Learning Apache Cassandra

Cassandra is a distributed database that stands out for its robust feature set and intuitive interface, while still providing the high availability and scalability of a distributed store.

Starting from installing Cassandra and creating your first keyspace, to mastering the different table structures Cassandra offers and exploring the latest and most powerful features of the Cassandra Query Language, CQL2, this book explores each topic through the lens of a real-world example application. With plenty of examples, tips, and clear explanations, you’ll master compound primary keys, collection columns, lightweight transactions, and many other advanced aspects of Cassandra.

By the end of the book, you’ll be fully equipped to build powerful, scalable Cassandra database layers for your applications.

Who this book is written for

If you’re an application developer familiar with SQL databases such as MySQL or Postgres, and you want to explore distributed databases such as Cassandra, this is the perfect guide for you. Even if you’ve never worked with a distributed database before, Cassandra’s intuitive programming interface coupled with the step-by-step examples in this book will have you building highly scalable persistence layers for your applications in no time.

What you will learn from this book

- Install Cassandra and create your first keyspace
- Choose the right table structure for the task at hand in a variety of scenarios
- Use range slice queries for efficient data access
- Effortlessly handle concurrent updates with collection columns
- Ensure data integrity with lightweight transactions and logged batches
- Understand eventual consistency and use the right consistency level for your situation
- Implement best practices for data modeling and access
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 "Getting Up and Running with Cassandra"
- A synopsis of the book’s content
- More information on Learning Apache Cassandra

About the Author

Mat Brown is a professional software engineer in Brooklyn, New York. In his career, he has focused on building consumer-facing web and mobile applications for several companies; he currently works at Genius.

Mat is an enthusiastic contributor to the Ruby open source ecosystem. He is the maintainer of Cequel, a Ruby object mapper for Cassandra, and was the original author of Sunspot, a library that seamlessly integrates Solr search into Rails applications.

I would like to thank my wife, Pamela, and my parents for their love and support. I’d also like to thank my friends, especially those who seemed impressed when I told them I was writing a book. My cat, Taco, though not good for much, did keep me company during some writing sessions and thus deserves a mention here as well.
Learning Apache Cassandra

The crop of distributed databases that has come to the market in recent years appeals to application developers for several reasons. Their storage capacity is nearly limitless, bounded only by the number of machines you can afford to spin up. Masterless replication makes them resilient to adverse events, handling even a complete machine failure without any noticeable effect on the applications that rely on them. Log-structured storage engines allow these databases to handle highvolume write loads without blinking an eye.

But compared to traditional relational databases, not to mention newer document stores, distributed databases are typically feature-poor and inconvenient to work with. Read and write functionality is frequently confined to simple keyvalue operations, with more complex operations demanding arcane map-reduce implementations. Happily, Cassandra provides all of the benefits of a fully-distributed data store while also exposing a familiar, user-friendly data model and query interface.

I first encountered Cassandra working on an application that stored our users' extended social graphs across a variety of services. With a hundred or so alpha users in the system, it became clear that, even at relatively modest traction, our storage needs would go beyond what our PostgreSQL database could comfortably handle. After surveying the landscape of horizontally scalable data stores, we decided to migrate to Cassandra because its table-based data structure seemed to provide an easy migration path from the application we had already built.

Our first deployment of Cassandra ported our previous PostgreSQL schema tablefor-table. Only after taking the application to production did we come to realize that our expertise designing schemas for a relational world didn't map directly to a distributed store such as Cassandra. While we were happy to be storing tons of data at very high write volumes, it was clear we weren't getting maximum performance out of the database.

The story has a happy ending: after a rough initial launch, we went back to the drawing board and reworked our data model from the ground up to take advantage of Cassandra's strengths. With that new version deployed, Cassandra effortlessly handled our application scaling to hundreds of thousands of users' social graphs.

The goal of this book is to teach you the easy way what we learned the hard way: how to use Cassandra effectively, powerfully, and efficiently. We'll explore Cassandra's ins and outs by designing the persistence layer for a messaging service that allows users to post status updates that are visible to their friends. By the end of the book, you'll be fully prepared to build your own highly scalable, highly available applications.
What This Book Covers

Chapter 1, *Getting Up and Running with Cassandra*, introduces the major reasons to choose Cassandra over a traditional relational or document database. It then provides step-by-step instructions on installing Cassandra, creating a keyspace, and interacting with the database using the CQL language and cqlsh tool.

