspaceDefinition object with the specified keyspace name.
createKeyspaceDefinition(String keyspaceName, String strategyClass, int replicationFactor, List<ColumnFamilyDefinition> cfDefs)	Creates a KeyspaceDefinition object with the specified keyspace name, strategy class, replication factor, and list of column family definitions. The strategy class refers to the strategy used for replica placement across nodes in the cluster. The constant ThriftKsDef.DEF_STRATEGY_CLASS specifies org.apache.cassandra.locator.SimpleStrategy. Replication is the total number of nodes in which data is placed, not the number of other nodes in which data is placed.

Add a method `createSchema()` to create a column family definition and a keyspace definition for the schema. Then create a column family definition for a column family named "catalog", a keyspace named `HectorKeyspace`, and a comparator named `ComparatorType.BYTESTYPE`:

```
ColumnFamilyDefinition cfDef = HFactory.createColumnFamilyDefinition
("HectorKeyspace", "catalog", ComparatorType.BYTESTYPE);
```

Using a replication factor of 1, create a KeyspaceDefinition instance from the preceding column family definition. The replication factor is the number of copies or replicas of each row of data stored in a cluster node. Specify the strategy class as `org.apache.cassandra.locator.SimpleStrategy` using the constant `ThriftKsDef.DEF_STRATEGY_CLASS`:

```
KeyspaceDefinition keyspace = HFactory.createKeyspaceDefinition
("HectorKeyspace", ThriftKsDef.DEF_STRATEGY_CLASS, replicationFactor,
Arrays.asList(cfDef));
```

Cassandra supports the strategy classes, which refer to the replica placement strategy class, discussed in Table 1.15.

Table 1.15 Strategy Classes

Class	Description
org.apache.cassandra.locator.SimpleStrategy	Used for a single data center only. The first replica is placed on a node as determined by the partitioner. Subsequent replicas are placed on the next node/s in a clockwise manner in the ring of nodes without consideration to topology. The replication factor is required only if the SimpleStrategy class is used.
org.apache.cassandra.locator.NetworkTopologyStrategy	Used with multiple data centers. Specifies how many replicas to store in each data center. Attempts to store replicas on different racks within the same data center because nodes in the same rack are more likely to fail together.

Having created a keyspace definition, you need to add the keyspace definition to the Cluster instance. The Cluster interface provides the methods discussed in Table 1.16 to add a keyspace definition.

Table 1.16 Cluster Interface Methods

Method	Description
addKeyspace(KeyspaceDefinition ksdef)	Adds a keyspace definition and does not wait for a schema agreement
addKeyspace(KeyspaceDefinition ksdef, boolean blockUntilComplete)	Adds a keyspace definition and waits for a schema agreement

Add the keyspace definition to the Cluster instance. With the blockUntilComplete set to true, the method blocks until schema agreement is received from the server:

```
cluster.addKeyspace(keyspace, true);
```

Adding a keyspace definition to a Cluster instance does not create a keyspace. In the next section, you will create a keyspace. Invoke the createSchema() method based on whether

the KeyspaceDefinition is not already defined. The Cluster interface provides a method `describeKeyspace(String)` to find out whether a KeyspaceDefinition is already defined. If the method returns null, the KeyspaceDefinition is not defined.

```
KeyspaceDefinition keyspaceDef = cluster.describeKeyspace("HectorKeyspace");
if (keyspaceDef == null) {
    createSchema();
}
```

CREATING A KEYSPACE

Having added a keyspace definition, you need to create a keyspace. A keyspace is represented with the `me.prettyprint.hector.api.Keyspace` interface. The `HFactory` class provides static methods to create a keyspace from a `Cluster` instance to which a keyspace definition has been added. Invoke the method `createKeyspace(String keyspace, Cluster cluster)` to create a keyspace with the name `HectorKeyspace`:

```
private static void createKeyspace() {
    keyspace = HFactory.createKeyspace("HectorKeyspace", cluster);
}
```

CREATING A TEMPLATE

Templates provide a reusable construct containing the fields common to all Hector client operations. Create an instance of `ThriftColumnFamilyTemplate` using a class constructor `ThriftColumnFamilyTemplate(Keyspace keyspace, String columnFamily, Serializer<K> keySerializer, Serializer<N> topSerializer)`. Use the `keyspace` instance created in the preceding section and specify the column family name as "catalog".

```
ThriftColumnFamilyTemplate template = new ThriftColumnFamilyTemplate<String, String>(keyspace, "catalog", StringSerializer.get(), StringSerializer.get());
```

Next, you will add table data to the column family "catalog" in the keyspace `HectorKeyspace`.

ADDING TABLE DATA

As discussed, the `me.prettyprint.hector.api.mutation` package provides the `Mutator` class to add data. First, you need to create an instance of `Mutator` using the static method `createMutator(Keyspace keyspace, Serializer<K> keySerializer)` in `HFactory`. Supply the `keyspace` instance previously created as well as a `StringSerializer` instance.

```
Mutator<String> mutator = HFactory.createMutator(keyspace, StringSerializer.get());
```

Column data may be added as a single column or a batch of columns. We will discuss each of these approaches in the next two sections.

ADDING A SINGLE COLUMN OF DATA IN A TABLE

First, you'll learn how to add a single column of data. The Mutator class provides the method discussed in Table 1.17 to add a single column of data.

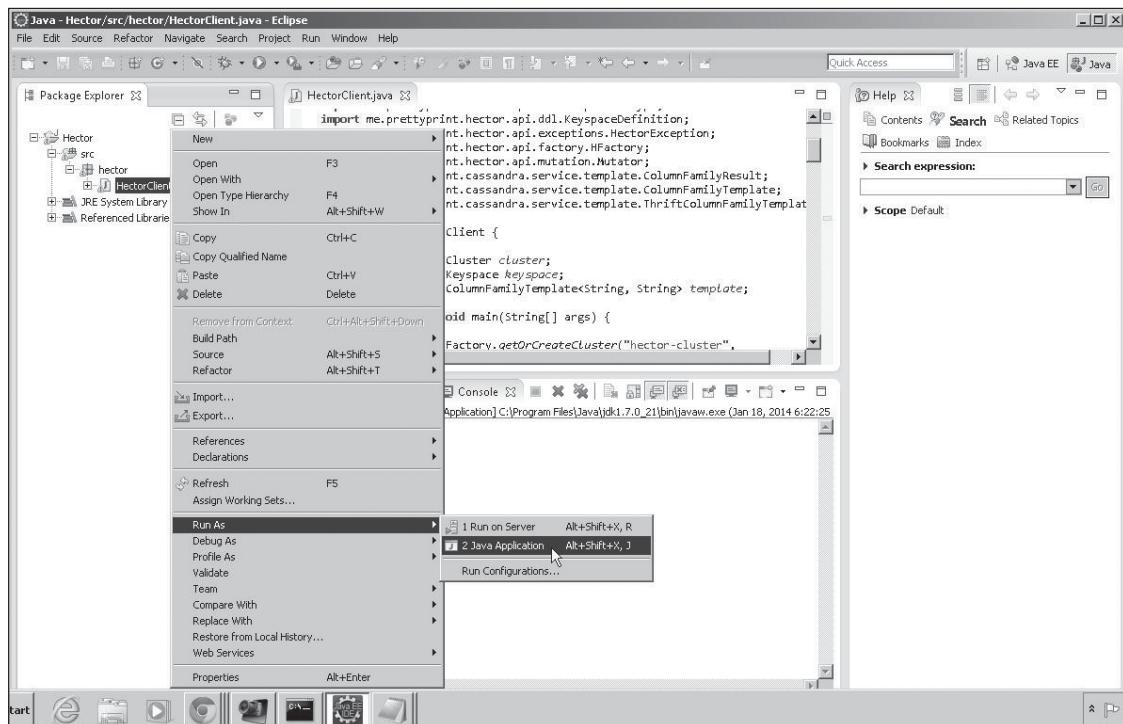
Table 1.17 Mutator Class Method

Method	Description
<code>insert(final K key, final String cf, final HColumn<N,V> c)</code>	Adds a single column as specified in the primary key value, column family name, and HColumn instance. The HColumn instance is the column to be added.

Add a column with the `insert` method using primary key column "catalog3" and the column family name "catalog". Create the `HColumn` instance using the `HFactory` static method `createStringColumn(String name, String value)`.

```
private static void addTableDataColumn() {
    Mutator<String> mutator = HFactory.createMutator(keyspace,
    StringSerializer.get());
    MutationResult result=mutator.insert("catalog3", "catalog",
    HFactory.createStringColumn("journal", "Oracle Magazine"));
    System.out.println(result);
}
```

Output the `MutationResult` returned by the `insert` method. The `HFactory` class also provides several overloaded `createColumn` methods that return an `HColumn` instance. To run the `HectorClient` class and invoke the `addTableDataColumn()` method, add an invocation of the method in the `main` method. To run the class, right-click the `HectorClient` Java file in Package Explorer and select Run As > Java Application, as shown in Figure 1.22.

**Figure 1.22**

Running the HectorClient.java application.

Source: Eclipse Foundation.

A single column is added, as shown by `MutationResult`. The output in Eclipse, shown in Figure 1.23, also has the column added, having been retrieved using a column query, which is discussed later in this chapter.

```

Console Palette
terminated> HectorClient [Java Application] C:\Program Files\Java\jdk1.7.0_21\bin\javaw.exe (Feb 1, 2014, 6:16:43 PM)
log4j:WARN No appenders could be found for logger (me.prettyprint.cassandra.connection.CassandraHostRetryService).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
MutationResult took (14594us) for query (n/a) on host: localhost(127.0.0.1):9160
HColumn(journal=Oracle Magazine)

```

Figure 1.23

Single column added.

Source: Eclipse Foundation.

In the next section, you will add multiple columns.

ADDING MULTIPLE COLUMNS OF DATA IN A TABLE

The Mutator class provides the method discussed in Table 1.18 to add an HColumn instance and return the Mutator instance, which may be used again to add another HColumn instance. You can add a series of HColumn instances by invoking the Mutator instance sequentially.

Table 1.18 Mutator Class Method to Add a Series of Columns

Method	Description
addInsertion(K key, String cf, HColumn<N,V> c)	Adds an HColumn instance using the specified key to the specified column family and returns the Mutator instance, which may be used to invoke the addInsertion method again. Using the method, a batch of HColumn instances can be added. The mutations added to the Mutator instance are not applied until the execute() method is called.

Add a static method addTableData() to make multiple mutations using the same instance of Mutator. Add multiple columns to a Mutator instance using the addInsertion invocations in series.

```
Mutator<String> mutator = HFactory.createMutator(keyspace, StringSerializer
    .get());
mutator.addInsertion("catalog1", "catalog", HFactory.createStringColumn
    ("journal", "Oracle Magazine")).addInsertion("catalog1", "catalog", HFactory.
    createStringColumn("publisher", "Oracle Publishing")).addInsertion
    ("catalog1", "catalog", HFactory.createStringColumn("edition", "November-December
    2013")).addInsertion("catalog1", "catalog", HFactory.createStringColumn
    ("title", "Quintessential and Collaborative")).addInsertion("catalog1",
    "catalog", HFactory.createStringColumn("author", "Tom Haunert"));
```

Instances of HColumn added using the same KEY constitute a row. The preceding example creates a row of data with KEY "catalog1" in the "catalog" column family. Another row with KEY "catalog2" could be added similarly.

```
mutator.addInsertion("catalog2", "catalog", HFactory.createStringColumn
    ("journal", "Oracle Magazine"))
    .addInsertion("catalog2", "catalog", HFactory.createStringColumn
    ("publisher", "Oracle Publishing")).addInsertion("catalog2", "catalog", HFactory.
    createStringColumn("edition", "November-December 2013")).addInsertion
```

```
("catalog2", "catalog", HFactory.createStringColumn("title", "Engineering as a Service")).addInsertion("catalog2", "catalog", HFactory.createStringColumn("author", "David A. Kelly"));
```

The mutations added to the Mutator instance are not sent to the Cassandra server yet. To send them, you invoke the `execute()` method. This runs the batch of mutations added to the Mutator instance.

```
mutator.execute();
```

Invoke the `addTableData()` method from the `main` method and run the `HectorClient` class to add data in a batch.

RETRIEVING TABLE DATA

In this section, you will retrieve the previously added table data. As discussed, the `me.prettyprint.hector.api.query` package provides several interfaces representing different types of queries. First, you will query a single column.

Querying Single Column

The `ColumnQuery<K,N,V>` interface represents a single standard column query. `HFactory` provides the methods discussed in Table 1.19 to query a single column.

Table 1.19 HFactory Methods to Query a Single Column

Method	Description
<code>createColumnQuery(Keyspace keyspace, Serializer<K> keySerializer, Serializer<N> nameSerializer, Serializer<V> valueSerializer)</code>	Returns an instance of <code>ColumnQuery</code> when supplied with a <code>Keyspace</code> instance and serializers for key, column name, and value.
<code>createStringColumnQuery(Keyspace keyspace)</code>	Returns a <code>ColumnQuery<String, String, String></code> instance when supplied with a <code>Keyspace</code> instance. Uses <code>StringSerializer</code> instances for key, column name, and value.

Create a `ColumnQuery` instance using the static method `createStringColumnQuery` (`Keyspace keyspace`):

```
ColumnQuery<String, String, String> columnQuery = HFactory.createStringColumnQuery(keyspace);
```

The `ColumnQuery` interface provides the methods discussed in Table 1.20 to set the fields of the query, each of which return a `ColumnQuery<K,N,V>` instance.

Table 1.20 HFactory Methods to Query a Single Column

Method	Description
<code>setKey(K key)</code>	Sets the primary key value
<code>setName(N name)</code>	Sets the column name
<code>setColumnFamily(String cf)</code>	Sets the column family name

Set the column family name to "catalog", the primary key value to "catalog3", and the column name to "journal":

```
private static void retrieveTableDataColumnQuery() {
    columnQuery.setColumnFamily("catalog").setKey("catalog3").setName("journal");
}
```

The `QueryResult<T>` interface represents the return type from queries, with the type parameter `T` being the type of result. After setting the query attributes, invoke the `execute()` method to return a `QueryResult<HColumn<String, String>>` object.

```
QueryResult<HColumn<String, String>> result = columnQuery.execute();
```

Next, output the result value using the method `get()` in the `QueryResult` interface:

```
System.out.println(result.get());
```

Finally, invoke the `retrieveTableDataColumnQuery()` method from the `main` method to output the result of the query, as shown in Figure 1.24.

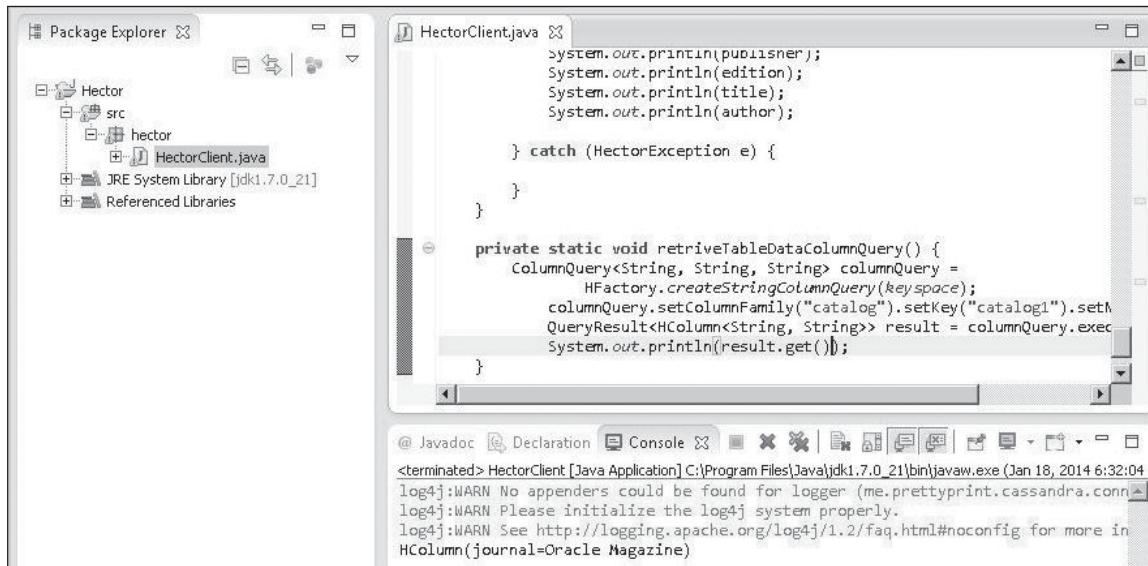


Figure 1.24

The result of the query.

Source: Eclipse Foundation.

Querying Multiple Columns

In this section, you will query multiple columns using an instance of `ThriftColumnFamilyTemplate`. This provides a reusable template with the common query attributes set to make repeated Hector queries. You created an instance of `ThriftColumnFamilyTemplate` in an earlier section. The `ThriftColumnFamilyTemplate` class provides several overloaded methods called `queryColumns` to query multiple columns in the same query, as discussed in Table 1.21.

Table 1.21 Overloaded queryColumns Methods to Query Multiple Columns

Method	Description
queryColumns(K key)	Queries the columns in the row corresponding to the provided key value.
queryColumns(Iterable<K> keys)	Queries the columns in the rows corresponding to the provided iterable of key values.

(Continued)

Table 1.21 Overloaded queryColumns Methods to Query Multiple Columns (Continued)

Method	Description
queryColumns(K key, List<N> columns)	Queries the columns in the row corresponding to the provided key value. Only the columns in the provided list are queried.
queryColumns(Iterable<K> keys, List<N> columns)	Queries the columns in the rows corresponding to the provided iterable of key values. Only the columns in the provided list are queried.

Each of the methods in Table 1.21 returns a `ColumnFamilyResult` instance. Add a `retrieveTableData()` method to query multiple columns. Using the template, query the columns in the row corresponding to the "catalog1" key.

```
ColumnFamilyResult<String, String> res = template.queryColumns("catalog1");
```

The `ColumnFamilyResult` interface provides several get methods to get the different types of results, as discussed in Table 1.22.

Table 1.22 ColumnFamilyResult Interface Methods

Method	Description
getUUID(N columnName)	Returns a UUID value given a column name
getString(N columnName)	Returns a string value given a column name
getLong(N columnName)	Returns a long value given a column name
getInteger(N columnName)	Returns an integer value given a column name
getFloat(N columnName)	Returns a float value given a column name
getDouble(N columnName)	Returns a double value given a column name
getBoolean(N columnName)	Returns a boolean value given a column name
getByteArray(N columnName)	Returns a byte[] value given a column name
getDate(N columnName)	Returns a date value given a column name
getColumnNames()	Returns a collection of column names
getColumn(N columnName)	Returns an HColumn instance given a column name

You can use the `hasResults()` method to find out whether a `ColumnFamilyResult` instance has a result. Output the String column values in the `ColumnFamilyResult` instance obtained from the preceding query.

```
if(res.hasResults()){
    String journal = res.getString("journal");
    String publisher = res.getString("publisher");
    String edition = res.getString("edition");
    String title = res.getString("title");
    String author = res.getString("author");

    System.out.println(journal);
    System.out.println(publisher);
    System.out.println(edition);
    System.out.println(title);
    System.out.println(author);
}
```

Similarly, query the columns corresponding with the row with the "catalog2" key and output the result. Invoke the `retrieveTableData()` method in the `main` method and run the `HectorClient` class to output the query result, as shown in Figure 1.25.

```
@ Javadoc Declaration Console <> X X | <> <> <> <> <> <> <> <> <>
<terminated> HectorClient [Java Application] C:\Program Files\Java\jdk1.7.0_21\bin\javaw.exe (Jan 18, 2014 6:22:25
log4j:WARN No appenders could be found for logger (me.prettyprint.cassandra.con
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more in
Oracle Magazine
Oracle Publishing
November-December 2013
Quintessential and Collaborative
Tom Haunert
Oracle Magazine
Oracle Publishing
November-December 2013
Engineering as a Service
David A. Kelly
```

Figure 1.25

The query result for multiple columns.

Source: Eclipse Foundation.

Querying with a Slice Query

A *slice query* is a query of only a slice of columns—that is, columns that are either specified or in a certain range indicated. A set of columns is represented with the `ColumnSlice<N,V>` interface. A slice query is represented with the `SliceQuery<K,N,V>` interface.

The `SliceQuery<K,N,V>` interface provides the methods discussed in Table 1.23 to set the attributes of the query.

Table 1.23 SliceQuery Interface Methods

Method	Description
<code>setKey(K key)</code>	Sets the key value for the row from which the columns are to be queried.
<code>setColumnNames(N... columnNames)</code>	Sets the column names using a vararg method.
<code>setRange(N start, N finish, boolean reversed, int count)</code>	Sets a range of columns with a start and a finish. The reversed parameter of type boolean indicates whether columns are to be queried in reverse order. The count parameter of type int indicates the number of columns to query.
<code>setColumnFamily(String cf)</code>	Sets the column family to query.

Add a `retrieveTableDataSliceQuery()` method to the query using a slice query. The `HFactory` class provides the method discussed in Table 1.24 to create a `SliceQuery` instance.

Table 1.24 HFactory Class Method to Create a SliceQuery Instance

Method	Description
<code>createSliceQuery(Keyspace keyspace, Serializer<K> keySerializer, Serializer<N> nameSerializer, Serializer<V> valueSerializer)</code>	Creates a <code>SliceQuery</code> given a Keyspace instance and serializers for key, column name, and column value

Using the Keyspace instance previously created, create a `SliceQuery<String, String>` instance using the `createSliceQuery()` method. Set the column family as "catalog" and set the row key as "catalog2". Use `StringSerializer` instances for the column name, key, and column value.

```
SliceQuery<String, String, String> query = HFactory.createSliceQuery(keyspace,
StringSerializer.get(), StringSerializer.get(), StringSerializer.get()).setKey(
("catalog2").setColumnFamily("catalog");
```

The `ColumnSliceIterator` class is used to iterate over the columns in a `SliceQuery` instance and to retrieve the column values. The `ColumnSliceIterator` class provides the constructors discussed in Table 1.25.

Table 1.25 ColumnSliceIterator Class Constructors

Constructor	Description
<code>ColumnSliceIterator(SliceQuery<K,N,V> query, N start, final N finish, boolean reversed)</code>	The query parameter is the base slice query to execute. The start and finish are the start and finish points of the range of columns. The reversed boolean indicates whether the columns are to be queried in reverse order.
<code>ColumnSliceIterator(SliceQuery<K,N,V> query, N start, final N finish, boolean reversed, int count)</code>	In addition to the parameters for the preceding, the method includes the count parameter for the number of columns to query.
<code>ColumnSliceIterator(SliceQuery<K,N,V> query, N start, ColumnSliceFinish finish, boolean reversed)</code>	In addition to the query attributes specified for the first method in the table, the method includes a parameter of type <code>ColumnSliceFinish</code> , which allows for a user-defined function that returns a new finish point.
<code>ColumnSliceIterator(SliceQuery<K,N,V> query, N start, ColumnSliceFinish finish, boolean reversed, int count)</code>	In addition to the parameters for the preceding, the method includes the count parameter for the number of columns to query.

Create a `ColumnSliceIterator` instance using a `start` for the column name of "\u0000", which is the smallest value of type `char`, and using a `finish` of "\uFFFF", the largest value of type `char`. Specify the `SliceQuery` instance and set the `reversed` parameter to `false`.

```
ColumnSliceIterator<String, String, String> iterator = new
ColumnSliceIterator<String, String, String>(query, "\u0000", "\uFFFF", false);
```

Then iterate over the columns to get the column name and column value for each of the columns.

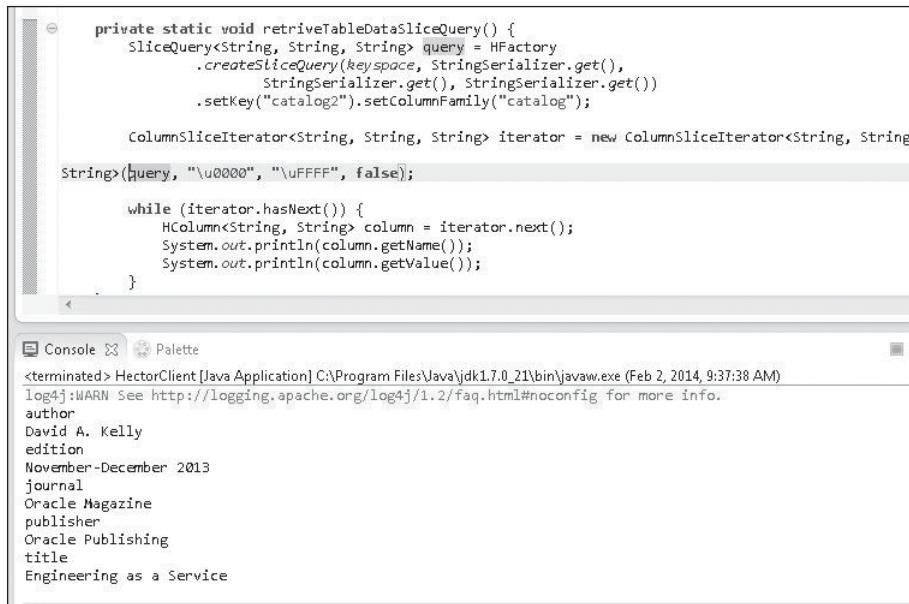
```
while (iterator.hasNext()) {
    HColumn<String, String> column = iterator.next();
```

```

        System.out.println(column.getName());
        System.out.println(column.getValue());
    }
}

```

Invoke the `retrieveTableDataSliceQuery()` method from the `main` method to output the column names and column values, as shown in Eclipse in Figure 1.26, when the `HectorClient` application is run.



The screenshot shows the Eclipse IDE interface with the Java code for the `retrieveTableDataSliceQuery()` method in the editor. Below the code, the Eclipse Console window displays the execution results of the application. The log output shows the following data:

```

<terminated> HectorClient [Java Application] C:\Program Files\Java\jdk1.7.0_21\bin\javaw.exe (Feb 2, 2014, 9:37:38 AM)
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
author
David A. Kelly
edition
November-December 2013
journal
Oracle Magazine
publisher
Oracle Publishing
title
Engineering as a Service

```

Figure 1.26
Query result for SliceQuery.
Source: Eclipse Foundation.

Querying with the MultigetSliceQuery

In the preceding section, you queried multiple columns from only a single row. In this section, you will query columns from multiple rows. The `MultigetSliceQuery<K,N,V>` interface is used for a query over multiple rows. The `MultigetSliceQuery<K,N,V>` interface provides the methods discussed in Table 1.26 to set and get query fields.

Table 1.26 MultigetSliceQuery Interface Methods

Method	Description
setKeys(K... keys)	Sets the row keys using a vararg method
setKeys(Iterable<K> keys)	Sets the row keys as an iterable
setColumnNames(N... columnNames)	Sets column names using a vararg method
getColumnNames()	Gets column names as a collection
setColumnFamily(String cf)	Sets the column family name
setRange(N start, N finish, boolean reversed, int count)	Sets the range of column names and the number of columns and a boolean arg to indicate if the column order is to be reversed

All the methods in Table 1.26 return a `MultigetSliceQuery` instance except the `getColumnNames()` method. First, however, you need to create an instance of `MultigetSliceQuery`. The `HFactory` class provides the method discussed in Table 1.27 to create an instance of `MultigetSliceQuery`.

Table 1.27 HFactory Class Method to Create a MultigetSliceQuery Instance

Method	Description
<code>createMultigetSliceQuery(Keyspace keyspace, Serializer<K> keySerializer, Serializer<N> nameSerializer, Serializer<V> valueSerializer)</code>	Returns a <code>MultigetSliceQuery</code> instance when a <code>Keyspace</code> instance and serializers for key, column name, and column value are supplied

Add a `retrieveTableDataMultigetSliceQuery()` method to the query using a multi-get query. Using the `Keyspace` instance created earlier and `StringSerializer` instances, create an instance of `MultigetSliceQuery<String, String, String>` using the `HFactory` method `createMultigetSliceQuery`.

```
MultigetSliceQuery<String, String, String> multigetSliceQuery =
HFactory.createMultigetSliceQuery(keyspace, StringSerializer.get(),
StringSerializer.get(), StringSerializer.get());
```

Next, set the column family as "catalog" and row keys as "catalog1", "catalog2", and "catalog3".

```
multigetSliceQuery.setColumnFamily("catalog");
multigetSliceQuery.setKeys("catalog1", "catalog2",
"catalog3");
```

Set the range of columns with the `setRange` method. Empty strings for start and finish imply that all the columns are to be queried. Set the number of columns to get to 5 and set the reversed boolean to false.

```
multigetSliceQuery.setRange("", "", false, 5);
```

Next, invoke the `execute()` method on the `MultigetSliceQuery<String, String>` instance to get the query result as a `QueryResult<Rows<String, String, String>>` instance.

```
QueryResult<Rows<String, String, String>> result = multigetSliceQuery.execute();
```

Get the result value using the `get()` method in the `QueryResult` interface. The type of the result is `Rows<String, String, String>`. Get each of the Row instances in Rows using the `getByKey(K key)` method. The `Row<K,N,V>` interface is a tuple consisting of a Key and a column slice.

```
System.out.println(result.get().getByKey("catalog1"));
System.out.println(result.get().getByKey("catalog2"));
System.out.println(result.get().getByKey("catalog3"));
```

Invoke the `retrieveTableDataMultigetSliceQuery()` method from the `main` method to output the result of the `multigetSliceQuery` instance, as shown in Figure 1.27.

```
@ Javadoc Declaration Console 
@terminated> HectorClient [Java Application] C:\Program Files\Java\jdk1.7.0_21\bin\javaw.exe (Jan 18, 2014 7:27:04 PM)
log4j:WARN No appenders could be found for logger (me.prettyprint.cassandra.connection.CassandraHostRetryService).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Row(catalog1,ColumnSlice([HColumn(author=Tom Haunert), HColumn(edition=November-December 2013), HColumn(journal=Oracle Magazine), HColumn(publisher=Oracle Publishing), HColumn(title=Quintessential and Collaborative)]))
Row(catalog2,ColumnSlice([HColumn(author=David A. Kelly), HColumn(edition=November-December 2013), HColumn(journal=Oracle Magazine), HColumn(publisher=Oracle Publishing), HColumn(title=Engineering as a Service)]))
Row(catalog3,ColumnSlice([HColumn(journal=Oracle Magazine)]))
```

Figure 1.27

Query result for the `multigetSliceQuery` instance.

Source: Eclipse Foundation.

In another run of the application, set the number of columns in the query to 3.

```
multigetSliceQuery.setRange("", "", false, 3);
```

As shown in Figure 1.28, only three of the columns are included in the query result.

```
@ Javadoc Declaration Console 
<terminated> HectorClient [Java Application] C:\Program Files\Java\jdk1.7.0_21\bin\javaw.exe (Jan 18, 2014 7:24:11 PM)
log4j:WARN No appenders could be found for logger (me.prettyprint.cassandra.connection.CassandraHostRetryService).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Row(catalog1,ColumnSlice([HColumn(author=Tom Haunert), HColumn(edition=November-December 2013), HColumn(journal=Oracle Magazine)]))
Row(catalog2,ColumnSlice([HColumn(author=David A. Kelly), HColumn(edition=November-December 2013), HColumn(journal=Oracle Magazine)]))
Row(catalog3,ColumnSlice([HColumn(journal=Oracle Magazine)]))
```

Figure 1.28

Query result for `multigetSliceQuery` instance for three columns.

Source: Eclipse Foundation.

Querying with a Range Slices Query

The `MultigetSliceQuery` interface discussed in the preceding section sets the row keys for which columns are to be retrieved explicitly. Alternatively, you can use the `RangeSlicesQuery<K,N,V>` interface to set the row keys as a range instead of setting each key explicitly. For example, if row key values "catalog1", "catalog2", "catalog3", "catalog4", and "catalog5" are defined, you could set the range to start at "catalog1" and end at "catalog5" to include all the row key values in between. Some of the methods in the `RangeSlicesQuery<K,N,V>` interface are discussed in Table 1.28.

Table 1.28 RangeSlicesQuery Interface Methods

Method	Description
<code>setKeys(K start, K end)</code>	Sets the row keys
<code>setRowCount(int rowCount)</code>	Sets the row count
<code>setColumnNames(N... columnNames)</code>	Sets the column names
<code>setColumnFamily(String cf)</code>	Sets the column family
<code>setRange(N start, N finish, boolean reversed, int count)</code>	Sets the range of columns with a start and finish, a boolean to indicate if the order of columns is to be reversed, and an int count of the columns

Add a `retrieveTableDataRangeSlicesQuery()` method to use the `RangeSlicesQuery<K,N,V>` interface. The `HFactory` class provides the method discussed in Table 1.29 to create a `RangeSlicesQuery` instance.

Table 1.29 HFactory Class Method to Create a RangeSlicesQuery Instance

Method	Description
<code>createRangeSlicesQuery(Keyspace keyspace, Serializer<K> keySerializer, Serializer<N> nameSerializer, Serializer<V> valueSerializer)</code>	Returns a <code>RangeSlicesQuery</code> instance when supplied with a <code>Keyspace</code> instance and serializers for key, column name, and column value

Using `StringSerializer` instances, create a `RangeSlicesQuery<String, String, String>` instance using the `HFactory` method `createRangeSlicesQuery`.

```
RangeSlicesQuery<String, String, String> rangeSlicesQuery =HFactory.  
createRangeSlicesQuery(keyspace, StringSerializer.get(),  
StringSerializer.get(), StringSerializer.get());
```

Next, set the column family to "catalog" and set the range of keys to start at "catalog1" and end at "catalog3".

```
rangeSlicesQuery.setColumnFamily("catalog");  
rangeSlicesQuery.setKeys("catalog1", "catalog3");
```

Set the range of columns to include all the columns as indicated by the empty strings for start and finish. Set the number of columns to get to 5.

```
rangeSlicesQuery.setRange("", "", false, 5);
```

Next, invoke the `execute()` method on the `RangeSlicesQuery<String, String, String>` instance to make the query. The result is returned as a `QueryResult<0rderedRows<String, String, String>>` instance.

```
QueryResult<0rderedRows<String, String, String>> result = rangeSlicesQuery.  
execute();
```

Invoke the `get()` method on the `QueryResult` instance to get the result value. Then invoke the `getByKey` method on each of the `Row` instances to get the row retrieved.

```
System.out.println(result.get().getByKey("catalog1"));  
System.out.println(result.get().getByKey("catalog2"));  
System.out.println(result.get().getByKey("catalog3"));
```

Invoke the `retrieveTableDataRangeSlicesQuery()` method in the `main` method and run the `HectorClient` class to output the result. The result of the query as output in Eclipse is shown in Figure 1.29.



```

private static void retrieveTableDataRangeSlicesQuery() {
    RangeSlicesQuery<String, String, String> rangeSlicesQuery =
        HFactory.createRangeSlicesQuery(keyspace, StringSerializer.get(), StringSerializer.get(), StringSerializer.get());
    rangeSlicesQuery.setColumnFamily("catalog");
    rangeSlicesQuery.setKeys("catalog1", "catalog3");
    rangeSlicesQuery.setRange("", "", false, 5);
    QueryResult<OrderedRows<String, String, String>> result = rangeSlicesQuery.execute();
    System.out.println(result.get().getByKey("catalog1"));
    System.out.println(result.get().getByKey("catalog2"));
    System.out.println(result.get().getByKey("catalog3"));
}

```

@ Javadoc Declaration Console

<terminated> HectorClient [Java Application] C:\Program Files\Java\jdk1.7.0_21\bin\javaw.exe (Jan 18, 2014 7:35:01 PM)

log4j:WARN No appenders could be found for logger (me.prettyprint.cassandra.connection.CassandraHostRetryService).

log4j:WARN Please initialize the log4j system properly.

log4j:WARN See <http://logging.apache.org/log4j/1.2/faq.html#noconfig> for more info.

Row(catalog1,ColumnSlice([HColumn(author=Tom Haunert), HColumn(edition=November-December 2013), HColumn(journal=Oracle Magazine), HColumn(publisher=Oracle Publishing), HColumn(title=Quintessential and Collaborative)]))

Row(catalog2,ColumnSlice([HColumn(author=David A. Kelly), HColumn(edition=November-December 2013), HColumn(journal=Oracle Magazine), HColumn(publisher=Oracle Publishing), HColumn(title=Engineering as a Service)]))

Row(catalog3,ColumnSlice([HColumn(journal=Oracle Magazine)]))

Figure 1.29

Query result for a `RangeSlicesQuery` instance.

Source: Eclipse Foundation.

UPDATING DATA

In this section, you will update row data added previously. The `ColumnFamilyUpdater<K,N>` class is used to update a row of data and provides the constructors discussed in Table 1.30.

Table 1.30 ColumnFamilyUpdater Class Constructors

Constructor	Description
<code>ColumnFamilyUpdater<K,N>(ColumnFamilyTemplate<K,N> template, ColumnFactory columnFactory)</code>	Creates a <code>ColumnFamilyUpdater</code> instance with the supplied <code>ColumnFamilyTemplate</code> instance and <code>ColumnFactory</code> instance
<code>ColumnFamilyUpdater<K,N>(ColumnFamilyTemplate<K,N> mutator, Mutator<K> mutator)</code>	Includes a <code>Mutator</code> instance in addition to the parameters <code>template</code> , <code>ColumnFactory columnFactory</code> for the preceding constructor

Alternatively, a `ColumnFamilyUpdater` may be created using a `ThriftColumnFamilyTemplate` instance, which provides the methods discussed in Table 1.31 for creating a `ColumnFamilyUpdater`.

Table 1.31 ThriftColumnFamilyTemplate Methods to create a ColumnFamilyUpdater

Method	Description
<code>createUpdater()</code>	Creates a <code>ColumnFamilyUpdater</code> using the query fields in the template
<code>createUpdater(K key)</code>	Creates a <code>ColumnFamilyUpdater</code> using the query fields in the template and the supplied row key
<code>createUpdater(K key, Mutator<K> mutator)</code>	Creates a <code>ColumnFamilyUpdater</code> using the query fields in the template and the supplied row key and <code>Mutator</code> instance

Add an `updateTableData()` method to update a row of data. Create a `ColumnFamilyUpdater<String, String>` instance using the `createUpdater(K key)` method with the supplied key—for example, "catalog2".

```
ColumnFamilyUpdater<String, String> updater = template.createUpdater("catalog2");
```

The `ColumnFamilyUpdater` interface provides several methods for setting an updated value for a column, some of which are listed in Table 1.32.

Table 1.32 ColumnFamilyUpdater Interface Methods

Method	Description
<code>setString(N columnName, String value)</code>	Sets a string value for a column to update
<code>setUUID(N columnName, UUID value)</code>	Sets a UUID value for a column to update
<code>setLong(N columnName, Long value)</code>	Sets a long value for a column to update
<code>setInteger(N columnName, Integer value)</code>	Sets an integer value for a column to update
<code>setFloat(N columnName, Float value)</code>	Sets a float value for a column to update
<code>setDouble(N columnName, Double value)</code>	Sets a double value for a column to update
<code>setBoolean(N columnName, Boolean value)</code>	Sets a boolean value for a column to update
<code>setByteArray(N columnName, byte[] value)</code>	Sets a <code>byte[]</code> value for a column to update
<code>setByteBuffer(N columnName, ByteBuffer value)</code>	Sets a <code>ByteBuffer</code> value for a column to update
<code>setDate(N columnName, Date value)</code>	Sets a date value for a column to update

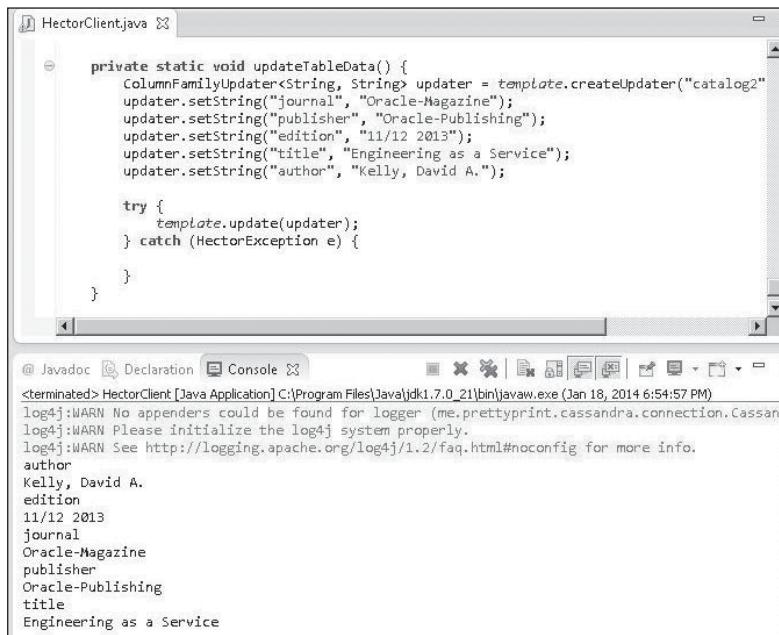
Set the updated values for the columns in the row for key "catalog2".

```
updater.setString("journal", "Oracle-Magazine");
updater.setString("publisher", "Oracle-Publishing");
updater.setString("edition", "11/12 2013");
updater.setString("title", "Engineering as a Service");
updater.setString("author", "Kelly, David A.");
```

When a `ColumnFamilyUpdater` instance has been constructed with the updated values, you can invoke the `update(ColumnFamilyUpdater<K, N> updater)` method to update.

```
try {
    template.update(updater);
} catch (HectorException e) {
}
```

Invoke the `updateTableData()` method from the `main` method and run the `HectorClient` application to update the row with key "catalog2". Then query row "catalog2" using the `retrieveTableData()` method to output the updated values, as shown in Figure 1.30.



The screenshot shows the Eclipse IDE interface. The top part displays the Java code for the `updateTableData()` method. The bottom part shows a terminal window with the application's log output.

```
private static void updateTableData() {
    ColumnFamilyUpdater<String, String> updater = template.createUpdater("catalog2");
    updater.setString("journal", "Oracle-Magazine");
    updater.setString("publisher", "Oracle-Publishing");
    updater.setString("edition", "11/12 2013");
    updater.setString("title", "Engineering as a Service");
    updater.setString("author", "Kelly, David A.");

    try {
        template.update(updater);
    } catch (HectorException e) {

    }
}
```

```
@ Javadoc Declaration Console
@terminated> HectorClient [Java Application] C:\Program Files\Java\jdk1.7.0_21\bin\javaw.exe (Jan 18, 2014 6:54:57 PM)
log4j:WARN No appenders could be found for logger (me.prettyprint.cassandra.connection.CassandraConnection).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
author
Kelly, David A.
edition
11/12 2013
journal
Oracle-Magazine
publisher
Oracle-Publishing
title
Engineering as a Service
```

Figure 1.30
Updated column values.

Source: Eclipse Foundation.

DELETING TABLE DATA

Next, you will delete data from Cassandra database. As when adding row column(s), you need to create a `Mutator` instance using a `Keyspace` instance and a `StringSerializer`.

```
Mutator<String> mutator = HFactory.createMutator(keyspace, StringSerializer.get());
```

As with adding data, you can delete data as a single column or delete multiple columns of data as a batch.

Deleting a Single Column

The `Mutator` interface provides the method discussed in Table 1.33 for deleting a column.

Table 1.33 Mutator Interface Method to Delete a Column

Method	Description
<code>delete(final K key, final String cf, final N columnName, final Serializer<N> nameSerializer)</code>	Deletes a specified column name for a specified row key in a specified column family using the specified serializer

Add a `deleteTableDataColumn()` method to the `HectorClient` class. Then delete the "journal" column in the "catalog" column family in the row with key as "catalog3" and using a `StringSerializer`.

```
mutator.delete("catalog3", "catalog", "journal", StringSerializer.get());
```

Invoke the `deleteTableDataColumn()` method in the `main` method and run the `HectorClient` application. The `delete` method returns a `MutationResult` instance. Invoke the `retrieveTableDataMultigetSliceQuery()` method after invoking the `deleteTableDataColumn()` method to output the modified row set. The row set output using the `retrieveTableDataMultigetSliceQuery()` method before a single column is deleted is shown in Figure 1.31.

```

@ Javadoc Declaration Console
<terminated> HectorClient [Java Application] C:\Program Files\Java\jdk1.7.0_21\bin\javaw.exe (Jan 19, 2014 8:52:27 AM)
log4j:WARN No appenders could be found for logger (me.prettyprint.cassandra.con
ection.CassandraHostRetryService).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more in
fo.
Row(catalog1,ColumnSlice([HColumn(author=Tom Haunert), HColumn(edition=November-
December 2013), HColumn(journal=Oracle Magazine), HColumn(publisher=Oracle Publi
shing), HColumn(title=Quintessential and Collaborative)]))
Row(catalog2,ColumnSlice([HColumn(author=David A. Kelly), HColumn(edition=Novemb
er-December 2013), HColumn(journal=Oracle Magazine), HColumn(publisher=Oracle Pu
blishing), HColumn(title=Engineering as a Service)]))
Row(catalog3,ColumnSlice([HColumn(journal=Oracle Magazine)]))

```

Figure 1.31

Result of query before deleting a row.

Source: Eclipse Foundation.

Figure 1.32 shows the row set output using the `retrieveTableDataMultiGetSliceQuery()` method after a single column is deleted. The journal column is not included in the "catalog3" row because the column has been deleted.

```

@ Javadoc Declaration Console
<terminated> HectorClient [Java Application] C:\Program Files\Java\jdk1.7.0_21\bin\javaw.exe (Jan 19, 2014 8:53:20 AM)
log4j:WARN No appenders could be found for logger (me.prettyprint.cassandra.con
nection.CassandraHostRetryService).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more in
fo.
Row(catalog1,ColumnSlice([HColumn(author=Tom Haunert), HColumn(edition=November-
December 2013), HColumn(journal=Oracle Magazine), HColumn(publisher=Oracle Publi
shing), HColumn(title=Quintessential and Collaborative)]))
Row(catalog2,ColumnSlice([HColumn(author=David A. Kelly), HColumn(edition=Novemb
er-December 2013), HColumn(journal=Oracle Magazine), HColumn(publisher=Oracle Pu
blishing), HColumn(title=Engineering as a Service)]))
Row(catalog3,ColumnSlice([]))

```

Figure 1.32

Result of query after deleting a row.

Source: Eclipse Foundation.

Deleting Multiple Columns

In this section, you will delete multiple columns from a row. The Mutator interface provides the overloaded `addDeletion` methods for deleting multiple columns from a row. Some of the overloaded `addDeletion` methods are listed in Table 1.34.

Table 1.34 Mutator Interface Methods for Deleting Multiple Columns

Method	Description
<code>addDeletion(K key, String cf, N columnName, Serializer<N> nameSerializer)</code>	Adds a deletion mutation to the Mutator instance using a specified key, column family, column name, and column name serializer
<code>addDeletion(K key, String cf)</code>	Adds a deletion mutation to the Mutator instance using a specified key and column family
<code>addDeletion(Iterable<K> keys, String cf)</code>	Adds a deletion mutation to the Mutator instance using a specified iterable of keys and column family

All the `addDeletion` methods return a `Mutator` instance, which can be used to invoke the `addDeletion` method again to link a batch of deletions. Add a `deleteTableData()` method to delete a batch of columns. Then create a `Mutator` instance from the `HFactory` class.

```
Mutator<String> mutator = HFactory.createMutator(keyspace, StringSerializer
.get());
```

Invoke the `addDeletion()` method multiple times in sequence to add delete mutations for the "journal", "publisher", and "edition" columns from the "catalog2" row in the "catalog" column family. Adding delete mutations with the `addDeletion()` method does not delete the columns by itself. Invoke the `execute()` method to delete the mutations added to the `Mutator` instance.

```
mutator.addDeletion("catalog2", "catalog",
"journal", StringSerializer.get()).addDeletion("catalog2", "catalog",
"publisher",
StringSerializer.get())
.addDeletion("catalog2", "catalog", "edition",
StringSerializer.get()).execute();
```

Invoke the `deleteTableData()` method in the `main` method and run the `HectorClient` application to delete the columns added using the `addDeletion()` method. If the `retrieveTableData()` method is invoked before the batch deletions are applied, the query result shown in Figure 1.33 is output.

The screenshot shows the Eclipse IDE's Console view. The output is a list of books from a Cassandra database. The books are listed in two groups: one for 'Oracle Magazine' and one for 'Engineering as a Service'. Each group contains three entries: 'Oracle Publishing', 'November-December 2013', and 'Quintessential and Collaborative'. Below each group is the author's name, Tom Haunert, and David A. Kelly.

```
@ Javadoc Declaration Console
<terminated> HectorClient [Java Application] C:\Program Files\Java\jdk1.7.0_21\bin\javaw.exe (Jan 19, 2014 8:55:57 AM)
log4j:WARN No appenders could be found for logger (me.prettyprint.cassandra.connection.CassandraHostRetryService).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.

Oracle Magazine
Oracle Publishing
November-December 2013
Quintessential and Collaborative
Tom Haunert
Oracle Magazine
Oracle Publishing
November-December 2013
Engineering as a Service
David A. Kelly
```

Figure 1.33

Result of query before batch deletions.

Source: Eclipse Foundation.

If the `retrieveTableData()` method is invoked after the batch deletions are applied, the query result shown in Figure 1.34 is output. The "journal", "publisher", and "edition" columns are shown as deleted.

The screenshot shows the Eclipse IDE's Console view. The output is identical to Figure 1.33, but the 'journal', 'publisher', and 'edition' columns for all books are now displayed as null values.

```
@ Javadoc Declaration Console
<terminated> HectorClient [Java Application] C:\Program Files\Java\jdk1.7.0_21\bin\javaw.exe (Jan 19, 2014 8:57:26 AM)
log4j:WARN No appenders could be found for logger (me.prettyprint.cassandra.connection.CassandraHostRetryService).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.

Oracle Magazine
Oracle Publishing
November-December 2013
Quintessential and Collaborative
Tom Haunert
null
null
null
Engineering as a Service
David A. Kelly
```

Figure 1.34

Result of query after batch deletions.

Source: Eclipse Foundation.

THE HectorClient CLASS

The `HectorClient` class appears in Listing 1.1.

Listing 1.1 The `HectorClient` Class

```
package hector;

import java.util.Arrays;
import me.prettyprint.cassandra.serializers.StringSerializer;
```

```
import me.prettyprint.cassandra.service.ColumnSliceIterator;
import me.prettyprint.cassandra.service.ThriftKsDef;
import me.prettyprint.hector.api.Cluster;
import me.prettyprint.hector.api.Keyspace;
import me.prettyprint.hector.api.beans.HColumn;
import me.prettyprint.hector.api.beans.OrderedRows;
import me.prettyprint.hector.api.beans.Rows;
import me.prettyprint.hector.api.ddl.ColumnFamilyDefinition;
import me.prettyprint.hector.api.ddl.ComparatorType;
import me.prettyprint.hector.api.ddl.KeyspaceDefinition;
import me.prettyprint.hector.api.exceptions.HectorException;
import me.prettyprint.hector.api.factory.HFactory;
import me.prettyprint.hector.api.mutation.MutationResult;
import me.prettyprint.hector.api.mutation.Mutator;
import me.prettyprint.hector.api.query.ColumnQuery;
import me.prettyprint.hector.api.query.MultigetSliceQuery;
import me.prettyprint.hector.api.query.Query;
import me.prettyprint.hector.api.query.QueryResult;
import me.prettyprint.hector.api.query.RangeSlicesQuery;
import me.prettyprint.hector.api.query.SliceQuery;
import me.prettyprint.cassandra.service.template.ColumnFamilyResult;
import me.prettyprint.cassandra.service.template.ColumnFamilyTemplate;
import me.prettyprint.cassandra.service.template.ColumnFamilyUpdater;
import me.prettyprint.cassandra.service.template.ThriftColumnFamilyTemplate;

public class HectorClient {

    private static Cluster cluster;
    private static Keyspace keyspace;
    private static ColumnFamilyTemplate<String, String> template;

    public static void main(String[] args) {
        cluster = HFactory.getOrCreateCluster("hector_cluster",
                                              "localhost:9160");

        KeyspaceDefinition keyspaceDef = cluster
            .describeKeyspace("HectorKeyspace");
        if (keyspaceDef == null) {
            createSchema();
        }
        createKeyspace();
        createTemplate();
        addTableData();
    }
}
```

```
// addTableDataColumn();
// deleteTableDataColumn();
// addTableDataColumn();
// retrieveTableDataColumnQuery();
// updateTableData();
// deleteTableDataColumn();
// retrieveTableDataColumnQuery();
// deleteTableData();
// retrieveTableData();
// retrieveTableDataSliceQuery();
    retrieveTableDataMultigetSliceQuery();
}

private static void createSchema() {
    int replicationFactor = 1;
    ColumnFamilyDefinition cfDef = HFactory.createColumnFamily
Definition(
                    "HectorKeyspace", "catalog", ComparatorType.
BYTESTYPE);
    KeyspaceDefinition keyspace = HFactory.createKeyspaceDefinition(
                    "HectorKeyspace", ThriftKsDef.DEF_STRATEGY_CLASS,
                    replicationFactor, Arrays.asList(cfDef));
    cluster.addKeyspace(keyspace, true);
}

private static void createKeyspace() {
    keyspace = HFactory.createKeyspace("HectorKeyspace", cluster);
}

private static void createTemplate() {
    template = new ThriftColumnFamilyTemplate<String, String>
(keyspace,
                    "catalog", StringSerializer.get(),
StringSerializer.get());
}

private static void addTableData() {
    Mutator<String> mutator = HFactory.createMutator(keyspace,
                    StringSerializer.get());
    mutator.addInsertion("catalog1", "catalog",
                    HFactory.createStringColumn("journal", "Oracle
Magazine"))
        .addInsertion(
                    "catalog1",
```

```
        "catalog",
        HFactory.createStringColumn
("publisher",
                                         "Oracle Publishing"))
    .addInsertion(
        "catalog1",
        "catalog",
        HFactory.createStringColumn
("edition",
                                         "November-December
2013"))
    .addInsertion(
        "catalog1",
        "catalog",
        HFactory.createStringColumn
("title",
                                         "Quintessential and
Collaborative"))
    .addInsertion("catalog1", "catalog",
        HFactory.createStringColumn
("author", "Tom Haunert"));
    mutator.addInsertion("catalog2", "catalog",
        HFactory.createStringColumn("journal", "Oracle
Magazine"))
    .addInsertion(
        "catalog2",
        "catalog",
        HFactory.createStringColumn
("publisher",
                                         "Oracle Publishing"))
    .addInsertion(
        "catalog2",
        "catalog",
        HFactory.createStringColumn
("edition",
                                         "November-December
2013"))
    .addInsertion(
        "catalog2",
        "catalog",
        HFactory.createStringColumn
("title",
```

```
        "Engineering as a
Service"))
    .addInsertion("catalog2", "catalog",
                  HFactory.createStringColumn
("author", "David A. Kelly"));
    mutator.execute();
}

private static void retrieveTableData() {
    try {
        ColumnFamilyResult<String, String> res = template
            .queryColumns("catalog1");
        if(res.hasResults()){
            String journal = res.getString("journal");
            String publisher = res.getString("publisher");
            String edition = res.getString("edition");
            String title = res.getString("title");
            String author = res.getString("author");

            System.out.println(journal);
            System.out.println(publisher);
            System.out.println(edition);
            System.out.println(title);
            System.out.println(author);
        }
        res = template.queryColumns("catalog2");
        if(res.hasResults()){
            journal = res.getString("journal");
            publisher = res.getString("publisher");
            edition = res.getString("edition");
            title = res.getString("title");
            author = res.getString("author");

            System.out.println(journal);
            System.out.println(publisher);
            System.out.println(edition);
            System.out.println(title);
            System.out.println(author);
        }
    } catch (HectorException e) {
    }
}
```

```
private static void retrieveTableDataColumnQuery() {
    ColumnQuery<String, String, String> columnQuery = HFactory
        .createStringColumnQuery(keyspace);
    columnQuery.setColumnFamily("catalog").setKey("catalog3")
        .setName("journal");
    //
columnQuery.setColumnFamily("catalog").setKey("catalog1").setName("journal");
    QueryResult<HColumn<String, String>> result = columnQuery.execute
();
    System.out.println(result.get());
}

private static void retrieveTableDataSliceQuery() {
    SliceQuery<String, String, String> query = HFactory
        .createSliceQuery(keyspace, StringSerializer.get(),
            StringSerializer.get(),
StringSerializer.get())
        .setKey("catalog2").setColumnFamily("catalog");

    ColumnSliceIterator<String, String, String> iterator = new
ColumnSliceIterator<String, String,
String>(query, "\u0000", "\uFFFF", false);

    while (iterator.hasNext()) {
        HColumn<String, String> column = iterator.next();
        System.out.println(column.getName());
        System.out.println(column.getValue());
    }
}

private static void addTableDataColumn() {
    Mutator<String> mutator = HFactory.createMutator(keyspace,
        StringSerializer.get());
    MutationResult result=mutator.insert("catalog3", "catalog",
        HFactory.createStringColumn("journal", "Oracle
Magazine"));
    System.out.println(result);
}

private static void updateTableData() {
    ColumnFamilyUpdater<String, String> updater = template
        .createUpdater("catalog2");
    updater.setString("journal", "Oracle-Magazine");
    updater.setString("publisher", "Oracle-Publishing");
```

```
    updater.setString("edition", "11/12 2013");
    updater.setString("title", "Engineering as a Service");
    updater.setString("author", "Kelly, David A.");
}

try {
    template.update(updater);
} catch (HectorException e) {
}
}

private static void deleteTableDataColumn() {
    Mutator<String> mutator = HFactory.createMutator(keyspace,
        StringSerializer.get());
    mutator.delete("catalog3", "catalog", "journal",
StringSerializer.get());
}

private static void deleteTableData() {
    Mutator<String> mutator = HFactory.createMutator(keyspace,
        StringSerializer.get());
    mutator.addDeletion("catalog2", "catalog", "journal",
        StringSerializer.get())
        .addDeletion("catalog2", "catalog", "publisher",
            StringSerializer.get())
        .addDeletion("catalog2", "catalog", "edition",
            StringSerializer.get()).execute();
}

private static void retrieveTableDataMultigetSliceQuery() {
    MultigetSliceQuery<String, String, String> multigetSliceQuery =
        HFactory.createMultigetSliceQuery(keyspace,
StringSerializer.get(),
StringSerializer.get(), StringSerializer.get());
    multigetSliceQuery.setColumnFamily("catalog");
    multigetSliceQuery.setKeys("catalog1", "catalog2",
        "catalog3");
    //multigetSliceQuery.setRange("", "", false, 3);
    //multigetSliceQuery.setRange("", "", false, 2);
    multigetSliceQuery.setRange("", "", false, 5);
    QueryResult<Rows<String, String, String>> result =
multigetSliceQuery.execute();
    System.out.println(result.get().getByKey("catalog1"));
    System.out.println(result.get().getByKey("catalog2"));
}
```

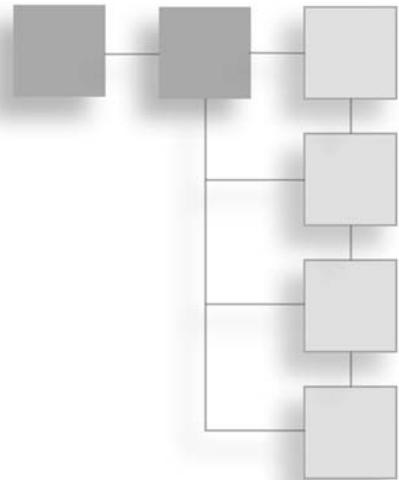
```
        System.out.println(result.get().getByKey("catalog3"));
    }
private static void retrieveTableDataRangeSlicesQuery() {
    RangeSlicesQuery<String, String, String> rangeSlicesQuery =
        HFactory.createRangeSlicesQuery(keyspace,
StringSerializer.get(),
StringSerializer.get(), StringSerializer.get());
    rangeSlicesQuery.setColumnFamily("catalog");
    rangeSlicesQuery.setKeys("catalog1", "catalog3");
//rangeSlicesQuery.setRange("", "", false, 5);
//rangeSlicesQuery.setRange("", "", false, 3);
    QueryResult<0rderedRows<String, String, String>>
result =
rangeSlicesQuery.execute();
    System.out.println(result.get().getByKey
("catalog1"));
    System.out.println(result.get().getByKey
("catalog2"));
    System.out.println(result.get().getByKey
("catalog3"));
}
}
```

SUMMARY

This chapter discussed using the Hector Java client to access the Apache Cassandra database and make create, read, update, and delete (CRUD) operations on the database data. The Hector client supports adding and deleting column data as single columns or a batch of columns. Hector supports retrieving column data as single columns or a column slice. Row data may be queried one row at a time or multiple rows in the same query. This chapter discussed the various interfaces and classes involved in making the CRUD operations. The next chapter will discuss the Cassandra Query Language (CQL) for querying Cassandra. You will use the Hector Java client to run the CQL queries.

CHAPTER 2

QUERYING CASSANDRA WITH CQL



If you are transitioning from a relational database and SQL, you will find Cassandra Query Language (CQL) easy to use for accessing the Cassandra server. CQL has a syntax similar to SQL and can be used from a command line shell (`cqlsh`) or from client APIs such as the Hector API introduced in Chapter 1, “Using Cassandra with Hector.” Although Cassandra is a NoSQL database, Cassandra’s data model is similar to a relational database with a storage model based on column families, columns, and rows. Instead of querying a relational database table, you query a column family. Instead of querying relational database columns and rows, you query Cassandra’s columns and rows. This chapter introduces CQL using the Hector client API for running CQL statements. Another API that supports CQL may be used just as well for running the CQL statements.

OVERVIEW OF CQL

CQL 3 is the latest version of CQL. Being a query language for a non-relational database, some constructs used in SQL are not supported in CQL—for example, `JOINS`. CQL 3 identifiers are case-insensitive unless enclosed in double quotes. CQL 3 keywords are also case-insensitive. An identifier in CQL is a letter followed by any sequence of letters, digits, and the underscore. A string literal in CQL is specified with single quotes, and to use a single quotation mark in a query, it must be delimited, or escaped, with another single quote. CQL 3 data types were discussed in Chapter 1. The CQL 3 commands are discussed in Table 2.1.

Table 2.1 CQL 3 Commands

Command	Description
ALTER TABLE or ALTER COLUMNFAMILY	Alters column family metadata such as the data storage type of columns and column family properties. Also used to add/drop columns.
ALTER KEYSPACE	Alters the keyspace attributes. The attributes supported by a keyspace are replica placement strategy, strategy options, and durable_writes. Replica placement strategy was discussed in Chapter 1, with the two supported types being SimpleStrategy and NetworkTopologyStrategy. Strategy options are the configuration options for the chosen replica placement strategy. The durable_writes attribute makes data more durable and prevents data loss by creating a commit log. It is set to true by default.
BATCH	Runs multiple data modification language (DML) statements as a batch.
CREATE TABLE	Creates a new column family, also called a table.
CREATE INDEX	Creates a secondary index on the specified column in the specified column family.
CREATE KEYSPACE	Creates a keyspace, including the replica placement strategy.
DELETE	Deletes one or more columns from a row.
DROP TABLE	Removes a column family.
DROP INDEX	Removes a secondary index.
DROP KEYSPACE	Removes a keyspace.
INSERT	Adds column data to a column family row.
SELECT	Retrieves data from a column family.
TRUNCATE	Truncates a column family; removes all data from a column family.
UPDATE	Updates column data in a column family.
USE	Sets the keyspace to use.

For a complete syntax of CQL 3 commands, see <http://cassandra.apache.org/doc/cql3/CQL.html>.

Note that not all Java clients support CQL 3. For example, Hector does not support CQL 3, but supports CQL 2.0 (<http://cassandra.apache.org/doc/cql/CQL.html>). Subsequent sections discuss most CQL 2 statements with an example. Later in the chapter, we will discuss some of the new features in CQL 3. You will use the CQL 3 commands in Chapter 3, “Using Cassandra with DataStax Java Driver,” on the DataStax Java driver.

SETTING THE ENVIRONMENT

You will use Hector Java client to run CQL statements. Download the following software:

- Apache Cassandra apache-cassandra-2.0.4-bin.tar.gz or a later version from <http://cassandra.apache.org/download/>
- Hector Java client hector-core-1.1-4.jar or a later version from <https://github.com/hector-client/hector/downloads>
- Eclipse IDE for Java EE developers from <http://www.eclipse.org/downloads/>
- Java SE 7 from <http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html>

Then follow these steps:

1. Install the Eclipse IDE.
2. Extract the Apache Cassandra TAR file to a directory (for example, C:\Cassandra\apache-cassandra-2.0.4).
3. Add the bin folder, C:\Cassandra\apache-cassandra-2.0.4\bin, to the PATH environment variable.
4. Start Apache Cassandra server with the following command:
`cassandra -f`

The Cassandra server starts and begins listening for CQL clients on localhost:9042. Cassandra listens for Thrift clients on localhost:9160.

CREATING A JAVA PROJECT

In this section, you will create a Java project in Eclipse for running CQL statements using a Hector client. Follow these steps:

1. Select File > New > Other in the Eclipse IDE.
2. In the New window select the Java Project wizard and click Next, as shown in Figure 2.1.

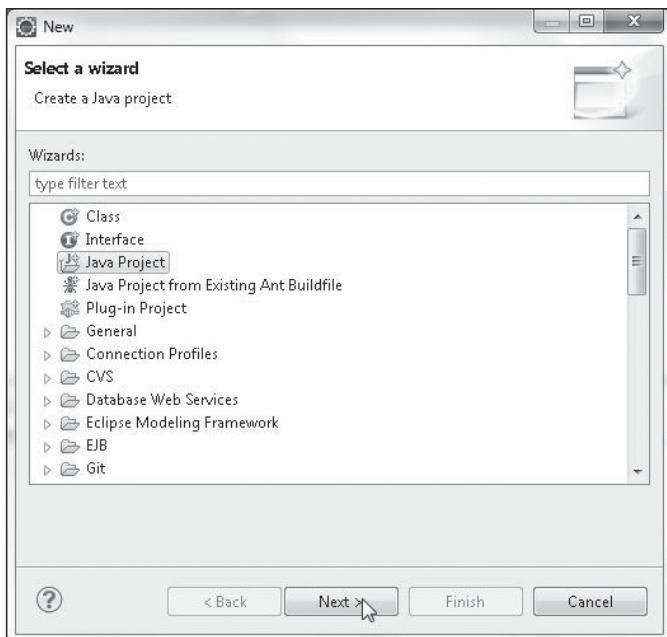


Figure 2.1
Selecting the Java Project wizard.

Source: Eclipse Foundation.

3. In the Create a Java Project screen, specify a project name (for example, CQL), select the directory location for the project (or choose Use Default Location), select the JRE, and click Next, as shown in Figure 2.2.

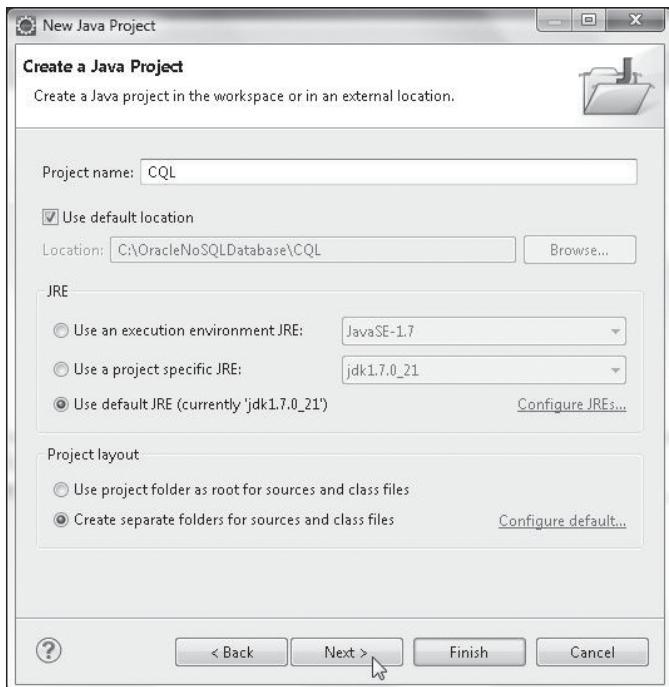


Figure 2.2
Creating a new Java project.

Source: Eclipse Foundation.

4. In the Java Settings screen, select the default settings and click Finish, as shown in Figure 2.3. A new Java project is created in Eclipse, as shown in the Package Explorer (see Figure 2.4).



Figure 2.3
The Java Settings screen.

Source: Eclipse Foundation.

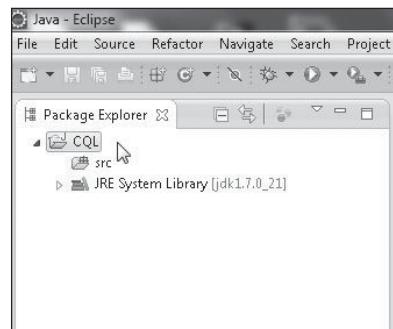


Figure 2.4
The new Java project in the Package Explorer.

Source: Eclipse Foundation.

5. You need to add the same JAR files to the project Java build path as for the Chapter 1 project. To begin, right-click the project node in the Package Explorer and select Properties.
6. In the Properties for CQL dialog box, select the Java Build Path node and click the Add External JARs button to add JAR files, as shown in Figure 2.5. Then click OK.

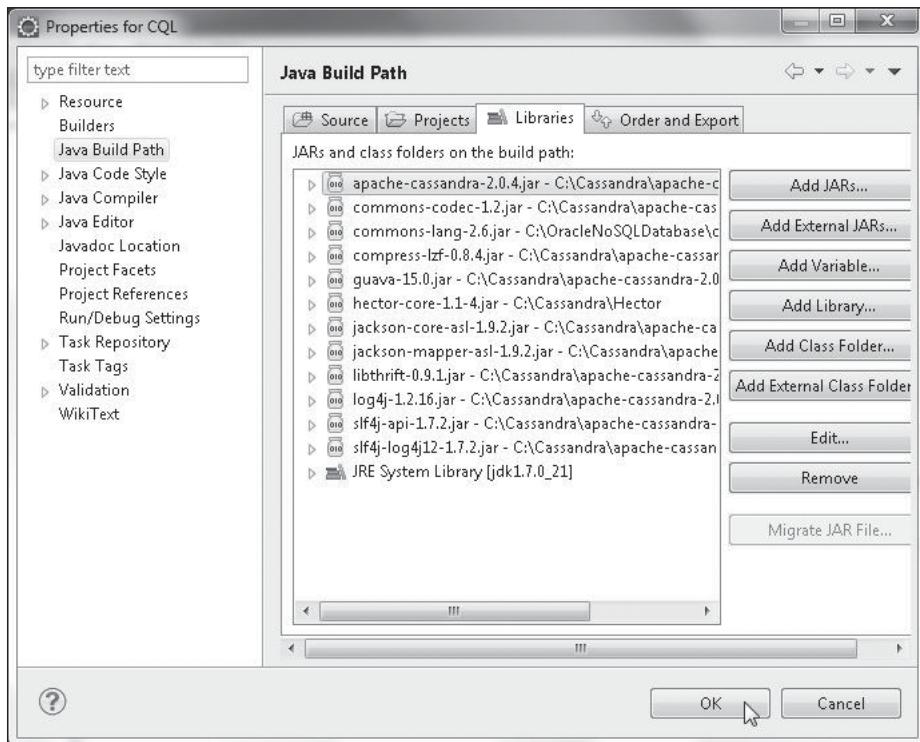


Figure 2.5
Adding JAR files.

Source: Eclipse Foundation.

7. Add a Java class for the Hector client application. Select File > New > Other and, in the New window, select Java > Class, as shown in Figure 2.6. Then click Next.

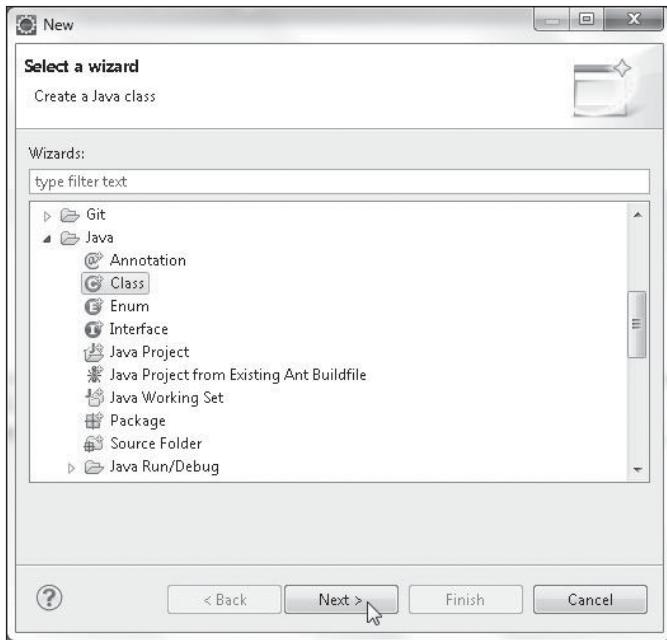


Figure 2.6
Selecting Java > Class.
Source: Eclipse Foundation.

8. In the New Java Class wizard, select the source folder (CQL/src), specify a package (cql), and specify a class name (CQLClient). Then select the main method stub to create and click Finish, as shown in Figure 2.7. The CQLClient Java class is created and added to the Eclipse Java project, as shown in the Package Explorer in Figure 2.8.



Figure 2.7
Creating a new Java class.
Source: Eclipse Foundation.

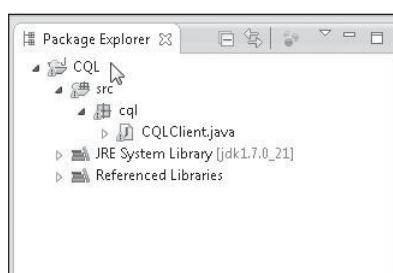


Figure 2.8
The Java class CQLClient.
Source: Eclipse Foundation.

CREATING A KEYSPACE

In CQL 3, the syntax for creating a keyspace is as follows:

```
CREATE KEYSPACE <keyspace_name> WITH <property1> = {} AND <property2> = {};
```

The properties supported are discussed in Table 2.2.

Table 2.2 CREATE KEYSPACE Command Properties				
Property	Type	Required	Default Value	Description
replication	Map	Yes	1	The replication strategy and options
durable_writes	Simple	No	true	If the data written to the keyspace is to be stored in the commit log

If SimpleStrategy is used as the replication strategy, an example of a command to create a keyspace is as follows:

```
CREATE KEYSPACE CQLKeyspace
WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 1}
AND durable_writes = false;
```

The 'replication_factor' sub-option can be used only with SimpleStrategy. If NetworkTopologyStrategy is used, an example of a command to create a keyspace is as follows:

```
CREATE KEYSPACE CQLKeyspace
WITH replication = {'class': 'NetworkTopologyStrategy', 'DC1' : 1, 'DC2' : 1}
AND durable_writes = true;
```

The DC1 and DC2 refer to the data centers DC1 and DC2. The sub-option values are the individual replication factors for each data center.

The CQL 2 syntax for creating a keyspace is as follows:

```
CREATE KEYSPACE <ks_name>
WITH strategy_class = <value>
[ AND strategy_options:<option>=<value> [strategy_options:<option>=<value>]];
```

For example:

```
CREATE KEYSPACE CQLKeyspace WITH strategy_class = 'SimpleStrategy'
AND strategy_options:replication_factor = 1;
```

If Cassandra CLI (client interface utility) is used to create a keyspace, the syntax of the CREATE KEYSPACE command is different than that discussed. Cassandra CLI does not completely support CQL, and the Thrift API is supported. To create a keyspace in Cassandra CLI, start Cassandra CLI with the following command:

```
cassandra-cli
```

Run the following command to create a keyspace with the name CQLKeyspace, the replication strategy SimpleStrategy, and a replication factor of 1:

```
CREATE KEYSPACE CQLKeyspace WITH placement_strategy= 'org.apache.cassandra.locator.SimpleStrategy' AND strategy_options={replication_factor:1};
```

A keyspace CQLKeyspace is created and the output from the command is shown in Figure 2.9. To use the keyspace run the following command:

```
use CQLKeyspace;
```

As indicated by the message output in Figure 2.9, the CQLKeyspace is authenticated.

```
cmd Command Prompt - cassandra-cli
Microsoft Windows [Version 6.1.7601]
Copyright <c> 2009 Microsoft Corporation. All rights reserved.

C:\Users\Deepak Vohra>cd ..
C:\Users>cd ..
C:\>cd cassandra

C:\Cassandra>cassandra-cli
Starting Cassandra Client
Connected to: "Test Cluster" on 127.0.0.1/9160
Welcome to Cassandra CLI version 2.0.4

The CLI is deprecated and will be removed in Cassandra 3.0. Consider migrating
to cqlsh.
CQL is fully backwards compatible with Thrift data; see http://www.datastax.com/
dev/blog/thrift-to-cql3

Type 'help;' or '?' for help.
Type 'quit;' or 'exit;' to quit.

[default@unknown] CREATE KEYSPACE CQLKeyspace
...   with placement_strategy = 'org.apache.cassandra.locator.SimpleStrategy'
...     and strategy_options = {replication_factor:1};
895eda5c-4e5c-3dd3-b171-e6004b1f1fe8
[default@unknown] use CQLKeyspace
Authenticated to keyspace: CQLKeyspace
```

Figure 2.9

The CQLKeyspace is authenticated.

