DATASTAX

Ruby Driver 2.1 for Apache Cassandra

Documentation

${ds.localized.time}

Contents

About the Ruby driver

Use this driver in production applications to pass CQL statements from the client to a cluster and retrieve, manipulate, or remove data.

The Ruby driver is a modern, feature-rich and highly tunable Java client library for Apache Cassandra (1.2+) and DataStax Enterprise (3.1+) using exclusively Cassandra's binary protocol and Cassandra Query Language v3.

Use this driver in production applications to pass CQL statements from the client to a cluster and retrieve, manipulate, or remove data. Cassandra Query Language (CQL) is the primary language for communicating with the Cassandra database. Documentation for CQL is available in CQL for Cassandra 2.x. DataStax also provides DataStax DevCenter, which is a free graphical tool for creating and running CQL statements against Apache Cassandra and DataStax Enterprise. Other administrative tasks can be accomplished using OpsCenter.

What's new?

Here are the new and noteworthy features of the Ruby driver:

- support for
  - named parameters
  - increased number of stream ids
  - the driver uses native protocol version 3
  - user-defiend types
  - tuples
- update
  - collection serialization
  - schema change event decoding
  - BATCH request encoding
- added an API for inspecting and manipulating CQL data types

Architecture

An overview of the Ruby driver architecture.

The driver architecture is a layered one. At the bottom lies the driver core. This core handles everything related to the connections to a Cassandra cluster (for example, connection pool, discovering new nodes, etc.) and exposes a simple, relatively low-level API on top of which a higher level layer can be built.

The driver has the following features:

- Asynchronous: the driver uses the new CQL binary protocol asynchronous capabilities. Only a relatively low number of connections per nodes needs to be maintained open to achieve good performance.
- Configurable load balancing: the driver allows for custom routing and load balancing of queries to Cassandra nodes. Out of the box, round robin is provided with optional data-center awareness (only nodes from the local data-center are queried (and have connections maintained to)) and optional token awareness (that is, the ability to prefer a replica for the query as coordinator).
- Node discovery: the driver automatically discovers and uses all nodes of the Cassandra cluster, including newly bootstrapped ones.
- Retry policies can be set to define a precise behavior to adopt on query execution exceptions (for example, timeouts, unavailability). This avoids polluting client code with retry-related code.
- Schema access: the driver exposes a Cassandra schema in a usable way.
- Trace handling: tracing can be set on a per-query basis and the driver provides a convenient API to retrieve the trace.
- Transparent failover: if Cassandra nodes fail or become unreachable, the driver automatically and transparently tries other nodes and schedules reconnection to the dead nodes in the background.
- Tunability: the default behavior of the driver can be changed or fine tuned by using tuning policies and connection options.

The driver and its dependencies

The Ruby driver only supports the Cassandra Binary Protocol and CQL3

Cassandra binary protocol

The driver uses the binary protocol that was introduced in Cassandra 1.2. It only works with a version of Cassandra greater than or equal to 1.2. Furthermore, the binary protocol server is not started with the default configuration file in Cassandra 1.2. You must edit the `cassandra.yaml` file for each node:

```
start_native_transport: true
```

Then restart the node.

Cassandra compatibility

The 1.0 version of the driver handles a single version of the Cassandra native protocol for the sake of simplicity. Cassandra does the multiple version handling. This makes it possible to do a rolling upgrade of a Cassandra cluster from 1.2 to 2.0 and then to upgrade the drivers in the application layer from 1.0 to 2.0. Because the application code needs to be changed anyway to leverage the new features of Cassandra 2.0, this small constraint appear to be fair.

	Ruby driver 1.0.x
Cassandra 1.2.x	Compatible
Cassandra 2.0.x	Compatible

Build environment dependencies

The driver works with the following versions of Ruby:

- Ruby 1.9.3 and 2.0
- JRuby 1.7
- Rubinius 2.1

Writing your first client

This section walks you through a small sample client application that uses the Ruby driver to connect to a Cassandra cluster, print out some metadata about the cluster, execute some queries, and print out the results.

Connecting to a Cassandra cluster

About this task

The driver provides a Cluster class which is your client application's entry point for connecting to a Cassandra cluster and retrieving metadata.

Before you begin

This tutorial assumes you have the following software installed, configured, and that you have familiarized yourself with them:

- Apache Cassandra 1,2 or greater
- One of the following:
  - Ruby 1.9.3 and 2.0
  - JRuby 1.7
  - Rubinius 2.1
- Add the driver dependency to your project's Gemfile: `gem 'cassandra-driver', => 'https://github.com/datastax/ruby-driver.git'`

About this task

Using a Cluster object, the client connects to a node in your cluster and then retrieves metadata about the cluster and prints it out.

Procedure

1. Using a text editor, create a file `simple.rb`.
   Add the following require statements:

```
require 'bundler/setup'
require 'cassandra'
```

2. Create a new Ruby class, `SimpleClient`.
   a) Add an instance field, `cluster`, to hold a Cluster reference and initialize it to `nil`.

```
class SimpleClient
   def initialize()
      @cluster = nil
   end
end
```

   b) Add an instance method, `connect`, to your new class.
   The connect method:
   - adds a contact point (node IP address) using the `Cassandra.connect` method
   - builds a cluster instance
   - retrieves metadata from the cluster