Chapter 2, *The First Table*, is a walk-through of creating a table, inserting data, and retrieving rows by primary key. Along the way, it discusses how Cassandra tables are structured, and provides a tour of the Cassandra type system.

Chapter 3, *Organizing Related Data*, introduces more complex table structures that group related data together using compound primary keys.

Chapter 4, *Beyond Key-Value Lookup*, puts the more robust schema developed in the previous chapter to use, explaining how to query for sorted ranges of rows.

Chapter 5, *Establishing Relationships*, develops table structures for modeling relationships between rows. The chapter introduces static columns and row deletion.

Chapter 6, *Denormalizing Data for Maximum Performance*, explains when and why storing multiple copies of the same data can make your application more efficient.

Chapter 7, *Expanding Your Data Model*, demonstrates the use of lightweight transactions to ensure data integrity. It also introduces schema alteration, row updates, and single-column deletion.

Chapter 8, *Collections, Tuples, and User-defined Types*, introduces collection columns and explores Cassandra's support for advanced, atomic collection manipulation. It also introduces tuples and user-defined types.

Chapter 9, *Aggregating Time-Series Data*, covers the common use case of collecting high-volume time-series data and introduces counter columns.

Chapter 10, *How Cassandra Distributes Data*, explores what happens when you save a row to Cassandra. It considers eventual consistency and teaches you how to use tunable consistency to get the right balance between consistency and fault-tolerance.

Appendix A, *Peeking Under the Hood*, peels away the abstractions provided by CQL to reveal how Cassandra represents data at the lower column family level.

Appendix B, *Authentication and Authorization*, introduces ways to control access to your Cassandra cluster and specific data structures within it.
As an application developer, you have almost certainly worked with databases extensively. You must have built products using relational databases like MySQL and PostgreSQL, and perhaps experimented with a document store like MongoDB or a key-value database like Redis. While each of these tools has its strengths, you will now consider whether a distributed database like Cassandra might be the best choice for the task at hand.

In this chapter, we'll talk about the major reasons to choose Cassandra from among the many database options available to you. Having established that Cassandra is a great choice, we'll go through the nuts and bolts of getting a local Cassandra installation up and running. By the end of this chapter, you'll know:

- When and why Cassandra is a good choice for your application
- How to install Cassandra on your development machine
- How to interact with Cassandra using cqlsh
- How to create a keyspace

**What Cassandra offers, and what it doesn't**

Cassandra is a fully distributed, masterless database, offering superior scalability and fault tolerance to traditional single master databases. Compared with other popular distributed databases like Riak, HBase, and Voldemort, Cassandra offers a uniquely robust and expressive interface for modeling and querying data. What follows is an overview of several desirable database capabilities, with accompanying discussions of what Cassandra has to offer in each category.
Horizontal scalability

Horizontal scalability refers to the ability to expand the storage and processing capacity of a database by adding more servers to a database cluster. A traditional single-master database's storage capacity is limited by the capacity of the server that hosts the master instance. If the data set outgrows this capacity, and a more powerful server isn't available, the data set must be sharded among multiple independent database instances that know nothing of each other. Your application bears responsibility for knowing to which instance a given piece of data belongs.

Cassandra, on the other hand, is deployed as a cluster of instances that are all aware of each other. From the client application's standpoint, the cluster is a single entity; the application need not know, nor care, which machine a piece of data belongs to. Instead, data can be read or written to any instance in the cluster, referred to as a node; this node will forward the request to the instance where the data actually belongs.

The result is that Cassandra deployments have an almost limitless capacity to store and process data; when additional capacity is required, more machines can simply be added to the cluster. When new machines join the cluster, Cassandra takes care of rebalancing the existing data so that each node in the expanded cluster has a roughly equal share.

Cassandra is one of the several popular distributed databases inspired by the Dynamo architecture, originally published in a paper by Amazon. Other widely used implementations of Dynamo include Riak and Voldemort. You can read the original paper at http://s3.amazonaws.com/AllThingsDistributed/sosp/amazon-dynamo-sosp2007.pdf.

High availability

The simplest database deployments are run as a single instance on a single server. This sort of configuration is highly vulnerable to interruption: if the server is affected by a hardware failure or network connection outage, the application's ability to read and write data is completely lost until the server is restored. If the failure is catastrophic, the data on that server might be lost completely.