Source: Microsoft Corporation.

CREATING A COLUMN FAMILY

You will use the Hector client to run the CQL statement to create a column family. To begin, add a `createCF()` method to the `CQLClient` class and invoke the method from the `main` method. Hector provides the `me.prettyprint.cassandra.model.CqlQuery` class to run CQL statements. The constructor for the class is `CqlQuery(Keyspace k, Serializer<K>`

`keySerializer,Serializer<N> nameSerializer, Serializer<V> valueSerializer).` But, before you may run CQL statements, you need to create a Cluster instance and a Keyspace instance as discussed in Chapter 1.

```
Cluster cluster = HFactory.getOrCreateCluster("cql-cluster", "localhost:9160");
Keyspace keyspace = HFactory.createKeyspace("CQLKeyspace", cluster);
```

Create a CQLQuery instance using the class constructor with StringSerializer instances for key, column name, and column value.

```
CqlQuery cqlQuery = new CqlQuery(keyspace, StringSerializer.get(),
StringSerializer.get(), StringSerializer.get());
```

Next, set the CQL query to create a column family. Set the comparator as UTF8Type, which implies that columns are sorted based on UTF8Type sorting and columns are displayed as UTF8Type text. The other supported types are AsciiType, BytesType (the default), CounterColumnType, IntegerType, LexicalUUIDType and LongType. The default validation class is set using the `default_validation` parameter set to UTF8Type and is the validator to use for column values. The supported types and default setting are the same as for the comparator.

```
cqlQuery.setQuery("CREATE COLUMNFAMILY catalog (catalog_id text PRIMARY KEY,
journal text,publisher text,edition text,title text,author text)WITH
comparator=UTF8Type AND default_validation_class=UTF8Type");
```

Some of the other supported options are discussed in Table 2.3, all of the column family options being optional. Only the column family name is a required parameter.

Table 2.3 CREATE COLUMNFAMILY Command Options

Parameter	Description
<code>caching</code>	If keys and/or rows are to be cached. Supported values are <code>all</code> , <code>keys_only</code> , <code>rows_only</code> , and <code>none</code> .
<code>replicate_on_write</code>	If data is to be replicated on write. Set to <code>true</code> by default. The <code>false</code> value is supported only for counter values, but is not recommended.

To run the CQL query, invoke the `execute()` method:

```
cqlQuery.execute();
```

The CQLClient application to create a column family catalog using the Hector client to run the CQL statement appears in Listing 2.1.

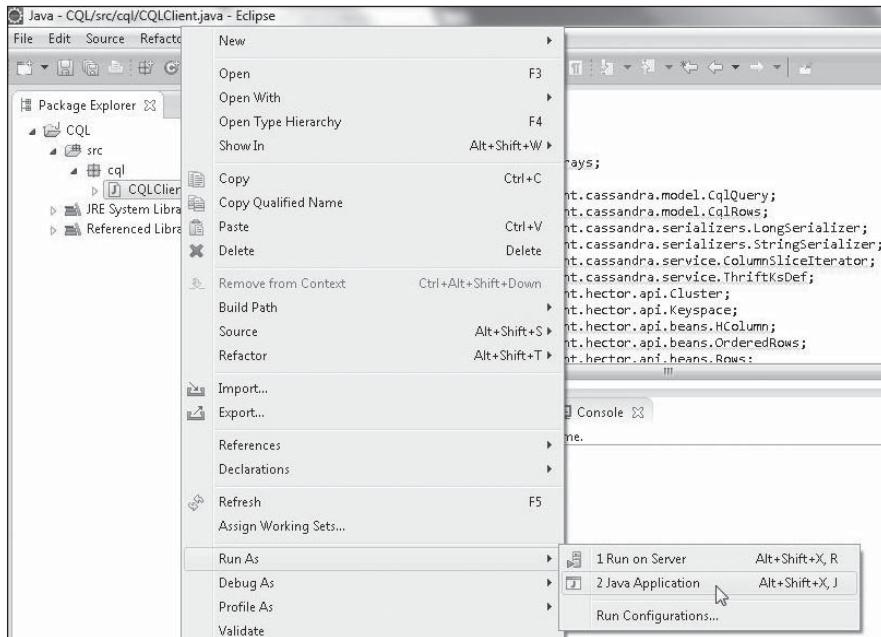
Listing 2.1 CQLClient Application

```
package cql;

import me.prettyprint.cassandra.model.CqlQuery;
import me.prettyprint.cassandra.serializers.StringSerializer;
import me.prettyprint.hector.api.Cluster;
import me.prettyprint.hector.api.Keyspace;
import me.prettyprint.hector.api.factory.HFactory;

public class CQLClient {
    private static Cluster cluster;
    private static Keyspace keyspace;
    public static void main(String[] args) {
        cluster = HFactory.getOrCreateCluster("cql-cluster",
"localhost:9160");
        createKeyspace();
        createCF();
    }
    private static void createKeyspace() {
        keyspace = HFactory.createKeyspace("CQLKeyspace", cluster);
    }
    private static void createCF() {
        CqlQuery cqlQuery = new CqlQuery(keyspace, StringSerializer.get(),
StringSerializer.get(), StringSerializer.get());
        cqlQuery.setQuery("CREATE COLUMNFAMILY catalog (catalog_id text
PRIMARY KEY,journal text,publisher text,edition text,title text,author text) WITH
comparator=UTF8Type AND default_validation=UTF8Type AND caching=keys_only AND
replicate_on_write=true");
        cqlQuery.execute();
    }
}
```

If it's not already started, start Cassandra, right-click CQLClient, and select Run As > Java Application as shown in Figure 2.10.

**Figure 2.10**

Running the CQLClient application.

Source: Eclipse Foundation.

The "catalog" column family is created in the `CQLKeyspace` keyspace. (This keyspace must be created prior to running the CQL statement to create the column family.) In subsequent sections, you will add other methods to the `CQLClient` class to run CQL statements and invoke the methods from the `main` method. The primary key column of the "catalog" column family is named something other than `KEY`, which makes it unsuitable for being specified in the `WHERE` clause of CQL 2 queries, as you will see in a later section. To create a primary key column called `KEY`, run the following CQL query:

```
cqlQuery.setQuery("CREATE COLUMNFAMILY catalog2 (KEY text PRIMARY KEY, journal
text,publisher text,edition text,title text,author text)");
```

One of the columns must be a primary key column. If a primary key is not specified, the following exception is generated:

```
InvalidRequestException(why:You must specify a PRIMARY KEY)
```

USING THE INSERT STATEMENT

In this section, you will run the `INSERT` CQL statement. The syntax for the `INSERT` statement with the required clauses is as follows:

```
INSERT INTO <tablename> (<column1>, <column2>, <column>) VALUES (<value1>,
<value2>, <valueN>)
```

The number of values must match the number of columns or the following exception is generated:

```
InvalidRequestException(why:unmatched column names/values)
```

However, the number of columns/values may be less than in the schema for the column family. The primary key column must be specified, as the primary key identifies a row. Add an `insert()` method to the `CQLClient` class and invoke the method from the `main` method. Create a `CQLQuery` object as before.

```
CqlQuery cqlQuery = new CqlQuery(keyspace, StringSerializer.get(),
StringSerializer.get(), StringSerializer.get());
```

Set the query to add a row to the catalog table using the `setQuery(String)` method:

```
cqlQuery.setQuery("INSERT INTO catalog (catalog_id, journal, publisher, edition,
title,author) VALUES ('catalog1','Oracle Magazine', 'Oracle Publishing',
'November-December 2013', 'Engineering as a Service', 'David A. Kelly')");
```

Then run the query with the `execute()` method:

```
cqlQuery.execute();
```

Similarly, add another row:

```
cqlQuery.setQuery("INSERT INTO catalog (catalog_id, journal, publisher, edition,
title,author) VALUES ('catalog2','Oracle Magazine', 'Oracle Publishing',
'November-December 2013', 'Quintessential and Collaborative', 'Tom Haunert')");
cqlQuery.execute();
```

The `INSERT` statement adds a new row if one does not exist and replaces the row if a row with the same primary key already exists. Run the `CQLClient` application to invoke the `insert()` method and add data to the "catalog" column family. Then add the rows in Cassandra CLI. To fetch the row with the "catalog1" and "catalog2" keys, run the following commands:

```
get catalog['catalog1'];
get catalog['catalog2'];
```

The output from the command fetches the rows added with the INSERT statement, as shown in Figure 2.11.

```
[default@CQLKeyspace] get catalog1['catalog1'];
=> <name=author, value=David A. Kelly, timestamp=1390411951725000>
=> <name=edition, value=November-December 2013, timestamp=1390411951725002>
=> <name=journal, value=Oracle Magazine, timestamp=1390411951725003>
=> <name=publisher, value=Oracle Publishing, timestamp=1390411951725001>
=> <name=title, value=Engineering as a Service, timestamp=1390411951725004>
Returned 5 results.
Elapsed time: 101 msec(s).
[default@CQLKeyspace]
[default@CQLKeyspace] get catalog1['catalog2'];
=> <name=author, value=Tom Haunert, timestamp=1390411951790000>
=> <name=edition, value=November-December 2013, timestamp=1390411951790002>
=> <name=journal, value=Oracle Magazine, timestamp=1390411951790003>
=> <name=publisher, value=Oracle Publishing, timestamp=1390411951790001>
=> <name=title, value=Quintessential and Collaborative, timestamp=1390411951790004>
Returned 5 results.
Elapsed time: 11 msec(s).
[default@CQLKeyspace]
```

Figure 2.11

Adding rows with the INSERT statement.

Source: Microsoft Corporation.

You add a row to a column family with the name KEY in a similar manner:

```
cqlQuery.setQuery("INSERT INTO catalog2 (KEY, journal, publisher, edition, title,
author) VALUES ('catalog1', 'Oracle Magazine', 'Oracle Publishing', 'November-
December 2013', 'Engineering as a Service', 'David A. Kelly')");
cqlQuery.execute();
```

When a row is added, all the columns/values do not have to be specified. For example, the following CQL query adds a row without the journal column. Flexible schema is one of the features of the Cassandra database and of NoSQL databases in general.

```
cqlQuery.setQuery("INSERT INTO catalog (catalog_id, publisher, edition, title,
author) VALUES ('catalog4', 'Oracle Publishing', 'November-December 2013',
'Engineering as a Service', 'David A. Kelly')");
cqlQuery.execute();
```

USING THE SELECT STATEMENT

In this section, you will query using the SELECT statement. The SELECT statement must have the following required clauses and keywords:

`SELECT <select-clause> FROM <tablename>`

The SELECT statement queries one or more columns from one or more rows and returns the result as a rowset, with each row having the columns specified in the query. Even if a

column name not defined in the column family schema is specified in the SELECT statement's <select-clause>, the column value is returned—a null value for a non-existent column. The columns whose values are to be selected are specified in the <select-clause> as comma-separated column names. Alternatively, to select all columns, specify *. The <tablename> is the column family or table from which to select.

Add a method called `select()` to the CQLClient application and invoke the method from the `main` method. Then create a `CQLQuery` object as before.

```
CqlQuery<String, String, String> cqlQuery = new CqlQuery<String, String, String>(
    keyspace, StringSerializer.get(), StringSerializer.get(), StringSerializer.get());
```

As an example, select all columns using *:

```
cqlQuery.setQuery("select * from catalog");
```

Invoke the `execute()` method to run the CQL statement. The result of the query is returned as a `QueryResult<CqlRows<K, N, V>>` object.

```
QueryResult<CqlRows<String, String, String>> result = cqlQuery.execute();
```

Fetch the result using the `get()` method and create an `Iterator` over the result using the `iterator()` method.

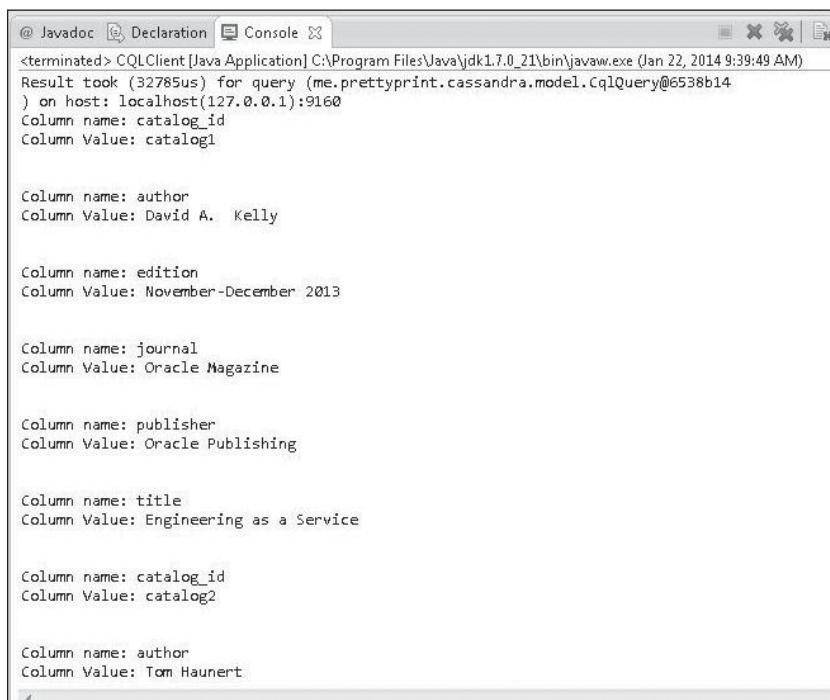
```
Iterator iterator = result.get().iterator();
```

Iterate over the result to fetch individual rows. A row is represented with the `Row` interface, and a `Row` instance consists of a key/column slice tuple. Get the key value using the `getKey()` method and get the column slice represented with the `ColumnSlice` interface using the `getColumnSlice()` method. Fetch the collection of columns from the `ColumnSlice` instance using the `getColumns` method. Create another `Iterator` over the list of columns and iterate over the columns to fetch individual `HColumn` instances, which represent the columns in the column slice. Output the column name using the `getName()` method from `HColumn` and output the column value using the `getValue()` method.

```
while (iterator.hasNext()) {
    Row row = (Row) iterator.next();
    String key = (String) row.getKey();
    ColumnSlice columnSlice = row.getColumnSlice();
    List columnList = columnSlice.getColumns();
    Iterator iter = columnList.iterator();
    while (iter.hasNext()) {
        HColumn column = (HColumn) iter.next();
    }
}
```

```
        System.out.println("Column name: " +  
column.getName() + " ");  
        System.out.println("Column Value: " +  
column.getValue());  
        System.out.println("\n");  
    }  
}
```

Run the CQLClient application to fetch all the column values from the catalog table. The catalog1 row columns are output as shown in Figure 2.12.



The screenshot shows the Eclipse IDE's Console tab with the following output:

```
<terminated> CQLClient [Java Application] C:\Program Files\Java\jdk1.7.0_21\bin\javaw.exe (Jan 22, 2014 9:39:49 AM)  
Result took (32785us) for query (me.prettyprint.cassandra.model.CqlQuery@6538b14  
) on host: localhost(127.0.0.1):9160  
Column name: catalog_id  
Column Value: catalog1  
  
Column name: author  
Column Value: David A. Kelly  
  
Column name: edition  
Column Value: November-December 2013  
  
Column name: journal  
Column Value: Oracle Magazine  
  
Column name: publisher  
Column Value: Oracle Publishing  
  
Column name: title  
Column Value: Engineering as a Service  
  
Column name: catalog_id  
Column Value: catalog2  
  
Column name: author  
Column Value: Tom Haunert
```

Figure 2.12

The result of a SELECT statement.

Source: Eclipse Foundation.

The catalog2 row columns are output as shown in Figure 2.13.

The screenshot shows the Eclipse CQLClient Java Application Console window. The title bar reads '@ Javadoc Declaration Console <terminated> CQLClient [Java Application] C:\Program Files\'. The console output displays the results of a SELECT statement, listing various columns and their values for a specific row.

```

Column name: publisher
Column Value: Oracle Publishing

Column name: title
Column Value: Engineering as a Service

Column name: catalog_id
Column Value: catalog2

Column name: author
Column Value: Tom Haunert

Column name: edition
Column Value: November-December 2013

Column name: journal
Column Value: Oracle Magazine

Column name: publisher
Column Value: Oracle Publishing

Column name: title
Column Value: Quintessential and Collaborative

```

Figure 2.13
The result of a SELECT statement (continued).

Source: Eclipse Foundation.

The SELECT statement also supports a WHERE clause to filter a query based on the value of another column.

`SELECT <select-clause> FROM <tablename> WHERE <where-clause>`

CQL requires that the WHERE clause with the = comparison be used with the table key alone or an indexed column alone. The column in the = comparison after WHERE must either be the primary key column called KEY or some other column that has a secondary index. Before we discuss how to filter a SELECT query using the WHERE clause, let's add a secondary index on a column.

CREATING A SECONDARY INDEX

CQL provides the CREATE INDEX command to create a secondary index on a column already defined in a column family. For example, the following command will add a secondary index called titleIndex on column called title in table called catalog.

All existing data for the column is indexed asynchronously. When new data is added, it is indexed automatically at the time of insertion.

```
CREATE INDEX titleIndex ON catalog (title)
```

Add a `createIndex()` method to the `CQLClient` class to create a secondary index on a column. Then specify and run the preceding CQL query using a `CQLQuery` instance.

```
cqlQuery.setQuery("CREATE INDEX titleIndex ON catalog (title)");
cqlQuery.execute();
```

Invoke the `createIndex()` method in the `main` method and run the `CQLClient` application to create a secondary index on the `title` column in the `catalog` table.

USING THE SELECT STATEMENT WITH THE WHERE CLAUSE

As mentioned, CQL requires the column in an `=` comparison specified in the `WHERE` clause to be an indexed column or a primary key column called `KEY`. If you run a CQL query using the `WHERE` clause on a primary key column that is not called `KEY` or on some other column that has not been indexed, the following exception is generated:

```
Caused by: InvalidRequestException(why:No indexed columns present in by-columns
clause with "equals" operator)
```

The following `CQLQuery` query would generate the preceding exception because `catalog_id` used with the `=` operator is not an indexed column, and even though it is a primary key column, it is not called `KEY`.

```
cqlQuery.setQuery("SELECT catalog_id, journal, publisher, edition, title, author
FROM catalog WHERE catalog_id='catalog1'");
```

The same goes for the following query because the `journal` column used in the `=` comparison is not an indexed column.

```
cqlQuery.setQuery("SELECT KEY, journal, publisher, edition, title, author FROM
catalog WHERE journal='Oracle Magazine'");
```

Because you created a secondary index on the `title` column in the `catalog` table, you can use the `title` column in the `=` comparison after the `WHERE` clause:

```
cqlQuery.setQuery("SELECT catalog_id, journal, publisher, edition, title, author
FROM catalog WHERE title='Engineering as a Service'");
```

For example, if catalog1 is the only column with the title “Engineering as a Service,” then the preceding query would generate the following result using the same iteration over the `QueryResult<CqlRows<String, String, String>>` result returned by the query:

```
Column name: catalog_id
Column Value: catalog1
Column name: journal
Column Value: Oracle Magazine
Column name: publisher
Column Value: Oracle Publishing
Column name: edition
Column Value: November-December 2013
Column name: title
Column Value: Engineering as a Service
Column name: author
Column Value: David A. Kelly
```

The `SELECT` statement with the `WHERE` clause may also be used with the `KEY` column in the `=` comparison—for example, to select the columns where `KEY` is `catalog1`.

```
cqlQuery.setQuery("SELECT KEY, journal, publisher, edition, title, author FROM
catalog2 WHERE KEY='catalog1'");
```

The result of the query is shown in Figure 2.14.

```
<terminated> CQLClient [Java Application] C:\Program Files\Java\jdk1.7.0_21\bin\javaw.exe (Feb 4, 2014, 8:33:58 AM)
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Result took (21159us) for query (me.prettyprint.cassandra.model.CqlQuery@1be32243) on host: localhost(127.0.0.1):9160
Column name: journal
Column Value: Oracle Magazine

Column name: publisher
Column Value: Oracle Publishing

Column name: edition
Column Value: November-December 2013

Column name: title
Column Value: Engineering as a Service

Column name: author
Column Value: David A. Kelly
```

Figure 2.14

The result of a `SELECT` statement with a `WHERE` query.

Source: Eclipse Foundation.

USING THE UPDATE STATEMENT

The UPDATE statement is used to update the column values of row(s). You update a row using an UPDATE CQL statement. The syntax of the UPDATE statement is as follows:

```
UPDATE <tablename> ( USING <option> ( AND <option> )* )? SET <assignment1> ( ','
<assignmentN> )* WHERE <where-clause>;
```

Add a method called `update()` to the `CQLClient` class. Then set the CQL UPDATE statement as the query in the `CQLQuery` object.

```
cqlQuery.setQuery("UPDATE catalog USING CONSISTENCY ALL SET 'edition' = '11/12
2013', 'author' = 'Kelley, David A.' WHERE CATALOG_ID = 'catalog1'");
```

The column in the `WHERE` clause to select the row must be the primary key column. If some other column is used, the following exception is generated:

```
Caused by: InvalidRequestException(why:Expected key 'CATALOG_ID' to be present in
WHERE clause for 'catalog')
```

UPDATE does not try to determine whether the row identified by the primary key column exists. If the row does not exist, a row is created. Run a `SELECT` query after the UPDATE statement. The result of the query indicates that the columns were updated, as shown in Figure 2.15.

The screenshot shows the Eclipse IDE's Console view with the following text output:

```
@ Javadoc Declaration Console 
<terminated> CQLClient [Java Application] C:\Program Files\Java\jdk1.7.0_21\bin\javaw.exe (Jan 22, 2014 10:26:02 AM)
Result took (66696us) for query (me.prettyprint.cassandra.model.CqlQuery@6538b14
) on host: localhost(127.0.0.1):9160
Column name: catalog_id
Column Value: catalog1

Column name: author
Column Value: Kelley, David A.

Column name: edition
Column Value: 11/12 2013

Column name: journal
Column Value: Oracle Magazine

Column name: publisher
Column Value: Oracle Publishing

Column name: title
Column Value: Engineering as a Service
```

Figure 2.15

The result of a `SELECT` statement after an `UPDATE` statement.

Source: Eclipse Foundation.

USING THE BATCH STATEMENT

The **BATCH** statement is used to run a group of modification statements (insertions, updates, deletions) in a batch as a single statement. Only **UPDATE**, **INSERT**, and **DELETE** statements may be grouped in a **BATCH** statement. Running multiple statements as a single statement saves round trips between the client and the server. The syntax of the **BATCH** statement is as follows:

```
BEGIN BATCH (USING <option> ( AND <option> )*)? <modification-stmt> ( ';' 
<modification-stmt> )* APPLY BATCH;
```

Add a method called **batch()** to the **CQLClient** class. Set a **BATCH** statement in the **CQLQuery** object. The **BATCH** statement includes two **INSERT** statements and two **UPDATE** statements.

```
cqlQuery.setQuery("BEGIN BATCH USING CONSISTENCY QUORUM UPDATE catalog SET 
'edition' = '11/12 2013', 'author' = 'Haunert, Tom' WHERE CATALOG_ID = 'catalog2' 
INSERT INTO catalog (catalog_id, journal, publisher, edition, title, author) VALUES 
('catalog3', 'Oracle Magazine', 'Oracle Publishing', 'November-December 2013', 
'', '') INSERT INTO catalog (catalog_id, journal, publisher, edition, title, author) 
VALUES ('catalog4', 'Oracle Magazine', 'Oracle Publishing', 'November-December 
2013', '', '') UPDATE catalog SET 'edition' = '11/12 2013' WHERE CATALOG_ID = 
'catalog3' APPLY BATCH");
```

The consistency level cannot be set for individual statements within a **BATCH** statement. If the consistency level is set on individual statements, the following error is generated:

Caused by: **InvalidRequestException(why:Consistency level must be set on the BATCH, not individual statements)**

Invoke the **batch()** method from the **main** method and run the **CQLClient** class in the Eclipse IDE. All the statements grouped in the **BATCH** statement are run and applied. Next, invoke the **select()** method after the **batch()** method to output all the columns in all the rows. The result of the query, shown here, indicates that the **BATCH** statement has been applied.

```
Result took (38195us) for query (me.prettyprint.cassandra.model.CqlQuery@65b57dc
c) on host: localhost(127.0.0.1):9160
Column name: catalog_id
Column Value: catalog1
Column name: author
Column Value: Kelley, David A.
Column name: edition
Column Value: 11/12 2013
```

```
Column name: journal
Column Value: Oracle Magazine
Column name: publisher
Column Value: Oracle Publishing
Column name: title
Column Value: Engineering as a Service
Column name: catalog_id
Column Value: catalog2
Column name: author
Column Value: Haunert, Tom
Column name: edition
Column Value: 11/12 2013
Column name: journal
Column Value: Oracle Magazine
Column name: publisher
Column Value: Oracle Publishing
Column name: title
Column Value: Quintessential and Collaborative
Column name: catalog_id
Column Value: catalog3
Column name: author
Column Value:
Column name: edition
Column Value: 11/12 2013
Column name: journal
Column Value: Oracle Magazine
Column name: publisher
Column Value: Oracle Publishing
Column name: title
Column Value:
Column name: catalog_id
Column Value: catalog4
Column name: author
Column Value:
Column name: edition
Column Value: November-December 2013
Column name: journal
Column Value: Oracle Magazine
Column name: publisher
Column Value: Oracle Publishing
Column name: title
Column Value:
```

USING THE DELETE STATEMENT

The `DELETE` statement is used to delete columns and rows. The syntax of the `DELETE` statement is as follows:

```
DELETE ( <selection> ( ',' <selection> )* )? FROM <tablename> WHERE <where-clause>
```

The `<selection>` items refer to the columns to be deleted. The column in the `WHERE` clause must be the primary key column. If no column is specified, all the columns are deleted. The row itself is not deleted because the primary key column is not deleted even if the primary column is specified in the `<selection>` items. Add a method called `delete()` to `CQLClient` class. Then set a query to delete the `journal` and `publisher` columns from the `catalog` table from the row with the primary key "catalog3".

```
cqlQuery.setQuery("DELETE journal, publisher from catalog WHERE  
catalog_id='catalog3'");  
cqlQuery.execute();
```

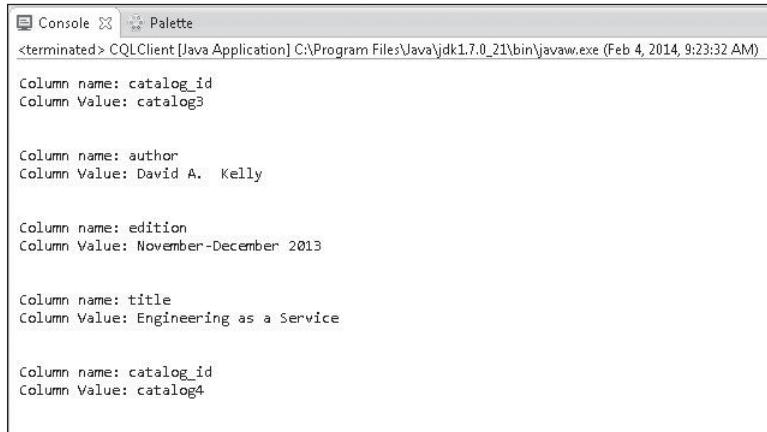
Next, set a query to delete all columns from the `catalog` table from the row with the primary key "catalog4".

```
cqlQuery.setQuery("DELETE from catalog WHERE catalog_id='catalog4'");  
cqlQuery.execute();
```

To demonstrate that the primary key column cannot be deleted, include the `catalog_id` column in the columns to delete:

```
cqlQuery.setQuery("DELETE catalog_id, journal, publisher, edition, title, author  
from catalog WHERE catalog_id='catalog4'");  
cqlQuery.execute();
```

Invoke the `delete()` method from the `main` method and run the `CQLClient` class in the Eclipse IDE. Then invoke the `select()` method after the `delete()` method to query the rows after deletion. As shown in the Eclipse IDE in Figure 2.16, the `journal` and `publisher` columns are deleted from the `catalog3` row, and all the columns have been deleted from the `catalog4` row. The primary key column is not deleted.



The screenshot shows a Java application window titled "CQLClient [Java Application]". The console output displays the results of a SELECT statement. It lists five columns: catalog_id, author, edition, title, and catalog_id. The values for catalog_id are catalog3 and catalog4. The values for author, edition, and title are David A. Kelly, November-December 2013, and Engineering as a Service respectively.

```

Console Palette
<terminated> CQLClient [Java Application] C:\Program Files\Java\jdk1.7.0_21\bin\javaw.exe (Feb 4, 2014, 9:23:32 AM)

Column name: catalog_id
Column Value: catalog3

Column name: author
Column Value: David A. Kelly

Column name: edition
Column Value: November-December 2013

Column name: title
Column Value: Engineering as a Service

Column name: catalog_id
Column Value: catalog4

```

Figure 2.16

The result of a SELECT statement after a DELETE statement.

Source: Eclipse Foundation.

USING THE ALTER COLUMNFAMILY STATEMENT

The ALTER COLUMNFAMILY or ALTER TABLE statement is used to alter the column family definitions to add columns, drop columns, change the type of existing columns, and update the table options. The syntax of the statement is as follows:

`ALTER (TABLE | COLUMNFAMILY) <tablename> <instruction>`

The `<instruction>` supports the alterations using the keywords discussed in Table 2.4.

Table 2.4 ALTER Command Keywords

Keyword	Description
ALTER	Modifies the column type
ADD	Adds a column
DROP	Drops a column
WITH	Updates table options

Add `updateCF()` and `updateCF2()` methods to the `CQLClient` class. In the `updateCF()` method, change the column type of the `edition` column to `int` in the `catalog` table using statement `ALTER COLUMNFAMILY catalog ALTER edition TYPE int`.

```
cqlQuery.setQuery("ALTER COLUMNFAMILY catalog ALTER edition TYPE int");
cqlQuery.execute();
```

Invoke the `updateCF()` method in the `main` method and run the `CQLClient` class. The `edition` column type gets changed to `int`. The value in the `edition` column is still of type `text`. A subsequent `select()` method returns the value of the `edition` column as `text`. In `updateCF2()`, change the type of the `edition` column back to `text`.

```
cqlQuery.setQuery("ALTER COLUMNFAMILY catalog ALTER edition TYPE text");
cqlQuery.execute();
```

If a column type is modified, a column value that was previously addable becomes non-addable. For example, set the column type of the `journal` column to `int`:

```
cqlQuery.setQuery("ALTER COLUMNFAMILY catalog ALTER journal TYPE int");
cqlQuery.execute();
```

Next, add a `journal` column value of type `text`:

```
cqlQuery.setQuery("INSERT INTO catalog (catalog_id, journal, publisher, edition,
title,author) VALUES ('catalog5','Oracle Magazine', 'Oracle Publishing',
'November-December 2013', '', '')");
cqlQuery.execute();
```

The following exception is generated, indicating that the `text` value cannot be added to an `int` type column:

```
HInvalidRequestException: InvalidRequestException(why:unable to make int from
'Oracle Magazine')
```

DROPPING THE COLUMN FAMILY

The `DROP TABLE` or `DROP COLUMNFAMILY` statement may be used to drop a column family, including all the data in the column family. Add a `dropCF()` method to the `CQLClient` class. Then set the query on a `CQLQuery` object to be `DROP COLUMNFAMILY catalog`, which would drop the `catalog` column family.

```
cqlQuery.setQuery("DROP COLUMNFAMILY catalog");
cqlQuery.execute();
```

Next, invoke the `dropCF()` method from the `main` method and run the `CQLClient` application. The `catalog` column family gets dropped. If only the table data is to be removed but not the table, use the `TRUNCATE` statement:

`TRUNCATE <tablename>`

DROPPING THE KEYSPACE

You can use the `DROP KEYSPACE` statement to drop a keyspace:

```
DROP KEYSPACE <identifier>
```

Add a `dropKeyspace()` method to drop a keyspace. Drop the `CQLKeyspace` by setting the `CQLQuery` object query to `DROP KEYSPACE CQLKeyspace`.

```
cqlQuery.setQuery("DROP KEYSPACE CQLKeyspace");
cqlQuery.execute();
```

Invoke the `dropKeyspace()` method from the `main` method and run the `CQLClient` application to drop the `CQLKeyspace`. The `execute()` method must be invoked after you set a query with `setQuery()`. Queries do not get added to the `CQLQuery` object so they can be run in a batch. If a keyspace is used after it has been dropped, the following error is generated:

Caused by: `InvalidRequestException(why:Keyspace 'CQLKeyspace' does not exist)`

THE CQLCLIENT APPLICATION

The `CQLClient` application appears in Listing 2.2. Some of the method invocations in the `main` method have been commented out and should be uncommented as required to run individually or in sequence.

Listing 2.2 The CQLClient Application

```
package cql;

import java.util.Iterator;
import java.util.List;

import me.prettyprint.cassandra.model.CqlQuery;
import me.prettyprint.cassandra.model.CqlRows;
import me.prettyprint.cassandra.serializers.StringSerializer;
import me.prettyprint.hector.api.Cluster;
import me.prettyprint.hector.api.Keyspace;
import me.prettyprint.hector.api.beans.ColumnSlice;
import me.prettyprint.hector.api.beans.HColumn;
import me.prettyprint.hector.api.beans.Row;
import me.prettyprint.hector.api.factory.HFactory;
import me.prettyprint.hector.api.query.QueryResult;

public class CQLClient {

    private static Cluster cluster;
    private static Keyspace keyspace;
```

```
public static void main(String[] args) {
    cluster = HFactory.getOrCreateCluster("cql-cluster",
"localhost:9160");
    /*Some of the method invocations in the main method have been commented
out and should be uncommented as required to run individually or in sequence.*/
    createKeyspace();
    createCF();
    // insert();
    // select();
    // createIndex();
    // selectFilter();
    // update();
    // select();
    // batch();
    // select();
    // delete();
    // update2();
    // select();
    // updateCF();
    // select();
    // updateCF2();
    // dropCF();
    // dropKeyspace();
}
/*Creates a Cassandra keyspace*/
private static void createKeyspace() {
    keyspace = HFactory.createKeyspace("CQLKeyspace", cluster);
}
/*Drops a Cassandra keyspace*/
private static void dropKeyspace() {
    CqlQuery cqlQuery = new CqlQuery(keyspace, StringSerializer.get(),
                                    StringSerializer.get(), StringSerializer.get());
    cqlQuery.setQuery("DROP KEYSPACE CQLKeyspace");
    cqlQuery.execute();
}
/*Creates an index*/
private static void createIndex() {
    CqlQuery cqlQuery = new CqlQuery(keyspace, StringSerializer.get(),
                                    StringSerializer.get(), StringSerializer.get());
    cqlQuery.setQuery("CREATE INDEX titleIndex ON catalog (title)");
    cqlQuery.execute();
}
```

```
/*Creates a column family*/
private static void createCF() {
    CqlQuery cqlQuery = new CqlQuery(keyspace, StringSerializer.get(),
        StringSerializer.get(), StringSerializer.get());
    cqlQuery.setQuery("CREATE COLUMNFAMILY catalog (catalog_id text
PRIMARY KEY,journal text,publisher text,edition text,title text,author text) WITH
comparator=UTF8Type AND default_validation=UTF8Type AND caching=keys_only AND
replicate_on_write=true");
    cqlQuery.execute();
    cqlQuery.setQuery("CREATE COLUMNFAMILY catalog2 (KEY text PRIMARY
KEY,journal text,publisher text,edition text,title text,author text)");
    cqlQuery.execute();
}
/*Adds data to a column family*/
private static void insert() {
    CqlQuery cqlQuery = new CqlQuery(keyspace, StringSerializer.get(),
        StringSerializer.get(), StringSerializer.get());
    cqlQuery.setQuery("INSERT INTO catalog (catalog_id, journal,
publisher, edition,title,author) VALUES ('catalog1','Oracle Magazine', 'Oracle
Publishing', 'November-December 2013', 'Engineering as a Service','David
A. Kelly')");
    cqlQuery.execute();
    cqlQuery.setQuery("INSERT INTO catalog (catalog_id, journal,
publisher, edition,title,author) VALUES ('catalog2','Oracle Magazine', 'Oracle
Publishing', 'November-December 2013', 'Quintessential and Collaborative','Tom
Haunert')");
    cqlQuery.execute();
    cqlQuery.setQuery("INSERT INTO catalog (catalog_id, journal,
publisher, edition,title,author) VALUES ('catalog3','Oracle Magazine', 'Oracle
Publishing', 'November-December 2013', 'Engineering as a Service','David A.
Kelly')");
    cqlQuery.execute();
    cqlQuery.setQuery("INSERT INTO catalog (catalog_id, publisher,
edition,title,author) VALUES ('catalog4', 'Oracle Publishing', 'November-
December 2013', 'Engineering as a Service','David A. Kelly')");
    cqlQuery.execute();
    cqlQuery.setQuery("INSERT INTO catalog2 (KEY, journal, publisher,
edition,title,author) VALUES ('catalog1','Oracle Magazine', 'Oracle Publishing',
'November-December 2013', 'Engineering as a Service','David A. Kelly')");
    cqlQuery.execute();
}
}
```

```
/*Selects data from a column family*/
private static void select() {
    CqlQuery<String, String, String> cqlQuery = new CqlQuery<String,
String, String>(
                                keyspace, StringSerializer.get(), StringSerializer.
get(),
                                StringSerializer.get());
    cqlQuery.setQuery("select * from catalog");
    QueryResult<CqlRows<String, String, String>> result = cqlQuery
        .execute();
    System.out.println(result);
    Iterator iterator = result.get().iterator();
    while (iterator.hasNext()) {
        Row row = (Row) iterator.next();
        String key = (String) row.getKey();
        ColumnSlice columnSlice = row.getColumnSlice();
        List columnList = columnSlice.getColumns();
        Iterator iter = columnList.iterator();
        while (iter.hasNext()) {
            HColumn column = (HColumn) iter.next();
            System.out.println("Column name: " +
column.getName() + " ");
            System.out.println("Column Value: " +
column.getValue());
            System.out.println("\n");
        }
    }
}

/*Selects data from a column family using a WHERE clause*/
private static void selectFilter() {
    CqlQuery<String, String, String> cqlQuery = new CqlQuery<String,
String, String>(
                                keyspace, StringSerializer.get(), StringSerializer.
get(),
                                StringSerializer.get());
    //cqlQuery.setQuery("SELECT catalog_id, journal, publisher,
edition,title,author FROM catalog WHERE title='Engineering as a Service'");
    cqlQuery.setQuery("SELECT journal, publisher, edition,title,
author FROM catalog2 WHERE KEY='catalog1'");
```

```
//cqlQuery.setQuery("SELECT catalog_id, journal, publisher,
edition,title,author FROM catalog WHERE catalog_id='catalog1'");//Generates
exception
    QueryResult<CqlRows<String, String, String>> result = cqlQuery
        .execute();
    System.out.println(result);
    Iterator iterator = result.get().iterator();
    while (iterator.hasNext()) {
        Row row = (Row) iterator.next();
        String key = (String) row.getKey();
        ColumnSlice columnSlice = row.getColumnSlice();
        List columnList = columnSlice.getColumns();
        Iterator iter = columnList.iterator();
        while (iter.hasNext()) {
            HColumn column = (HColumn) iter.next();
            System.out.println("Column name: " +
column.getName() + " ");
            System.out.println("Column Value: " +
column.getValue());
            System.out.println("\n");
        }
    }
}
/*Updates a row or rows of data in a column family*/
private static void update() {
    CqlQuery cqlQuery = new CqlQuery(keyspace, StringSerializer.get(),
        StringSerializer.get(), StringSerializer.get());
    cqlQuery.setQuery("UPDATE catalog USING CONSISTENCY ALL  SET
'edition' = '11/12 2013', 'author' = 'Kelley, David A.' WHERE CATALOG_ID =
'catalog1'");
    cqlQuery.execute();
}
/*Updates a row or rows of data in a column family*/
private static void update2() {
    CqlQuery cqlQuery = new CqlQuery(keyspace, StringSerializer.get(),
        StringSerializer.get(), StringSerializer.get());
    cqlQuery.setQuery("UPDATE catalog USING CONSISTENCY ALL  SET
'edition' = 'November-December 2013', 'author' = 'Kelley, David A.' WHERE
CATALOG_ID = 'catalog1'");
    cqlQuery.execute();
}
```

```
/*Deletes columns from a row or rows of data in a column family*/
private static void delete() {
    CqlQuery cqlQuery = new CqlQuery<String, String, String>(keyspace,
        StringSerializer.get(), StringSerializer.get(),
        StringSerializer.get());
    cqlQuery.setQuery("DELETE journal, publisher from catalog WHERE
catalog_id='catalog3'");
    cqlQuery.execute();
    cqlQuery.setQuery("DELETE from catalog WHERE
catalog_id='catalog4'");
    cqlQuery.execute();
    cqlQuery.setQuery("DELETE catalog_id, journal, publisher, edition,
title, author from catalog WHERE catalog_id='catalog4'");
    cqlQuery.execute();
}
/*Runs multiple statements in a batch*/
private static void batch() {
    CqlQuery cqlQuery = new CqlQuery(keyspace, StringSerializer.get(),
        StringSerializer.get(), StringSerializer.get());
    cqlQuery.setQuery("BEGIN BATCH USING CONSISTENCY QUORUM UPDATE
catalog SET 'edition' = '11/12 2013', 'author' = 'Haunert, Tom' WHERE CATALOG_ID =
'catalog2' INSERT INTO catalog (catalog_id, journal, publisher, edition,title,
author) VALUES ('catalog3','Oracle Magazine', 'Oracle Publishing', 'November-
December 2013', '', '') INSERT INTO catalog (catalog_id, journal, publisher,
edition,title,author) VALUES ('catalog4','Oracle Magazine', 'Oracle Publishing',
'November-December 2013', '', '') UPDATE catalog      SET 'edition' = '11/12 2013'
WHERE CATALOG_ID = 'catalog3' APPLY BATCH");
    cqlQuery.execute();
}
/*Updates a column family*/
private static void updateCF() {
    CqlQuery cqlQuery = new CqlQuery(keyspace, StringSerializer.get(),
        StringSerializer.get(), StringSerializer.get());
    cqlQuery.setQuery("ALTER COLUMNFAMILY catalog ALTER edition TYPE
int");
    cqlQuery.execute();
}
/*Updates a column family*/
private static void updateCF2() {
    CqlQuery cqlQuery = new CqlQuery(keyspace, StringSerializer.get(),
        StringSerializer.get(), StringSerializer.get());
```

```

        cqlQuery.setQuery("ALTER COLUMNFAMILY catalog ALTER edition TYPE
text");
        cqlQuery.execute();
        cqlQuery = new CqlQuery(keyspace, StringSerializer.get(),
                               StringSerializer.get(), StringSerializer.get());
        cqlQuery.setQuery("ALTER COLUMNFAMILY catalog ALTER journal TYPE
int");
        cqlQuery.execute();
        /* CF gets updated with column to a type different from column value*/
        cqlQuery.setQuery("INSERT INTO catalog (catalog_id, journal,
publisher, edition, title, author) VALUES ('catalog5', 'Oracle Magazine', 'Oracle
Publishing', 'November-December 2013', '', '')");
        cqlQuery.execute();
    }
    /*Drops a column family*/
    private static void dropCF() {
        CqlQuery cqlQuery = new CqlQuery(keyspace, StringSerializer.get(),
                                         StringSerializer.get(), StringSerializer.get());
        cqlQuery.setQuery("DROP COLUMNFAMILY catalog");
        cqlQuery.execute();
    }
}

```

NEW FEATURES IN CQL 3

CQL 3 has added support for several new features and is backward-compatible. The keyword COLUMNFAMILY has been replaced with TABLE. Some of the salient new features are discussed next.

Compound Primary Key

The CREATE TABLE command has added a provision for a multiple column primary key, also called a compound primary key. The CREATE COLUMNFAMILY example in this chapter makes use of a simple primary key—a primary key with only one column. A compound primary key for the catalog table may be declared as follows:

```

CREATE TABLE catalog (
    catalog_id text,
    journal text,
    edition text,
    title text,

```

```

author text,
PRIMARY KEY (catalog_id, journal)
);

```

The preceding statement creates a table using the `catalog_id` and `journal` columns to form a compound primary key. A table that has a compound primary key must have at least one column that is not included in the primary key.

To run an `INSERT` statement on a table with a compound primary key, each of the columns in the primary key must be specified. In addition, at least one of the non-primary key columns must be specified.

The `WHERE` clause may specify each of the columns in the compound primary key using `AND` as follows:

```
UPDATE catalog    SET  'edition' = 'November-December 2013', 'author' = 'Kelley,
David A.' WHERE CATALOG_ID = 'catalog1' AND journal='Oracle Magazine';
```

If a compound primary key is used in a `WHERE` clause, key-component columns other than the first may have a `>` (greater than) or `<` (less than) comparison. If all the preceding key-component columns have been identified with an `=` comparison, the last key-component may specify any kind of relation.

Conditional Modifications

The `CREATE` statements for `KEYSPACE`, `TABLE`, and `INDEX` support an `IF NOT EXISTS` condition. In CQL 2.0, the `CREATE` statement for `KEYSPACE`, `TABLE`, and `INDEX` throws an exception if the construct already exists.

```
CREATE KEYSPACE IF NOT EXISTS CQLKeyspace WITH replication = { 'class':
'SimpleStrategy', 'replication_factor' : 1 };
CREATE TABLE IF NOT EXISTS catalog (catalog_id text PRIMARY KEY, journal text,
publisher text,edition text,title text,author text);
```

The `DROP` statements support an `IF EXISTS` condition:

```
DROP KEYSPACE IF EXISTS CQLKeyspace;
```

The `INSERT` statement supports an `IF NOT EXISTS` condition. CQL 3 has added the provision to add a new row only if a row by the same primary key value does not already exist. The CQL 3 clause to add conditionally is `IF NOT EXISTS`. In CQL 2, the `INSERT` statement was run even if a row by the same primary key was already defined. The following CQL 3 statement adds a row only if a row identified by `catalog1` does not exist:

```
INSERT INTO catalog (catalog_id, journal, publisher, edition, title, author) VALUES  
('catalog1', 'Oracle Magazine', 'Oracle Publishing', 'November-December 2013',  
'Engineering as a Service', 'David A. Kelly') IF NOT EXISTS;
```

The UPDATE statement supports an IF condition:

```
UPDATE table_name  
USING option1 AND optionN  
SET assignment1, assignmentN  
WHERE <where-clause>  
IF column_name1 = literal AND column_nameN = literal
```

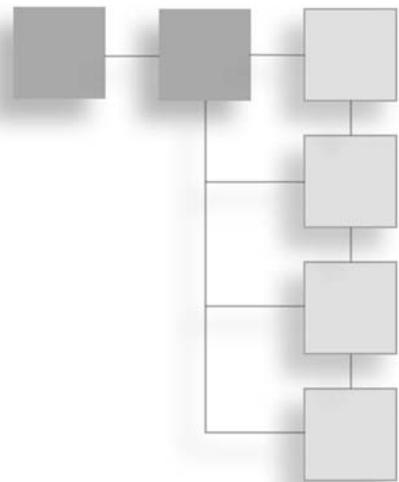
The columns in the IF clause may be different from the columns to be updated. The IF condition incurs a negligible performance overhead, as Paxos is used internally. Paxos is a consensus protocol for a distributed system.

SUMMARY

This chapter introduced Cassandra Query Language (CQL), including the CQL commands. You used CQL 2 queries with the Hector Java client to add, select, update, and delete data from a Cassandra column family. You also discovered the salient new features in CQL 3. The next chapter discusses the DataStax Java driver, which supports CQL 3.

CHAPTER 3

USING CASSANDRA WITH DATASTAX JAVA DRIVER



The DataStax Java driver is designed for CQL 3. The driver provides connection pooling, node discovery, automatic failover, and load balancing. The driver supports prepared statements. Queries can be run synchronously or asynchronously. The driver provides a layered architecture. At the bottom is the core layer, which handles connections to the Cassandra cluster. The core layer exposes a low-level API on which a higher-level layer may be built. In this chapter, you will connect with Cassandra server using the DataStax Java driver and perform create, read, update, delete (CRUD) operations on the database.

OVERVIEW OF DATASTAX JAVA DRIVER

The main package for the DataStax Java driver core is `com.datastax.driver.core`. The main classes in the package are shown in Figure 3.1.

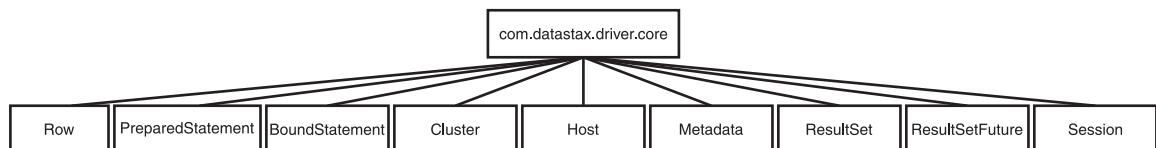


Figure 3.1
DataStax Java Driver Classes.

The classes shown in Figure 3.1 are discussed in Table 3.1.

Table 3.1 Classes in the `com.datastax.driver.core` Package

Class	Description
Row	A CQL row in a result set.
PreparedStatement	Represents a prepared statement—a query with bound variables that has been prepared by the database.
BoundStatement	A prepared statement with values bound to the bind variables.
Cluster	The entry point of the driver. Keeps information on the state and topology of the cluster.
Host	Represents a Cassandra node.
Metadata	Metadata of the connected cluster.
ResultSet	Result set of a query.
ResultSetFuture	A future on a result set.
Session	Encapsulates connections to a cluster, making it query-able.

SETTING THE ENVIRONMENT

To set the environment, you must download the following software:

- DataStax Java driver for Apache Cassandra—Core from <http://mvnrepository.com/artifact/com.datastax.cassandra/cassandra-driver-core/2.0.1>
- Eclipse IDE for Java EE developers from <http://www.eclipse.org/downloads/moreinfo/jee.php>
- Apache Cassandra apache-cassandra-2.0.4-bin.tar.gz or a later version from <http://cassandra.apache.org/download/>
- Java SE 7 from <http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html>
- Metrics Core metrics-core-3.0.1.jar from <http://mvnrepository.com/artifact/com.codahale.metrics/metrics-core/3.0.1>

Then follow these steps:

1. Extract the Apache Cassandra TAR file to a directory (for example, C:\Cassandra\apache-cassandra-2.0.4).
2. Add the bin folder, C:\Cassandra\apache-cassandra-2.0.4\bin, to the PATH environment variable.
3. Start Apache Cassandra server with the following command:
`cassandra -f`

CREATING A JAVA PROJECT

In this section, you will use the DataStax Java driver in a Java application for which you need to create a Java project in Eclipse IDE. Follow these steps:

1. Select File > New > Other.
2. In the New window, select the Java Project wizard as shown in Figure 3.2. Then click Next.

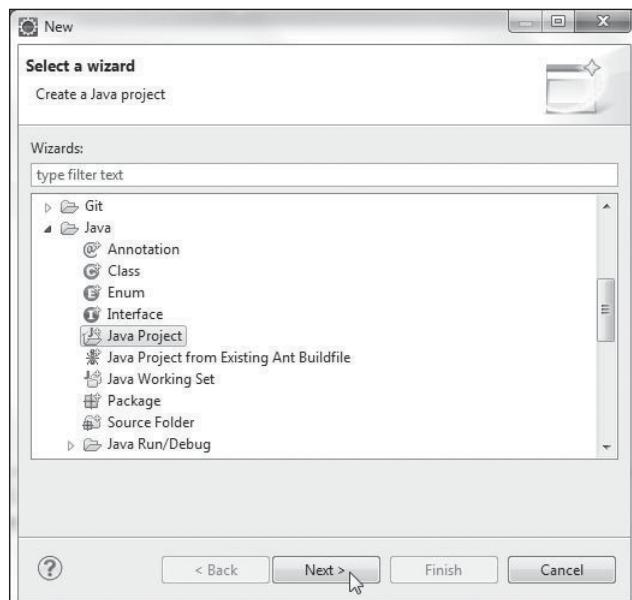


Figure 3.2
Selecting the Java Project wizard.

Source: Eclipse Foundation.

3. In the Create a Java Project screen, specify a project name (Datastax) and choose a directory location or select the Use Default location checkbox. Then select the default JRE, which has been set to 1.7, and click Next, as shown in Figure 3.3.

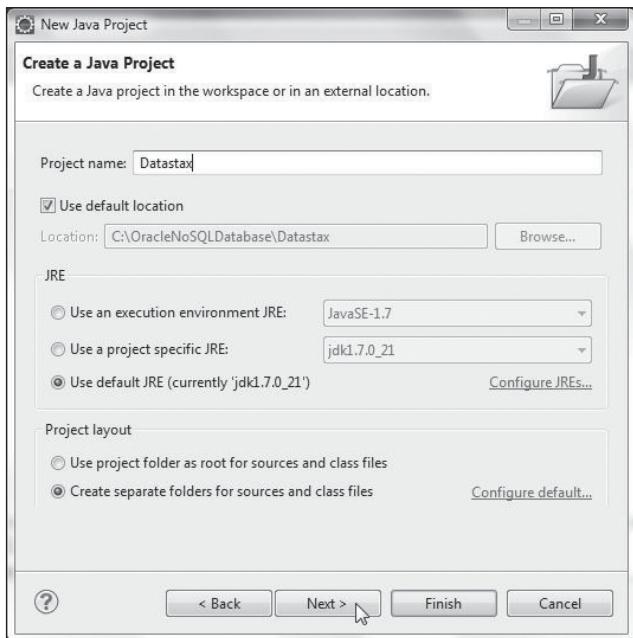


Figure 3.3
Creating a new Java project.
Source: Eclipse Foundation.

4. Select the default options in the Java Settings screen and click Finish, as shown in Figure 3.4. A Java project is created.

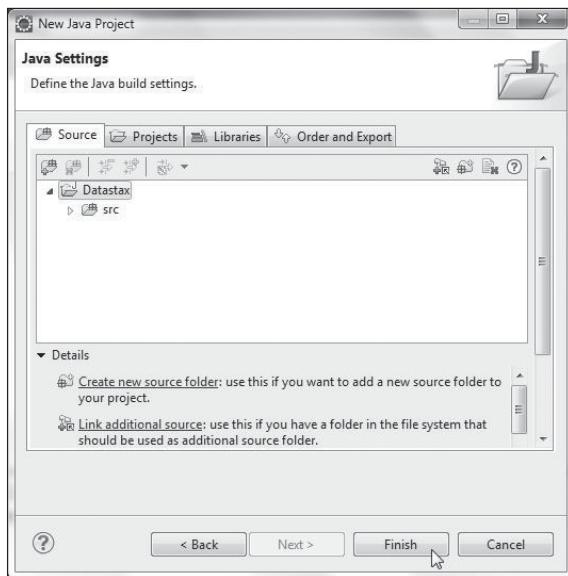


Figure 3.4
The Java Settings screen.
Source: Eclipse Foundation.

5. Add a Java class to the project. To begin, choose File > New > Other. Then, in the New dialog box, select Java > Java Class and click Next, as shown in Figure 3.5.

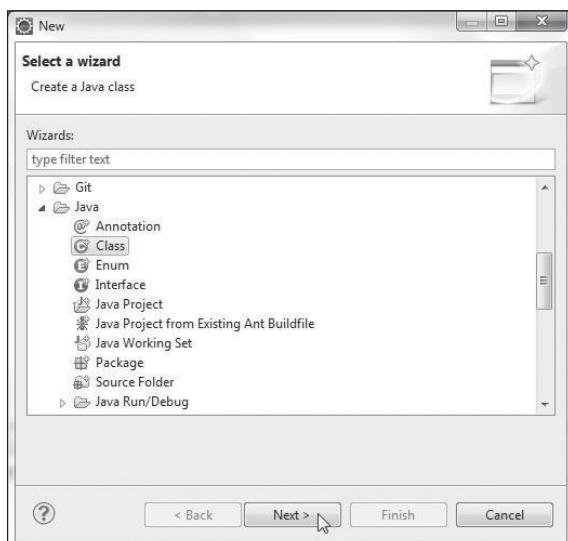


Figure 3.5
Selecting the Java Class wizard.
Source: Eclipse Foundation.

6. In the New Java Class wizard, select a source folder (Datastax/src) and specify the package as datastax. Then specify the Java class name (CQLClient) and click Finish, as shown in Figure 3.6. A Java class is added to the Java project, as shown in the Package Explorer in Figure 3.7.

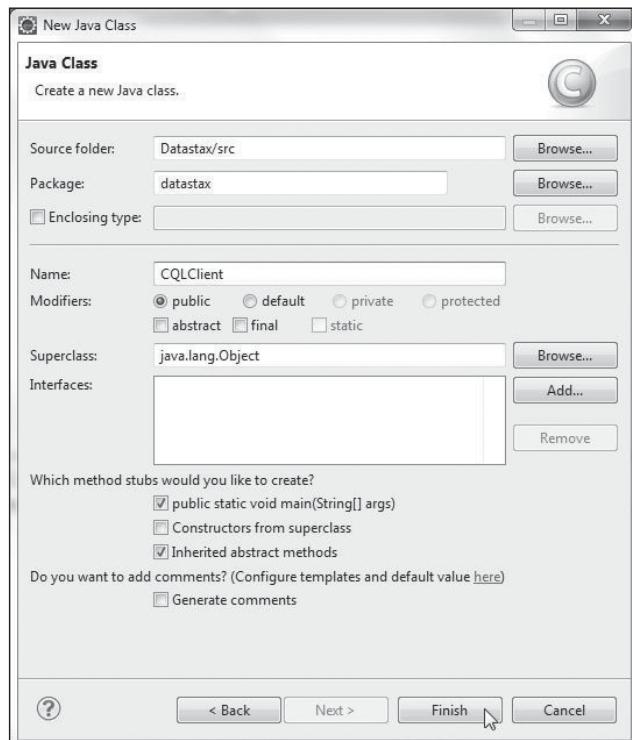


Figure 3.6
Creating a new Java class.

Source: Eclipse Foundation.

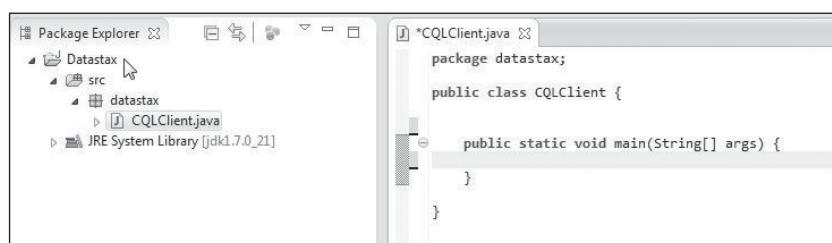


Figure 3.7
The new Java class.

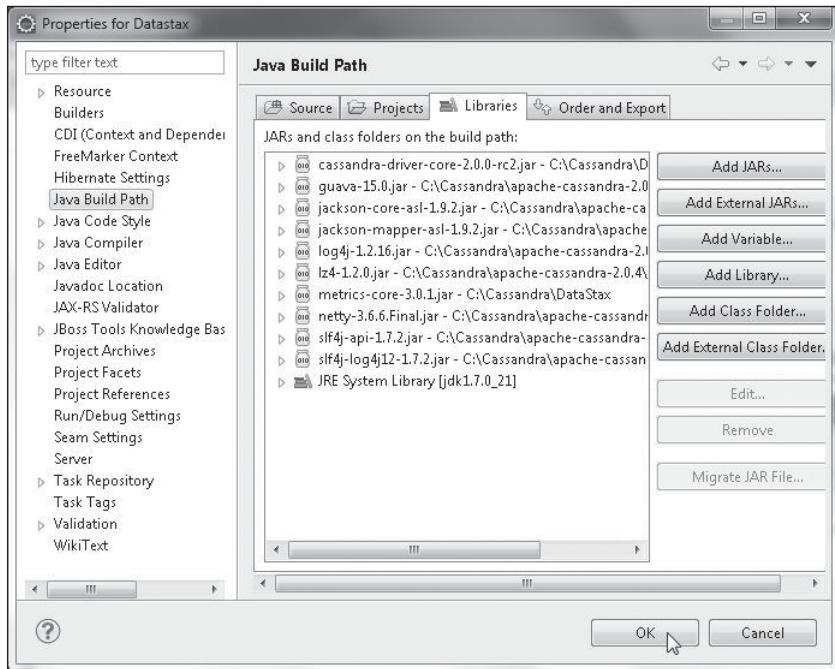
Source: Eclipse Foundation.

7. To be able to access Cassandra from the Java application using DataStax, you need to add some JAR files to the application's Java build path. Right-click the Datastax project node in Package Explorer and select Properties. Then, in the Properties window, select the Java Build Path node and click the Add External JARs button to add external JAR files. Finally, add the JAR files listed in Table 3.2.

Table 3.2 JAR Files

JAR File	Description
cassandra-driver-core-2.0.0-rc2.jar	A driver for Apache Cassandra designed exclusively for CQL 3.
jackson-core-asl-1.9.2.jar	Jackson, a high-performance JSON processor (parser and generator).
jackson-mapper-asl-1.9.2.jar	Data Mapper, a high-performance data-binding package built on Jackson JSON processor.
lz4-1.2.0.jar	Java ports and bindings of the LZ4 compression algorithm.
guava-15.0.jar	Google's core libraries used in Java projects: collections, caching, primitives support, concurrency, common annotations, string processing, and I/O, to list a few.
metrics-core-3.0.1.jar	A Java library for getting metrics in production. The Metrics Core library required is different from the version packaged with Cassandra.
netty-3.6.6.Final.jar	NIO client server framework for efficient development of network applications.
log4j-1.2.16.jar	A logging library for Java.
slf4j-api-1.7.2.jar	Simple Logging Framework for Java (SLF4J), which provides abstraction for various logging frameworks.
slf4j-log4j12-1.7.2.jar	Provides the SLF4J-log4j binding.

8. The external JAR files required for accessing Cassandra from a DataStax Java client application are shown in the Eclipse IDE Properties wizard. Click OK after adding the required JAR files, as shown in Figure 3.8.

**Figure 3.8**

The JAR files in the Datastax project.

Source: Eclipse Foundation.

In later sections, you will develop a Java application to connect with the Cassandra server using the DataStax Java driver and run CQL 3 queries to create, select, update, and delete data from the server. First, however, we will discuss how to connect with the Cassandra server.

CREATING A CONNECTION

In this section, you will connect to the Cassandra server. To begin, add a `connection()` method to the `CQLClient` application. In the method, create an instance of `Cluster`, which is the main entry point for the driver. The `Cluster` instance maintains a connection with one of the server nodes to keep information on the state and current topology of the cluster. The driver discovers all the nodes in the cluster using auto-discovery of nodes, including new nodes that join later. Build a `Cluster.Builder` instance, which is a helper class to build `Cluster` instances, using the static method `builder()`.

You need to provide the connection address of at least one of the nodes in the Cassandra cluster for the DataStax driver to be able to connect with the cluster and discover other nodes in the cluster using auto-discovery. Using the `addContactPoint(String)` method of `Cluster.Builder`, add the address of the Cassandra server running on the `localhost` (`127.0.0.1`). Next, invoke the `build()` method to build the `Cluster` using the configured address(es). The methods may be invoked in sequence, as you don't need the intermediary `Cluster.Builder` instance.

```
Cluster cluster = Cluster.builder().addContactPoint("127.0.0.1").build();
```

Get the metadata of the cluster using the `getMetadata()` method. The metadata includes the nodes in the cluster with their status. Creating a `Cluster` instance does not by itself create a connection with the server. Getting metadata requires a connection with the server for which a connection is established, unless the `getMetadata()` method is invoked after the `init()` or `connect()` method is invoked, which establishes a connection with the server. Obtain the cluster name using the `getClusterName()` method in the `Metadata` class. The `getAllHosts()` method returns a set of all the known hosts in the cluster. Iterate over the set to output the hosts' data center, address, and rack. The `Cluster` class provides the methods discussed in Table 3.3 to connect the Cassandra server.

Table 3.3 Cluster Class Methods

Method	Description
<code>connect()</code>	Creates a new session on the cluster. A session maintains multiple connections to the cluster.
<code>connect(String keyspace)</code>	Creates a new session on the cluster and sets it to the specified keyspace.

Next, invoke the `connect()` method to create a session on the cluster. A session is represented with the `Session` class, which holds multiple connections to the cluster. A `Session` instance is used to query the cluster. The `Session` instance provides policies on which node in the cluster to use for querying the cluster. The default policy is to use a round-robin on all the nodes in the cluster. `Session` is also used to handle retries of failed queries. `Session` instances are thread-safe, and a single instance is sufficient for an

application. But a separate Session instance is required if connecting to multiple keyspaces, as a single Session instance is specific to a particular keyspace only.

```
Session session = cluster.connect();
```

The initial CQLClient application to create a connection with the server appears in Listing 3.1. You will develop the application in upcoming sections to add a keyspace, a table and run CQL 3 queries.

Listing 3.1 CQLClient Class

```
package datastax;

import com.datastax.driver.core.Cluster;
import com.datastax.driver.core.Host;
import com.datastax.driver.core.Metadata;
import com.datastax.driver.core.Session;

public class CQLClient {

    private static Cluster cluster;
    private static Session session;
    public static void main(String[] args) {
        connection();
    }
    private static void connection() {
        cluster = Cluster.builder().addContactPoint("127.0.0.1").
build();
        Metadata metadata = cluster.getMetadata();
        System.out.printf("Connected to cluster: %s\n",
                           metadata.getClusterName());
        for (Host host : metadata.getAllHosts()) {
            System.out.printf("Datacenter: %s; Host: %s; Rack: %s
\n",
                               host.getDatacenter(), host.getAddress(),
                               host.getRack());
        }
        session = cluster.connect();
    }
}
```

Right-click the CQLClient application and select Run As > Java Application, as shown in Figure 3.9.

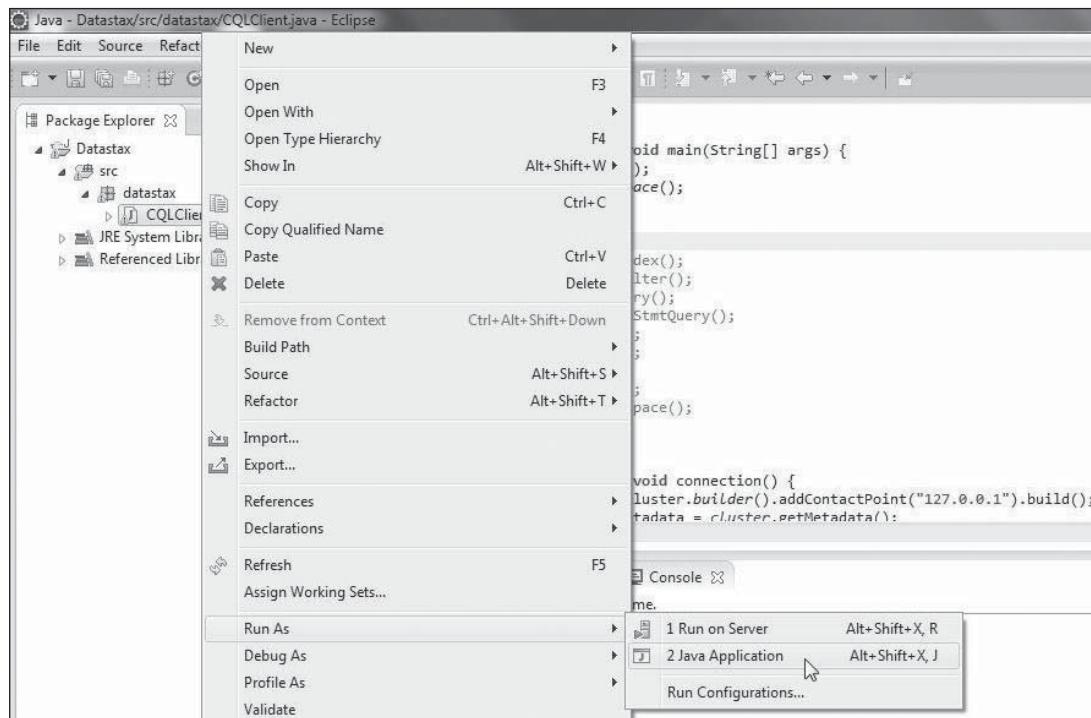


Figure 3.9
Running the CQLClient Java application.

Source: Eclipse Foundation.

A connection with the server is established and the cluster's data center, host, and rack information is output, as shown in Figure 3.10.

```
Console | Palette
CQLClient (1) [Java Application] C:\Program Files\Java\jdk1.7.0_21\bin\javaw.exe
log4j:WARN No appenders could be found for logger (com.datastax.driver.core.Cluster)
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig
Connected to cluster: Test Cluster
Datacenter: datacenter1; Host: /127.0.0.1; Rack: rack1
```

Figure 3.10
Cluster information.

Source: Eclipse Foundation.

If the Cassandra server is not running, the following exception is generated when a connection is attempted:

```
com.datastax.driver.core.exceptions.NoHostAvailableException: All host(s) tried
for query failed (tried: /127.0.0.1 (com.datastax.driver.core.
TransportException: [/127.0.0.1] Cannot connect))
at com.datastax.driver.core.ControlConnection.reconnectInternal
(ControlConnection.java:179)
at com.datastax.driver.core.ControlConnection.connect(ControlConnection.
java:77)
at com.datastax.driver.core.Cluster$Manager.init(Cluster.java:890)
at com.datastax.driver.core.Cluster$Manager.access$100(Cluster.java:806)
at com.datastax.driver.core.Cluster.getMetadata(Cluster.java:217)
at datastax.CQLClient.connection(CQLClient.java:43)
at datastax.CQLClient.main(CQLClient.java:23)
```

OVERVIEW OF THE Session CLASS

The Session class provides several methods to prepare and run queries on the server. The methods to prepare or run queries are discussed in Table 3.4.

Table 3.4 Session Class Methods

Method	Description
prepare(String query)	Prepares the CQL 3 query string to return a prepared statement represented by the PreparedStatement interface
prepare(RegularStatement statement)	Prepares the CQL 3 query provided as a regular statement represented by the RegularStatement class to return a prepared statement
execute(Statement statement)	Executes the query provided as a Statement object to return a result set represented by the ResultSet interface
execute(String query)	Executes the query provided as a String object to return a result set
execute(String query, Object... values)	Executes the query provided as a String object and uses the specified values to return a result set
executeAsync(Statement statement)	Executes the query provided as a Statement object asynchronously to return a result set

<code>executeAsync(String query)</code>	Executes the query provided as a <code>String</code> object asynchronously to return a result set
<code>executeAsync(String query, Object... values)</code>	Executes the query provided as a <code>String</code> object and uses the specified values asynchronously to return a result set

You need to create a keyspace in which to store tables. In the next section, you will create a keyspace.

CREATING A KEYSPACE

In this section, you will create a keyspace using the `Session` object to run a CQL 3 statement. Add a `createKeyspace()` method to create a keyspace in the `CQLClient` application. CQL 3 has added support to run `CREATE` statements conditionally, which is only if the object to be constructed does not already exist. The `IF NOT EXISTS` clause is used to create conditionally. Create a keyspace called `datastax` using replication with the strategy class `SimpleStrategy` and a replication factor of 1.

```
private static void createKeyspace() {
    session.execute("CREATE KEYSPACE IF NOT EXISTS datastax WITH
replication "
    + "= {'class':'SimpleStrategy', 'replication_factor':1};");
}
```

Invoke the `createKeyspace()` method in the `main` method and run the `CQLClient` application to create a keyspace.

CREATING A TABLE

Next, you will create a column family, which is called a table in CQL 3. Add a `createTable()` method to `CQLClient`. The `CREATE TABLE` command also supports `IF NOT EXISTS` to create a table conditionally. CQL 3 has added the provision to create a compound primary key—that is, a primary key created from multiple component primary key columns. In a compound primary key, the first column is called the *partition key*. To demonstrate different aspects of using a compound primary key, create three different tables, `catalog`, `catalog2`, and `catalog3`. Each of the tables has columns `catalog_id`, `journal`, `publisher`, `edition`, `title`, and `author`. In the `catalog` table, the compound

primary key is made from the `catalog_id` and `journal` columns, with `catalog_id` being the partition key. In `catalog2`, the same two columns are used in the compound key, but the `journal` column is used as the partition key. In `catalog3`, three columns are used in the compound key: `catalog_id`, `journal`, and `publisher`. Invoke the `execute(String)` method to create three tables, `catalog`, `catalog2`, and `catalog3`, as follows:

```
private static void createTable() {
    session.execute("CREATE TABLE IF NOT EXISTS datastax.catalog (catalog_id text,
        journal text,publisher text, edition text,title text,author text,PRIMARY KEY
        (catalog_id, journal))");
    session.execute("CREATE TABLE IF NOT EXISTS datastax.catalog2 (catalog_id text,
        journal text,publisher text, edition text,title text,author text,PRIMARY KEY
        (journal, catalog_id))");
    session.execute("CREATE TABLE IF NOT EXISTS datastax.catalog3 (catalog_id text,
        journal text,publisher text, edition text,title text,author text,PRIMARY KEY
        (journal, catalog_id, publisher))");
}
```

Prefix the table name with the keyspace name. Invoke the `createTable()` method in the `main` method and run the CQLClient application to create the three tables.

RUNNING THE INSERT STATEMENT

Next, you will add data to the three tables—`catalog`, `catalog2`, and `catalog3`—using the `INSERT` statement. Use the `IF NOT EXISTS` keyword to add rows conditionally. When a compound primary key is used, all the component primary key columns must be specified, including the values for the compound key columns. For example, run the following CQL 3 query using a `Session` object:

```
session.execute("INSERT INTO datastax.catalog (catalog_id, publisher, edition,
    title,author) VALUES ('catalog1', 'Oracle Publishing', 'November-December 2013',
    'Engineering as a Service', 'David A. Kelly') IF NOT EXISTS");
```

Because the primary key component column, `journal`, is not specified in the CQL 3 statement, the following exception is generated.

```
Exception in thread "main" com.datastax.driver.core.exceptions.
InvalidQueryException: Missing mandatory PRIMARY KEY part journal
```

Add an `insert()` method to the `CQLClient` class and invoke the method in the `main` method. Then add three rows identified by the row IDs `catalog1`, `catalog2`, and

catalog3 to each of the tables (catalog, catalog2, and catalog3). For example, the three rows are added to the catalog table as follows:

```
private static void insert() {
    session.execute("INSERT INTO datastax.catalog (catalog_id, journal, publisher,
edition,title,author) VALUES ('catalog1','Oracle Magazine', 'Oracle Publishing',
'November-December 2013', 'Engineering as a Service', 'David A. Kelly') IF NOT
EXISTS");
    session.execute("INSERT INTO datastax.catalog (catalog_id, journal, publisher,
edition,title,author) VALUES ('catalog2','Oracle Magazine', 'Oracle Publishing',
'November-December 2013', 'Quintessential and Collaborative', 'Tom Haunert') IF NOT
EXISTS");
    session.execute("INSERT INTO datastax.catalog (catalog_id, journal, publisher)
VALUES ('catalog3', 'Oracle Magazine', 'Oracle Publishing') IF NOT EXISTS");
}
```

Run the CQLClient application to add the three rows of data to each of the tables.

RUNNING A SELECT STATEMENT

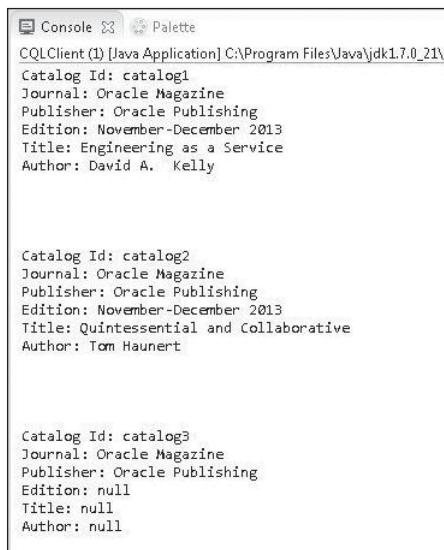
Next, you will run a SELECT statement to select columns from a table. Add a select() method to run SELECT statement(s). First, select all the columns from the catalog table using * for column selection:

```
ResultSet results = session.execute("select * from datastax.catalog");
```

A row in the result set, represented by the ResultSet interface, is represented with the Row class. Iterate over the result set to output the column value or each of the columns:

```
private static void select() {
    ResultSet results =
        session.execute("select * from datastax.catalog");
    for (Row row : results) {
        System.out.println("Catalog Id: " + row.getString("catalog_id"));
        System.out.println("Journal: " + row.getString("journal"));
        System.out.println("Publisher: " + row.getString("publisher"));
        System.out.println("Edition: " + row.getString("edition"));
        System.out.println("Title: " + row.getString("title"));
        System.out.println("Author: " + row.getString("author"));
        System.out.println("\n");
        System.out.println("\n");
    }
}
```

Run the CQLClient application to select the rows from the `datastax.catalog` table and output the columns as shown in Figure 3.11.