- prints out:
  - the name of the cluster
  - host IP address and id, the datacenter, and rack for each of the nodes in the cluster

```
def connect(node)
   puts "Connecting to cluster."
   @cluster = Cassandra.cluster(hosts: node)
   puts "Cluster: #{@cluster.name}"
   @cluster.each_host do |host|
      puts "Host #{host.ip}: id = #{host.id} datacenter =
 #{host.datacenter} rack = #{host.rack}"
   end
end
```

c) Add an instance method, `close`, to shut down the cluster instance once you are finished with it.

```
def close()
    @cluster.close
end
```

d) Add code at the end of the file to instantiate a `SimpleClient` object, call connect on it, and close it.

```
client = SimpleClient.new
client.connect(['127.0.0.1'])
client.close
```

3. Save the file and run it.

```
$ ruby simple.rb
```

Code listing

The complete code listing illustrates:

- connecting to a cluster
- retrieving metadata and printing it out
- closing the connection to the cluster

```
# encoding: utf-8

require 'bundler/setup'
require 'cassandra'

class SimpleClient
    def initialize()
        @cluster = nil
    end

    def connect(node)
        puts "Connecting to cluster."
        @cluster = Cassandra.cluster(hosts: node)
        puts "Cluster: #{@cluster.name}"
        @cluster.each_host do |host|
          puts "Host #{host.ip}: id = #{host.id} datacenter =
 #{host.datacenter} rack = #{host.rack}"
        end
    end
```

```
    def close()
        @cluster.close
    end
end

client = SimpleClient.new
client.connect(['127.0.0.1'])
client.close
```

When run the client program prints out this metadata on the cluster's constituent nodes in the console pane:

```
Connecting to cluster.
Cluster: darius
Host 127.0.0.1: id = 0073a525-a00d-493b-9b55-191120dd2c7e datacenter =
 datacenter1 rack = rack1
Host 127.0.0.3: id = d51d0c22-da0d-4e2b-ba43-18aa99e3bffb datacenter =
 datacenter1 rack = rack1
Host 127.0.0.2: id = b05c941f-572a-4d88-8ba6-d2efb01d653e datacenter =
 datacenter1 rack = rack1
```

Using a session to execute CQL statements

About this task

Once you have connected to a Cassandra cluster using a cluster object, you retrieve a session, which allows you to execute CQL statements to read and write data.

Before you begin

This tutorial uses a CQL schema which is described in a post on the DataStax developer blog. Reading that post, could help with some of the CQL concepts used here.

About this task

Getting metadata for the cluster is good, but you also want to be able to read and write data to the cluster. The Ruby driver lets you execute CQL statements using a session instance that you retrieve from the Cluster object. You will add code to your client for:

- creating tables
- inserting data into those tables
- querying the tables
- printing the results

Procedure

1. Modify your SimpleClient class.
   a) Add a Session instance field and initialize it to `nil` in the `initialize` method.

```
    def initialize()
        @cluster = nil
        @session = nil
    end
```

   b) Get a session from your cluster and store the reference to it.

Add the following line to the end of the `connect` method:

```
@session = @cluster.connect
```

You can execute queries by calling the execute method on your session object. The session maintains a connections pool to the cluster nodes, provides policies to choose which node to use for each query (round-robin on all nodes of the cluster by default), and handles retries for failed queries when it makes sense.

Session instances are thread-safe and usually a single instance is all you need per application. However, a given session can only be set to one keyspace at a time, so one instance per keyspace is necessary. Your application typically only needs a single cluster object, unless you're dealing with multiple physical clusters.

2. Add an instance method, `create_schema`, to the `SimpleClient` class implementation.

```
def create_schema()
end
```

3. Add the code to create a new schema.
   a) Execute a statement that creates a new keyspace.

   Add to the `create_schema` method:

```
@session.execute("CREATE KEYSPACE IF NOT EXISTS simplex WITH replication
 " +
   "= {'class':'SimpleStrategy', 'replication_factor':3};")
```

   In this example, you create a new keyspace, `simplex`.
   b) Execute statements to create two new tables, songs and playlists.
   Add to the `createSchema` method:

```
@session.execute("CREATE TABLE IF NOT EXISTS simplex.songs (" +
       "id uuid PRIMARY KEY," +
       "title text," +
       "album text," +
       "artist text," +
       "tags set<text>," +
       "data blob" +
    ");")
@session.execute("CREATE TABLE IF NOT EXISTS simplex.playlists (" +
       "id uuid," +
       "title text," +
       "album text, " +
       "artist text," +
       "song_id uuid," +
       "PRIMARY KEY (id, title, album, artist)" +
    ");")
puts 'Simplex keyspace and schema created.'
```

4. Add an instance method, `loadData`, to the SimpleCient class implementation.

```
def load_data()
end
```

5. Add the code to insert data into the new schema.

```
@session.execute(
    "INSERT INTO simplex.songs (id, title, album, artist, tags) " +
    "VALUES (" +
        "756716f7-2e54-4715-9f00-91dcbea6cf50," +
        "'La Petite Tonkinoise'," +
        "'Bye Bye Blackbird'," +
        "'Joséphine Baker'," +
        "{'jazz', '2013'})" +
    ";")
@session.execute(
    "INSERT INTO simplex.playlists (id, song_id, title, album, artist) " +
    "VALUES (" +
        "2cc9ccb7-6221-4ccb-8387-f22b6a1b354d," +
        "756716f7-2e54-4715-9f00-91dcbea6cf50," +
        "'La Petite Tonkinoise'," +
        "'Bye Bye Blackbird'," +
        "'Joséphine Baker'" +
    ");")
```

6. Add calls to the new `createSchema` and `loadData` methods at the end of the file before the call to `close`.

```
client = SimpleClient.new
client.connect(['127.0.0.1'])
client.create_schema
client.load_data
client.close
```