A master-follower architecture improves this picture a bit. The master instance receives all write operations, and then these operations are replicated to follower instances. The application can read data from the master or any of the follower instances, so a single host becoming unavailable will not prevent the application from continuing to read data. A failure of the master, however, will still prevent the application from performing any write operations, so while this configuration provides high read availability, it doesn't completely provide high availability.
Cassandra, on the other hand, has no **single point of failure** for reading or writing data. Each piece of data is replicated to multiple nodes, but none of these nodes holds the authoritative master copy. If a machine becomes unavailable, Cassandra will continue writing data to the other nodes that share data with that machine, and will queue the operations and update the failed node when it rejoins the cluster. This means in a typical configuration, two nodes must fail simultaneously for there to be any application-visible interruption in Cassandra's availability.

**How many copies?**

When you create a keyspace—Cassandra's version of a database—you specify how many copies of each piece of data should be stored; this is called the **replication factor**. A replication factor of 3 is a common and good choice for many use cases.

**Write optimization**

Traditional relational and document databases are optimized for read performance. Writing data to a relational database will typically involve making in-place updates to complicated data structures on disk, in order to maintain a data structure that can be read efficiently and flexibly. Updating these data structures is a very expensive operation from a standpoint of disk I/O, which is often the limiting factor for database performance. Since writes are more expensive than reads, you'll typically avoid any unnecessary updates to a relational database, even at the expense of extra read operations.

Cassandra, on the other hand, is highly optimized for write throughput, and in fact never modifies data on disk; it only appends to existing files or creates new ones. This is much easier on disk I/O and means that Cassandra can provide astonishingly high write throughput. Since both writing data to Cassandra, and storing data in Cassandra, are inexpensive, **denormalization** carries little cost and is a good way to ensure that data can be efficiently read in various access scenarios.

Because Cassandra is optimized for write volume, you shouldn't shy away from writing data to the database. In fact, it's most efficient to **write without reading** whenever possible, even if doing so might result in redundant updates.

Just because Cassandra is optimized for writes doesn't make it bad at reads; in fact, a well-designed Cassandra database can handle very heavy read loads with no problem. We'll cover the topic of efficient data modeling in great depth in the next few chapters.
Structured records

The first three database features we looked at are commonly found in distributed data stores. However, databases like Riak and Voldemort are purely key-value stores; these databases have no knowledge of the internal structure of a record that's stored at a particular key. This means useful functions like updating only part of a record, reading only certain fields from a record, or retrieving records that contain a particular value in a given field are not possible.

Relational databases like PostgreSQL, document stores like MongoDB, and, to a limited extent, newer key-value stores like Redis do have a concept of the internal structure of their records, and most application developers are accustomed to taking advantage of the possibilities this allows. None of these databases, however, offer the advantages of a masterless distributed architecture.

In Cassandra, records are structured much in the same way as they are in a relational database—using tables, rows, and columns. Thus, applications using Cassandra can enjoy all the benefits of masterless distributed storage while also getting all the advanced data modeling and access features associated with structured records.

Secondary indexes

A secondary index, commonly referred to as an index in the context of a relational database, is a structure allowing efficient lookup of records by some attribute other than their primary key. This is a widely useful capability: for instance, when developing a blog application, you would want to be able to easily retrieve all of the posts written by a particular author. Cassandra supports secondary indexes; while Cassandra's version is not as versatile as indexes in a typical relational database, it's a powerful feature in the right circumstances.

Efficient result ordering

It's quite common to want to retrieve a record set ordered by a particular field; for instance, a photo sharing service will want to retrieve the most recent photographs in descending order of creation. Since sorting data on the fly is a fundamentally expensive operation, databases must keep information about record ordering persisted on disk in order to efficiently return results in order. In a relational database, this is one of the jobs of a secondary index.
In Cassandra, secondary indexes can’t be used for result ordering, but tables can be structured such that rows are always kept sorted by a given column or columns, called clustering columns. Sorting by arbitrary columns at read time is not possible, but the capacity to efficiently order records in any way, and to retrieve ranges of records based on this ordering, is an unusually powerful capability for a distributed database.

**Immediate consistency**

When we write a piece of data to a database, it is our hope that that data is immediately available to any other process that may wish to read it. From another point of view, when we read some data from a database, we would like to be guaranteed that the data we retrieve is the most recently updated version. This guarantee is called immediate consistency, and it’s a property of most common single-master databases like MySQL and PostgreSQL.

Distributed systems like Cassandra typically do not provide an immediate consistency guarantee. Instead, developers must be willing to accept eventual consistency, which means when data is updated, the system will reflect that update at some point in the future. Developers are willing to give up immediate consistency precisely because it is a direct tradeoff with high availability.

