CHAPTER 8
Clients

We're used to connecting to relational databases using drivers. For example, in Java, JDBC is an API that abstracts the vendor implementation of the relational database to present a consistent way of storing and retrieving data using `Statements`, `PreparedStatements`, `ResultSets`, and so forth. To interact with the database, you get a driver that works with the particular database you're using, such as Oracle, SQL Server, or MySQL; the implementation details of this interaction are hidden from the developer. Drivers are typically provided for a wide variety of programming languages to connect to a wide variety of databases.

There are a number of client drivers available for Cassandra as well, including support for most popular languages. There are benefits to these clients, in that you can easily embed them in your own applications (which we'll see how to do) and that they frequently offer more features than the CQL native interface does, including connection pooling and JMX integration and monitoring. In the following sections, we'll learn about the various clients available and the features they offer.

Hector, Astyanax, and Other Legacy Clients

In the early days of Cassandra, the community produced a number of client drivers for different languages. These contributions were a key enabler of Cassandra adoption. Some of the most notable early drivers included Hector and Astyanax.

Named after Cassandra's brother, a warrior of Troy in Greek mythology, Hector was one of the first Cassandra clients. Hector provided a simple Java interface that helped many early developers avoid the challenges of writing to the Thrift API, and served as the inspiration for several other drivers. The project is no longer active, but you can access it at *https://github.com/hector-client/hector*.

Astyanax was a Java client originally built by Netflix on top of the Thrift API as a logical successor to the Hector driver (Astyanax was Hector's son). Once the DataStax Java driver was introduced, Netflix adapted Astyanax to support the Java driver in addition to the original Thrift implementation. This helped many users transition from Thrift to CQL. However, as the Java driver gained prominence, activity on Astyanax slowed considerably, and the project was retired in February, 2016. You can still access the project at *https://github.com/Netflix/astyanax*.

Other clients included Pycassa for Python, Perlcassa for Perl, Helenus for Node.js, and Cassandra-Sharp for the Microsoft .NET framework and C#. Most of these clients are no longer actively maintained, as they were based on the now-deprecated Thrift interface. You can find a comprehensive list of both current and legacy drivers at *http://www.planetcassandra.org/client-drivers-tools*.

DataStax Java Driver

The introduction of CQL was the impetus for a major shift in the landscape of Cassandra client drivers. The simplicity and familiar syntax of CQL made the development of client programs similar to traditional relational database drivers. DataStax made a strategic investment of open source drivers for Java and several additional languages in order to continue to fuel Cassandra adoption. These drivers quickly became the de facto standard for new development projects. You can access the drivers as well as additional connectors and tools at *https://github.com/datastax*.

More Information on DataStax Drivers

Visit the driver matrix page to access documentation and identify driver versions that are compatible with your server version.

The DataStax Java driver is the oldest and most mature of these drivers. For this reason, we'll focus on using the Java driver and use this as an opportunity to learn about the features that are provided by the DataStax drivers across multiple languages.

Development Environment Configuration

First, we'll need to access the driver in our development environment. We could download the driver directly from the URL listed before and manage the dependencies manually, but it is more typical in modern Java development to use a tool like Maven to manage dependencies. If you're using Maven, you'll need to add something like the following to your project *pom.xml* file:

```
<dependency>
  <groupId>com.datastax.cassandra</groupId>
  <artifactId>cassandra-driver-core</artifactId>
```

```
    <version>3.0.0</version>
</dependency>
```

You can find the Javadoc for the Java driver at *http://docs.datastax.com/en/drivers/java/3.0/index.html*. Alternatively, the Javadocs are also part of the source distribution.

All of the DataStax drivers are managed as open source projects on GitHub. If you're interested in seeing the Java driver source, you can get a read-only trunk version using this command:

```
$ git clone https://github.com/datastax/java-driver.git
```

Clusters and Contact Points

Once we've configured our environment, we're ready to start coding. We'll create a client application based on the hotel data model we created in Chapter 5. All of the source code used in this chapter and throughout the rest of the book is available at *https://github.com/jeffreyscarpenter/cassandra-guide*.

To start building our application, we'll use the driver's API to connect to our cluster. In the Java driver, this is represented by the `com.datastax.driver.core.Cluster` and `Session` classes.

The `Cluster` class is the main entry point of the driver. It supports a fluent-style API using the builder pattern. For example, the following lines create a connection to a node running on the local host:

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

This one statement represents the minimum required information to create a cluster: a single contact point. We can also specify multiple contact points. Contact points are similar to the concept seed nodes that a Cassandra node uses to connect to other nodes in the same cluster.

Creating a Custom Cluster Initializer

The `Cluster.Builder` class implements an interface called `Cluster.Initializer`. This allows us to plug in a different mechanism to initialize a `Cluster` using the static method `Cluster.buildFrom(Initializer initializer)`. This could be useful if we wanted to load the connection information from a configuration file, for example.

There are several other options that we can configure on a `Cluster`, such as a metrics, default query options, and policies for reconnection, retry, and speculative execution.

We'll examine each of these options in later sections after we take a look at some other connection-related options: protocol version, compression, and authentication.

Protocol version

The driver supports multiple versions of the CQL native protocol. Cassandra 3.0 supports version 4, as we learned in our overview of Cassandra's release history in Chapter 2.

By default, the driver uses the protocol version supported by the first node it connects to. While in most cases this is sufficient, you may need to override this behavior if you're working with a cluster based on an older version of Cassandra. You can select your protocol version by passing the desired value from the `com.datastax.driver.core.ProtocolVersion` enumeration to the `Cluster.Builder.withProtocolVersion()` operation.

Compression

The driver provides the option of compressing messages between your client and Cassandra nodes, taking advantage of the compression options supported by the CQL native protocol. Enabling compression reduces network bandwidth consumed by the driver, at the cost of additional CPU usage for the client and server.

Currently there are two compression algorithms available, `LZ4` and `SNAPPY`, as defined by the `com.datastax.driver.core.ProtocolOptions.Compression` enumeration. The compression defaults to `NONE` but can be overridden by calling the `Cluster.Builder.withCompression()` operation.