7. Add an instance method, `querySchema`, that executes a `SELECT` statement on the tables and then prints out the results.

   a) Add code to execute the query.
   Query the playlists table for one of the two records.

```
results = @session.execute(
    'SELECT * FROM simplex.playlists ' +
    'WHERE id = 2cc9ccb7-6221-4ccb-8387-f22b6a1b354d;')
```

   The execute method returns a `Result` that holds rows returned by the `SELECT` statement.

   b) Add code to iterate over the rows and print them out.

```
puts "%-30s\t%-20s\t%-20s\n%s" %
    ['title', 'album', 'artist', '-------------------------------
+----------------------+--------------------']
results.each do |row|
    puts "%-30s\t%-20s\t%-20s" % [ row['title'], row['album'],
 row['artist'] ]
end
```

8. Add a call to the new `querySchema` method.

```
client = SimpleClient.new
client.connect(['127.0.0.1'])
client.create_schema
client.load_data
client.query_schema
client.close
```

9. Add an instance method, `dropSchema`, that takes a single parameter and implement it to drop the keyspace.

```
def drop_schema(keyspace)
    @session.execute("DROP KEYSPACE " + keyspace + ";")
    puts keyspace + " keyspace dropped."
end
```

Code listing

The complete code listing illustrates:

- retrieving a session object
- calling execute CQL statements on the session to
  - create a keyspace
  - create tables in the keyspace
  - insert data into the tables
  - query for data in the tables

```
# encoding: utf-8

require 'bundler/setup'
require 'cassandra'

class SimpleClient
    def initialize()
        @cluster = nil
        @session = nil
    end

    def connect(node)
        puts "Connecting to cluster."
        @cluster = Cassandra.cluster(hosts: node)
        puts "Cluster: #{@cluster.name}"
        @cluster.each_host do |host|
          puts "Host #{host.ip}: id = #{host.id} datacenter =
 #{host.datacenter} rack = #{host.rack}"
        end
        @session = @cluster.connect
    end

    def create_schema()
        @session.execute("CREATE KEYSPACE IF NOT EXISTS simplex WITH
 replication " +
            "= {'class':'SimpleStrategy', 'replication_factor':3};")
        @session.execute("CREATE TABLE IF NOT EXISTS simplex.songs (" +
               "id uuid PRIMARY KEY," +
               "title text," +
               "album text," +
               "artist text," +
               "tags set<text>," +
               "data blob" +
            ");")
        @session.execute("CREATE TABLE IF NOT EXISTS simplex.playlists (" +
               "id uuid," +
               "title text," +
               "album text, " +
               "artist text," +
               "song_id uuid," +
```

```
                "PRIMARY KEY (id, title, album, artist)" +
            ");")
        puts "Simplex keyspace and schema created."
    end

    def load_data()
        @session.execute(
            "INSERT INTO simplex.songs (id, title, album, artist, tags) " +
            "VALUES (" +
                "756716f7-2e54-4715-9f00-91dcbea6cf50," +
                "'La Petite Tonkinoise'," +
                "'Bye Bye Blackbird'," +
                "'Joséphine Baker'," +
                "{'jazz', '2013'})" +
            ";")
        @session.execute(
            "INSERT INTO simplex.playlists (id, song_id, title, album,
 artist) " +
            "VALUES (" +
                "2cc9ccb7-6221-4ccb-8387-f22b6a1b354d," +
                "756716f7-2e54-4715-9f00-91dcbea6cf50," +
                "'La Petite Tonkinoise'," +
                "'Bye Bye Blackbird'," +
                "'Joséphine Baker'" +
            ");")
    end

    def query_schema()
        results = @session.execute(
            "SELECT * FROM simplex.playlists " +
            "WHERE id = 2cc9ccb7-6221-4ccb-8387-f22b6a1b354d;")
        puts "%-30s\t%-20s\t%-20s\n%s" %
            ['title', 'album', 'artist', '-------------------------------
+----------------------+--------------------']
        results.each do |row|
            puts "%-30s\t%-20s\t%-20s" % [ row['title'], row['album'],
 row['artist'] ]
        end
    end

    def drop_schema(keyspace)
        @session.execute("DROP KEYSPACE " + keyspace + ";")
        puts keyspace + " keyspace dropped."
    end

    def close()
        @cluster.close
    end
end

client = SimpleClient.new
client.connect(['127.0.0.1'])
client.create_schema
client.load_data
client.drop_schema("simplex")
client.query_schema
client.close
```

When run the client program prints out something like the following in the console pane:

```
Connecting to cluster.
Cluster: darius
```

```
Host 127.0.0.1: id = 0073a525-a00d-493b-9b55-191120dd2c7e datacenter =
 datacenter1 rack = rack1
Host 127.0.0.3: id = d51d0c22-da0d-4e2b-ba43-18aa99e3bffb datacenter =
 datacenter1 rack = rack1
Host 127.0.0.2: id = b05c941f-572a-4d88-8ba6-d2efb01d653e datacenter =
 datacenter1 rack = rack1
Simplex keyspace and schema created.
title                          album                artist
-------------------------------+----------------------+--------------------
La Petite Tonkinoise           Bye Bye Blackbird    Joséphine Baker
```

Using prepared statements

About this task

In the previous tutorial, you added a `loadData` method which creates a new statement for each `INSERT`, but you may also use prepared statements and bind new values to the columns each time before execution. Doing this increases performance, especially for repeated queries. You add code to your client for:

- creating a prepared statement
- creating a bound statement from the prepared statement and binding values to its variables
- executing the bound statement to insert data