In the case of Cassandra, that tradeoff is made explicit through tunable consistency. Each time you design a write or read path for data, you have the option of immediate consistency with less resilient availability, or eventual consistency with extremely resilient availability. We’ll cover consistency tuning in great detail in Chapter 10, *How Cassandra Distributes Data*.

**Discretely writable collections**

While it’s useful for records to be internally structured into discrete fields, a given property of a record isn’t always a single value like a string or an integer. One simple way to handle fields that contain collections of values is to serialize them using a format like JSON, and then save the serialized collection into a text field. However, in order to update collections stored in this way, the serialized data must be read from the database, decoded, modified, and then written back to the database in its entirety. If two clients try to perform this kind of modification to the same record concurrently, one of the updates will be overwritten by the other.
For this reason, many databases offer built-in collection structures that can be discretely updated: values can be added to, and removed from collections, without reading and rewriting the entire collection. Cassandra is no exception, offering list, set, and map collections, and supporting operations like "append the number 3 to the end of this list". Neither the client nor Cassandra itself needs to read the current state of the collection in order to update it, meaning collection updates are also blazingly efficient.

**Relational joins**

In real-world applications, different pieces of data relate to each other in a variety of ways. Relational databases allow us to perform queries that make these relationships explicit, for instance, to retrieve a set of events whose location is in the state of New York (this is assuming events and locations are different record types). Cassandra, however, is not a relational database, and does not support anything like joins. Instead, applications using Cassandra typically denormalize data and make clever use of clustering in order to perform the sorts of data access that would use a join in a relational database.

For data sets that aren't already denormalized, applications can also perform client-side joins, which mimic the behavior of a relational database by performing multiple queries and joining the results at the application level. Client-side joins are less efficient than reading data that has been denormalized in advance, but offer more flexibility. We'll cover both of these approaches in Chapter 6, *Denormalizing Data for Maximum Performance*.

**MapReduce**

*MapReduce* is a technique for performing aggregate processing on large amounts of data in parallel; it's a particularly common technique in data analytics applications. Cassandra does not offer built-in MapReduce capabilities, but it can be integrated with Hadoop in order to perform MapReduce operations across Cassandra data sets, or Spark for real-time data analysis. The DataStax Enterprise product provides integration with both of these tools out-of-the-box.
Comparing Cassandra to the alternatives

Now that you've got an in-depth understanding of the feature set that Cassandra offers, it's time to figure out which features are most important to you, and which database is the best fit. The following table lists a handful of commonly used databases, and key features that they do or don't have:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Cassandra</th>
<th>PostgreSQL</th>
<th>MongoDB</th>
<th>Redis</th>
<th>Riak</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structured records</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Limited</td>
<td>No</td>
</tr>
<tr>
<td>Secondary indexes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Discretely writable collections</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Relational joins</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Built-in MapReduce</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Fast result ordering</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Immediate consistency</td>
<td>Configurable at query level</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Configurable at cluster level</td>
</tr>
<tr>
<td>Transparent sharding</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>No single point of failure</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>High throughput writes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

As you can see, Cassandra offers a unique combination of scalability, availability, and a rich set of features for modeling and accessing data.
Installing Cassandra

Now that you’re acquainted with Cassandra’s substantial powers, you’re no doubt chomping at the bit to try it out. Happily, Cassandra is free, open source, and very easy to get running.

Installing on Mac OS X

First, we need to make sure that we have an up-to-date installation of the Java Runtime Environment. Open the Terminal application, and type the following into the command prompt:

$ java -version

You will see an output that looks something like the following:

java version "1.8.0_25"
Java(TM) SE Runtime Environment (build 1.8.0_25-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.25-b02, mixed mode)
Pay particular attention to the java version listed: if it's lower than 1.7.0_25, you'll need to install a new version. If you have an older version of Java or if Java isn't installed at all, head to https://www.java.com/en/download/mac_download.jsp and follow the download instructions on the page.

You'll need to set up your environment so that Cassandra knows where to find the latest version of Java. To do this, set up your JAVA_HOME environment variable to the install location, and your PATH to include the executable in your new Java installation as follows:

```
$ export JAVA_HOME="/Library/Internet Plug-Ins/JavaAppletPlugin.plugin/Contents/Home"
$ export PATH="$JAVA_HOME/bin":$PATH
```

**Downloading the example code**

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

You should put these two lines at the bottom of your .bashrc file to ensure that things still work when you open a new terminal.