The screenshot shows the CQLClient application interface with three rows of catalog data. The first row has a Catalog ID of `catalog1`, Journal of `Oracle Magazine`, Publisher of `Oracle Publishing`, Edition of `November-December 2013`, Title of `Engineering as a Service`, and Author of `David A. Kelly`. The second row has a Catalog ID of `catalog2`, Journal of `Oracle Magazine`, Publisher of `Oracle Publishing`, Edition of `November-December 2013`, Title of `Quintessential and Collaborative`, and Author of `Tom Haunert`. The third row has a Catalog ID of `catalog3`, Journal of `Oracle Magazine`, Publisher of `Oracle Publishing`, Edition of `null`, Title of `null`, and Author of `null`.

```
Console Palette
CQLClient (1) [Java Application] C:\Program Files\Java\jdk1.7.0_21\
Catalog Id: catalog1
Journal: Oracle Magazine
Publisher: Oracle Publishing
Edition: November-December 2013
Title: Engineering as a Service
Author: David A. Kelly

Catalog Id: catalog2
Journal: Oracle Magazine
Publisher: Oracle Publishing
Edition: November-December 2013
Title: Quintessential and Collaborative
Author: Tom Haunert

Catalog Id: catalog3
Journal: Oracle Magazine
Publisher: Oracle Publishing
Edition: null
Title: null
Author: null
```

Figure 3.11

Result output with SELECT statement.

Source: Eclipse Foundation.

CQL 3 has added support for the `ORDER BY` clause to order the result in ascending order (ASC) by default. But the `ORDER BY` clause is supported only if the partition key is restricted by an `EQ` or `IN`. To demonstrate, run the following query with `ORDER BY` on the `catalog_id` column:

```
ResultSet results = session.execute("select * from datastax.catalog ORDER BY catalog_id DESC");
```

This generates the following exception:

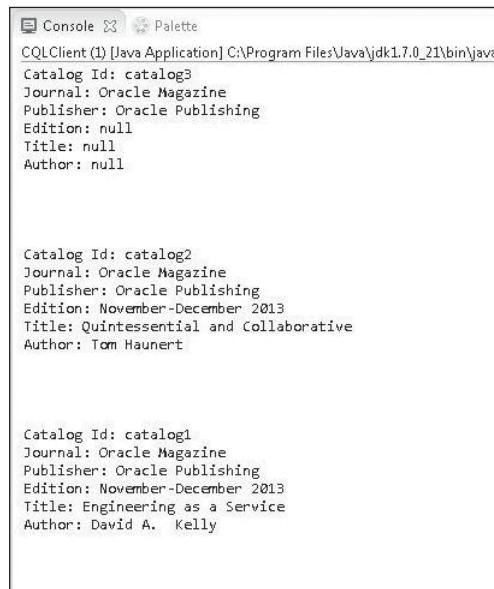
Caused by: com.datastax.driver.core.exceptions.InvalidQueryException: ORDER BY is only supported when the partition key is restricted by an EQ or an IN.

The `catalog_id` column is the partition key in the `catalog` table, so if `ORDER BY` is to be used on that table, then the `catalog_id` column must be restricted with an `EQ` or `IN`. But restricting `catalog_id` would not be useful to demonstrate ordering of rows, as the result has only one row. Instead, use the `catalog2` table, which has the `journal` column as the

partition column. Restrict the journal column and use the ORDER BY clause on the catalog_id column as follows:

```
ResultSet results = session.execute("select * from datastax.catalog2 WHERE journal='Oracle Magazine' ORDER BY catalog_id DESC");
```

When the application is run, the rows are selected in descending order of the catalog_id—that is, catalog3, then catalog2, and then catalog1—as indicated by the output in Figure 3.12.



The screenshot shows the Eclipse IDE's Console tab with three distinct sections of output, each representing a row from the catalog2 table. The first section (catalog_id: catalog3) contains fields: Catalog Id: catalog3, Journal: Oracle Magazine, Publisher: Oracle Publishing, Edition: null, Title: null, Author: null. The second section (catalog_id: catalog2) contains fields: Catalog Id: catalog2, Journal: Oracle Magazine, Publisher: Oracle Publishing, Edition: November-December 2013, Title: Quintessential and Collaborative, Author: Tom Haunert. The third section (catalog_id: catalog1) contains fields: Catalog Id: catalog1, Journal: Oracle Magazine, Publisher: Oracle Publishing, Edition: November-December 2013, Title: Engineering as a Service, Author: David A. Kelly.

```
Console [3] Palette
CQLClient (1) [Java Application] C:\Program Files\Java\jdk1.7.0_21\bin\java
Catalog Id: catalog3
Journal: Oracle Magazine
Publisher: Oracle Publishing
Edition: null
Title: null
Author: null

Catalog Id: catalog2
Journal: Oracle Magazine
Publisher: Oracle Publishing
Edition: November-December 2013
Title: Quintessential and Collaborative
Author: Tom Haunert

Catalog Id: catalog1
Journal: Oracle Magazine
Publisher: Oracle Publishing
Edition: November-December 2013
Title: Engineering as a Service
Author: David A. Kelly
```

Figure 3.12
Result for SELECT with ORDER BY.

Source: Eclipse Foundation.

If the compound primary key has more than two columns, the ORDER BY condition must be used on the second column. To demonstrate, use ORDER BY on the publisher column in the catalog3 table, which has three columns—journal, catalog_id, and publisher, with publisher being the third column.

```
ResultSet results = session.execute("select * from datastax.catalog3 WHERE journal='Oracle Magazine' ORDER BY publisher");
```

When the preceding query is run, the following exception is generated:

Caused by: com.datastax.driver.core.exceptions.InvalidQueryException: Order by currently only support the ordering of columns following their declared order in the PRIMARY KEY

To demonstrate the use of ORDER BY with more than two columns in the primary key, specify the EQ on the partition key, which is journal in catalog3, and the ORDER BY on catalog_id, which is the second column in the compound primary key:

```
ResultSet results = session.execute("select * from datastax.catalog3 WHERE journal='Oracle Magazine' ORDER BY catalog_id");
```

When the application is run, the rows are selected in ascending order of the catalog_id—catalog1, then catalog2, followed by catalog3. (Refer to Figure 3.11.)

Next, we will discuss filtering a query with the WHERE clause. The columns used for filtering in the WHERE clause must be indexed. The primary key column(s) is indexed automatically, so the primary key column(s) can be used in the WHERE clause as such. If a non-indexed column is used in the WHERE clause, the following exception is generated:

```
com.datastax.driver.core.exceptions.InvalidQueryException: No indexed columns present in by-columns clause with Equal operator
```

In the next section, you will create a secondary index on a non-primary key column title in the catalog table.

CREATING AN INDEX

A new secondary index on a column in a table is created with the CREATE INDEX command. Add a `createIndex()` method in the `CQLQuery` class and invoke the method in the `main` method. Then add a secondary index to the title column using the CREATE INDEX command. The CREATE INDEX command supports the IF NOT EXISTS clause. The IF NOT EXISTS clause does not take into consideration whether a previously created index by the same name is for the same table definition as the new index or a different table definition. For example, if a previously created index named `titleIndex` is for some table definition and a new index named `titleIndex` is for a different table definition, and the IF NOT EXISTS clause is used, it would still not create the new index named `titleIndex` even though the new index has a different table definition. The IF NOT EXISTS clause should be used only if a previously created index by the same name could not have been created or is unlikely to have been created previously for another table with a different table definition (perhaps a primary key with a single column instead of a compound primary key).

```
private static void createIndex() {
    session.execute("CREATE INDEX titleIndex ON datastax.catalog (title)");
}
```

Run the CQLQuery application to create a secondary index on the title column in the catalog table. If the following exception is generated, it is better to drop the index and create it again if it is not certain that the index by the same name was created for the same table as required.

```
com.datastax.driver.core.exceptions.InvalidQueryException: Index already exists
```

SELECTING WITH SELECT AND A WHERE FILTER

You can refine a SELECT query using a WHERE clause. The WHERE clause must specify the primary key component column(s), which is automatically indexed, or a column with a secondary index. We will discuss using SELECT with WHERE using different columns. Add a selectFilter() method to the CQLQuery class and invoke the method in the main method. In the first example, select all the columns using the title column in the WHERE clause. The title column has a secondary index defined on it and therefore can be used in the WHERE clause.

```
private static void selectFilter() {
    ResultSet results = session.execute("SELECT catalog_id, journal,
    publisher, edition, title, author FROM datastax.catalog WHERE title='Engineering as
    a Service'");
    for (Row row : results) {
        System.out.println("Journal: " + row.getString("journal"));
        System.out.println("Publisher: " + row.getString("publisher"));
        System.out.println("Edition: " + row.getString("edition"));
        System.out.println("Title: " + row.getString("title"));
        System.out.println("Author: " + row.getString("author"));
        System.out.println("\n");
        System.out.println("\n");
    }
}
```

The output from the preceding query is as follows:

```
Journal: Oracle Magazine
Publisher: Oracle Publishing
Edition: November-December 2013
Title: Engineering as a Service
Author: David A. Kelly
```

Select all columns from the catalog table where the catalog_id is "catalog2". The catalog_id is the partition key in the catalog table. Iterate over the result set to output the columns:

```
private static void selectFilter() {
    ResultSet results = session.execute("SELECT catalog_id, journal,
publisher, edition, title, author FROM datastax.catalog WHERE
catalog_id='catalog2'");
    for (Row row : results) {
        System.out.println("Journal: " + row.getString("journal"));
        System.out.println("Publisher: " + row.getString
("publisher"));
        System.out.println("Edition: " + row.getString("edition"));
        System.out.println("Title: " + row.getString("title"));
        System.out.println("Author: " + row.getString("author"));
        System.out.println("\n");
        System.out.println("\n");
    }
}
```

The following output is generated:

```
Journal: Oracle Magazine
Publisher: Oracle Publishing
Edition: November-December 2013
Title: Quintessential and Collaborative
Author: Tom Haunert
```

Different versions of the selectFilter() method are included in the code listing for CQLClient at the end of this chapter with some or all versions commented out. De-comment the version that is to be tested. If the primary key is a compound key, the partition key can be used in the WHERE clause without the other primary key component columns. However, a non-partition key cannot be used alone in a similar manner. To demonstrate, run the following query:

```
ResultSet results = session.execute("SELECT catalog_id, journal, publisher,
edition, title, author FROM datastax.catalog WHERE journal='Oracle Magazine'");
```

The following exception is generated:

```
Caused by: com.datastax.driver.core.exceptions.InvalidQueryException: Cannot
execute this query as it might involve data filtering and thus may have unpredictable
performance. If you want to execute this query despite the performance
unpredictability, use ALLOW FILTERING
```

To run the preceding query, add ALLOW FILTERING to the SELECT statement:

```
ResultSet results = session.execute("SELECT catalog_id, journal, publisher,  
edition,title,author FROM datastax.catalog WHERE journal='Oracle Magazine' ALLOW  
FILTERING");
```

The following output is generated with the preceding query:

```
Journal: Oracle Magazine  
Publisher: Oracle Publishing  
Edition: November-December 2013  
Title: Engineering as a Service  
Author: David A. Kelly  
Journal: Oracle Magazine  
Publisher: Oracle Publishing  
Edition: November-December 2013  
Title: Quintessential and Collaborative  
Author: Tom Haunert  
Journal: Oracle Magazine  
Publisher: null  
Edition: null  
Title: null  
Author: null
```

All the component columns in a compound primary key can be used in the WHERE clause in any order, as in the following example:

```
ResultSet results = session.execute("SELECT catalog_id, journal, publisher,  
edition,title,author FROM datastax.catalog WHERE journal='Oracle Magazine' AND  
catalog_id='catalog2'");
```

This query generates the following output:

```
Journal: Oracle Magazine  
Publisher: Oracle Publishing  
Edition: November-December 2013  
Title: Quintessential and Collaborative  
Author: Tom Haunert
```

Another example of using the WHERE clause is using the IN clause with a primary key column:

```
ResultSet results = session.execute("SELECT catalog_id, journal, publisher,  
edition,title,author FROM datastax.catalog WHERE catalog_id IN ('catalog2',  
'catalog3')");
```

The preceding query generates the following output:

```
Journal: Oracle Magazine
Publisher: Oracle Publishing
Edition: November-December 2013
Title: Quintessential and Collaborative
Author: Tom Haunert
Journal: Oracle Magazine
Publisher: null
Edition: null
Title: null
Author: null
```

The IN predicates can be used only on primary key columns. For example, if the IN predicate is used on the title column, which is an indexed column, the following exception is generated:

```
Exception in thread "main" com.datastax.driver.core.exceptions.
InvalidQueryException: IN predicates on non-primary-key columns (title) is not yet
supported
```

In CQL 3, the WHERE clause allows greater than (>) and less than (<) relations on all the columns other than the first, which still must have the = comparison. In the following example, the second column in the WHERE clause has the > relation:

```
ResultSet results = session .execute("SELECT catalog_id, journal, publisher,
edition,title,author FROM datastax.catalog2 WHERE journal='Oracle Magazine' AND
catalog_id > 'catalog1'");
```

The output from the preceding query is as follows:

```
Catalog Id: catalog2
Journal: Oracle Magazine
Publisher: Oracle Publishing
Edition: November-December 2013
Title: Quintessential and Collaborative
Author: Tom Haunert
Catalog Id: catalog3
Journal: Oracle Magazine
Publisher: Oracle Publishing
Edition: null
Title: null
Author: null
```

The last column in a WHERE clause can have any type of relation if all the preceding columns have been specified with the = comparison. In the following example, the last column has the \geq relation with all the preceding columns being identified with the = comparison:

```
ResultSet results = session .execute("SELECT catalog_id, journal, publisher,
edition,title,author FROM datastax.catalog2 WHERE journal='Oracle Magazine' AND
catalog_id >= 'catalog1'");
```

The result of the query is as follows:

Catalog Id: catalog1

Journal: Oracle Magazine

Publisher: Oracle Publishing

Edition: November-December 2013

Title: Engineering as a Service

Author: David A. Kelly

Catalog Id: catalog2

Journal: Oracle Magazine

Publisher: Oracle Publishing

Edition: November-December 2013

Title: Quintessential and Collaborative

Author: Tom Haunert

Catalog Id: catalog3

Journal: Oracle Magazine

Publisher: Oracle Publishing

Edition: null

Title: null

Author: null

If the primary key is a compound key and the partition key is used in the WHERE clause, only the EQ and IN relations are supported on the partition key. To demonstrate, use the $>$ relation on the partition key:

```
ResultSet results = session.execute("SELECT catalog_id, journal, publisher,
edition,title,author FROM datastax.catalog WHERE catalog_id > 'catalog1'");
```

The following exception is generated:

```
Caused by: com.datastax.driver.core.exceptions.InvalidQueryException: Only EQ
and IN relation are supported on the partition key (unless you use the token()
function) at com.datastax.driver.core.Responses$  
Error.asException(Responses.java:96)
```

RUNNING AN ASYNC QUERY

As discussed, the Session class supports two methods to run the CQL 3 query asynchronously: `executeAsync(Query query)` and `executeAsync(String query)`. *Asynchronously* implies that the method returns immediately and the processing of the application continues, the result being returned later. The Async methods return a `ResultSetFuture` object. A `ResultSetFuture` object is not a `ResultSet` object but a future on a `ResultSet` object. The `ResultSetFuture` class provides the methods listed in Table 3.5 to get the result of the query.

Table 3.5 ResultSetFuture Class Methods to Get Query Result

Method	Description
<code>getUninterruptibly()</code>	Waits for the query to return and returns its result. More convenient than and preferable to the <code>get()</code> method because it waits for the result uninterruptedly and doesn't throw <code>InterruptedException</code> or <code>ExecutionException</code> exceptions.
<code>getUninterruptibly(long timeout, TimeUnit unit)</code>	Waits for the specified time for the query to return the result. More convenient than and preferable to the <code>get(long timeout, TimeUnit unit)</code> method because it waits for the result uninterruptedly and doesn't throw <code>InterruptedException</code> or <code>ExecutionException</code> exceptions.
<code>get()</code>	Waits for the execution to complete and returns its result. Throws an <code>InterruptedException</code> exception if the current thread is interrupted before or during the call, even if the value has been retrieved.
<code>get(long timeout, TimeUnit unit)</code>	Waits for the execution to complete at most for the specified time and returns its result. Throws an <code>InterruptedException</code> exception if the current thread is interrupted before or during the call, even if the value has been retrieved.

The `ResultSetFuture` class provides the methods in Table 3.6 to cancel, or interrupt a future result set object.

Table 3.6 **ResultSetFuture** Class Methods to Cancel or Interrupt a Result Set Future

Method	Description
cancel(boolean mayInterruptIfRunning)	Attempts to cancel the execution of the task. Returns a Boolean to indicate if the cancellation was successful. The attempt fails if the task has already completed or has already been cancelled or could not be cancelled for some other reason. If invoked before a task has started and if the cancellation is successful, the task should not start to run. The mayInterruptIfRunning parameter determines whether the thread running the task should be interrupted in an attempt to stop the task.
isCancelled()	If the cancel() method returns true, the isCancelled method also returns true.
interruptTask()	The default implementation does not interrupt a task, but a subclass may override the method to provide an implementation. If cancel(true) returns true, the interruptTask() method is invoked automatically.
wasInterrupted()	Returns true if the future was cancelled with mayInterruptIfRunning set to true.

The `ResultSetFuture` class provides some other methods, which are discussed in Table 3.7.

Table 3.7 Other Methods in the **ResultSetFuture** Class

Method	Description
set(V value)	Sets the value of the future and returns true if the value could be set successfully.
setException(Throwable throwable)	Sets the future to having failed with the given exception and returns true if the exception could be set successfully. Returns false if the future has already been set or has been cancelled. The Throwable error set becomes the result of the future. Sets the state of the future to <code>AbstractFuture.Sync.COMPLETED</code> and invokes the listeners if the state has been set successfully. The <code>get()</code> methods wrap the exception in <code>ExecutionException</code> and return the error.

(Continued)

Table 3.7 Other Methods in the ResultSetFuture Class (Continued)

Method	Description
addListener(Runnable listener, Executor exec)	Registers a listener.
isDone()	Returns true if the task has completed. Returns true after the cancel() method has returned.

Add an `asyncQuery()` method to the `CQLClient` class and invoke the method from the `main` method. Then invoke the `executeAsync(String)` method to return a `ResultSetFuture` object.

```
ResultSetFuture resultsFuture = session.executeAsync("Select * from
datastax.catalog");
```

Invoke the `getUninterruptibly(long timeout, TimeUnit unit)` method on the `ResultSetFuture` object with the `timeout` set to 1,000,000 ms.

```
ResultSet results = resultsFuture.getUninterruptibly(1000000, TimeUnit.
MILLISCONDS);
```

Iterate over the `ResultSet` object to output the result of the query. If `getUninterruptibly` throws a `TimeoutException`, invoke the `cancel(true)` method to cancel the future.

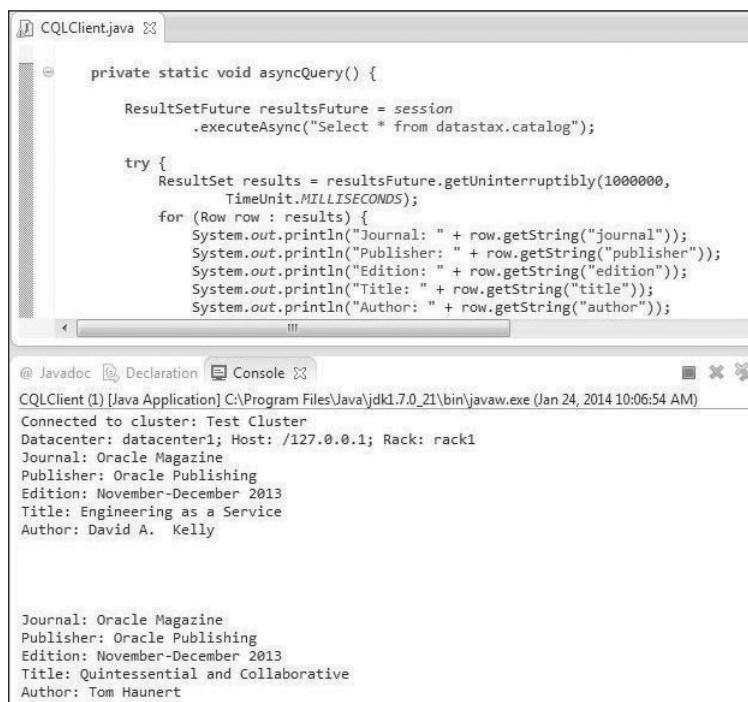
```
try {
    ResultSet results = resultsFuture.getUninterruptibly(1000000,
    TimeUnit.MILLISCONDS);
    for (Row row : results) {
        System.out.println("Journal: " + row.getString
("journal"));
        System.out.println("Publisher: " + row.getString
("publisher"));
        System.out.println("Edition: " + row.getString
("edition"));
        System.out.println("Title: " + row.getString("title"));
        System.out.println("Author: " + row.getString("author"));
        System.out.println("\n");
        System.out.println("\n");
    }
}
```

```

    } catch (TimeoutException e) {
        resultsFuture.cancel(true);
        System.out.println(e);
}
}

```

Run the CQLClient application to output the result of the query. The result of the query is the same as it would be with the synchronous execute() method, as shown in Figure 3.13.



The screenshot shows the Eclipse IDE interface. The top part displays the Java code for `CQLClient.java`:

```

private static void asyncQuery() {
    ResultSetFuture resultsFuture = session
        .executeAsync("Select * from datastax.catalog");

    try {
        ResultSet results = resultsFuture.getUninterruptibly(1000000,
            TimeUnit.MILLISECONDS);
        for (Row row : results) {
            System.out.println("Journal: " + row.getString("journal"));
            System.out.println("Publisher: " + row.getString("publisher"));
            System.out.println("Edition: " + row.getString("edition"));
            System.out.println("Title: " + row.getString("title"));
            System.out.println("Author: " + row.getString("author"));
        }
    }
}

```

The bottom part shows the output in the Console view:

```

@ Javadoc Declaration Console ×
CQLClient (1) [Java Application] C:\Program Files\Java\jdk1.7.0_21\bin\javaw.exe (Jan 24, 2014 10:06:54 AM)
Connected to cluster: Test Cluster
Datacenter: datacenter1; Host: /127.0.0.1; Rack: rack1
Journal: Oracle Magazine
Publisher: Oracle Publishing
Edition: November-December 2013
Title: Engineering as a Service
Author: David A. Kelly

Journal: Oracle Magazine
Publisher: Oracle Publishing
Edition: November-December 2013
Title: Quintessential and Collaborative
Author: Tom Haunert

```

Figure 3.13
Result for `async query`.
Source: Eclipse Foundation.

Why use the `async` version? If the query is expected to take an inordinate amount of time, it may be suitable to use the `async` version while the processing of the application continues and to cancel or interrupt the query if required. Next, you'll see how to cancel a query result set future after a specified duration. Set the timeout to 1 ms. Then run the

CQLClient method with the timeout set to 1 ms. Even a short running query may not return with such a small timeout. As indicated by the TimeoutException in Figure 3.14, the result set future gets timed out before the result can be retrieved.

```

private static void asyncQuery() {
    ResultSetFuture resultsFuture = session
        .executeAsync("Select * from datastax.catalog");

    try {
        ResultSet results = resultsFuture.getUninterruptibly(1,
            TimeUnit.MILLISECONDS);
        for (Row row : results) {
            System.out.println("Journal: " + row.getString("journal"));
            System.out.println("Publisher: " + row.getString("publisher"));
        }
    } catch (TimeoutException e) {
        e.printStackTrace();
    }
}

```

@ Javadoc Declaration Console

CQLClient (1) [Java Application] C:\Program Files\Java\jdk1.7.0_21\bin\javaw.exe (Jan 24, 2014 10:07:51 AM)

log4j:WARN No appenders could be found for logger (com.datastax.driver.core.FrameCompressor).

log4j:WARN Please initialize the log4j system properly.

log4j:WARN See <http://logging.apache.org/log4j/1.2/faq.html#noconfig> for more information.

Connected to cluster: Test Cluster

Datacenter: datacenter1; Host: /127.0.0.1; Rack: rack1

java.util.concurrent.TimeoutException: Timeout waiting for task.

Figure 3.14
TimeoutException.

Source: Eclipse Foundation.

RUNNING A PreparedStatement QUERY

DataStax driver has the provision to create a prepared statement, which is a query with bind variables. The BoundStatement is used to bind values to the bind variables of a PreparedStatement. In this section, you will create a prepared statement and subsequently bind values to the bind variables using a BoundStatement. The BoundStatement class extends the Query class. You will run the query in the BoundStatement using the Session class method execute(Query query). Add a preparedStmtQuery() method to the CQLClient class and invoke the method in the main method. Create a PreparedStatement using the Session class method prepare(String query).

```
PreparedStatement stmt = session.prepare("SELECT catalog_id, journal, publisher,
edition,title,author FROM datastax.catalog WHERE title=?");
```

The prepared statement has a bind variable for the title column. Create a BoundStatement from the PreparedStatement object using the BoundStatement(PreparedStatement statement) constructor.

```
BoundStatement boundStmt = new BoundStatement(stmt);
```

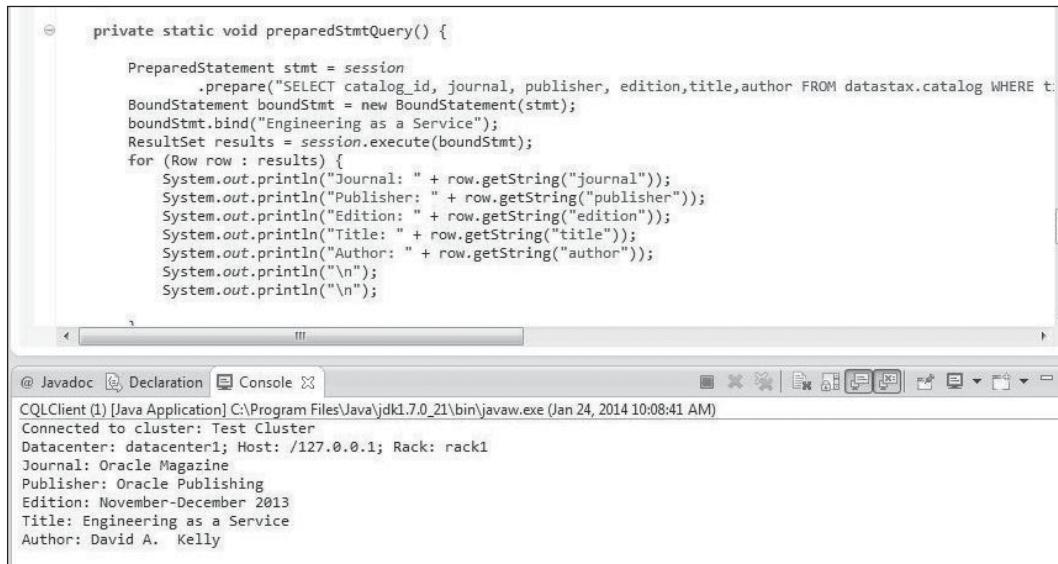
The BoundStatement class provides the bind(Object... values) method to bind values to the bind variables of a PreparedStatement. The values are bound to the bind variables in the order specified. The first value is bound to the first bind variable, the second value to the second bind variable. Set the value of the title variable:

```
boundStmt.bind("Engineering as a Service");
```

Run the query in the BoundStatement, which extends Query, using the execute(Query query) method in the Session class. Iterate over the ResultSet using an enhanced for loop to output the columns.

```
ResultSet results = session.execute(boundStmt);
    for (Row row : results) {
        System.out.println("Journal: " + row.getString
("journal"));
        System.out.println("Publisher: " + row.getString
("publisher"));
        System.out.println("Edition: " + row.getString
("edition"));
        System.out.println("Title: " + row.getString("title"));
        System.out.println("Author: " + row.getString("author"));
        System.out.println("\n");
        System.out.println("\n");
    }
```

The result of running a query with a prepared statement is shown in the Eclipse IDE in Figure 3.15.



The screenshot shows the Eclipse IDE interface. In the top-left pane, there is a code editor with Java code. Below it, the bottom-right pane is a terminal window titled 'Console' showing the output of a Java application. The output displays the results of a SELECT query on the 'catalog' table.

```

private static void preparedStmtQuery() {
    PreparedStatement stmt = session
        .prepare("SELECT catalog_id, journal, publisher, edition, title, author FROM datastax.catalog WHERE t:");
    BoundStatement boundStmt = new BoundStatement(stmt);
    boundStmt.bind("Engineering as a Service");
    ResultSet results = session.execute(boundStmt);
    for (Row row : results) {
        System.out.println("Journal: " + row.getString("journal"));
        System.out.println("Publisher: " + row.getString("publisher"));
        System.out.println("Edition: " + row.getString("edition"));
        System.out.println("Title: " + row.getString("title"));
        System.out.println("Author: " + row.getString("author"));
        System.out.println("\n");
        System.out.println("\n");
    }
}

```

```

@ Javadoc Declaration Console
CQLClient (1) [Java Application] C:\Program Files\Java\jdk1.7.0_21\bin\javaw.exe (Jan 24, 2014 10:08:41 AM)
Connected to cluster: Test Cluster
Datacenter: datacenter1; Host: /127.0.0.1; Rack: rack1
Journal: Oracle Magazine
Publisher: Oracle Publishing
Edition: November-December 2013
Title: Engineering as a Service
Author: David A. Kelly

```

Figure 3.15
Query result with PreparedStatement.

Source: Eclipse Foundation.

RUNNING THE UPDATE STATEMENT

The UPDATE statement is used to update the columns in one or more rows based on a relation specified in the WHERE clause. CQL 3 has added a provision to run the UPDATE conditionally based on the condition in the IF clause. Run the following UPDATE statement to update the edition and author columns in the catalog1 table based on the condition in the IF clause:

```
session.execute("UPDATE datastax.catalog SET edition = '11/12 2013', author = 'Kelley, David A.' WHERE catalog_id = 'catalog1' AND journal='Oracle Magazine' IF edition='November-December 2013'");
```

Next, run a SELECT statement to output the modified columns:

```
ResultSet results = session.execute("SELECT catalog_id, journal, publisher, edition, title, author FROM datastax.catalog WHERE catalog_id='catalog1'");
for (Row row : results) {
    System.out.println("Journal: " + row.getString("journal"));
    System.out.println("Publisher: " + row.getString("publisher"));
}
```

```

        System.out.println("Edition: " + row.getString
("edition"));
        System.out.println("Title: " + row.getString("title"));
        System.out.println("Author: " + row.getString("author"));
        System.out.println("\n");
        System.out.println("\n");
    }
}

```

The catalog1 row column values after the update are shown in Eclipse IDE in Figure 3.16.

The screenshot shows the Eclipse IDE interface. The top part displays the Java code for CQLClient.java, which contains a static method update(). This method performs an UPDATE operation on the datastax.catalog table, setting edition to '11/12 2013' and author to 'Kelley, David A.' for the catalog_id 'catalog1'. It then executes a SELECT statement to retrieve all columns for catalog_id 'catalog1'. Inside the loop, it prints the journal, publisher, edition, title, and author for each row. The bottom part shows the Console tab with the output of the executed query. The log shows initialization messages for Log4j and a connection to a Test Cluster. The output then lists the updated row with columns: Journal, Publisher, Edition, Title, and Author.

```

private static void update() {
    session.execute("UPDATE datastax.catalog SET edition = '11/12 2013', author = 'Kelley, David A.' WHERE catalog_id = 'catalog1' AND
    ResultSet results = session
        .execute("SELECT catalog_id, journal, publisher, edition,title,author FROM datastax.catalog WHERE catalog_id='catalog1'");
    for (Row row : results) {
        System.out.println("Journal: " + row.getString("journal"));
        System.out.println("Publisher: " + row.getString("publisher"));
        System.out.println("Edition: " + row.getString("edition"));
        System.out.println("Title: " + row.getString("title"));
        System.out.println("Author: " + row.getString("author"));
        System.out.println("\n");
        System.out.println("\n");
    }
}

Console
CQLClient (1) [Java Application] C:\Program Files\Java\jdk1.7.0_21\bin\javaw.exe (Feb 4, 2014, 3:46:10 PM)
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Connected to cluster: Test Cluster
Datacenter: datacenter1; Host: /127.0.0.1; Rack: rack1
Journal: Oracle Magazine
Publisher: Oracle Publishing
Edition: 11/12 2013
Title: Engineering as a Service
Author: Kelley, David A.

```

Figure 3.16
Query result with updated column values.

Source: Eclipse Foundation.

Because the primary key is a compound primary key, all the component columns in the primary key must be specified in the WHERE clause. For example, if only the catalog-id column is specified in the WHERE clause, the following exception is generated:

```

Exception in thread "main" com.datastax.driver.core.exceptions.
InvalidQueryException: Missing mandatory PRIMARY KEY part journal

```

RUNNING THE DELETE STATEMENT

The `DELETE` statement is used to delete some selected columns from table row(s) or all the columns from table row(s). With a compound primary key, using the `DELETE` statement is somewhat different than if using a single column primary key. The partition key may be used for the row specification in the `WHERE` clause to delete the entire row. For example, the following deletes the `catalog1` row:

```
private static void delete() {  
    session.execute("DELETE from datastax.catalog WHERE catalog_id='catalog1'");  
}
```

A `SELECT` query after the deletion outputs only the `catalog2` and `catalog3` rows:

```
Catalog Id: catalog2  
Journal: Oracle Magazine  
Publisher: Oracle Publishing  
Edition: November-December 2013  
Title: Quintessential and Collaborative  
Author: Tom Haunert  
  
Catalog Id: catalog3  
Journal: Oracle Magazine  
Publisher: null  
Edition: null  
Title: null  
Author: null
```

Although the partition key may be used alone to identify a row in the `WHERE` clause, the other columns may not be used individually. To demonstrate, specify only the primary key component column `journal` in the `WHERE` clause:

```
session.execute("DELETE from datastax.catalog WHERE journal='Oracle Magazine'");
```

This generates the following exception:

```
Caused by: com.datastax.driver.core.exceptions.InvalidQueryException: Missing  
mandatory PRIMARY KEY part catalog_id
```

The `journal` column may be specified in the `WHERE` clause in addition to the partition key `catalog_id`:

```
session.execute("DELETE from datastax.catalog WHERE catalog_id='catalog1' AND  
journal='Oracle Magazine'");
```

Individual columns to be deleted may be specified in the `DELETE` statement, but a primary key component column cannot be deleted with column specification. To demonstrate, include the journal column to delete using the following query:

```
session.execute("DELETE journal, publisher from datastax.catalog WHERE  
catalog_id='catalog2'");
```

This generates the following exception:

```
Caused by: com.datastax.driver.core.exceptions.InvalidQueryException: Invalid  
identifier journal for deletion (should not be a PRIMARY KEY part)
```

If columns are to be deleted selectively, all the primary key component columns must be specified in the `WHERE` clause to identify the row. To demonstrate, run the following query to delete the `publisher` and `edition` columns from the `catalog` table, but don't specify the `journal` column in the `WHERE` clause:

```
session.execute("DELETE publisher, edition from datastax.catalog WHERE  
catalog_id='catalog2'");
```

This generates the following exception:

```
Caused by: com.datastax.driver.core.exceptions.InvalidQueryException: Missing  
mandatory PRIMARY KEY part journal since edition specified
```

To delete columns selectively, you must specify all the primary key component columns in the `WHERE` clause:

```
session.execute("DELETE publisher, edition from datastax.catalog WHERE  
catalog_id='catalog2' AND journal='Oracle Magazine'");
```

If a `SELECT` query is run after the deletion, `null` is output for the deleted columns `publisher` and `edition` for the `catalog2` row. (See Figure 3.17.)

The screenshot shows the Eclipse IDE interface with the CQLClient.java file open in the editor. The code performs a DELETE operation on the catalog table and then selects all rows from the catalog table. The results of the query are displayed in the Console tab.

```

CQLClient.java  JBoss Central
session.execute("DELETE publisher, edition from datastax.catalog WHERE catalog_id='catalog2' AND journal='Oracle Magazine'");
ResultSet results = session.execute("select * from datastax.catalog");
    
```

Console | Palette

CQLClient [1] [Java Application] C:\Program Files\Java\jdk1.7.0_21\bin\javaw.exe (Feb 4, 2014, 3:58:53 PM)

Connected to cluster: Test cluster

Datacenter: datacenter1; Host: /127.0.0.1; Rack: rack1

Catalog Id: catalog1

Journal: Oracle Magazine

Publisher: Oracle Publishing

Edition: November-December 2013

Title: Engineering as a Service

Author: David A. Kelly

Catalog Id: catalog2

Journal: Oracle Magazine

Publisher: null

Edition: null

Title: Quintessential and Collaborative

Author: Tom Haunert

Catalog Id: catalog3

Journal: Oracle Magazine

Publisher: null

Edition: null

Title: null

Author: null

Figure 3.17

Query result after DELETE.

Source: Eclipse Foundation.

If some of the column values have been deleted, a subsequent INSERT with all the columns specified and with an IF NOT EXISTS condition does not add the new row values, even though some of the column values have been deleted. For example, if, after the preceding deletion of two columns, you attempt to run the INSERT statement, the INSERT statement is not run and the publisher and edition column values stay null.

```
session.execute("INSERT INTO datastax.catalog (catalog_id, journal, publisher, edition, title, author) VALUES ('catalog2', 'Oracle Magazine', 'Oracle Publishing', 'November-December 2013', 'Quintessential and Collaborative', 'Tom Haunert') IF NOT EXISTS");
```

RUNNING THE BATCH STATEMENT

The BATCH statement is used to run a batch or group of INSERT, UPDATE, and DELETE statements. Add a batch() method to the CQLQuery class and invoke the method from the main method. To demonstrate the BATCH statement, create a table catalog4.

```
session.execute("CREATE TABLE IF NOT EXISTS datastax.catalog4 (catalog_id text,  
journal text,publisher text, edition text,title text,author text,PRIMARY KEY  
(journal, catalog_id, publisher))");
```

Run a BATCH statement to add three rows of data to the catalog4 table.

```
session.execute("BEGIN BATCH      INSERT INTO datastax.catalog4 (catalog_id,  
journal, publisher, edition,title,author) VALUES ('catalog1','Oracle Magazine',  
'Oracle Publishing', 'November-December 2013', 'Quintessential and  
Collaborative','Tom Haunert') INSERT INTO datastax.catalog4 (catalog_id, journal,  
publisher, edition,title,author) VALUES ('catalog2','Oracle Magazine', 'Oracle  
Publishing', 'November-December 2013', '', '')  INSERT INTO datastax.catalog4  
(catalog_id, journal, publisher, edition,title,author) VALUES  
(catalog3','Oracle Magazine', 'Oracle Publishing', 'November-December 2013',  
'', '') APPLY BATCH");
```

Run a SELECT query after the BATCH statement. The following rows are output:

Catalog Id: catalog1

Journal: Oracle Magazine

Publisher: Oracle Publishing

Edition: November-December 2013

Title: Quintessential and Collaborative

Author: Tom Haunert

Catalog Id: catalog2

Journal: Oracle Magazine

Publisher: Oracle Publishing

Edition: November-December 2013

Title:

Author:

Catalog Id: catalog3

Journal: Oracle Magazine

Publisher: Oracle Publishing

Edition: November-December 2013

Title:

Author:

The IF NOT EXISTS condition, which may be used with individual INSERT, UPDATE, and DELETE statements, cannot be used with the same statements in a BATCH statement, either applied to individual statements or the batch. To demonstrate, run the following BATCH statement:

```
session.execute("BEGIN BATCH      INSERT INTO datastax.catalog (catalog_id,  
journal, publisher, edition,title,author) VALUES ('catalog2','Oracle Magazine',
```

```
'Oracle Publishing', 'November-December 2013', 'Quintessential and
Collaborative', 'Tom Haunert') IF NOT EXISTS INSERT INTO datastax.catalog
(catalog_id, journal, publisher, edition, title, author) VALUES
('catalog3', 'Oracle Magazine', 'Oracle Publishing', 'November-December 2013',
'', '') IF NOT EXISTS INSERT INTO datastax.catalog (catalog_id, journal, publisher,
edition, title, author) VALUES ('catalog4', 'Oracle Magazine', 'Oracle Publishing',
'November-December 2013', '', '') IF NOT EXISTS APPLY BATCH");
```

This generates the following exception:

```
Caused by: com.datastax.driver.core.exceptions.InvalidQueryException:
Conditional updates are not allowed in batches
```

DROPPING AN INDEX

CQL 3 has added the provision to drop an index conditionally using the IF EXISTS clause. First, run the USE command to select a keyspace. Then drop the titleIndex conditionally as follows:

```
private static void dropIndex() {
    session.execute("USE datastax");
    session.execute("DROP INDEX IF EXISTS titleIndex");
}
```

DROPPING A TABLE

CQL 3 has added the provision to drop a table conditionally. For example, drop the catalog table in the datastax keyspace using the IF EXISTS clause as follows:

```
private static void dropTable() {
    session.execute("DROP TABLE IF EXISTS datastax.catalog");
}
```

DROPPING A KEYSPACE

Dropping a keyspace may also be done conditionally using the IF EXISTS clause. For example, drop the datastax keyspace as follows:

```
private static void dropKeyspace() {
    session.execute("DROP KEYSPACE IF EXISTS datastax");
}
```

The Cassandra cluster connection may be closed using the Cluster.close() method in the closeConnection() method.

THE CQLCLIENT APPLICATION

The CQLClient application used in this chapter appears in Listing 3.2. Sections of the code that demonstrate different aspects or usages of an API have been commented out and may be de-commented for testing.

Listing 3.2 The CQLClient Application

```
package datastax;  
import java.util.concurrent.TimeUnit;  
import java.util.concurrent.TimeoutException;  
  
import com.datastax.driver.core.BoundStatement;  
import com.datastax.driver.core.Cluster;  
import com.datastax.driver.core.Host;  
import com.datastax.driver.core.Metadata;  
import com.datastax.driver.core.PreparedStatement;  
import com.datastax.driver.core.ResultSet;  
import com.datastax.driver.core.ResultSetFuture;  
import com.datastax.driver.core.Row;  
import com.datastax.driver.core.Session;  
import com.google.common.util.concurrent.AbstractFuture;  
  
public class CQLClient {  
  
    private static Cluster cluster;  
    private static Session session;  
  
    public static void main(String[] args) {  
        connection();  
        createKeyspace();  
        createTable();  
        insert();  
        // select();  
        // dropIndex();  
        // createIndex();  
        // selectFilter();  
        // asyncQuery();  
        // preparedStmtQuery();  
        // update();  
        // delete();  
        batch();  
        // dropTable();  
        // dropKeyspace();  
        // closeConnection();  
    }  
}
```

```
private static void connection() {
    cluster = Cluster.builder().addContactPoint("127.0.0.1").build();
    Metadata metadata = cluster.getMetadata();
    System.out.printf("Connected to cluster: %s\n",
        metadata.getClusterName());
    for (Host host : metadata.getAllHosts()) {
        System.out.printf("Datacenter: %s; Host: %s; Rack: %s\n",
            host.getDatacenter(), host.getAddress(), host.getRack());
    }
    session = cluster.connect();
}
private static void createKeyspace() {
    session.execute("CREATE KEYSPACE IF NOT EXISTS datastax WITH replication "
        + "= {'class':'SimpleStrategy', 'replication_factor':1};");
}
private static void createTable() {
    session.execute("CREATE TABLE IF NOT EXISTS datastax.catalog (catalog_id text,
        journal text,publisher text, edition text,title text,author text,PRIMARY KEY
        (catalog_id, journal))");
    session.execute("CREATE TABLE IF NOT EXISTS datastax.catalog2 (catalog_id text,
        journal text,publisher text, edition text,title text,author text,PRIMARY KEY
        (journal, catalog_id))");
    session.execute("CREATE TABLE IF NOT EXISTS datastax.catalog3 (catalog_id text,
        journal text,publisher text, edition text,title text,author text,PRIMARY KEY
        (journal, catalog_id, publisher))");
}
private static void insert() {
    session.execute("INSERT INTO datastax.catalog (catalog_id, journal, publisher,
        edition,title,author) VALUES ('catalog1','Oracle Magazine', 'Oracle Publishing',
        'November-December 2013', 'Engineering as a Service', 'David A. Kelly') IF NOT
        EXISTS");
    session.execute("INSERT INTO datastax.catalog (catalog_id, journal, publisher,
        edition,title,author) VALUES ('catalog2','Oracle Magazine', 'Oracle Publishing',
        'November-December 2013', 'Quintessential and Collaborative', 'Tom Haunert') IF NOT
        EXISTS");
    session.execute("INSERT INTO datastax.catalog (catalog_id, journal, publisher)
        VALUES ('catalog3', 'Oracle Magazine', 'Oracle Publishing') IF NOT EXISTS");
    session.execute("INSERT INTO datastax.catalog2 (catalog_id, journal, publisher,
        edition,title,author) VALUES ('catalog1','Oracle Magazine', 'Oracle Publishing',
        'November-December 2013', 'Engineering as a Service', 'David A. Kelly') IF NOT
        EXISTS");
```

```
session.execute("INSERT INTO datastax.catalog2 (catalog_id, journal, publisher,
edition,title,author) VALUES ('catalog2','Oracle Magazine', 'Oracle Publishing',
'November-December 2013', 'Quintessential and Collaborative','Tom Haunert') IF NOT
EXISTS");
session.execute("INSERT INTO datastax.catalog2 (catalog_id, journal, publisher)
VALUES ('catalog3', 'Oracle Magazine','Oracle Publishing') IF NOT EXISTS");

session.execute("INSERT INTO datastax.catalog3 (catalog_id, journal, publisher,
edition,title,author) VALUES ('catalog1','Oracle Magazine', 'Oracle Publishing',
'November-December 2013', 'Engineering as a Service','David A. Kelly') IF NOT
EXISTS");
session.execute("INSERT INTO datastax.catalog3 (catalog_id, journal, publisher,
edition,title,author) VALUES ('catalog2','Oracle Magazine', 'Oracle Publishing',
'November-December 2013', 'Quintessential and Collaborative','Tom Haunert') IF NOT
EXISTS");
session.execute("INSERT INTO datastax.catalog3 (catalog_id, journal, publisher)
VALUES ('catalog3', 'Oracle Magazine','Oracle Publishing') IF NOT EXISTS");
}

private static void select() {
// ResultSet results =
// session.execute("select * from datastax.catalog");
// ResultSet results =
// session.execute("select * from datastax.catalog2 WHERE journal='Oracle
Magazine' ORDER BY catalog_id DESC");
// ResultSet results =
// session.execute("select * from datastax.catalog3 WHERE journal='Oracle
Magazine' ORDER BY publisher");
// generates error
ResultSet results = session
.execute("select * from datastax.catalog3 WHERE journal='Oracle Magazine' ORDER BY
catalog_id");
for (Row row : results) {
System.out.println("Catalog Id: " + row.getString("catalog_id"));
System.out.println("Journal: " + row.getString("journal"));
System.out.println("Publisher: " + row.getString("publisher"));
System.out.println("Edition: " + row.getString("edition"));
System.out.println("Title: " + row.getString("title"));
System.out.println("Author: " + row.getString("author"));
System.out.println("\n");
System.out.println("\n");
}
}
```

```
private static void createIndex() {
    session.execute("CREATE INDEX titleIndex ON datastax.catalog (title)");
}

private static void selectFilter() {
/*
 * ResultSet results = session .execute(
 * "SELECT catalog_id, journal, publisher, edition,title,author FROM
datastax.catalog2 WHERE journal='Oracle Magazine' AND catalog_id > 'catalog1'"
 * ); for (Row row : results) { System.out.println("Catalog Id: " +
* row.getString("catalog_id")); System.out.println("Journal: " +
* row.getString("journal")); System.out.println("Publisher: " +
* row.getString("publisher")); System.out.println("Edition: " +
* row.getString("edition")); System.out.println("Title: " +
* row.getString("title")); System.out.println("Author: " +
* row.getString("author")); System.out.println("\n");
* System.out.println("\n"); }
*/
/*
 * ResultSet results = session .execute(
 * "SELECT catalog_id, journal, publisher, edition,title,author FROM
datastax.catalog WHERE catalog_id > 'catalog1'"
 * ); for (Row row : results) { System.out.println("Catalog Id: " +
* row.getString("catalog_id")); System.out.println("Journal: " +
* row.getString("journal")); System.out.println("Publisher: " +
* row.getString("publisher")); System.out.println("Edition: " +
* row.getString("edition")); System.out.println("Title: " +
* row.getString("title")); System.out.println("Author: " +
* row.getString("author")); System.out.println("\n");
* System.out.println("\n"); }
*/
/*
 * ResultSet results = session .execute(
 * "SELECT catalog_id, journal, publisher, edition,title,author FROM
datastax.catalog2 WHERE journal='Oracle Magazine' AND catalog_id >= 'catalog1'"
 * ); for (Row row : results) { System.out.println("Catalog Id: " +
* row.getString("catalog_id")); System.out.println("Journal: " +
* row.getString("journal")); System.out.println("Publisher: " +
* row.getString("publisher")); System.out.println("Edition: " +
* row.getString("edition")); System.out.println("Title: " +
* row.getString("title")); System.out.println("Author: " +
```

```
* row.getString("author")); System.out.println("\n");
* System.out.println("\n"); }
*/
/*
* ResultSet results = session .execute(
* "SELECT catalog_id, journal, publisher, edition,title,author FROM
datastax.catalog WHERE title='Engineering as a Service'"
* ); for (Row row : results) { System.out.println("Journal: " +
* row.getString("journal")); System.out.println("Publisher: " +
* row.getString("publisher")); System.out.println("Edition: " +
* row.getString("edition")); System.out.println("Title: " +
* row.getString("title")); System.out.println("Author: " +
* row.getString("author")); System.out.println("\n");
* System.out.println("\n");
*
* }
*/
/*
* ResultSet results = session .execute(
* "SELECT catalog_id, journal, publisher, edition,title,author FROM
datastax.catalog WHERE catalog_id='catalog2'"
* );
*/
/*
* ResultSet results = session .execute(
* "SELECT catalog_id, journal, publisher, edition,title,author FROM
datastax.catalog WHERE journal='Oracle Magazine' ALLOW FILTERING"
* );
*/
/*
* ResultSet results = session .execute(
* "SELECT catalog_id, journal, publisher, edition,title,author FROM
datastax.catalog WHERE  journal='Oracle Magazine' AND catalog_id='catalog2'"
* ); for (Row row : results) { System.out.println("Journal: " +
* row.getString("journal")); System.out.println("Publisher: " +
* row.getString("publisher")); System.out.println("Edition: " +
* row.getString("edition")); System.out.println("Title: " +
* row.getString("title")); System.out.println("Author: " +
* row.getString("author")); System.out.println("\n");
* System.out.println("\n");
*
```

```
* }
*/
/*
 * ResultSet results = session.execute(
 * "SELECT catalog_id, journal, publisher, edition,title,author FROM
datastax.catalog WHERE catalog_id IN ('catalog2', 'catalog3')"
* );
*/
/*
 * ResultSet results = session.execute(
 * "SELECT catalog_id, journal, publisher, edition,title,author FROM
datastax.catalog WHERE title IN ('Quintessential and Collaborative', 'Engineering
as a Service')"
* );
*/
/*
 * for (Row row : results) { System.out.println("Journal: " +
* row.getString("journal")); System.out.println("Publisher: " +
* row.getString("publisher")); System.out.println("Edition: " +
* row.getString("edition")); System.out.println("Title: " +
* row.getString("title")); System.out.println("Author: " +
* row.getString("author")); System.out.println("\n");
* System.out.println("\n");
* }
*/
}
private static void asyncQuery() {
ResultSetFuture resultsFuture = session
.executeAsync("Select * from datastax.catalog");
try {
ResultSet results = resultsFuture.getUninterruptibly(1000000,
TimeUnit.MILLISECONDS);
for (Row row : results) {
System.out.println("Journal: " + row.getString("journal"));
System.out.println("Publisher: " + row.getString("publisher"));
System.out.println("Edition: " + row.getString("edition"));
System.out.println("Title: " + row.getString("title"));
System.out.println("Author: " + row.getString("author"));
System.out.println("\n");
System.out.println("\n");
}
}
```

```
    } catch (TimeoutException e) {
        resultsFuture.cancel(true);
        System.out.println(e);
    }
}

private static void preparedStmtQuery() {
    PreparedStatement stmt = session
        .prepare("SELECT catalog_id, journal, publisher, edition, title, author FROM
datastax.catalog WHERE title=?");
    BoundStatement boundStmt = new BoundStatement(stmt);
    boundStmt.bind("Engineering as a Service");
    ResultSet results = session.execute(boundStmt);
    for (Row row : results) {
        System.out.println("Journal: " + row.getString("journal"));
        System.out.println("Publisher: " + row.getString("publisher"));
        System.out.println("Edition: " + row.getString("edition"));
        System.out.println("Title: " + row.getString("title"));
        System.out.println("Author: " + row.getString("author"));
        System.out.println("\n");
        System.out.println("\n");
    }
}

private static void update() {
    session.execute("UPDATE datastax.catalog  SET edition = '11/12 2013', author =
'Kelley, David A.'  WHERE catalog_id = 'catalog1' AND journal='Oracle Magazine' IF
edition='November-December 2013'");
    ResultSet results = session
        .execute("SELECT catalog_id, journal, publisher, edition, title, author FROM
datastax.catalog WHERE catalog_id='catalog1'");
    for (Row row : results) {
        System.out.println("Journal: " + row.getString("journal"));
        System.out.println("Publisher: " + row.getString("publisher"));
        System.out.println("Edition: " + row.getString("edition"));
        System.out.println("Title: " + row.getString("title"));
        System.out.println("Author: " + row.getString("author"));
        System.out.println("\n");
        System.out.println("\n");
    }
}
```

```
private static void delete() {
    // session.execute("DELETE journal, publisher from datastax.catalog WHERE
    catalog_id='catalog2'");//generates
    // error

    // session.execute("DELETE from datastax.catalog WHERE catalog_id='catalog1'");
    // //equivalent
    // session.execute("DELETE from datastax.catalog WHERE journal='Oracle
    Magazine');//generates
    // error
    /*
     * session.execute(
     * "DELETE from datastax.catalog WHERE catalog_id='catalog1' AND journal='Oracle
    Magazine'"
     * );//equivalent
    ResultSet results =
    * session.execute("select * from datastax.catalog"); for (Row row :
    * results) { System.out.println("Catalog Id: " +
    * row.getString("catalog_id")); System.out.println("Journal: " +
    * row.getString("journal")); System.out.println("Publisher: " +
    * row.getString("publisher")); System.out.println("Edition: " +
    * row.getString("edition")); System.out.println("Title: " +
    * row.getString("title")); System.out.println("Author: " +
    * row.getString("author")); System.out.println("\n");
    * System.out.println("\n");
    *
    * }
    */
    /*
     * Caused by: com.datastax.driver.core.exceptions.InvalidQueryException:
     * Missing mandatory PRIMARY KEY part journal since publisher specified
     */
    // session.execute("DELETE publisher, edition from datastax.catalog WHERE
    catalog_id='catalog2'");//generates
    // error

    // session.execute("DELETE publisher, edition from datastax.catalog WHERE
    catalog_id='catalog1' AND journal='Oracle Magazine'");
    // session.execute("DELETE from datastax.catalog WHERE catalog_id='catalog1'");
    // session.execute("DELETE from datastax.catalog WHERE journal='Oracle
    Magazine');//generates
    // error
    // session.execute("DELETE publisher, edition from datastax.catalog WHERE
    catalog_id='catalog2'");
```

```
// session.execute("DELETE publisher, edition from datastax.catalog WHERE
catalog_id='catalog2' AND journal='Oracle Magazine'");
ResultSet results = session.execute("select * from datastax.catalog");
for (Row row : results) {
System.out.println("Catalog Id: " + row.getString("catalog_id"));
System.out.println("Journal: " + row.getString("journal"));
System.out.println("Publisher: " + row.getString("publisher"));
System.out.println("Edition: " + row.getString("edition"));
System.out.println("Title: " + row.getString("title"));
System.out.println("Author: " + row.getString("author"));
System.out.println("\n");
System.out.println("\n");
}
}

private static void batch() {
// session.execute("BEGIN BATCH      INSERT INTO datastax.catalog (catalog_id,
journal, publisher, edition,title,author) VALUES ('catalog2','Oracle Magazine',
'Oracle Publishing', 'November-December 2013', 'Quintessential and
Collaborative','Tom Haunert') IF NOT EXISTS INSERT INTO datastax.catalog
(catalog_id, journal, publisher, edition,title,author) VALUES
('catalog3','Oracle Magazine', 'Oracle Publishing', 'November-December 2013',
'', '') IF NOT EXISTS INSERT INTO datastax.catalog (catalog_id, journal, publisher,
edition,title,author) VALUES ('catalog4','Oracle Magazine', 'Oracle Publishing',
'November-December 2013', '', '') IF NOT EXISTS APPLY BATCH");

session.execute("CREATE TABLE IF NOT EXISTS datastax.catalog4 (catalog_id text,
journal text,publisher text, edition text,title text,author text,PRIMARY KEY
(journal, catalog_id, publisher))");

session.execute("BEGIN BATCH      INSERT INTO datastax.catalog4 (catalog_id,
journal, publisher, edition,title,author) VALUES ('catalog1','Oracle Magazine',
'Oracle Publishing', 'November-December 2013', 'Quintessential and
Collaborative','Tom Haunert') INSERT INTO datastax.catalog4 (catalog_id, journal,
publisher, edition,title,author) VALUES ('catalog2','Oracle Magazine', 'Oracle
Publishing', 'November-December 2013', '', '')  INSERT INTO datastax.catalog4
(catalog_id, journal, publisher, edition,title,author) VALUES
('catalog3','Oracle Magazine', 'Oracle Publishing', 'November-December 2013',
'', '') APPLY BATCH");

ResultSet results = session.execute("select * from datastax.catalog4");
for (Row row : results) {
System.out.println("Catalog Id: " + row.getString("catalog_id"));
System.out.println("Journal: " + row.getString("journal"));
```

```
System.out.println("Publisher: " + row.getString("publisher"));
System.out.println("Edition: " + row.getString("edition"));
System.out.println("Title: " + row.getString("title"));
System.out.println("Author: " + row.getString("author"));
System.out.println("\n");
System.out.println("\n");
}

private static void dropIndex() {
session.execute("USE datastax");
session.execute("DROP INDEX IF EXISTS titleIndex");
}

private static void dropTable() {
session.execute("DROP TABLE IF EXISTS datastax.catalog");
}

private static void dropKeyspace() {
session.execute("DROP KEYSPACE IF EXISTS datastax");
}

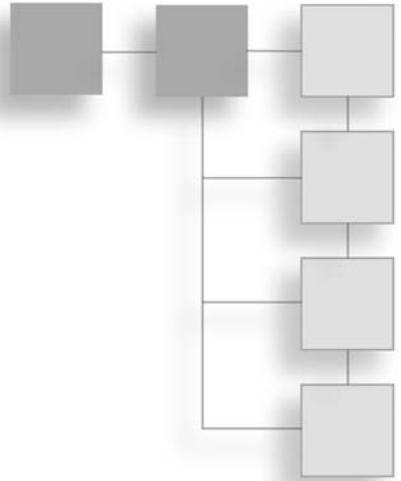
private static void closeConnection() {
cluster.close();
}
```

SUMMARY

In this chapter, you used the DataStax Java driver to access the Cassandra server from a Java application developed in the Eclipse IDE. You used CQL 3 statements to create a keyspace, create a table, insert rows in the table, create an index, select rows and columns from the table, update table rows, delete table rows and columns, run a batch of statements, drop an index, drop a table, and drop a keyspace. In the next chapter, you will learn how to use Apache Cassandra with PHP, an open source, object-oriented, server-side language.

PART II

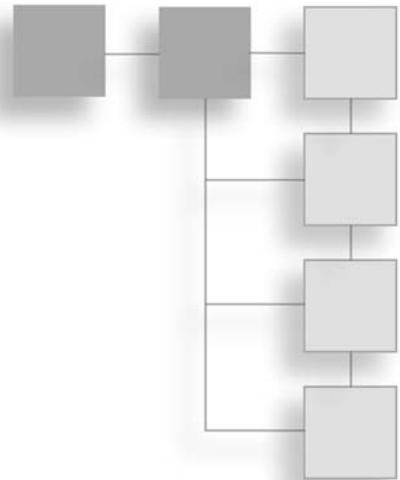
SCRIPTING LANGUAGES



This page intentionally left blank

CHAPTER 4

USING APACHE CASSANDRA WITH PHP



PHP is one of the most commonly used scripting languages, and its usage for developing websites continues to increase. PHP is an open source, object-oriented, server-side language and has the advantages of simplicity with support for all or most operating systems and Web servers. A few PHP clients for Cassandra are available, including phpcassa, which is a PHP client library for Apache Cassandra with support for PHP 5.3+. In this chapter, you will use phpcassa to access Cassandra and perform CRUD operations on Cassandra from PHP scripts.

AN OVERVIEW OF PHPCASSA

Phpcassa provides several namespaces for a PHP client to access Apache Cassandra. A PHP namespace is an abstraction to encapsulate related, reusable code elements such as classes, interfaces, functions, and constants. The top-level namespace is phpcassa. The main classes within the top level namespace are shown in Figure 4.1.

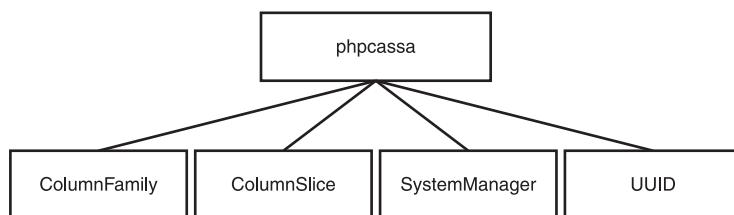


Figure 4.1
Main classes in the top-level namespace phpcassa.

The main classes in the `phpcassa` namespace are discussed in Table 4.1.

Table 4.1 Main Classes in the `phpcassa` Namespace

Class	Description
<code>ColumnFamily</code>	Represents a column family in Cassandra.
<code>ColumnSlice</code>	Represents a slice/range of columns in a row or multiple rows.
<code>SystemManager</code>	Provides information about the state and configuration of a Cassandra cluster. It also provides a means to view or modify information about the schema.
<code>UUID</code>	Represents a UUID, a unique identifier.

The top-level namespace `phpcassa` has several sub-namespaces, which are outlined in Figure 4.2.

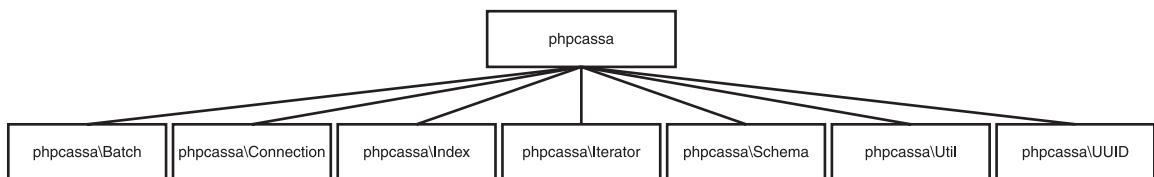


Figure 4.2
Sub-namespaces in the top-level namespace `phpcassa`.

The sub-namespaces in the `phpcassa` namespace are discussed in Table 4.2.

Table 4.2 Sub-Namespace in the `phpcassa` Namespace

Namespace	Description
<code>phpcassa\Batch</code>	Batch operations classes
<code>phpcassa\Connection</code>	Cassandra connection classes
<code>phpcassa\Index</code>	Column index classes
<code>phpcassa\Iterator</code>	Column family iteration classes
<code>phpcassa\Schema</code>	Column family schema classes
<code>phpcassa\Util</code>	Utility classes
<code>phpcassa\UUID</code>	UUID-related classes

Each of the namespaces contains classes specific to the namespace. The main classes in the `phpcassa\Batch` namespace are shown in Figure 4.3.

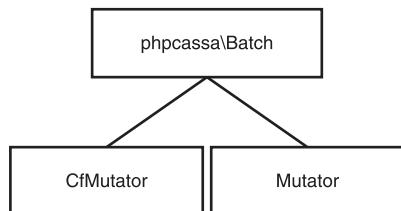


Figure 4.3
Main classes in the `phpcassa\Batch` namespace.

The `phpcassa\Batch` namespace main classes are discussed in Table 4.3.

Table 4.3 Main Classes in the `phpcassa\Batch` Namespace

Class	Description
AbstractMutator	An abstract class with methods common to both <code>Mutator</code> and <code>CfMutator</code> .
Mutator	Groups multiple mutations across one or more rows and column families into a batch operation. Subclass of <code>AbstractMutator</code> .
CfMutator	A sub-class of <code>Mutator</code> for batch operations on a single column family. Subclass of <code>AbstractMutator</code> .

The `phpcassa\Connection` namespace provides the `ConnectionPool` class and the exceptions in Figure 4.4.

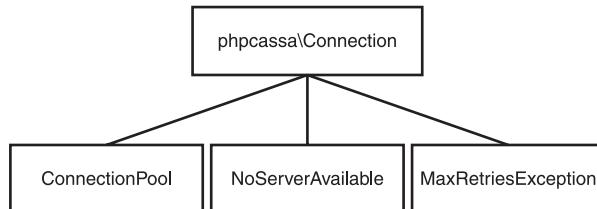


Figure 4.4
Classes in the `phpcassa\Connection` namespace.

The classes and exceptions in the `phpcassa\Connection` namespace are discussed in Table 4.4.

Table 4.4 Main Classes in the `phpcassa\Connection` Namespace

Class	Description
<code>ConnectionPool</code>	A connection pool for servers in a cluster. Specific to a keyspace.
<code>MaxRetriesException</code>	Thrown if a connection pool has retried an operation as many times as configured in the <code>\$max_retries</code> setting.
<code>NoServerAvailable</code>	Thrown if a connection pool is not able to open a connection to any of the servers after retrying each server twice.

The `phpcassa\Index` namespace provides the classes in Figure 4.5.

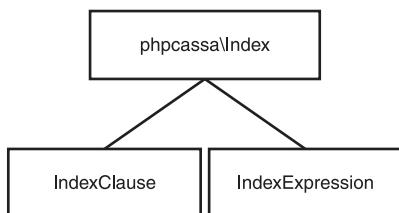


Figure 4.5
Classes in the `phpcassa\Index` namespace.

The classes in the `phpcassa\Index` namespace are discussed in Table 4.5.

Table 4.5 Main Classes in the `phpcassa\Index` Namespace

Class	Description
<code>IndexClause</code>	Constructs an index clause to be used to get indexed column slices
<code>IndexExpression</code>	Constructs an index expression to be used in an index clause

The classes in the `phpcassa\Schema` namespace are outlined in Figure 4.6.

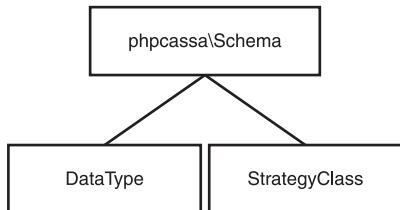


Figure 4.6
Classes in the `phpcassa\Schema` namespace.

The classes in the `phpcassa\Schema` namespace are discussed in Table 4.6.

Table 4.6 Main Classes in the <code>phpcassa\Schema</code> Namespace	
Class	Description
<code>DataType</code>	Provides the different data types
<code>StrategyClass</code>	Represents replication strategies for keyspaces

The `DataType` class provides various data types as string constants. Some of the main data types are listed in Figure 4.7.

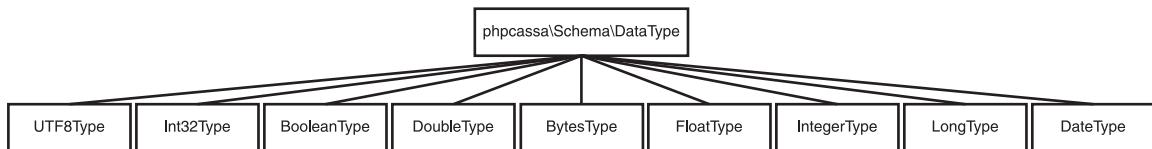


Figure 4.7
Data types in the `DataType` class.

Some of the main data types in the `DataType` class are discussed in Table 4.7.

Table 4.7 Data Types in the <code>DataType</code> Class	
Class	Description
<code>UTF8Type</code>	UTF8 data type
<code>Int32Type</code>	Int32 data type
<i>(Continued)</i>	

Table 4.7 Data Types in the `DataType` Class (Continued)

Class	Description
BooleanType	Boolean data type
DoubleType	Double data type
BytesType	Bytes data type
FloatType	Float data type
IntegerType	Integer data type
LongType	Long data type
DateType	Date data type.

SETTING THE ENVIRONMENT

In addition to installing Apache Cassandra server, you must install the following software:

- PHP
- PHP client library for Cassandra

Then follow these steps:

1. Add the bin folder, C:\Cassandra\apache-cassandra-2.0.4\bin, to the PATH environment variable.
2. Start the Cassandra server with the following command:

```
cassandra -f
```

Installing PHP

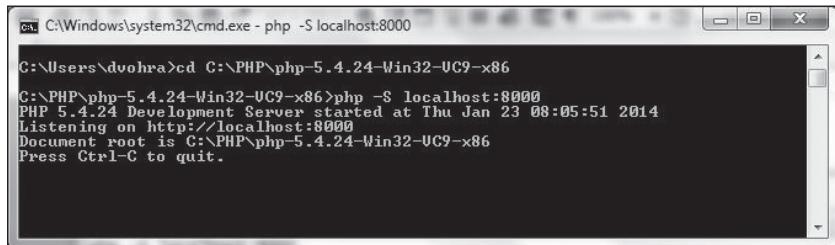
PHP 5.4 and later versions include a Web server packaged in the PHP installation and do not require the Web server to be installed separately. To install PHP, follow these steps:

1. Download the latest version of the PHP TAR file from <http://php.net/downloads.php>.
2. Extract the TAR file to a directory (C:\PHP is used in this chapter) with the following command:
`tar -xzf php-5.4.23.tar.gz`
3. Rename the `php.ini-development` or `php.ini-production` file in the root directory of the PHP installation (C:\PHP\php-5.4.24-Win32-VC9-x86) to `php.ini`.

4. Connect to the packaged Web server with the following command from the C:\PHP\php-5.4.24-Win32-VC9-x86 directory:

```
php -S localhost:8000
```

The output from the command indicates that the development server has been started and is listening on http://localhost:8000. (See Figure 4.8.)



A screenshot of a Windows Command Prompt window titled 'cmd C:\Windows\system32\cmd.exe - php -S localhost:8000'. The window shows the command 'cd C:\PHP\php-5.4.24-Win32-VC9-x86' being run, followed by 'php -S localhost:8000'. The output indicates that the PHP 5.4.24 Development Server has started at Thu Jan 23 08:05:51 2014, listening on http://localhost:8000, and the document root is C:\PHP\php-5.4.24-Win32-VC9-x86. It also says to Press Ctrl-C to quit.

Figure 4.8

Starting the development server.

Source: Microsoft Corporation.

The document root is the directory to which the TAR file is extracted, C:\PHP\php-5.4.24-Win32-VC9-x86. Any PHP script phpscript.php copied to the document root directory may be run on the integrated Web server with the URL http://localhost:8000/phpscript.php. You can copy PHP scripts to a subdirectory of the document root directory and run them by including the directory path, starting from the document root, in the URL.

Installing Phpcassa

To install phpcassa, follow these steps:

1. Download the phpcassa library from <https://github.com/thobbs/phpcassa>. The Download ZIP downloads the phpcassa-master.zip file.
2. Extract the phpcassa-master.zip file to the C:\PHP\php-5.4.24-Win32-VC9-x86 directory.
3. Create a subdirectory, scripts, in the phpcassa-master directory.

You will add PHP scripts to the phpcassa-master\scripts directory and run the scripts in the integrated Web server. The URL to run a script phpscript.php in the phpcassa-master\scripts directory is http://localhost:8000/phpcassa-master/scripts/phpscript.php.

CREATING A KEYSpace

A *keyspace* is the top-level namespace for storing data in a Cassandra database. First, you need to create a keyspace in Cassandra. Create a PHP script, `createKeyspace.php`, in the `phpcassa-master\scripts` directory. Include the `phpcassa` library in the PHP script with the following statement:

```
require_once(__DIR__ . '/../lib/autoload.php');
```

Import the `ConnectionPool`, `SystemManager`, and `StrategyClass` classes using `use` statements. Create a `SystemManager` object using the following class constructor:

```
_construct( string $server = 'localhost:9160', array $credentials = NULL, integer  
$send_timeout = 15000, integer $recv_timeout = 15000 )
```

The constructor parameters are discussed in Table 4.8.

Table 4.8 SystemManager Constructor Parameters

Parameter	Type	Description	Default Value
\$server	string	'localhost:9160'	The host and port to connect to in the format <code>host:port</code> . The default value for host is <code>localhost</code> and the default value for port is <code>9160</code> .
\$credentials	array	NULL	Username and password credentials for authorization and authentication with Cassandra in the format <code>array("username" => username, "password" => password)</code> .
\$send_timeout	integer	15000	Socket send timeout in milliseconds.
\$recv_timeout	integer	15000	Socket receive timeout in milliseconds.

Create an instance of `SystemManager` as follows.

```
$sys = new SystemManager('127.0.0.1');
```

Create a keyspace using the `create_keyspace($keyspace, $attrs)` method from the `SystemManager` class. The `$keyspace` parameter is the keyspace name and the `$attrs`

parameter is for the attributes of the keyspace. The attributes discussed in Table 4.9 are supported.

Table 4.9 Keyspace Attributes

Attributes	Description
strategy_class	The strategy class to use for replication, the default being SimpleStrategy.
strategy_options	The replication strategy options.
replication_factor	The number of nodes to replicate to. The default replication factor is 1. The replication_factor is specified as a strategy option.

Create a keyspace using Simple_Strategy and a replication_factor of 1:

```
$sys->create_keyspace('php_catalog', array(
"strategy_class" => StrategyClass::SIMPLE_STRATEGY,
"strategy_options" => array('replication_factor' => '1')));
```

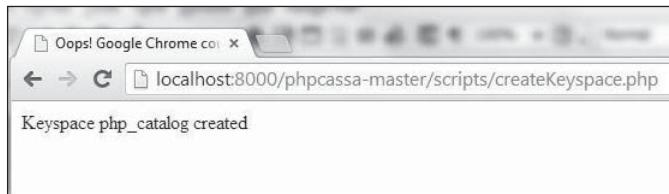
The PHP script createKeyspace.php appears in Listing 4.1.

Listing 4.1 The createKeyspace.php Script

```
<?php

require_once(__DIR__.'../../../lib/autoload.php');
use phpcassa\Connection\ConnectionPool;
use phpcassa\SystemManager;
use phpcassa\Schema\StrategyClass;
$sys = new SystemManager('127.0.0.1');
$sys->create_keyspace('php_catalog', array(
"strategy_class" => StrategyClass::SIMPLE_STRATEGY,
"strategy_options" => array('replication_factor' => '1')));
echo 'Keyspace php_catalog created';
?>
```

With the Cassandra server running and the PHP integrated server started, invoke the PHP script with the URL <http://localhost:8000/phpcassa-master/scripts/createKeyspace.php>. The php_catalog keyspace is created in Cassandra, as shown in Figure 4.9.

**Figure 4.9**

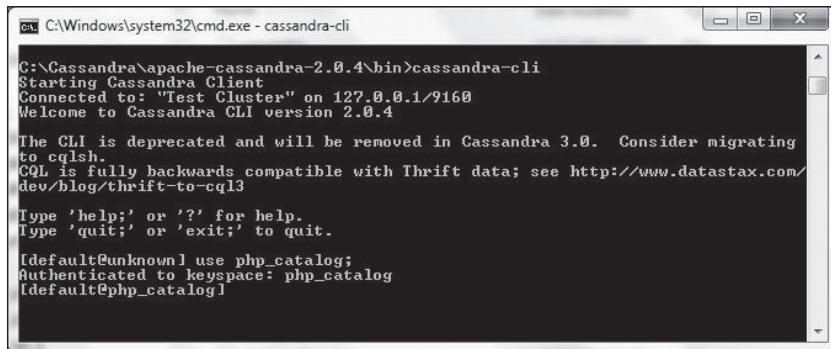
Creating a keyspace.

Source: Google Inc.

Log in to the Cassandra client with the `cassandra-cli` batch application. Then run the following command to use the newly created keyspace `php_catalog`:

```
use php_catalog;
```

As indicated by the output in the Cassandra client, the `php_catalog` keyspace is authenticated. (See Figure 4.10.)

**Figure 4.10**

Authenticating a keyspace.

Source: Microsoft Corporation.

Next, you will create a column family in Cassandra.

CREATING A COLUMN FAMILY AND CONNECTION POOL

A column family or a table is the data structure to store data in Cassandra. To create a column family, first create a PHP script, `createCF.php`, in the `phpcassa-master\scripts` directory. Include the `phpcassa` library in the PHP script and import the

`ConnectionPool`, `SystemManager`, and `StrategyClass` classes as in the preceding section. Create a `SystemManager` object using the class constructor, also as in the preceding section. The `SystemManager` class provides the following method to create a column family:

```
create_column_family( string $keyspace, string $column_family, array $attrs = null )
```

The method parameters are discussed in Table 4.10.