Procedure

1. Using a text editor, create a file `bound.rb`.
   Add the following require statements:

```
require 'bundler/setup'
require 'cassandra'
require_relative 'simple'
```

2. Create a new class, `BoundStatementsClient` which overrides the `SimpleClient` class.

```
class BoundStatementsClient < SimpleClient
end
```

   a) Add two instance fields, `insertSongStatement` and `insertPlaylistStatement`, to hold the prepared statements and initialize them to `nil`.

```
def initialize()
   @insertSongStatement = nil
   @insertPlaylistStatement = nil
end
```

   b) Add an instance method, `prepareStatements`, to the class.

```
def prepare_statements()
    @insertSongStatement = @session.prepare(
        "INSERT INTO simplex.songs " +
            "(id, title, album, artist, tags) " +
            "VALUES (?, ?, ?, ?, ?);")
    @insertPlaylistStatement = @session.prepare(
        "INSERT INTO simplex.playlists " +
```

```
                "(id, song_id, title, album, artist) " +
                "VALUES (?, ?, ?, ?, ?);")
    end
```

3. Override the `loadData` method and implement it.

```
def load_data()
end
```

   a) Add code to bind values to the prepared statement's variables and execute it.
   You create a bound statement by calling its constructor and passing in the prepared statement. Use the bind method to bind values and execute the bound statement on the your session..

```
@session.execute(
    @insertSongStatement,
    Cassandra::Uuid.new("756716f7-2e54-4715-9f00-91dcbea6cf50"),
    "La Petite Tonkinoise",
    "Bye Bye Blackbird",
    "Joséphine Baker",
    ['jazz', '2013'].to_set)
```

   Note that you cannot pass in string representations of UUIDs or sets as you did in the `loadData` method.

   b)

4. Add code to create a new bound statement for inserting data into the `simplex.playlists` table.

```
@session.execute(
    @insertPlaylistStatement,
    Cassandra::Uuid.new("2cc9ccb7-6221-4ccb-8387-f22b6a1b354d"),
    Cassandra::Uuid.new("756716f7-2e54-4715-9f00-91dcbea6cf50"),
    "La Petite Tonkinoise",
    "Bye Bye Blackbird",
    "Joséphine Baker")
```

5. Add a call to the `BoundStatementsClient`constructor and the `prepareStatement` method.

```
client = BoundStatementsClient.new
client.connect(['127.0.0.1'])
client.create_schema
client.prepare_statements
client.load_data
client.query_schema
client.close
```

Code listing

```
# SimpleClient code skipped, see previous topic.

class BoundStatementsClient < SimpleClient
    def initialize()
        @insertSongStatement = nil
        @insertPlaylistStatement = nil
    end

    def prepare_statements()
        @insertSongStatement = @session.prepare(
```

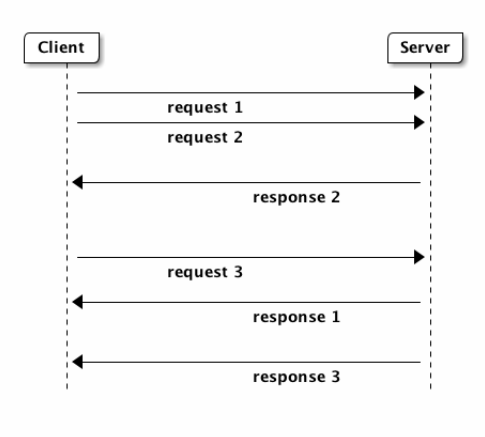
```
            "INSERT INTO simplex.songs " +
                "(id, title, album, artist, tags) " +
                "VALUES (?, ?, ?, ?, ?);")
        @insertPlaylistStatement = @session.prepare(
            "INSERT INTO simplex.playlists " +
                "(id, song_id, title, album, artist) " +
                "VALUES (?, ?, ?, ?, ?);")
    end

    def load_data()
        @session.execute(
            @insertSongStatement,
            Cassandra::Uuid.new("756716f7-2e54-4715-9f00-91dcbea6cf50"),
            "La Petite Tonkinoise",
            "Bye Bye Blackbird",
            "Joséphine Baker",
            ['jazz', '2013'].to_set)
        @session.execute(
            @insertPlaylistStatement,
            Cassandra::Uuid.new("2cc9ccb7-6221-4ccb-8387-f22b6a1b354d"),
            Cassandra::Uuid.new("756716f7-2e54-4715-9f00-91dcbea6cf50"),
            "La Petite Tonkinoise",
            "Bye Bye Blackbird",
            "Joséphine Baker")
    end
end

client = BoundStatementsClient.new
client.connect(['127.0.0.1'])
client.createSchema
client.prepareStatements
client.loadData
client.querySchema
client.dropSchema("simplex")
client.close
```

Ruby driver reference

Reference for the Ruby driver.

Asynchronous I/O

Cassandra's native binary protocol supports request pipelining, which lets a single connection be used for several simultaneous and independent request-response exchanges. Additionally, the driver doesn't use any blocking APIs internally and runs all requests in the background reactor thread.



All asynchronous methods end with `_async`, (for example, `Session#execute_async`) and return a `Future` object. Blocking methods like `Session#prepare`, `Session#execute`, and `Session#close` are thin wrappers around `Session#prepare_async`, `Session#execute_async`, and `Session#close_async` accordingly. These wrapper methods call their asynchronous counterpart and block waiting for the resulting future to be resolved. A `Future` can be used to:

- block the application thread until execution has completed
- register a listener to be notified when a result is available

When describing different asynchronous method results, we use a `Cassandra::Future<Type>` notation to signal the type of the result of the future. For example, `Cassandra::Future<Cassandra::Result>` is a future that returns an instance of `Cassandra::Result` when calling its get method.