The installation instructions given earlier assume that you're using the latest version of Mac OS X (at the time of writing this, 10.10 Yosemite). If you're running a different version of OS X, installing Java might require different steps. Check out https://www.java.com/en/download/faq/java_mac.xml for detailed installation information.

Once you've got the right version of Java, you're ready to install Cassandra. It's very easy to install Cassandra using Homebrew; simply type the following:

```
$ brew install cassandra
$ pip install cassandra-driver cql
$ cassandra
```

Here's what we just did:

- Installed Cassandra using the Homebrew package manager
- Installed the CQL shell and its dependency, the Python Cassandra driver
- Started the Cassandra server
Installing on Ubuntu

First, we need to make sure that we have an up-to-date installation of the Java Runtime Environment. Open the Terminal application, and type the following into the command prompt:

```
$ java -version
```

You will see an output that looks similar to the following:

```
java version "1.7.0_65"
OpenJDK Runtime Environment (IcedTea 2.5.3) (7u71-2.5.3-0ubuntu0.14.04.1)
OpenJDK 64-bit Server VM (build 24.65-b04, mixed mode)
```

Pay particular attention to the java version listed: it should start with 1.7. If you have an older version of Java, or if Java isn't installed at all, you can install the correct version using the following command:

```
$ sudo apt-get install openjdk-7-jre-headless
```

Once you've got the right version of Java, you're ready to install Cassandra. First, you need to add Apache's Debian repositories to your sources list. Add the following lines to the file `/etc/apt/sources.list`:

```
deb http://www.apache.org/dist/cassandra/debian 21x main
deb-src http://www.apache.org/dist/cassandra/debian 21x main
```

In the Terminal application, type the following into the command prompt:

```
$ gpg --keyserver pgp.mit.edu --recv-keys F758CE318D77295D
$ gpg --export --armor F758CE318D77295D | sudo apt-key add -
$ gpg --keyserver pgp.mit.edu --recv-keys 2B5C1B00
$ gpg --export --armor 2B5C1B00 | sudo apt-key add -
$ gpg --keyserver pgp.mit.edu --recv-keys 0353B12C
$ gpg --export --armor 0353B12C | sudo apt-key add -
$ sudo apt-get update
$ sudo apt-get install cassandra
$ cassandra
```
Here's what we just did:

- Added the Apache repositories for Cassandra 2.1 to our sources list
- Added the public keys for the Apache repo to our system and updated our repository cache
- Installed Cassandra
- Started the Cassandra server

**Installing on Windows**

The easiest way to install Cassandra on Windows is to use the DataStax Community Edition. DataStax is a company that provides enterprise-level support for Cassandra; they also release Cassandra packages at both free and paid tiers. DataStax Community Edition is free, and does not differ from the Apache package in any meaningful way.

DataStax offers a graphical installer for Cassandra on Windows, which is available for download at planetcassandra.org/cassandra. On this page, locate **Windows Server 2008/Windows 7 or Later (32-Bit)** from the Operating System menu (you might also want to look for 64-bit if you run a 64-bit version of Windows), and choose MSI Installer (2.x) from the version columns.

Download and run the MSI file, and follow the instructions, accepting the defaults:
Getting Up and Running with Cassandra

Once the installer completes this task, you should have an installation of Cassandra running on your machine.

**Bootstrapping the project**
Throughout the remainder of this book, we will build an application called MyStatus, which allows users to post status updates for their friends to read. In each chapter, we'll add new functionality to the MyStatus application; each new feature will also introduce a new aspect of Cassandra.

**CQL – the Cassandra Query Language**
Since this is a book about Cassandra and not targeted to users of any particular programming language or application framework, we will focus entirely on the database interactions that MyStatus will require. Code examples will be in Cassandra Query Language (CQL). Specifically, we'll use version 3.1.1 of CQL, which is available in Cassandra 2.0.6 and later versions.

As the name implies, CQL is heavily inspired by SQL; in fact, many CQL statements are equally valid SQL statements. However, CQL and SQL are not interchangeable. CQL lacks a grammar for relational features such as JOIN statements, which are not possible in Cassandra. Conversely, CQL is not a subset of SQL; constructs for retrieving the update time of a given column, or performing an update in a lightweight transaction, which are available in CQL, do not have an SQL equivalent.

Throughout this book, you'll learn the important constructs of CQL. Once you've completed reading this book, I recommend you to turn to the DataStax CQL documentation for additional reference. This documentation is available at [http://www.datastax.com/documentation/cql/3.1](http://www.datastax.com/documentation/cql/3.1).