Authentication and encryption

The driver provides a pluggable authentication mechanism that can be used to support a simple username/password login, or integration with other authentication systems. By default, no authentication is performed. You can select an authentication provider by passing an implementation of the `com.datastax.driver.core.AuthProvider` interface such as the `PlainTextAuthProvider` to the `Cluster.Builder.withAuthProvider()` operation.

The driver can also encrypt its communications with the server to ensure privacy. Client-server encryption options are specified by each node in its *cassandra.yaml* file. The driver complies with the encryption settings specified by each node.

We'll examine authentication, authorization, and encryption from both the client and server perspective in more detail in Chapter 13.

Sessions and Connection Pooling

After we create our `Cluster` instance, it is not connected to any Cassandra nodes until we initialize it by calling the `init()` method:

```
cluster.init();
```

When this method is invoked, the driver connects to one of the configured contact points in order to obtain metadata about the cluster. This operation will throw a `NoHostAvailableException` if none of the contact points is available, or an `AuthenticationException` if authentication fails. We'll discuss authentication in more detail in Chapter 13.

Once we have initialized our `Cluster` object, we need to establish a session in order to formulate our queries. We can obtain a `com.datastax.driver.core.Session` object by calling one of the `Cluster.connect()` operations. You can optionally provide the name of a keyspace to connect to, as we do in this example that connects to the hotel keyspace:

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

There is also a `connect()` operation with no parameters, which creates a `Session` that can be used with multiple keyspaces. If you choose this option, you'll have to qualify every table reference in your queries with the appropriate keyspace name. Note that it is not strictly required to call `Cluster.init()` explicitly, as it is also invoked behind the scenes when we call `connect()`.

Each `Session` manages connections to a Cassandra cluster, which are used to execute queries and control operations using the Cassandra native protocol. The session contains a pool of TCP connections for each host.

Sessions Are Expensive

Because a session maintains connection pools for multiple nodes, it is a relatively heavyweight object. In most cases, you'll want to create a single `Session` and reuse it throughout your application, rather than continually building up and tearing down `Sessions`. Another acceptable option is to create a `Session` per keyspace, if your application is accessing multiple keyspaces.

Because the CQL native protocol is asynchronous, it allows multiple simultaneous requests per connection; the maximum is 128 simultaneous requests in protocol v2, while v3 and v4 allow up to 32,768 simultaneous requests. Because of this larger number of simultaneous requests, fewer connections per node are required. In fact, the default is a single connection per node.

The driver supports the ability to scale the number of connections up or down based on the number of requests per connection. These connection pool settings are configurable via the `PoolingOptions` class, which sets the maximum and minimum (or "core") number of connections to use for local and remote hosts. If the core and maximum values are different, the driver scales the size of the connection pool for each node up or down depending on the amount of requests made by the client. The settings of minimum and maximum thresholds of requests per connection are used to determine when new connections are created, and when underused connections can be reclaimed. There is also a buffer period to prevent the continual building up and tearing down of connections.

The `PoolingOptions` can be set when creating the `Cluster` using `ClusterBuilder.withPoolingOptions()`, or manipulated after the `Cluster` is created using `Cluster.getConfiguration().getPoolingOptions()`. Here is an example of creating a `Cluster` that limits the maximum number of connections to remote nodes to one:

```
PoolingOptions poolingOptions = new PoolingOptions().
  setMaxConnectionsPerHost(HostDistance.REMOTE, 1);

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

The driver provides a connection heartbeat which is used to make sure that connections are not closed prematurely by intervening network devices. This defaults to 30 seconds but can be overridden using the operation `PoolingOptions.setHeartbeatIntervalSeconds()`. However, this only applies to connections established after the value is set, so you'll most likely want to configure this when creating your `Cluster`.

Statements

Up until this point, we have only configured our connection to the cluster, and haven't yet performed any reads or writes. To begin doing some real application work, we'll create and execute statements using the `com.datastax.driver.core.Statement` class and its various subclasses. `Statement` is an abstract class with several implementations, including `SimpleStatement`, `PreparedStatement`, `BoundStatement`, `BatchStatement`, and `BuiltStatement`.

The simplest way to create and execute a statement is to call the `Session.execute()` operation with a string representing the statement. Here's an example of a statement that will return the entire contents of our hotels table:

```
session.execute("SELECT * from hotel.hotels");
```

Why using session.execute() is not a good practice?

This statement creates and executes a query in a single method call. In practice, this could turn out to be a very expensive query to execute in a large database, but it does serve as a useful example of a very simple query. Most queries we need to build will be more complex, as we'll have search criteria to specify or specific values to insert. We can certainly use Java's various string utilities to build up the syntax of our query by hand, but this of course is error prone. It may even expose our application to injection attacks, if we're not careful to sanitize strings that come from end users.

Simple statement

Thankfully, we needn't make things so hard on ourselves. The Java driver provides the `SimpleStatement` class to help construct parameterized statements. As it turns out, the `execute()` operation we saw before is actually a convenience method for creating a `SimpleStatement`.

Let's try building a query by asking our Session object to create a `SimpleStatement`. Here's an example of a statement that will insert a row in our `hotels` table, which we can then execute:

```
SimpleStatement hotelInsert = session.newSimpleStatement(
  "INSERT INTO hotels (hotel_id, name, phone) VALUES (?, ?, ?)",
  "AZ123", "Super Hotel at WestWorld", "1-888-999-9999");
session.execute(hotelInsert);
```

The first parameter to the call is the basic syntax of our query, indicating the table and columns we are interested in. The question marks are used to indicate values that we'll be providing in additional parameters. We use simple strings to hold the values of the hotel ID, name, and phone number.

If we've created our statement correctly, the insert will execute successfully (and silently). Now let's create another statement to read back the row we just inserted:

```
SimpleStatement hotelSelect = session.newSimpleStatement(
  "SELECT * FROM hotels WHERE id=?", "AZ123");
ResultSet hotelSelectResult = session.execute(hotelSelect);
```

Again, we make use of parameterization to provide the ID for our search. This time, when we execute the query, we make sure to receive the `ResultSet` which is returned from the `execute()` method. We can iterate through the rows returned by the `ResultSet` as follows:

```
for (Row row : hotelSelectResult) {
  Systen.out.format("hotel_id: %s, name: %s, phone: %s\n",
  row.getString("hotel_id"), row.getString("name"), row.getString("phone"));
}
```

This code uses the `ResultSet.iterator()` option to get an `Iterator` over the rows in the result set and loop over each row, printing out the desired column values. Note that we use special accessors to obtain the value of each column depending on the

desired type—in this case, `Row.getString()`. As we might expect, this will print out a result such as:

```
hotel_id: AZ123, name: Super Hotel at WestWorld, phone: 1-888-999-9999
```

Using a Custom Codec

As we already noted, we need to know the type of the columns we are requesting when interacting with the `Rows` in our `ResultSets`. If we were to request the `id` column using `Row.getString()`, we would receive a `CodecNotFoundException`, indicating that the driver does not know how to map the CQL type `uuid` to `java.lang.String`.

What is happening here is that the driver maintains a default list of mappings between Java and CQL types called a *codec*, which it uses to translate back and forth between your application and Cassandra. The driver provides a way to add additional mappings by extending the class `com.datastax.driver.core.TypeCodec<T>` and registering it with the `CodecRegistry` managed by the `Cluster`:

```
cluster.getConfiguration().getCodecRegistry().
  register(myCustomCodec)
```

The custom codec mechanism is very flexible, as demonstrated by the following use cases:

- Mapping to alternate date/time formats (e.g., Joda time for pre-Java 8 users)
- Mapping string data to/from formats such as XML and JSON
- Mapping lists, sets, and maps to various Java collection types

You can find code samples for working with `SimpleStatements` in the example `com.cassandraguide.clients.SimpleStatementExample`.

Asynchronous execution

The `Session.execute()` operation is synchronous, which means that it blocks until a result is obtained or an error occurs, such as a network timeout. The driver also provides the asynchronous `executeAsync()` operation to support non-blocking interactions with Cassandra. These non-blocking requests can make it simpler to send multiple queries in parallel to speed performance of your client application.

Let's take our operation from before and modify it to use the asynchronous operation:

```
ResultSetFuture result =  session.executeAsync(statement);
```

The result is of the type `ResultSetFuture`, which is an implementation of the `java.util.concurrent.Future` interface. A `Future` is a Java generic type used to capture the result of an asynchronous operation. Each `Future` can be checked to see

whether the operation has completed, and then queried for the result of the operation according to the bound type. There are also blocking `wait()` operations to wait for the result. A `Future` can also be cancelled if the caller is no longer interested in the result of the operation. The `Future` class is a useful tool for implementing asynchronous programming patterns, but requires either blocking or polling to wait for the operation to complete.

To address this drawback, the Java driver leverages the `ListenableFuture` interface from Google's Guava framework. The `ListenableFuture` interface extends `Future`, and adds an `addListener()` operation that allows the client to register a callback method that is invoked when the `Future` completes. The callback method is invoked in a thread managed by the driver, so it is important that the method complete quickly to avoid tying up driver resources. The `ResultSetFuture` is bound to the `ResultSet` type.

Additional Asynchronous Operations

In addition to the `Session.executeAsync()` operation, the driver supports several other asynchronous operations, including `Cluster.closeAsync()`, `Session.prepareAsync()`, and several operations on the object mapper.

Prepared statement ⟶ This provide protection against injection attack

While `SimpleStatements` are quite useful for creating ad hoc queries, most applications tend to perform the same set of queries repeatedly. The `PreparedStatement` is designed to handle these queries more efficiently. The structure of the statement is sent to nodes a single time for preparation, and a handle for the statement is returned. To use the prepared statement, only the handle and the parameters need to be sent.

As you're building your application, you'll typically create `PreparedStatements` for reading data, corresponding to each access pattern you derive in your data model, plus others for writing data to your tables to support those access patterns.

Let's create some `PreparedStatements` to represent the same hotel queries as before, using the `Session.prepare()` operation:

```
PreparedStatement hotelInsertPrepared = session.prepare(
  "INSERT INTO hotels (hotel_id, name, phone) VALUES (?, ?, ?)");

PreparedStatement hotelSelectPrepared = session.prepare(
  "SELECT * FROM hotels WHERE hotel_id=?");
```

Note that the `PreparedStatement` uses the same parameterized syntax we used earlier for our `SimpleStatement`. A key difference, however, is that a `PreparedStatement` is

not a subtype of `Statement`. This prevents the error of trying to pass an unbound `PreparedStatement` to the session to execute.

Before we get to that, however, let's take a step back and discuss what is happening behind the scenes of the `Session.prepare()` operation. The driver passes the contents of our `PreparedStatement` to a Cassandra node and gets back a unique identifier for the statement. This unique identifier is referenced when you create a `BoundStatement`. If you're curious, you can actually see this reference by calling `PreparedStatement.getPreparedID()`.

You can think of a `PreparedStatement` as a template for creating queries. In addition to specifying the form of our query, there are other attributes that we can set on a `PreparedStatement` that will be used as defaults for statements it is used to create, including a default consistency level, retry policy, and tracing.

In addition to improving efficiency, `PreparedStatements` also improve security by separating the query logic of CQL from the data. This provides protection against injection attacks, which attempt to embed commands into data fields in order to gain unauthorized access.

Bound statement

Now our `PreparedStatement` is available for us to use to create queries. In order to make use of a `PreparedStatement`, we bind it with actual values by calling the `bind()` operation. For example, we can bind the `SELECT` statement we created earlier as follows:

```
BoundStatement hotelSelectBound = hotelSelectPrepared.bind("AZ123");
```

The `bind()` operation we've used here allows us to provide values that match each variable in the `PreparedStatement`. It is possible to provide the first *n* bound values, in which case the remaining values must be bound separately before executing the statement. There is also a version of `bind()` which takes no parameters, in which case all of the parameters must be bound separately. There are several `set()` operations provided by `BoundStatement` that can be used to bind values of different types. For example, we can take our `INSERT` prepared statement from above and bind the name and phone values using the `setString()` operation:

```
BoundStatement hotelInsertBound = hotelInsertPrepared.bind("AZ123");
hotelInsertBound.setString("name", "Super Hotel at WestWorld");
hotelInsertBound.setString("phone", "1-888-999-9999");
```

Once we have bound all of the values, we execute a `BoundStatement` using `Session.execute()`. If we have failed to bind any of the values, they will be ignored on the server side, if protocol v4 (Cassandra 3.0 or later) is in use. The driver behav-

ior for older protocol versions is to throw an `IllegalStateException` if there are any unbound values.

You can find code samples for working with `PreparedStatement` and `BoundStatement` in the example `com.cassandraguide.clients.PreparedStatementExample`.

Built statement and the Query Builder

The driver also provides the `com.datastax.driver.core.querybuilder.QueryBuilder` class, which provides a fluent-style API for building queries. This is suitable for cases where there is variation in the query structure (such as optional parameters) that would make using `PreparedStatements` difficult. Similar to `PreparedStatement`, it also provides some protection against injection attacks.

We construct a `QueryBuilder` using a simple constructor that takes our `Cluster` object:

```
QueryBuilder queryBuilder = new QueryBuilder(cluster);
```

The `QueryBuilder` produces queries that are represented using the `BuiltStatement` class and its subclasses. The methods on each class return instances of `BuiltStatement` that represent content added to a query as it is being built up. You'll likely find your IDE quite useful in helping to identify the allowed operations as you're building queries.

Let's reproduce the queries from before using the `QueryBuilder` to see how it works. First, we'll build a CQL `INSERT` query:

```
BuiltStatement hotelInsertBuilt =
  queryBuilder.insertInto("hotels")
  .value("hotel_id", "AZ123")
  .value("name", "Super Hotel at WestWorld")
  .value("phone", "1-888-999-9999");
```

The first operation calls the `QueryBuilder.insertInto()` operation to create an `Insert` statement for the hotels table. If desired, we could then add a CQL `USING` clause to our statement with `Insert.using()`, but instead we choose to start adding values to our query. The `Insert.value()` operation continues returning `Insert` statements as we add values. The resulting `Insert` can be executed like any other `Statement` using `Session.execute()` or `executeAsync()`.

The construction of the CQL `SELECT` command is similar:

```
BuiltStatement hotelSelectBuilt = queryBuilder.select()
  .all()
  .from("hotels")
  .where(eq("hotel_id", "AZ123"));
```

For this query, we call `QueryBuilder.select()` to create a `Select` statement. We use the `Select.all()` operation to select all columns, although we could also have used the `column()` operation to select specific columns. We add a CQL `WHERE` clause via the `Select.where()` operation, which accepts an instance of the `Clause` class. We create `Clauses` using static operations provided by the `QueryBuilder`. In this case, we use the `eq()` operation to check for equality with our ID.

To access these static operations, we need to add additional import statements to our Java source files such as:

```
import static com.datastax.driver.core.querybuilder.QueryBuilder.eq;
```

For a complete code sample using the `QueryBuilder` and `BuiltStatement`, see the class `com.cassandraguide.clients.QueryBuilderExample`.

Object mapper

We've explored several techniques for creating and executing query statements with the driver. There is one final technique that we'll look at that provides a bit more abstraction. The Java driver provides an object mapper that allows you to focus on developing and interacting with domain models (or data types used on APIs). The object mapper works off of annotations in source code that are used to map Java classes to tables or user-defined types (UDTs).

The object mapping API is provided as a separate library from the rest of the driver in the *cassandra-driver-mapping.jar* file, so you will need to include this additional Maven dependency in order to use Mapper in your project:

What is the benefit of using an ObjectMapper?

```
<dependency>
   <groupId>com.datastax.cassandra</groupId>
   <artifactId>cassandra-driver-mapping</artifactId>
   <version>3.0.0</version>
</dependency>
```

For example, let's create and annotate a `Hotel` domain model class corresponding to our `hotels` table:

```
import com.datastax.driver.mapping.annotations.Column;
import com.datastax.driver.mapping.annotations.PartitionKey;
import com.datastax.driver.mapping.annotations.Table;

@Table(keyspace = "hotel", name = "hotels")
public class Hotel {

    @PartitionKey
    private String id;

    @Column (name = "name")
    private String name;
```

```
    @Column (name = "phone")
    private String phone;

    @Column (name = "address")
    private String address;

    @Column (name = "pois")
    private Set<String> pointsOfInterest;

    // constructors, get/set methods, hashcode, equals
}
```

Now we use the `com.datastax.driver.mapping.MappingManager` to attach to our `Session` and create a `Mapper` for our annotated domain model class:

```
MappingManager mappingManager = new MappingManager(session);
Mapper<Hotel> hotelMapper = MappingManager.mapper(Hotel.class);
```

Let's assume the `Hotel` class has a simple constructor that just takes a UUID, name, and phone number, which we'll use to create a simple hotel that we can save using the object mapper:

```
Hotel hotel = new Hotel("AZ123", "Super Hotel at WestWorld",
  "1-888-999-9999");
hotelMapper.save(hotel);
```

The `Mapper.save()` operation is all we need to execute to perform a CQL `INSERT` or `UPDATE`, as these are really the same operation to Cassandra. The `Mapper` builds and executes the statement on our behalf.

To retrieve an object, we use the `Mapper.get()` operation, passing in an argument list that matches the elements of the partition key:

```
Hotel retrievedHotel = hotelMapper.get(hotelId);
```

The syntax for deleting an object is similar:

```
hotelMapper.delete(hotelId);
```

As with the `save()` operation, `get()` and `delete()` completely handle the details of executing statements with the driver on our behalf. There are also `saveAsync()`, `getAsync()` and `deleteAsync()` operations that support asynchronous execution using the `ListenableFuture` interface we discussed earlier.

If you want to be able to configure the queries before they are executed, there are also operations on the `Mapper` that return `Statements`: `saveQuery()`, `getQuery()`, and `deleteQuery()`.

The object mapper is a useful tool for abstracting some of the details of interacting with your code, especially if you have an existing domain model. If your domain model contains classes that reference other classes, you can annotate the referenced

classes as user-defined types with the `@UDT` annotation. The object mapper processes objects recursively using your annotated types.

Achilles: An Advanced Object Mapper

DuyHai Doan has developed an advanced object mapper for Java called Achilles. Achilles provides support for more advanced functionality such as complex key mappings, lightweight transactions, user defined functions, and more. You can check it out at *https://github.com/doanduyhai/Achilles*.

Policies

The Java driver provides several policy interfaces that can be used to tune the behavior of the driver. These include policies for load balancing, retrying requests, and managing connections to nodes in the cluster.

Load balancing policy

As we learned in Chapter 6, a query can be made to any node in a cluster, which is then known as the coordinator node for that query. Depending on the contents of the query, the coordinator may communicate with other nodes in order to satisfy the query. If a client were to direct all of its queries at the same node, this would produce an unbalanced load on the cluster, especially if other clients are doing the same.

To get around this issue, the driver provides a pluggable mechanism to balance the query load across multiple nodes. Load balancing is implemented by selecting an implementation of the `com.datastax.driver.core.policies.LoadBalancingPolicy` interface.

Each `LoadBalancingPolicy` must provide a `distance()` operation to classify each node in the cluster as local, remote, or ignored, according to the `HostDistance` enumeration. The driver prefers interactions with local nodes and maintains more connections to local nodes than remote nodes. The other key operation is `newQueryPlan()`, which returns a list of nodes in the order they should be queried. The `LoadBalancingPolicy` interface also contains operations that are used to inform the policy when nodes are added or removed, or go up or down. These operations help the policy avoid including down or removed nodes in query plans.

The driver provides two basic load balancing implementations: the `RoundRobinPolicy`, which is the default, and the `DCAwareRoundRobinPolicy`.

The `RoundRobinPolicy` allocates requests across the nodes in the cluster in a repeating pattern to spread the processing load. The `DCAwareRoundRobinPolicy` is similar, but focuses its query plans on nodes in the local data center. This policy can add a

configurable number of nodes in remote data centers to query plans, but the remote nodes will always come after local nodes in priority. The local data center can be identified explicitly, or you can allow the driver to discover it automatically.

A second mode is token awareness, which uses the token value of the partition key in order to select a node which is a replica for the desired data, thus minimizing the number of nodes that must be queried. This is implemented by wrapping the selected policy with a `TokenAwarePolicy`.

The `LoadBalancingPolicy` is set on the `Cluster` when it is built. For example, the following statement will initialize a `Cluster` to have token awareness and to prefer nodes in the local data center:

```
Cluster.builder().withLoadBalancingPolicy(
  new TokenAwarePolicy(new DCAwareRoundRobinPolicy.Builder().build());
```

Retry policy

When Cassandra nodes fail or become unreachable, the driver automatically and transparently tries other nodes and schedules reconnection to the dead nodes in the background. Because temporary changes in network conditions can also make nodes appear offline, the driver also provides a mechanism to retry queries that fail due to network-related errors. This removes the need to write retry logic in client code.

The driver retries failed queries according to the provided implementation of the `com.datastax.driver.core.RetryPolicy` interface. The `onReadTimeout()`, `onWriteTimeout()`, and `onUnavailable()` operations define the behavior that should be taken when a query fails with the network-related exceptions `ReadTimeoutException`, `WriteTimeoutException`, or `UnavailableException`, respectively.

DataStax Java Driver Exceptions

The various exceptions and errors that can be generated by the Java driver are collected in the `com.datastax.driver.core.exceptions` package.

The `RetryPolicy` operations return a `RetryDecision`, which indicates whether the query should be retried, and if so, at what consistency level. If the exception is not retried, it can be rethrown, or ignored, in which case the query operation will return an empty `ResultSet`.

The Java driver provides several `RetryPolicy` implementations:

- The `DefaultRetryPolicy` is a conservative implementation that only retries queries under a narrow set of conditions.

- The `FallthroughRetryPolicy` never recommends a retry, always recommending that the exception be rethrown.
- The `DowngradingConsistencyRetryPolicy` is a more aggressive policy which downgrades the consistency level required, as an attempt to get the query to succeed.

A Word on DowngradingConsistencyRetryPolicy

This policy comes with a warning attached: if you are willing to accept a downgraded consistency level under some circumstances, do you really require a higher consistency level for the general case?

The `RetryPolicy` can be set on a `Cluster` when it is built, as shown by the following statement, which selects the `DowngradingConsistencyRetryPolicy` and wraps it with a `LoggingRetryPolicy` so that each retry attempt will be logged:

```
Cluster.builder().withRetryPolicy(new LoggingRetryPolicy(
  DowngradingConsistencyRetryPolicy.INSTANCE));
```

The `RetryPolicy` on a cluster will be used for all queries executed on that cluster, unless overridden on any individual query via the `Statement.setRetryPolicy()` operation.

Speculative execution policy

While it's great to have a retry mechanism that automates our response to network timeouts, we don't often have the luxury of being able to wait for timeouts or even long garbage collection pauses. To speed things up, the driver provides a speculative execution mechanism. If the original coordinator node for a query fails to respond in a predetermined interval, the driver preemptively starts an additional execution of the query against a different coordinator node. When one of the queries returns, the driver provides that response and cancels any other outstanding queries.

The speculative execution behavior is set on a `Cluster` by specifying an implementation of `com.datastax.driver.core.policies.SpeculativeExecutionPolicy`.

The default is the `NoSpeculativeExecutionPolicy`, which does not schedule any speculative executions. There is also a `ConstantSpeculativeExecutionPolicy`, which schedules up to a maximum number of retries with a fixed delay in milliseconds. The `PercentileSpeculativeExecutionPolicy` is a newer policy that is still considered a Beta as of the 3.0 driver release. It triggers speculative executions at a delay based on the observed latency to the original coordinator node.

The policy is set using the `Cluster.Builder`, for example:

```
Cluster.builder().withSpeculativeExecutionPolicy(
  new ConstantSpeculativeExecutionPolicy (
    200, // delay in ms
    3    // max number of speculative executions
  );
```

This policy cannot be changed later, or overridden on individual `Statements`.

Address translator

In the examples we've seen so far, each node is identified by the IP address configured as the node's `rpc_address` in its *cassandra.yaml* file. In some deployments, that address may not be reachable by the client. To handle this case, the driver provides a pluggable capability to translate addresses via the `com.datastax.driver.core.policies.AddressTranslator` interface (in versions of the driver prior to 3.0, "translator" is misspelled as "translater" throughout the API).

For example, the Java driver comes with the `IdentityTranslator`, a default translator that leaves the IP address unchanged, and the `EC2MultiRegionAddressTranslator`, which is useful for Amazon EC2 environments. This translator is useful in cases where a client may need to access a node in another data center via a public IP address. We'll discuss EC2 deployments in more detail in Chapter 14.

Metadata

To access the cluster metadata, we invoke the `Cluster.getMetadata()` method. The `com.datastax.driver.core.Metadata` class provides information about the cluster including the cluster name, the schema including keyspaces and tables, and the known hosts in the cluster. We can obtain the name of the cluster via the following code:

```
Metadata metadata = cluster.getMetadata();
System.out.printf("Connected to cluster: %s\n",
  metadata.getClusterName(), cluster.getClusterName());
```

Assigning a Cluster Name

Somewhat confusingly, the `Cluster.Builder` class allows us to assign a name to the `Cluster` instance as it is being built. This name is really just a way for the client to keep track of multiple `Cluster` objects, and can be different than the name known by the nodes within the actual Cassandra cluster. This second cluster name is the one we obtain via the `Metadata` class.

If we do not specify a name for the `Cluster` on construction, it is assigned a default name such as "cluster1", "cluster2", and so on (if multiple clusters are created). You can see this value if you modify the example from before to change `metadata.getClusterName()` to `cluster.getClusterName()`.

Node discovery

A `Cluster` object maintains a permanent connection to one of the contact points, which it uses to maintain information on the state and topology of the cluster. Using this connection, the driver will discover all the nodes currently in the cluster. The driver uses the `com.datastax.driver.core.Host` class to represent each node. The following code shows an example of iterating over the hosts to print out their information:

```
for (Host host : cluster.getMetadata.getAllHosts())
{
  System.out.printf("Data Center: %s; Rack: %s; Host: %s\n",
  host.getDatacenter(), host.getRack(), host.getAddress());
}
```

You can find this code in the class `com.cassandraguide.clients.SimpleConnectionExample`.

If we're running a multi-node cluster such as the one we created in Chapter 7 using the Cassandra Cluster Manager (`ccm`), the output of this program will look something like the following:

```
Connected to cluster: my_cluster
Data Center: datacenter1; Rack: rack1; Host: /127.0.0.1
Data Center: datacenter1; Rack: rack1; Host: /127.0.0.2
Data Center: datacenter1; Rack: rack1; Host: /127.0.0.3
```

Using the connection, the driver can also discover all the nodes currently in the cluster. The driver also can detect when new nodes are added to a cluster. You can register a listener to do this by implementing the `Host.StateListener` interface. This requires us to implement several operations such as `onAdd()` and `onRemove()`, which are called when nodes are added or removed from the cluster, as well as `onUp()` and `onDown()`, which indicate when nodes go up or down. Let's look at a portion of a sample class that registers a listener with the cluster:

```
public class ConnectionListenerExample implements Host.StateListener {

  public String getHostString(Host host) {
    return new StringBuilder("Data Center: " + host.getDatacenter() +
      " Rack: " + host.getRack() +
      " Host: " + host.getAddress().toString() +
      " Version: " + host.getCassandraVersion() +
      " State: " + host.getState());
  }

  public void onUp(Host host) {
    System.out.printf("Node is up: %s\n", getHostString(host));
  }

  public void onDown(Host host) {
    System.out.printf("Node is down: %s\n", getHostString(host));
  }

  // other required methods omitted...
  public static void main(String[] args) {

    List<Host.StateListener> list =
      ArrayList<Host.StateListener>();
    list.add(new ConnectionListenerExample());

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

    cluster.init();
  }
}
```

This code simply prints out a status message when a node goes up or down. You'll note that we make use of a bit more information about each node than our previous example, including the Cassandra version in use by each of the nodes. You can find the full code listing in the class `com.cassandraguide.clients.ConnectionListenerExample`.

Let's run this sample program. Because our listener was added before calling `init()`, we immediately get the following output:

```
Node added: Data Center: datacenter1 Rack: rack1
  Host: /127.0.0.1 Version: 3.0.0 State: UP
Node added: Data Center: datacenter1 Rack: rack1
  Host: /127.0.0.2 Version: 3.0.0 State: UP
Node added: Data Center: datacenter1 Rack: rack1
  Host: /127.0.0.3 Version: 3.0.0 State: UP
```

Now let's use the `ccm stop` command to shut down one of our nodes, and we'll see something like the following:

```
Node is down: Data Center: datacenter1 Rack: rack1
  Host: /127.0.0.1 Version: 3.0.0 State: DOWN
```

Similarly, if we bring the node back up, we'll see a notification that the node is back online:

```
Node is up: Data Center: datacenter1 Rack: rack1
  Host: /127.0.0.1 Version: 3.0.0 State: UP
```

Schema access

The `Metadata` class also allows the client to learn about the schema in a cluster. The `exportSchemaAsString()` operation creates a `String` describing all of the keyspaces and tables defined in the cluster, including the system keyspaces. This output is equivalent to the `cqlsh` command `DESCRIBE FULL SCHEMA`. Additional operations support browsing the contents of individual keyspaces and tables.

We've previously discussed Cassandra's support for eventual consistency at great length in Chapter 2. Because schema information is itself stored using Cassandra, it is also eventually consistent, and as a result it is possible for different nodes to have different versions of the schema. As of the 3.0 release, the Java driver does not expose the schema version directly, but you can see an example by running the `nodetool describecluster` command:

```
$ ccm node1 nodetool describecluster

Cluster Information:
     Name: test_cluster
     Snitch: org.apache.cassandra.locator.DynamicEndpointSnitch
     Partitioner: org.apache.cassandra.dht.Murmur3Partitioner
     Schema versions:
         ea46580a-4ab4-3e70-b68f-5e57da189ac5:
           [127.0.0.1, 127.0.0.2, 127.0.0.3]
```

This output shows us a couple of things. First, we see that the schema version is a UUID value. This value is calculated based on a hash of all of the keyspace and table definitions a node knows about. The fact that all three nodes share the same schema version means that they all have the same schema defined.

Of course, the schema version in use can change over time as keyspaces and tables are created, altered, and deleted. The driver provides a notification mechanism for clients to learn about these changes by registering a `com.datastax.driver.core.SchemaChangeListener` with the `Cluster`.

You can find an example of these calls by running the example `com.cassandraguide.clients.SimpleSchemaExample`.

In addition to the schema access we've just examined in the `Metadata` class, the Java driver also provides a facility for managing schema in the `com.datastax.driver.core.schemabuilder` package. The `SchemaBuilder` provides a fluent-style API for creating `SchemaStatements` representing operations such as `CREATE`, `ALTER`, and `DROP` operations on keyspaces, tables, indexes, and user-defined types (UDTs).

For example, the following code could be used to create our hotels keyspace:

```
SchemaStatement hotelSchemaStatement = SchemaBuilder.createTable("hotels").
  addPartitionKey("id", DataType.text()).
  addColumn("name", DataType.text()).
  addColumn("phone", DataType.text()).
  addColumn("address", DataType.text()).
  addColumn("pois", DataType.set(DataType.text()));

session.execute(hotelSchemaStatement);
```

We also import `com.datastax.driver.core.DataType` so that we can leverage its static operations to define the data types of each column.

Avoid Conflicts When Using Programmatic Schema Definition

Many developers have noted that this programmatic schema management capability can be used as a "lazy initialization" technique for simplified application deployment: if the schema our application uses doesn't exist, we can simply create it programmatically. However, this technique is not recommended when running multiple clients, even with `IF NOT EXISTS` semantics. `CREATE TABLE` or `ALTER TABLE` statements from multiple concurrent clients can result in inconsistent state between nodes, requiring manual repairs.

Debugging and Monitoring

The driver provides features for monitoring and debugging your client's use of Cassandra, including facilities for logging and metrics. There is also a query tracing capability, which we'll learn about in Chapter 12.

Logging

As we will learn in Chapter 10, Cassandra uses a logging API called Simple Logging Facade for Java (SLF4J). The Java driver uses the SLF4J API as well. In order to enable logging on your Java client application, you need to provide a compliant SLF4J implementation on the classpath.

Here's an example of a dependency we can add to our Maven POM file to select the Logback project as the implementation:

```
<dependency>
      <groupId>ch.qos.logback</groupId>
      <artifactId>logback-classic</artifactId>
      <version>1.1.3</version>
</dependency>
```

You can learn more about Logback at *http://logback.qos.ch/*.

By default, the Java driver is set to use the `DEBUG` logging level, which is fairly verbose. We can configure logging by taking advantage of Logback's configuration mechanism, which supports separate configuration for test and production environments. Logback inspects the classpath first for the file *logback-test.xml* representing the test configuration, and then if no test configuration is found, it searches for the file *logback.xml*.

For more detail on Logback configuration, including sample configuration files for test and production environments, see the configuration page.

Metrics

Sometimes it can be helpful to monitor the behavior of client applications over time in order to detect abnormal conditions and debug errors. The Java driver collects metrics on its activities and makes these available using the Dropwizard Metrics library. The driver reports metrics on connections, task queues, queries, and errors such as connection errors, read and write timeouts, retries, and speculative executions.

You can access the Java driver metrics locally via the `Cluster.getMetrics()` operation. The Metrics library also integrates with the Java Management Extensions (JMX) to allow remote monitoring of metrics. JMX reporting is enabled by default, but this can be overridden in the `Configuration` provided when building a `Cluster`.

DataStax Python Driver

The most popular early Python client was Pycassa, which was built on the Thrift interface. The Pycassa project is no longer maintained, however, and the recommendation is to use the DataStax Python Driver for all new development.

The first full version of the DataStax Python Driver was introduced January 2014 and provided session management, node discovery, connection pooling, synchronous/asynchronous queries, load balancing, query tracing, metrics (using the Scales library), logging, authentication and SSL. Features such as support for Cassandra 2.1 and later paging, lightweight transactions, and Python 3 support were added in the 2.0 release in May 2014. The Python Driver is compatible with Cassandra releases 1.2 and later and runs on Python 2.6, 2.7, 3.3, and 3.4. The driver runs on Linux, Mac OS, and Windows.

The official documentation for the driver is available at the DataStax website, while the source driver is available on GitHub. You can install the driver by running the Python installer *pip*:

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

Installing Python and PIP

To use the example code, you'll need a compatible version of Python for your platform (as listed earlier), and `pip`. You can install `pip` by downloading the script *https://bootstrap.pypa.io/get-pip.py* and running the command `python get-pip.py`. You may need to run this command via `sudo` on Unix systems.

Here's a simple example of connecting to a cluster and inserting a row in the `hotels` table:

```
from cassandra.cluster import Cluster
cluster = Cluster(['127.0.0.1'])
session = cluster.connect('hotel')
session.execute("""
  insert into hotels (id, name, phone)
  values (%s, %s, %s)
  """,
  ('AZ123', 'Super Hotel at WestWorld', '1-888-999-9999')
)
```

The Python driver includes an object mapper called *cqlengine*, which is accessed through the `cassandra.cqlengine.models.Model` class. The Python driver also makes use of third party libraries for performance, compression, and metrics. Some C extensions using Cython are used to speed up performance. The driver may also be run on PyPy, an alternative Python runtime that uses a JIT compiler. The reduced CPU consumption leads to improved throughput, up to two times better than regular Python. Compression requires installation of either the `lz4` or `python-snappy` libraries, depending on the desired algorithm.

DataStax Node.js Driver

The original Thrift-based client bindings for Node.js were provided by Helenus, and the *node-cassandra-cql* package by Jorge Bay subsequently provided a CQL native client.

The DataStax Node.js driver, which was officially available in October 2014, is based on *node-cassandra-cql*, adding in the standard features common across the other DataStax drivers for Apache Cassandra. You can access the driver at *https://github.com/datastax/nodejs-driver*.

The Node.js driver is installed via the node package manager (NPM):

```
$ npm install cassandra-driver
```

Installing the Node.js Runtime and Package Manager

If you don't have experience using Node, you can get an installation for your platform at *https://nodejs.org* that includes both Node.js and NPM. These are typically installed at */usr/local/bin/node* and */usr/local/bin/npm* on Unix systems.

The syntax is a bit different, in that you access a `Client` object instead of a `Cluster` as in other language drivers. The other constructs are very similar:

```
var cassandra = require('cassandra-driver');
var client = new cassandra.Client({ contactPoints: ['127.0.0.1'],
  keyspace: 'hotel'});
```

Building and executing a parameterized query looks like this:

```
var query = 'SELECT * FROM hotels WHERE id=?';
client.execute(query, ['AZ123'], function(err, result) {
  assert.ifError(err);
  console.log('got hotel with name ' + result.rows[0].name);
});
```

DataStax Ruby Driver

Fauna was an early Thrift-based Ruby client created for Cassandra by Twitter. Starting in early 2013, Theo Hultberg led development of the `cql-rb` gem, which became the basis of the DataStax Ruby Driver released in November 2014. You can access the Ruby driver at *https://github.com/datastax/ruby-driver*.

You can install the driver using Ruby Gems:

```
$ gem install cassandra-driver
```

Here's an example of creating a cluster and session and executing a simple asynchronous query that iterates over the contents of our `hotels` table:

```
require 'cassandra'

cluster = Cassandra.cluster(hosts: ['127.0.0.1'])
session  = cluster.connect('hotel')

future = session.execute_async('SELECT * FROM hotels')
future.on_success do |rows|
  rows.each do |row|
    puts "Hotel: #{row['id']} Name: #{row['name']}"
  end
```

```
end
future.join
```

The Ruby driver runs on standard Ruby, but can also be run on JRuby 1.7 or later for improved performance. The driver runs on Linux, Mac OS, but Windows is not supported.

DataStax C# Driver

First released in July 2013, the DataStax C# driver provides support for Windows clients using the .NET framework. For this reason, it is also frequently referred to as the ".NET Driver."

The C# Driver is available on NuGet, the package manager for the Microsoft development platform. Within PowerShell, run the following command at the Package Manager Console:

```
PM> Install-Package CassandraCSharpDriver
```

To use the driver, create a new project in Visual Studio and add a `using` directive that references the Cassandra namespace. The following example connects to our `hotel` keyspace and inserts a new record into the `hotels` table:

```
Cluster Cluster = Cluster.Builder()
  .AddContactPoint("127.0.0.1")  .Build();

ISession Session = Cluster.Connect("hotel");
Session.Execute(
  "INSERT INTO hotels (id, name, phone) " +
  "VALUES (" +
    "'AZ123'," +
    "'Super Hotel at WestWorld'," +
    "'1-888-999-9999'," +
    ");");
```

The C# driver integrates with Language Integrated Query (LINQ), a Microsoft .NET Framework component that adds query capabilities to .NET languages; there is a separate object mapper available as well.

A Sample Application: KillrVideo

Luke Tillman, Patrick McFadin, and others have created a video sharing application called KillrVideo. KillrVideo is an open source .NET application built using the DataStax C# driver and deployed to Microsoft's Azure cloud. It also makes use of DataStax Enterprise features such as integration with Apache Spark and Apache SOLR. You can download the source on GitHub.

DataStax C/C++ Driver

The DataStax C/C++ Driver was released in February 2014. You can access the driver at *https://github.com/datastax/cpp-driver*, and the documentation at *http://datastax.github.io/cpp-driver*.

The C/C++ Driver is a bit different than the other drivers in that its API focuses on asynchronous operations to the exclusion of synchronous operations. For example, creating a session is an asynchronous operation which returns a future:

```
#include <cassandra.h>
#include <stdio.h>

int main() {
  CassFuture* connect_future = NULL;
  CassCluster* cluster = cass_cluster_new();
  CassSession* session = cass_session_new();

  cass_cluster_set_contact_points(cluster, "127.0.0.1");

  connect_future = cass_session_connect(session, cluster);

  if (cass_future_error_code(connect_future) == CASS_OK) {
  /* proceed with processing... */
```

As shown in the example, however, synchronous semantics are easily supported by immediately blocking on the future. Building and executing a simple query looks like this:

```
CassStatement* select_statement
  = cass_statement_new("SELECT * "
                       "FROM hotel.hotels", 0);

CassFuture* hotel_select_future =
  cass_session_execute(session, select_statement);

if(cass_future_error_code(result_future) == CASS_OK) {

  const CassResult* result = cass_future_get_result(result_future);
  CassIterator* rows = cass_iterator_from_result(result);

  while(cass_iterator_next(rows)) {
    const CassRow* row = cass_iterator_get_row(rows);
    const CassValue* value =
      cass_row_get_column_by_name(row, "name");
    const char* name;
    size_t name_length;
    cass_value_get_string(value, &name, &name_length);
    printf("Hotel_name: '%.*s'\n", (int)name_length, name);
```

```
    }
}
```

Remember that memory management is very important in C/C++ programs; we've omitted statements to free objects such as clusters, sessions, futures, and results for brevity.

The C/C++ driver uses the *libuv* library for asynchronous I/O operations, and optionally uses the OpenSSL library if needed for encrypted client-node connections. Instructions for compilation and linking vary by platform, so see the driver documentation for details.

DataStax PHP Driver

The DataStax PHP driver supports the PHP server side scripting language. Released in 2015, the driver wraps the DataStax C/C++ Driver and supports both Unix and Windows environments.

There are multiple installation options for the driver, but the simplest is to use the PECL repository:

```
pecl install cassandra
```

The following short example selects rows from the hotels table and prints out their values using the asynchronous API:

```
<?php

$keyspace  = 'hotel';
$session   = $cluster->connect($keyspace);
$statement = new Cassandra\SimpleStatement(
    'SELECT * FROM hotels'
);

$future    = $session->executeAsync($statement);
$result    = $future->get();
foreach ($result as $row) {
  printf("id: %s, name: %s, phone: %s\n",
    $row['id'], $row['name'], $row['phone']);
}
```

You can access the PHP driver documentation at *https://github.com/datastax/php-driver*, and the source code at *https://datastax.github.io/php-driver*.

Summary

You should now have an understanding of the various client interfaces available for Cassandra, the features they provide, and how to install and use them. We gave particular attention to the DataStax Java driver in order to get some hands-on experi-

ence, which should serve you well even if you choose to use one of the other DataStax drivers. We'll continue to use the DataStax Java driver in the coming chapters as we do more reading and writing.