Asynchronous I/O example

About this task

You can execute statements on a session objects in two different ways. Calling `execute` blocks the calling thread until the statement finishes executing, but a session also allows for asynchronous and non-blocking I/O by calling the `execute_async` method.

Modify the functionality of the `SimpleClient` class by extending it and execute queries asynchronously on a cluster.

Procedure

1. Add a new class, `AsynchronousExample`, to your `simple.rb` file. It should extend the `SimpleClient` class.

```
class AsynchronousExample < SimpleClient
end
```

2. Overload the `querySchema` method and implement it.
   a) Add the method declaration.

```
def querySchema()
end
```

   b) Execute a query using the `execute_async` method on the session object.

```
def querySchema()
    future = @session.execute_async(
        "SELECT * FROM simplex.playlists " +
        "WHERE id = 2cc9ccb7-6221-4ccb-8387-f22b6a1b354d;")
end
```

   c) Call the `on_success` method on the returned future object and then loop over the rows return, printing out the columns

```
def getRows()
    future = @session.execute_async(
        "SELECT * FROM simplex.playlists " +
        "WHERE id = 2cc9ccb7-6221-4ccb-8387-f22b6a1b354d;")
    puts "%-30s\t%-20s\t%-20s\n%s" %
        ['title', 'album', 'artist', '-------------------------------
+----------------------+--------------------']
    results = future.get()
    results.each do |row|
        puts "%-30s\t%-20s\t%-20s" % [ row['title'], row['album'],
 row['artist'] ]
    end
end
```

3. Modify the code at the end of your class implementation, adding calls to instantiate a new client object, create the schema, load the data, and then query it using the `querySchema` method.

```
client = AsynchronousExample.new()
client.connect(['127.0.0.1'])
client.createSchema()
client.loadData()
client.querySchema()
client.dropSchema("simplex")
client.close()
```

Of course, in our implementation, the call to `Future#get` blocks until the future has completed execution of the statement on the session object. Functionally it is no different from executing the SELECT query synchronously. Alternately, you can register listener using `Future#on_success`.

AsynchronousExample code listing

```
class AsynchronousExample < SimpleClient
    def querySchema()
        future = @session.execute_async(
            "SELECT * FROM simplex.playlists " +
            "WHERE id = 2cc9ccb7-6221-4ccb-8387-f22b6a1b354d;")
        puts "%-30s\t%-20s\t%-20s\n%s" %
```

```
            ['title', 'album', 'artist', '-------------------------------
+----------------------+--------------------']
        results = future.get()
        results.each do |row|
            puts "%-30s\t%-20s\t%-20s" % [ row['title'], row['album'],
 row['artist'] ]
        end
    end
end

client = AsynchronousExample.new()
client.connect(['127.0.0.1'])
client.createSchema()
client.loadData()
client.querySchema()
client.dropSchema("simplex")
client.close()
```

BATCH statements

Session objects can be used to construct a logged BATCH statement and later execute it.

The examples assume this schema:

```
CREATE KEYSPACE simplex WITH replication = {'class': 'SimpleStrategy',
 'replication_factor': 3};
USE simplex;
CREATE TABLE songs (
    id uuid PRIMARY KEY,
    title text,
    album text,
    artist text,
    tags set<text>,
    data blob
);
CREATE TABLE cas_batch (k text, v int, PRIMARY KEY (k, v));
```

Executing a BATCH of simple statements

Running this code:

```
require 'cassandra'

cluster = Cassandra.cluster
session = cluster.connect("simplex")

rows = session.execute("SELECT * FROM songs")

puts "songs contain #{rows.size} rows"

batch   = session.batch do |b|
    b.add("INSERT INTO songs (id, title, album, artist, tags)
         VALUES (
            756716f7-2e54-4715-9f00-91dcbea6cf50,
            'La Petite Tonkinoise',
            'Bye Bye Blackbird',
            'Joséphine Baker',
```

```
            {'jazz', '2013'}
         )")
    b.add("INSERT INTO songs (id, title, album, artist, tags)
         VALUES (
            f6071e72-48ec-4fcb-bf3e-379c8a696488,
            'Die Mösch',
            'In Gold',
            'Willi Ostermann',
            {'kölsch', '1996', 'birds'}
         )")
    b.add("INSERT INTO songs (id, title, album, artist, tags)
         VALUES (
            fbdf82ed-0063-4796-9c7c-a3d4f47b4b25,
            'Memo From Turner',
            'Performance',
            'Mick Jager',
            {'soundtrack', '1991'}
         )")
end

puts "inserting rows in a batch"

session.execute(batch, consistency: :all)
rows = session.execute("SELECT * FROM songs")

puts "songs contain #{rows.size} rows"
```

Contains this output:

```
songs contain 0 rows
inserting rows in a batch
songs contain 3 rows
```