Table 4.10 `create_column_family` Method Parameters

Parameter	Description
<code>\$keyspace</code>	The keyspace name.
<code>\$column_family</code>	The column family name.
<code>\$attrs</code>	The column family attributes. Some of the supported attributes are <code>column_type</code> , <code>default_validation_class</code> , <code>comparator_type</code> , and <code>key_validation_class</code> . For regular column families, <code>column_type</code> defaults to <code>Standard</code> . The default <code>comparator_type</code> is <code>org.apache.cassandra.db.marshal.AsciiType</code> .

Create a column family called `catalog` in the `php_catalog` keyspace using `UTF8Type` for `comparator_type`, `key_validation_class`, and `default_validation_class`.

```
$sys->create_column_family('php_catalog', 'catalog', array(
    "column_type" => "Standard",
    "comparator_type" => "UTF8Type",
    "key_validation_class" => "UTF8Type",
    "default_validation_class" => "UTF8Type"
));
```

Next, create a `ConnectionPool` instance using the following class constructor:

```
__construct( string $keyspace, mixed $servers = NULL, integer $pool_size = NULL,
integer $max_retries = phpcassa\Connection\ConnectionPool::DEFAULT_MAX_RETRIES,
integer $send_timeout = 5000, integer $recv_timeout = 5000, integer $recycle =
phpcassa\Connection\ConnectionPool::DEFAULT_RECYCLE, mixed $credentials = NULL,
boolean $framed_transport = true )
```

The constructor supports the parameters discussed in Table 4.11.

Table 4.11 ConnectionPool Constructor Parameters

Parameter	Type	Description	Default Value
\$keyspace	string	The keyspace used by all connections.	No default value; the only required parameter
\$servers	mixed	Array of strings for servers with each string in the format 'host:port'.	NULL, which implies 'localhost:9160'
\$pool_size	integer	The number of open connections to keep in the pool.	NULL, which implies $\max(5, \text{count}(\$servers) * 2)$
\$max_retries	integer	The number of times an operation is retried before throwing MaxRetriesException. A setting of 0 disables retries. A setting of -1 is for unlimited retries.	5
\$send_timeout	integer	Socket send timeout in milliseconds.	5000
\$recv_timeout	integer	Socket receive timeout in milliseconds.	5000
\$recycle	integer	A connection is closed and reopened after the specified times, the default being 10,000.	10000
\$credentials	mixed	The username and password credentials specified as array ("username" => username, "password" => password)	NULL
\$framed_transport	boolean	If framed transport is to be used. Framed transport is the default implementation provided by Thrift. The alternative is buffered transport. With buffered transport, an internal buffer is created to store data.	true

Create a `ConnectionPool` instance using the `php_catalog` keyspace and the '127.0.0.1' host.

```
$pool = new ConnectionPool('php_catalog', array('127.0.0.1'));
```

You created a column family earlier. Create a `ColumnFamily` instance using the following class constructor:

```
_construct($pool, $column_family, $autopack_names=true, $autopack_values=true,
$read_consistency_level=ConsistencyLevel::ONE, $write_consistency_level=
ConsistencyLevel::ONE, $buffer_size=self::DEFAULT_BUFFER_SIZE)
```

The `ColumnFamily` class constructors are discussed in Table 4.12.

Table 4.12 ColumnFamily Constructor Parameters

Parameter	Type	Description	Default Value
<code>\$pool</code>	<code>phpcassa\Connection\ConnectionPool</code>	The connection pool to use as a <code>ConnectionPool</code> instance	
<code>\$column_family</code>	<code>string</code>	The column family to use	
<code>\$autopack_names</code>	<code>boolean</code>	If column names are to be converted automatically to and from their binary representation in Cassandra based on their comparator type	<code>true</code>
<code>\$autopack_values</code>	<code>boolean</code>	If column values are to be converted automatically to and from their binary representation in Cassandra based on their validator type	<code>true</code>
<code>\$read_consistency_level</code>	<code>ConsistencyLevel</code>	The default consistency level on read operations on the column family	<code>ConsistencyLevel::ONE</code>

(Continued)

Table 4.12 ColumnFamily Constructor Parameters (*Continued*)

Parameter	Type	Description	Default Value
\$write_consistency_level	ConsistencyLevel	The default consistency level on write operations on the column family	ConsistencyLevel::ONE
\$buffer_size	int	The number of rows to buffer when fetching many rows to prevent Cassandra from overallocating memory and failing	100

Create a `ColumnFamily` instance called `catalog` using the `ConnectionPool` instance created earlier and the `catalog` column family. Default values are used for the other attributes.

```
$catalog = new ColumnFamily($pool, 'catalog');
```

The PHP script `createCF.php` appears in Listing 4.2.

Listing 4.2 The createCF.php Script

```
<?php
require_once(__DIR__.'/../lib/autoload.php');
use phpcassa\Connection\ConnectionPool;
use phpcassa\ColumnFamily;
use phpcassa\SystemManager;
$sys = new SystemManager('127.0.0.1');
$sys->create_column_family('php_catalog', 'catalog', array(
"column_type" => "Standard",
"comparator_type" => "UTF8Type",
"key_validation_class" => "UTF8Type",
"default_validation_class" => "UTF8Type"
));
$pool = new ConnectionPool('php_catalog', array('127.0.0.1'));
$catalog = new ColumnFamily($pool, 'catalog');
echo 'column family catalog created';
?>
```

With the Cassandra server running and the PHP integrated server started, invoke the PHP script with the URL <http://localhost:8000/phpcassa-master/scripts/createCF.php>. The catalog column family is created in Cassandra. (See Figure 4.11.)

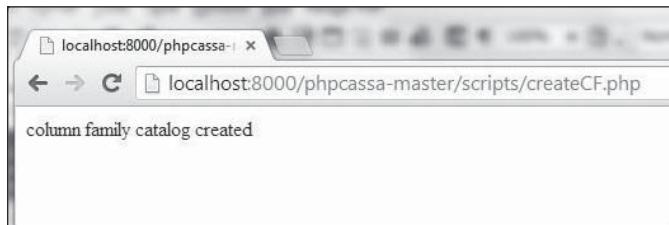


Figure 4.11
Creating a column family.

Source: Google Inc.

Next, you will add data to the column family created in this section.

ADDING DATA

Cassandra stores data in rows and columns in a column family. To see how this works, create a PHP script, add.php, in the phpcassa-master\scripts directory for adding data to Cassandra. Include the phpcassa library in the PHP script. Import the ConnectionPool, SystemManager, and StrategyClass classes and create a SystemManager object using the class constructor. The ColumnFamily class provides the insert() method to add data to columns in a row. The required parameters of the insert() method are discussed in Table 4.13.

Table 4.13 insert() Method Parameters

Parameter	Type	Description
\$key	string	The row primary key in which to add column data
\$columns	mixed[]	An array of columns to add, represented as array (column_name => column_value)

Create a `ColumnFamily` instance as discussed in the previous section.

```
$pool = new ConnectionPool('php_catalog', array('127.0.0.1'));
$catalog = new ColumnFamily($pool, 'catalog');
```

Add two rows of data to rows identified by "catalog1" and "catalog2". Then create an array of column name/value pairs for the journal, publisher, edition, title, and author columns.

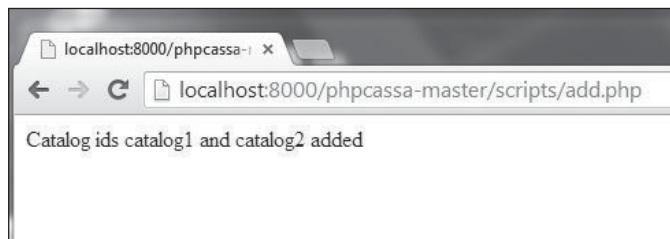
```
$catalog->insert('catalog1', array("journal" => "Oracle Magazine", "publisher" =>
"Oracle Publishing", "edition" => "November-December 2013", "title" =>
"Quintessential and Collaborative", "author" => "Tom Haunert"));
$catalog->insert('catalog2', array("journal" => "Oracle Magazine", "publisher" =>
"Oracle Publishing", "edition" => "November-December 2013", "title" => "Engineering
as a Service", "author" => "David A. Kelly"));
```

The PHP script `add.php` appears in Listing 4.3.

Listing 4.3 The add.php Script

```
<?php
require_once(__DIR__.'/../lib/autoload.php');
use phpcassa\Connection\ConnectionPool;
use phpcassa\ColumnFamily;
use phpcassa\SystemManager;
$sys = new SystemManager('127.0.0.1');
$pool = new ConnectionPool('php_catalog', array('127.0.0.1'));
$catalog = new ColumnFamily($pool, 'catalog');
$catalog->insert('catalog1', array("journal" => "Oracle Magazine", "publisher" =>
"Oracle Publishing", "edition" => "November-December 2013", "title" =>
"Quintessential and Collaborative", "author" => "Tom Haunert"));
$catalog->insert('catalog2', array("journal" => "Oracle Magazine", "publisher" =>
"Oracle Publishing", "edition" => "November-December 2013", "title" => "Engineering
as a Service", "author" => "David A. Kelly"));
echo 'Catalog ids catalog1 and catalog2 added';
// $catalog->remove("catalog1");
// $catalog->remove("catalog2");
?>
```

With the Cassandra server running and the PHP integrated server started, invoke the PHP script with the URL `http://localhost:8000/phpcassa-master/scripts/add.php`. Two rows of data are added to the column family `catalog`. (See Figure 4.12.)

**Figure 4.12**

Adding data.

Source: Google Inc.

In this section, you added only a row of data. In the next section, you will add multiple rows in the same statement. A row of data may be removed with the `remove($key)` method. If the same key identifiers are to be used for adding multiple rows, remove the rows added in this section by running the `add.php` script with the `remove()` method invocations commented out.

ADDING DATA IN A BATCH

In this section, you will add data to multiple rows in a batch. Create a PHP script, `add_batch.php`, in the `phpcassa-master\scripts` directory for adding data to Cassandra. Include the `phpcassa` library in the PHP script. Import the `ConnectionPool`, `SystemManager`, and `StrategyClass` classes and create a `SystemManager` object using the class constructor. The `ColumnFamily` class provides the `insert()` method to add data to columns in a row. The required parameters of the `batch_insert` method are discussed in Table 4.14.

Table 4.14 batch_insert Method Parameters

Parameter	Description
<code>\$rows</code>	An array of rows, each of which maps to an array of columns. Of the format <code>array(key => array(column_name => column_value))</code> .

Add the `catalog1` and `catalog2` rows in a batch using the `batch_insert` method as follows:

```
$catalog->batch_insert(array("catalog1" => array("journal" => "Oracle Magazine",
"publisher" => "Oracle Publishing", "edition" => "November-December 2013", "title"
=> "Quintessential and Collaborative", "author" => "Tom Haunert"),
```

```
"catalog2" => array("journal" => "Oracle Magazine", "publisher" => "Oracle Publishing", "edition" => "November-December 2013", "title" => "Engineering as a Service", "author" => "David A. Kelly"));
```

The `ColumnFamily` class provides the `multiget()` method to fetch multiple rows of data. The `multiget()` method is discussed later in this chapter, in the section, “Getting Columns from Multiple Rows.” In this section, use the `multiget()` method to fetch the rows added with the `batch_insert` method.

```
$catalogs = $catalog->multiget(array('catalog1', 'catalog2'));
```

The `multiget()` method returns mixed array(`key => array(column_name => column_value)`). Iterate over the array to output individual columns.

```
foreach($catalogs as $catalog_id => $columns) {
    echo "Journal: ".$columns["journal"]."\n";
    echo "Publisher: ".$columns["publisher"]."\n";
    echo "Edition: ".$columns["edition"]."\n";
    echo "Title: ".$columns["title"]."\n";
    echo "Author: ".$columns["author"]."\n";
    echo "<br>\n";
}
```

The `add_batch.php` PHP script appears in Listing 4.4.

Listing 4.4 The add_batch.php Script

```
<?php
require_once(__DIR__.'/../lib/autoload.php');
use phpcassa\Connection\ConnectionPool;
use phpcassa\ColumnFamily;
use phpcassa\SystemManager;
use phpcassa\Schema\StrategyClass;
$sys = new SystemManager('127.0.0.1');

$sys->create_keyspace('batch_catalog', array(
    "strategy_class" => StrategyClass::SIMPLE_STRATEGY,
    "strategy_options" => array('replication_factor' => '1')));

$sys->create_column_family('batch_catalog', 'catalog', array(
    "column_type" => "Standard",
    "comparator_type" => "UTF8Type",
    "key_validation_class" => "UTF8Type",
    "default_validation_class" => "UTF8Type"
));
```

```

$pool = new ConnectionPool('batch_catalog', array('127.0.0.1'));
$catalog = new ColumnFamily($pool, 'catalog');
$catalog->batch_insert(array("catalog1" => array("journal" => "Oracle Magazine",
"publisher" => "Oracle Publishing", "edition" => "November-December 2013", "title"
=> "Quintessential and Collaborative", "author" => "Tom Haunert"),
"catalog2" => array("journal" => "Oracle Magazine", "publisher" => "Oracle
Publishing", "edition" => "November-December 2013", "title" => "Engineering as a
Service", "author" => "David A. Kelly")));
echo 'Catalog ids catalog1 and catalog2 added as a batch ';
echo "<br>\n";
$catalogs = $catalog->multiget(array('catalog1', 'catalog2'));
foreach($catalogs as $catalog_id => $columns) {
echo "Journal: ".$columns["journal"]."\n";
echo "Publisher: ".$columns["publisher"]."\n";
echo "Edition: ".$columns["edition"]."\n";
echo "Title: ".$columns["title"]."\n";
echo "Author: ".$columns["author"]."\n";
echo "<br>\n";
}
$sys->drop_keyspace("batch_catalog");
$pool->close();
$sys->close();
?>

```

With the Cassandra server running and the PHP integrated server started, invoke the PHP script with the URL http://localhost:8000/phpcassa-master/scripts/add_batch.php. Two rows of data are added to the column family catalog, and are fetched and output. (See Figure 4.13.)

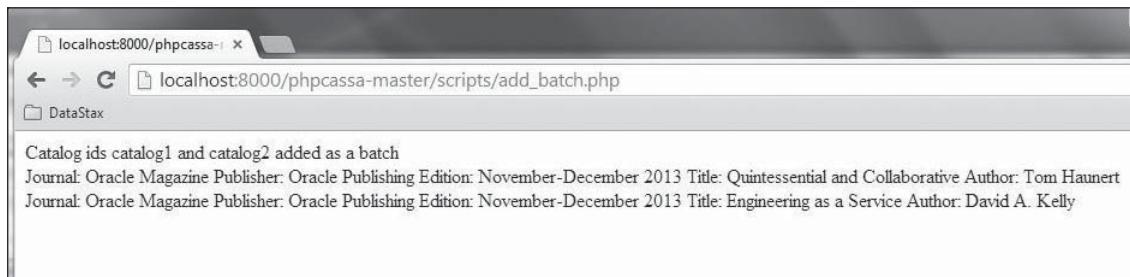


Figure 4.13
Adding data in a batch.
Source: Google Inc.

Having added data, you will next retrieve data from Cassandra.

RETRIEVING DATA

In this section, you will retrieve data from Cassandra using phpcassa. Create a PHP script, get.php, in the phpcassa-master\scripts directory for getting data from Cassandra. As in other sections, include the phpcassa library in the PHP script. Import the ConnectionPool, SystemManager, and StrategyClass classes, and create a SystemManager object using the class constructor. The ColumnFamily class provides the following method to get():

```
get($key,$column_slice=null,$column_names=null,$consistency_level=null)
```

The parameters of the get() method are discussed in Table 4.15.

Table 4.15 Parameters in the get() Method

Parameter	Type	Description	Default Value
\$key	string	The row primary key to fetch.	
\$column_slice	\phpcassa \\ColumnSlice	A slice of columns to fetch.	
\$column_names	mixed[]	List of columns to fetch. By default all columns are fetched if none are specified.	true
\$consistency_level	Consistency Level	The number of nodes that must respond before the method returns.	

The get() method returns mixed array(column_name => column_value). Get the array of columns for row key 'catalog1'.

```
$catalog1=$catalog->get('catalog1');
```

The following method returns the number of columns in a row:

```
get_count($key,$column_slice=null,$column_names=null,$consistency_level=null)
```

The parameters of the get_count() method are discussed in Table 4.16.

Table 4.16 Parameters in the get_count() Method

Parameter	Type	Description	Default Value
\$key	string	The row key to fetch	
\$column_slice	\phpcassa \ColumnSlice	The slice of columns to fetch	null
\$column_names	mixed[]	List of column names to fetch	null
\$consistency_level	ConsistencyLevel	The number of nodes that must respond before the method returns	null

The `get_count()` method returns an `int` value. Get the number of columns in the '`catalog1`' row.

```
echo $catalog->get_count('catalog1');
```

Get the column values by dereferencing the array using column names. For example, the column value for the `journal` column is output as follows:

```
echo $journal = $catalog1["journal"];
```

Similarly, get the array of columns in the '`catalog2`' row.

```
$catalog2= $catalog->get('catalog2');
```

The `get.php` script appears in Listing 4.5.

Listing 4.5 The get.php Script

```
<?php
require_once(__DIR__.'/../lib/autoload.php');
use phpcassa\Connection\ConnectionPool;
use phpcassa\ColumnFamily;
use phpcassa\SystemManager;
$sys = new SystemManager('127.0.0.1');
$pool = new ConnectionPool('php_catalog', array('127.0.0.1'));
$catalog = new ColumnFamily($pool, 'catalog');
$catalog1= $catalog->get('catalog1');
echo "catalog1";
echo "<br>\n";
```

```
echo "Number of Columns in catalog1: ";
echo $catalog->get_count('catalog1');
echo "<br>\n";
echo "Journal: ";
echo $journal = $catalog1["journal"];
echo "<br>\n";
echo "Publisher: ";
echo $publisher = $catalog1["publisher"];
echo "<br>\n";
echo "Edition: ";
echo $edition = $catalog1["edition"];
echo "<br>\n";
echo "Title: ";
echo $title = $catalog1["title"];
echo "<br>\n";
echo "Author: ";
echo $author = $catalog1["author"];
echo "<br>\n";
echo "<br>\n";
$catalog2= $catalog->get('catalog2');
echo "catalog2";
echo "<br>\n";
echo "Number of Columns in catalog2: ";
echo $catalog->get_count('catalog2');
echo "<br>\n";
echo "Journal: ";
echo $journal = $catalog2["journal"];
echo "<br>\n";
echo "Publisher: ";
echo $publisher = $catalog2["publisher"];
echo "<br>\n";
echo "Edition: ";
echo $edition = $catalog2["edition"];
echo "<br>\n";
echo "Title: ";
echo $title = $catalog2["title"];
echo "<br>\n";
echo "Author: ";
echo $author = $catalog2["author"];
echo "<br>\n";
?>
```

With the Cassandra server running and the PHP integrated server started, invoke the PHP script with the URL `http://localhost:8000/phpcassa-master/scripts/get.php`. The two rows of data are fetched individually and the column values are output. (See Figure 4.14.)

```
catalog1
Number of Columns in catalog1: 5
Journal: Oracle Magazine
Publisher: Oracle Publishing
Edition: November-December 2013
Title: Quintessential and Collaborative
Author: Tom Haunert

catalog2
Number of Columns in catalog2: 5
Journal: Oracle Magazine
Publisher: Oracle Publishing
Edition: November-December 2013
Title: Engineering as a Service
Author: David A. Kelly
```

Figure 4.14

Getting data.

Source: Google Inc.

In this section, you fetched all the columns, but all columns don't have to be fetched. In the next section, you will fetch only selected columns.

GETTING SELECTED COLUMNS

You will use the same method to fetch selected columns:

```
get_count($key,$column_slice=null,$column_names=null,$consistency_level=null)
```

Create a PHP script, `get_columns.php`, in the `phpcassa-master\scripts` directory for getting selected columns from Cassandra. Then invoke the `get()` method with the row key as '`'catalog1'`'. For the `$column_names` argument, specify three columns: `journal`, `title`, and `author`.

```
$columns = $catalog->get('catalog1', $column_slice=null, $column_names=array
("journal", "title", "author"));
```

Output the column values using the array dereferencing using the column name.

```
echo "Journal: ".$columns["journal"].",\n";
echo "Title: ".$columns["title"].",\n";
echo "Author: ".$columns["author"].",\n";
```

The get_columns.php script appears in Listing 4.6.

Listing 4.6 The get_columns.php Script

```
<?php
require_once(__DIR__.'/../lib/autoload.php');
use phpcassa\Connection\ConnectionPool;
use phpcassa\ColumnFamily;
use phpcassa\SystemManager;
$sys = new SystemManager('127.0.0.1');
$pool = new ConnectionPool('php_catalog', array('127.0.0.1'));
$catalog = new ColumnFamily($pool, 'catalog');
$columns = $catalog->get('catalog1', $column_slice=null, $column_names=array
("journal", "title", "author"));
echo "Journal: ".$columns["journal"].",\n";
echo "Title: ".$columns["title"].",\n";
echo "Author: ".$columns["author"].",\n";
?>
```

Invoke the PHP script with the URL http://localhost:8000/phpcassa-master/scripts/get_columns.php. The three columns of data are fetched and column values are output. (See Figure 4.15.)

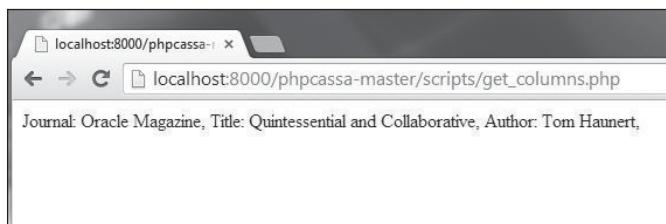


Figure 4.15

Getting data for selected columns.

Source: Google Inc.

In the preceding two sections, you fetched a column from a single row. In the next section, you will fetch columns from multiple rows.

GETTING COLUMNS FROM MULTIPLE ROWS

The `ColumnFamily` class provides the following method to fetch columns from multiple rows:

```
multiget($keys,$column_slice=null,$column_names=null,$consistency_level=null,
$buffer_size=16)
```

The parameters of the method are discussed in Table 4.17.

Table 4.17 Parameters in the `multiget()` Method

Parameter	Type	Description	Default Value
<code>\$keys</code>	<code>string[]</code>	A list of rows specified as strings to fetch.	
<code>\$column_slice</code>	<code>\phpcassa\ColumnSlice</code>	A slice of columns to fetch.	<code>null</code>
<code>\$column_names</code>	<code>mixed[]</code>	A list of column names to fetch.	<code>null</code>
<code>\$consistency_level</code>	<code>ConsistencyLevel</code>	The number of nodes that must respond before the method returns.	
<code>\$buffer_size</code>	<code>int</code>	The number of rows to fetch at a time. If the rows are large, a high buffer size degrades performance. If the rows are small, a high buffer size could benefit.	<code>16</code>

The `multiget()` method returns `mixed array(key => array(column_name => column_value))`. Create a PHP script, `get_multi.php`, in the `phpcassa-master\scripts` directory for fetching multiple rows from Cassandra. Then invoke the `multiget()` method with row arrays for 'catalog1' and 'catalog2'. For the `$column_names` argument, specify three columns: `journal`, `title`, and `author`. Iterate over the array returned by the method to output column values for the `title` and `author` columns.

```

foreach($catalogs as $catalog_id => $columns) {
    echo "Title: ".$columns["title"]."\n";
    echo "Author: ".$columns["author"]."\n";
    echo "<br>\n";
}

```

The `ColumnFamily` class provides the following method to get the column count for multiple rows in the same statement:

```

multiget_count($keys,$column_slice=null,$column_names=null,
$consistency_level=null)

```

The parameters for the `multiget_count()` method are the same as for the `multiget()` method except that the `multiget_count()` method does not have `$buffer_size` as a parameter. The method returns `mixed array(row_key => row_count)`. Invoke the method for the 'catalog1' and 'catalog2' rows.

```

$array= $catalog->multiget_count(["catalog1", "catalog2"]);

```

Output the array returned using `var_dump`:

```

var_dump($array);

```

The `get_multi.php` script appears in Listing 4.7.

Listing 4.7 The `get_multi.php` Script

```

<?php
require_once(__DIR__.'../../../lib/autoload.php');
use phpcassa\Connection\ConnectionPool;
use phpcassa\ColumnFamily;
use phpcassa\SystemManager;
$sys = new SystemManager('127.0.0.1');
$pool = new ConnectionPool('php_catalog', array('127.0.0.1'));
$catalog = new ColumnFamily($pool, 'catalog');
$catalogs = $catalog->multiget(array('catalog1', 'catalog2'));
foreach($catalogs as $catalog_id => $columns) {
    echo "Title: ".$columns["title"]."\n";
    echo "Author: ".$columns["author"]."\n";
    echo "<br>\n";
}
echo "<br>\n";
echo "Column Count: ";
$array= $catalog->multiget_count(["catalog1", "catalog2"]);
var_dump($array);
?>

```

Invoke the PHP script with the URL `http://localhost:8000/phpcassa-master/scripts/get_multi.php`. The two rows of data are fetched, and two columns of data are output. (See Figure 4.16.)

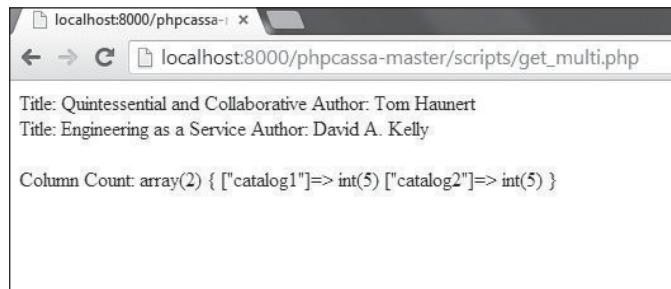


Figure 4.16

Getting data for selected columns from multiple rows.

Source: Google Inc.

GETTING COLUMN SLICES

You did not use all of the parameters in the `get()` method to fetch columns. Specifically, you did not use the `$column_slice` parameter, which fetches only the specified slice of columns. Next, you will use the `$column_slice` parameter to fetch a slice of columns. A slice of columns is represented with the `phpcassa\ColumnSlice` class. The constructor for the `ColumnSlice` class is as follows:

```
__construct ($start="", $finish="", $count=self::DEFAULT_COLUMN_COUNT,
$reversed=False)
```

The constructor parameters are discussed in Table 4.18.

Table 4.18 ColumnSlice Class Constructor Parameters

Parameter	Type	Description	Default Value
<code>\$start</code>	<code>mixed</code>	The column to start with. A value of '' implies the beginning of the row. The first column is column 1.	''
<code>\$finish</code>	<code>mixed</code>	The column to end with. A value of '' implies the end of the row.	''

(Continued)

Table 4.18 ColumnSlice Class Constructor Parameters (*Continued*)

Parameter	Type	Description	Default Value
\$count	int	The number of columns to fetch.	100
\$reversed	bool	If the column slice is to be reversed. The start becomes the finish and vice versa.	false

We will discuss `ColumnSlice` with several examples. Create a PHP script, `column_slices.php`, in the `phpcassa-master\scripts` directory. Then get the columns, starting with the first column from '`catalog2`' row.

```
$slice = new ColumnSlice(1);
var_dump($catalog->get('catalog2', $slice));
```

Next, get the column slice starting from the second column and ending with the fifth column from the '`catalog1`' row.

```
$slice = new ColumnSlice(2, 5);
var_dump($catalog->get('catalog1', $slice));
```

To demonstrate the `$count` parameter, get columns from the row '`catalog1`'. Although the start and finish are specified as ''', which implies all columns are to be fetched, only three columns are fetched because the `$count` is specified as 3. Three columns are fetched starting from the first column.

```
$slice = new ColumnSlice('', '', $count=3);
var_dump($catalog->get('catalog1', $slice));
```

Next, specify `start` and `finish` as '' and specify `$count` as 5. Also set `$reversed` to true. Five columns starting from the end are fetched. Because the total number of columns is five, all columns get fetched.

```
$slice = new ColumnSlice('', '', $count=5, $reversed=true);
var_dump($catalog->get('catalog1', $slice));
```

As another example, specify `$start` as 3 and `$finish` as '''. Then specify `$count` as 2 and set `$reversed` to true. Two columns, starting from the third column and moving toward the start of the row, are fetched.

```
$slice = new ColumnSlice(3, '', $count=2, $reversed=true);
var_dump($catalog->get('catalog1', $slice));
```

Because the start and finish index are specified as numbers, the `comparator_type` for the column family must be set to `LongType`.

```
$sys->create_column_family('catalogks', 'catalog', array(
    "column_type" => "Standard",
    "comparator_type" => "LongType",
    "key_validation_class" => "UTF8Type",
    "default_validation_class" => "UTF8Type"
));

```

The `column_slices.php` script appears in Listing 4.8.

Listing 4.8 The `column_slices.php` Script

```
<?php
require_once(__DIR__.'/../lib/autoload.php');
use phpcassa\Connection\ConnectionPool;
use phpcassa\ColumnFamily;
use phpcassa\ColumnSlice;
use phpcassa\SystemManager;
use phpcassa\Schema\StrategyClass;
$sys = new SystemManager('127.0.0.1');
$sys->create_keyspace('catalogks', array(
    "strategy_class" => StrategyClass::SIMPLE_STRATEGY,
    "strategy_options" => array('replication_factor' => '1')
));
$sys->create_column_family('catalogks', 'catalog', array(
    "column_type" => "Standard",
    "comparator_type" => "LongType",
    "key_validation_class" => "UTF8Type",
    "default_validation_class" => "UTF8Type"
));
// Start a connection pool, create our ColumnFamily instance
$pool = new ConnectionPool('catalogks', array('127.0.0.1'));
$catalog = new ColumnFamily($pool, 'catalog');
$columns = array(1 => "Oracle Magazine", 2 => "Oracle Publishing", 3 => "November-December 2013", 4 => "Quintessential and Collaborative", 5 => "Tom Haunert");
$catalog->insert('catalog1', $columns);
$columns = array(1 => "Oracle Magazine", 2 => "Oracle Publishing", 3 => "November-December 2013", 4 => "Engineering as a Service", 5 => "David A. Kelly");
$catalog->insert('catalog2', $columns);
$slice = new ColumnSlice(1);
var_dump(
$catalog->get('catalog2', $slice));
```

```

echo "<br>\n";
echo "<br>\n";
$slice = new ColumnSlice(2, 5);
var_dump(
$catalog->get('catalog1', $slice));
echo "<br>\n";
echo "<br>\n";
$slice = new ColumnSlice('', '', $count=3);
var_dump(
$catalog->get('catalog1', $slice));
echo "<br>\n";
echo "<br>\n";
$slice = new ColumnSlice('', '', $count=5, $reversed=true);
var_dump(
$catalog->get('catalog1', $slice));
echo "<br>\n";
echo "<br>\n";
$slice = new ColumnSlice(3, '', $count=2, $reversed=true);
var_dump(
$catalog->get('catalog1', $slice));
$sys->drop_keyspace("catalogks");
$pool->close();
$sys->close();
?>

```

Invoke the PHP script with the URL http://localhost:8000/phpcassa-master/scripts/column_slices.php. The results for the various `ColumnSlice` examples are output. (See Figure 4.17.)

```

array(5) { [1]=> string(15) "Oracle Magazine" [2]=> string(17) "Oracle Publishing" [3]=> string(22) "November-December 2013" [4]=> string(24) "Engineering as a Service" [5]=> string(14) "David A. Kelly" }

array(4) { [2]=> string(17) "Oracle Publishing" [3]=> string(22) "November-December 2013" [4]=> string(32) "Quintessential and Collaborative" [5]=> string(11) "Tom Haunert" }

array(3) { [1]=> string(15) "Oracle Magazine" [2]=> string(17) "Oracle Publishing" [3]=> string(22) "November-December 2013" }

array(5) { [5]=> string(11) "Tom Haunert" [4]=> string(32) "Quintessential and Collaborative" [3]=> string(22) "November-December 2013" [2]=> string(17) "Oracle Publishing" [1]=> string(15) "Oracle Magazine" }

array(2) { [3]=> string(22) "November-December 2013" [2]=> string(17) "Oracle Publishing" }

```

Figure 4.17
Getting column slices.

Source: Google Inc.

GETTING A RANGE OF ROWS AND COLUMNS

The ColumnFamily class provides yet another method to fetch columns of data from the server:

```
get_range($key_start="", $key_finish="", $row_count=self::DEFAULT_ROW_COUNT,
$column_slice=null, $column_names=null, $consistency_level=null,
$buffer_size=null)
```

It fetches a range of rows and columns. The method parameters are discussed in Table 4.19.

Table 4.19 Parameters in the get_range() Method

Parameter	Type	Description	Default Value
\$key_start	mixed	The start key to fetch rows.	""
\$key_finish	mixed	The finish key to fetch rows.	""
\$row_count	int	The number of rows to fetch.	100
\$column_slice	\phpcassa \ColumnSlice	The column slice to fetch.	null
\$column_names	mixed[]	A list of column names to fetch. By default all columns are fetched.	null
\$consistency_level	ConsistencyLevel	The number of nodes that must respond before the method returns.	null
\$buffer_size	int	The size of the buffer, in number of rows, to buffer intermediate results so that the Cassandra server does not overallocate memory and fail.	null

Create a PHP script, `get_range.php`, in the `phpcassa-master\scripts` directory. As an example, specify the range using the default `$key_start` and `$key_finish`, which is to include all the keys. Specify the number of rows to fetch with `$row_count` as `1000000`. Specifying a large number of rows does not fetch the rows if as many rows aren't in the server. Specify the array of columns to fetch as `array("1", "2", "3", "4", "5")`, which fetches columns 1 to 5.

```
$rows = $catalog->get_range("", "", 1000000, null, array("1", "2", "3", "4", "5"));
```

The `get_range()` method returns a `phpcassa\Iterator\RangeColumnFamilyIterator`, which may be iterated over using a `for` loop:

```
foreach($rows as $key => $columns) {
    echo $columns["1"]." ".$columns["2"]." ".$columns["3"]." ".$columns["4"]." ".
    ".$columns["5"];
}
```

If the `$key_start` and `$key_finish` are specified to be the same row, only one row is fetched. For example, the following invocation of `get_range()` fetches the `catalog1` row.

```
$rows = $catalog->get_range("catalog1", "catalog1", 1000000, null, array
("1", "2", "3", "4", "5"));
```

The `get_range.php` script appears in Listing 4.9.

Listing 4.9 The `get_range.php` Script

```
<?php
require_once(__DIR__.'/../lib/autoload.php');
use phpcassa\Connection\ConnectionPool;
use phpcassa\ColumnFamily;
use phpcassa\ColumnSlice;
use phpcassa\SystemManager;
use phpcassa\Schema\StrategyClass;
$sys = new SystemManager('127.0.0.1');
$sys->create_keyspace('ks', array(
    "strategy_class" => StrategyClass::SIMPLE_STRATEGY,
    "strategy_options" => array('replication_factor' => '1')
));
$sys->create_column_family('ks', 'catalog', array(
    "column_type" => "Standard",
    "comparator_type" => "LongType",
    "key_validation_class" => "UTF8Type",
    "default_validation_class" => "UTF8Type"
));
// Start a connection pool, create our ColumnFamily instance
$pool = new ConnectionPool('ks', array('127.0.0.1'));
$catalog = new ColumnFamily($pool, 'catalog');
$columns = array(1 => "Oracle Magazine", 2 => "Oracle Publishing", 3 => "November-
December 2013", 4 => "Quintessential and Collaborative", 5 => "Tom Haunert");
$catalog->insert('catalog1', $columns);
```

```
$columns = array(1 => "Oracle Magazine", 2 => "Oracle Publishing", 3 => "November-  
December 2013", 4 => "Engineering as a Service", 5 => "David A. Kelly");$catalog-  
>insert('catalog2', $columns);  
$rows = $catalog->get_range("", "", 1000000, null, array("1", "2", "3", "4", "5"));  
foreach($rows as $key => $columns) {  
    echo $columns["1"]." ".$columns["2"]." ".$columns["3"]." ".$columns["4"]."."  
".$columns["5"];  
}  
// $rows = $catalog->get_range("catalog1", "catalog1", 1000000, null, array("1",  
"2", "3", "4", "5"));  
// foreach($rows as $key => $columns) {  
//     echo $columns["1"]." ".$columns["2"]." ".$columns["3"]." ".$columns["4"]."."  
".$columns["5"];  
// }  
$sys->drop_keyspace("ks");  
$pool->close();  
$sys->close();  
?>
```

Invoke the PHP script with the URL http://localhost:8000/phpcassa-master/scripts/get_range.php. All the columns from the two rows in the database are output. (See Figure 4.18.)

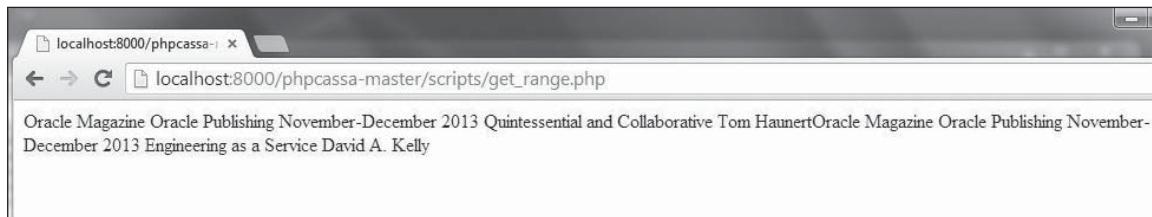
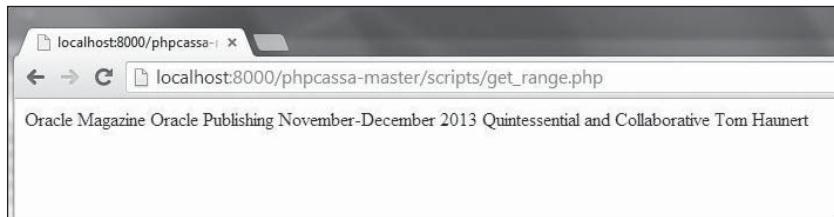


Figure 4.18

Getting data for a range of rows and columns.

Source: Google Inc.

In the example in which `$key_start` is the same as the `$key_finish`, only the specified key, `catalog1`, is output. (See Figure 4.19.)

**Figure 4.19**

The result when the start key is the same as the finish key.

Source: Google Inc.

UPDATING DATA

In this section, you will update data. Create a PHP script, `update.php`, in the `phpcassa-master\scripts` directory. The `insert()` method, which you used to add data, may also be used to update data. In the `update.php` script, add `catalog1` and `catalog2` rows to the `catalog_update` column family in the `catalog_update` keyspace. The required keyspace and column family are created in the script. Create a `ConnectionPool` and a `ColumnFamily` instance as before. Add data using the `insert()` method. Then invoke the `insert()` method again but with slightly modified column values. Output the column names and values before modification and after modification. The `update.php` script appears in Listing 4.10.

Listing 4.10 The `update.php` Script

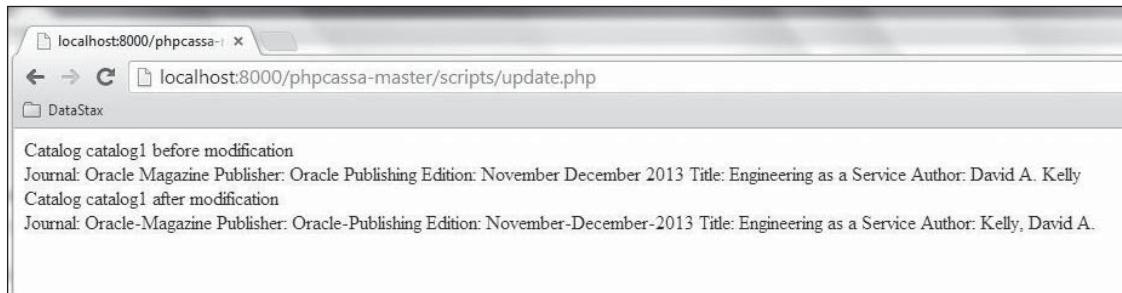
```
<?php
require_once(__DIR__.'/../lib/autoload.php');
use phpcassa\Connection\ConnectionPool;
use phpcassa\ColumnFamily;
use phpcassa\SystemManager;
use phpcassa\Schema\StrategyClass;
$sys = new SystemManager('127.0.0.1');
$sys->create_keyspace('catalog_update', array(
    "strategy_class" => StrategyClass::SIMPLE_STRATEGY,
    "strategy_options" => array('replication_factor' => '1')));
$sys->create_column_family('catalog_update', 'catalog', array(
    "column_type" => "Standard",
    "comparator_type" => "UTF8Type",
```

```

"key_validation_class" => "UTF8Type",
"default_validation_class" => "UTF8Type"
));
$pool = new ConnectionPool('catalog_update', array('127.0.0.1'));
$catalog = new ColumnFamily($pool, 'catalog');
$catalog->insert('catalog1', array("journal" => "Oracle Magazine", "publisher" => "Oracle Publishing", "edition" => "November December 2013", "title" => "Engineering as a Service", "author" => "David A. Kelly"));
echo 'Catalog catalog1 before modification ';
echo "<br>\n";
$columns = $catalog->get('catalog1');
echo "Journal: ".$columns["journal"]."\n";
echo "Publisher: ".$columns["publisher"]."\n";
echo "Edition: ".$columns["edition"]."\n";
echo "Title: ".$columns["title"]."\n";
echo "Author: ".$columns["author"]."\n";
echo "<br>\n";
$catalog->insert('catalog1', array("journal" => "Oracle-Magazine", "publisher" => "Oracle-Publishing", "edition" => "November-December-2013", "title" => "Engineering as a Service", "author" => "Kelly, David A."));
echo 'Catalog catalog1 after modification ';
echo "<br>\n";
$columns = $catalog->get('catalog1');
echo "Journal: ".$columns["journal"]."\n";
echo "Publisher: ".$columns["publisher"]."\n";
echo "Edition: ".$columns["edition"]."\n";
echo "Title: ".$columns["title"]."\n";
echo "Author: ".$columns["author"]."\n";
echo "<br>\n";
$sys->drop_keyspace("catalog_update");
$pool->close();
$sys->close();
?>

```

Invoke the PHP script with the URL <http://localhost:8000/phpcassa-master/scripts/update.php>. All the columns from the two rows in the database are output before and after the update. (See Figure 4.20.)

**Figure 4.20**

Updating data.

Source: Google Inc.

Next, you will delete the data added and also delete the column family and keyspace.

DELETING DATA

The `ColumnFamily` provides the following method to remove columns from a row:

```
remove($key, $column_names=null, $consistency_level=null)
```

The method parameters are discussed in Table 4.20.

Table 4.20 Parameters of the `remove()` Method

Parameter	Type	Description	Default Value
<code>\$key</code>	<code>string</code>	The row key to remove.	
<code>\$column_names</code>	<code>mixed[]</code>	The array of columns to remove. By default all columns are removed.	<code>null</code>
<code>\$consistency_level</code>	<code>ConsistencyLevel</code>	The number of nodes that must respond the method returns.	<code>null</code>

Create a PHP script, `delete.php`, in the `phpcassa-master\scripts` directory. Then create a `ColumnFamily` instance as before. The `remove()` method must be invoked for each row key to remove. Remove the `catalog1` and `catalog2` rows.

```
$catalog->remove("catalog1");
$catalog->remove("catalog2");
```

The delete.php script appears in Listing 4.11.

Listing 4.11 The delete.php Script

```
<?php
require_once(__DIR__.'/../lib/autoload.php');
use phpcassa\Connection\ConnectionPool;
use phpcassa\ColumnFamily;
use phpcassa\SystemManager;
$sys = new SystemManager('127.0.0.1');
$pool = new ConnectionPool('php_catalog', array('127.0.0.1'));

$catalog = new ColumnFamily($pool, 'catalog');
$catalog->remove("catalog1");
$catalog->remove("catalog2");
//$catalog->remove("catalog3");
//$catalog->remove("catalog4");
//$catalog->remove("catalog5");
echo 'Catalog ids catalog1 and catalog2 removed';
?>
```

Invoke the PHP script with the URL <http://localhost:8000/phpcassa-master/scripts/delete.php>. The two rows, catalog1 and catalog2, are removed. (See Figure 4.21.)

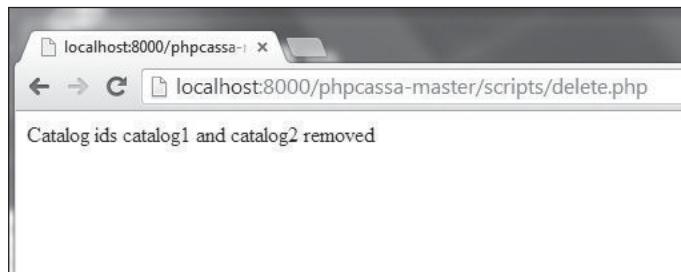


Figure 4.21

Deleting data.

Source: Google Inc.

DROPPING THE KEYSPACE AND COLUMN FAMILY

The SystemManager class provides the methods discussed in Table 4.21 to remove a column family or data from a column family as well as to remove a keyspace.

Table 4.21 SystemManager Class Methods to Remove a Column Family

Method	Description
<code>drop_column_family(\$keyspace, \$column_family)</code>	Drops a column family from a keyspace
<code>truncate_column_family(\$keyspace, \$column_family)</code>	Deletes all data from a column family
<code>drop_keyspace(mixed \$keyspace)</code>	Drops a keyspace

The `ColumnFamily` class provides the `truncate()` method to delete all data from a column family. Create a PHP script, `dropCFKeyspace.php`, in the `phpcassa-master\scripts` directory. Then create a `ColumnFamily` instance as before.

```
$sys = new SystemManager('127.0.0.1');
$pool = new ConnectionPool('php_catalog', array('127.0.0.1'));
$catalog = new ColumnFamily($pool, 'catalog');
```

Next, invoke the `truncate()` method to remove all data from the column family. Invoke the `drop_keyspace(mixed $keyspace)` method to delete the `php_catalog` keyspace.

```
$catalog->truncate();
$sys->drop_keyspace("php_catalog");
```

The `dropCFKeyspace.php` script appears in Listing 4.12.

Listing 4.12 The dropCFKeyspace.php Script

```
<?php
require_once(__DIR__.'/../lib/autoload.php');
use phpcassa\Connection\ConnectionPool;
use phpcassa\ColumnFamily;
use phpcassa\SystemManager;
use phpcassa\Schema\StrategyClass;
$sys = new SystemManager('127.0.0.1');
$pool = new ConnectionPool('php_catalog', array('127.0.0.1'));
$catalog = new ColumnFamily($pool, 'catalog');
$catalog->truncate();
$sys->drop_keyspace("php_catalog");
$pool->close();
$sys->close();
echo 'removed Column Family and Keyspace';
?>
```

Invoke the PHP script with the URL `http://localhost:8000/phpcassa-master/scripts/dropCFKeyspace.php`. The column family is truncated, but is not removed. Subsequently, the keyspace is removed, which also removes the column family. (See Figure 4.22.)

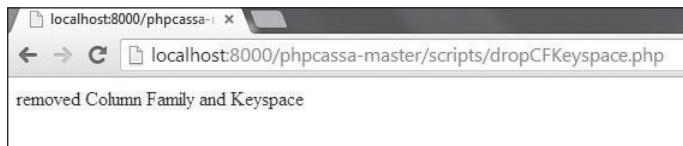


Figure 4.22
Dropping a keyspace.

Source: Google Inc.

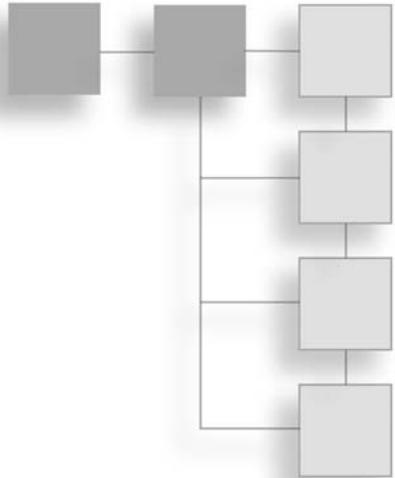
SUMMARY

This chapter discussed the phpcassa PHP client library for Apache Cassandra to connect to Cassandra server, create a keyspace, create a column family, add data, fetch data, update data, delete data, and drop the keyspace. In the next chapter, you will use the Ruby client for Cassandra to access Cassandra and perform similar create, read, update, delete (CRUD) operations.

This page intentionally left blank

CHAPTER 5

USING A RUBY CLIENT WITH CASSANDRA



Apache Cassandra stores data in a table format. A relational database also stores data in a table. The difference is that Cassandra's table format is not based on a fixed schema. Rather, it is based on a flexible schema. In a relational database table, each row has the same columns, column types, and number of columns. In Cassandra, each table row could have different column types and number of columns.

Ruby is an open source programming language, most commonly used in the Ruby on Rails framework. This chapter discusses using a Ruby client to access and make data changes in Cassandra.

SETTING THE ENVIRONMENT

Download the following software for Ruby:

- RubyInstaller rubyinstaller-1.9.3-p484.exe or a later version from <http://rubyinstaller.org/>.
- RubyGems.
- RubyInstaller development kit DevKit-mingw64-64-4.7.2-20130224-1432-sfx.exe from <http://rubyinstaller.org/downloads/>. The Development Kit file is different based on the Ruby version used and the OS architecture (32 bit or 64 bit).

To install Ruby, RubyGems, and the RubyInstaller development kit, follow these steps:

1. Double-click the RubyInstaller application.
2. Choose a setup language in Select Setup Language screen.
3. Accept the license agreement and click Next.
4. Select a destination folder in which to install Ruby. The directory path should contain no spaces.
5. Select the Add Ruby Executables to Your PATH checkbox and click Next, as shown in Figure 5.1. Installation begins, as shown in Figure 5.2.

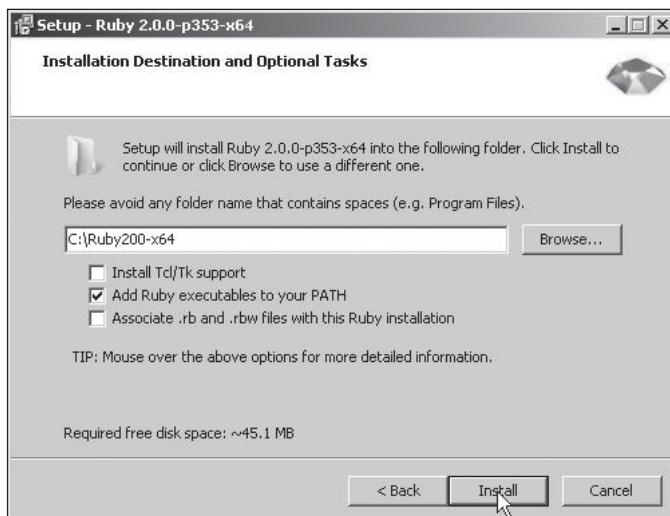


Figure 5.1
Specifying installation location and optional tasks for installing Ruby.

Source: RubyInstaller Contributors.

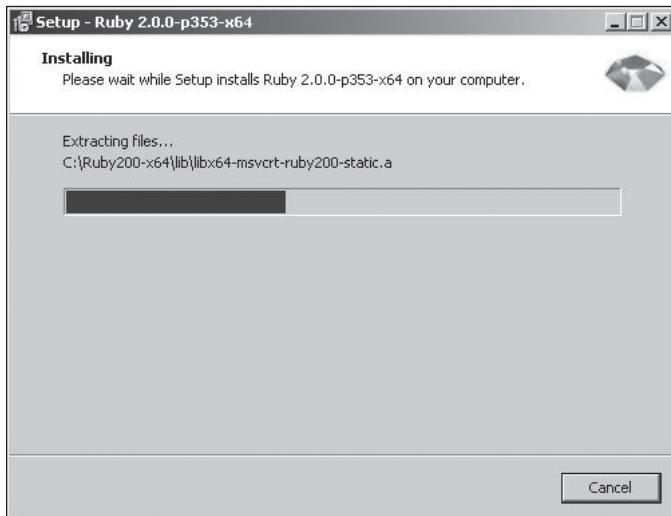


Figure 5.2
Installing Ruby.

Source: RubyInstaller Contributors.

6. When the Ruby installation is complete, click Finish, as shown in Figure 5.3. Add the Ruby installation bin directory to the PATH user variable for the user logged into the operating system.

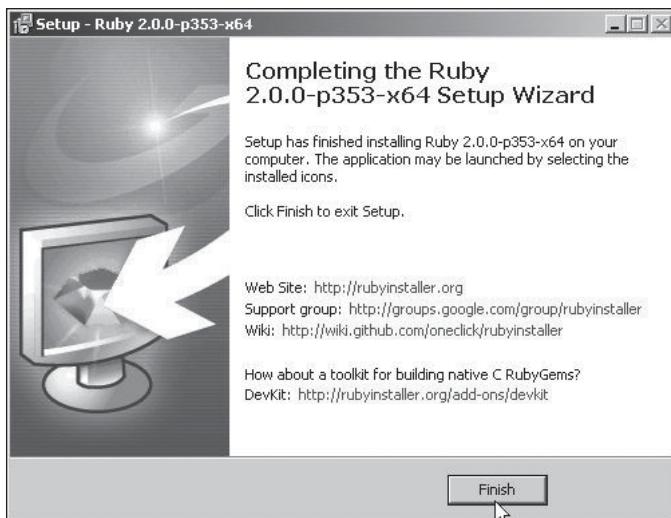


Figure 5.3
Completing the Ruby installation.
Source: RubyInstaller Contributors.

7. Next, install RubyGems, which is a package-management framework for Ruby. Use the following command:

```
gem install rubygems-update
```

The output from this command indicates that RubyGems has been installed, as shown in Figure 5.4.

A screenshot of a Windows Command Prompt window titled "Command Prompt". The window shows the following text output:
C:\>gem install rubygems-update
Fetching: rubygems-update-2.2.1.gem (100%)
Successfully installed rubygems-update-2.2.1
Parsing documentation for rubygems-update-2.2.1
Installing ri documentation for rubygems-update-2.2.1
1 gem installed
C:\>

Figure 5.4
Installing RubyGems.
Source: Microsoft Corporation.

8. Install the RubyInstaller development kit, which is a toolkit to build C/C++ extensions for Ruby. To begin, double-click the application to extract the application files to a directory, the same directory in which RubyGems was installed, as shown in Figure 5.5.

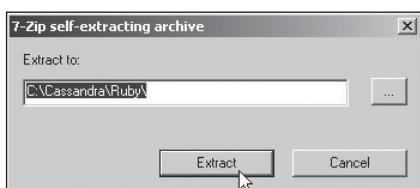


Figure 5.5
Extracting the development kit.
Source: RubyInstaller Contributors.

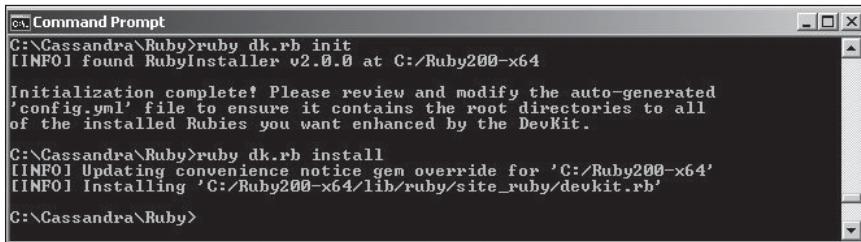
9. Run the following two commands to initialize and install the development kit, but only run the first command initially, as some configuration is required before running the second command:

```
ruby dk.rb init  
ruby dk.rb install
```

10. The output from the first command, shown in Figure 5.6, indicates that initialization generates a config.yml file in the same directory from which the first command is run. Modify the config.yml to add the following line:

```
- C:/Ruby200-x64
```

C:/Ruby200-x64 is the directory in which Ruby is installed. Run the subsequent (second) command after modifying config.yml. The subsequent command enhances the installed Rubies.



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The command "ruby dk.rb init" is run, followed by "Initialization complete! Please review and modify the auto-generated 'config.yml' file to ensure it contains the root directories to all of the installed Rubies you want enhanced by the DevKit.". Then, the command "ruby dk.rb install" is run, updating the convenience notice gem override for 'C:/Ruby200-x64' and installing 'C:/Ruby200-x64/lib/ruby/site_ruby/devkit.rb'. The prompt then changes to "C:\Cassandra\Ruby>".

Figure 5.6
Installing DevKit.

Source: Microsoft Corporation.

11. Install Apache Cassandra and start Cassandra with the following command:
`cassandra -f`

INSTALLING A RUBY CLIENT WITH CASSANDRA

In this section, you'll install the Ruby client for Cassandra. Run the following command from Windows command prompt:

```
gem install cassandra
```

As the output in Figure 5.7 indicates, cassandra 0.23.0 gem is installed.

```
C:\>Cassandra\Ruby>gem install cassandra
Temporarily enhancing PATH to include DevKit...
Building native extensions. This could take a while...
Successfully installed thrift-0.8.0
Fetching: thrift_client-0.8.4.gem <100%
Successfully installed thrift_client-0.8.4
Fetching: simple_uuid-0.4.0.gem <100%
Successfully installed simple_uuid-0.4.0
Fetching: cassandra-0.23.0.gem <100%
Building native extensions. This could take a while...
Successfully installed cassandra-0.23.0
Parsing documentation for thrift-0.8.0
unable to convert "\x86" from ASCII-8BIT to UTF-8 for ext/binary_protocol_accele
rated.o, skipping
unable to convert "\x86" from ASCII-8BIT to UTF-8 for ext/compact_protocol.o, sk
ipping
unable to convert "\x86" from ASCII-8BIT to UTF-8 for ext/memory_buffer.o, skip
ping
unable to convert "\x86" from ASCII-8BIT to UTF-8 for ext/protocol.o, skipping
unable to convert "\x86" from ASCII-8BIT to UTF-8 for ext/strncpy.o, skipping
unable to convert "\x86" from ASCII-8BIT to UTF-8 for ext/struct.o, skipping
unable to convert "\x86" from ASCII-8BIT to UTF-8 for ext/thrift_native.o, skip
ping
unable to convert "\x90" from ASCII-8BIT to UTF-8 for ext/thrift_native.so, skip
ping
unable to convert "\x90" from ASCII-8BIT to UTF-8 for lib/thrift_native.so, skip
ping
Installing ri documentation for thrift-0.8.0
Parsing documentation for thrift_client-0.8.4
Installing ri documentation for thrift_client-0.8.4
Parsing documentation for simple_uuid-0.4.0
Installing ri documentation for simple_uuid-0.4.0
Parsing documentation for cassandra-0.23.0
Installing ri documentation for cassandra-0.23.0
4 gems installed
C:\>Cassandra\Ruby>
```

Figure 5.7

Installing Ruby client for Cassandra.

Source: Microsoft Corporation.

CREATING A CONNECTION

To create a connection with Cassandra using Ruby, create a Ruby script, connection.rb. Add a require statement to import the default version of the Ruby client library for Cassandra. Using the class constructor for the Cassandra class, create an instance of Cassandra. Supply the constructor args (arguments) for the keyspace and servers.

```
require 'cassandra'
client = Cassandra.new('system', '127.0.0.1:9160')
```

A connection to Cassandra database is created. Some of the attributes provided by the Cassandra class are listed in Table 5.1.

Table 5.1 Cassandra Class Attributes

Attribute	Description
keyspace	Returns the keyspace
servers	Returns the servers array
thrift_client_class	Returns the Thrift client class
thrift_client_options	Returns the Thrift client options

The Cassandra class also provides some instance methods to get information about the database, as discussed in Table 5.2.

Table 5.2 Cassandra Class Methods to Get Information About the Cluster

Method	Description
cluster_name	Returns the cluster name.
keyspaces	Returns an array of keyspaces.
partitioner	Returns a string for the partitioner used in the cluster. The default partitioner will be Murmur3Partitioner for Cassandra versions 1.2 and later or RandomPartitioner for versions prior to 1.2.
ring	Returns an array of tokens indicating the servers.
inspect	Returns a string containing @keyspace,@schema,@servers.
version	The Cassandra Thrift version.

Using the Cassandra instance client, invoke some of these attributes and methods. The connection.rb script appears in Listing 5.1.

Listing 5.1 The connection.rb Script

```
print client.keyspace
print "\n"
print client.servers
print "\n"
print client.thrift_client_class
print "\n"
```

```
print client.thrift_client_options
print "\n"
print client.keyspaces
print "\n"
print client.version
print "\n"
print client.inspect
```

Run the script with the following command:

```
ruby connection.rb
```

The output from the script is shown in Figure 5.8.

Figure 5.8

Connecting with Cassandra.

Source: Microsoft Corporation.

CREATING A KEYSPACE

Next, you will create a keyspace in the Cassandra database. Create a Ruby script, `createKeyspace.rb`, and add the require statement for the Ruby client library for Cassandra. Create an instance of the `Cassandra` class as in the previous section. Invoke the `disable_node_auto_discovery!` method, which is used primarily if the Cassandra cluster is communicating internally on a different IP address than the IP address on which a client connects. Create an instance of the `Cassandra::Keyspace` class.

```
ks = Cassandra::Keyspace.new
```

Set the `name`, `strategy_class`, `ks.strategy_options`, and `ks.cf_defs` attributes for the `Keyspace` class instance. To create a Keyspace named `catalog`, set the name to '`'catalog'`'. Specify the replica placement strategy for the new keyspace to `org.apache.cassandra.locator.SimpleStrategy` using the `strategy_class` attribute. Set the `replication_factor` to 1 with the `strategy_options` attribute. Set the column family definitions to an empty array using the `cf_defs` attribute.

```
ks.name = 'catalog'
ks.strategy_class = 'org.apache.cassandra.locator.SimpleStrategy'
ks.strategy_options={'replication_factor'=>'1'}
ks.cf_defs = []
```

Add the keyspace to the Cassandra database using the `add_keyspace(ks_def)` method of the `Cassandra` class.

```
client.add_keyspace(ks)
```

Add a `print` statement for the keyspaces, which should include the newly added `catalog` keyspace.

```
print client.keyspaces
```

The `createKeyspace.rb` script appears in Listing 5.2.

Listing 5.2 The `createKeyspace.rb` Script

```
require 'cassandra'
client = Cassandra.new('system', '127.0.0.1:9160')
client.disable_node_auto_discovery!
ks = Cassandra::Keyspace.new
ks.name = 'catalog'
ks.strategy_class = 'org.apache.cassandra.locator.SimpleStrategy'
ks.strategy_options={'replication_factor'=>'1'}
ks.cf_defs = []
client.add_keyspace(ks)
print "\n"
print client.keyspaces
```

Run the script with the following command:

```
ruby createKeyspace.rb
```

A new keyspace called `catalog` is created and listed in the `keyspaces` array, as shown in Figure 5.9.

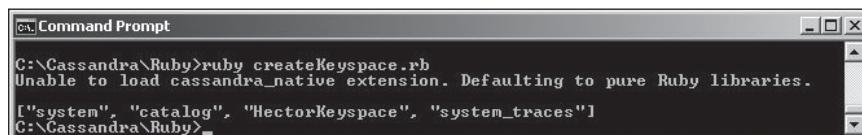


Figure 5.9

Creating a keyspace.

Source: Microsoft Corporation.

CREATING A COLUMN FAMILY

Having added a keyspace to the Cassandra database, you will next add a column family to the keyspace. Create a Ruby script, `createCF.rb`. Import the Ruby client library for Cassandra and create a connection to the Cassandra database as before. Also invoke the `disable_node_auto_discovery!` method. A column family is represented with the `Cassandra::ColumnFamily` class. Create an instance of the `ColumnFamily` class using the `Cassandra::ColumnFamily.new` class constructor. Specify the `:keyspace` arg for the keyspace to be used and the `:name` arg for the column family to be created. Add a `catalog` column family to a `catalog` keyspace as follows:

```
cf_def = Cassandra::ColumnFamily.new(:keyspace => "catalog", :name => "catalog")
```

Add the column family to the Cassandra database using the `Cassandra` class method `add_column_family(cf_def)`:

```
client.add_column_family(cf_def)
```

Print the column families using the `print column_families` method:

```
print client.column_families
```

The `createCF.rb` script appears in Listing 5.3.

Listing 5.3 The `createCF.rb` Script

```
require 'cassandra'
client = Cassandra.new('catalog','127.0.0.1:9160')
client.disable_node_auto_discovery!
cf_def = Cassandra::ColumnFamily.new(:keyspace => "catalog", :name => "catalog")
client.add_column_family(cf_def)
print client.column_families
```

Run the `createCF.rb` script with the following command:

```
ruby createCF.rb
```

The output from the Ruby script lists the newly created `catalog` column family, as shown in Figure 5.10.

```
C:\Cassandra\Ruby>ruby createCF.rb
Unable to load cassandra_native extension. Defaulting to pure Ruby libraries.
<"catalog">=>CassandraThrift::CFDef keyspace:"catalog", name:"catalog", column_type:"Standard", comparator_type:"org.apache.cassandra.db.marshal.BytesType", comment:"", row_cache_size:0, key_cache_size:200000, read_repair_chance:1.0, column_metadata:[], gc_grace_seconds:864000, default_validation_class:"org.apache.cassandra.db.marshal.BytesType", min_compaction_threshold:4, max_compaction_threshold:32, replicate_on_write:true, key_validation_class:"org.apache.cassandra.db.marshal.BytesType", row_cache_provider:"org.apache.cassandra.cache.ConcurrentLinkedHashCacheProvider", key_alias:6b6579>>
C:\Cassandra\Ruby>
```

Figure 5.10

Creating a column family.

Source: Microsoft Corporation.

A new column family can be added only to a user-created keyspace. For example, if a new column family is added to the system keyspace, the following error is generated:

```
system keyspace is not user-modifiable. (CassandraThrift::  
InvalidRequestException)  
from C:/Ruby200-x64/lib/ruby/gems/2.0.0/gems/cassandra-0.23.0/vendor/0.8/  
gen-rb/cassandra.rb:417:in 'system_add_column_family'
```

ADDING DATA TO A TABLE

Having added a column family, you will next add data to the column family (table). Create a Ruby script, add.rb. Create a connection to the Cassandra database as before. The Cassandra class provides the `insert(column_family, key, hash, options = {})` method to add a row to a database column family. A row is identified by a key. The columns in a row are supplied using a hash of key/value pairs, with each key/value pair representing the column name and the column value. Add two rows of data identified by catalog1 and catalog2 using the `insert()` method. Each row has columns journal, publisher, edition, title and author.

```
print client.insert(:catalog, "catalog1", {'journal' => 'Oracle Magazine',
'publisher' => 'Oracle Publishing', 'edition' => 'November-December 2013', 'title' => 'Engineering as a Service', 'author' => 'David A. Kelly'})
print client.insert(:catalog, "catalog2", {'journal' => 'Oracle Magazine',
'publisher' => 'Oracle Publishing', 'edition' => 'November-December 2013', 'title' => 'Quintessential and Collaborative', 'author' => 'Tom Haunert'})
```

The Cassandra class provides several methods to get information about data in rows—for example, the number of columns and whether a particular column exists. Some of these methods are discussed in Table 5.3.

Table 5.3 Cassandra Class Methods to Get Information About Columns and Rows

Method	Description
count_columns	Returns the number of columns in the specified row
count_range	Returns the range count, which is the number of keys in the range
multi_count_columns	Returns the number of columns in the specified rows
get_range_keys	Returns an array containing all the keys in the given range
exists?	Returns a Boolean (true or false) to indicate if the requested path exists

Output the number of rows in the catalog column family as follows:

```
print client.count_range(:catalog)
```

Output the number of columns in the row identified by the catalog1 key in the catalog column family:

```
print client.count_columns(:catalog, "catalog1")
```

Output the number of columns in the catalog1 and catalog2 rows in the catalog column family:

```
print client.multi_count_columns(:catalog, ["catalog1", "catalog2"])
```

Output the range of keys in the catalog column family with key count limited to 2:

```
print client.get_range_keys(:catalog, :key_count => 2)
```

Find out whether the journal column in the catalog1 row in the catalog column family exists:

```
print client.exists?(:catalog, "catalog1", 'journal')
```

The add.rb Ruby script appears in Listing 5.4.

Listing 5.4 The add.rb Script

```
require 'cassandra'
client = Cassandra.new('catalog', '127.0.0.1:9160')
client.disable_node_auto_discovery!
```

```

print client.insert(:catalog, "catalog1", {'journal' => 'Oracle Magazine',
'publisher' => 'Oracle Publishing', 'edition' => 'November-December 2013', 'title'
=> 'Engineering as a Service', 'author' => 'David A. Kelly'})
print client.insert(:catalog, "catalog2", {'journal' => 'Oracle Magazine',
'publisher' => 'Oracle Publishing', 'edition' => 'November-December 2013', 'title'
=> 'Quintessential and Collaborative', 'author' => 'Tom Haunert'})
print client.count_range(:catalog)
print "\n"
print client.count_columns(:catalog, "catalog1")
print "\n"
print client.multi_count_columns(:catalog, ["catalog1", "catalog2"])
print "\n"
print client.get_range_keys(:catalog, :key_count => 2)
print "\n"
print client.exists?(:catalog, "catalog1", 'journal')
print "\n"

```

Run the add.rb script with the following command:

```
ruby add.rb
```

Two rows, catalog1 and catalog2, are added, and the requested information about the rows and columns is output. The range count for the catalog column family is 2. The number of columns in the catalog1 row is 5. The number of columns in the catalog1 and catalog2 rows is 5 each. The range of keys in the catalog column family with the key count limited to 2 is catalog1 and catalog2 output as an array. The journal column in the catalog1 row exists. The output from add.rb is shown in Figure 5.11.

```

C:\Cassandra\Ruby>ruby add.rb
Unable to load cassandra_native extension. Defaulting to pure Ruby libraries.
2
5
<"catalog1">=5, <"catalog2">=5>
[<"catalog1", "catalog2"]]
true
C:\Cassandra\Ruby>

```

Figure 5.11
Adding data to Cassandra.
Source: Microsoft Corporation.

You added two rows with the same columns, but different rows may have a different number of columns, a different order of columns, or different types of columns. The flexible schema for the Cassandra column family is what makes Cassandra suitable for

heterogeneous data. For example, the following three rows may be added to the same column family:

```
print client.insert(:catalog, "catalog1", {'journal' => 'Oracle Magazine',
'publisher' => 'Oracle Publishing', 'edition' => 'November-December 2013', 'title'
=> 'Engineering as a Service', 'author' => 'David A. Kelly'})
print client.insert(:catalog, "catalog2", {'journal' => 'Oracle Magazine',
'publisher' => 'Oracle Publishing', 'edition' => 'November-December 2013'})
print client.insert(:catalog, "catalog3", { 'publisher' => 'Oracle Publishing',
'journal' => 1, 'edition' => '11122013'})
```

ADDING ROWS IN BATCH

In this section, you will add rows in a batch. Create a Ruby script, `add_batch.rb`. The `Cassandra` class provides the `batch` method to make mutations in a batch. A mutation could be an insert/delete. The `batch` method takes two options, discussed in Table 5.4.

Table 5.4 Batch Method Options

Option	Description
<code>:consistency</code>	The consistency level from individual mutations.
<code>:queue_size</code>	The maximum number of mutations to send at once. The last batch of mutations could be less than <code>queue_size</code> . By default, all mutations are sent as a single batch.

The batch of mutations is sent using the following `do end` construct in which `client` is the connection object to the Cassandra database:

```
client.batch do
end
```

As an example, add five rows of data with the keys `catalog1`, `catalog2`, `catalog3`, `catalog4`, and `catalog5`. Then invoke the `exists?` method to determine whether each of the rows did get added. The `add_batch.rb` script appears in Listing 5.5.

Listing 5.5 The `add_batch.rb` Script

```
require 'cassandra'
client = Cassandra.new('catalog', '127.0.0.1:9160')
client.disable_node_auto_discovery!
client.batch do
```

```

client.insert(:catalog, "catalog1", {'journal' => 'Oracle Magazine', 'publisher' => 'Oracle Publishing', 'edition' => 'November-December 2013', 'title' => 'Engineering as a Service', 'author' => 'David A. Kelly'})
client.insert(:catalog, "catalog2", {'journal' => 'Oracle Magazine', 'publisher' => 'Oracle Publishing', 'edition' => 'November-December 2013', 'title' => 'Quintessential and Collaborative', 'author' => 'Tom Haunert'})
client.insert(:catalog, "catalog3", {'journal' => 'Oracle Magazine', 'publisher' => 'Oracle Publishing', 'edition' => 'November-December 2013'})
client.insert(:catalog, "catalog4", {'journal' => 'Oracle Magazine', 'publisher' => 'Oracle Publishing', 'edition' => 'November-December 2013'})
client.insert(:catalog, "catalog5", {'journal' => 'Oracle Magazine', 'publisher' => 'Oracle Publishing', 'edition' => 'November-December 2013'})
#client.remove(:catalog, "catalog3")
end
# catalog2 catalog3 catalog4 catalog5 catalog1
print client.exists?(:catalog, "catalog1")
print "\n"
print client.exists?(:catalog, "catalog2")
print "\n"
print client.exists?(:catalog, "catalog3")
print "\n"
print client.exists?(:catalog, "catalog4")
print "\n"
print client.exists?(:catalog, "catalog5")

```

Run the `add_batch.rb` script with the following command:

```
ruby add_batch.rb
```

The output for each of the `exists?` method invocations is `true`, as shown in Figure 5.12.

```
C:\>Administrator: C:\Windows\system32\cmd.exe
C:\>Ruby\ruby_cassandra>ruby add_batch.rb
Unable to load cassandra_native extension. Defaulting to pure Ruby libraries.
true
true
true
true
true
C:\>Ruby\ruby_cassandra>
```

Figure 5.12

Adding data in a batch.

Source: Microsoft Corporation.

With the default (`Cassandra>1.2`) `Murmur3Partitioner`, which is similar to the `RandomPartitioner` (`Cassandra<1.2`), the order in which the rows are added is random

because `Murmur3Partitioner/RandomPartitioner` distributes rows across the cluster evenly by the md5 encryption hash. For example, in the preceding example, the order is not the following:

```
catalog1 catalog2 catalog3 catalog4 catalog5
```

Instead, the order in which the rows are added is as follows:

```
catalog2 catalog3 catalog4 catalog5 catalog1
```

If the rows are to be added in the order specified, `ByteOrderedPartitioner/OrderPreservingPartitioner` must be used.

Next, we will discuss the different partitioners supported by Cassandra and how to set a non-default partitioner. Cassandra supports the partitioners listed in Table 5.5.

Table 5.5 Cassandra Partitioners

Partitioner	Description
<code>RandomPartitioner</code>	Distributes rows across the cluster evenly by md5. This was the default prior to version 1.2 and is retained for compatibility.
<code>Murmur3Partitioner</code>	Similar to <code>RandomPartitioner</code> , but uses the Murmur3_128 hash function instead of md5. When in doubt, this is the best option.
<code>ByteOrderedPartitioner</code>	Orders rows lexically by key bytes. Allows the scanning of rows in key order, but the ordering can generate hot spots for sequential insertion workloads.
<code>OrderPreservingPartitioner</code>	An obsolete (deprecated) form of <code>ByteOrderedPartitioner</code> that stores keys in a less-efficient format. Works only with keys that are UTF8-encoded strings.
<code>CollatingOPP</code>	Collates according to EN,US (the language code for English-United States) rules rather than lexical byte ordering. Use this as an example if you need custom collation.

The partitioner is set with the `partitioner` key in the `C:\Cassandra\apache-cassandra-2.0.4\conf\cassandra.yaml` configuration file. To add rows in the order specified, set `partitioner` to `OrderPreservingPartitioner` or `ByteOrderedPartitioner`.

```
partitioner: org.apache.cassandra.dht.OrderPreservingPartitioner
```

Restart Cassandra after modifying the `cassandra.yaml` file.

RETRIEVING DATA FROM A TABLE

The Cassandra class provides the `get()` method to return a hash (`Cassandra::OrderedHash`) representing the element at the `column_family:key:[column]` path supplied to the method. The `get()` method takes the parameters discussed in Table 5.6.

Table 5.6 Parameters for `get()` Method

Parameter	Description
<code>column_family</code>	The column family
<code>key</code>	The row key
<code>columns</code>	The list of columns in the row
<code>sub_columns</code>	The list of subcolumns to select
<code>options</code>	Options, further describing the data to get

Only the `column_family` and `key` are required parameters. The options supported by the `get()` method are discussed in Table 5.7.

Table 5.7 `get()` Method Options

Namespace	Description
<code>:count</code>	The number of columns to be returned. By default, all columns are returned.
<code>:start</code>	The starting column for selecting a range of columns.
<code>:finish</code>	The final value for selecting a range of columns.
<code>:reversed</code>	A Boolean indicating whether the columns are to be reversed. Set to <code>false</code> by default.
<code>:consistency</code>	The read consistency.