**Interacting with Cassandra**
Most common programming languages have drivers for interacting with Cassandra. When selecting a driver, you should look for libraries that support the CQL binary protocol, which is the latest and most efficient way to communicate with Cassandra.

The CQL binary protocol is a relatively new introduction; older versions of Cassandra used the Thrift protocol as a transport layer. Although Cassandra continues to support Thrift, avoid Thrift-based drivers, as they are less performant than the binary protocol.
Here are CQL binary drivers available for some popular programming languages:

<table>
<thead>
<tr>
<th>Language</th>
<th>Driver</th>
<th>Available at</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>DataStax Java Driver</td>
<td>github.com/datastax/java-driver</td>
</tr>
<tr>
<td>Python</td>
<td>DataStax Python Driver</td>
<td>github.com/datastax/python-driver</td>
</tr>
<tr>
<td>Ruby</td>
<td>DataStax Ruby Driver</td>
<td>github.com/datastax/ruby-driver</td>
</tr>
<tr>
<td>C++</td>
<td>DataStax C++ Driver</td>
<td>github.com/datastax/cpp-driver</td>
</tr>
<tr>
<td>C#</td>
<td>DataStax C# Driver</td>
<td>github.com/datastax/csharp-driver</td>
</tr>
<tr>
<td>JavaScript (Node.js)</td>
<td>node-cassandra-cql</td>
<td>github.com/jorgebay/node-cassandra-cql</td>
</tr>
<tr>
<td>PHP</td>
<td>phpbinarycql</td>
<td>github.com/rmcfrazier/phpbinarycql</td>
</tr>
</tbody>
</table>

While you will likely use one of these drivers in your applications, to try out the code examples in this book, you can simply use the `cqlsh` tool, which is a command-line interface for executing CQL queries and viewing the results. To start cqlsh on OS X or Linux, simply type `cqlsh` into your command line; you should see something like this:

```
$ cqlsh
Connected to Test Cluster at localhost:9160.
[cqlsh 4.1.1 | Cassandra 2.0.7 | CQL spec 3.1.1 | Thrift protocol 19.39.0]
Use HELP for help.
cqlsh>
```

On Windows, you can start cqlsh by finding the Cassandra CQL Shell application in the DataStax Community Edition group in your applications. Once you open it, you should see the same output we just saw.

**Creating a keyspace**

A **keyspace** is a collection of related tables, equivalent to a database in a relational system. To create the keyspace for our MyStatus application, issue the following statement in the CQL shell:

```cql
CREATE KEYSPACE "my_status"
WITH REPLICATION = {
  'class': 'SimpleStrategy', 'replication_factor': 1
};
```
Getting Up and Running with Cassandra

Here we created a keyspace called my_status, which we will use for the remainder of this book. When we create a keyspace, we have to specify replication options. Cassandra provides several strategies for managing replication of data; SimpleStrategy is the best strategy as long as your Cassandra deployment does not span multiple data centers. The replication_factor value tells Cassandra how many copies of each piece of data are to be kept in the cluster; since we are only running a single instance of Cassandra, there is no point in keeping more than one copy of the data. In a production deployment, you would certainly want a higher replication factor; 3 is a good place to start.

A few things at this point are worth noting about CQL's syntax:

- It's syntactically very similar to SQL; as we further explore CQL, the impression of similarity will not diminish.
- Double quotes are used for identifiers such as keyspace, table, and column names. As in SQL, quoting identifier names is usually optional, unless the identifier is a keyword or contains a space or another character that will trip up the parser.
- Single quotes are used for string literals; the key-value structure we use for replication is a map literal, which is syntactically similar to an object literal in JSON.
- As in SQL, CQL statements in the CQL shell must terminate with a semicolon.

Selecting a keyspace

Once you've created a keyspace, you would want to use it. In order to do this, employ the use command:

USE "my_status";

This tells Cassandra that all future commands will implicitly refer to tables inside the my_status keyspace. If you close the CQL shell and reopen it, you'll need to reissue this command.
Summary

In this chapter, you explored the reasons to choose Cassandra from among the many databases available, and having determined that Cassandra is a great choice, you installed it on your development machine.

You had your first taste of the Cassandra Query Language when you issued your first command via the CQL shell in order to create a keyspace. You're now poised to begin working with Cassandra in earnest.

In the next chapter, we'll begin building the MyStatus application, starting out with a simple table to model users. We'll cover a lot more CQL commands, and before you know it, you'll be reading and writing data like a pro.
Where to buy this book

You can buy Learning Apache Cassandra from the Packt Publishing website. Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.