Executing a BATCH with parameterized statements

Running this code:

```
require 'cassandra'

cluster = Cassandra.cluster
session = cluster.connect("simplex")

rows = session.execute("SELECT * FROM songs")

puts "songs contain #{rows.size} rows"

batch   = session.batch do |b|
    b.add("INSERT INTO songs (id, title, album, artist, tags) VALUES
 (?, ?, ?, ?, ?)",
        [Cassandra::Uuid.new('756716f7-2e54-4715-9f00-91dcbea6cf50'),
        'La Petite Tonkinoise',
        'Bye Bye Blackbird',
        'Joséphine Baker',
        Set['jazz', '2013']]
    )
    b.add("INSERT INTO songs (id, title, album, artist, tags) VALUES
 (?, ?, ?, ?, ?)",
        [Cassandra::Uuid.new('f6071e72-48ec-4fcb-bf3e-379c8a696488'),
        'Die Mösch',
        'In Gold',
        'Willi Ostermann',
```

```
            Set['kölsch', '1996', 'birds']]
        )
        b.add("INSERT INTO songs (id, title, album, artist, tags) VALUES
     (?, ?, ?, ?, ?)",
            [Cassandra::Uuid.new('fbdf82ed-0063-4796-9c7c-a3d4f47b4b25'),
            'Memo From Turner',
            'Performance',
            'Mick Jager',
            Set['soundtrack', '1991']]
        )
    end

    puts "inserting rows in a batch"

    session.execute(batch, consistency: :all)
    rows = session.execute("SELECT * FROM songs")

    puts "songs contain #{rows.size} rows"
```

Contains this output:

```
songs contain 0 rows
inserting rows in a batch
songs contain 3 rows
```

Executing a BATCH with named parameterized statements

```
    require 'cassandra'

    cluster = Cassandra.cluster
    session = cluster.connect("simplex")

    rows = session.execute("SELECT * FROM songs")

    puts "songs contain #{rows.size} rows"

    insert  = session.prepare("INSERT INTO songs (id, title, album, artist,
     tags) VALUES (:a, :b, :c, :d, :e)")
    batch   = session.batch do |b|
        b.add(insert,
            {:a => Cassandra::Uuid.new('756716f7-2e54-4715-9f00-91dcbea6cf50'),
            :b => 'La Petite Tonkinoise',
            :c =>'Bye Bye Blackbird',
            :d => 'Joséphine Baker',
            :e => Set['jazz', '2013']}
        )
        b.add(insert,
            {:a => Cassandra::Uuid.new('f6071e72-48ec-4fcb-bf3e-379c8a696488'),
            :b => 'Die Mösch',
            :c => 'In Gold',
            :d => 'Willi Ostermann',
            :e => Set['kölsch', '1996', 'birds']}
        )
        b.add(insert,
            {:a => Cassandra::Uuid.new('fbdf82ed-0063-4796-9c7c-a3d4f47b4b25'),
            :b => 'Memo From Turner',
            :c => 'Performance',
            :d => 'Mick Jager',
            :e => Set['soundtrack', '1991']}
        )
    end
```

```
puts "inserting rows in a batch"

session.execute(batch, consistency: :all)
rows = session.execute("SELECT * FROM songs")

puts "songs contain #{rows.size} rows"
```

Contains this output:

```
songs contain 0 rows
inserting rows in a batch
songs contain 3 rows
```

A cas_batch is never applied more than once

Running this code:

```
require 'cassandra'

cluster = Cassandra.cluster
session = cluster.connect("simplex")

statement = session.prepare("INSERT INTO cas_batch (k, v) VALUES (?, ?) IF
 NOT EXISTS")
batch = session.batch

batch.add("INSERT INTO cas_batch (k, v) VALUES ('key1', 0)")
batch.add(statement, ["key1", 1])
batch.add(statement, ["key1", 2])

results =  session.execute(batch)
rows = results.first
puts "batch applied? #{rows["[applied]"]}"

results =  session.execute(batch)
rows = results.first
puts "batch applied? #{rows["[applied]"]}"
```

Contains this output:

```
batch applied? true
batch applied? false
```

Cassandra 1.2 does not support BATCH statements

Running this code:

```
require 'cassandra'
cluster = Cassandra.cluster
session = cluster.connect("simplex")
batch   = session.batch
batch.add("INSERT INTO cas_batch (k, v) VALUES ('key1', 0)")
begin
    session.execute(batch)
rescue => e
    puts "#{e.class.name}: #{e.message}"
end
```

Contains this output:

```
Cassandra::Errors::ClientError: Batch statements are not supported by the
 current version of Apache Cassandra
```

Cluster configuration

You can modify the tuning policies and connection options for a cluster as you build it.

The configuration of a cluster cannot be changed after it has been built. There are some miscellaneous properties (such as whether metrics are enabled, contact points, and which authentication information provider to use when connecting to a Cassandra cluster).

Tuning policies

Tuning policies determine load balancing, retrying queries, and reconnecting to a node.

Load-balancing policy

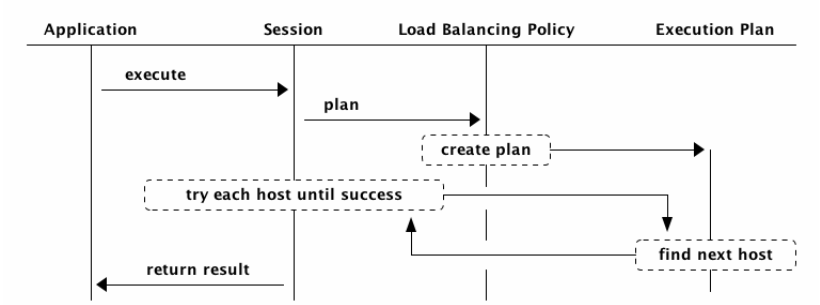
The load-balancing policy determines which node to execute a query on.

Description

Load-balancing policies are responsible for routing requests, determining which nodes the driver must connect to as well as the order in which different hosts are tried in case of network failures. Load-balancing policies therefore must also be state listeners and receive notifications about cluster membership and availability changes.