The column family specified must be valid. For example, try supplying the column family as `catalog_journal`. The following error is generated:

```
'column_family_property': Invalid column family "catalog_journal"
(Cassandra::AccessError)
```

SELECTING A SINGLE ROW

Create a Ruby script, get.rb. Create a connection with the Cassandra database using an instance of the Cassandra class. Get the catalog1 row in the catalog column family as follows:

```
print client.get(:catalog, "catalog1")
```

Get the title column in the catalog1 row in the catalog column family as follows:

```
print client.get(:catalog, "catalog1", 'title')
```

Get three columns from the catalog1 row in the catalog column family as follows:

```
print client.get(:catalog, "catalog1", :count => 3)
```

Get the columns in the catalog2 row in reversed order as follows.

```
print client.get(:catalog, "catalog2", :reversed=>true)
```

The get.rb script appears in Listing 5.6.

Listing 5.6 The get.rb Script

```
require 'cassandra'
client = Cassandra.new('catalog', '127.0.0.1:9160')
client.disable_node_auto_discovery!
print client.get(:catalog, "catalog1")
print "\n"
print client.get(:catalog, "catalog1", 'title')
print "\n"
print client.get(:catalog, "catalog1", :count => 3)
print "\n"
print client.get(:catalog, "catalog2", :reversed=>true)
```

Run the get.rb script with the following command:

```
ruby get.rb
```

The data requested with the get() method is output, as shown in Figure 5.13.

```
C:\Cassandra\Ruby>ruby get.rb
Unable to load cassandra_native extension. Defaulting to pure Ruby libraries.
<"author">"David A. Kelly", <"edition">"November-December 2013", <"journal">"Oracle Magazine", <"publisher">"Oracle Publishing", <"title">"Engineering as a Service"
<"author">"David A. Kelly", <"edition">"November-December 2013", <"journal">"Oracle Magazine"
<"title">"Quintessential and Collaborative", <"publisher">"Oracle Publishing", <"journal">"Oracle Magazine", <"author">"Tom Haunert"
C:\Cassandra\Ruby>
```

Figure 5.13

Getting data from Cassandra.

Source: Microsoft Corporation.

The columns are not output in the order in which they were specified in the `get()` method, which is `journal`, `publisher`, `edition`, `title`, `author`. Rather, they are output in an order determined by the comparator. The comparators supported by Cassandra are discussed in Table 5.8.

Table 5.8 Cassandra Comparators

Namespace	Description
<code>AsciiType</code>	Based on the US-ASCII bytes.
<code>BytesType</code>	Based on the lexical comparison of bytes in each column. The default.
<code>CounterColumnType</code>	Based on a 64-bit signed integer. Distributed counter type column. Counter type is discussed in Table 1.1 of Chapter 1, "Using Cassandra with Hector."
<code>IntegerType</code>	Based on generic variable-length integer values.
<code>LexicalUUIDType</code>	Based on a 128-bit UUID byte value.
<code>LongType</code>	Based on the 64-bit long values.
<code>UTF8Type</code>	Based on the UTF-8 encoded strings.

As another demonstration of using comparators, run the following Ruby script, `get_comparator.rb`, which adds columns `k1`, `k2`, `k3`, `k4`, and `k5` to five different rows. It then invokes the `get()` method to retrieve columns from the row represented with key "1" and specifies the `:start` and `:finish` options as "`k1`" and "`k5`", respectively.

```
require 'cassandra'
client = Cassandra.new('catalog', '127.0.0.1:9160')
client.disable_node_auto_discovery!
client.insert(:catalog, "1", {'k1' => 'v1', 'k2' => 'v2', 'k3' => 'v3', 'k4' => 'v4',
'k5' => 'v5'})
client.insert(:catalog, "2", {'k1' => 'v1', 'k2' => 'v2', 'k3' => 'v3', 'k4' => 'v4',
'k5' => 'v5'})
client.insert(:catalog, "3", {'k1' => 'v1', 'k2' => 'v2', 'k3' => 'v3', 'k4' => 'v4',
'k5' => 'v5'})
client.insert(:catalog, "4", {'k1' => 'v1', 'k2' => 'v2', 'k3' => 'v3', 'k4' => 'v4',
'k5' => 'v5'})
client.insert(:catalog, "5", {'k1' => 'v1', 'k2' => 'v2', 'k3' => 'v3', 'k4' => 'v4',
'k5' => 'v5'})
print client.get(:catalog, "1", :start=>"k1", :finish=>"k5")
```

Because the columns are added using the BytesType comparator, which is based on the lexical comparison of bytes in each column, the columns are added in the order k1, k2, k3, k4, k5, as shown in Figure 5.14.

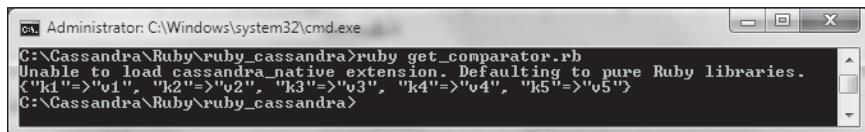


Figure 5.14

Adding columns in a lexical order.

Source: Microsoft Corporation.

SELECTING MULTIPLE Rows

In this section, you will retrieve multiple rows. For this, the Cassandra class provides the `multi_get()` method. The `multi_get()` method provides the same parameters as the `get()` method except the `key` parameter specifies an array of keys to select. The `multi_get()` method supports the same options as the `get()` method.

Create a Ruby script, `get_multi.rb`, to get multiple rows. Get rows `catalog1` and `catalog2` specified as an array from the `catalog` column family. Set the `:reversed` option to `true`. The `get_multi.rb` script appears in Listing 5.7.

Listing 5.7 The get_multi.rb Script

```
require 'cassandra'
client = Cassandra.new('catalog','127.0.0.1:9160')
client.disable_node_auto_discovery!
print client.multi_get(:catalog, ['catalog1', 'catalog2'], :reversed=>true)
```

Run the `get_multi.rb` script to return the two rows, `catalog1` and `catalog2`, as shown in Figure 5.15.

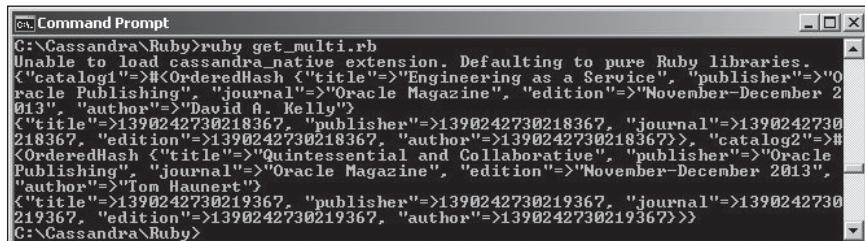


Figure 5.15

Selecting multiple rows.

Source: Microsoft Corporation.

ITERATING OVER A RESULT SET

The `Cassandra` class provides the `each_key` method to iterate through each key in the given range parameters (the `start_key` and `finish_key` parameters). The method just invokes the `Cassandra` `get_range` method. The `get_range` method parameters, column family, and options may be specified.

Create a Ruby script, `each.rb`. Invoke the `each_key` method with the `catalog` column family as the range parameter. Iterate over each key in the column family and output the key. The `each.rb` script appears in Listing 5.8. Before running the script, add some rows by running the `add_batch.rb` script.

Listing 5.8 The `each.rb` Script

```
require 'cassandra'
client = Cassandra.new('catalog', '127.0.0.1:9160')
client.disable_node_auto_discovery!
client.each_key(:catalog) do |key|
  print key
  print "\n"
end
```

Run the `each.rb` script to output the rows added, as shown in Figure 5.16.

```
C:\Administrator: C:\Windows\system32\cmd.exe
C:\Cassandra\Ruby\ruby_cassandra>ruby each.rb
Unable to load cassandra_native extension. Defaulting to pure Ruby libraries.
1
2
3
4
5
catalog1
catalog2
catalog3
catalog4
catalog5
C:\Cassandra\Ruby\ruby_cassandra>
```

Figure 5.16

Iterating over a result set.

Source: Microsoft Corporation.

SELECTING A RANGE OF ROWS

This section discusses selecting a range of rows. A range is defined with a start row and an end row. The `Cassandra` class provides the `get_range()` method to select a range of rows. The parameters supported by the method are discussed in Table 5.9.

Table 5.9 Parameters in the get_range() Method

Parameter	Description
column_family	The column family to select a range of rows
options	The options for selecting a range

The options discussed in Table 5.10 are supported by the `get_range()` method.

Table 5.10 Options supported by the get_range() Method

Option	Description
<code>:start_key</code>	The starting row for selecting a range. Supported only if <code>OrderPreservingPartitioner</code> is used.
<code>:finish_key</code>	The ending row for selecting a range. Supported only if <code>OrderPreservingPartitioner</code> is used.
<code>:key_count</code>	The total number of keys to select.
<code>:batch_size</code>	The total number of rows to select per query until all records have been selected.
<code>:columns</code>	A list of columns to return.
<code>:count</code>	The number of columns requested to be returned.
<code>:start</code>	The starting value for selecting a slice of columns.
<code>:finish</code>	The ending value for selecting a slice of columns.
<code>:reversed</code>	A Boolean indicating whether the order of columns is to be reversed. The order is based on the comparator, as discussed earlier.
<code>:consistency</code>	The read consistency.

The `get_range()` method is a wrapper around the `get_range_single()` method. If a `:batch_size` is specified, the `get_range()` method is a wrapper around the `get_range_batch()` method.

USING A RANDOM PARTITIONER

If the RandomPartitioner or the default Murmur3Partitioner is used to add rows, the rows are not added in the order specified. When the `get_range()` method is used to get rows, the rows are returned in the order added. Next, you will test the effect of the partitioner used in selecting a range of rows. Create a Ruby script, `get_range.rb`. Then obtain a range of rows using the `get_range()` method with the catalog column family as an argument.

```
require 'cassandra'
client = Cassandra.new('catalog', '127.0.0.1:9160')
client.disable_node_auto_discovery!
print client.get_range(:catalog)
```

The `get_range.rb` script returns a range of rows based on the rows added. Test the `get_range.rb` script after adding two rows with the `add.rb` script. The two rows, `catalog1` and `catalog2`, are returned, as shown in Figure 5.17.

```
C:\Cassandra\Ruby>ruby get_range.rb
Unable to load cassandra_native extension. Defaulting to pure Ruby libraries.
<"catalog1">#<OrderedHash {"author"=>"David A. Kelly", "edition"=>"November-December 2013", "journal"=>"Oracle Magazine", "publisher"=>"Oracle Publishing", "title"=>"Engineering as a Service"}
<"author"=>1390250170208910, "edition"=>1390250170208910, "journal"=>1390250170208910>, "catalog2">#<OrderedHash {"author"=>"Tom Haunert", "edition"=>"November-December 2013", "journal"=>"Oracle Magazine", "publisher"=>"Oracle Publishing", "title"=>"Quintessential and Collaborative"}
<"author"=>1390250170219911, "edition"=>1390250170219911, "journal"=>1390250170219911>>true true true true true true
C:\Cassandra\Ruby>
```

Figure 5.17

Getting a range of rows.

Source: Microsoft Corporation.

Next, run the `get_range.rb` script after running the `add_batch.rb` script. The rows `catalog1`, `catalog2`, `catalog3`, `catalog4`, and `catalog5` are returned—not in the lexical order, but in the order they were added using the Murmur3Partitioner. See Figure 5.18.

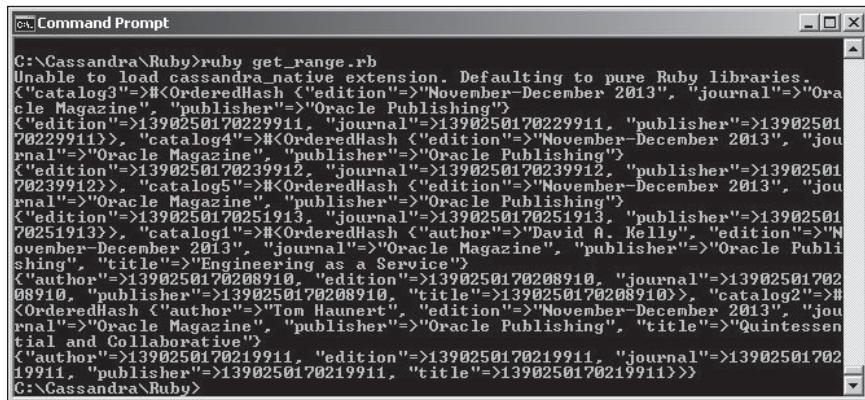


Figure 5.18

Columns added in a non-lexical order with Murmur3Partitioner.

Source: Microsoft Corporation.

The `:start_key` and `:finish_key` options cannot be used with the `get_range()` method if data has been added with a random order partitioner. For example, specify the `:start_key` and `:finish_key` as follows:

```
print client.get_range(:catalog, :start_key=>'catalog1', :finish_key=>'catalog5')
```

Because the rows are added in the order catalog3, catalog4, catalog5, catalog1, catalog2, the catalog1 key sorts after the catalog5 key. The following exception is generated:

start key's token sorts after end key's token. this is not allowed; you probably should not specify end key at all #except with an ordered partitioner
(CassandraThrift::InvalidRequestException)

The `:start_key` and `:finish_key` options may still be used with the `get_range()` method, but you must consider the order in which the rows have been added. For example, specify the `:start_key` as `catalog3` and the `:finish_key` as `catalog1`:

```
print client.get_range(:catalog, :start_key=>'catalog3', :finish_key=>'catalog1')
```

Because the `:finish_key` sorts after the `:start_key`, no exception is generated and a result is returned, as shown in Figure 5.19.

```
C:\>Cassandra>Ruby>ruby get_range.rb
Unable to load cassandra_native extension. Defaulting to pure Ruby libraries.
<"catalog3">=>#<OrderedHash <"edition">"November-December 2013", "journal">"Oracle Magazine", "publisher">"Oracle Publishing"
<"edition">=>1390246128109715, "journal">1390246128109715, "publisher">1390246128109715
<"catalog4">=>#<OrderedHash <"edition">"November-December 2013", "journal">"Oracle Magazine", "publisher">"Oracle Publishing"
<"edition">=>1390246128109715, "journal">1390246128109715, "publisher">1390246128109715
<"catalog5">=>#<OrderedHash <"edition">"November-December 2013", "journal">"Oracle Magazine", "publisher">"Oracle Publishing"
<"edition">=>1390246128109715, "journal">1390246128109715, "publisher">1390246128109715
<"catalog1">=>#<OrderedHash <"author">"David A. Kelly", "edition">"November-December 2013", "journal">"Oracle Magazine", "publisher">"Oracle Publishing", "title">"Engineering as a Service"
<"author">=>1390246128108715, "edition">1390246128108715, "journal">1390246128108715
<"author">=>1390246128108715, "publisher">1390246128108715, "title">1390246128108715>>
C:\>Cassandra>Ruby>
```

Figure 5.19

The :finish_key sorts after the :start_key.

Source: Microsoft Corporation.

As another example, specify the :start_key as catalog2:

```
print client.get_range(:catalog, :start_key=>'catalog2')
```

Only the catalog2 row is returned, because it is the last row. (See Figure 5.20.)

```
C:\>Cassandra>Ruby>ruby get_range.rb
Unable to load cassandra_native extension. Defaulting to pure Ruby libraries.
<"catalog2">=>#<OrderedHash <"author">"Tom Haunert", "edition">"November-December 2013", "journal">"Oracle Magazine", "publisher">"Oracle Publishing", "title">"Quintessential and Collaborative"
<"author">=>1390246128109715, "edition">1390246128109715, "journal">1390246128109715
<"author">=>1390246128109715, "publisher">1390246128109715, "title">1390246128109715>>
C:\>Cassandra>Ruby>
```

Figure 5.20

Getting the last row.

Source: Microsoft Corporation.

The get_range.rb script appears in Listing 5.9.

Listing 5.9 The get_range.rb Script

```
require 'cassandra'
client = Cassandra.new('catalog', '127.0.0.1:9160')
client.disable_node_auto_discovery!
client.insert(:catalog, "1", {'k1' => 'v1', 'k2' => 'v2', 'k3' => 'v3', 'k4' => 'v4',
'k5' => 'v5'})
client.insert(:catalog, "2", {'k1' => 'v1', 'k2' => 'v2', 'k3' => 'v3', 'k4' => 'v4',
'k5' => 'v5'})
client.insert(:catalog, "3", {'k1' => 'v1', 'k2' => 'v2', 'k3' => 'v3', 'k4' => 'v4',
'k5' => 'v5'})
client.insert(:catalog, "4", {'k1' => 'v1', 'k2' => 'v2', 'k3' => 'v3', 'k4' => 'v4',
'k5' => 'v5'})
```

```
client.insert(:catalog, "5", {'k1' => 'v1', 'k2' => 'v2', 'k3' => 'v3', 'k4' => 'v4',
'k5' => 'v5'})
print client.get_range(:catalog, :start_key=>"1", :finish_key=>"5")
```

USING AN ORDER-PRESERVING PARTITIONER

In this section, you will use an order-preserving partitioner. Set the partitioner to OrderPreservingPartitioner as discussed earlier. Then run the connection.rb script with the following statement:

```
print client.partitionер
```

The output indicates that the partitioner is OrderPreservingPartitioner, as shown in Figure 5.21.

```
cmd Command Prompt
C:\Cassandra\Ruby>ruby connection.rb
Unable to load cassandra_native extension. Defaulting to pure Ruby libraries.
system
["127.0.0.1:9160"]
ThriftClient
{:transport_wrapper=>Thrift::FramedTransport, :thrift_client_class=>ThriftClient
, :protocol=>Thrift::BinaryProtocolAccelerated}
[{"system", "catalog", "system_traces"}]
19.39.0
<Cassandra:27581980, @keyspace="system", @schema=<>, @servers=[ "127.0.0.1:9160" ]
>
org.apache.cassandra.dht.OrderPreservingPartitioner
C:\Cassandra\Ruby>
```

Figure 5.21

Outputting the partitioner used as OrderPreservingPartitioner.

Source: Microsoft Corporation.

Remove the previously added rows in the catalog column family as they were added, using a random-order partitioner. Re-add the rows using the add_batch.rb script. Then run the get_range.rb script, which appears in Listing 5.10.

Listing 5.10 The get_range.rb Script

```
require 'cassandra'
client = Cassandra.new('catalog', '127.0.0.1:9160')
client.disable_node_auto_discovery!
print client.get_range(:catalog)
```

The rows are returned in the order added, catalog1, catalog2, catalog3, catalog4, catalog5. (See Figure 5.22.)

Figure 5.22

Rows added with OrderPreservingPartitioner.

Source: Microsoft Corporation.

You can use the `:start_key` and `:finish_key` options to select a specific range of rows. The following statement returns the same result as the preceding:

```
print client.get_range(:catalog, :start_key=>'catalog1', :finish_key=>'catalog5')
```

As another example, set the :start_key is to catalog3:

```
print client.get_range(:catalog, :start_key=>'catalog3')
```

Rows catalog3, catalog4, and catalog5 are returned with the order preserved, as shown in Figure 5.23.

```
C:\Cassandra\Ruby>ruby get_range.rb
Unable to load cassandra_native extension. Defaulting to pure Ruby libraries.
[{"catalog3": {"edition": "November-December 2013", "journal": "Oracle Magazine", "publisher": "Oracle Publishing"}, {"edition": 1390255893530266, "journal": >1390255893530266, "publisher": >1390255893530266}, {"catalog4": {"edition": "November-December 2013", "journal": "Oracle Magazine", "publisher": "Oracle Publishing"}, {"edition": 1390255893530266, "journal": >1390255893530266, "publisher": >1390255893530266}, {"catalog5": {"edition": "November-December 2013", "journal": "Oracle Magazine", "publisher": "Oracle Publishing"}, {"edition": 1390255893530266, "journal": >1390255893530266, "publisher": >1390255893530266}]}
C:\Cassandra\Ruby>
```

Figure 5.23

Adding a range of rows with OrderPreservingPartitioner.

Source: Microsoft Corporation.

GETTING A SLICE OF COLUMNS

The `Cassandra` class provides the methods discussed in Table 5.11 to get a slice of columns.

Table 5.11 Cassandra Class Methods to Get a Slice of Columns

Method	Description
<code>get_columns(column_family, key, *columns_and_options)</code>	Returns a list of columns from the specified column family and specified row. You can specify the specific columns to get with the <code>columns_and_options</code> parameter, which is optional. By default, all columns are returned.
<code>multi_get_columns(column_family, keys, *options)</code>	Returns a hash of columns for the specified keys from the specified column family. The options are specified using the <code>options</code> parameter.

Create a Ruby script, `get_columns.rb`, and invoke the `get_columns()` method to return the `title`, `journal`, and `author` columns from the `catalog1` row. Then invoke the `multi_get_columns()` method to return the `title`, `journal`, and `author` columns from the `catalog1` and `catalog2` rows. The `get_columns.rb` script appears in Listing 5.11.

Listing 5.11 The `get_columns.rb` Script

```
require 'cassandra'
client = Cassandra.new('catalog', '127.0.0.1:9160')
client.disable_node_auto_discovery!
print client.get_columns(:catalog, "catalog1", ["title", "journal", "author"])
print "\n"
print client.multi_get_columns(:catalog, ["catalog1", "catalog2"], ["title", "journal", "author"])
```

Run the `get_columns.rb` script to return the slices of columns, as shown in Figure 5.24.

```
C:\Cassandra\Ruby>ruby get_columns.rb
Unable to load cassandra_native extension. Defaulting to pure Ruby libraries.
[{"catalog1":>["Engineering as a Service", "Oracle Magazine", "David A. Kelly"]},
 {"catalog2":>["Quintessential and Collaborative", "Oracle Magazine", "Tom Haunert"]}
]
C:\Cassandra\Ruby>
```

Figure 5.24

Getting a slice of columns.

Source: Microsoft Corporation.

UPDATING DATA IN A TABLE

Create a Ruby script, update.rb, to update data in Cassandra. The `insert` method, which is used to add data, is also used to update data. Make slight modifications to `catalog1` and `catalog2` rows as listed below.

```
require 'cassandra'
client = Cassandra.new('catalog', '127.0.0.1:9160')
client.disable_node_auto_discovery!
print client.insert(:catalog, "catalog1", {'journal' => 'Oracle-Magazine',
'publisher' => 'Oracle-Publishing', 'edition' => 'November-December 2013', 'title' => 'Engineering as a Service', 'author' => 'Kelly, David'})
print client.insert(:catalog, "catalog2", {'journal' => 'Oracle-Magazine',
'publisher' => 'Oracle-Publishing', 'edition' => 'November-December 2013', 'title' => 'Quintessential-and-Collaborative', 'author' => 'Tom Haunert'})
```

Run the `update.rb` script. Then run the `get.rb` script to return the result shown in Figure 5.25.

```
C:\Cassandra\Ruby>ruby update.rb
Unable to load cassandra_native extension. Defaulting to pure Ruby libraries.
C:\Cassandra\Ruby>ruby get.rb
Unable to load cassandra_native extension. Defaulting to pure Ruby libraries.
[{"author":>"Kelly, David", "edition":>"November-December 2013", "journal":>"Oracle-Magazine", "publisher":>"Oracle-Publishing", "title":>"Engineering as a Service"},
 {"author":>"Kelly, David", "edition":>"November-December 2013", "journal":>"Oracle-Magazine"}, {"title":>"Quintessential-and-Collaborative", "publisher":>"Oracle-Publishing", "journal":>"Oracle Magazine", "edition":>"November-December 2013", "author":>"Tom Haunert"}
]
C:\Cassandra\Ruby>
```

Figure 5.25

Updating data.

Source: Microsoft Corporation.

DELETING DATA IN A TABLE

The Cassandra class provides the `remove()` method to remove data. The `remove()` method takes the parameters discussed in Table 5.12.

Table 5.12 Parameters in the Cassandra Class's `remove()` Method

Parameter	Description
<code>column_family</code>	The column family.
<code>key</code>	The row key.
<code>columns</code>	The list of columns.
<code>options</code>	The options.
<code>sub_columns</code>	The sub columns. The super/sub columns are not discussed in the book.

The supported options are discussed in Table 5.13.

Table 5.13 Options Supported by the `remove()` Method

Option	Description
<code>:timestamp</code>	The timestamp or the current time by default
<code>:consistency</code>	The read consistency

Create a Ruby script, `remove.rb`, to delete data in the Cassandra database. Then remove the rows `catalog1`, `catalog2`, `catalog3`, `catalog4`, and `catalog5` by invoking the `remove()` method. The `remove.rb` script appears in Listing 5.12.

Listing 5.12 The `remove.rb` Script

```
require 'cassandra'
client = Cassandra.new('catalog', '127.0.0.1:9160')
client.disable_node_auto_discovery!
print client.remove(:catalog, 'catalog1')
print client.remove(:catalog, 'catalog2')
print client.remove(:catalog, 'catalog3')
print client.remove(:catalog, 'catalog4')
print client.remove(:catalog, 'catalog5')
```

Run the script with the following command:

```
ruby remove.rb
```

The five rows of data are removed.

UPDATING A COLUMN FAMILY

The `Cassandra` class provides the `update_column_family(cf_def)` method to update a column family. Create a Ruby script, `updateCF.rb`, to update a column family. Then create a new column family definition with some of the parameters set to non-default value. For example, set the `:max_compaction_threshold` to 16 to replace the default 32. Next, set `:replicate_on_write` to `false` to replace the default `true`. Invoke the `update_column_family()` method to update the column family, and then print all the column families in the catalog keyspace.

```
client.update_column_family(cf_def)
print client.column_families
```

The `updateCF.rb` script appears in Listing 5.13.

Listing 5.13 The `updateCF.rb` Script

```
require 'cassandra'
cf_def = Cassandra::ColumnFamily.new(:keyspace => "catalog", :name => "catalog",
:max_compaction_threshold=>16, :replicate_on_write=>false)
client = Cassandra.new('catalog','127.0.0.1:9160')
client.disable_node_auto_discovery!
cf_def = Cassandra::ColumnFamily.new(:keyspace => "catalog", :name => "catalog",
:max_compaction_threshold=>16, :replicate_on_write=>false)
client.update_column_family(cf_def)
print client.column_families
```

Run the `updateCF.rb` script to update the column family and output the updated column family, as shown in Figure 5.26.

```
C:\Cassandra\Ruby>ruby updateCF.rb
Unable to load cassandra_native extension. Defaulting to pure Ruby libraries.
<"catalog">=>CassandraThrift::CfDef keyspace:"catalog", name:"catalog", column_type:"Standard", comparator_type:"org.apache.cassandra.db.marshal.BytesType", comment:"", row_cache_size:0, key_cache_size:20000, read_repair_chance:1.0, column_metadata:[], gc_grace_seconds:864000, default_validation_class:"org.apache.cassandra.db.marshal.BytesType", min_compaction_threshold:4, max_compaction_threshold:16, replicate_on_write:false, key_validation_class:"org.apache.cassandra.db.marshal.BytesType", row_cache_provider:"org.apache.cassandra.cache.ConcurrentLinkedHashCacheProvider", key_alias:6b6579>
C:\Cassandra\Ruby>
```

Figure 5.26

Updating the column family.

Source: Microsoft Corporation.

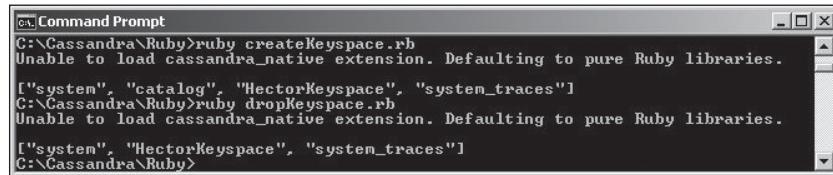
DROPPING A KEYSPACE

The `Cassandra` class provides the `drop_keyspace(String keyspace)` method to drop a keyspace. Create a Ruby script, `dropKeyspace.rb`, to drop the `catalog` keyspace. Also include a `print` statement for the `keyspaces` attribute after dropping the keyspace. The `dropKeyspace.rb` script appears in Listing 5.14.

Listing 5.14 The `dropKeyspace.rb` Script

```
require 'cassandra'
client = Cassandra.new('system', '127.0.0.1:9160')
client.disable_node_auto_discovery!
client.drop_keyspace('catalog')
print "\n"
print client.keyspaces
```

Run the script to drop the `catalog` keyspace. The `catalog` keyspace is not listed, as shown in Figure 5.27.



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The window contains the following text:

```
C:\Cassandra\Ruby>ruby createKeyspace.rb
Unable to load cassandra_native extension. Defaulting to pure Ruby libraries.
[{"system", "catalog", "HectorKeyspace", "system_traces"}]
C:\Cassandra\Ruby>ruby dropKeyspace.rb
Unable to load cassandra_native extension. Defaulting to pure Ruby libraries.
[{"system", "HectorKeyspace", "system_traces"}]
```

Figure 5.27

Dropping a keyspace.

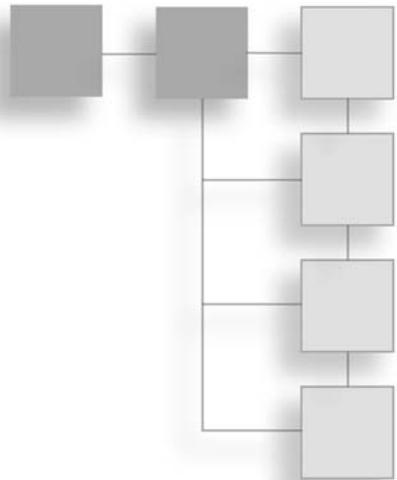
Source: Microsoft Corporation.

SUMMARY

This chapter discussed using the Ruby client for Cassandra to add, get, update, and remove data, including adding, updating, and removing keyspaces and column families. In the next chapter, you will learn how to use Node.js with Cassandra. Node.js is a light-weight, efficient platform.

CHAPTER 6

USING NODE.JS WITH CASSANDRA



The client/server paradigm is the most commonly used paradigm in Web applications. Typically, however, a client/server application requires an application/Web server. Node.js is a lightweight, efficient platform based on the event-driven model and built on Chrome's JavaScript runtime for developing fast, scalable, data-intensive, real-time, network applications. Node.js is suitable for the cloud environment because Node.js applications can run on distributed devices. This chapter discusses accessing Cassandra with Node.js and making data modifications in the database using the Node.js driver for Cassandra.

OVERVIEW OF NODE.JS DRIVER FOR CASSANDRA CQL

The Node.js driver for Cassandra is a JavaScript-based library for accessing Cassandra. It provides several features such as connection pooling, load balancing, automatic failover, and support for prepared statements and query batches. The Node.js driver for Cassandra provides two classes, Client and Connection, as illustrated in Figure 6.1.

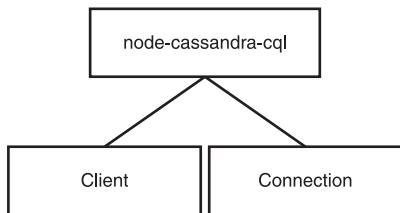


Figure 6.1
Classes in the Node.js driver for Cassandra.

The classes are discussed in Table 6.1.

Table 6.1 Classes in Node.js

Class	Description
Client	A Client instance provides a connection pool to a Cassandra cluster without requiring the explicit opening and closing of connections. The Client class is the preferred interface for connection to Cassandra.
Connection	The Connection class provides a low-level, fine-grained connection to a Cassandra node. The disadvantage of Connection is that a connection is required to be opened and closed explicitly.

The Client Class

A Client class function may be created with the class constructor using new Client(options). The options discussed in Table 6.2 are supported.

Table 6.2 Client Class Options

Option	Description
hosts	The hosts on Cassandra, presented as an array of strings in host:port format. The port specification is optional and defaults to 9042. The only required option.
keyspace	The keyspace name.
username	The username for authentication.
password	The password for authentication.
staleTime	The time after which a connection to a node is retried.
maxExecuteRetries	The maximum number of times an execute can be retried. Connection to another node is used if a node becomes unavailable.
getAConnectionTimeout	The maximum time to wait for a connection from a connection pool, in milliseconds.
poolSize	The number of connections in the connection pool for each host. The default is 1.

The Client class provides several methods, as discussed in Table 6.3.

Table 6.3 Client Class Methods

Method	Description
connect([callback])	Connects the pool if not already connected, as by default the connect() method is called internally when a query is run. The optional callback function is invoked after a connection is established. If a connection already exists, the callback parameter is called instantly.
execute(query, [params], [consistency], callback)	Executes a CQL query. The params are parameters for the ? placeholders. consistency defaults to quorum, which is a strong consistency with some tolerance for failure. The callback takes two args: err and result.
executeAsPrepared(query, [params], [consistency], callback)	Executes a prepared statement. The first time the method is invoked, the query is prepared and run. If the same query is subsequently run again, the query is not prepared a second time. Rather, the prepared query is run. params is the parameters for the placeholders in the query. The default consistency is quorum. The callback takes two args: err and result.
executeBatch(queries, [consistency], [options], callback)	Executes a batch of queries. Other method parameters are the same as the preceding method executeAsPrepared().
eachRow(query, [params], [consistency], rowCallback, endCallback)	Prepares and runs a query similarly to executeAsPrepared(). rowCallback(n, row) is the callback function after each row is received, with n being the row index. The row object contains the definition of the row columns. endCallback(err, rowLength) is run when all rows have been received or when there has been an error getting a row.
streamField(query, [params], [consistency], rowCallback, [endCallback])	Prepares and runs a query similarly to eachRow(). Streams the last field of each row. rowCallback(n, row, streamField) is invoked for each row after the first chunk of the last field is received. The row object contains the definition of the row columns except the last column. streamField is a Readable Streams2 object. endCallback(err, rowLength) is run when all rows have been received or when there has been an error getting a row.

(Continued)

Table 6.3 Client Class Methods (*Continued*)

Method	Description
<code>stream(query, [params], [consistency], [callback])</code>	Prepares and runs a query similarly to <code>streamField()</code> except that the whole row is streamed as a Readable Streams2 object. When a row can be read from a stream, a readable event is emitted. <code>callback(err)</code> is invoked when all rows have been received or when there has been an error getting a row.
<code>shutdown([callback])</code>	Closes all connections in the pool and closes the pool. The optional <code>callback</code> parameter is invoked when the pool is disconnected.

The Connection Class

An instance of the `Connection` class can be created with the class constructor using `new Connection(options)`. The options are the same as for the `Client` class. The `Connection` class provides the methods discussed in Table 6.4.

Table 6.4 Connection Class Options

Option	Description
<code>open(callback)</code>	Opens a connection and authenticates and sets a keyspace. The optional <code>callback</code> function is invoked after a connection is established.
<code>close(callback)</code>	Closes a connection. The optional <code>callback</code> function is invoked after a connection is closed.
<code>execute(query, args, consistency, callback)</code>	Executes a CQL query. The <code>args</code> are parameters for the <code>?</code> placeholders. <code>consistency</code> defaults to <code>quorum</code> , which is a strong consistency with some tolerance for failure.
<code>prepare(query, callback)</code>	Prepares a CQL query with an optional <code>callback</code> . Does not run the query.
<code>executePrepared(queryId, args, consistency, callback)</code>	Executes a previously prepared query. A <code>queryId</code> identifies a query. The <code>args</code> are parameters for the <code>?</code> placeholders.

EVENT-DRIVEN LOGGING

Node.js is event-driven and provides the `EventEmitter` class in the `events` module for emitting events. An example of using `EventEmitter` would be to first import the `events` module using `require()`. Subsequently, an instance of `EventEmitter` may be created.

```
var events=require('events');
var eventEmitter=new events.EventEmitter();
var logEvent= function logEvent(){
  console.log('A logging event occurred');
}
eventEmitter.on('log', logEvent);
eventEmitter.emit('log');
```

You don't need to create `EventEmitter` instances because `Client` and `Connection` classes are instances of `EventEmitter`. The `Client` and `Connection` classes emit the `log` event for logging. The function that is invoked when a `log` event occurs may be defined as follows:

```
var logEvent= function logEvent(level, message){
  console.log('log event: %s -- %j', level, message);
}
```

Register the `log` event using the `on()` method and emit the `log` event using the `emit` method:

```
client.on('log', logEvent);
client.emit('log');
```

The log level can be `info` or `error`.

MAPPING DATA TYPES

The Node.js driver for Cassandra provides mapping of JavaScript types to Cassandra data types, and all Cassandra data types are supported. The mapping from Cassandra data types to JavaScript data types is discussed in Table 6.5.

Table 6.5 Mapping Cassandra Data Types to JavaScript Data Types

Cassandra Data Type	JavaScript Data Type
Bigint	Long
List/Set	Array

(Continued)

Table 6.5 Mapping Cassandra Data Types to JavaScript Data Types (*Continued*)

Cassandra Data Type	JavaScript Data Type
Map	Object with keys as properties
Timestamp	Date
Decimal and Varint	Not parsed, yielded as byte buffers

The mapping from JavaScript data types to Cassandra data types is discussed in Table 6.6.

Table 6.6 Mapping JavaScript Data Types to Cassandra Data Types

JavaScript Data Type	Cassandra Data Type
string	text
Date	timestamp
Number	int
Long	bigint
Array	list
Buffers	blob

SETTING THE ENVIRONMENT

The Node.js driver for Cassandra does not provide an API for creating a keyspace and a column family. You will create a keyspace and a column family in Cassandra-Cli. In this section, you will also install Node.js and the Node.js driver for Cassandra.

The following software is required for this chapter:

- Apache Cassandra 2.04 or later
- Node.js
- Node.js driver for Apache Cassandra

Follow these steps:

1. Download Apache Cassandra apache-cassandra-2.0.4-bin.tar.gz (or later version) from <http://cassandra.apache.org/download/>.

2. Extract the tar.gz file to a directory.
3. Add the bin directory from the Apache Cassandra installation to the PATH environment variable.

Creating a Keyspace and a Column Family

To create a keyspace called `catalog_nodejs` with a replica `placement_strategy` of `SimpleStrategy` and a `replication_factor` of 1, run the following command in Cassandra-Cli:

```
CREATE KEYSPACE catalog_nodejs
with placement_strategy = 'org.apache.cassandra.locator.SimpleStrategy'
and strategy_options = {replication_factor:1};
```

The output in Cassandra-Cli indicates that the keyspace has been created, as shown in Figure 6.2.

```
C:\>cassandra-cli
Starting Cassandra Client
Connected to: "Test Cluster" on 127.0.0.1/9160
Welcome to Cassandra CLI version 2.0.4

The CLI is deprecated and will be removed in Cassandra 3.0. Consider migrating
to cqlsh.
CQL is fully backwards compatible with Thrift data; see http://www.datastax.com/
dev/blog/thrift-to-cql3

Type 'help;' or '??' for help.
Type 'quit;' or 'exit;' to quit.

[default@unknown] [default@unknown] CREATE KEYSPACE catalog_nodejs
...     with placement_strategy = 'org.apache.cassandra.locator.SimpleStrategy'
...     and strategy_options = {replication_factor:1};
c1042193-2d2d-3b3f-98c5-a02b4006daf4
[default@unknown] -
```

Figure 6.2

Creating a keyspace.

Source: Microsoft Corporation.

Next, run the following command in Cassandra-Cli to use the `catalog_nodejs` keyspace:

```
USE catalog_nodejs;
```

Create a column family called `nodejscatalog` with a `UTF8Type` Comparator and a `UTF8Type` key validation class. The column family definition must include a column called `key`, which is the primary key of the table. Also define columns named `journal`, `publisher`, `edition`, `title`, and `author`, all of type `UTF8Type`. The `title` column is indexed.

```
CREATE COLUMN FAMILY nodejscatalog
WITH comparator = UTF8Type
```

```

AND key_validation_class=UTF8Type
AND column_metadata = [
{column_name: key, validation_class: UTF8Type},
{column_name: journal, validation_class: UTF8Type},
    {column_name: publisher, validation_class: UTF8Type},
    {column_name: edition, validation_class: UTF8Type},
    {column_name: title, validation_class: UTF8Type, index_type: KEYS},
    {column_name: author, validation_class: UTF8Type}
];

```

A column family called nodejscatalog is created, as shown in Figure 6.3.

```

c:\ Command Prompt - cassandra-cli
[default@cassandra-nodejs]
[default@cassandra-nodejs] CREATE COLUMN FAMILY nodejscatalog
...   WITH comparator = UTF8Type
...   AND key_validation_class=UTF8Type
...   AND column_metadata = [
...     {column_name: key, validation_class: UTF8Type},
...     {column_name: journal, validation_class: UTF8Type},
...       {column_name: publisher, validation_class: UTF8Type},
...       {column_name: edition, validation_class: UTF8Type},
...       {column_name: title, validation_class: UTF8Type, index_type: KEYS},
...       {column_name: author, validation_class: UTF8Type}
... ];
... b3dd3696-2f08-3733-89d7-d1d7c9045dfa
[default@cassandra-nodejs]

```

Figure 6.3

Creating a column family.

Source: Microsoft Corporation.

Installing Node.js

To install Node.js, follow these steps:

1. Download the node-v0.10.26-x64.exe application from <http://Node.js.org/>.
2. Double-click the EXE application to install Node.js. The Node.js version may be found with the node –version command.
3. To test that Node.js has been installed, run the sample Node.js script provided on <http://Node.js.org/>. The sample script creates a Node.js server and responds with a “Hello” message for every request. Copy the following script to example.js:

```

var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, '127.0.0.1');
console.log('Server running at http://127.0.0.1:1337/');

```

4. To run the server, run the example.js script with the following command:

```
node example.js
```

The message “Server running at http://127.0.0.1:1337/” indicates that the server is running, as shown in Figure 6.4.

```
C:\ Command Prompt - node example.js
C:\Cassandra\NodeJS>node --version
v0.10.24
C:\Cassandra\NodeJS>node example.js
Server running at http://127.0.0.1:1337/
```

Figure 6.4

Running the example.js Node.js script.

Source: Microsoft Corporation.

Installing Node.js driver for Apache Cassandra

To install the Node.js driver for Cassandra, run the following command:

```
npm install node-cassandra-cql
```

Node.js driver for Cassandra is installed, as shown in Figure 6.5.

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Deepak Uohra>cd C:\Cassandra\NodeJS

C:\Cassandra\NodeJS>npm install node-cassandra-cql
npm http GET https://registry.npmjs.org/node-cassandra-cql
npm http 200 https://registry.npmjs.org/node-cassandra-cql
npm http GET https://registry.npmjs.org/node-cassandra-cql/-/node-cassandra-cql-0.4.3.tgz
npm http 200 https://registry.npmjs.org/node-cassandra-cql/-/node-cassandra-cql-0.4.3.tgz
npm http GET https://registry.npmjs.org/long
npm http GET https://registry.npmjs.org/node-uuid/1.4.0
npm http GET https://registry.npmjs.org/async
npm http 200 https://registry.npmjs.org/node-uuid/1.4.0
npm http 200 https://registry.npmjs.org/long
npm http GET https://registry.npmjs.org/node-uuid/-/node-uuid-1.4.0.tgz
npm http GET https://registry.npmjs.org/long/-/long-1.1.2.tgz
npm http 200 https://registry.npmjs.org/node-uuid/-/node-uuid-1.4.0.tgz
npm http 200 https://registry.npmjs.org/long/-/long-1.1.2.tgz
npm http 200 https://registry.npmjs.org/async
node-cassandra-cql@0.4.3 node_modules\node-cassandra-cql
|   node-uuid@1.4.0
|   async@0.2.9
|   long@1.1.2
C:\Cassandra\NodeJS>
```

Figure 6.5

Installing Node.js Driver for Cassandra.

Source: Microsoft Corporation.

You also need to start Apache Cassandra with the following command:

```
cassandra -f
```

CREATING A CONNECTION WITH CASSANDRA

As discussed, the `Client` and `Connection` classes are used to connect to Cassandra. The `Client` class is preferred because it provides a connection pool without the need to explicitly open a connection to a node. Create a JavaScript file, `connection-cassandra.js`, for connecting to Cassandra. Import the `Node.js` driver for Cassandra using the following statement:

```
var cql = require('node-cassandra-cql');
```

Create an instance of the `Client` class using the class constructor with the `hosts:` option set to `localhost:9042` and the `keyspace` option set to `catalog_Node.js`:

```
var client = new cql.Client({hosts: ['localhost:9042'], keyspace: 'catalog_Node.js'});
```

Although a `Client` connection pool automatically connects to a Cassandra cluster when a query is run, a connection may be made explicitly using the `connect(callback)` method. The `callback` function takes an `err` parameter and may be defined as follows to log an error message if an error is generated or to output a message if an error is not generated:

```
function established(err){
    if (err)
        console.log(err);
    else
        console.log('Connection with Cassandra established');
}
```

The other option for creating a connection is the `Connection` class. Create an instance of the `Connection` class with the same two options as arguments, `hosts` and `keyspace`, as for the `Client` class example:

```
var client2 = new cql.Connection({hosts: ['localhost:9042'], keyspace: 'catalog_Node.js'});
```

The `Connection` class's similarity with the `Client` class ends with the constructor use. While a `Client` instance is connected to Cassandra without requiring an explicit connection, the `Connection` class requires an explicit connection using the `open(callback)` method.

```
client2.open(function established(err){
    if (err)
        console.log(err);
    else
        console.log('Connection with Cassandra established');
});
```

The connection-cassandra.js script appears in Listing 6.1.

Listing 6.1 The connection-cassandra.js Script

```
var cql = require('node-cassandra-cql');
var client = new cql.Client({hosts: ['localhost:9042'], keyspace:
  'catalog_nodejs'});
client.connect(function established(err){if (err) console.log(err);
else console.log('Connection with Cassandra established');});
var client2 = new cql.Connection({hosts: ['localhost:9042'], keyspace:
  'catalog'});
client2.open(function established(err){if (err) console.log(err);
else console.log('Connection with Cassandra established');});
```

Run the connection-cassandra.js script with the following command:

```
node connection-cassandra.js
```

Output from both the Client and the Connection classes is the message to indicate that a connection has been established, as shown in Figure 6.6.

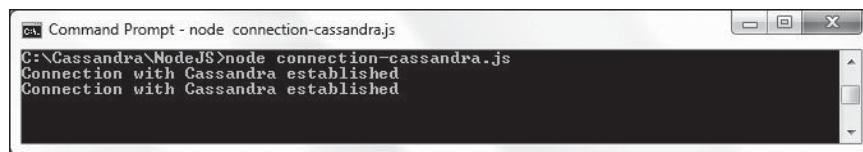


Figure 6.6

Establishing a connection with the Client and Connection class.

Source: Microsoft Corporation.

The Cassandra database must be running to be able to connect to it. If the database is not running, the following error is generated:

```
{ name: 'PoolConnectionError',
  info: 'Represents a error while trying to connect the pool, all the
connections failed.',
  individualErrors:
  [ { [Error: connect ECONNREFUSED]
    code: 'ECONNREFUSED',
    errno: 'ECONNREFUSED',
    syscall: 'connect' } ] }
{ name: 'PoolConnectionError',
  info: 'Represents a error while trying to connect the pool, all the
connections failed.',
```

```
individualErrors:
[ { [Error: connect ECONNREFUSED]
  code: 'ECONNREFUSED',
  errno: 'ECONNREFUSED',
  syscall: 'connect' } ] }
```

ADDING DATA TO A TABLE

In this section, you will add two rows of data to the `nodejscatalog` column family (table). Create a JavaScript script, `add.js`. Create a `Client` instance to establish a connection pool with Cassandra.

```
var cql = require('node-cassandra-cql');
var client = new cql.Client({hosts: ['localhost:9042'], keyspace:
  'catalog_nodejs'});
```

Invoke the `execute(query, [params], [consistency], callback)` method to run a CQL3 query to add a row. The CQL query statement is an `INSERT` statement. `consistency` is specified as `cql.types.consistencies.quorum`, which is also the default. `callback` takes an `err` parameter to log an error message if an error message is generated or, if not, to log a message to indicate that a table row has been added.

```
client.execute("INSERT INTO nodejscatalog (key, journal, publisher, edition,
title,author) VALUES ('catalog1','Oracle Magazine', 'Oracle Publishing',
'November-December 2013', 'Engineering as a Service','David A. Kelly')",
  cql.types.consistencies.quorum,
  function(err) {
    if (err) console.log(err);
    else console.log('table row added');
  }
);
```

Similarly, add a second row. The `add.js` script appears in Listing 6.2.

Listing 6.2 The add.js Script

```
var cql = require('node-cassandra-cql');
var client = new cql.Client({hosts: ['localhost:9042'], keyspace:
  'catalog_nodejs'});
client.execute("INSERT INTO nodejscatalog (key, journal, publisher, edition,
title,author) VALUES ('catalog1','Oracle Magazine', 'Oracle Publishing',
'November-December 2013', 'Engineering as a Service','David A. Kelly')",
  cql.types.consistencies.quorum,
```

```
function(err) {
  if (err) console.log(err);
  else console.log('table row added');
}
);
client.execute("INSERT INTO nodejscatalog (key, journal, publisher, edition,
title,author) VALUES ('catalog2','Oracle Magazine', 'Oracle Publishing',
'November-December 2013', 'Quintessential and Collaborative','Tom Haunert')",
cql.types.consistencies.quorum,
function(err) {
  if (err) console.log(err);
  else console.log('table row added');
}
);
```

Run the add.js script with the following command:

```
node add.js
```

The output message indicates that a table row has been added, as shown in Figure 6.7.



Figure 6.7

Adding a table row.

Source: Microsoft Corporation.

You provided a column value for all the columns in the column family, but the flexible schema supported by Cassandra does not require each column value in a row except that the key column value is required. For example, the following CQL query would add a two-column row excluding the key column, which is required:

```
client.execute("INSERT INTO nodejscatalog (title,author) VALUES
('catalog1','Engineering as a Service','David A. Kelly'),
function(err) {
  if (err) console.log(err);
  else console.log('table row added');
}
);
```

A column called key is required to add a row with the Node.js driver for Cassandra. If the key column is not provided, the following error message is generated:

```
{ name: 'ResponseError',
  message: 'Missing mandatory PRIMARY KEY part key',
  info: 'Represents a error message from the server',
  code: 8704,
  isServerUnhealthy: false,
  query: 'INSERT INTO nodejscatalog (journal, publisher, edition,title,author)
VALUES (\'Oracle Magazine\', \'Oracle Publishing\', \'November-December 2013\' ,
\'Engineering as a Service\', \'David A. Kelly\')'
}
```

RETRIEVING DATA FROM A TABLE

Next, you will retrieve data from Cassandra. To do so, create a script, get.js. Then create a Client instance as before. Run a SELECT CQL query to get a result set. The consistency and callback functions may be defined as before or omitted. The get.js script appears in Listing 6.3 below.

Listing 6.3 The get.js Script

```
var cql = require('node-cassandra-cql');
var client = new cql.Client({hosts: ['localhost:9042']}, keyspace:
  'catalog_Node.js');
client.execute("SELECT key, journal, publisher, edition,title,author FROM
Node.jscatalog",
  cql.types.consistencies.quorum,
  function(err, result) {
    if (err) console.log(err);
    else console.log(result);
  }
);
```

Run the get.js script with the following command:

```
node get.js
```

The two rows in the nodejscatalog table are retrieved, as shown in Figure 6.8.

```
C:\>Cassandra>node get.js
{
  rows: [
    {
      columns: [Object],
      key: 'catalog1',
      journal: 'Oracle Magazine',
      publisher: 'Oracle Publishing',
      edition: 'November-December 2013',
      title: 'Engineering as a Service',
      author: 'David A. Kelly'
    },
    {
      columns: [Object],
      key: 'catalog2',
      journal: 'Oracle Magazine',
      publisher: 'Oracle Publishing',
      edition: 'November-December 2013',
      title: 'Quintessential and Collaborative',
      author: 'Tom Haunert'
    }
  ],
  meta: {
    global_tables_spec: true,
    keyspace: 'catalog_nodejs',
    table: 'nodejscatalog',
    columns: [
      [Object],
      [Object],
      [Object],
      [Object],
      [Object],
      [Object],
      [Object],
      _col_key: 0,
      _col_journal: 1,
      _col_publisher: 2,
      _col_edition: 3,
      _col_title: 4,
      _col_author: 5
    ]
  }
}
```

Figure 6.8

Getting data.

Source: Microsoft Corporation.

FILTERING THE QUERY

In the preceding section, you selected all the rows from the `nodejscatalog` column family. If necessary, however, you can filter rows using the `WHERE` clause. Create a JavaScript file, `getfilter.js`. Create a connection to Cassandra using a `Client` class instance. Run a CQL query using the `execute(query, [params], [consistency], callback)` method and include a `WHERE` clause to select row with the key '`'catalog2'`'. The `callback` function takes two args, `err` and `result`. If an error is generated, log the error to the console. If an error is not generated, access the `rows` returned in the `result` and output the `journal` column value in the first row in the `result` as follows:

```
result.rows[0].journal
```

The `getfilter.js` script appears in Listing 6.4.

Listing 6.4 The getfilter.js Script

```
var cql = require('node-cassandra-cql');
var client = new cql.Client({hosts: ['localhost:9042'], keyspace:
  'catalog_nodejs'});
client.execute("SELECT key, journal, publisher, edition, title, author FROM
nodejscatalog WHERE key=?", ['catalog2'],
  cql.types.consistencies.quorum,
  function(err, result) {
    if (err) console.log(err);
    else {console.log('key: ' +result.rows[0].key);
    console.log(' journal: ' +result.rows[0].journal);
    console.log(' publisher: ' +result.rows[0].publisher);
    console.log(' edition: ' +result.rows[0].edition);
    console.log(' title: ' +result.rows[0].title);
    console.log(' author: ' +result.rows[0].author);
  }
}
);
```

Run the getfilter.js script with the following command:

```
node getfilter.js
```

The row with key catalog2 is retrieved and output in the console, as shown in Figure 6.9.

```
C:\Cassandra\NodeJS>node getfilter.js
key: catalog2
journal: Oracle Magazine
publisher: Oracle Publishing
edition: November-December 2013
title: Quintessential and Collaborative
author: Tom Haunert
```

Figure 6.9

Getting the row with the key catalog2.

Source: Microsoft Corporation.

QUERYING WITH A PREPARED STATEMENT

A prepared statement is a CQL query with placeholders using ?. When the prepared statement is run, parameter values are provided to substitute the placeholders. The executeAsPrepared() method in the Client class is used to run a prepared statement. The first time a CQL query with placeholders (?) is run, the CQL query is prepared, and a prepared statement is generated for subsequent use to run the same query multiple times, if required. The advantage of using a prepared statement consisting of placeholders is that the CQL query does not have to be compiled each time the query is run.

Create a JavaScript file, preparedquery.js, to run a prepared statement. Create a Client instance for a connection with the Cassandra database. Invoke the executeAsPrepared() method with the prepared statement query as SELECT key, journal, publisher, edition, title, author FROM nodejscatalog WHERE key=? . This has a placeholder for the key column. In the first invocation of the executeAsPrepared() method, supply the parameter as 'catalog2'. In the second invocation, supply the parameter as 'catalog1'. Output the result as in the preceding section. The preparedquery.js script appears in Listing 6.5.

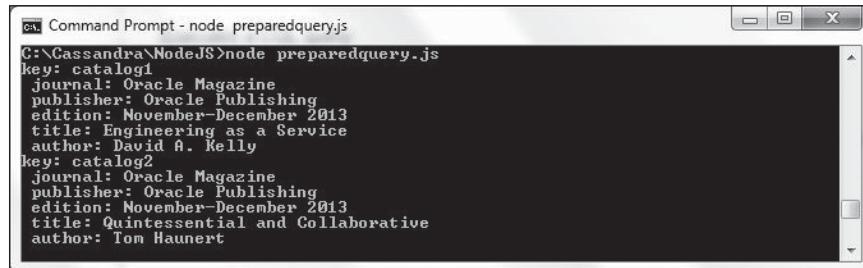
Listing 6.5 The preparedquery.js Script

```
var cql = require('node-cassandra-cql');
var client = new cql.Client({hosts: ['localhost:9042'], keyspace:
  'catalog_nodejs'});
client.executeAsPrepared("SELECT key, journal, publisher, edition, title, author
FROM nodejscatalog WHERE key=?", ['catalog2'],
  cql.types.consistencies.quorum,
  function(err, result) {
    if (err) console.log(err);
    else {console.log('key: ' +result.rows[0].key);
  console.log(' journal: ' +result.rows[0].journal);
  console.log(' publisher: ' +result.rows[0].publisher);
  console.log(' edition: ' +result.rows[0].edition);
  console.log(' title: ' +result.rows[0].title);
  console.log(' author: ' +result.rows[0].author);
}
  });
client.executeAsPrepared("SELECT key, journal, publisher, edition, title, author
FROM nodejscatalog WHERE key=?", ['catalog1'],
  cql.types.consistencies.quorum,
  function(err, result) {
    if (err) console.log(err);
    else {console.log('key: ' +result.rows[0].key);
  console.log(' journal: ' +result.rows[0].journal);
  console.log(' publisher: ' +result.rows[0].publisher);
  console.log(' edition: ' +result.rows[0].edition);
  console.log(' title: ' +result.rows[0].title);
  console.log(' author: ' +result.rows[0].author);
}
  });
);
```

Run the preparedquery.js script with the following command:

```
node preparedquery.js
```

The rows catalog1 and catalog2 are output in the console, as shown in Figure 6.10.



The screenshot shows a Windows Command Prompt window titled "Command Prompt - node preparedquery.js". The command entered is "C:\Cassandra\NodeJS>node preparedquery.js". The output displays two rows of data, each representing a book entry from a Cassandra database. The first row is for "catalog1" and the second for "catalog2". Each row includes fields such as key, journal, publisher, edition, title, and author.

```
C:\Cassandra\NodeJS>node preparedquery.js
key: catalog1
journal: Oracle Magazine
publisher: Oracle Publishing
edition: November-December 2013
title: Engineering as a Service
author: David A. Kelly
key: catalog2
journal: Oracle Magazine
publisher: Oracle Publishing
edition: November-December 2013
title: Quintessential and Collaborative
author: Tom Haunert
```

Figure 6.10

Getting the rows catalog1 and catalog2.

Source: Microsoft Corporation.

STREAMING QUERY ROWS

The `eachRow(query, [params], [consistency], rowCallback, endCallback)` method is used to stream rows as they are received. The `rowCallback(n, row)` callback function is invoked after each row is received, and the `endCallback(err, rowLength)` function is invoked after all rows have been received. The `rowCallback(n, row)` function may be used to output the row and the row number. Row columns are properties of the row object. For example, the `row.title` property is the value of the `title` column. The `endCallback(err, rowLength)` callback function may be used to output an error (if any) or the row length.

Create a JavaScript file, `streaming_query_row.js`, and create an instance of the `Client` class. Invoke the `eachRow()` method to run the CQL query `SELECT key, journal, publisher, edition, title, author FROM nodejscatalog`. In the row callback function, output the row number and the `title` and `author` columns.

```
function(n, row) {
    //the callback will be invoked per each row as soon as they are received
    console.log('title: ', n, row.title);
    console.log('author: ', n, row.author);
}
```

In the callback function called after all rows have been received, output the error (if any) or the row length:

```
function (err, rowLength) {
    if (err) console.log(err);
    console.log('%d rows where returned', rowLength);
}
```

The streaming_query_row.js script appears in Listing 6.6.

Listing 6.6 The streaming_query_row.js Script

```
var cql = require('node-cassandra-cql');
var client = new cql.Client({hosts: ['localhost:9042'], keyspace:
  'catalog_nodejs'});
client.eachRow('SELECT key, journal, publisher, edition, title, author FROM
nodejscatalog',
  function(n, row) {
    //the callback will be invoked per each row as soon as they are received
    console.log('title: ', n, row.title);
    console.log('author: ', n, row.author);
  },
  function (err, rowLength) {
    if (err) console.log(err);
    console.log('%d rows where returned', rowLength);
  }
);
```

Run the streaming_query_row.js script with the following command:

```
node streaming_query_row.js
```

The row number and the title and author columns are output for each row. The total number of rows received is also output after all rows have been received, as shown in Figure 6.11.

```
C:\Cassandra\NodeJS>node streaming_query_row.js
title: 0 Engineering as a Service
author: 0 David A. Kelly
title: 1 Quintessential and Collaborative
author: 1 Tom Haunert
2 rows where returned
```

Figure 6.11

Output from streaming query rows.

Source: Microsoft Corporation.

STREAMING A FIELD

Suppose you want to stream the last field in the result to a text file. The `streamField(query, [params], [consistency], rowCallback, [endCallback])` method streams the last field in a row as the first chunk of the field is received. The callback function `rowCallback(n, row, streamField)`—in which `n` is the row index, `row` is the row object, and `streamField` is the last field to stream—is used to stream the last field. The `rowCallback` function is invoked as the first few raw bytes of the last field are received. `streamField` is a `Readable Stream2` object. The `row` in the `rowCallback` function is also an object similar to the `row` object in the `rowCallback` function in the `eachRow()` method. The `row` object does not include the last column/field of the row, however, because the last column is to be streamed and included in the `streamField` object.

Create a JavaScript file, `streaming_field.js`, to stream the last field of row(s) as it is received. Import the Node.js driver for Cassandra as in other scripts. Also import the File System module `fs`.

```
var fs = require('fs');
```

The File System module is used to create a `WriteStream` object to which to stream the last field. Create a `Client` instance for a connection to Cassandra. Invoke the `streamField(query, [params], [consistency], rowCallback, [endCallback])` method with the CQL query as a prepared statement query, `SELECT key, journal, publisher, edition, title, author FROM nodejscatalog WHERE key=?`, with a placeholder (?) for the key column. Provide the key value in the `params` arg as `['catalog']`. Define the `rowCallback` function to stream the last field to a text file, `output.txt`. The `streamField` object is an instance of the `stream.Readable` class, which is an abstraction of a data source. Data is emitted by a `Readable` stream, but only after a destination is ready to receive the data. The `Readable` class generates the events discussed in Table 6.7.

Table 6.7 Readable Class Events

Event	Description
<code>readable</code>	When a chunk of data can be read from a stream.
<code>data</code>	Represents a chunk of data. When a data event listener is registered with <code>Readable</code> , which is <code>streamField</code> , the stream emits the chunk of data to the handler function.
<code>end</code>	Emitted when no more data is available in the stream.
<code>close</code>	Emitted when the resource is closed. Not emitted by all streams.
<code>error</code>	Emitted if an error is generated receiving the data.

In the `rowCallback` function called `function(n, row, streamField)`, create a `WriteStream` object from the File System object `fs` using the `createWriteStream()` method. Set `output.txt` as the destination file.

```
var writable = fs.createWriteStream('output.txt');
```

Log the row index and the `row` object to the console:

```
console.log(n);
console.log(row);
```

The `Readable` class provides the `pipe(destination, [options])` method to pipe the stream to a destination such as a file. The destination must be a `Writable Stream`, which is a `WriteStream` object you created for the `output.txt` file. The `pipe` method pulls data out of the readable stream and pipes it to the writable destination such that the destination is not overwhelmed by the fast readable stream. Invoke the `pipe` method on the `streamField` object, which is a `Readable` type, with the `writable` object as the argument.

```
streamField.pipe(writable);
```

Register the `data` event with the `streamField` object and provide a callback function to handle the chunk of data. Log the chunk length to the console.

```
streamField.on('data', function(chunk) {
  console.log('got %d bytes of data', chunk.length);
})
```

In the `endCallback` function, log the error message if an error is generated. Alternatively, log the row length if a row is returned.

```
function (err, rowLength) {
  if (err) console.log(err);
  console.log('%d rows where returned', rowLength);
}
```

The `streaming_field.js` script appears in Listing 6.7.

Listing 6.7 The streaming_field.js Script

```
var cql = require('node-cassandra-cql');
var fs = require('fs');
var client = new cql.Client({hosts: ['localhost:9042'], keyspace:
  'catalog_nodejs'});
client.streamField('SELECT key, journal, publisher, edition, title, author  FROM
nodejscatalog WHERE key=?', ['catalog1'],
  function(n, row, streamField) {
```

```
//the callback will be invoked per each row as soon as they are received.  
var writable = fs.createWriteStream('output.txt');  
console.log(n);  
console.log(row);  
streamField.pipe(writable);  
streamField.on('data', function(chunk) {  
    console.log('got %d bytes of data', chunk.length);  
});  
    //The stream is a Readable Stream2 object  
, function (err, rowLength) {  
    if (err) console.log(err);  
    console.log('%d rows were returned', rowLength);  
}  
};
```

Run the streaming_field.js script with the following command:

```
node streaming_field.js
```

In the console output, the row index is logged as 0. Next, the row object is logged. The last field in the row, author, is not logged to the console because it is streamed to the output.txt file. The endCallback function logs that “1 rows were returned.” When the data event is emitted by the streamField object, the callback function registered for the data event outputs the bytes of data emitted, as shown in Figure 6.12.

```
C:\Cassandra\NodeJS>node streaming_field.js  
0  
{ columns:  
  [ { name: 'key', type: [Object] },  
    { name: 'journal', type: [Object] },  
    { name: 'publisher', type: [Object] },  
    { name: 'edition', type: [Object] },  
    { name: 'title', type: [Object] },  
    { name: 'author', type: [Object] } ],  
  _col_key: 0,  
  _col_journal: 1,  
  _col_publisher: 2,  
  _col_edition: 3,  
  _col_title: 4,  
  _col_author: 5 1,  
  key: 'catalog1',  
  journal: 'Oracle Magazine',  
  publisher: 'Oracle Publishing',  
  edition: 'November-December 2013',  
  title: 'Engineering as a Service' }  
1 rows were returned  
got 15 bytes of data
```

Figure 6.12
Output from streaming a field script.

Source: Microsoft Corporation.

The output.txt file is generated in the same directory as the one in which the streaming_field.js script is run. The output.txt file has only the last field in the catalog1 row, which is David A. Kelly, as shown in Figure 6.13.



Figure 6.13

The field streamed to a text file.

Source: Microsoft Corporation.

STREAMING THE RESULT

The `stream(query, [params], [consistency], [callback])` method streams each row as a row becomes available. The method returns a Readable Streams2 object and emits the readable event when a row can be read from a stream. The readable event is discussed in Table 6.7. The stream may be piped to a text file.

Create a JavaScript file, `streaming_result.js`, and import the File System module as in the previous section. Create a `Client` instance for a connection with Cassandra. Invoke the `stream` method using a CQL query for a prepared statement, `SELECT key, journal, publisher, edition, title, author FROM nodejscatalog WHERE key=?`. Provide the `params` argument as `['catalog1']`. The method returns a `stream.Readable` object. Register the readable event with the `Readable` object.

```
client.stream('SELECT key, journal, publisher, edition, title, author FROM nodejscatalog WHERE key=?', ['catalog1']).on('readable', function () { })
```

In the callback function, create a `WriteStream` object for a text file `output2.txt`, which is to be the destination of the query result stream.

```
var writable = fs.createWriteStream('output2.txt');
```

The `readable` event is emitted as soon as a row is received and parsed. Readable streams are either in flowing mode or non-flowing mode. The `streamField` in the previous section switches to flowing mode when the `data` event is registered with the stream. The stream returned by the `stream` method is in non-flowing mode and switches to flowing mode when the `read()` method is invoked on the stream.

```
var row;
  while (row = this.read()) {
}
```

Invoke the `writable.write(chunk, [encoding], [callback])` method to write a chunk of data to `output2.txt`.

```
writable.write(row.journal+ ' ');
writable.write(row.publisher+ ' ');
writable.write(row.edition+ ' ');
writable.write(row.title+ ' ');
writable.write(row.author+ '');
```

Log the title and author to the console.

```
console.log('title %s and author %s', row.title, row.author);
```

Callback functions for other events emitted by `Readable` may also be registered with the `Readable` stream. For example, a callback function for the `end` event may be registered to indicate that a stream has ended. A callback function for the `error` event may be registered to indicate an error. The `streaming_result.js` script appears in Listing 6.8.

Listing 6.8 The streaming_result.js Script

```
var cql = require('node-cassandra-cql');
var fs = require('fs');
var client = new cql.Client({hosts: ['localhost:9042'], keyspace:
  'catalog_nodejs'});
client.stream('SELECT key, journal, publisher, edition, title, author FROM
  nodejscatalog WHERE key=?', ['catalog1']).on('readable', function () {
  var writable = fs.createWriteStream('output2.txt');
```

```
//readable is emitted as soon a row is received and parsed
var row;
while (row = this.read()) {
  writable.write(row.journal+' ');
  writable.write(row.publisher+' ');
  writable.write(row.edition+' ');
  writable.write(row.title+' ');
  writable.write(row.author+' ');
  console.log('title %s and author %s', row.title, row.author);
}
})
.on('end', function () {
  //stream ended, there aren't any more rows
})
.on('error', function (err) {
  console.log(err);
  //Something went wrong: err is a response error from Cassandra
});
```

Invoke the streaming_result.js script with the following command:

```
node streaming_result.js
```

The title and author columns for row catalog1 are output to the console, as shown in Figure 6.14.



```
C:\ Command Prompt - node streaming_result.js
C:\Cassandra\NodeJS>node streaming_result.js
title Engineering as a Service and author David A. Kelly
```

Figure 6.14

Output from a script to stream a result.

Source: Microsoft Corporation.

The complete row, including the title and author columns, is streamed to an output file, as shown in Figure 6.15.

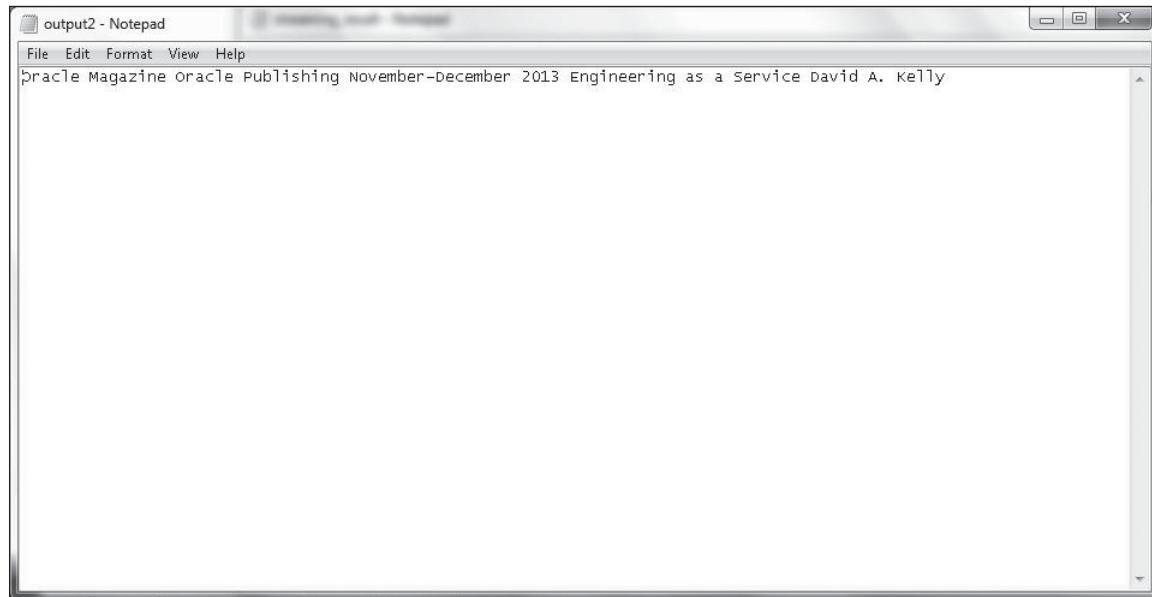


Figure 6.15

A complete row streamed to a text file.

Source: Microsoft Corporation.

UPDATING DATA IN TABLE

In this section, you will update row data. You can use the `execute(query, [params], [consistency], callback)` method to run an UPDATE CQL statement. Create a script, `update.js`, and create a Client connection as before. Run the prepared statement `UPDATE Node.jscatalog SET edition=? WHERE key=?` and provide the params as `['11/12 2013', 'catalog1']`. Output an error (if any) in the callback function or output a message to indicate that a row has been updated. Then run a CQL query to select the `catalog1` row to find out if the row got updated. The `update.js` script appears in Listing 6.9.

Listing 6.9 The `update.js` Script

```
var cql = require('node-cassandra-cql');
var client = new cql.Client({hosts: ['localhost:9042']}, keyspace:
  'catalog_Node.js');
client.execute("UPDATE Node.jscatalog SET edition=? WHERE key=?", ['11/12
2013', 'catalog1'],
  cql.types.consistencies.quorum,
  function(err) {
    if (err) console.log(err);
```

```

    else console.log('table row updated');
}
);
client.execute("SELECT key, journal, publisher, edition,title,author FROM
Node.jscatalog WHERE key=?", ['catalog1'],
cql.types.consistencies.quorum,
function(err, result) {
  if (err) console.log(err);
  else {console.log('key: ' +result.rows[0].key);
  console.log(' journal: ' +result.rows[0].journal);
  console.log(' publisher: ' +result.rows[0].publisher);
  console.log(' edition: ' +result.rows[0].edition);
  console.log(' title: ' +result.rows[0].title);
  console.log(' author: ' +result.rows[0].author);
}
}
);

```

Run the update.js script with the following command:

```
node update.js
```

As indicated in the output in Figure 6.16, the catalog1 row is updated.

```
C:\Cassandra\NodeJS>node update.js
key: catalog1
journal: Oracle Magazine
publisher: Oracle Publishing
edition: 11/12 2013
title: Engineering as a Service
author: David A. Kelly
table row updated
```

Figure 6.16

Updating a row.

Source: Microsoft Corporation.

DELETING A COLUMN

You can use the `execute(query, [params], [consistency], callback)` method to delete a column from a row. Create a script, `deleteColumn.js`, and create a connection with the Cassandra database using a Client instance. Run the CQL prepared statement `DELETE journal FROM Node.jscatalog where key=?` with params as catalog1 to delete the journal column from the catalog1 row. Then run a CQL query to select all columns from the catalog1 row to find out if a column got deleted. The `deleteColumn.js` script appears in Listing 6.10.

Listing 6.10 The deleteColumn.js Script

```
var cql = require('node-cassandra-cql');
var client = new cql.Client({hosts: ['localhost:9042']}, keyspace:
'catalog_nodejs');
client.execute("DELETE journal FROM nodejscatalog where key=?", ['catalog1'],
  cql.types.consistencies.quorum,
  function(err) {
    if (err) console.log(err);
    else {console.log('table column deleted');
  client.execute("SELECT * FROM nodejscatalog WHERE key=?", ['catalog1'],
    cql.types.consistencies.quorum,
    function(err, result) {
      if (err) console.log(err);
      else {console.log(result);
    }
  }
);
}
);
});
```

Run the `deleteColumn.js` script with the following command:

node deleteColumn.js

As indicated in the output in Figure 6.17, the journal column is null.

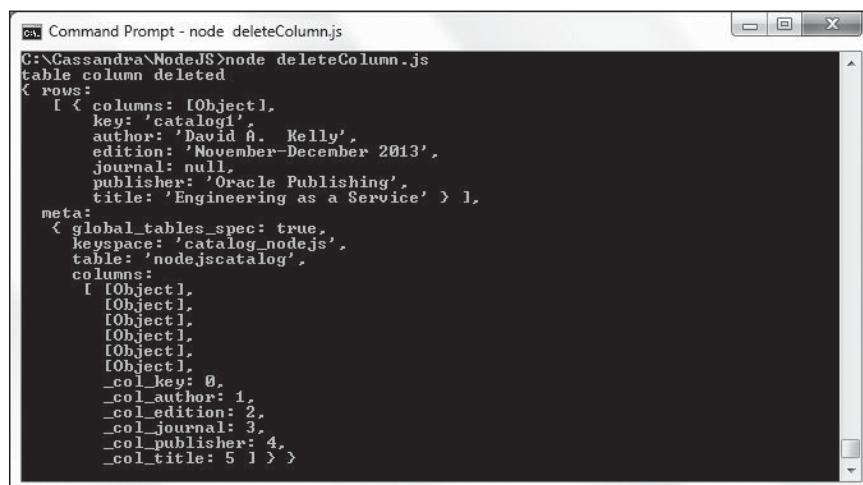


Figure 6.17

Deleting a column.

Source: Microsoft Corporation.

DELETING A ROW

Next, you will delete an entire row using the `execute(query, [params], [consistency], callback)` method. Create a script, `deleteRow.js`, and create a connection with the Cassandra cluster using a `Client` instance. Run a CQL prepared statement `DELETE FROM Node.jscatalog where key=?` with `params` as `catalog1` to delete the row with the key `catalog1`. Then run a CQL query to select all columns from the row for the `catalog1` key. The `deleteRow.js` script appears in Listing 6.11.

Listing 6.11 The `deleteRow.js` Script

```
var cql = require('node-cassandra-cql');
var client = new cql.Client({hosts: ['localhost:9042']}, keyspace:
  'catalog_nodejs');
client.execute("DELETE FROM nodejscatalog where key=?", ['catalog1'],
  cql.types.consistencies.quorum,
  function(err) {
    if (err) console.log(err);
    else {
      console.log('table row deleted');
      client.execute("SELECT * FROM nodejscatalog WHERE key=?", ['catalog1'],
        cql.types.consistencies.quorum,
        function(err, result) {
          if (err) console.log(err);
          else {console.log(result);
        }
      }
    }
  );
});
```

Run the `deleteRow.js` script with the following command:

```
node deleteRow.js
```

The row for the key `catalog1` is deleted and the subsequent query to select the `catalog1` row returns an empty result set, as shown in Figure 6.18.



```
Command Prompt - node deleteRow.js
C:\Cassandra\NodeJS>node deleteRow.js
table row deleted
{ rows: [1] }
```

Figure 6.18

Deleting a row.

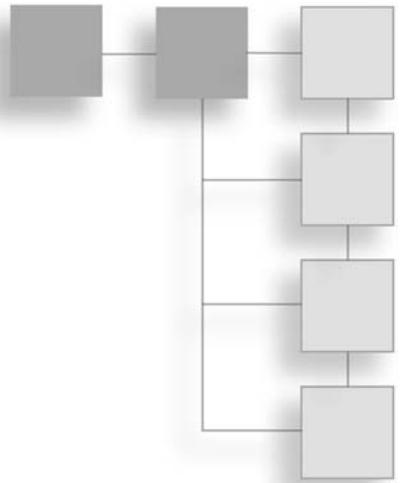
Source: Microsoft Corporation.

SUMMARY

In this chapter, you used the Node.js driver for Apache Cassandra to connect with Cassandra, add data, select data, update data, and delete data. In the next chapter, you will migrate MongoDB, another NoSQL database, to Apache Cassandra.

PART III

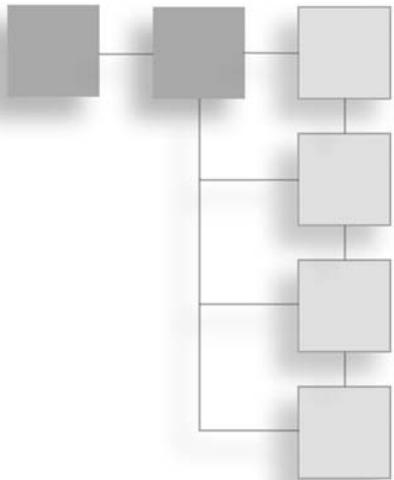
Migration



This page intentionally left blank

CHAPTER 7

MIGRATING MongoDB TO CASSANDRA



MongoDB is an open source NoSQL database written in C++. MongoDB stores documents in a JSON-like format called BSON. MongoDB's BSON format is much different from the flexible table format of Cassandra. This chapter discusses the procedure to migrate a BSON document stored in the MongoDB server to a table in a Cassandra database.

SETTING THE ENVIRONMENT

To set the environment, you must install the following software:

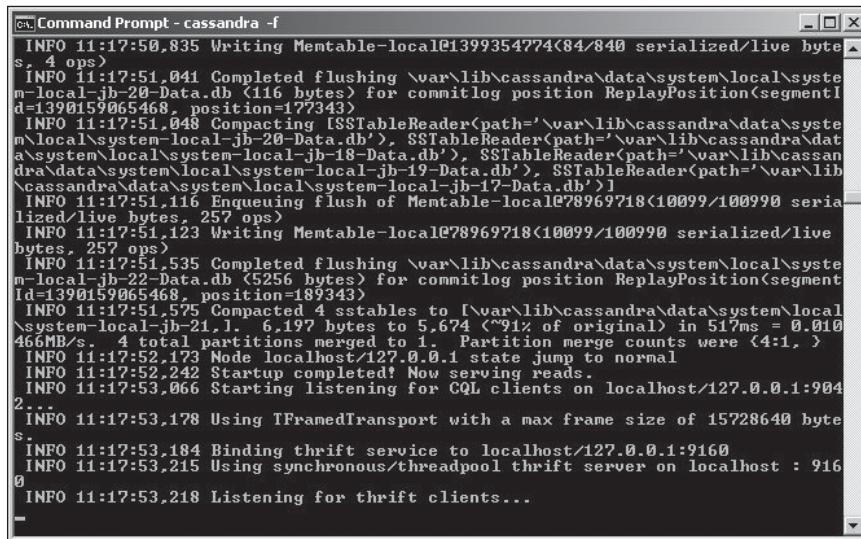
- MongoDB Windows binaries from <http://www.mongodb.org/downloads>. Extract the TGZ or ZIP file to a directory and add C:\MongoDB\mongodb-win32-x86_64-2008plus-2.4.9\bin to the PATH environment variable.
- MongoDB Java driver JAR from <http://central.maven.org/maven2/org/mongodb/mongo-java-driver/>.
- Eclipse IDE for Java EE developers from <http://www.eclipse.org/downloads/>.
- Apache Commons Lang 2.6 commons-lang-2.6-bin.zip from http://commons.apache.org/proper/commons-lang/download_lang.cgi. Extract it to the commons-lang-2.6-bin directory.

- Hector Java client hector-core-1.1-4.jar or a later version from <http://repo2.maven.org/maven2/org/heap/hectorclient/hector-core/1.1-4/>.
- Apache Cassandra 2.04 from <http://cassandra.apache.org/download/>.
Add C:\Cassandra\apache-cassandra-2.0.4\bin to the PATH variable.

Start Apache Cassandra server with the following command:

```
>cassandra -f
```

Apache Cassandra is started, as shown in Figure 7.1.



The screenshot shows a Windows Command Prompt window titled "Command Prompt - cassandra -f". The window displays the startup logs for the Apache Cassandra server. The logs include various INFO messages about flushing, compacting, and merging SSTables, as well as the start of the thrift service and listening for clients. The log output is as follows:

```

INFO 11:17:50.835 Writing Memtable-local@1399354774<84/840 serialized/live bytes, 4 ops
INFO 11:17:51.041 Completed flushing \var\lib\cassandra\data\system\local\system-local-jb-20-Data.db (116 bytes) for commitlog position ReplayPosition(segmentId=1390159065468, position=177343)
INFO 11:17:51.048 Compacting {SSTableReader(path='\\var\\lib\\cassandra\\data\\system\\local\\system-local-jb-20-Data.db'), SSTableReader(path='\\var\\lib\\cassandra\\data\\system\\local\\system-local-jb-18-Data.db'), SSTableReader(path='\\var\\lib\\cassandra\\data\\system\\local\\system-local-jb-19-Data.db'), SSTableReader(path='\\var\\lib\\cassandra\\data\\system\\local\\system-local-jb-17-Data.db')}
INFO 11:17:51.116 Enqueuing flush of Memtable-local@78969718<10099/100990 serialized/live bytes, 257 ops
INFO 11:17:51.123 Writing Memtable-local@78969718<10099/100990 serialized/live bytes, 257 ops
INFO 11:17:51.535 Completed flushing \var\lib\cassandra\data\system\local\system-local-jb-22-Data.db (5256 bytes) for commitlog position ReplayPosition(segmentId=1390159065468, position=189343)
INFO 11:17:51.575 Compacted 4 sstables to [\\var\\lib\\cassandra\\data\\system\\local\\system-local-jb-21.l. 6,197 bytes to 5,674 (<91% of original) in 517ms = 0.010466MB/s. 4 total partitions merged to 1. Partition merge counts were <4:1, >
INFO 11:17:52.173 Node localhost/127.0.0.1 state jump to normal
INFO 11:17:52.242 Startup completed! Now serving reads.
INFO 11:17:53.066 Starting listening for CQL clients on localhost/127.0.0.1:9042.
INFO 11:17:53.178 Using TFramedTransport with a max frame size of 15728640 bytes.
INFO 11:17:53.184 Binding thrift service to localhost/127.0.0.1:9160
INFO 11:17:53.215 Using synchronous/threadpool thrift server on localhost : 9160
INFO 11:17:53.218 Listening for thrift clients...

```

Figure 7.1

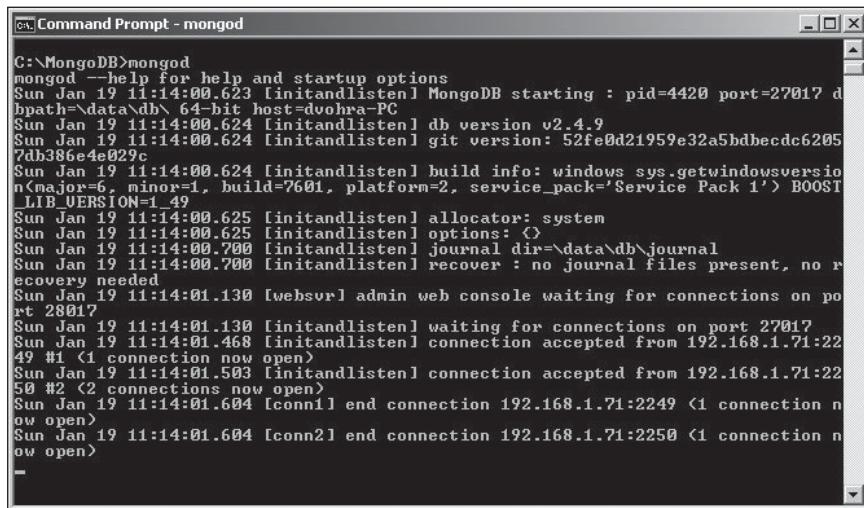
Starting Apache Cassandra.

Source: Microsoft Corporation.

Start MongoDB server with the following command:

```
>mongod
```

MongoDB server is started, as shown in Figure 7.2.



The screenshot shows a Windows Command Prompt window titled "Command Prompt - mongod". The window displays the output of the "mongod" command, which starts the MongoDB database. The logs include details about the MongoDB version (v2.4.9), build information (52fe0d21959e32a5bdbecdc6205), and system details. It also shows connections being accepted from 192.168.1.71:2249 and 192.168.1.71:2250. The log ends with a message indicating no journal files are present.

```
C:\MongoDB>mongod
mongod --help for help and startup options
Sun Jan 19 11:14:00.623 [initandlisten] MongoDB starting : pid=4420 port=27017 dbpath=\data\db\ 64-bit host=dvohma-PC
Sun Jan 19 11:14:00.624 [initandlisten] db version v2.4.9
Sun Jan 19 11:14:00.624 [initandlisten] git version: 52fe0d21959e32a5bdbecdc6205
7db386e4e029c
Sun Jan 19 11:14:00.624 [initandlisten] build info: windows sys.getwindowsversion(major=6, minor=1, build=7601, platform=2, service_pack='Service Pack 1') BOOST_LIB_VERSION=1_49
Sun Jan 19 11:14:00.625 [initandlisten] allocator: system
Sun Jan 19 11:14:00.625 [initandlisten] options: {}
Sun Jan 19 11:14:00.700 [initandlisten] journal dir=\data\db\journal
Sun Jan 19 11:14:00.700 [initandlisten] recover : no journal files present, no recovery needed
Sun Jan 19 11:14:01.130 [websvr] admin web console waiting for connections on port 28017
Sun Jan 19 11:14:01.130 [initandlisten] waiting for connections on port 27017
Sun Jan 19 11:14:01.468 [initandlisten] connection accepted from 192.168.1.71:2249 #1 <1 connection now open>
Sun Jan 19 11:14:01.503 [initandlisten] connection accepted from 192.168.1.71:2250 #2 <2 connections now open>
Sun Jan 19 11:14:01.604 [conn1] end connection 192.168.1.71:2249 <1 connection now open>
Sun Jan 19 11:14:01.604 [conn2] end connection 192.168.1.71:2250 <1 connection now open>
```

Figure 7.2

Starting MongoDB.

Source: Microsoft Corporation.

CREATING A JAVA PROJECT

In this section, you will create a Java project in Eclipse IDE to migrate a MongoDB document to Apache Cassandra. Follow these steps:

1. Select File > New > Other.
2. In the New dialog box, select Java Project or Java > Java Project. Then click Next, as shown in Figure 7.3.

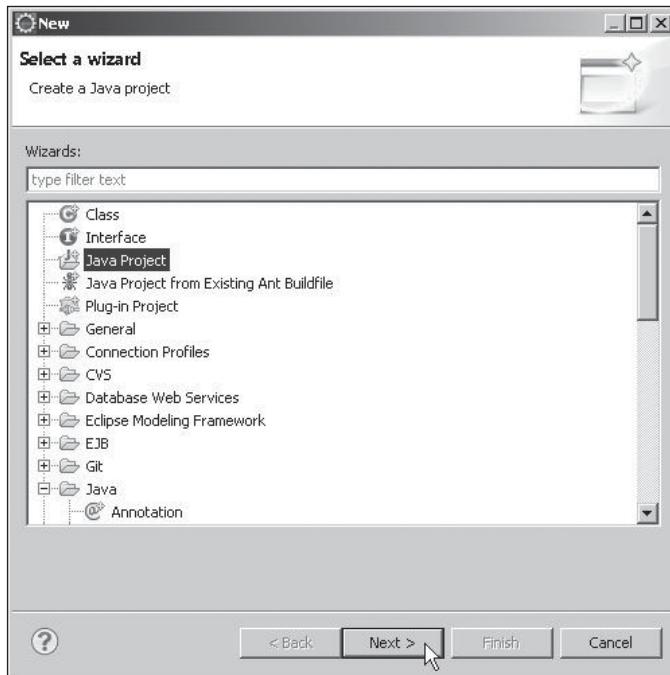


Figure 7.3
Selecting the Java Project wizard.

Source: Eclipse Foundation.

3. In the New Java Project dialog box, specify a project name (MigrateMongoDB), select the Use Default Location checkbox, select JDK 1.7 as the JRE (Use Default JRE may already be selected), and click Next, as shown in Figure 7.4.

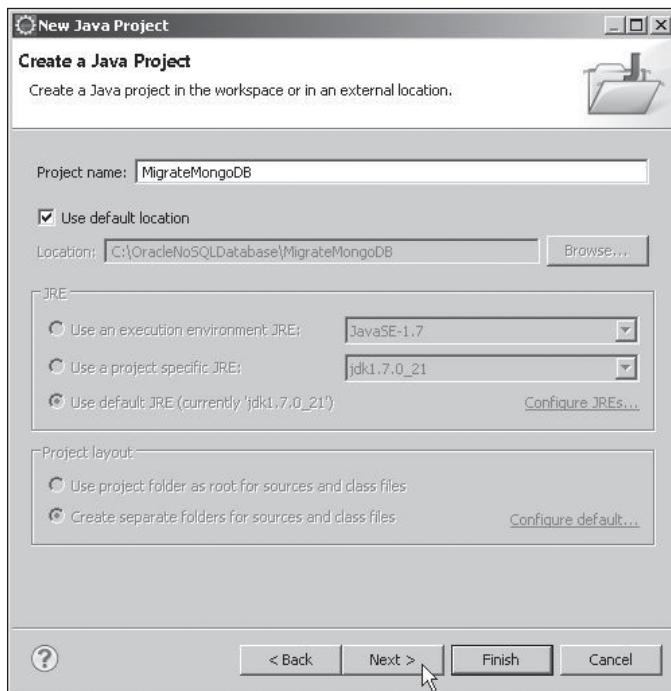
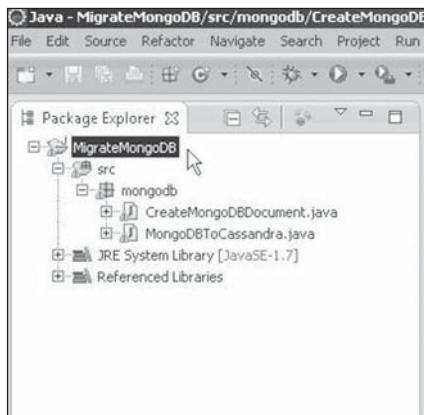


Figure 7.4
Specifying a project name.
Source: Eclipse Foundation.

4. In the Java Settings dialog box, select the default settings. Select Allow Output Folders for Source Folders. Then click Finish. A Java project, MigrateMongoDB, is created.
5. Add two Java classes, `CreateMongoDBDocument` and `MongoDBToCassandra`. The `CreateMongoDBDocument` class is for creating a BSON document in MongoDB and the `MongoDBToCassandra` class is for migrating the BSON document from MongoDB to Apache Cassandra. To add a Java class, select File > New > Other. Then, in the New dialog box, select Java > Class and click Next. Finally, in the New Java Class wizard, specify a package name and a class name and click Finish. The directory structure of the MigrateMongoDB project is shown in Figure 7.5.

**Figure 7.5**

The directory structure of the MigrateMongoDB project.

Source: Eclipse Foundation.

6. Next, you must add some JAR files for Cassandra and MongoDB to the project class path. Add the JAR files listed in Table 7.1. These JAR files are from the Cassandra server download, the MongoDB server download, the Hector Java client for Cassandra, and some third-party JARs.

Table 7.1 JAR Files for Migration

JAR	Description
apache-cassandra-2.0.4.jar	Apache Cassandra
hector-core-1.1-4.jar	High-level Java client for Cassandra
commons-codec-1.5	Provides implementation of common encoders and decoders
commons-lang-2.6	Provides extra classes for the manipulation of Java core classes
guava-15.0	Google's core libraries used in Java-based projects
libthrift-0.9.1.jar	Software framework for scalable cross-language services development
log4j-1.2.16	Logging library for Java
mongo-java-driver-2.11.3.jar	MongoDB Java driver
slf4j-api-1.7.2	Simple Logging Facade for Java (SLF4J), which serves as an abstraction for various logging frameworks
slf4j-log4j12-1.7.2	An SLF4J abstraction for log4j

To add the required JARs, right-click the project node in Package Explorer and select Properties. Then, in the Properties dialog box, select Java Build Path. Finally, click the Add External JARs button to add the external JAR files. The JARs added to the migration project are shown in Figure 7.6.

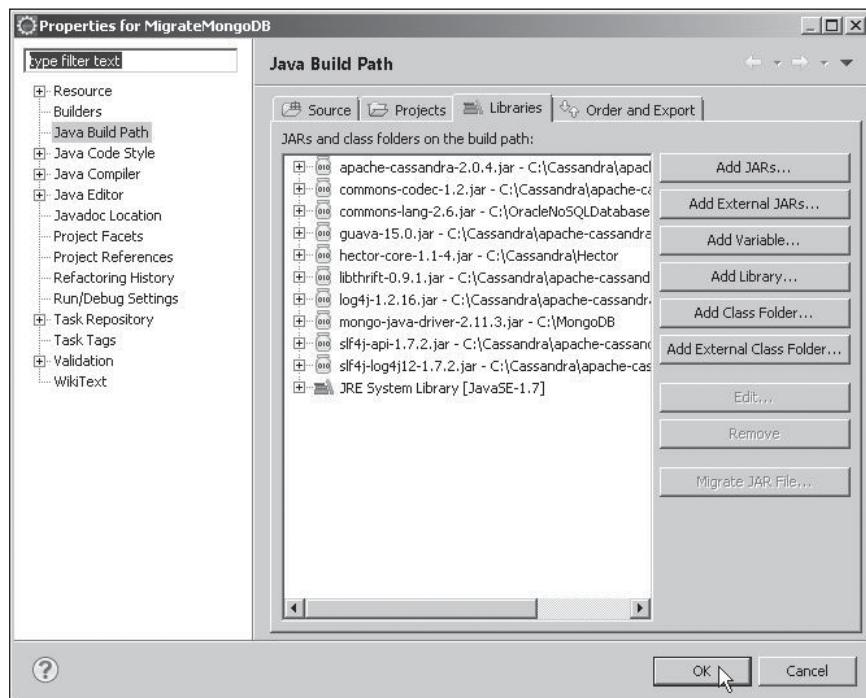


Figure 7.6

Adding JARs to the Java build path.

Source: Eclipse Foundation.

CREATING A BSON DOCUMENT IN MongoDB

You need to add some data to MongoDB to migrate the data to the Cassandra database. Here, you will create a document in MongoDB using the Java application CreateMongoDBDocument. The main package for the MongoDB Java driver is com.mongodb. A MongoDB client to connect to MongoDB server is represented with the MongoClient class. A MongoClient object provides connection pooling and only one instance is required for the application. Create a MongoClient instance using the MongoClient(List<ServerAddress> seeds) constructor. Supply the IPv4 address of the host and port as 27017.

```
MongoClient mongoClient = new MongoClient(Arrays.asList(new ServerAddress
("localhost", 27017)));
```

A logical database in MongoDB is represented with the `com.mongodb.DB` class. Obtain a `com.mongodb.DB` instance for the local database, which is a default MongoDB database instance, using the `getDB(String dbname)` method in the `MongoClient` class. MongoDB stores data in collections. Get all collections from the database instance using the `getCollectionNames()` method in `com.mongodb.DB` class.

```
Set<String> colls = db.getCollectionNames();
```

The `getCollectionNames()` method returns a `Set<String>` of collections. Iterate over the collection to output the collection names.

```
for (String s : colls) {
    System.out.println(s);
}
```

A MongoDB collection is represented with the `DBCollection` class. Create a new `DBCollection` instance using the `createCollection(String name,DBObject options)` method in the `com.mongodb.Db` class. You specify the options to create a collection using a key/value map represented with the `DBObject` interface. The options that may be specified are listed in Table 7.2.

Table 7.2 Options to Create a DBCollection

Option	Type	Description
capped	boolean	Enables a cap on the collection. Set to <code>false</code> by default. If set to <code>true</code> , must also specify the <code>size</code> option.
size	int	The size of the collection in terms of the number of documents.
max	int	The maximum cap in terms of number of documents.

Create a collection called `catalog` and set the options to `null`:

```
DBCollection coll = db.createCollection("catalog", null);
```

A MongoDB-specific BSON object is represented with the `BasicDBObject` class, which implements the `DBObject` interface. The `BasicDBObject` class provides the constructors listed in Table 7.3 to create a new instance.

Table 7.3 BasicDBObject Class Constructors

Constructor	Description
BasicDBObject()	Creates an empty object
BasicDBObject(int size)	Creates an object of a specified size
BasicDBObject(Map m)	Creates an object from a map
BasicDBObject(String key, Object value)	Creates an object with key/value pairs

The `BasicDBObject` class provides some other utility methods, some of which are listed in Table 7.4.

Table 7.4 BasicDBObject Class Utility Methods

Method	Description
append(String key, Object val)	Appends a key/value pair to the object and returns a new instance
toString()	Returns a JSON serialization of the object

Create a `BasicDBObject` instance using the `BasicDBObject(String key, Object value)` constructor and use the `append(String key, Object val)` method to append key/value pairs:

```
BasicDBObject catalog = new BasicDBObject("journal", "Oracle Magazine").append("publisher", "Oracle Publishing").append("edition", "November December 2013").append("title", "Engineering as a Service").append("author", "David A. Kelly");
```

The `DBCollection` class provides an overloaded `insert` method to add an instance(s) of `BasicDBObject` to a collection. Add the `catalog` `BasicDBObject` to the `DBCollection` instance for the `catalog` collection:

```
coll.insert(catalog);
```

The `DBCollection` class also provides an overloaded `findOne()` method to find a `DBObject` instance. Obtain the document added using the `findOne()` method:

```
DBObject catalog = coll.findOne();
```

Output the DBObject object found by iterating over the Set obtained from the DBObject using the keySet() method. The keySet() method returns a Set<String>. Create an Iterator from the Set<String> using the iterator() method. While the Iterator has elements as determined by the hasNext() method, obtain the elements using the next() method. Each element is a key in the DBObject fetched. Obtain the value for the key using the get(String key) method in DBObject.

```
Set<String> set=catalog.keySet();
Iterator iter=set.iterator();
while(iter.hasNext()){
Object obj=iter.next();
System.out.println(obj);
System.out.println(catalog.get(obj.toString()));
}
```

The CreateMongoDBDocument class appears in Listing 7.1.

Listing 7.1 The CreateMongoDBDocument Class

```
package mongodb;

import java.net.UnknownHostException;
import java.util.Arrays;
import java.util.Iterator;
import java.util.Set;

import com.mongodb.MongoClient;
import com.mongodb.MongoException;
import com.mongodb.WriteConcern;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.BasicDBObject;
import com.mongodb.DBObject;
import com.mongodb.DBCursor;
import com.mongodb.ServerAddress;

public class CreateMongoDBDocument {

    public static void main(String[] args) {
        try {
            MongoClient mongoClient = new MongoClient(
                Arrays.asList(new ServerAddress("localhost", 27017)));

```

```
/*for (String s : mongoClient.getDatabaseNames()) {
System.out.println(s);
}
*/
DB db = mongoClient.getDB("local");
/*Set<String> colls = db.getCollectionNames();
for (String s : colls) {
System.out.println(s);
}*/
DBCollection coll = db.createCollection("catalog", null);
/*BasicDBObject catalog = new BasicDBObject("journal",
"Oracle Magazine").append("publisher", "Oracle
Publishing")
.append("edition", "November December 2013")
.append("title", "Engineering as a Service")
.append("author", "David A. Kelly");*/
//coll.insert(catalog);
DBObject catalog = coll.findOne();
//System.out.println(catalog);
Set<String> set=catalog.keySet();
Iterator iter=set.iterator();
while(iter.hasNext()){
    Object obj=iter.next();
    System.out.println(obj);
    System.out.println(catalog.get(obj.toString()));
}
} catch (UnknownHostException e) {
    e.printStackTrace();
}
}
}
```

To run the CreateMongoDBDocument application, right-click the CreateMongoDB-Document.java file in Package Explorer and select Run As > Java Application, as shown in Figure 7.7.

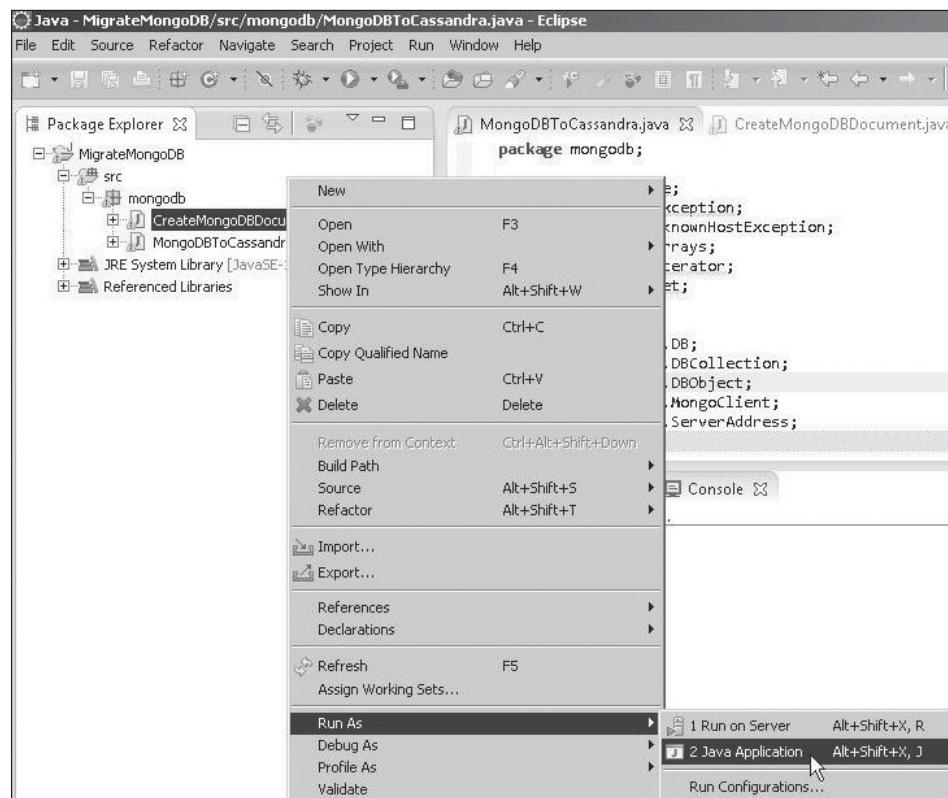


Figure 7.7
Running the CreateMongoDBDocument application.

Source: Eclipse Foundation.

A new BSON document is stored in a new collection, catalog, in the MongoDB database. The document stored is also output as such and as key/value pairs, as shown in Figure 7.8.

```
@ Javadoc Declaration Console ✎
<terminated> CreateMongoDBDocument [Java Application]
_id
52dac0e5ca056c2f7125db37
journal
Oracle Magazine
publisher
Oracle Publishing
edition
November December 2013
title
Engineering as a Service
author
David A. Kelly
```

Figure 7.8
Storing a document in MongoDB.

Source: Eclipse Foundation.

MIGRATING THE MONGODB DOCUMENT TO CASSANDRA

In this section, you will query the BSON document stored earlier in the MongoDB server and migrate the BSON document to a Cassandra database. You will use the `MongoDBToCassandra` class to migrate the BSON document from the MongoDB server to Cassandra. Create a `MongoClient` instance, which is required for migrating, as discussed in the previous section to add a document.

```
MongoClient mongoClient = new MongoClient(Arrays.asList(new ServerAddress
("localhost", 27017)));
```

Create a `DB` object for the local database instance using the `getDB(String dbname)` method in `MongoClient`. Using the `DB` instance gets the `catalog` collection as a `DBCollection` object. Create a `DBObject` instance from the document stored in MongoDB in the previous section using the `findOne()` method in the `DBCollection` class.

```
DB db = mongoClient.getDB("local");
DBCollection coll = db.getCollection("catalog");
DBObject catalog = coll.findOne();
```

Next, you will migrate the resulting `DBObject` to the Cassandra database. Some of the procedures for migrating MongoDB to Cassandra are the same as for migrating Couchbase to Cassandra, which is discussed in Chapter 8, “Migrating Couchbase to Cassandra.”

The `me.prettyprint.hector.api.Cluster` interface represents a cluster of Cassandra hosts. To access a Cassandra cluster, create a `Cluster` instance for a Cassandra cluster using the `getOrCreateCluster(String clusterName, String hostIp)` method as follows:

```
Cluster cluster =
HFactory.getOrCreateCluster("migration-cluster", "localhost:9160");
```

Next, create a schema if not already defined. A schema consists of a column family definition and a keyspace definition. Use the `describeKeyspace` method in `Cluster` to obtain a `KeyspaceDefinition` object for `HectorKeyspace` keyspace. If the keyspace definition object is `null`, invoke a `createSchema()` method to create a schema.

```
KeyspaceDefinition keyspaceDef = cluster.describeKeyspace("HectorKeyspace");
if (keyspaceDef == null) {
    createSchema();
}
```

As discussed in Chapter 1, “Using Cassandra with Hector,” add a `createSchema()` method to create a column family definition and a keyspace definition for the schema. Create a

column family definition for a column family named "catalog", a keyspace named HectorKeyspace, and a comparator named ComparatorType.BYTESTYPE.

```
ColumnFamilyDefinition cfDef = HFactory.createColumnFamilyDefinition(
    "HectorKeyspace", "catalog", ComparatorType.BYTESTYPE);
```

Use a replicationFactor of 1 to create a KeyspaceDefinition instance from the preceding column family definition. Specify the strategy class as org.apache.cassandra.locator.SimpleStrategy using the constant ThriftKsDef.DEF_STRATEGY_CLASS.

```
int replicationFactor = 1;
KeyspaceDefinition keyspace = HFactory.createKeyspaceDefinition(
    "HectorKeyspace", ThriftKsDef.DEF_STRATEGY_CLASS,
    replicationFactor, Arrays.asList(cfDef));
```

Add the keyspace definition to the Cluster instance. With blockUntilComplete set to true, the method blocks until schema agreement is received.

```
cluster.addKeyspace(keyspace, true);
```

Adding a keyspace definition to a Cluster instance does not create a keyspace. Having added a keyspace definition, you need to create a keyspace. Add a createKeyspace() method to create a keyspace and invoke the method from the main method. A keyspace is represented with the me.prettyprint.hector.api.Keyspace interface. The HFactory class provides static methods to create a Keyspace instance from a Cluster instance to which a keyspace definition has been added. Invoke the createKeyspace(String keyspace, Cluster cluster) method to create a Keyspace instance with the name HectorKeyspace.

```
private static void createKeyspace() {
    keyspace = HFactory.createKeyspace("HectorKeyspace", cluster);
}
```

Next, create a template and add a createTemplate() method to it. Invoke the method from the main method. Templates provide a reusable construct containing the fields common to all Hector client operations. Create an instance of ThriftColumnFamilyTemplate using a class constructor ThriftColumnFamilyTemplate(Keyspace keyspace, String columnFamily, Serializer<K> keySerializer, Serializer<N> topSerializer). Use the Keyspace instance created earlier and specify the column family name as "catalog".

```
ThriftColumnFamilyTemplate template = new ThriftColumnFamilyTemplate<String,
String>(keyspace, "catalog", StringSerializer.get(), StringSerializer.get());
```

Next, you will migrate the data represented with the DBObject instance retrieved from MongoDB to the column family "catalog" in the keyspace HectorKeyspace. Add a

method called `migrate()` and invoke it from the `main` method. In the `migrate()` method, you will migrate the `DBObject` object retrieved from the MongoDB BSON document to Cassandra. In the Hector API, the `Mutator` class is used to add data. First, you need to create an instance of `Mutator` using the static method `createMutator(Keyspace keyspace, Serializer<K> keySerializer)` in `HFactory`. Supply the `Keyspace` instance previously created and also supply a `StringSerializer` instance.

```
Mutator<String> mutator = HFactory.createMutator(keyspace,
StringSerializer.get());
```

Obtain a `Set` object from the `DBObject` using the `keySet()` method and create an `Iterator` from the `Set` object.

```
Set<String> set = catalog.keySet();
Iterator iter = set.iterator();
```

The `Mutator` class provides the `addInsertion(K key, String cf, HColumn<N, V> c)` method to add an `HColumn` instance and return the `Mutator` instance, which may be used again to add another `HColumn` instance. You can add a series of `HColumn` instances by invoking the `Mutator` instance sequentially. Using the `Iterator` obtained from the key set in the `DBObject` from MongoDB BSON document, you will add multiple columns to a `Mutator` instance using `addInsertion()` invocations in series.

Using the `Iterator` and the `hasNext()` method, obtain a BSON document's key in the key/value pairs as an `Object`. Specify the Key for the Cassandra row as `catalog1`. The column family name is `catalog`. Using the `while` loop, add multiple columns to a `Mutator` instance using `addInsertion()` invocations in series. Add `HColumn<String, String>` instances, which represent columns, using the static method `createStringColumn(String name, String value)`. By iterating over the key set, obtain the column names using the `obj.toString()` method. Obtain the corresponding column value from the `DBObject` instance created from the BSON document using the `catalog.get(obj.toString()).toString()` method invocation.

```
while (iter.hasNext()) {
    Object obj = iter.next();
    mutator = mutator.addInsertion("catalog1", "catalog",
HFactory.createStringColumn(obj.toString(),
catalog.get(obj.toString()).toString()));
}
```

The mutations added to the `Mutator` instance are not sent to the Cassandra server until the `execute()` method is invoked:

```
mutator.execute();
```

The BSON document from MongoDB is migrated to Cassandra. To find the table data created in Cassandra from the MongoDB BSON document, add a `retrieveTableData()` method and invoke it from the `main` method. In the `retrieveTableData()` method, use the `ThriftColumnFamilyTemplate` instance to query multiple columns with the `queryColumns(K key)` method. This queries the columns in the row corresponding to the provided Key value `ColumnFamilyResult` instance. Using the template, query the columns in the row corresponding to "catalog" key.

```
ColumnFamilyResult<String, String> res = template.queryColumns("catalog");
```

Obtain and output the `String` column values in the `ColumnFamilyResult` instance obtained from the preceding query.

```
String journal = res.getString("journal");
String publisher = res.getString("publisher");
String edition = res.getString("edition");
String title = res.getString("title");
String author = res.getString("author");
System.out.println(journal);
System.out.println(publisher);
System.out.println(edition);
System.out.println(title);
System.out.println(author);
```

The `MongoDBToCassandra` class appears in Listing 7.2.

Listing 7.2 The MongoDBToCassandra Class

```
package mongodb;
import java.net.UnknownHostException;
import java.util.Arrays;
import java.util.Iterator;
import java.util.Set;

import me.prettyprint.cassandra.serializers.StringSerializer;
import me.prettyprint.cassandra.service.ThriftKsDef;
import me.prettyprint.hector.api.Cluster;
import me.prettyprint.hector.api.Keyspace;
import me.prettyprint.hector.api.ddl.ColumnFamilyDefinition;
import me.prettyprint.hector.api.ddl.ComparatorType;
```

```
import me.prettyprint.hector.api.ddl.KeyspaceDefinition;
import me.prettyprint.hector.api.exceptions.HectorException;
import me.prettyprint.hector.api.factory.HFactory;
import me.prettyprint.hector.api.mutation.Mutator;
import me.prettyprint.cassandra.service.template.ColumnFamilyResult;
import me.prettyprint.cassandra.service.template.ColumnFamilyTemplate;
import me.prettyprint.cassandra.service.template.ThriftColumnFamilyTemplate;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.DBObject;
import com.mongodb.MongoClient;
import com.mongodb.ServerAddress;
public class MongoDBToCassandra {

    private static DBObject catalog;
    private static Cluster cluster;
    private static Keyspace keyspace;
    private static ColumnFamilyTemplate<String, String> template;
    public static void main(String[] args) {
        try {
            cluster = HFactory.getOrCreateCluster("hector-cluster",
                "localhost:9160");
            KeyspaceDefinition keyspaceDef = cluster
                .describeKeyspace("HectorKeyspace");
            if (keyspaceDef == null) {
                createSchema();
            }
            createKeyspace();
            createTemplate();
            MongoClient mongoClient = new MongoClient(
                Arrays.asList(new ServerAddress
                ("localhost", 27017)));
            DB db = mongoClient.getDB("local");
            DBCollection coll = db.getCollection("catalog");
            catalog = coll.findOne();
            migrate();
            retrieveTableData();
        } catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }
    private static void migrate() {
```

```
        Mutator<String> mutator = HFactory.createMutator(keyspace,
                StringSerializer.get());
        Set<String> set = catalog.keySet();
        Iterator iter = set.iterator();
        while (iter.hasNext()) {
            Object obj = iter.next();
            mutator = mutator.addInsertion(
                    "catalog1",
                    "catalog",
                    HFactory.createStringColumn(obj.toString(),
                            catalog.get(obj.toString()).toString()));
        }
        mutator.execute();
    }
    private static void createSchema() {
        int replicationFactor = 1;
        ColumnFamilyDefinition cfDef =
HFactory.createColumnFamilyDefinition(
                    "HectorKeyspace", "catalog",
        ComparatorType.BYTESTYPE);
        KeyspaceDefinition keyspace = HFactory.createKeyspaceDefinition(
                    "HectorKeyspace", ThriftKsDef.DEF_STRATEGY_CLASS,
                    replicationFactor, Arrays.asList(cfDef));
        cluster.addKeyspace(keyspace, true);
    }
    private static void createKeyspace() {
        keyspace = HFactory.createKeyspace("HectorKeyspace", cluster);
    }
    private static void createTemplate() {
        template = new ThriftColumnFamilyTemplate<String, String>
(keyspace,
                    "catalog", StringSerializer.get(),
StringSerializer.get());
    }
    private static void retrieveTableData() {
        try {
            ColumnFamilyResult<String, String> res = template
                    .queryColumns("catalog1");
            String journal = res.getString("journal");
            String publisher = res.getString("publisher");
            String edition = res.getString("edition");
            String title = res.getString("title");
        }
    }
}
```

```
        String author = res.getString("author");
        System.out.println(journal);
        System.out.println(publisher);
        System.out.println(edition);
        System.out.println(title);
        System.out.println(author);
    } catch (HectorException e) {
    }
}
}
```

Run the MongoDBToCassandra application in the Eclipse IDE. Right-click MongoDBToCassandra and select Run As > Java Application, as shown in Figure 7.9.

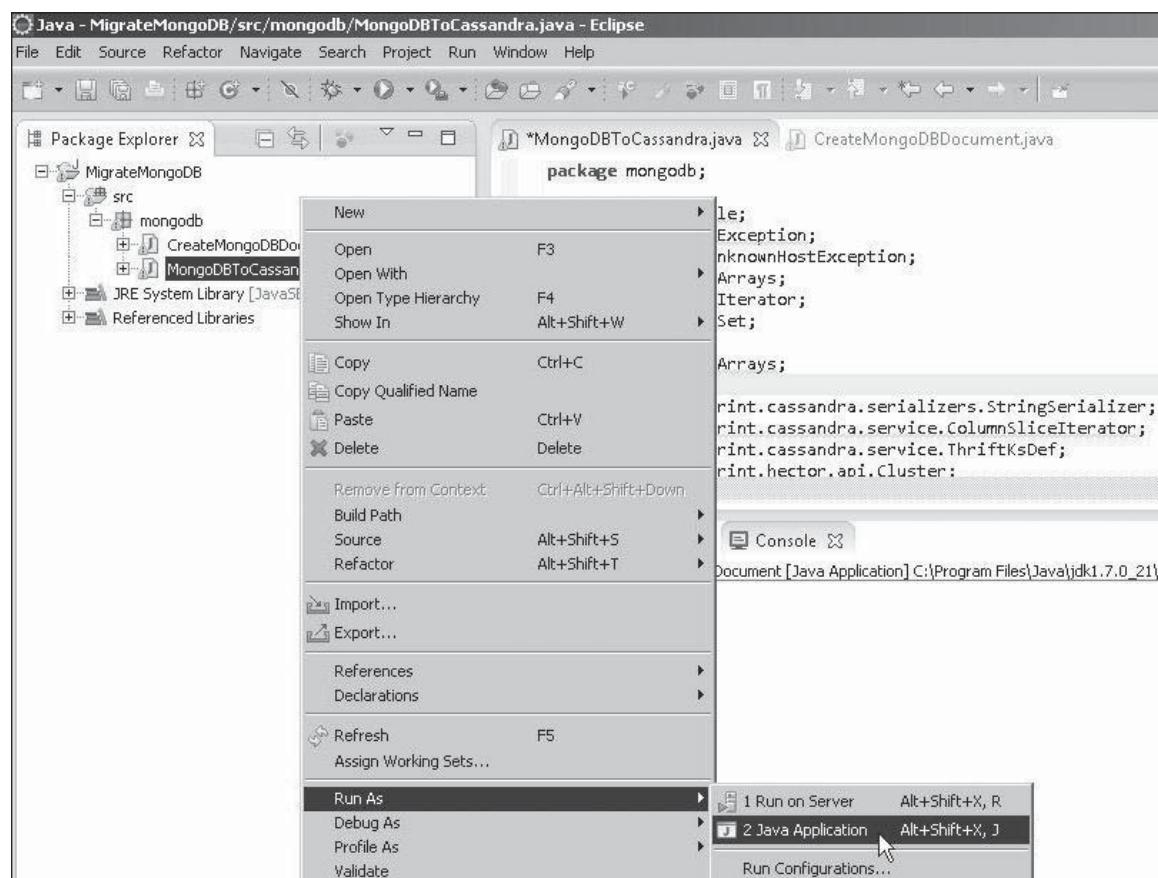


Figure 7.9

Running the MongoDBToCassandra application.

Source: Eclipse Foundation.

The BSON document from the MongoDB server is migrated to Cassandra. Subsequently, the Cassandra table column values created for the migrated BSON document are output in the Eclipse IDE, as shown in Figure 7.10.

A screenshot of the Eclipse IDE interface, specifically the 'Console' tab. The tab bar at the top includes 'JavaDoc', 'Declaration', 'Console', and 'X'. The console window displays the following log output:

```
@ Javadoc Declaration Console X
<terminated> MongoDBToCassandra [Java Application]
log4j:WARN No appenders could be found for logger org.apache.cassandra.reconnection.CassandraHostRetryService).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig.
Oracle Magazine
Oracle Publishing
November December 2013
Engineering as a Service
David A. Kelly
```

Figure 7.10

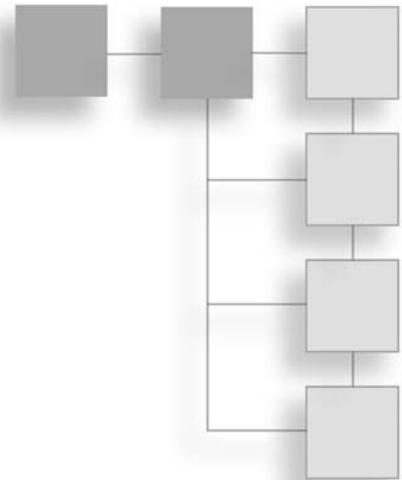
The MongoDB document is migrated to Cassandra.

Source: Eclipse Foundation.

SUMMARY

In this chapter, you migrated a MongoDB BSON document to Apache Cassandra. You used the MongoDB Java driver to access MongoDB and the Hector Java driver to access Cassandra. You used a Java application developed in the Eclipse IDE for the migration. In the next chapter, you will migrate a JSON document from a Couchbase server to a Cassandra database.

CHAPTER 8



MIGRATING COUCHBASE TO CASSANDRA

Couchbase server is one of the leading NoSQL databases, and is based on the JSON document model. Couchbase's document model is different from Cassandra's, as Cassandra is based on the flexible table (column family) data model. But, it is still feasible to migrate a Couchbase Server document to Cassandra. In this chapter, you will migrate a JSON document from Couchbase Server to Cassandra in the Eclipse IDE.

SETTING THE ENVIRONMENT

To set the environment, you must install the following software:

- Couchbase Server Community Edition couchbase-server-community_x86_64_2.0.1. setup.exe from <http://www.couchbase.com/download>. Double-click the EXE file to launch the installer for Couchbase database and install it.
- Couchbase Server Java SDK client library Couchbase-Java-Client-1.2.2.zip file from <http://www.couchbase.com/develop/java/previous> and extract to Couchbase-Java-Client-1.2.2 directory.
- Eclipse IDE for Java EE developers from <http://www.eclipse.org/downloads/>.
- Apache Commons BeanUtils 1.9.0 commons-beanutils-1.9.0-bin.zip file from <http://commons.apache.org/proper/commons-beanutils/download.Beanutils.cgi>. Extract to the commons-beanutils-1.9.0-bin directory.

- Apache Commons Collections 3.2.1 from <http://archive.apache.org/dist/commons/collections/binaries/>. Extract to the commons-collections-3.2.1-bin directory.
- Apache Commons Lang 2.6 commons-lang-2.6-bin.zip from http://commons.apache.org/proper/commons-lang/download_lang.cgi. Extract to the commons-lang-2.6-bin directory.
- Apache Commons Logging 1.1.3 from <http://commons.apache.org/proper/commons-logging/>. Extract to the commons-logging-1.1.3-bin directory.
- Json-lib JAR json-lib-2.4-jdk15.jar from <http://sourceforge.net/projects/json-lib/files/json-lib/>.
- EZMorph ezmorph-1.0.6.jar from <http://sourceforge.net/projects/ezmorph/files/>.
- Hector Java client hector-core-1.1-4.jar or a later version from <http://repo2.maven.org/maven2/org/hectorclient/hector-core/1.1-4/>.
- Apache Cassandra 2.04 or a later version from <http://cassandra.apache.org/download/>. Add C:\Cassandra\apache-cassandra-2.0.4\bin to the PATH variable.

Start Apache Cassandra server with the following command:

```
>cassandra -f
```

CREATING A JAVA PROJECT

You will migrate Couchbase Server to Cassandra using a Java application in the Eclipse IDE. To do so, create a Java project in Eclipse IDE. Follow these steps:

1. Select File > New > Other.
2. In the New dialog box, select Java > Java Project. Then click Next, as shown in Figure 8.1.

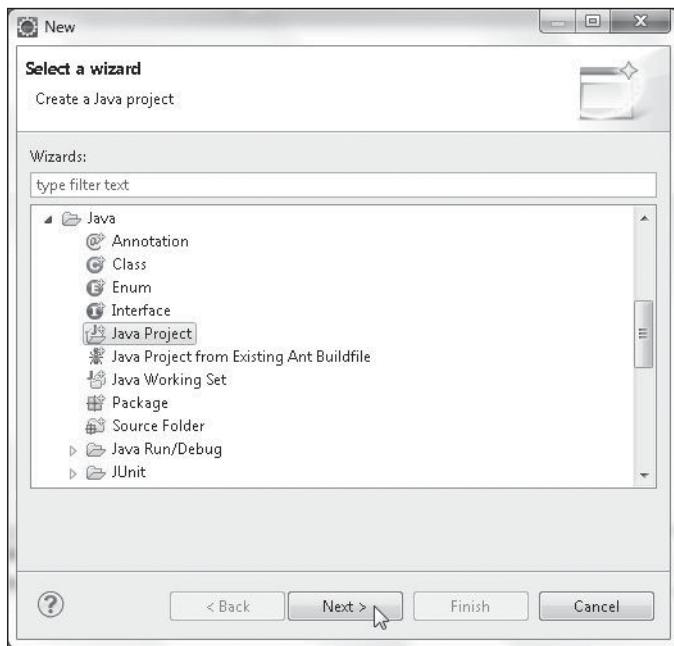


Figure 8.1
Selecting Java > Java Project.
Source: Eclipse Foundation.

3. In the New Java Project wizard, specify a project name (MigrateCouchbaseToCassandra), select JDK 1.7 as the JRE, and click Next, as shown in Figure 8.2.

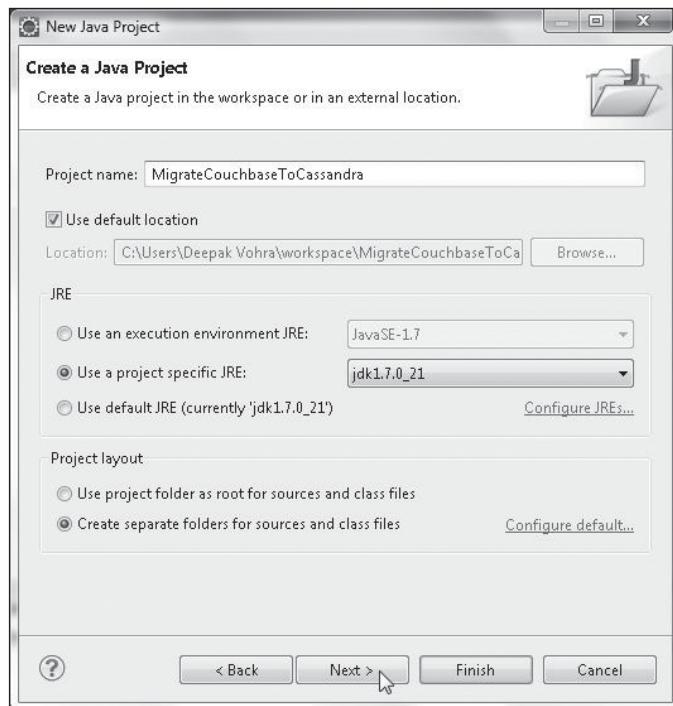


Figure 8.2
Specifying the project name.

Source: Eclipse Foundation.

4. In the Java Settings dialog box, select the Allow Output Folders checkbox and select the default output folder MigrateCouchbaseToCassandra/bin. Then click Finish, as shown in Figure 8.3.

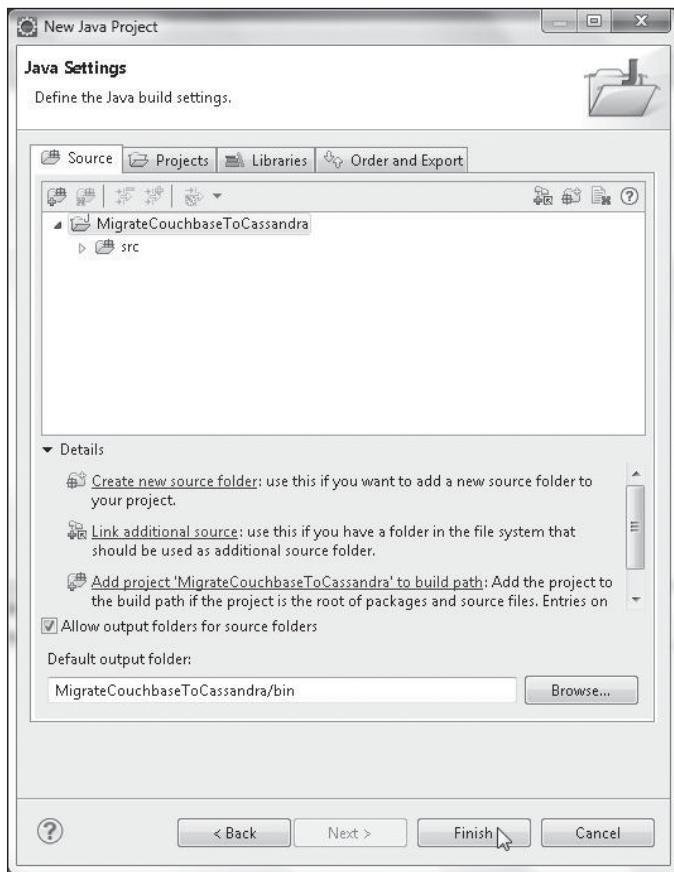


Figure 8.3
Configuring Java settings.
Source: Eclipse Foundation.

5. Next, add two Java classes to the Java project—one to create a JSON document in Couchbase and another to migrate the JSON document to Cassandra. To create the first class, select File > New > Other. Then, in the New dialog box, select Java > Class and click Next, as shown in Figure 8.4.

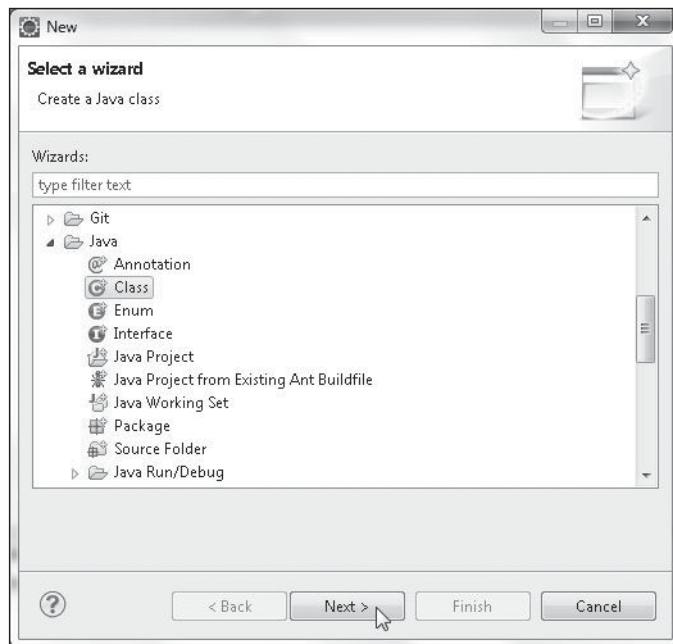


Figure 8.4
Selecting Java > Class.
Source: Eclipse Foundation.

6. In the New Java Class wizard, specify a package (`cassandra`) and the class name (`CreateCouchbaseJSON`). Then click Finish, as shown in Figure 8.5.

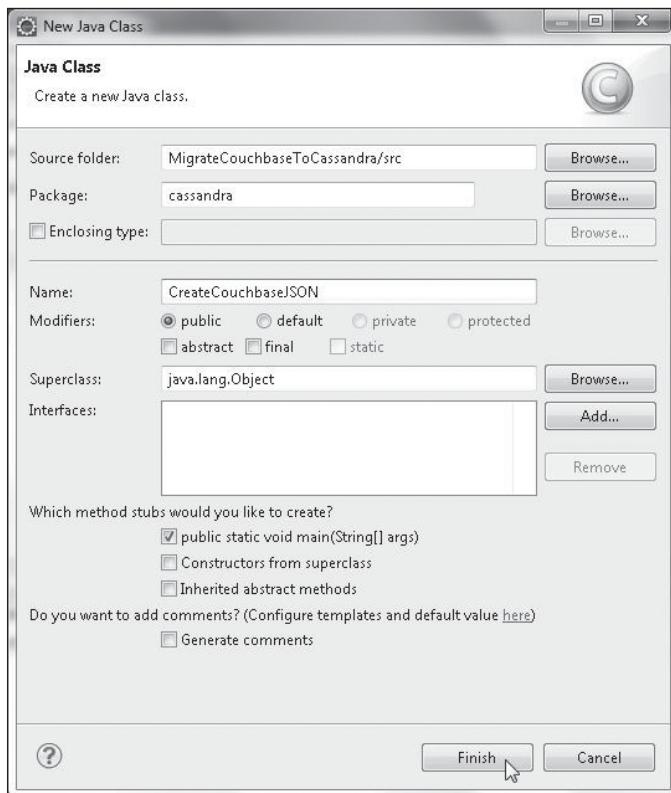
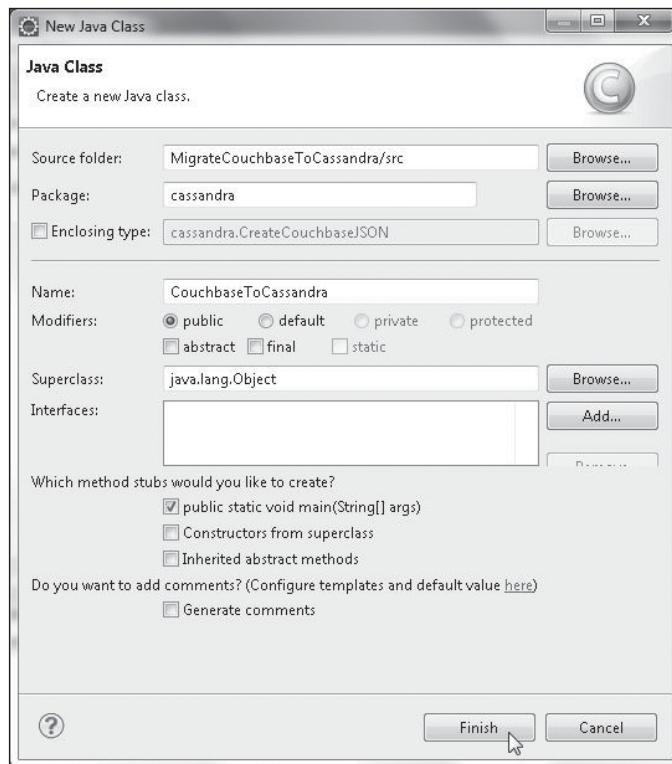


Figure 8.5
Configuring a Java class to store a Couchbase document.

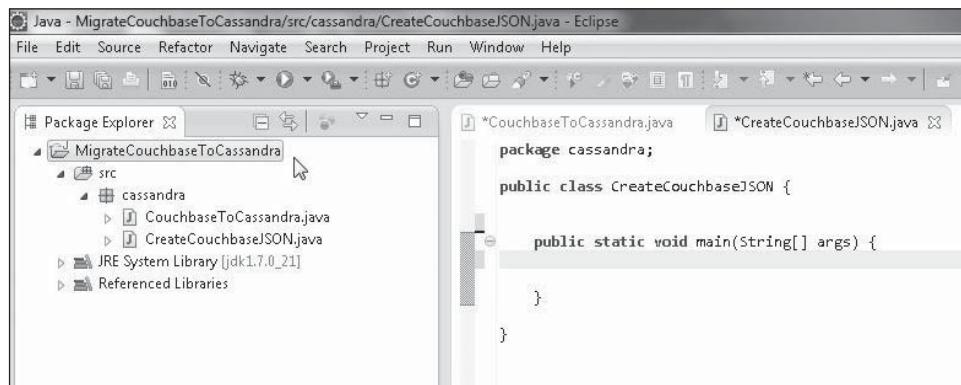
Source: Eclipse Foundation.

7. Repeat steps 5 and 6 to create another Java class, `CouchbaseToCassandra`, as shown in Figure 8.6. The directory structure of the `MigrateCouchbaseToCassandra` Java project with the two classes is shown in Figure 8.7.

**Figure 8.6**

Configuring a Java class to migrate Couchbase to Cassandra.

Source: Eclipse Foundation.

**Figure 8.7**

The directory structure of the MigrateCouchbaseToCassandra Java project.

Source: Eclipse Foundation.

8. Next, you must add some JAR files to the project class path. Add the JAR files listed in Table 8.1, which are from the Cassandra server download, Couchbase Server download, Hector Java client for Cassandra, and some third-party JARs.

Table 8.1 JAR Files for Migration

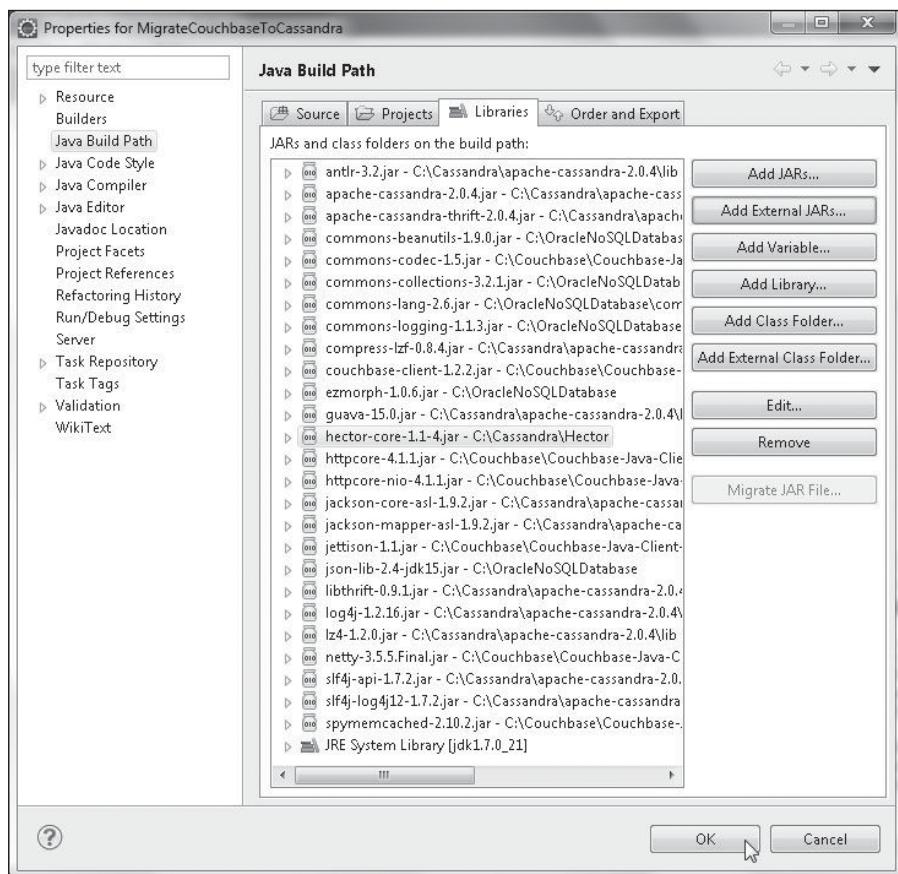
JAR	Description
antlr-3.2.jar	Parser generator for structured text or binary files
apache-cassandra-2.0.4.jar	Apache Cassandra
apache-cassandra-thrift-2.0.4.jar	Apache Cassandra Thrift
compress-lzf-0.8.4.jar	Compression codec for LZF encoding
hector-core-1.1-4.jar	High-level Java client for Cassandra
commons-beanutils-1.9.0	Utility JAR for Java classes developed with the JavaBeans pattern
commons-codec-1.5	Provides implementation of common encoders and decoders
commons-lang-2.6	Provides extra classes for manipulation of Java core classes
commons-logging-1.1.3	Interface for common logging implementations
couchbase-client-1.2.2	Couchbase Server Java client library
ezmorph-1.0.6	Provides conversion from one object to another and used to convert between non-JSON objects and JSON objects
guava-15.0	Google's core libraries used in Java-based projects
httpcore-4.1.1	Provides a set of HTTP transport components to build custom client and server HTTP services
httpcore-nio-4.1.1	HTTP core for the event-driven I/O model based on Java NIO
jackson-core-asl-1.9.2	High-performance JSON processor
jackson-mapper-asl-1.9.2	High-performance data-binding package built on Jacson JSON processor

(Continued)

Table 8.1 JAR Files for Migration (*Continued*)

JAR	Description
jettison-1.1	A collection of Java APIs (STax and DOM) to read and write JSON
json-lib-2.4-jdk15	Java library to transform between beans, maps, collections, Java arrays and XML, and JSON
libthrift-0.9.1.jar	Software framework for scalable cross-language services development
log4j-1.2.16	Logging library for Java
lz4-1.2.0	Fast compression algorithm
netty-3.6.6.Final	NIO client server framework to develop network applications such as protocol servers and clients
slf4j-api-1.7.2	Simple Logging Façade for Java (SLF4J), which serves as an abstraction for various logging frameworks
slf4j-log4j12-1.7.2	An SLF4J abstraction for log4j
spymemcached-2.10.2	Java client for memcached

To add the required JARs, right-click the project node in Package Explorer and select Properties. Then, in the Properties dialog box, select Java Build Path. Finally, click the Add External JARs button to add the external JAR files. The JARs added to the migration project are shown in Figure 8.8.

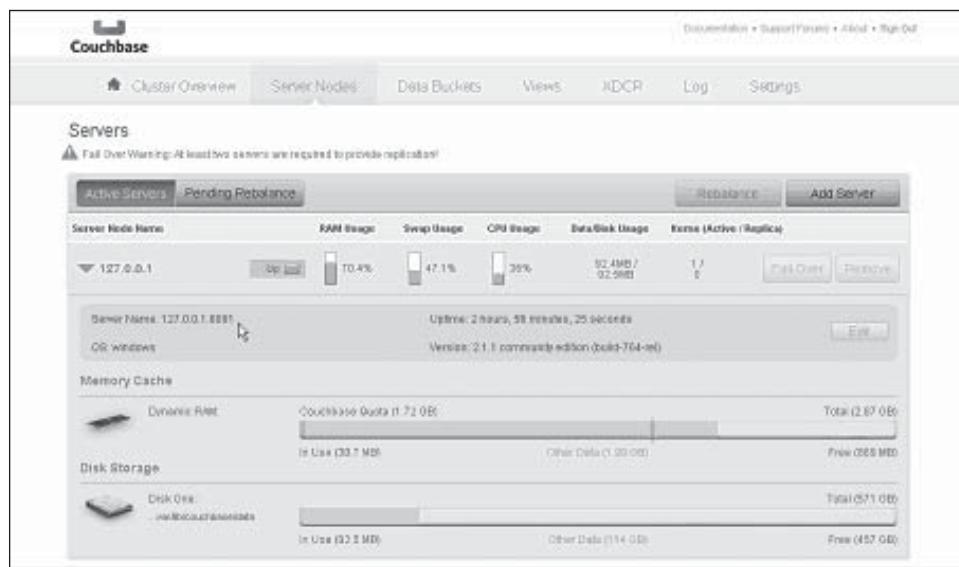
**Figure 8.8**

JAR files in the Java build path.

Source: Eclipse Foundation.

CREATING A JSON DOCUMENT IN COUCHBASE

The `com.couchbase.client.CouchbaseClient` class is the client class for Couchbase Server and is the entry point to access the Couchbase cluster, which may consist of one or more servers. In the `CreateCouchbaseJSON` class, you will use the `CouchbaseClient` class to create and store a JSON document in Couchbase Server. The `CouchbaseClient` class provides three constructors to create an instance. You will use the constructor `CouchbaseClient(java.util.List<java.net.URI> baseList, java.lang.String bucketName, java.lang.String pwd)`, in which `baseList` is the URI List of one or more servers in the Couchbase cluster. To obtain the server URIs, log in to the Couchbase Server cluster and select Server Nodes. Then click the Server Node Name link(s) to obtain the server name, as shown in Figure 8.9.

**Figure 8.9**

Couchbase server name.

Source: Couchbase Inc.

The server name is 127.0.0.1:8091. By default, Couchbase Server uses `http://localhost:8091/pools` to connect to clients. Specify 127.0.0.1 or localhost in the connection URI. Alternatively, use the IPv4 address of the host obtained with the `ipconfig /all` command. Create a `LinkedList<URI>` object and add the URI to the List using the `add()` method:

```
List<URI> uris = new LinkedList<URI>();
uris.add(URI.create("http://192.168.1.71:8091/pools"));
```

Couchbase Server stores documents in data buckets. The `bucketName` parameter specifies the bucket name to use. The default data bucket is called "default". The default username for a data bucket is the same as the bucket name. The "default" bucket does not require a password. Specify password as an empty String.

```
CouchbaseClient couchbaseClient = new CouchbaseClient(uris, "default", "");
Couchbase server stores documents as JSON. Specify the JSON String to store.
String value = "{\"journal\":\"Oracle Magazine\",\"publisher\":\"Oracle Publishing\",\"edition\":\"March April 2013\",\"title\":\"Engineering as a Service\", \"author\":\"David A. Kelly\"}";
```

The `CouchbaseClient` class provides overloaded `set` methods to store/add documents to Couchbase Server. You will use the `set(java.lang.String key,int exp,java.lang.Object value,PersistTo req)` method. The first parameter `key` of type `String` is the

document key. The second parameter `exp` of type `int` is the expiry time. A value of `0` for `exp` makes the document persistent without expiration. The `value` parameter of type `Object` is the JSON value to store. The `req` parameter of type `PersistTo` is the number of nodes the document should be persisted to.

```
OperationFuture<java.lang.Boolean> operationFuture =
couchbaseClient.set("catalog", 0, value, PersistTo.ONE);
```

The `set` method returns an `OperationFuture<java.lang.Boolean>` object that may be used to find if the document got stored or not.

```
if (operationFuture.get().booleanValue()) {
    System.out.println("Set Succeeded");
} else {
    System.err.println("Set failed: " +
operationFuture.getStatus().getMessage());
}
```

Couchbase Server uses a view processor to process documents stored in the server. A view processor takes the unstructured or semi-structured data stored in Couchbase Server, extracts the fields from the document, and indexes the data. As a result, a view of the data stored in the Couchbase data store is created. A view makes it easier to iterate, select, and query the data stored in the server. The view processor relies on the data being stored in JSON format in Couchbase Server. Design documents are used to encapsulate one or more views. The `com.couchbase.client.protocol.views.DesignDocument` class represents a design document. Create a `DesignDocument` using a class constructor:

```
DesignDocument designDoc = new DesignDocument("JSONDocument");
```

The `com.couchbase.client.protocol.views.ViewDesign` class represents a view to be stored and retrieved from the Couchbase data store. The `ViewDesign` class provides two constructors, both of which take a view name and a `map()` function as arguments. One of the constructors also takes a `reduce()` function as an argument. Specify a view name.

```
String viewName = "by_name";
```

A mapping function, `map()`, must be supplied to map the JSON data stored in Couchbase Server and the output results of the view. The first argument to the `map()` function is the JSON document and the second argument is the metadata associated with the JSON document, such as the document name and type. The `map()` function emits rows (zero or more) of information using the invocations of the `emit()` function.

```
String mapFunction = "  function(doc,meta) {\n"
+ "    if (meta.type == 'json') {\n"
+ "      emit(doc.name, doc);\n"
+ "    }\n" + "}" ;
```

Create a ViewDesign object using the view name and the map() function:

```
ViewDesign viewDesign = new ViewDesign(viewName, mapFunction);
```

Add the ViewDesign object to the DesignDocument class invoking the getViews() method on the DesignDocument object to obtain the list of ViewDesigns and subsequently invoking the add method on the List object.

```
designDoc.getViews().add(viewDesign);
```

Store the DesignDocument class in the cluster using the asyncCreateDesignDoc(DesignDocument doc) method in the CouchbaseClient class:

```
HttpFuture<java.lang.Boolean> httpFuture =
couchbaseClient.asyncCreateDesignDoc(designDoc);
```

You can use the HttpFuture object returned to determine whether the design document got stored:

```
if (httpFuture.get().booleanValue()) {
    System.out.println("Design Document Store Succeeded");
} else {
    System.err.println("Design Document Store failed: "
        + httpFuture.getStatus().getMessage());
}
```

The CreateCouchbaseJSON class appears in Listing 8.1.

Listing 8.1 The CreateCouchbaseJSON Class

```
package cassandra;
import java.io.IOException;
import java.io.UnsupportedEncodingException;
import java.net.URI;
import java.util.LinkedList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import net.spy.memcached.PersistTo;
import net.spy.memcached.internal.OperationFuture;
import com.couchbase.client.CouchbaseClient;
import com.couchbase.client.internal.HttpFuture;
import com.couchbase.client.protocol.views.DesignDocument;
import com.couchbase.client.protocol.views.Query;
import com.couchbase.client.protocol.views.Stale;
import com.couchbase.client.protocol.views.View;
```

```
import com.couchbase.client.protocol.views.ViewDesign;
import com.couchbase.client.protocol.views.ViewResponse;

public class CreateCouchbaseJSON {
    private static CouchbaseClient couchbaseClient;
    public static void main(String[] args) {
        try{
            List<URI> uris = new LinkedList<URI>();
            uris.add(URI.create("http://192.168.1.71:8091/pools"));
            CouchbaseClient couchbaseClient = new CouchbaseClient(uris,
"default", "");
            String value = "{\"journal\":\"Oracle Magazine\", \"publisher\":"
"\"Oracle Publishing\", \"edition\":\"March April 2013\", \"title\":"
"\"Engineering as a Service\", \"author\":\"David A. Kelly\"}";
            OperationFuture<java.lang.Boolean> operationFuture =
couchbaseClient
                .set("catalog", 0, value, PersistTo.ONE);
            if (operationFuture.get().booleanValue()) {
                System.out.println("Set Succeeded");
            } else {
                System.err.println("Set failed: "
                    + operationFuture.getStatus().getMessage());
            }
            DesignDocument designDoc = new DesignDocument("JSONDocument");
            String viewName = "by_name";
            String mapFunction = "    function(doc,meta) {\n"
                + "      if (meta.type == 'json') {\n"
                + "        emit(doc.name, doc);\n"
                + "      }\n    }";
            ViewDesign viewDesign = new ViewDesign(viewName, mapFunction);
            designDoc.getViews().add(viewDesign);
            HttpFuture<java.lang.Boolean> httpFuture = couchbaseClient
                .asyncCreateDesignDoc(designDoc);
            if (httpFuture.get().booleanValue()) {
                System.out.println("Design Document Store
Succeeded");
            } else {
                System.err.println("Design Document Store failed: "
                    + httpFuture.getStatus().getMessage());
            }
        } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
        }
    }
}
```

```

        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

To create the JSON document in Couchbase Server, run the CreateCouchbaseJSON.java source file. Right-click the CreateCouchbaseJSON.java file in Package Explorer and select Run As > Java Application, as shown in Figure 8.10. The output from the application, shown in Figure 8.11, indicates that the JSON document is stored in the Couchbase Server and the design document is also stored.

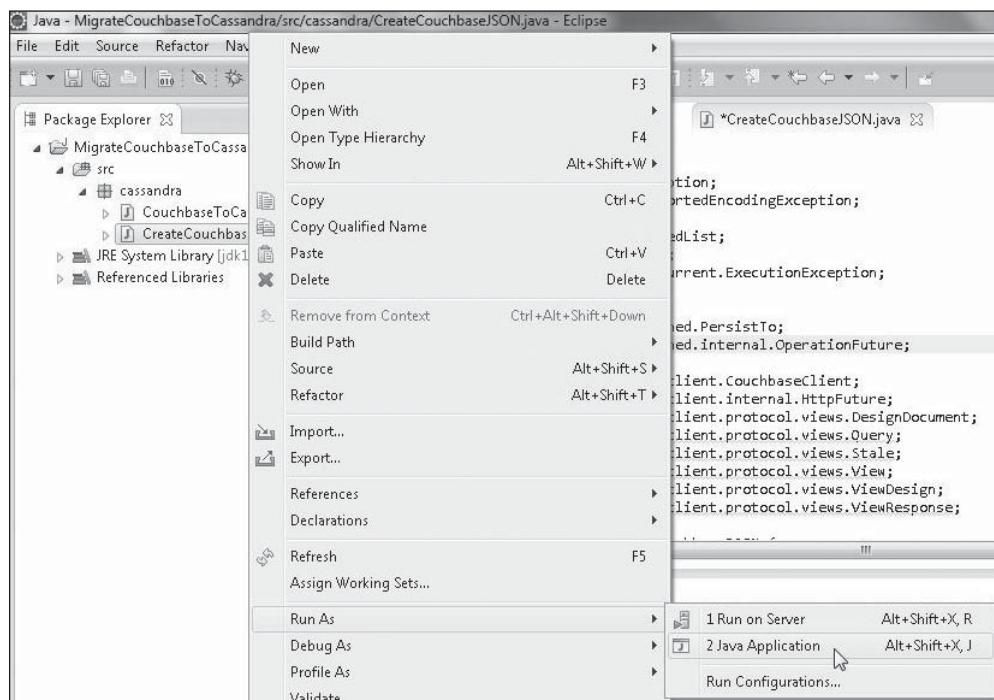


Figure 8.10
Running the CreateCouchbaseJSON application.