Each cluster can be configured with a specific load-balancing policy to be used, and the same policy is used for all requests across all sessions managed by that cluster instance.

Here is a high-level diagram of how a load-balancing policy is used during query



execution:

Default load-balancy policy

By default, the driver uses a combination of token-aware and datacenter-aware round-robin policies for load balancing, because this combination proved to be the most performant of of all the built-in load-balancing policies.

When the name of the local datacenter is not specified explicitly using `Cassandra.cluster`, the first datacenter seen by the load-balancing policy is considered local. Therefore, care must be taken to only include addresses of the nodes in the same datacenter as the application using the driver in the `:hosts` option to `Cassandra.cluster`, or to provide `:datacenteroption` explicitly.

Round-robin policy

The round-robin load balancing policy dispatches requests evenly on cluster nodes. The effects of the policy can be seen by enabling request tracing. The coordinator node that served every request is the last host in execution info.

The round-robin policy ignores datacenters.

Datacenter-aware round-robin policy

A specialized round-robin load balancing policy allows for querying remote datacenters only when all local nodes are down. This policy will round robin requests across hosts in the local datacenter, falling back to remote datacenter if necessary. The name of the local datacenter must be supplied by the user.

All known remote hosts are tried when local nodes are not available. However, you can configure the exact number of remote hosts that are used by passing that number when constructing a policy instance.

By default, this policy will not attempt to use remote hosts for local consistencies (`:local_one` or `:local_quorum`), however, it is possible to change that behavior via constructor.

Token-aware load-balancing policy

The token-aware load-balancing policy is used to reduce network hops whenever possible by sending requests directly to the node that owns the data. The token-aware load-balancing policy acts as a filter, wrapping another load balancing policy.

Token-aware load-balancing policy uses schema metadata available in the cluster to determine the right partitioners and replication strategies for a given keyspace and locate replicas for a given statement.

In case replica node(s) cannot be found or reached, this policy falls back onto the wrapped policy plan.

Whitelist load-balancing policy

The whitelist load-balancing policy wraps a subpolicy and ensures that only hosts from a provided whitelist are used. It can be used to limit effects of automatic peer discovery to executing queries only on a given set of hosts.

Implementing a load-balancing policy

To implement a load-balancing policy, you must implement all of the methods specified in `Cassandra::LoadBalancing::Policy`. Currently, load-balancing policies are required to be thread-safe.

The object returned from the `plan` method must implement all methods of `Cassandra::LoadBalancing::Plan` Plan will be accessed from multiple threads, but never in parallel and it doesn't have to be thread-safe. For example:

File `ignoring_keyspace_policy.rb`:

```
class IgnoringKeyspacePolicy
    class Plan
        def has_next?
            false
        end

        def next
            nil
        end
    end

    def initialize(keyspace_to_ignore, original_policy)
        @keyspace = keyspace_to_ignore
        @policy   = original_policy
    end

    def setup(cluster)
    end
```

```
    def plan(keyspace, statement, options)
        if @keyspace == keyspace
            Plan.new
        else
            @policy.plan(keyspace, statement, options)
        end
    end

    def distance(host)
        @policy.distance(host)
    end

    def host_found(host)
        @policy.host_found(host)
    end

    def host_lost(host)
        @policy.host_lost(host)
    end

    def host_up(host)
        @policy.host_up(host)
    end

    def host_down(host)
        @policy.host_down(host)
    end
end
```

`ignoring_keyspace_policy.rb` listing:

```
require 'cassandra'
require_relative 'ignoring_keyspace_policy'

policy  = IgnoringKeyspacePolicy.new('simplex',
 Cassandra::LoadBalancing::Policies::RoundRobin.new)
cluster = Cassandra.cluster(load_balancing_policy: policy)
session = cluster.connect('simplex')

begin
    session.execute("SELECT * FROM songs")
    puts "Failure."
rescue Cassandra::Errors::NoHostsAvailable
    puts "Success."
end
```

Run `ignoring_keyspace_policy.rb` against a Cassandra cluster using the example schema.

Output:

```
Success.
```

Reconnection policy

Automatic reconnection

The driver automatically reestablishes failed connections to Cassandra nodes. It uses a reconnection policy to determine retry intervals for reconnection.

`printing_listener.rb` listing

```
class MembershipChangePrintingListener
    def initialize(io)
        @out = io
    end

    def host_found(host)
        @out.puts("Host #{host.ip} is found")
        @out.flush
    end

    def host_lost(host)
        @out.puts("Host #{host.ip} is lost")
        @out.flush
    end

    def host_up(host)
        @out.puts("Host #{host.ip} is up")
        @out.flush
    end

    def host_down(host)
        @out.puts("Host #{host.ip} is down")
        @out.flush
    end
end
```

`listener_example.rb` listing:

```
require 'cassandra'
require 'printing_listener'

interval = 2 # reconnect every 2 seconds
policy   = Cassandra::Reconnection::Policies::Constant.new(interval)
cluster  = Cassandra.cluster(
        listeners: [PrintingListener.new($stdout)],
        reconnection_policy: policy,
        consistency: :one
        )
session = cluster.connect

$stdout.puts("=== START ===")
$stdout.flush
until (input = $stdin.gets).nil? # block until closed
    query = input.chomp
    begin
        execution_info = session.execute(query).execution_info
        $stdout.puts("Query #{query.inspect} fulfilled by
 #{execution_info.hosts.last.ip}")
    rescue => e
        $stdout.puts("Query #{query.inspect} failed with #{e.class.name}:
 #{e.message}")
    end
    $stdout.flush
end
$stdout.puts("=== STOP ===")
$stdout.flush
```

1. Run `listener_example.rb` against a three-node cluster using the example schema.

2. Wait for "=== START ===" to be printed to the console.
3. Stops nodes one through three.
4. Enter SELECT * FROM simplex.songs;.
5. Restart the node one.
6. Enter the statement again: SELECT * FROM simplex.songs;.
7. Enter nothing to quit the program.