Source: Eclipse Foundation.



```

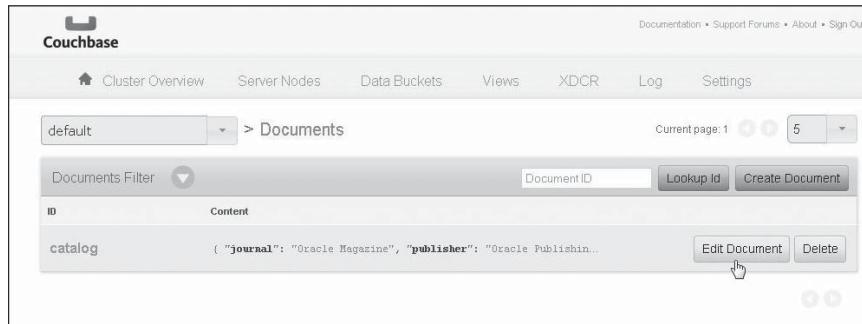
Console [X]
CreateCouchbaseJSON (1) [Java Application] C:\Program Files\Java\jdk1.7.0_21\bin\javaw.exe (Jan 28, 2014 5:10:26 AM)
2014-01-28 05:10:30.272 INFO com.couchbase.client.CouchbaseConnection: Added {QA sa=/192.168.1.71:11210, #Rops=0, #Wops=0, #iq=0, topRops=0}
2014-01-28 05:10:30.344 INFO com.couchbase.client.CouchbaseConnection: Connection state changed for sun.nio.ch.SelectionKeyImpl@24f37f80
2014-01-28 05:10:31.713 INFO com.couchbase.client.ViewConnection: Added dvohra-PC to connect queue
2014-01-28 05:10:31.715 INFO com.couchbase.client.CouchbaseClient: viewmode property isn't defined. Setting viewmode to production mode
Set Succeeded
2014-01-28 05:10:35.398 INFO com.couchbase.client.CouchbaseClient: Creating Design Document:JSONDocument
2014-01-28 05:10:35.418 INFO com.couchbase.client.http.AsyncConnectionManager: Opening new Couchbase HTTP connection
2014-01-28 05:10:35.430 INFO com.couchbase.client.http.AsyncConnectionManager$ConnRequestCallback: dvohra-PC/192.168.1.71:8092 - Session Design Document Store Succeeded

```

Figure 8.11
Output from the CreateCouchbaseJSON application.

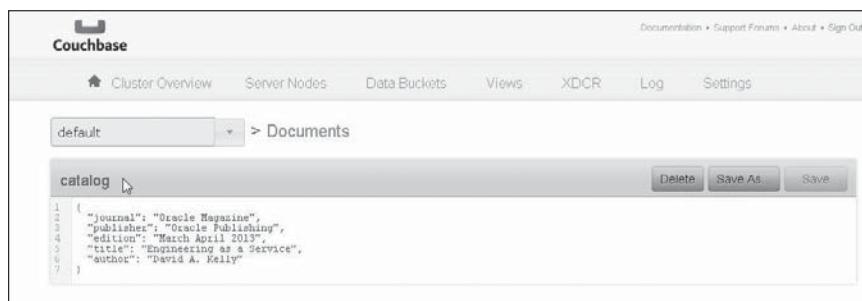
Source: Eclipse Foundation.

Log in to the Couchbase Server Administration Console. Then click Data Buckets and select the default bucket. The documents for the default bucket are listed. The JSON document with the ID catalog is also listed. Click the Edit Document button to edit or display the document, as shown in Figure 8.12. The JSON for the catalog ID documents is displayed, as shown in Figure 8.13.



The screenshot shows the Couchbase Server Administration Console interface. The top navigation bar includes links for Documentation, Support Forums, About, and Sign Out. Below the navigation is a header with Cluster Overview, Server Nodes, Data Buckets, Views, XDCR, Log, and Settings. A dropdown menu shows 'default' selected. The main area displays a list of documents under 'Documents'. A 'Documents Filter' dropdown is set to 'Content'. A table lists one document with the ID 'catalog'. The document content is shown as a JSON object: { "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "March April 2013", "title": "Engineering as a Service", "author": "David A. Kelly" }. Buttons for 'Edit Document' and 'Delete' are visible next to the document entry. A cursor is hovering over the 'Edit Document' button.

Figure 8.12
The document with the ID catalog in the Couchbase Server Administration Console.
Source: Couchbase Inc.



This screenshot shows the same Couchbase Server Administration Console interface as Figure 8.12, but the 'Edit Document' button has been clicked, opening a modal dialog. The dialog displays the JSON document content for the 'catalog' document. The JSON object is identical to the one shown in Figure 8.12: { "journal": "Oracle Magazine", "publisher": "Oracle Publishing", "edition": "March April 2013", "title": "Engineering as a Service", "author": "David A. Kelly" }. The modal has 'Delete', 'Save As', and 'Save' buttons at the bottom. A cursor is hovering over the 'Save' button.

Figure 8.13
Displaying the JSON document added to Couchbase Server in the Couchbase Server Administration Console.
Source: Couchbase Inc.

The `by_name` view is also created. Click Views in the Couchbase Administration Console (see Figure 8.14) to list the `by_name` view. The `by_name` view is displayed.

catalog
<pre> 1 { 2 "journal": "Oracle Magazine", 3 "publisher": "Oracle Publishing", 4 "edition": "March April 2013", 5 "title": "Engineering as a Service", 6 "author": "David A. Kelly" 7 } </pre>

Figure 8.14

Selecting Views in the Couchbase Administration Console.

Source: Couchbase Inc.

Click the Show button to list the view's `map()` and `reduce()` functions, as shown in Figure 8.15. The view code, including the `map()` function, is listed, as shown in Figure 8.16.

Name	Language	Status
_design/JSONDocum...	javascript	<input type="button" value="Compact"/> <input type="button" value="Delete"/> <input type="button" value="Copy to Dev"/>
by_name		<input type="button" value="Show"/>

Figure 8.15

Selecting Views > `by_name` > Show in Couchbase Administration Console.

Source: Couchbase Inc.

The screenshot shows the Couchbase Server interface for managing views. At the top, it displays the path: default > Views > _design/JSONDocument/_view/by_name. Below this, there's a section titled 'catalog' containing two JSON documents. The first document is a journal entry with fields like 'journal', 'publisher', 'edition', 'title', and 'author'. The second document is a meta-document for the view, with fields like 'id', 'rev', 'expiration', 'flags', and 'type'. Underneath, there's a 'VIEW CODE' section divided into 'Map' and 'Reduce' parts. The Map part contains the following JavaScript code:

```
1 function(doc,meta) {  
2   if (meta.type == 'json') {  
3     emit(doc.name, doc);  
4   }  
5 }
```

The Reduce part contains a single line: '1'. At the bottom, there are 'Filter Results' and 'Show Results' buttons, along with a note: 'To see the results of this view, click "Show Results" above.'

Figure 8.11

The view code for the by_name view.

Source: Couchbase Inc.

MIGRATING THE COUCHBASE DOCUMENT TO CASSANDRA

In this section, you will query the JSON document stored earlier in Couchbase Server and migrate the JSON document to the Apache Cassandra database. You will use the CouchbaseToCassandra class to migrate the JSON document from Couchbase Server to the Cassandra database. You added a view encapsulated in a design document to Couchbase Server; you can use this view to query Couchbase Server. The CouchbaseClient class provides the query(AbstractView view, Query query) method to query the server. AbstractView is the abstract superclass to the View class. The Query type parameter represents the type of query to run. But first, you must create an instance of View and an instance of Query to supply as arguments to the query() method. Create an instance of CouchbaseClient as discussed when storing a JSON document in Couchbase Server.

```
List<URI> uris = new LinkedList<URI>();  
uris.add(URI.create("http://192.168.1.71:8091/pools"));  
CouchbaseClient couchbaseClient = new CouchbaseClient(uris, "default", "");
```

Get access to the view stored in the design document in Couchbase Server using the `getView(java.lang.String designDocumentName, java.lang.String viewName)` method, which takes the design document name and view name as arguments.

```
View view = couchbaseClient.getView("JSONDocument", "by_name");
```

Create an instance of `Query` and invoke the `setIncludeDocs(boolean include)` method to include full documents in the result. Optionally, set a limit on the number of documents returned using the `setLimit(int limit)` method.

```
Query query = new Query();
query.setIncludeDocs(true).setLimit(20);
query.setStale(Stale.FALSE);
```

To disallow results from a stale view, invoke the `setStale(Stale stale)` method with the argument as `Stale.FALSE`. Invoke the `query(AbstractView view, Query query)` method using the `View` instance and `Query` instance to obtain a result as a `ViewResponse` object.

```
ViewResponse result = couchbaseClient.query(view, query);
```

Get a Map of key/value records in the `ViewResponse` using the `getMap()` method:

```
java.util.Map<java.lang.String, java.lang.Object> map = result.getMap();
```

Next, you will migrate the resulting Map to the Cassandra database. As discussed in Chapter 1, “Using Cassandra with Hector,” the `me.prettyprint.hector.api.Cluster` interface represents a cluster of Cassandra hosts. To access a Cassandra cluster, first you need to create a `Cluster` instance for a Cassandra cluster. Create a `Cluster` instance using the `getOrCreateCluster(String clusterName, String hostIp)` method as follows:

```
Cluster cluster = HFactory.getOrCreateCluster("migration-cluster",
"localhost:9160");
```

Next, create a schema if not already defined. A schema consists of a column family definition and a keyspace definition. Use the `describeKeyspace` method in `Cluster` to obtain a `KeyspaceDefinition` object for `MigrationKeyspace` keyspace. If the keyspace definition object is null, invoke a `createSchema()` method to create a schema.

```
KeyspaceDefinition keyspaceDef = cluster.describeKeyspace("MigrationKeyspace");
if (keyspaceDef == null) {
    createSchema();
}
```

As discussed in Chapter 1, add a `createSchema()` method to create a column family definition and a keyspace definition for the schema. Create a column family definition for a

column family named "catalog", a keyspace named MigrationKeyspace, and a comparator named ComparatorType.BYTESTYPE.

```
ColumnFamilyDefinition cfDef = HFactory.createColumnFamilyDefinition(
    "MigrationKeyspace", "catalog", ComparatorType.BYTESTYPE);
```

Using a replicationFactor of 1, create a KeyspaceDefinition instance from the preceding column family definition. Specify the strategy class as org.apache.cassandra.locator.SimpleStrategy using the constant ThriftKsDef.DEF_STRATEGY_CLASS.

```
int replicationFactor = 1;
KeyspaceDefinition keyspace = HFactory.createKeyspaceDefinition(
    "MigrationKeyspace", ThriftKsDef.DEF_STRATEGY_CLASS,
    replicationFactor, Arrays.asList(cfDef));
```

Add the keyspace definition to the Cluster instance. With blockUntilComplete set to true, the method blocks until schema agreement is received.

```
cluster.addKeyspace(keyspace, true);
```

Adding a keyspace definition to a Cluster instance does not create a keyspace. Having added a keyspace definition, you need to create a keyspace. Add a createKeyspace() method to create a keyspace and invoke the method from the main method. A keyspace is represented with the me.prettyprint.hector.api.Keyspace interface. The HFactory class provides static methods to create a Keyspace instance from a Cluster instance to which a keyspace definition has been added. Invoke the method createKeyspace (String keyspace, Cluster cluster) to create a Keyspace instance with the name MigrationKeyspace.

```
private static void createKeyspace() {
    keyspace = HFactory.createKeyspace("MigrationKeyspace", cluster);
}
```

Next, create a template and add a createTemplate() method to it. Invoke the method from the main method. Templates provide reusable constructs containing the fields common to all Hector client operations. Create an instance of ThriftColumnFamilyTemplate using a class constructor ThriftColumnFamilyTemplate(Keyspace keyspace, String columnFamily, Serializer<K> keySerializer, Serializer<N> topSerializer). Use the Keyspace instance created earlier and specify the column family name as "catalog".

```
ThriftColumnFamilyTemplate template = new ThriftColumnFamilyTemplate<String,
String>(keyspace, "catalog", StringSerializer.get(), StringSerializer.get());
```

Next, you will migrate the data retrieved from Couchbase Server to the column family "catalog" in the keyspace MigrationKeyspace. Add a method called migrate() and

invoke it from the `main` method. In the `migrate()` method, you will migrate the `Map` object retrieved from the Couchbase JSON document to Cassandra. In the Hector API, the `Mutator` class is used to add data. First, you need to create an instance of `Mutator` using the static method `createMutator(Keyspace keyspace, Serializer<K> keySerializer)` in `HFactory`. Supply the `Keyspace` instance previously created and also supply a `StringSerializer` instance.

```
Mutator<String> mutator = HFactory.createMutator(keyspace,
StringSerializer.get());
```

Next, iterate over the JSON document result `Map` obtained earlier using an enhanced `for` loop.

```
for (java.util.Map.Entry<String, Object> entry : map.entrySet()) {  
}
```

Next, add code within the `for` loop. Output the key/value pair(s) in the `Map` using the `java.util.Map.Entry.getKey()` and corresponding `getValue()` methods.

```
System.out.println(entry.getKey());  
System.out.println(entry.getValue());
```

An unordered collection of name/value pairs, which constitute a JSON document, is represented by the `net.sf.json.JSONObject` class. Its format is a string enclosing a JSON name/value pair using {}, with , separating the name/value pairs. The values in name/value pairs may be of one of the following types: `String`, `Boolean`, `JSONArray`, `JSONObject`, `Number`, or `JSONNull`. Create a `JSONObject` instance from the JSON object in the result `Map` using the `JSON-lib`. The `net.sf.json.JSONSerializer` class is used to transform Java objects to JSON and back. Invoke the `toJSON(Object object)` method to create a `JSONObject` instance from the JSON object in the result `Map`.

```
JSONObject json = (JSONObject) JSONSerializer.toJSON(entry  
.getValue().toString());
```

Obtain a `Set` object from the `JSONObject` and create an `Iterator` from the `Set` object.

```
Set set = json.keySet();  
Iterator iter = set.iterator();
```

The `Mutator` class provides the `addInsertion(K key, String cf, HColumn<N, V> c)` method to add an `HColumn` instance and return the `Mutator` instance, which may be used again to add another `HColumn` instance. You can add a series of `HColumn` instances by invoking the `Mutator` instance sequentially. Using the `Iterator` obtained from the result `Map` from

Couchbase JSON document, you will add multiple columns to a Mutator instance using addInsertion invocations in series.

Using the Iterator and the hasNext() method, obtain the JSON document as an Object. Obtain the key for the JSON document from the MapEntry object by invoking the getKey().toString() methods. The column family name is "catalog". Using the while loop, add multiple columns to a Mutator instance using addInsertion invocations in series. Add HColumn<String, String> instances, which represent columns using the static method createStringColumn(String name, String value). By iterating over the key set, obtain the column names using the obj.toString() method. Obtain the corresponding column values from the JSONObject instance created from the JSON document using the json.get(obj.toString()).toString() method invocation.

```
while (iter.hasNext()) {
    Object obj = iter.next();
    mutator = mutator.addInsertion(
        entry.getKey().toString(),
        "catalog",
        HFactory.createStringColumn(obj.toString(),
            json.get(obj.toString()).toString()));
}
```

The mutations added to the Mutator instance are not sent to the Cassandra server until the execute() method is invoked.

```
System.out.println(mutator.execute().getHostUsed());
```

The JSON document from Couchbase Server is migrated to Cassandra. To find the table data created in Cassandra from the Couchbase JSON document, add a retrieveTableData() method and invoke it from the main method. In the retrieveTableData() method, use the ThriftColumnFamilyTemplate instance to query multiple columns with the queryColumns (K key) method, which queries the columns in the row corresponding to the provided Key value ColumnFamilyResult instance. Using the template, query the columns in the row corresponding to the "catalog" key.

```
ColumnFamilyResult<String, String> res = template.queryColumns("catalog");
```

Obtain and output the String column values in the ColumnFamilyResult instance obtained from the preceding query.

```
String journal = res.getString("journal");
String publisher = res.getString("publisher");
String edition = res.getString("edition");
```

```
String title = res.getString("title");
String author = res.getString("author");
System.out.println(journal);
System.out.println(publisher);
System.out.println(edition);
System.out.println(title);
System.out.println(author);
```

The CouchbaseToCassandra class appears in Listing 8.2.

Listing 8.2 The CouchbaseToCassandra Class

```
package cassandra;

import java.io.IOException;
import java.net.URI;
import java.util.Arrays;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Set;
import me.prettyprint.hector.api.Keyspace;
import net.sf.json.JSONObject;
import net.sf.json.JSONSerializer;
import me.prettyprint.cassandra.serializers.StringSerializer;
import me.prettyprint.cassandra.service.ThriftKsDef;
import me.prettyprint.cassandra.service.template.ColumnFamilyResult;
import me.prettyprint.cassandra.service.template.ColumnFamilyTemplate;
import me.prettyprint.cassandra.service.template.ThriftColumnFamilyTemplate;
import me.prettyprint.hector.api.Cluster;
import com.couchbase.client.CouchbaseClient;
import com.couchbase.client.protocol.views.Query;
import com.couchbase.client.protocol.views.Stale;
import com.couchbase.client.protocol.views.View;
import com.couchbase.client.protocol.views.ViewResponse;
import me.prettyprint.hector.api.ddl.ColumnFamilyDefinition;
import me.prettyprint.hector.api.ddl.ComparatorType;
import me.prettyprint.hector.api.ddl.KeyspaceDefinition;
import me.prettyprint.hector.api.exceptions.HectorException;
import me.prettyprint.hector.api.factory.HFactory;
import me.prettyprint.hector.api.mutation.Mutator;

public class CouchbaseToCassandra {
    private static CouchbaseClient couchbaseClient;
    private static Cluster cluster;
```

```
private static Keyspace keyspace;
private static ColumnFamilyTemplate<String, String> template;
private static java.util.Map<java.lang.String, java.lang.Object> map;
public static void main(String[] args) {
    List<URI> uris = new LinkedList<URI>();
    uris.add(URI.create("http://192.168.1.71:8091/pools"));
    try {
        couchbaseClient = new CouchbaseClient(uris, "default", "");
        View view = couchbaseClient.getView("JSONDocument",
"by_name");
        Query query = new Query();
        query.setIncludeDocs(true).setLimit(20);
        query.setStale(Stale.FALSE);
        ViewResponse result = couchbaseClient.query(view, query);
        map = result.getMap();
        cluster = HFactory.getOrCreateCluster("migration-cluster",
"localhost:9160");
        KeyspaceDefinition keyspaceDef = cluster
            .describeKeyspace("MigrationKeyspace");
        if (keyspaceDef == null) {
            createSchema();
        }
        createKeyspace();
        createTemplate();
        migrate();
        retrieveTableData();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
private static void migrate() {
    Mutator<String> mutator = HFactory.createMutator(keyspace,
        StringSerializer.get());
    for (java.util.Map.Entry<String, Object> entry : map.entrySet()) {
        System.out.println(entry.getKey());
        System.out.println(entry.getValue());
        JSONObject json = (JSONObject) JSONSerializer.toJSON(entry
            .getValue().toString());
        Set set = json.keySet();
        Iterator iter = set.iterator();
        while (iter.hasNext()) {
            Object obj = iter.next();
```

```
        mutator = mutator.addInsertion(
            entry.getKey().toString(),
            "catalog",
            HFactory.createStringColumn(obj.toString(),
                json.get(obj.toString()).toString()));
    }
}
System.out.println(mutator.execute().getHostUsed());
}
private static void createSchema() {
    int replicationFactor = 1;
    ColumnFamilyDefinition cfDef = HFactory.createColumnFamily
Definition(
        "MigrationKeyspace", "catalog", ComparatorType.BYTESTYPE);
    KeyspaceDefinition keyspace = HFactory.createKeyspaceDefinition(
        "MigrationKeyspace", ThriftKsDef.DEF_STRATEGY_CLASS,
        replicationFactor, Arrays.asList(cfDef));
    cluster.addKeyspace(keyspace, true);
}
private static void createKeyspace() {
    keyspace = HFactory.createKeyspace("MigrationKeyspace", cluster);
}
private static void createTemplate() {
    template = new ThriftColumnFamilyTemplate<String, String>
(keyspace,
    "catalog", StringSerializer.get(),
StringSerializer.get());
}
private static void retrieveTableData() {
    try {
        ColumnFamilyResult<String, String> res = template
            .queryColumns("catalog");
        String journal = res.getString("journal");
        String publisher = res.getString("publisher");
        String edition = res.getString("edition");
        String title = res.getString("title");
        String author = res.getString("author");
        System.out.println(journal);
        System.out.println(publisher);
        System.out.println(edition);
        System.out.println(title);
        System.out.println(author);
    }
}
```

```
        } catch (HectorException e) {  
    }  
}
```

Run the CouchbaseToCassandra application in the Eclipse IDE. Right-click Couchbase-ToCassandra and select Run As > Java Application, as shown in Figure 8.17.

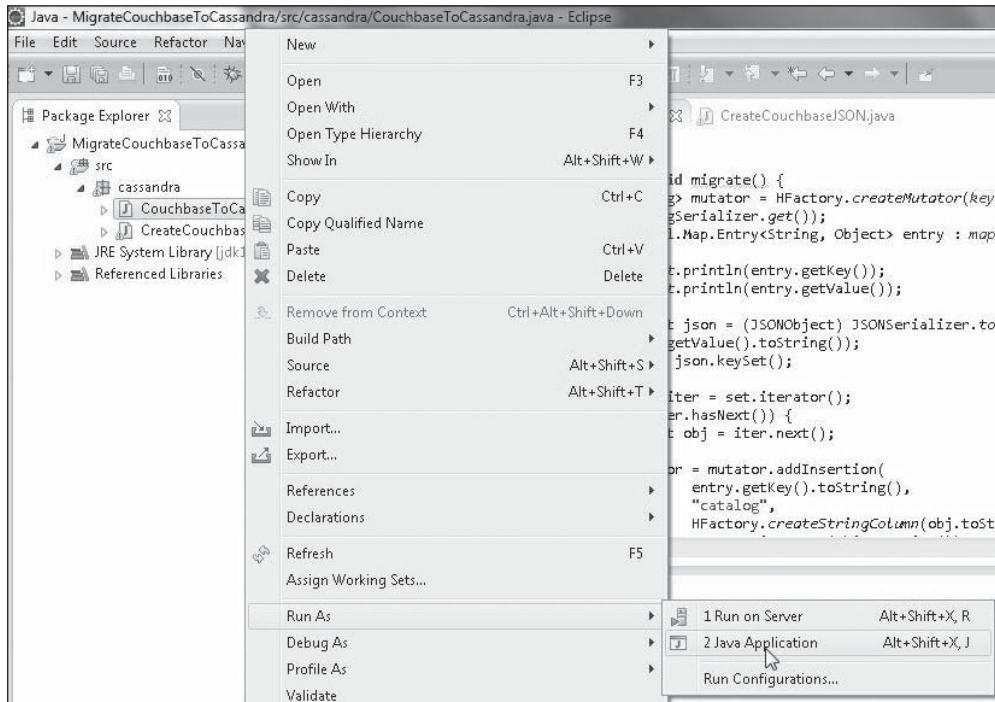


Figure 8.17
Running the CouchbaseToCassandra application.

Source: Eclipse Foundation.

The JSON document from Couchbase Server is migrated to Cassandra. Subsequently, the Cassandra table created for the migrated JSON document is output in the Eclipse IDE, as shown in Figure 8.18.



The screenshot shows a Java application window titled "Console". The log output is as follows:

```
CouchbaseToCassandra [Java Application] C:\Program Files\Java\jdk1.7.0_21\bin\javaw.exe (Jan 28, 2014 6:34:13 AM)
2014-01-28 06:34:14.114 INFO com.couchbase.client.CouchbaseConnection: Connection state changed for sun.nio.ch.SelectionKeyImpl@3d35cda0
2014-01-28 06:34:14.149 INFO com.couchbase.client.ViewConnection: Added dvohra-PC to connect queue
2014-01-28 06:34:14.151 INFO com.couchbase.client.CouchbaseClient: viewmode property isn't defined. Setting viewmode to production mode
2014-01-28 06:34:14.353 INFO com.couchbase.client.http.AsyncConnectionManager: Opening new Couchbase HTTP connection
2014-01-28 06:34:14.385 INFO com.couchbase.client.http.AsyncConnectionManager$ConnRequestCallback: dvohra-PC/192.168.1.71:8092 - Session
log4j:WARN No appenders could be found for logger (me.prettyprint.cassandra.connection.CassandraHostRetryService).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
catalog
{"journal":"Oracle Magazine","publisher":"Oracle Publishing","edition":"March April 2013","title":"Engineering as a Service","author":"David A. Kelly"
Oracle Magazine
Oracle Publishing
March April 2013
Engineering as a Service
David A. Kelly
```

Figure 8.18
JSON data migrated from Couchbase to Cassandra.

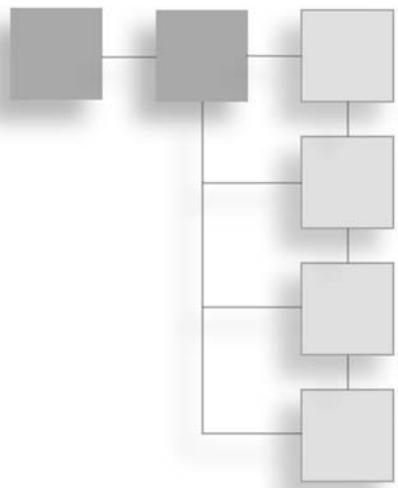
Source: Eclipse Foundation.

SUMMARY

In this chapter, you migrated a JSON document from Couchbase Server to a Cassandra database table. First you created a JSON document in Couchbase. Then you used a Java client for Couchbase to access the Couchbase database and get the JSON document. Finally, you used the Hector Java client to connect to Cassandra and transfer the JSON data got from Couchbase to the Cassandra database. The next chapter discusses using Cassandra with Kundera.

PART IV

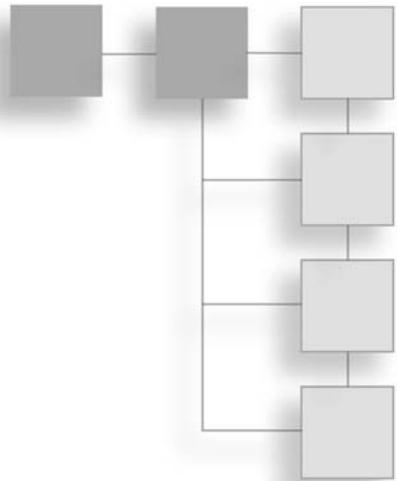
JAVA EE



This page intentionally left blank

CHAPTER 9

USING CASSANDRA WITH KUNDERA



The Java Persistence API (JPA) is the Java API for persistence management and object/relational mapping in a Java EE/Java SE environment with which a Java domain model is used to manage a relational database. JPA also provides a query language API with the Query interface for static and dynamic queries. JPA is designed primarily for relational databases.

Kundera is a JPA 2.0-compliant object–data store mapping library for NoSQL data stores. Kundera also supports relational databases and provides NoSQL data store–specific configuration for Apache Cassandra and some other NoSQL databases, including HBase and MongoDB. Using the Kundera library in the domain model, a NoSQL database can be accessed using the JPA. In this chapter, you will access Apache Cassandra with Kundera and run CRUD operations on Cassandra.

SETTING THE ENVIRONMENT

To set the environment, you must install the following software:

- The Kundera library for Apache Cassandra `kundera-cassandra-2.2.1-jar-with-dependencies.jar` from <https://github.com/impetus-opensource/Kundera/downloads>.
- A persistence framework including support for Java Persistence (JPA) 2.0 JSR 317–EclipseLink 2.4.2 from <http://www.eclipse.org/eclipselink/downloads/index.php#242>. Extract the `eclipselink-2.4.2.v20130514-5956486.zip` file to a directory.

- An implementation of JPA 2.0 eclipselink-2.4.2.jar from <http://repo1.maven.org/maven2/org/eclipse/persistence/eclipselink/2.4.2/eclipselink-2.4.2.jar>.
- Apache Cassandra 2.04 or a later version from <http://cassandra.apache.org/download/>.

Later versions than those listed may also be used.

You need to create a keyspace for object/relational mapping using Kundera. In Cassandra-Cli, run the following command to create a keyspace called Kundera using a replica placement strategy `org.apache.cassandra.locator.SimpleStrategy` and a `replication_factor` of 1.

```
CREATE KEYSPACE Kundera
```

```
with placement_strategy = 'org.apache.cassandra.locator.SimpleStrategy'
and strategy_options = {replication_factor:1};
```

Next, run the following command in Cassandra-Cli to use the Kundera keyspace:

```
use Kundera;
```

The output from the Cassandra-Cli commands is shown in Figure 9.1.

```
C:\>cassandra-cli
Starting Cassandra Client
Connected to: "Test Cluster" on 127.0.0.1/9160
Welcome to Cassandra CLI version 2.0.4

The CLI is deprecated and will be removed in Cassandra 3.0. Consider migrating
to cqlsh.
CQL is fully backwards compatible with Thrift data; see http://www.datastax.com/
dev/blog/thrift-to-cql3

Type 'help;' or '?' for help.
Type 'quit;' or 'exit;' to quit.

[default@unkn0wn] CREATE KEYSPACE Kundera
...     with placement_strategy = 'org.apache.cassandra.locator.SimpleStrategy'
...         and strategy_options = {replication_factor:1};
5454b07c-bf90-3519-b3b2-5ce7241ff400
[default@unkn0wn] use Kundera;
Authenticated to keyspace: Kundera
```

Figure 9.1

Creating a keyspace in Cassandra.

Source: Microsoft Corporation.

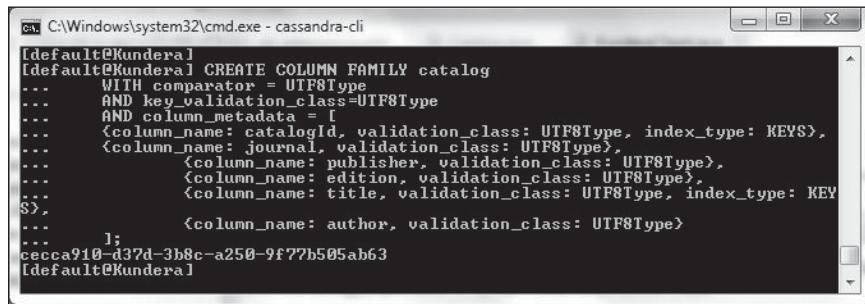
You also need to create a column family for object/relational persistence. Run the following command in Cassandra-Cli to create a column family called `catalog`:

```
CREATE COLUMN FAMILY catalog
WITH comparator = UTF8Type
AND key_validation_class=UTF8Type
```

```
AND column_metadata = [
{column_name: catalogId, validation_class: UTF8Type, index_type: KEYS},
{column_name: journal, validation_class: UTF8Type},
    {column_name: publisher, validation_class: UTF8Type},
    {column_name: edition, validation_class: UTF8Type},
    {column_name: title, validation_class: UTF8Type, index_type: KEYS},
    {column_name: author, validation_class: UTF8Type}
];

```

The output from the command is shown in Figure 9.2.



The screenshot shows a command-line interface window titled "cmd C:\Windows\system32\cmd.exe - cassandra-cli". The window displays the following text:

```
[default@Kundera]
[default@Kundera] CREATE COLUMN FAMILY catalog
...   WITH comparator = UTF8Type
...   AND key_validation_class=UTF8Type
...   AND column_metadata = [
...     {column_name: catalogId, validation_class: UTF8Type, index_type: KEYS},
...     {column_name: journal, validation_class: UTF8Type},
...       {column_name: publisher, validation_class: UTF8Type},
...       {column_name: edition, validation_class: UTF8Type},
...       {column_name: title, validation_class: UTF8Type, index_type: KEY
$},
...         {column_name: author, validation_class: UTF8Type}
...   ];
cecca910-d37d-3b8c-a250-9f77b505ab63
[default@Kundera]
```

Figure 9.2

Creating a column family in Cassandra.

Source: Microsoft Corporation.

CREATING A JPA PROJECT IN ECLIPSE

In this section, you will create a JPA project in the Eclipse IDE for the Kundera Cassandra application. Follow these steps:

1. Select File > New > Other.
2. In the New dialog box, select JPA > JPA Project. Then click Next, as shown in Figure 9.3.

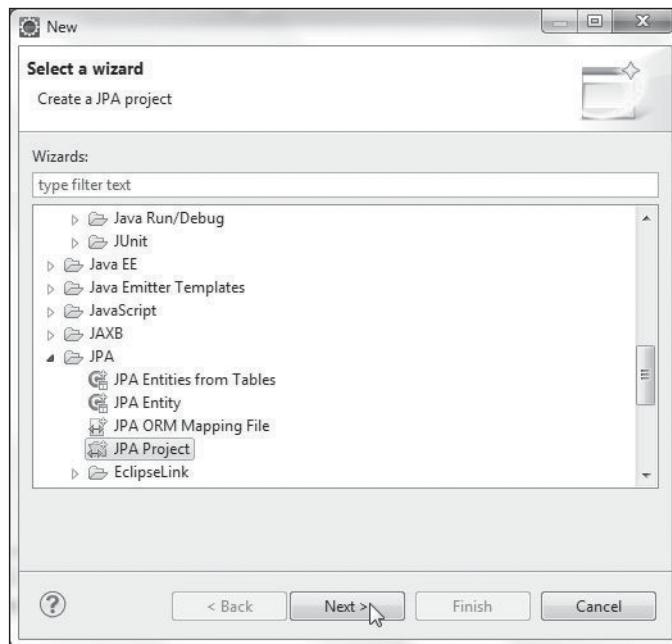


Figure 9.3
Selecting JPA > JPA Project.
Source: Eclipse Foundation.

3. In the New JPA Project wizard, specify a project name (Kundera), choose a project location, select JDK 1.7 as the target runtime, and 2.0 as the JPA version. In the Configuration drop-down list, select Default Configuration for jdk1.7.0_21 and click Next, as shown in Figure 9.4.

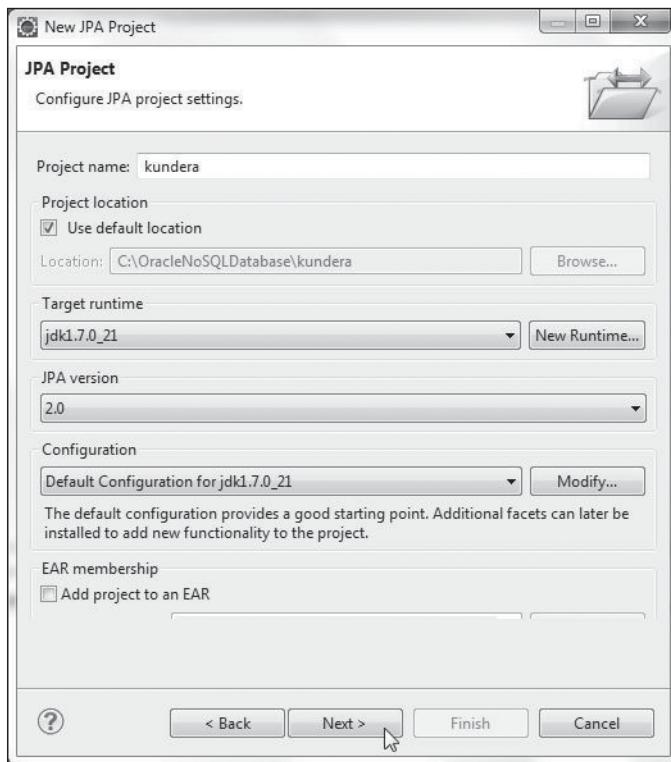


Figure 9.4
Configuring the JPA project.
Source: Eclipse Foundation.

4. In the Java Settings dialog box, choose `src` in the Source Folders on Build Path box and set the default output folder to `build\classes`. These are also the default Java settings. Then click Next, as shown in Figure 9.5.

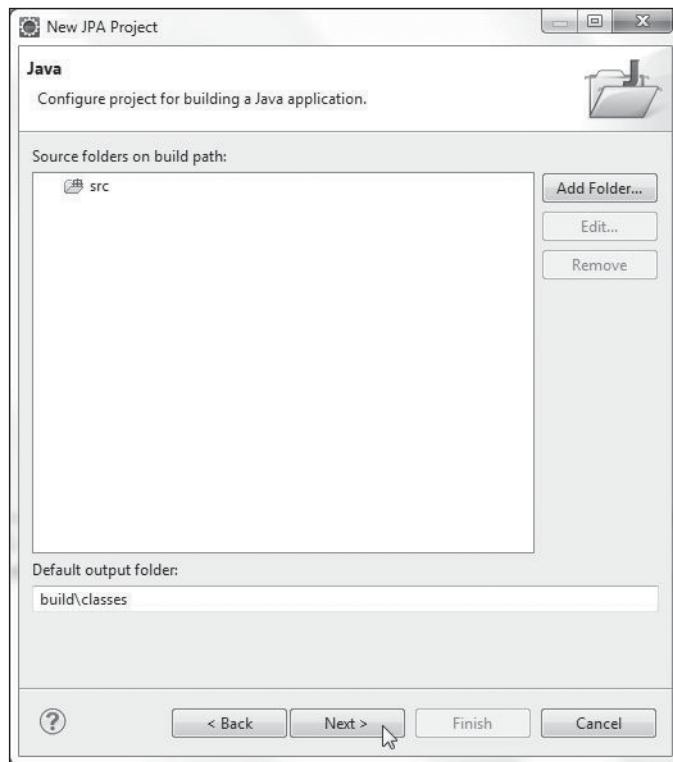


Figure 9.5
Configuring Java settings.
Source: Eclipse Foundation.

5. Configure a JPA facet. In the Platform drop-down list, choose EclipseLink 2.4.x. In the Type drop-down list under JPA Implementation, choose User Library. Then click the Manage Libraries button to create a new user library, as shown in Figure 9.6.

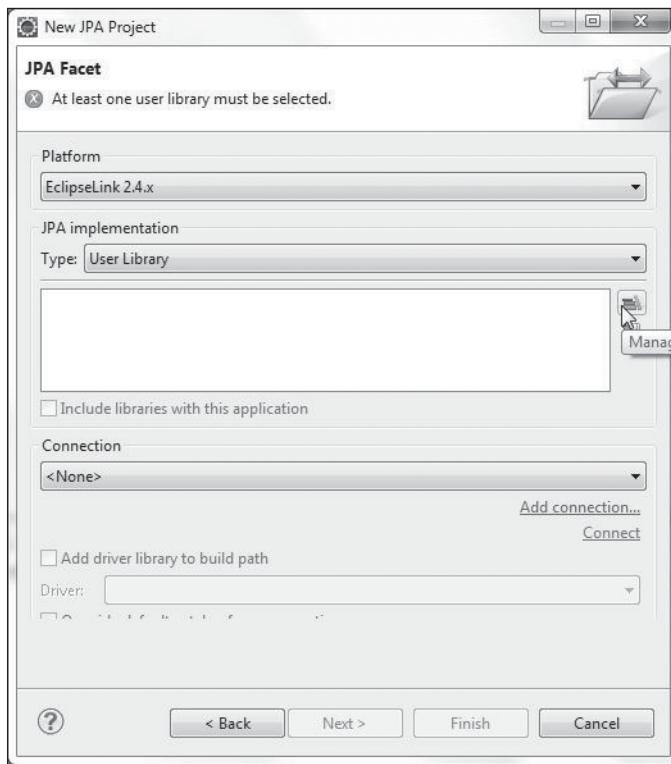


Figure 9.6
Creating a new user library.
Source: Eclipse Foundation.

6. Choose Preferences > User Libraries. Then click New to create a new user library for EclipseLink 2.4. In New User Library dialog box, specify a user library name (EclipseLink2.4) and click OK, as shown in Figure 9.7.

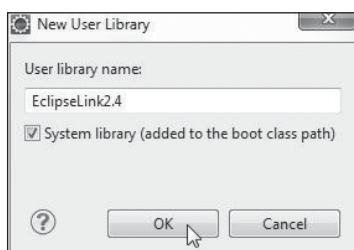


Figure 9.7
Specifying new user library name.
Source: Eclipse Foundation.

7. The EclipseLink2.4 user library is created. Click the Add External JARs button, shown in Figure 9.8, to add the javax.persistence_2.0.5.v201212031355.jar file from the jpa subfolder of the \\eclipselink-2.4.2.v20130514-5956486\\eclipselink\\jlib directory.

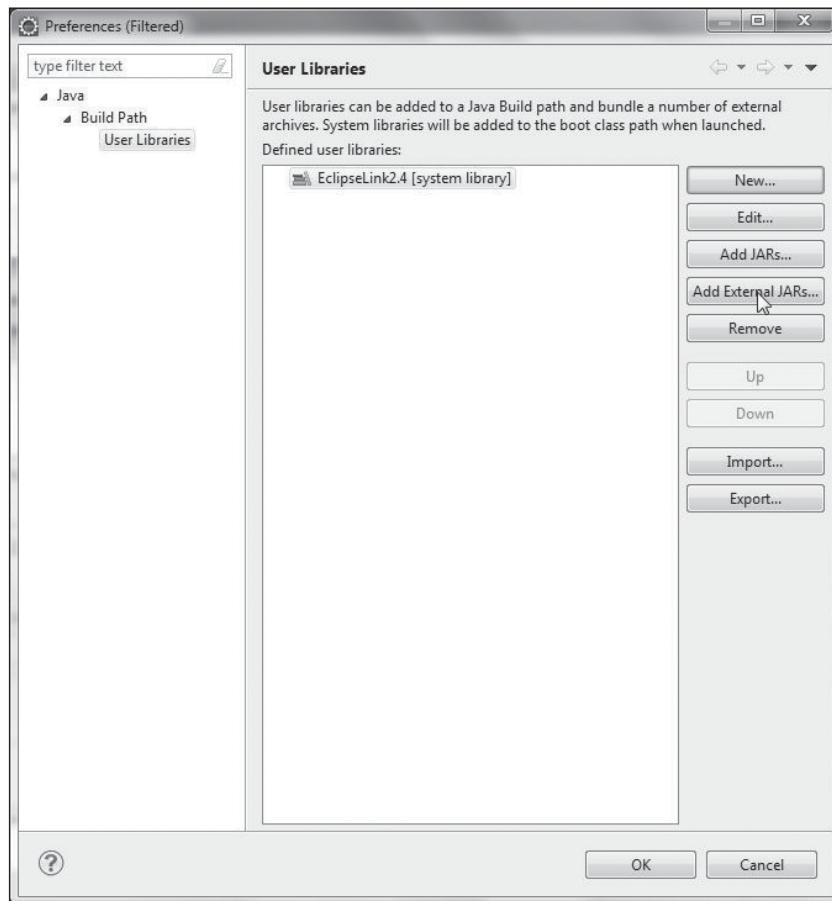
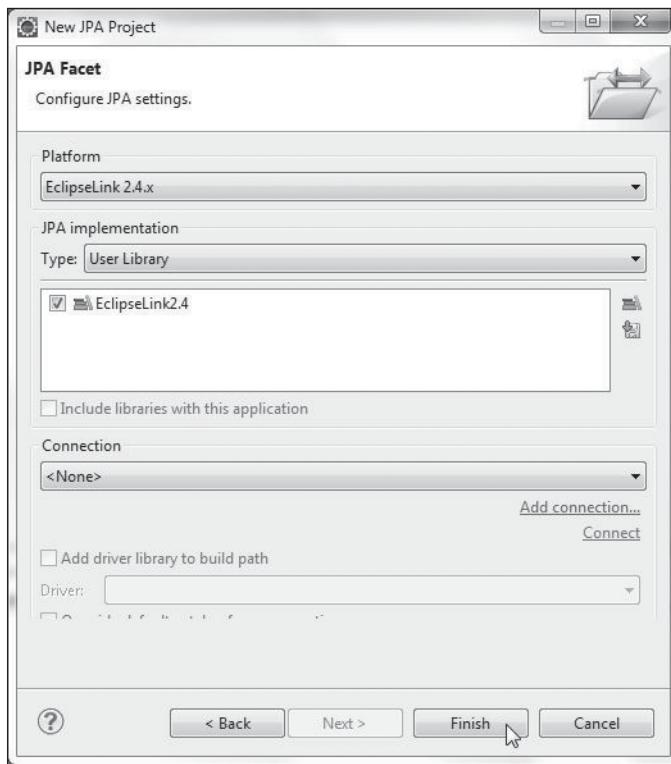


Figure 9.8

Adding external JARs to the user library.

Source: Eclipse Foundation.

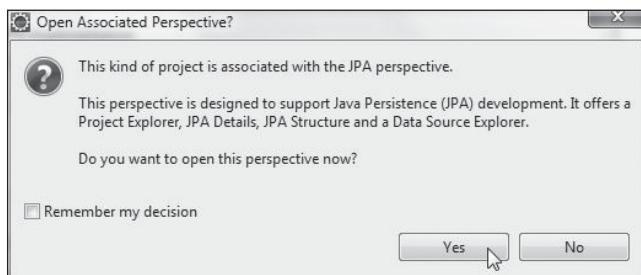
8. Add the eclipselink-2.4.2.jar file to the EclipseLink2.4 user library and click OK. The EclipseLink2.4 user library is added to new JPA project, as shown in Figure 9.9. Click Finish.

**Figure 9.9**

The new user library added to the JPA project.

Source: Eclipse Foundation.

9. An Open Associated Perspective dialog box prompts you to open the JPA perspective. Click Yes, as shown in Figure 9.10. The EclipseLink2.4 library is added to the Java build path of the Kundera JPA project, as shown in the Properties for Kundera dialog box.

**Figure 9.10**

Select Yes to open the JPA perspective.

Source: Eclipse Foundation.

10. Click the Add External JARs button to add the kundera-cassandra-2.2.1-jar-with-dependencies.jar file to the Java build path with the Add External JARs button. The JAR files listed in Table 9.1 are included in the Kundera project's Java build path. The libraries and JARs in the Java build path of the Kundera project are shown in Figure 9.11.

Table 9.1 Kundera Project JAR Files

JAR File	Description
kundera-cassandra-2.2.1-jar-with-dependencies.jar	Kundera library for Cassandra
javax.persistence_2.0.5.v201212031355.jar	JPA 2.0 API
eclipselink-2.4.2.jar	JPA 2.0 implementation

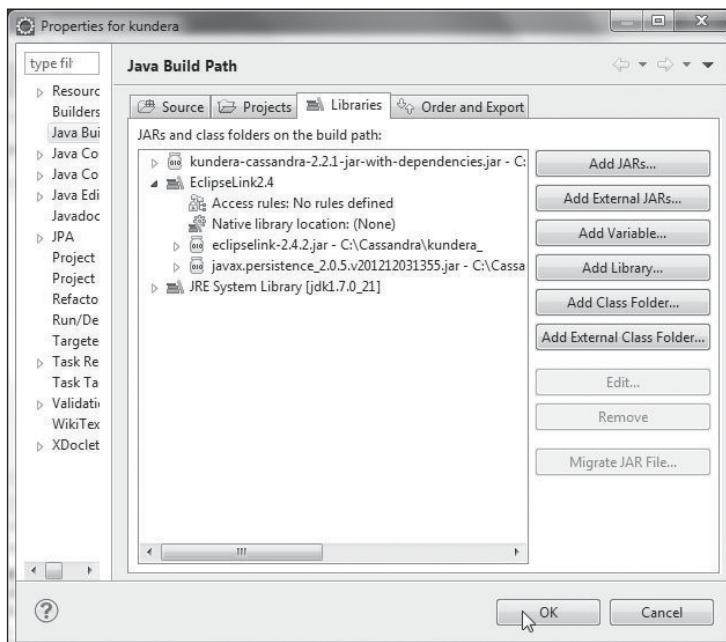


Figure 9.11

Libraries and JARs in the Kundera project.

Source: Eclipse Foundation.

The Kundera JPA project is created. The JPA project includes a META-INF/persistence.xml file for configuring properties for the object/relational mapping, as shown in Figure 9.12.

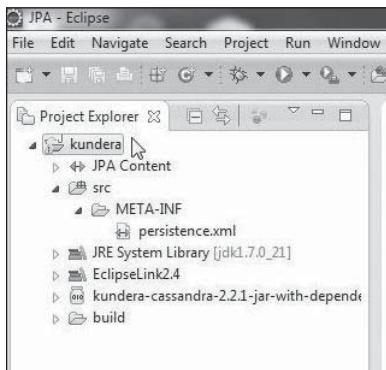


Figure 9.12
New JPA project.

Source: Eclipse Foundation.

CREATING A JPA ENTITY CLASS

The domain model for a JPA object/relational mapping application is defined in a JPA entity class. The domain model class is just a plain old Java object (POJO) that describes the Java object entity to be persisted, the object properties, and the Cassandra keyspace and column family to persist to.

In this section, you will create a JPA entity class for object/relational mapping using Kundera and the Cassandra database. Cassandra, though not a relational database, can be used with object/relational mapping using the Kundera library, which supports mainly NoSQL databases. Follow these steps:

1. Choose File > New > Other.
2. In the New dialog box, choose JPA > JPA Entity. Then click Next, as shown in Figure 9.13.

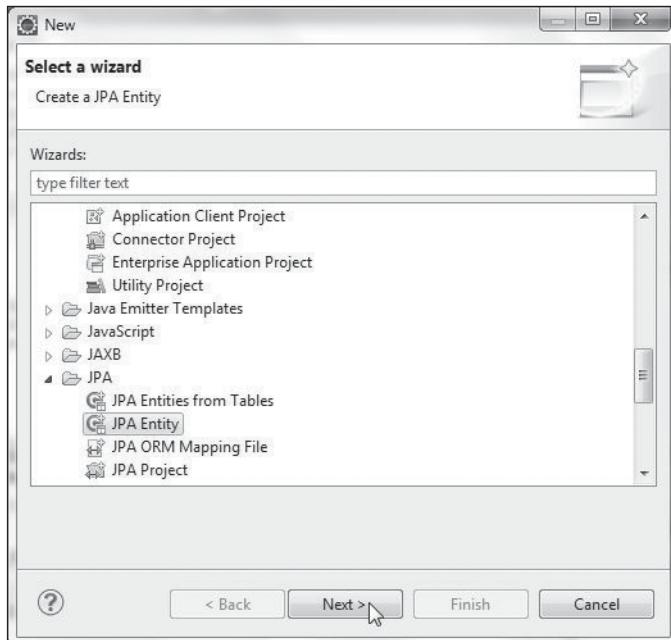


Figure 9.13

Select JPA > JPA Entity.

Source: Eclipse Foundation.

3. In the New JPA Entity wizard, select the Kundera project (a JPA project is required for a JPA entity), select a source folder (kundera/src), specify a Java package (kundera), and specify a class name (Catalog). In the Inheritance section, choose the Entity option button. Then click Next, as shown in Figure 9.14.

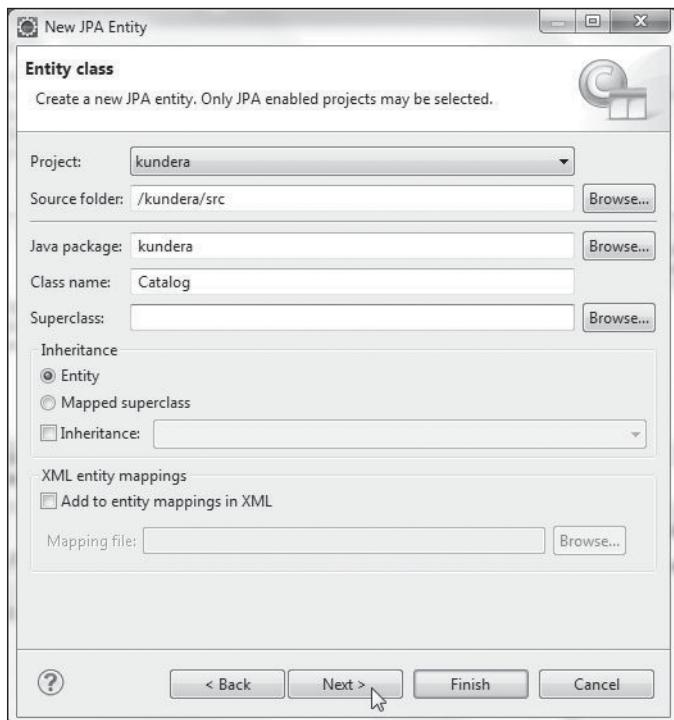
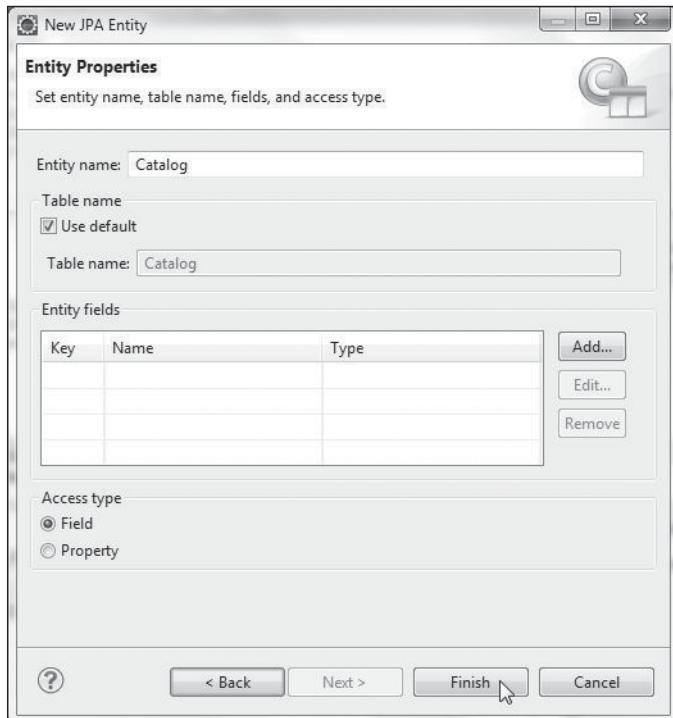


Figure 9.14
Configuring a JPA entity class.
Source: Eclipse Foundation.

4. In the Entity Properties dialog box, select the Use Default checkbox to select the default table name, Catalog. Then, under Access Type, leave the default setting, Field, selected. Finally, click Finish, as shown in Figure 9.15. The Catalog JPA entity is created.

**Figure 9.15**

Configuring JPA entity properties.

Source: Eclipse Foundation.

Annotate the `Catalog` class with an `@Entity` annotation to indicate that the class is a JPA entity class. By default, the entity name is the same as the entity class name. Annotate the class with `@Table` to indicate the Cassandra table name and schema. The table name is the column family name `catalog`. The schema is in `Keyspace@persistence-unit` format. For the Kundera keyspace and the kundera persistence unit name, which you will configure in the next section, the schema is `Kundera@kundera`.

```
@Entity(name = "catalog")
@Table(name = "catalog", schema = "Kundera@kundera")
```

The entity class implements the `Serializable` interface to serialize a cache-enabled entity bean to a cache when persisted to a database. To associate a version number with a serializable class by serialization runtime, specify a `serialVersionUID` variable.

```
private static final long serialVersionUID = 1L;
```

Annotate the `catalogId` field with the `@Id` annotation to indicate that the field is the primary key of the entity.

```
@Id  
private String catalogId;
```

The primary key column name in the Cassandra database is assumed to be the name of the primary key of the entity class. The field annotated with `@Id` must be one of the following types:

- Java primitive type, such as `int` or `double`
- Any primitive wrapper type, such as `Integer`, `Double`, `String`, `java.util.Date`, `java.sql.Date`, `java.math.BigDecimal`, or `java.math.BigInteger`

Add fields called `journal`, `publisher`, `edition`, `title`, and `author`, and annotate them with the `@Column` annotation to indicate that the fields are mapped to columns in the Cassandra table. (Recall that in Cassandra, a column family is also called a table.)

```
@Column(name = "journal")  
private String journal;  
  
@Column(name = "publisher")  
private String publisher;  
@Column(name = "edition")  
private String edition;  
  
@Column(name = "title")  
private String title;  
  
@Column(name = "author")  
private String author;
```

Add get/set methods for each of the fields. The JPA entity class `Catalog` appears in Listing 9.1.

Listing 9.1 The JPA Entity Class

```
package kundera;  
  
import java.io.Serializable;  
import javax.persistence.*;  
  
/**  
 * Entity implementation class for Entity: Catalog  
 *  
 */
```

```
@Entity(name = "catalog")
@Table(name = "catalog", schema = "Kundera@kundera")
public class Catalog implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    private String catalogId;
    public Catalog() {
        super();
    }
    @Column(name = "journal")
    private String journal;
    @Column(name = "publisher")
    private String publisher;
    @Column(name = "edition")
    private String edition;
    @Column(name = "title")
    private String title;
    @Column(name = "author")
    private String author;
    public String getCatalogId() {
        return catalogId;
    }
    public void setCatalogId(String catalogId) {
        this.catalogId = catalogId;
    }
    public String getJournal() {
        return journal;
    }
    public void setJournal(String journal) {
        this.journal = journal;
    }
    public String getPublisher() {
        return publisher;
    }
    public void setPublisher(String publisher) {
        this.publisher = publisher;
    }
    public String getEdition() {
        return edition;
    }
    public void setEdition(String edition) {
```

```

        this.edition = edition;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public String getAuthor() {
        return author;
    }
    public void setAuthor(String author) {
        this.author = author;
    }
}

```

CONFIGURING JPA IN PERSISTENCE.XML

A META-INF/persistence.xml configuration file was created when a JPA project was created in the Eclipse IDE. In this section, you will configure the object/relational mapping in the persistence.xml configuration file. Kundera supports some properties, specified in persistence.xml with the <property/> tag, common to all NoSQL data stores it supports. These common properties are discussed in Table 9.2.

Table 9.2 Kundera Properties for NoSQL Data Stores

Property	Description	Required/Optional
kundera.nodes	Node(s) on which NoSQL server is running.	Required
kundera.port	NoSQL database port.	Required
kundera.keyspace	NoSQL database keyspace.	Required
kundera.dialect	The NoSQL database dialect to determine the persistence provider. Valid values are cassandra, mongodb, and hbase.	Required
kundera.client.lookup.class	NoSQL database-specific client class for low-level data store operations.	Required

(Continued)

Table 9.2 Kundera Properties for NoSQL Data Stores (*Continued*)

Property	Description	Required/Optional
kundera.cache.provider.class	The L2 cache implementation class.	Required
kundera.cache.config.resource	File containing L2 cache implementation.	Required
kundera.ddl.auto.prepare	Specifies an option to automatically generate schema and tables for all entities. Valid options are as follows: <ul style="list-style-type: none"> ■ create: Drops schema if it already exists and creates schema/tables based on entity definitions. ■ create-drop: Same as create, but drops schema after operation ends. ■ update: Updates schema/tables based on entity definitions. ■ validate: Validates schema/table based on entity definitions and throws a SchemaGenerationException if validation fails. 	Optional
kundera.pool.size.max.active	Upper limit on the number of object instances managed by the pool per node.	Optional
kundera.pool.size.max.idle	Upper limit on the number of idle object instances in the pool.	Optional
kundera.pool.size.min.idle	Minimum number of idle object instances in the pool.	Optional
kundera.pool.size.max.total	Upper limit on the total number of object instances in the pool from all nodes combined.	Optional
index.home.dir	If Lucene indexes are chosen instead of the built-in secondary indexes, the directory path to store Lucene indexes.	Optional

kundera.client.property	Name of the NoSQL database-specific configuration file, which must be in the class path.	Optional
kundera.batch.size	Batch size in integer for bulk insert/update.	Optional
kundera.username	Username to authenticate Cassandra and MongoDB.	Optional
kundera.password	Password to authenticate Cassandra and MongoDB.	Optional

In the persistence.xml file for the Kundera project, specify the persistence-unit name as "kundera". Add a <provider/> element set to com.impetus.kundera.KunderaPersistence. Specify the JPA entity class as kundera.Catalog in the <class/> element. Add <property> tags grouped as sub-elements of the <properties> tag. Then add the properties discussed in Table 9.3.

Table 9.3 JPA Configuration Properties in Persistence.xml

Property	Value
kundera.nodes	localhost
kundera.port	9160
kundera.keyspace	Kundera
kundera.dialect	cassandra
kundera.client.lookup.class	com.impetus.client.cassandra.pelops.PelopsClientFactory
kundera.cache.provider.class	com.impetus.kundera.cache.ehcache.EhCacheProvider
kundera.cache.config.resource	/ehcache-test.xml

The persistence.xml configuration file appears in Listing 9.2.

Listing 9.2 The Persistence.xml Configuration File

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="kundera">
    <provider>com.impetus.kundera.KunderaPersistence</provider>
    <class>kundera.Catalog</class>
    <properties>
      <property name="kundera.nodes" value="localhost"/>
      <property name="kundera.port" value="9160"/>
      <property name="kundera.keyspace" value="Kundera"/>
      <property name="kundera.dialect" value="cassandra"/>
      <property name="kundera.client.lookup.class"
        value="com.impetus.client.cassandra.pelops.PelopsClientFactory" />
      <property name="kundera.cache.provider.class"
        value="com.impetus.kundera.cache.ehcache.EhCacheProvider"/>
      <property name="kundera.cache.config.resource"
        value="/ehcache-test.xml"/>
    </properties>
  </persistence-unit>
</persistence>
```

Some NoSQL database-specific properties may also be specified in persistence.xml file. For example, to configure Cassandra-specific properties, add the following property for the Cassandra-specific configuration file in persistence.xml:

```
<property name="kundera.client.property" value="kundera-cassandra.xml" />
```

The name of the Cassandra-specific configuration file, kundera-cassandra.xml, is arbitrary. Connection-, schema-, and table-specific properties may be specified. The connection-specific property that may be specified is cql.version. Some of the schema-specific properties supported are discussed in Table 9.4.

Table 9.4 Schema-Specific Properties in Persistence.xml

Property	Description
strategy.class	The replica placement strategy class. Valid values are SimpleStrategy and NetworkTopologyStrategy.
replication.factor	The replication factor for replica placement.
durable.writes	A Boolean to indicate whether writes are durable. The default value is true. All writes in Cassandra are written to memory and in commit logs. A write is considered a success only if it is written to both memory and the commit log. If the server crashes before a write to memory is flushed to the data store, the write to the commit log is applied when the server restarts.

The column family–specific properties supported by Cassandra are discussed in Table 9.5.

Table 9.5 Column Family–Specific Properties

Property	Description
default.validation.class	Default validation class for row key and columns.
key.validation.class	Row key validation class.
comment	Comment.
replicate.on.write	Replicates write operations to all affected replicas regardless of consistency. Applies only to counters.
comparator.type	Data type used to validate and sort column names.

A sample Cassandra-specific configuration file appears in Listing 9.3.

Listing 9.3 Sample Cassandra-Specific Configuration File

```
<?xml version="1.0" encoding="UTF-8"?>
<clientProperties>
  <datastores>
    <dataStore>
      <name>cassandra</name>
      <connection>
```

```
<properties>
    <property name="cql.version" value="3.0.0"></property>
</properties>
</connection>
<schemas>
    <schema>
        <name>KunderaCassandra</name>
        <properties>
            <property name="strategy.class" value="SimpleStrategy" />
            <property name="replication.factor" value="1" />
            <property name="durable.writes" value="true" />
        </properties>
        <tables>
            <table>
                <name>catalog</name>
                <properties>
                    <property name="default.validation.class"
value="UTF8Type"></property>
                    <property name="key.validation.class" value="UTF8Type">
                </property>
                <property name="replicate.on.write" value="true"></property>
                <property name="comparator.type" value="UTF8Type"></property>
            </properties>
        </table>
        </tables>
    </schema>
</schemas>
</dataStore>
</datastores>
</clientProperties>
```

The Cassandra-specific configuration file is not required if you are using the default values for the properties and you have not used any Cassandra-specific configuration files in this chapter. The Cassandra-specific configuration file listed is provided as a sample if non-default values are to be configured.

CREATING A JPA CLIENT CLASS

You have configured a JPA project for object/relational mapping to the Cassandra database. Next, you will run some CRUD operations using the JPA API. First, however, you

need to create a client class for the CRUD operations. You will use a Java class as a client class. Follow these steps:

1. Select File > New > Other.
2. In the New dialog box, select Java > Class. Then click Next, as shown in Figure 9.16.

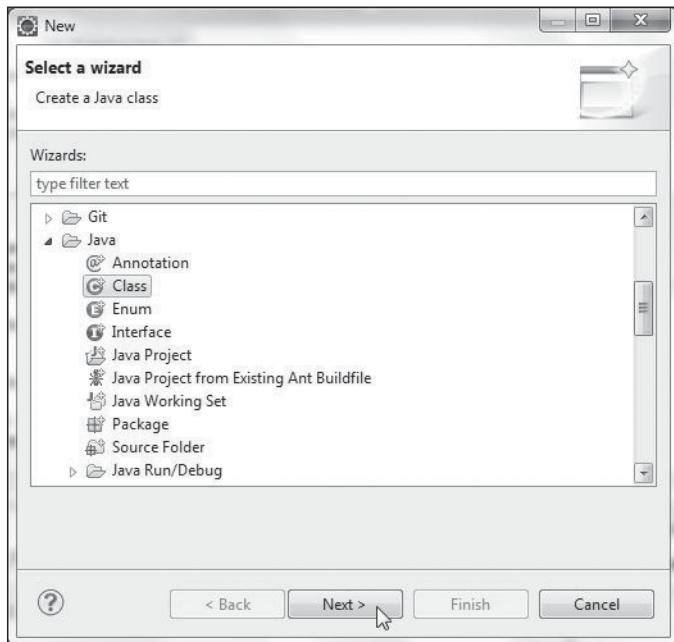


Figure 9.16
Selecting Java > Class.

Source: Eclipse Foundation.

3. In the New Java Class wizard, specify a package (`kundera`) and a class name (`KunderaClient`). Then select the method stub for the main method to add to the class. Finally, click Finish, as shown in Figure 9.17. The `kundera.KunderaClient` class is added to the Kundera project, as shown in Figure 9.18.

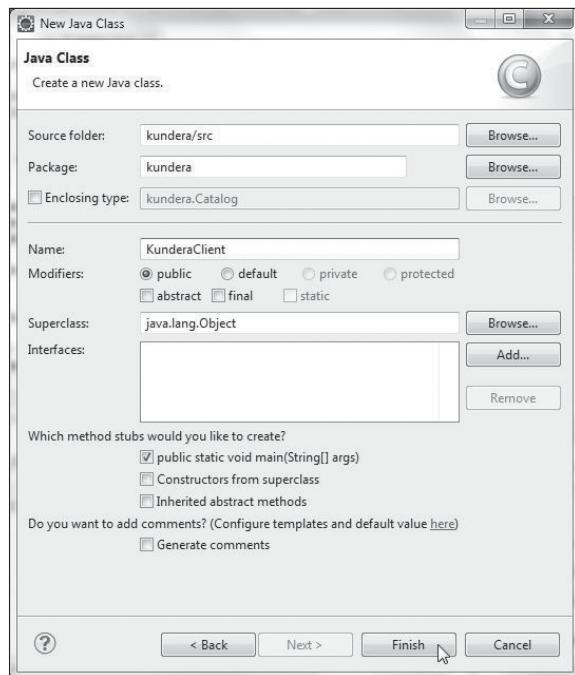


Figure 9.17
Creating a JPA client class.

Source: Eclipse Foundation.

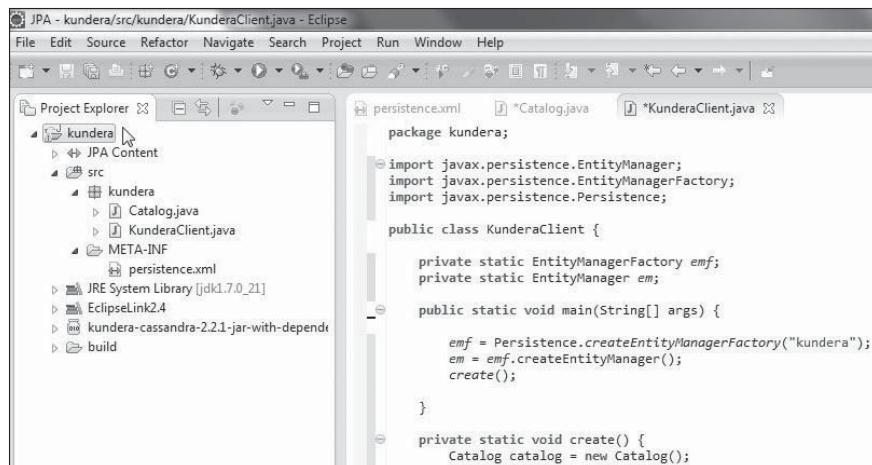


Figure 9.18
The JPA project with a JPA client class.

Source: Eclipse Foundation.

RUNNING JPA CRUD OPERATIONS

In the next few sections, you will create a catalog. You have already created a catalog table; next, you will add data to the catalog table, find a catalog entry, update a catalog entry, and delete a catalog entry.

Creating a Catalog

In this section, you will add some data to the catalog column family in Cassandra. Add a method called `create()` to the `KunderaClient` class and invoke the method from the `main` method so that the method is invoked when the application is run. The JPA API is defined in the `javax.persistence` package. The `EntityManager` interface is used to interact with the persistence context. The `EntityManagerFactory` interface is used to interact with the entity manager factory for the persistence unit. The `Persistence` class is used to obtain an `EntityManagerFactory` object in a Java SE environment. Create an `EntityManagerFactory` object using the `Persistence` class static method `createEntityManagerFactory(java.lang.String persistenceUnitName)`. Create an `EntityManager` instance from the `EntityManagerFactory` object using the `createEntityManager()` method.

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("kundera");
em = emf.createEntityManager();
```

In the `create()` method, create an instance of the JPA entity class `Catalog`. Using the set methods, set the `catalogId`, `journal`, `publisher`, `edition`, `title`, and `author` fields.

```
Catalog catalog = new Catalog();
catalog.setCatalogId("catalog1");
catalog.setJournal("Oracle Magazine");
catalog.setPublisher("Oracle Publishing");
catalog.setEdition("November-December 2013");
catalog.setTitle("Engineering as a Service");
catalog.setAuthor("David A. Kelly");
```

Use the `persist(java.lang.Object entity)` method in the `EntityManager` interface to make the domain model managed and persistent.

```
em.persist(catalog);
```

Similarly, other JPA instances may be persisted.

To run the KunderaClient application, right-click the KunderaClient.java file in the Package Explorer and select Run As > Java Application, as shown In Figure 9.19.

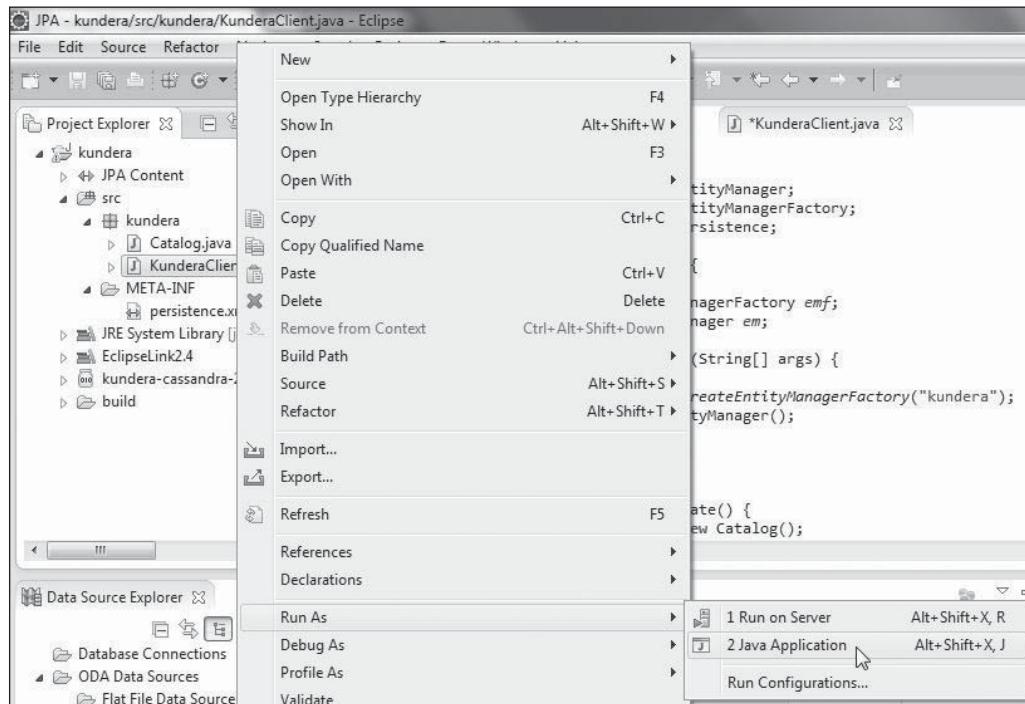


Figure 9.19

Running the KunderaClient application.

Source: Eclipse Foundation.

Three rows are added to the catalog column family. In Cassandra-Cli, run the following command to list the entity instances persisted using Kundera to the catalog column family:

```
list catalog;
```

The output lists the three rows added, as shown in Figure 9.20.

```
[default@Kundera] list catalog;
Using default limit of 100
Using default cell limit of 100

RowKey: catalog1
=> <name=author, value=David A. Kelly, timestamp=1390678269145>
=> <name=edition, value=November-December 2013, timestamp=1390678269145>
=> <name=journal, value=Oracle Magazine, timestamp=1390678269145>
=> <name=publisher, value=Oracle Publishing, timestamp=1390678269145>
=> <name=title, value=Engineering as a Service, timestamp=1390678269145>

RowKey: catalog2
=> <name=author, value=Tom Haunert, timestamp=1390678269173>
=> <name=edition, value=November-December 2013, timestamp=1390678269173>
=> <name=journal, value=Oracle Magazine, timestamp=1390678269173>
=> <name=publisher, value=Oracle Publishing, timestamp=1390678269173>
=> <name=title, value=Quintessential and Collaborative, timestamp=1390678269173>

RowKey: catalog3
=> <name=author, value=, timestamp=1390678269175>
=> <name=edition, value=November-December 2013, timestamp=1390678269175>
=> <name=journal, value=Oracle Magazine, timestamp=1390678269175>
=> <name=publisher, value=Oracle Publishing, timestamp=1390678269175>
=> <name=title, value=, timestamp=1390678269175>

3 Rows Returned.
Elapsed time: 152 msec(s.)
[default@Kundera]
```

Figure 9.20

Listing the three rows added to Cassandra.

Source: Microsoft Corporation.

Finding a Catalog Entry Using the Entity Class

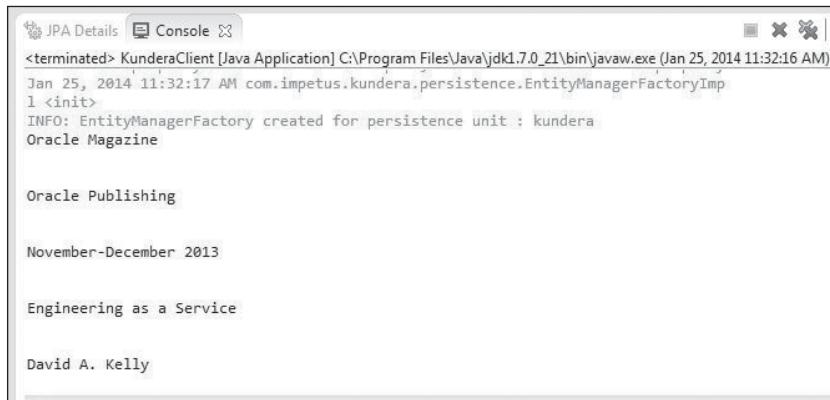
The EntityManager class provides several methods for finding an entity instance. In this section, you will find a Catalog entity instance using the `find(java.lang.Class<T> entityClass, java.lang.Object primaryKey)` method in which the first parameter is the entity class and the second parameter is the primary key for the row to find. Add a method called `findByClass()` to the KunderaClient class and invoke the method from the main method so that the method is invoked when the application is run. Invoke the `find(java.lang.Class<T> entityClass, java.lang.Object primaryKey)` method using `Catalog.class` as the first argument and "catalog1" as the second argument.

```
Catalog catalog = em.find(Catalog.class, "catalog1");
```

Invoke the get methods on the Catalog instance to output the entity fields.

```
System.out.println(catalog.getJournal());
System.out.println(catalog.getPublisher());
System.out.println(catalog.getEdition());
System.out.println(catalog.getTitle());
System.out.println(catalog.getAuthor());
```

Run the KunderaClient application in the Eclipse IDE. The column values for the row with the primary key "catalog1" are output, as shown in Figure 9.21.

**Figure 9.21**

Column values for row with primary key catalog1.

Source: Eclipse Foundation.

Finding a Catalog Entry Using a JPA Query

The `Query` interface is used to run a query in the Java Persistence query language and native SQL. The `EntityManager` interface provides several methods for creating a `Query` instance. In this section, you will run a Java Persistence query language statement by first creating an instance of `Query` with the `EntityManager` method `createQuery(java.lang.String qlString)` and then invoking the `getResultSet()` method on the `Query` instance. Add a method called `query()` to the `KunderaClient` class and invoke the method from the `main` method so that the method is invoked when the application is run. In the `query()` method, invoke the `createQuery(java.lang.String qlString)` method to create a `Query` instance. Supply the Java Persistence query language statement as `SELECT c FROM Catalog c`.

```
javax.persistence.Query query = em.createQuery("SELECT c FROM Catalog c");
```

Invoke the `getResultSet()` method on the `Query` instance to run the `SELECT` statement and return a `List<Catalog>` as the result.

```
List<Catalog> results = query.getResultList();
```

Iterate over the `List` object using an enhanced `for` statement to output the fields of the `Catalog` instance.

```
for (Catalog catalog : results) {
    System.out.println(catalog.getCatalogId());
    System.out.println(catalog.getJournal());
    System.out.println(catalog.getPublisher());
```

```
System.out.println(catalog.getEdition());
System.out.println(catalog.getTitle());
System.out.println(catalog.getAuthor());
}
```

Run the KunderaClient application to output the result of the Java Persistence query language query, as shown in Figure 9.22.



The screenshot shows the Eclipse IDE's Console tab with the title bar "JPA Details" and "Console". The content of the console window is as follows:

```
<terminated> KunderaClient [Java Application] C:\Program
catalog1

Oracle Magazine

Oracle Publishing

November-December 2013

Engineering as a Service

David A. Kelly
catalog2

Oracle Magazine

Oracle Publishing

November-December 2013

Quintessential and Collaborative

Tom Haunert
catalog3
```

Figure 9.22

Output from the Java Persistence query language query.

Source: Eclipse Foundation.

All three rows are output as follows:

```
catalog1
Oracle Magazine
Oracle Publishing
November-December 2013
Engineering as a Service
David A. Kelly
catalog2
Oracle Magazine
Oracle Publishing
```

November-December 2013
Quintessential and Collaborative
Tom Haunert
catalog3
Oracle Magazine
Oracle Publishing
November-December 2013

Updating a Catalog Entry

In this section, you will update a catalog entry using the Java Persistence API. The `persist()` method in `EntityManager` may be used to persist an updated entity instance. Add a method called `update()` to the `KunderaClient` class and invoke the method from the `main` method so that it is invoked when the application is run. For example, to update the `edition` column in the row with the primary key "catalog1", create an entity instance for the `catalog1` row using the `find(java.lang.Class<T> entityClass, java.lang.Object primaryKey)` method. Then set the `edition` field to the updated value using the `setEdition` method. Persist the updated `Catalog` instance using the `persist(java.lang.Object entity)` method.

```
Catalog catalog = em.find(Catalog.class, "catalog1");
catalog.setEdition("Nov-Dec 2013");
em.persist(catalog);
```

The Java Persistence query language provides the `UPDATE` clause to update a row. Create a `Query` instance using an `UPDATE` statement and the `createQuery(String)` method in `EntityManager`. Then invoke the `executeUpdate()` method to execute the `UPDATE` statement.

```
em.createQuery("UPDATE Catalog c SET c.journal =
'Oracle-Magazine'").executeUpdate();
```

The `journal` column in all the rows in the `catalog` column family is updated. Having applied updates, invoke the `query()` method to output the updated rows. The updated rows have the updated values, as shown in Figure 9.23.

```
<terminated> KunderaClient [Java Application] C:\catalog1

'Oracle-Magazine'

Oracle Publishing

Nov-Dec 2013

Engineering as a Service

David A. Kelly
catalog2

'Oracle-Magazine'

Oracle Publishing

November-December 2013

Quintessential and Collaborative

Tom Haunert
<
```

Figure 9.23
Updating Cassandra data.
Source: Eclipse Foundation.

The complete output for the updated rows is as follows:

```
catalog1
'Oracle-Magazine'
Oracle Publishing
Nov-Dec 2013
Engineering as a Service
David A. Kelly
catalog2
'Oracle-Magazine'
Oracle Publishing
November-December 2013
Quintessential and Collaborative
Tom Haunert
catalog3
'Oracle-Magazine'
Oracle Publishing
November-December 2013
```

Deleting a Catalog Entry

In this section, you will remove rows persisted in Cassandra using the Java Persistence API. The `remove(java.lang.Object entity)` method in `EntityManager` may be used to remove an entity instance. Add a method called `delete()` to the `KunderaClient` class and invoke the method from the `main` method so that it is invoked when the application is run. To remove the row with the primary key "catalog1", create an entity instance for the `catalog1` row using the `find(java.lang.Class<T> entityClass, java.lang.Object primaryKey)` method. Then invoke the `remove(java.lang.Object entity)` method to remove the `catalog1` row from Cassandra.

```
Catalog catalog = em.find(Catalog.class, "catalog1");
em.remove(catalog);
```

Similarly, rows `catalog2` and `catalog3` may removed.

```
catalog = em.find(Catalog.class, "catalog2");
em.remove(catalog);
catalog = em.find(Catalog.class, "catalog3");
em.remove(catalog);
```

The Java Persistence query language provides the `DELETE` clause to delete a row. Create a `Query` instance using a `DELETE` statement and the `createQuery(String)` method in `EntityManager`. Then invoke the `executeUpdate()` method to execute the `DELETE` statement.

```
em.createQuery("DELETE FROM Catalog c").executeUpdate();
```

All rows are deleted. The `DELETE` statement does not delete the row itself but deletes all the columns in the rows. Having performed the deletion, either using the `remove(java.lang.Object entity)` method or the `DELETE` Java Persistence query language statement, invoke the `query()` method to output any `Catalog` instances persisted to `catalog` table. Because the `catalog` table does not contain any persisted `Catalog` instances, the `NullPointerException` is generated as shown in Figure 9.24.

```

<terminated> KunderaClient [Java Application] C:\Program Files\Java\jdk1.7.0_21\bin\javaw.exe (Jan 25, 2014 12:10:39 PM)
INFO: On getResultList() executing query: DELETE FROM Catalog c

After removing all catalog entries
Jan 25, 2014 12:10:40 PM com.impetus.kundera.query.QueryImpl getResultList
INFO: On getResultList() executing query: SELECT c FROM Catalog c
Exception in thread "main" java.lang.NullPointerException
    at kundera.KunderaClient.query(KunderaClient.java:89)
    at kundera.KunderaClient.delete(KunderaClient.java:143)
    at kundera.KunderaClient.main(KunderaClient.java:34)

```

Figure 9.24

The NullPointerException after deleting Cassandra data.

Source: Eclipse Foundation.

The rows in the catalog column family may be listed in the Cassandra-Cli with the following command:

```
list catalog;
```

Empty rows are listed as the row columns are deleted, as shown in Figure 9.25.

```

C:\Windows\system32\cmd.exe - cassandra-cli
[default@Kundera] list catalog;
Using default limit of 100
Using default cell limit of 100
RowKey: catalog1
RowKey: catalog2
RowKey: catalog3
3 Rows Returned.
Elapsed time: 145 msec(s).
[default@Kundera] -

```

Figure 9.25

Listing empty rows after deleting Cassandra data.

Source: Microsoft Corporation.

The KunderaClient application appears in Listing 9.4.

Listing 9.4 The KunderaClient Application

```

package kundera;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import javax.persistence.EntityManager;

```

```
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.PersistenceContext;
import javax.persistence.PersistenceContextType;
import javax.persistence.PersistenceUnit;

public class KunderaClient {
    private static EntityManager em;
    private static EntityManagerFactory emf;
    public static void main(String[] args) {
        emf = Persistence.createEntityManagerFactory("kundera");
        em = emf.createEntityManager();
        create();
        // findByClass();
        // query();
        // update();
        // delete();
    }
    private static void create() {
        Catalog catalog = new Catalog();
        catalog.setCatalogId("catalog1");
        catalog.setJournal("Oracle Magazine");
        catalog.setPublisher("Oracle Publishing");
        catalog.setEdition("November-December 2013");
        catalog.setTitle("Engineering as a Service");
        catalog.setAuthor("David A. Kelly");
        em.persist(catalog);
        catalog = new Catalog();
        catalog.setCatalogId("catalog2");
        catalog.setJournal("Oracle Magazine");
        catalog.setPublisher("Oracle Publishing");
        catalog.setEdition("November-December 2013");
        catalog.setTitle("Quintessential and Collaborative");
        catalog.setAuthor("Tom Haunert");
        em.persist(catalog);
        catalog = new Catalog();
        catalog.setCatalogId("catalog3");
        catalog.setJournal("Oracle Magazine");
        catalog.setPublisher("Oracle Publishing");
        catalog.setEdition("November-December 2013");
        catalog.setTitle("");
        catalog.setAuthor("");
        em.persist(catalog);
    }
}
```

```
private static void findByClass() {
    Catalog catalog = em.find(Catalog.class, "catalog1");
    System.out.println(catalog.getJournal());
    System.out.println("\n");
    System.out.println(catalog.getPublisher());
    System.out.println("\n");
    System.out.println(catalog.getEdition());
    System.out.println("\n");
    System.out.println(catalog.getTitle());
    System.out.println("\n");
    System.out.println(catalog.getAuthor());
}
private static void query() {
    javax.persistence.Query query = em
        .createQuery("SELECT c FROM Catalog c");
    List<Catalog> results = query.getResultList();
    if(results != null) {
        for (Catalog catalog : results) {
            System.out.println(catalog.getCatalogId());
            System.out.println("\n");
            System.out.println(catalog.getJournal());
            System.out.println("\n");
            System.out.println(catalog.getPublisher());
            System.out.println("\n");
            System.out.println(catalog.getEdition());
            System.out.println("\n");
            System.out.println(catalog.getTitle());
            System.out.println("\n");
            System.out.println(catalog.getAuthor());
        }
    }
}
private static void update() {
    Catalog catalog = em.find(Catalog.class, "catalog1");
    catalog.setEdition("Nov-Dec 2013");
    em.persist(catalog);
    em.createQuery("UPDATE Catalog c SET c.journal = 'Oracle-
Magazine'")
        .executeUpdate();
    /*
     * em.createQuery(
     * "UPDATE Catalog c  SET c.author = 'Kelly, David A.' WHERE
     * c.catalogId='catalog1'"
    
```

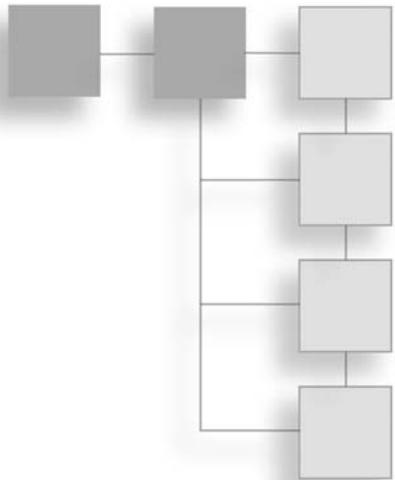
```
        * ) .executeUpdate(); update with WHERE does not get applied.  
    */  
    System.out.println("After updating");  
    System.out.println("\n");  
    query();  
}  
private static void delete() {  
    Catalog catalog = em.find(Catalog.class, "catalog1");  
    em.remove(catalog);  
    catalog = em.find(Catalog.class, "catalog2");  
    em.remove(catalog);  
    catalog = em.find(Catalog.class, "catalog3");  
    em.remove(catalog);  
    System.out.println("After removing catalog3");  
    query();  
    /*  
     * em.createQuery(  
     * "DELETE FROM Catalog c WHERE c.title='Engineering As a Service'"')  
     * .executeUpdate(); System.out.println("\n"); //  
     * System.out.println("After removing catalog1"); query();  
     */  
    // DELETE with WHERE does not get applied.  
    em.createQuery("DELETE FROM Catalog c").executeUpdate();  
    System.out.println("\n");  
    System.out.println("After removing all catalog entries");  
    query();  
}  
private static void close() {  
    em.close();  
    // emf.close();  
}  
}
```

SUMMARY

The JPA is designed for relational databases, but the Kundera library provides object/relational mapping using the JPA for NoSQL data stores Cassandra, MongoDB, and HBase. In this chapter, you used the Java Persistence API with Kundera to run CRUD operations on Cassandra. In the next chapter, you will use the Spring Data project with Apache Cassandra.

CHAPTER 10

USING SPRING DATA WITH CASSANDRA



Spring Data is designed for new data access technologies such as non-relational databases. The Spring Data Cassandra project adds Spring Data functionality to the Cassandra server. This chapter discusses how to use the Spring Data Cassandra project in Eclipse.

OVERVIEW OF THE SPRING DATA CASSANDRA PROJECT

The package for the conversion from Cassandra to Spring Data is `org.springframework.cassandra.convert`. The main interfaces and classes in the package are illustrated in Figure 10.1.

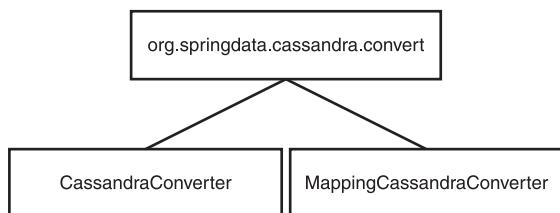


Figure 10.1

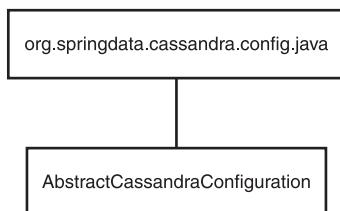
Main classes and interfaces in the `org.springframework.cassandra.convert` package.

The main interfaces and classes in the `org.springframework.cassandra.convert` package are discussed in Table 10.1.

Table 10.1 Main Classes and Interfaces in the org.springframework.data.cassandra.convert Package

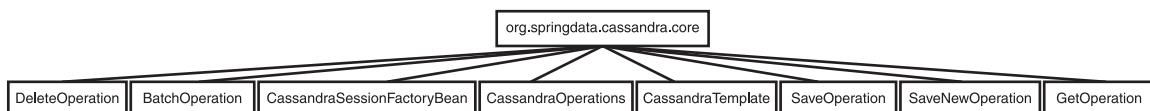
Class/Interface	Description
CassandraConverter	Central Cassandra-specific converter interface from object to row
MappingCassandraConverter	CassandraConverter for sophisticated mapping of domain objects to row

The package for the Spring Data Cassandra configuration is `org.springframework.data.cassandra.config.java`. This package has just one class, `AbstractCassandraConfiguration`, which is the base class for Spring Data Cassandra configuration using `JavaConfig`, as illustrated in Figure 10.2.

**Figure 10.2**

The `org.springframework.data.cassandra.config.java` package.

The package for the core classes in the Spring Data Cassandra project is `org.springframework.data.cassandra.core`. The package's main classes and interfaces are illustrated in Figure 10.3.

**Figure 10.3**

Main classes and interfaces in the `org.springframework.data.cassandra.core` package.

The main classes and interfaces in the `org.springframework.data.cassandra.core` package are discussed in Table 10.2.

Table 10.2 Main Classes and Interfaces in the org.springframework.cassandra.core Package

Class/Interface	Description
DeleteOperation	Base interface for delete operations.
BatchOperation	Base interface for batch operations.
CassandraSessionFactoryBean	Factory class for configuring a Cassandra session.
CassandraOperations	Operations for interacting with Cassandra. Also used by the SimpleCassandraRepository interface.
CassandraTemplate	Convenience API for all Cassandra operations using POJOs.
SaveOperation	Base interface for save (update) operations.
SaveNewOperation	Base interface for save (insert) operations.
GetOperation	Base interface for get (select) operations.

The core package for running CQL queries is `org.springframework.cassandra.cql.core`. The package has the classes and interfaces shown in Figure 10.4.

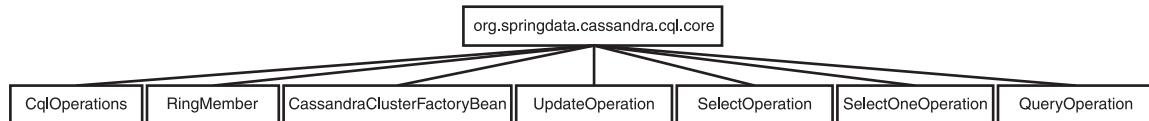


Figure 10.4
Classes and interfaces in the `org.springframework.cassandra.cql.core` package.

The main classes and interfaces in the `org.springframework.cassandra.core` package are discussed in Table 10.3.

Table 10.3 Main Classes and Interfaces in the org.springframework.cassandra.core Package

Class/Interface	Description
CqlOperations	Operations for interacting with Cassandra at the lowest level using CQL
RingMember	Represents a cluster node

(Continued)

**Table 10.3 Main Classes and Interfaces in the `org.springframework.cassandra.core` Package
(Continued)**

Class/Interface	Description
<code>CassandraClusterFactoryBean</code>	Factory class for configuring a Cassandra cluster
<code>UpdateOperation</code>	Base interface for update operations
<code>SelectOperation</code>	Base interface for select operations
<code>SelectOneOperation</code>	Base interface for the select operation to get a single result
<code>QueryOperation</code>	Base interface for query operations

The `org.springframework.cassandra.mapping` package defines the classes, interfaces, and annotation types for mapping a Spring Data domain object to Cassandra. Some of the annotation types in the package are illustrated in Figure 10.5.

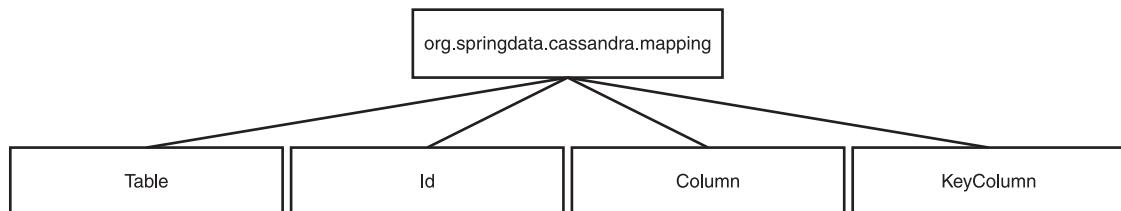


Figure 10.5
Main annotation types in the `org.springframework.cassandra.mapping` package.

The annotation types are discussed in Table 10.4.

Table 10.4 Main Annotation Types in the `org.springframework.cassandra.mapping` Package

Annotation Type	Description
<code>Table</code>	Domain object to be persisted to a Cassandra table
<code>Id</code>	Identifies a primary key ID in a Cassandra table
<code>Column</code>	Identifies a column in a Cassandra table
<code>KeyColumn</code>	Identifies a primary key column in a Cassandra table

The `org.springframework.cassandra.cql.config` package defines classes and enums for CQL configuration. Some of the classes are illustrated in Figure 10.6.

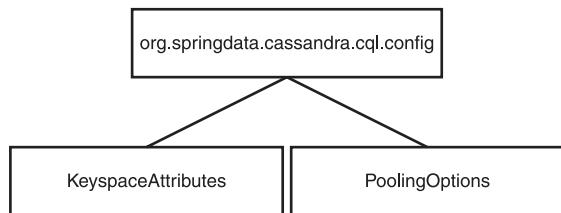


Figure 10.6
Main classes in the `org.springframework.cassandra.cql.config` package.

The classes are discussed in Table 10.5.

Table 10.5 Main Classes in the `org.springframework.cassandra.cql.config` Package

Class	Description
<code>KeyspaceAttributes</code>	Keyspace attributes used to create/validate or drop the keyspace on startup
<code>PoolingOptions</code>	Pooling options such as maximum connections and minimum/maximum simultaneous requests

SETTING THE ENVIRONMENT

To set the environment, you must install the following software:

- Eclipse IDE for Java EE developers from <http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/keplersr2>
- Apache Cassandra 2.04 or a later version from <http://cassandra.apache.org/download/>

Start Apache Cassandra with the following command:

`cassandra -f`

Create a Cassandra Keyspace called `springdata` with Cassandra-Cli. The `placement_strategy` option specifies the strategy used for replica placement and the `strategy_options` option specifies the replication factor as 1 via the `replication_factor` property.

```
CREATE KEYSPACE springdata
with placement_strategy = 'org.apache.cassandra.locator.SimpleStrategy'
and strategy_options = {replication_factor:1};
```

Set the `springdata` keyspace as the working keyspace using the following command:

```
use springdata;
```

The output from creating and setting a keyspace is shown in Figure 10.7.

```
C:\Windows\system32\cmd.exe - cassandra-cli
C:\>cassandra-cli
Starting Cassandra Client
Connected to: "Test Cluster" on 127.0.0.1/9160
Welcome to Cassandra CLI version 2.0.4

The CLI is deprecated and will be removed in Cassandra 3.0. Consider migrating
to cqlsh.
CQL is fully backwards compatible with Thrift data; see http://www.datastax.com/
dev/blog/thrift-to-cql3

Type 'help;' or '??' for help.
Type 'quit;' or 'exit;' to quit.

[default@unknown] CREATE KEYSPACE springdata
...     with placement_strategy = 'org.apache.cassandra.locator.SimpleStrategy'
...         and strategy_options = {replication_factor:1};
903559d5-3c47-3601-b2ef-312c5bc0f998
[default@unknown]
[default@unknown] use springdata;
Authenticated to keyspace: springdata
[default@springdata] -
```

Figure 10.7

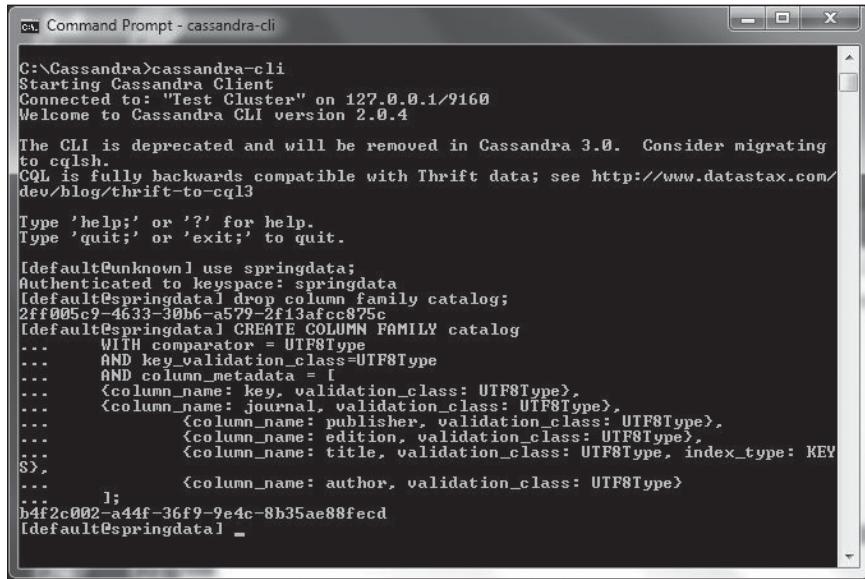
Creating a keyspace.

Source: Microsoft Corporation.

Next, create a column family called `catalog` in Cassandra-Cli. One of the columns must be named `key`. The comparator, used for column names, and the key validation class, used for the primary key value, are set to `UTF8Type`. The column metadata specifies columns `journal`, `publisher`, `edition`, `title`, and `author`. The validation class for columns, which is used to validate column values, is set to `UTF8Type`.

```
CREATE COLUMN FAMILY catalog
WITH comparator = UTF8Type
AND key_validation_class=UTF8Type
AND column_metadata =
[ {column_name: key, validation_class: UTF8Type},
{column_name: journal, validation_class: UTF8Type},
    {column_name: publisher, validation_class: UTF8Type},
    {column_name: edition, validation_class: UTF8Type},
    {column_name: title, validation_class: UTF8Type, index_type: KEYS},
    {column_name: author, validation_class: UTF8Type}
];
```

The output from the command is shown in Figure 10.8.



```
C:\>cassandra-cli
Starting Cassandra Client
Connected to: "Test Cluster" on 127.0.0.1/9160
Welcome to Cassandra CLI version 2.0.4

The CLI is deprecated and will be removed in Cassandra 3.0. Consider migrating
to cqlsh.
CQL is fully backwards compatible with Thrift data; see http://www.datastax.com/
dev/blog/thrift-to-cql3

Type 'help;' or '?' for help.
Type 'quit;' or 'exit;' to quit.

[default@unkn0wn1 use springdata;
Authenticated to keyspace: springdata
[default@springdata1 drop column family catalog;
2ff005c9-4633-30b6-a579-2f13afcc875c
[default@springdata1 CREATE COLUMN FAMILY catalog
...    WITH comparator = UTF8Type
...    AND key_validation_class=UTF8Type
...    AND column_metadata = [
...        <column_name: key, validation_class: UTF8Type>,
...        <column_name: journal, validation_class: UTF8Type>,
...        <column_name: publisher, validation_class: UTF8Type>,
...        <column_name: edition, validation_class: UTF8Type>,
...        <column_name: title, validation_class: UTF8Type, index_type: KEY>
...    ];
...    <column_name: author, validation_class: UTF8Type>
...];
b4f2c002-a44f-36f9-9e4c-8b35ae88feed
[default@springdata1 ]
```

Figure 10.8

Creating a column family.

Source: Microsoft Corporation.

CREATING A MAVEN PROJECT

Next, you will create a Maven project for Spring Data. Maven is a project management tool.

First, you need to create a Maven project in the Eclipse IDE. Follow these steps:

1. Select File > New > Other.
2. In the New dialog box, select Maven > Maven Project. Then click Next, as shown in Figure 10.9.

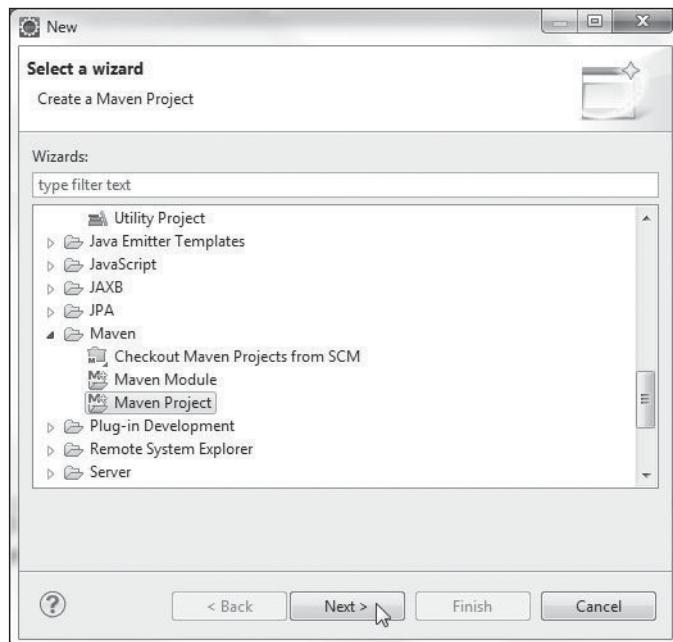


Figure 10.9
Selecting Maven > Maven Project.
Source: Eclipse Foundation.

3. The New Maven Project wizard starts. Select the Create a Simple Project checkbox and the Use Default Workspace Location checkbox. Then click Next, as shown in Figure 10.10.

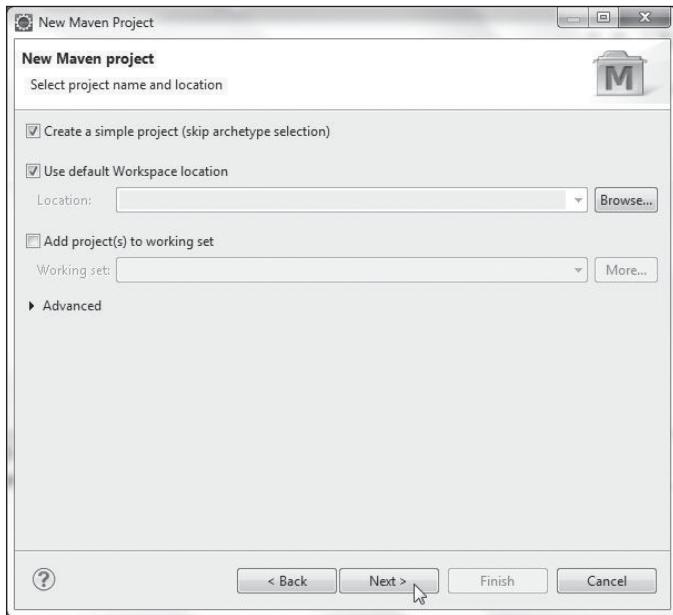


Figure 10.10

Selecting to create a simple project at the default location.

Source: Eclipse Foundation.

4. In the Configure Project screen, specify a group ID (com.cassandra.core), an artifact ID (SpringCassandra or spring-cassandra), a version number (1.0), the packaging used (jar), and a name (SpringCassandra). Then click Finish, as shown in Figure 10.11. A Maven project (SpringCassandra or spring-cassandra) is created, as shown in Figure 10.12. (Note that the downloadable project for this chapter is spring-cassandra rather than SpringCassandra, which is used in the chapter.)

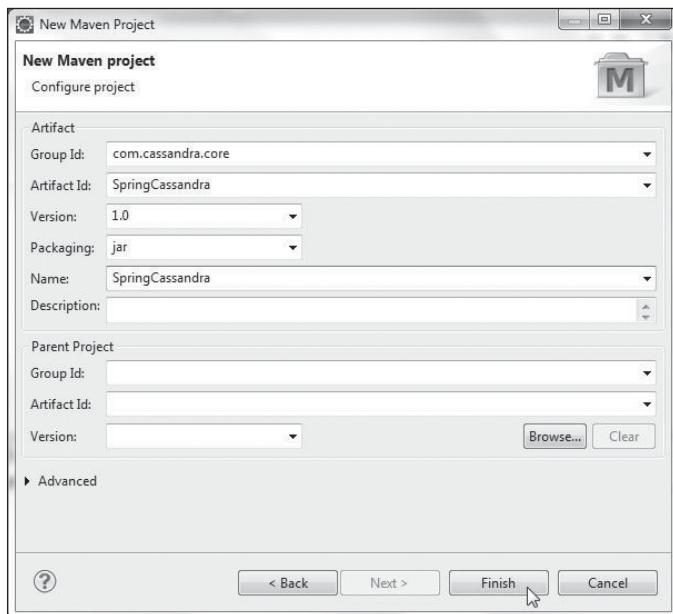


Figure 10.11
Configuring a new Maven project.

Source: Eclipse Foundation.

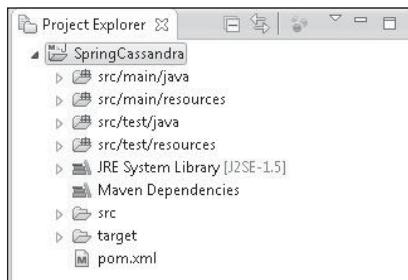


Figure 10.12
The new Maven project.

Source: Eclipse Foundation.

CONFIGURING THE MAVEN PROJECT

The Maven project includes a pom.xml file to specify the dependencies and build configuration for the project. In the pom.xml file, specify the dependencies listed in Table 10.6.

Table 10.6 Maven Project Dependencies

Dependency Group ID	Artifact ID	Version	Description
org.springframework	spring-core	3.2.5.RELEASE	Spring core
org.springframework	spring-context	3.2.5.RELEASE	Spring context
org.springdata	spring-data-cassandra	1.2.0.BUILD-SNAPSHOT	Spring Data Cassandra project core API

Specify the `maven-compiler-plugin` and `maven-eclipse-plugin` plug-ins in the build configuration. The pom.xml file to use the Spring Data Cassandra project appears in Listing 10.1.

Listing 10.1 The pom.xml File

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.cassandra.core</groupId>
  <artifactId>SpringCassandra</artifactId>
  <version>1.0</version>
  <name>SpringCassandra</name>
  <repositories>
    <repository>
      <id>sonatype-nexus-snapshots</id>
      <url>https://oss.sonatype.org/content/repositories/snapshots/</url>
    </repository>
  </repositories>
  <dependencies>
    <dependency>
      <groupId>org.springdata</groupId>
      <artifactId>spring-data-cassandra</artifactId>
      <version>1.2.0.BUILD-SNAPSHOT</version>
    </dependency>
    <!-- Spring framework -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
```

```
<version>3.2.5.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>3.2.5.RELEASE</version>
</dependency>
</dependencies>
<build>
    <plugins>
        <plugin>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.0</version>
            <configuration>
                <source>1.6</source>
                <target>1.6</target>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-eclipse-plugin</artifactId>
            <version>2.9</version>
            <configuration>
                <downloadSources>true</downloadSources>
                <downloadJavadocs>true</downloadJavadocs>
            </configuration>
        </plugin>
    </plugins>
</build>
</project>
```

CONFIGURING JavaConfig

Configure the Spring Data environment with POJOs using JavaConfig. The base class for Spring Data Cassandra configuration with JavaConfig is `org.springframework.cassandra.config.java.AbstractCassandraConfiguration`. In New Java Class wizard, create a Java class, `SpringCassandraApplicationConfig`, that extends the `org.springframework.cassandra.config.java.AbstractCassandraConfiguration` class as shown in Figure 10.13.

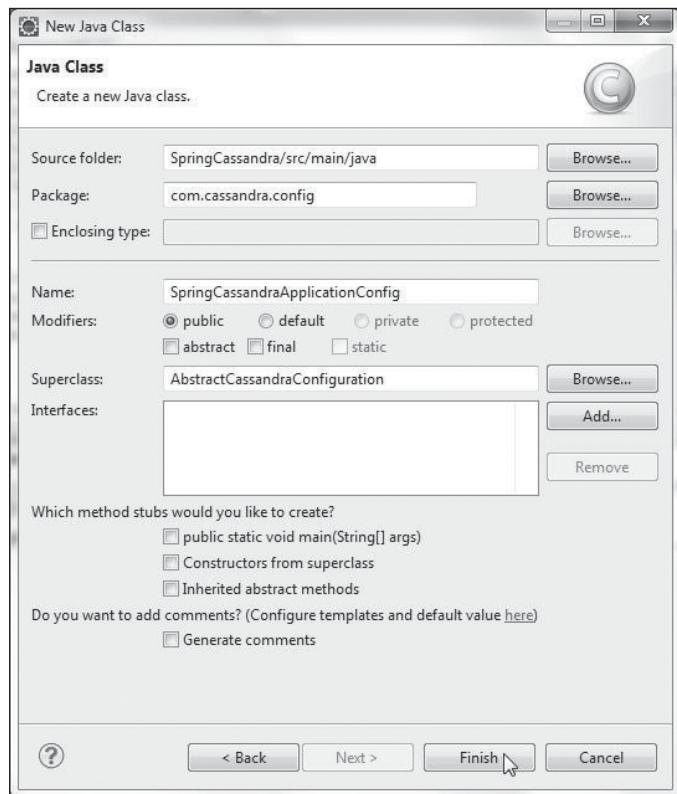


Figure 10.13
Creating a JavaConfig class.

Source: Eclipse Foundation.

Annotate the class with `@Configuration`, which indicates that the class is processed by the Spring container to generate bean definitions and service requests for the beans at runtime. The `SpringCassandraApplicationConfig` class must implement the inherited abstract method `keyspace()`. Return the keyspace name `springdata` as a `String`. The Spring Cassandra configuration class `SpringCassandraApplicationConfig` appears in Listing 10.2.

Listing 10.2 The Spring Configuration Class `SpringCassandraApplicationConfig`

```
package com.cassandra.config;

import java.beans.ConstructorProperties;
import org.springframework.cassandra.config.java.AbstractCassandraConfiguration;
import org.springframework.cassandra.convert.CassandraConverter;
import org.springframework.cassandra.convert.MappingCassandraConverter;
import org.springframework.cassandra.core.CassandraOperations;
import org.springframework.cassandra.core.CassandraSessionFactoryBean;
```

```
import org.springframework.cassandra.core.CassandraTemplate;
import org.springframework.cassandra.cql.config.KeyspaceAttributes;
import org.springframework.cassandra.cql.core.CassandraClusterFactoryBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import com.datastax.driver.core.Cluster;

@Configuration
public class SpringCassandraApplicationConfig extends
    AbstractCassandraConfiguration {
    public static final String KEYSPACE = "springdata";
    @Override
    protected String keyspace() {
        return KEYSPACE;
    }
}
```

CREATING A MODEL

Next, create the model class to use with the Spring Data Cassandra project. A domain object to be persisted to Cassandra server must be annotated with `org.springframework.cassandra.mapping.Table`. In the New Java Class wizard, create a POJO class named Catalog, as shown in Figure 10.14.

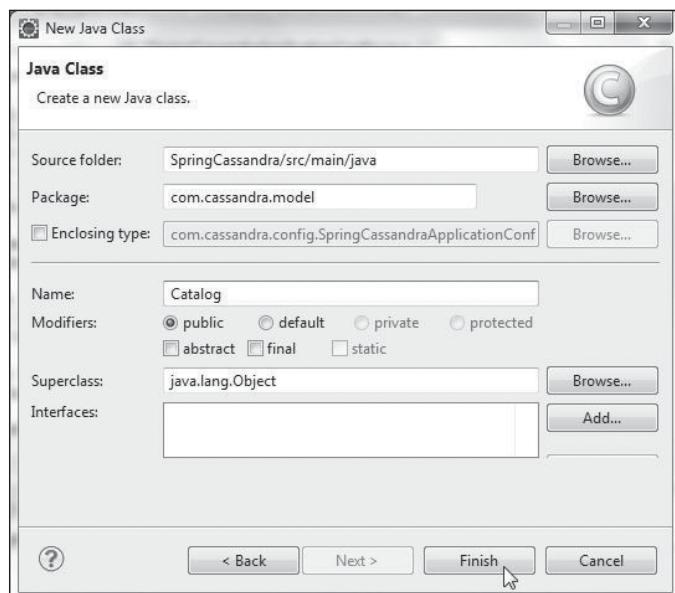


Figure 10.14
Creating a model POJO class named Catalog.

Source: Eclipse Foundation.

Add fields for key, journal, edition, publisher, title, and author and the corresponding get/set methods. Annotate the id field with @Id. Add constructors that may be used to construct a Catalog instance. The Catalog entity appears in Listing 10.3.

Listing 10.3 The Catalog Entity

```
package com.cassandra.model;
import org.springframework.data.annotation.Id;
import org.springframework.cassandra.mapping.Table;
@Table(name = "catalog")
public class Catalog {
    @Id
    private String key;
    private String journal;
    private String publisher;
    private String edition;
    private String title;
    private String author;
    public Catalog() {}
    public Catalog(String key, String journal, String publisher,
                  String edition, String title, String author) {
        this.key = key;
        this.journal = journal;
        this.publisher = publisher;
        this.edition = edition;
        this.title = title;
        this.author = author;
    }
    public String getKey() {
        return key;
    }
    public void setKey(String key) {
        this.key = key;
    }
    public String getJournal() {
        return journal;
    }
    public void setJournal(String journal) {
        this.journal = journal;
    }
    public String getPublisher() {
```

```
        return publisher;
    }
    public void setPublisher(String publisher) {
        this.publisher = publisher;
    }
    public String getEdition() {
        return edition;
    }
    public void setEdition(String edition) {
        this.edition = edition;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public String getAuthor() {
        return author;
    }
    public void setAuthor(String author) {
        this.author = author;
    }
}
```

The directory structure of the SpringCassandra project is shown in Figure 10.15.

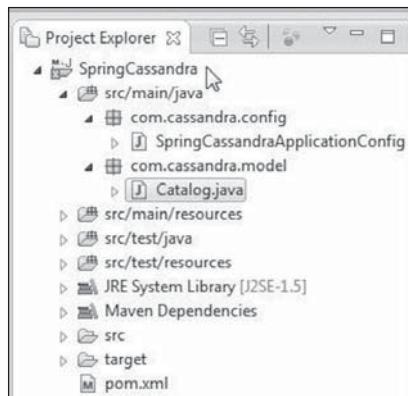


Figure 10.15

The directory structure of the SpringCassandra project.

Source: Eclipse Foundation.

USING SPRING DATA WITH CASSANDRA WITH TEMPLATE

The common CRUD operations on a Cassandra data source may be performed using the `org.springframework.cassandra.core.CassandraOperations` interface. The `org.springframework.cassandra.core.CassandraTemplate` class implements the `CassandraOperations` interface. In this section, you will run CRUD operations on Cassandra using the `CassandraTemplate` class. Create a Java client class (`CassandraClient`) for the Cassandra CRUD operations in New Java Class wizard, as shown in Figure 10.16.

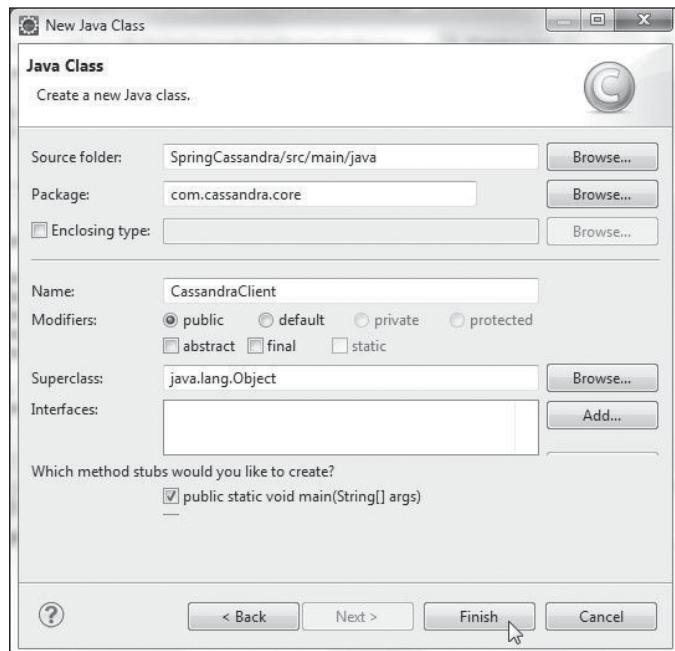
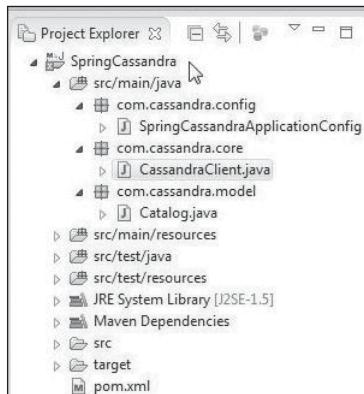


Figure 10.16
Creating a Java client class.
Source: Eclipse Foundation.

The directory structure of the `SpringCassandra` project is shown in Figure 10.17.

**Figure 10.17**

The directory structure of SpringCassandra project.

Source: Eclipse Foundation.

You can obtain a `CassandraTemplate` instance obtained using `ApplicationContext`. Create an `ApplicationContext` as follows:

```
ApplicationContext context = new
AnnotationConfigApplicationContext(SpringCassandraApplicationConfig.class);
```

The `getBean(Class requiredType)` method returns a named bean of the specified type. The class type is `CassandraOperations.class`.

```
CassandraOperations ops = context.getBean(CassandraOperations.class);
```

FINDING OUT ABOUT THE CASSANDRA CLUSTER

The `org.springframework.data.cassandra.core.CqlOperations` interface provides the overloaded `describeRing()` method to find the Cassandra cluster topology. Obtain a `CqlOperations` instance from the `CassandraOperations` instance using the `getCqlOperations()` method and invoke the `describeRing()` method to obtain a `List<RingMember>` instance. Iterate over the `List` to output the individual Cassandra node description.

```
for (RingMember member : ops.getQueryOperations().describeRing()) {
    System.out.println(member.toString());
}
```

Output the table name used for the specified entity class by the template using the `getTableName(Class<?> entityClass)` method in `CassandraOperations`.

```
System.out.println("Table name: " + ops.getTableName(Catalog.class));
```

To run the CassandraClient application, right-click the `CassandraClient.java` class in Package Explorer and select Run As > Java Application, as shown in Figure 10.18.

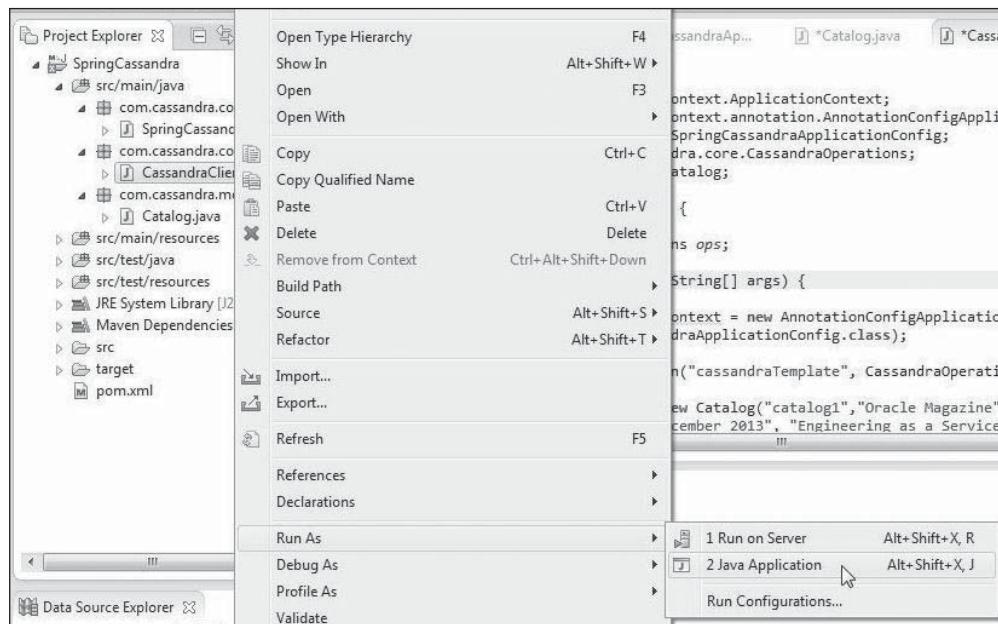


Figure 10.18
Running the CassandraClient application.

Source: Eclipse Foundation.

A description of the Cassandra node connected in the cluster is output, including the host name, address, data center, and rack. The Cassandra table name used for the Catalog entity class is also output, as shown in Figure 10.19.

```

Console

CassandraClient [Java Application] C:\Program Files\Java\jdk1.7.0_21\bin\javaw.exe (Jan 27, 2014 10:43:17 AM)
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
RingMember [hostName=localhost, address=127.0.0.1, DC=datacenter1, rack=rack1]
Table name: catalog

```

Figure 10.19
Cassandra node description.

Source: Eclipse Foundation.

RUNNING CASSANDRA CRUD OPERATIONS

You can use the `CassandraOperations` instance to perform various create, read, update, delete (CRUD) operations on a domain object stored in the Cassandra server. Add the methods discussed in Table 10.7 to the `CassandraClient` class and invoke the methods from the `main` method.

Table 10.7 CassandraClient Class Methods

Method	Description
<code>saveNew()</code>	Adds a new row in Cassandra
<code>saveNewInBatch()</code>	Adds multiple rows
<code>findAll()</code>	Finds all rows
<code>findAllSpecifiedIds()</code>	Finds all rows for specified IDs
<code>findById()</code>	Finds a single row by ID
<code>findAllByCql()</code>	Finds all rows by CQL
<code>findOneByCql()</code>	Finds one row by CQL
<code>countRows()</code>	Counts the number of rows
<code>exists()</code>	Finds if a specific row ID exists
<code>update()</code>	Updates a row
<code>updateInBatch()</code>	Updates multiple rows
<code>deleteById()</code>	Deletes a row by ID
<code>deleteByIdInBatch()</code>	Deletes all rows by ID
<code>delete()</code>	Deletes a single row
<code>deleteInBatch()</code>	Deletes a batch of rows

In subsequent sections, you will invoke these methods for CRUD operations. Comment out the method invocations not to be run in an application. For example, to invoke only the `saveNew()` method, uncomment the `saveNew()` method and comment out method invocations for all other methods when the application is run.

Save Operations

The `CassandraOperations` interface provides several methods for adding new row(s) to Cassandra. These are listed in Table 10.8.

Table 10.8 CassandraOperations Interface Methods for Adding New Rows

Method	Description
<code>saveNew(T entity)</code>	Adds a new row
<code>saveNewInBatch(Iterable<T> entities)</code>	Adds a batch of new rows

In the `saveNew()` method, create an instance of the entity class `Catalog`.

```
Catalog catalog1 = new Catalog("catalog1", "Oracle Magazine",
    "Oracle Publishing", "November-December 2013",
    "Engineering as a Service", "David A. Kelly");
```

Invoke the `saveNew(T entity)` method in `CassandraOperations` to save the `Catalog` entity instance.

```
ops.saveNew(catalog1);
```

Run the `CassandraClient` application to invoke the `saveNew()` method and save a new row in the `catalog` table. Then run the following command in `Cassandra-Cli`:

```
list catalog;
```

The output lists the row added, as shown in Figure 10.20.

```
[default@springdata ~] $ list catalog;
Using default limit of 100
Using default cell limit of 100
RowKey: catalog1
=> {name=author, value=David A. Kelly, timestamp=1390851814296000}
=> {name=edition, value=November-December 2013, timestamp=1390851814296000}
=> {name=journal, value=Oracle Magazine, timestamp=1390851814296000}
=> {name=publisher, value=Oracle Publishing, timestamp=1390851814296000}
=> {name=title, value=Engineering as a Service, timestamp=1390851814296000}

1 Row Returned.
Elapsed time: 4531 msec<s>.
[default@springdata ~]
```

Figure 10.20
Listing the new row added.

Source: Microsoft Corporation.

In the `saveNewInBatch()` method, use the `saveNewInBatch(Iterable<T> entities)` method to save a batch of rows. First create a `HashSet` instance, in which you will add the `Catalog` entity instances.

```
HashSet<Catalog> entities = new HashSet();
```

Create instances of the `Catalog` entity and add the entity instances to the `HashSet` using the `add(E e)` method.

```
Catalog catalog2 = new Catalog("catalog2", "Oracle Magazine",  
"Oracle Publishing", "November-December 2013",  
"Quintessential and Collaborative", "Tom Haunert");  
Catalog catalog3 = new Catalog("catalog3", "Oracle Magazine",  
"Oracle Publishing", "November-December 2013", "", "");  
Catalog catalog4 = new Catalog("catalog4", "Oracle Magazine",  
"Oracle Publishing", "November-December 2013", "", "");  
Catalog catalog5 = new Catalog("catalog5", "Oracle Magazine",  
"Oracle Publishing", "November-December 2013", "", "");  
Catalog catalog6 = new Catalog("catalog6", "Oracle Magazine",  
"Oracle Publishing", "November-December 2013", "", "");  
entities.add(catalog2);  
entities.add(catalog3);  
entities.add(catalog4);  
entities.add(catalog5);  
entities.add(catalog6);
```

Invoke the `saveNewInBatch(Iterable<T> entities)` method to save the `HashSet`. The batch save is not applied until the `saveNewInBatch()` method is invoked on the entities.

```
ops.saveNewInBatch(entities);
```

Then run the `list catalog` command in Cassandra-Cli to list the batch of rows added, as shown in Figure 10.21.

```
[default@springdata1 list catalog;
Using default limit of 100
Using default cell limit of 100

RowKey: catalog1
=> <name=author, value=David A. Kelly, timestamp=1390854105005000>
=> <name=edition, value=November-December 2013, timestamp=1390854105005000>
=> <name=journal, value=Oracle Magazine, timestamp=1390854105005000>
=> <name=publisher, value=Oracle Publishing, timestamp=1390854105005000>
=> <name=title, value=Engineering as a Service, timestamp=1390854105005000>

RowKey: catalog2
=> <name=author, value=Tom Haunert, timestamp=1390854105014000>
=> <name=edition, value=November-December 2013, timestamp=1390854105014000>
=> <name=journal, value=Oracle Magazine, timestamp=1390854105014000>
=> <name=publisher, value=Oracle Publishing, timestamp=1390854105014000>
=> <name=title, value=Quintessential and Collaborative, timestamp=1390854105014000>

RowKey: catalog3
=> <name=author, value=, timestamp=1390854105014000>
=> <name=edition, value=November-December 2013, timestamp=1390854105014000>
=> <name=journal, value=Oracle Magazine, timestamp=1390854105014000>
=> <name=publisher, value=Oracle Publishing, timestamp=1390854105014000>
=> <name=title, value=, timestamp=1390854105014000>

RowKey: catalog4
=> <name=author, value=, timestamp=1390854105014000>
=> <name=edition, value=November-December 2013, timestamp=1390854105014000>
=> <name=journal, value=Oracle Magazine, timestamp=1390854105014000>
=> <name=publisher, value=Oracle Publishing, timestamp=1390854105014000>
=> <name=title, value=, timestamp=1390854105014000>

RowKey: catalog5
=> <name=author, value=, timestamp=1390854105014000>
=> <name=edition, value=November-December 2013, timestamp=1390854105014000>
=> <name=journal, value=Oracle Magazine, timestamp=1390854105014000>
=> <name=publisher, value=Oracle Publishing, timestamp=1390854105014000>
=> <name=title, value=, timestamp=1390854105014000>

RowKey: catalog6
=> <name=author, value=, timestamp=1390854105014000>
=> <name=edition, value=November-December 2013, timestamp=1390854105014000>
=> <name=journal, value=Oracle Magazine, timestamp=1390854105014000>
=> <name=publisher, value=Oracle Publishing, timestamp=1390854105014000>
=> <name=title, value=, timestamp=1390854105014000>

6 Rows Returned.
Elapsed time: 81 msec(s).
[default@springdata1]
```

Figure 10.21

Listing the batch of rows added.

Source: Microsoft Corporation.

Find Operations

The `CassandraOperations` interface provides several methods to find row(s) from Cassandra, as listed in Table 10.9.

Table 10.9 CassandraOperations Interface Methods for Finding Rows

Method	Description
<code>find(Class<T> entityClass, String cql)</code>	Finds a single entity instance using CQL query
<code>findAll(Class<T> entityClass)</code>	Finds all entity instances
<code>findAll(Class<T> entityClass, Iterable<?> ids)</code>	Finds entity instances for the specified row IDs
<code>findById(Class<T> entityClass, Object id)</code>	Finds the entity instance for the specified row ID

In this section, you will find the rows added to Cassandra using the different find methods in `CassandraOperations`. In the `findAll()` method in `CassandraClient`, invoke the `findAll(Class<T> entityClass)` method in `CassandraOperations` with `Catalog.class` as argument. This method will return a list, from which you will get an `Iterator` to use over the result set.

```
Iterator<Catalog> iter = ops.findAll(Catalog.class).iterator();
```

Using a while loop, iterate over the result set and output the column values for each of the rows.

```
while (iter.hasNext()) {
    Catalog catalog = iter.next();
    System.out.println(catalog.getKey());
    System.out.println(catalog.getJournal());
    System.out.println(catalog.getPublisher());
    System.out.println(catalog.getEdition());
    System.out.println(catalog.getTitle());
    System.out.println(catalog.getAuthor());
}
```

Invoke the `findAll()` method from the `main` method to output the rows stored in Cassandra, as shown in Figure 10.22.

The screenshot shows a Java application window titled "Console". The output is a list of two rows from a Cassandra database. Each row contains the following fields:

- Table name: catalog
- catalog1
- Oracle Magazine
- Oracle Publishing
- November-December 2013
- Engineering as a Service
- David A. Kelly
- catalog2
- Oracle Magazine
- Oracle Publishing
- November-December 2013
- Quintessential and Collaborative

Figure 10.22

Finding all rows.

Source: Eclipse Foundation.

In the `findAllSpecifiedIds()` method in `CassandraClient`, invoke the `findAll(Class<T> entityClass, Iterable<?> ids)` method in `CassandraOperations` with `Catalog.class` as the first argument and a `HashSet` of row IDs as the second argument. This method will return a list, from which you will get an `Iterator` to use over the result set.

```
HashSet<String> ids = new HashSet();
ids.add("catalog1");
ids.add("catalog2");
Iterator<Catalog> iter = ops.findAll(Catalog.class, ids).iterator();
```

Using a while loop, iterate over the result set and output the column values for each of the rows.

```
while (iter.hasNext()) {
    Catalog catalog = iter.next();
    System.out.println(catalog.getKey());
    System.out.println(catalog.getJournal());
    System.out.println(catalog.getPublisher());
```

```
        System.out.println(catalog.getEdition());
        System.out.println(catalog.getTitle());
        System.out.println(catalog.getAuthor());
    }
```

Invoke the `findAllSpecifiedIds()` method from the `main` method to output the rows stored in Cassandra, as shown in Figure 10.23. The output for `findAllSpecifiedIds()` and `findAll` is the same because you specified all IDs in `findAllSpecifiedIds()`.

The screenshot shows a terminal window with the following text output:

```
Table name: catalog
catalog1

Oracle Magazine

Oracle Publishing

November-December 2013

Engineering as a Service

David A. Kelly
catalog2

Oracle Magazine

Oracle Publishing

November-December 2013
```

Figure 10.23
Finding Cassandra table rows by all specified IDs.

Source: Eclipse Foundation.

In the `findById()` method in `CassandraClient`, invoke the `findById(Class<T> entityClass, Object id)` method to find a row with `Catalog.class` as the first argument and "catalog1" as the second argument. This method will return a `Catalog` entity instance. Output the column values for the row selected.

```
System.out.println(catalog.getKey());
System.out.println(catalog.getJournal());
System.out.println(catalog.getPublisher());
System.out.println(catalog.getEdition());
System.out.println(catalog.getTitle());
System.out.println(catalog.getAuthor());
```

Invoke the `findById()` method from the `main` method to output the `catalog1` row stored in Cassandra, as shown in Figure 10.24.

Table name: catalog
catalog1
Oracle Magazine
Oracle Publishing
November-December 2013
Engineering as a Service
David A. Kelly

Figure 10.24

Finding a Cassandra table row by ID.

Source: Eclipse Foundation.

In the `findAllByCql()` method in `CassandraClient`, invoke the `find(Class<T> entityClass, String cql)` method in `CassandraOperations` to select rows using a CQL query. Specify `Catalog.class` as the first argument. As the second argument, specify a CQL query "SELECT * FROM catalog". Then invoke the `findAll()` method to return an `Iterator` for the result set.

```
Iterator<Catalog> iter = ops.findAll(Catalog.class).iterator();
```

Using a while loop, iterate over the result set and output the column values for each of the rows.

```
while (iter.hasNext()) {
    Catalog catalog = iter.next();
    System.out.println(catalog.getKey());
    System.out.println(catalog.getJournal());
    System.out.println(catalog.getPublisher());
    System.out.println(catalog.getEdition());
    System.out.println(catalog.getTitle());
    System.out.println(catalog.getAuthor());
}
```

Invoke the `findAllByCql()` method from the `main` method to output the rows stored in Cassandra, as shown in Figure 10.25. The output for `findAllByCql()` is the same as for `findAllSpecifiedIds()` and `findAll`.

The screenshot shows the Eclipse IDE's Console view with the title "Console" and the path "CassandraClient [Java Application] C:\Program". The output of the CQL query "SELECT * from catalog WHERE key='catalog1'" is displayed, showing two rows of data.

key	journal	publisher	edition	title	author
catalog1	Oracle Magazine	Oracle Publishing	November-December 2013	Engineering as a Service	David A. Kelly
catalog2	Oracle Magazine	Oracle Publishing	November-December 2013	Information and Collaboration	

Figure 10.25
Finding all table rows by CQL.

Source: Eclipse Foundation.

In the `findOneByCql()` method in `CassandraClient`, invoke the `findOne(Class<T> entityClass, String cql)` method to find a row with `Catalog.class` as the first argument and the CQL query "`SELECT * from catalog WHERE key='catalog1'`" as the second argument. Then execute the method to return a `Catalog` entity instance. Output the column values for the row selected.

```
System.out.println(catalog.getKey());
System.out.println(catalog.getJournal());
System.out.println(catalog.getPublisher());
System.out.println(catalog.getEdition());
System.out.println(catalog.getTitle());
System.out.println(catalog.getAuthor());
```

Invoke the `findOneByCql()` method from the `main` method to output the `catalog1` row stored in Cassandra, as shown in Figure 10.26. The output for `findOneByCql()` is the same as for `findById()` because you have specified the same ID, '`catalog1`'.

Table name: catalog
catalog1
Oracle Magazine
Oracle Publishing
November -December 2013
Engineering as a Service
David A. Kelly

Figure 10.26

Finding one table row by CQL.

Source: Eclipse Foundation.

Exists and Count Operations

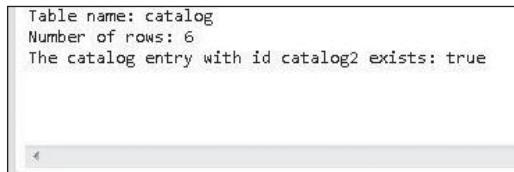
The `exists(Class<T> entityClass, Object id)` method in `CassandraOperations` finds whether an entity exists in the Cassandra database. In the `exists()` method in `CassandraClient`, invoke the `exists(Class<T> entityClass, Object id)` method with `Catalog.class` as the first argument and "catalog2" as the second argument to find out if the catalog2 ID exists in the Cassandra table catalog. Invoke the `execute()` method to run the operation. The `exists(Class<T> entityClass, Object id)` method returns a Boolean object. Invoke the `booleanValue()` method on the Boolean object to find if the catalog2 row exists.

```
System.out.println("The catalog entry with id catalog2 exists: "+ ops.exists(Catalog.class, "catalog2"));
```

The `countAll(Class<T> entityClass)` method in `CassandraOperations` returns Long for the number of rows for a specified entity. In the `countRows()` method in `CassandraClient`, invoke the `countAll(Class<T> entityClass)` method with `Catalog.class` as the argument.

```
System.out.println("Number of rows: " + ops.countAll(Catalog.class));
```

In the next run of the `CassandraClient` application, invoke the `exists()` method and the `countRows()` method. The output indicates that Cassandra has six rows and that the catalog2 row exists, as shown in Figure 10.27.

**Figure 10.27**

Finding if a Cassandra table row exists.

Source: Eclipse Foundation.

Update Operations

The `CassandraOperations` interface provides two methods for updating row(s) to Cassandra, as listed in Table 10.10.

Table 10.10 CassandraOperations Interface Methods for Updating Rows

Method	Description
<code>save(T entity)</code>	Updates a row
<code>saveInBatch(Iterable<T> entities)</code>	Updates a batch of rows

In the `update()` method in `CassandraClient` creates an instance of `Catalog` with the updated column values.

```
Catalog catalog1 = new Catalog("catalog1", "Oracle Magazine", "Oracle-Publishing",
"11/12 2013", "Engineering as a Service", "Kelly, David A.");
```

Invoke the `save(T entity)` method to update the `catalog1` row.

```
ops.save(catalog1);
```

Uncomment the `update()` method invocation in the `main` method. When the application is run, the `catalog1` row is updated. Then run the `list catalog` command in `cassandra-cli` to list the updated row `catalog1`, as shown in Figure 10.28.

```
[default@springdata1 list catalog;
Using default limit of 100
Using default cell limit of 100

RowKey: catalog
=> <name=author, value=Kelly, David A., timestamp=1390855535029000>
=> <name=edition, value=11/12 2013, timestamp=1390855535029000>
=> <name=journal, value=Oracle Magazine, timestamp=1390855535029000>
=> <name=publisher, value=Oracle-Publishing, timestamp=1390855535029000>
=> <name=title, value=Engineering as a Service, timestamp=1390855535029000>
```

Figure 10.28

Listing the updated row.

Source: Microsoft Corporation.

In the updateInBatch() method in CassandraClient, create two instances of Catalog with the updated column values.

```
Catalog catalog2 = new Catalog("catalog2", "Oracle Magazine",
"Oracle Publishing", "November-December 2013",
"Quintessential and Collaborative", "Haunert, Tom");

Catalog catalog3 = new Catalog("catalog3", "Oracle Magazine",
"Oracle-Publishing", "Nov-Dec 2013", "", "");
```

Create a HashSet and add the entity instances to it.

```
HashSet<Catalog> entities = new HashSet();
entities.add(catalog2);
entities.add(catalog3);
```

Invoke the saveNewInBatch(Iterable<T> entities) method to save the HashSet object. Uncomment the updateInBatch() method invocation in the main method. When the application is run, the catalog2 and catalog3 rows are updated. Next, run the list catalog command in Cassandra-Cli to list the updated rows catalog2 and catalog3, as shown in Figure 10.29.

```
RowKey: catalog2
=> <name=author, value=Haunert, Tom, timestamp=1390855960014000>
=> <name=edition, value=November-December 2013, timestamp=1390855960014000>
=> <name=journal, value=Oracle Magazine, timestamp=1390855960014000>
=> <name=publisher, value=Oracle Publishing, timestamp=1390855960014000>
=> <name=title, value=Quintessential and Collaborative, timestamp=1390855960014000>

RowKey: catalog3
=> <name=author, value=, timestamp=1390855960014000>
=> <name=edition, value=Nov-Dec 2013, timestamp=1390855960014000>
=> <name=journal, value=Oracle Magazine, timestamp=1390855960014000>
=> <name=publisher, value=Oracle-Publishing, timestamp=1390855960014000>
=> <name=title, value=, timestamp=1390855960014000>
```

Figure 10.29

Listing table rows updated in batch.

Source: Microsoft Corporation.

Remove Operations

`CassandraOperations` provides several methods for removing row(s) from Cassandra, as listed in Table 10.11.

Table 10.11 `CassandraOperations` Interface Methods for Removing Rows

Method	Description
<code>delete(T entity)</code>	Deletes a single entity instance
<code>deleteAll(Class<T> entityClass)</code>	Deletes all entity instances
<code>deleteById(Class<T> entityClass, Object id)</code>	Deletes a single instance by ID
<code>deleteByIdInBatch(Class<T> entityClass, Iterable<?> ids)</code>	Deletes a batch of instances for specified IDs
<code>deleteInBatch(Iterable<T> entities)</code>	Deletes a batch of instances

In the `deleteById()` method, invoke the `deleteById(Class<T> entityClass, Object id)` method with `Catalog.class` as the first argument and `catalog3` as the second argument.

```
ops.deleteById(Catalog.class, "catalog3");
```

When the `CassandraClient` application is run with the `deleteById()` method invocation uncommented, the `catalog3` row is deleted from the `catalog` table. Next, run the `list catalog` command in `Cassandra-Cli` to list the `catalog3` row columns as deleted, as shown in Figure 10.30.

```
RowKey: catalog3
=> <name=author, value=null, timestamp=1390854105014000>
=> <name=edition, value=null, timestamp=1390854105014000>
=> <name=journal, value=null, timestamp=1390854105014000>
=> <name=publisher, value=null, timestamp=1390854105014000>
=> <name=title, value=null, timestamp=1390854105014000>

RowKey: catalog4
=> <name=author, value=null, timestamp=1390854105014000>
=> <name=edition, value=null, timestamp=1390854105014000>
=> <name=journal, value=null, timestamp=1390854105014000>
=> <name=publisher, value=null, timestamp=1390854105014000>
```

Figure 10.30
Listing deleted rows by ID.

Source: Microsoft Corporation.

In the `deleteByIdInBatch()` method in `CassandraClient`, invoke the `deleteByIdInBatch(Class<T> entityClass, Iterable<?> ids)` method with `Catalog.class` as the first argument and a `HashSet` of IDs consisting of `catalog1` and `catalog2` as the second argument.

```
HashSet<String> ids = new HashSet();
ids.add("catalog1");
ids.add("catalog2");
ops.deleteByIdInBatch(Catalog.class, ids);
```

In the `delete()` method in `CassandraClient`, invoke the `delete(T entity)` with a `Catalog` instance for the `catalog4` ID as the argument.

```
Catalog catalog4 = new Catalog("catalog4", "Oracle Magazine",
"Oracle Publishing", "November-December 2013", "", "");
ops.delete(catalog4);
```

When the `CassandraClient` application is run with the `deleteByIdInBatch()` method and `delete()` method invocations uncommented, the `catalog1` and `catalog2` rows are deleted from the `catalog` table. The `catalog4` ID is also deleted. Next, run the `list catalog` command in `Cassandra-Cli` to list the `catalog1`, `catalog2`, `catalog3`, and `catalog4` row columns as deleted. (The `catalog3` row column was deleted earlier using the `deleteById()` method.) See Figure 10.31.

```
Using default cell limit of 100
RowKey: catalog1
RowKey: catalog2
RowKey: catalog3
RowKey: catalog4
```

Figure 10.31
Listing rows deleted by ID in a batch.

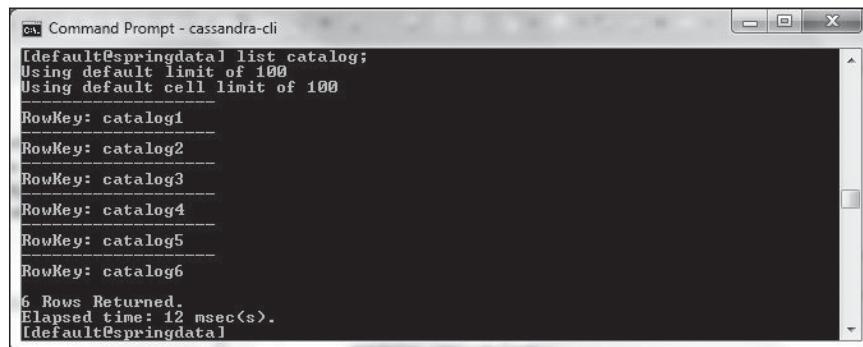
Source: Microsoft Corporation.

In the `deleteInBatch()` method in `CassandraClient`, invoke the `deleteInBatch(Iterable<T> entities)` method with a `HashSet` of entities consisting of `catalog5` and `catalog6` as the second argument. Then invoke the `execute()` method to apply the deletion.

```
HashSet<Catalog> entities = new HashSet();
Catalog catalog5 = new Catalog("catalog5", "Oracle Magazine",
"Oracle Publishing", "November-December 2013", "", "");
Catalog catalog6 = new Catalog("catalog6", "Oracle Magazine",
```

```
"Oracle Publishing", "November-December 2013", "", "");  
entities.add(catalog5);  
entities.add(catalog6);  
ops.deleteInBatch(entities);
```

Next, run the `list catalog` command in Cassandra-Cli to list the `catalog5` and `catalog6` row columns as deleted in addition to the other catalog IDs deleted earlier, as shown in Figure 10.32.



```
[default@springdata list catalog;  
Using default limit of 100  
Using default cell limit of 100  
RowKey: catalog1  
RowKey: catalog2  
RowKey: catalog3  
RowKey: catalog4  
RowKey: catalog5  
RowKey: catalog6  
6 Rows Returned.  
Elapsed time: 12 msec(s).  
[default@springdata]
```

Figure 10.32

Listing rows deleted in batch.

Source: Microsoft Corporation.

The CassandraClient application appears in Listing 10.4.

Listing 10.4 The CassandraClient Application

```
package com.cassandra.core;  
  
import java.util.HashSet;  
import java.util.Iterator;  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.annotation.AnnotationConfigApplicationContext;  
import com.cassandra.config.SpringCassandraApplicationConfig;  
import com.cassandra.model.Catalog;  
import org.springframework.data.cassandra.core.CassandraOperations;  
import org.springframework.cql.core.RingMember;  
  
public class CassandraClient {  
  
    static CassandraOperations ops;  
  
    public static void main(String[] args) {  
        ApplicationContext context = new AnnotationConfigApplicationContext(  
            SpringCassandraApplicationConfig.class);  
        ops = context.getBean(CassandraOperations.class);  
        ops.deleteInBatch(entities);  
    }  
}
```

```
        SpringCassandraApplicationConfig.class);
ops = context.getBean(CassandraOperations.class);
//      for (RingMember member : ops.getCqlOperations().describeRing()) {
//          System.out.println(member.toString());
//      }
System.out.println("Table name: " + ops.getTableName(Catalog.class));
//      saveNew();
//      saveNewInBatch();
//      findAll();
//      findAllSpecifiedIds();
//      findById();
//      findAllByCql();
//      findOneByCql();
//      countRows();
//      exists();
//      update();
//      updateInBatch();
//      deleteById();
//      deleteByIdInBatch();
//      delete();
//      deleteInBatch();
}
private static void saveNew() {
    Catalog catalog1 = new Catalog("catalog1", "Oracle Magazine",
        "Oracle Publishing", "November-December 2013",
        "Engineering as a Service", "David A. Kelly");
    ops.saveNew(catalog1);
}
private static void saveNewInBatch() {
    HashSet<Catalog> entities = new HashSet();
    Catalog catalog2 = new Catalog("catalog2", "Oracle Magazine",
        "Oracle Publishing", "November-December 2013",
        "Quintessential and Collaborative", "Tom Haunert");
    Catalog catalog3 = new Catalog("catalog3", "Oracle Magazine",
        "Oracle Publishing", "November-December 2013", "", "");
    Catalog catalog4 = new Catalog("catalog4", "Oracle Magazine",
        "Oracle Publishing", "November-December 2013", "", "");
    Catalog catalog5 = new Catalog("catalog5", "Oracle Magazine",
        "Oracle Publishing", "November-December 2013", "", "");
    Catalog catalog6 = new Catalog("catalog6", "Oracle Magazine",
        "Oracle Publishing", "November-December 2013", "", "");
```

```
entities.add(catalog2);
entities.add(catalog3);
entities.add(catalog4);
entities.add(catalog5);
entities.add(catalog6);
ops.saveNewInBatch(entities);
}

private static void countRows() {
    System.out.println("Number of rows: " + ops.countAll(Catalog.class));
}

private static void exists() {
    Catalog catalog3 = new Catalog("catalog1", "Oracle Magazine",
        "Oracle Publishing", "November-December 2013", "", "");
    //System.out.println("The catalog3 entity exists: "+
ops.exists(catalog3);
    //System.out.println("\n");
    System.out.println("The catalog entry with id catalog2 exists: " +
ops.exists(Catalog.class, "catalog2"));
}

private static void findAll() {
    Iterator<Catalog> iter = ops.findAll(Catalog.class).iterator();
    while (iter.hasNext()) {
        Catalog catalog = iter.next();
        System.out.println(catalog.getKey());
        System.out.println("\n");
        System.out.println(catalog.getJournal());
        System.out.println("\n");
        System.out.println(catalog.getPublisher());
        System.out.println("\n");
        System.out.println(catalog.getEdition());
        System.out.println("\n");
        System.out.println(catalog.getTitle());
        System.out.println("\n");
        System.out.println(catalog.getAuthor());
    }
}

private static void findAllSpecifiedIds() {
    HashSet<String> ids = new HashSet();
    ids.add("catalog1");
```

```
ids.add("catalog2");
Iterator<Catalog> iter = ops.findAll(Catalog.class, ids).iterator();

while (iter.hasNext()) {
    Catalog catalog = iter.next();
    System.out.println(catalog.getKey());
    System.out.println("\n");
    System.out.println(catalog.getJournal());
    System.out.println("\n");
    System.out.println(catalog.getPublisher());
    System.out.println("\n");
    System.out.println(catalog.getEdition());
    System.out.println("\n");
    System.out.println(catalog.getTitle());
    System.out.println("\n");
    System.out.println(catalog.getAuthor());
}
}

private static void findById() {
    Catalog catalog = ops.findById(Catalog.class, "catalog1");
    System.out.println(catalog.getKey());
    System.out.println("\n");
    System.out.println(catalog.getJournal());
    System.out.println("\n");
    System.out.println(catalog.getPublisher());
    System.out.println("\n");
    System.out.println(catalog.getEdition());
    System.out.println("\n");
    System.out.println(catalog.getTitle());
    System.out.println("\n");
    System.out.println(catalog.getAuthor());
}

private static void findAllByCql() {
    Iterator<Catalog> iter = ops.find(Catalog.class,
        "SELECT * FROM catalog").iterator();
    while (iter.hasNext()) {
        Catalog catalog = iter.next();
        System.out.println(catalog.getKey());
        System.out.println("\n");
        System.out.println(catalog.getJournal());
        System.out.println("\n");
        System.out.println(catalog.getPublisher());
```

```
        System.out.println("\n");
        System.out.println(catalog.getEdition());
        System.out.println("\n");
        System.out.println(catalog.getTitle());
        System.out.println("\n");
        System.out.println(catalog.getAuthor());
    }
}

private static void findOneByCql() {
    Catalog catalog = ops.findOne(Catalog.class,
        "SELECT * from catalog WHERE key='catalog1'");
    System.out.println(catalog.getKey());
    System.out.println("\n");
    System.out.println(catalog.getJournal());
    System.out.println("\n");
    System.out.println(catalog.getPublisher());
    System.out.println("\n");
    System.out.println(catalog.getEdition());
    System.out.println("\n");
    System.out.println(catalog.getTitle());
    System.out.println("\n");
    System.out.println(catalog.getAuthor());
}
private static void update() {
    Catalog catalog1 = new Catalog("catalog1", "Oracle Magazine",
        "Oracle-Publishing", "11/12 2013", "Engineering as a Service",
        "Kelly, David A.");
    ops.save(catalog1);
}
private static void updateInBatch() {
    HashSet<Catalog> entities = new HashSet();
    Catalog catalog2 = new Catalog("catalog2", "Oracle Magazine",
        "Oracle Publishing", "November-December 2013",
        "Quintessential and Collaborative", "Haunert, Tom");
    Catalog catalog3 = new Catalog("catalog3", "Oracle Magazine",
        "Oracle-Publishing", "Nov-Dec 2013", "", "");
    entities.add(catalog2);
    entities.add(catalog3);
    ops.saveInBatch(entities);
}
```

```
private static void deleteById() {
    ops.deleteById(Catalog.class, "catalog3");
}

private static void deleteByIdInBatch() {
    HashSet<String> ids = new HashSet();
    ids.add("catalog1");
    ids.add("catalog2");
    ops.deleteByIdInBatch(Catalog.class, ids);
}

private static void delete() {
    Catalog catalog4 = new Catalog("catalog4", "Oracle Magazine",
        "Oracle Publishing", "November-December 2013", "", "");
    ops.delete(catalog4);
}

private static void deleteInBatch() {
    HashSet<Catalog> entities = new HashSet();
    Catalog catalog5 = new Catalog("catalog5", "Oracle Magazine",
        "Oracle Publishing", "November-December 2013", "", "");
    Catalog catalog6 = new Catalog("catalog6", "Oracle Magazine",
        "Oracle Publishing", "November-December 2013", "", "");
    entities.add(catalog5);
    entities.add(catalog6);
    ops.deleteInBatch(entities);
}
}
```

SUMMARY

In this chapter, you used the Spring Data project for Cassandra to run CRUD operations in Apache Cassandra using a Maven project.