The output:

```
$ run listener_example.rb
Host 127.0.0.1 is found
Host 127.0.0.1 is up
Host 127.0.0.1 is down
Host 127.0.0.1 is lost
Host 127.0.0.1 is found
Host 127.0.0.1 is up
Host 127.0.0.3 is found
Host 127.0.0.3 is up
Host 127.0.0.2 is found
Host 127.0.0.2 is up
=== START ===
Host 127.0.0.1 is down
Host 127.0.0.2 is down
Host 127.0.0.3 is down
select * from simplex.songs;
Query "select * from simplex.songs;" failed with
 Cassandra::Errors::NoHostsAvailable: no hosts available, check #errors
 property for details
Host 127.0.0.1 is up
select * from simplex.songs;
Query "select * from simplex.songs;" fulfilled by 127.0.0.1

=== START ===
$
```

Retry policy

The retry policy determines a default behavior to adopt when a request either times out or if a node is unavailable.

Description

Retry policies allow the driver to retry a request upon encountering specific types of server errors:

- write timeout
- read timeout
- unavailable

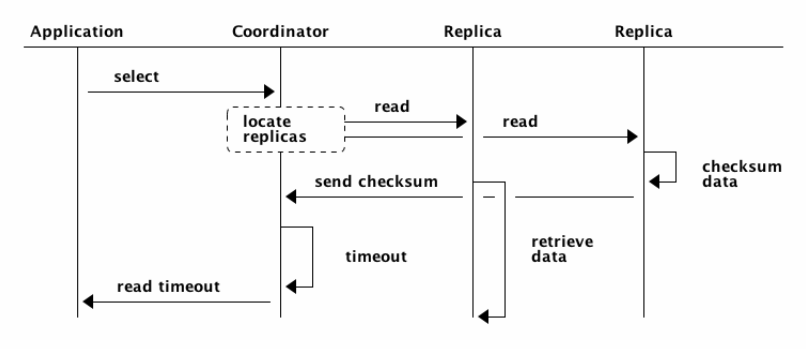
Write timeout

When a coordinator receives the request and sends the write to replica(s) but the replica(s) do not respond in time.

In this scenario, Cassandra::Retry::Policy#write_timeout is used to determine the desired course of action.

Read timeout

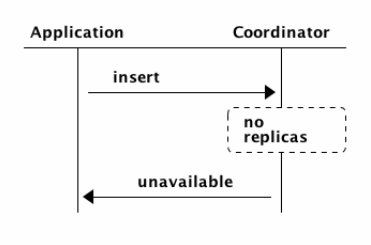
When a coordinator receives the request and sends the read to the replica(s) but the replica(s) do not respond in time.



In this scenario, `Cassandra::Retry::Policy#read_timeout` is used to determine the desired course of action.

Unavailable

When a coordinator is aware that there aren't enough replica online. No requests are sent to replica nodes in this scenario, because coordinator knows that the requested consistency level cannot be possibly satisfied.



In this scenario, `Cassandra::Retry::Policy#unavailable` is used to determine the desired course of action.

Downgrading consistency retry policy

The downgrading consistency retry policy retries failed queries with a lower consistency level than the one initially requested.

Doing so may break consistency guarantees. In other words, if you use this retry policy, there are cases where a read at `QUORUM` may not see a preceding write at `QUORUM`. Do not use this policy unless you have understood the cases where this can happen and that is all right.

```
require 'cassandra'

cluster = Cassandra.cluster(retry_policy:
 Cassandra::Retry::Policies::DowngradingConsistency.new)
session = cluster.connect('simplex')
result  = session.execute('SELECT * FROM songs', consistency: :all)

puts "Actual consistency: #{result.execution_info.consistency}"
```

Output:

```
Actual consistency: quorum
```

Fall-through retry policy

The fall-through retry policy prevents the driver from retrying queries when they have failed. It should be used when the retry policy has to be implemented in business logic.

```
require 'cassandra'

cluster = Cassandra.cluster(retry_policy:
 Cassandra::Retry::Policies::Fallthrough.new)
session = cluster.connect('simplex')

begin
    session.execute('SELECT * FROM songs', consistency: :all)
    puts "Failed."
rescue Cassandra::Errors::UnavailableError => e
    puts "Success."
end
```

Output:

```
Success.
```

Connections options

When you instantiate a cluster object using the `Cassandra.cluster` method, you can change connection options by passing in one or more of those following:

Table 1: Connection options

Key	Type	Description
`:hosts`	`Array<String, IPAddr>`	A list of initial addresses. The entire list of cluster nodes are discovered automatically once a connection to any host from the original list is successful. Default: ['127.0.0.1'].
`:port`	`Integer`	Cassandra native protocol port. Default: 9042.
`:datacenter`	`String`	Name of current datacenter. First datacenter found will be assumed current by default. You can skip this option if you specify only hosts from the local datacenter in `:hosts` option. Default: `nil`.
`:connection_timeout`	`Numeric`	Connection timeout in seconds. Default: 10.
`:timeout`	`Numeric`	Request execution timeout in seconds. Default: 10.
`:heartbeat_interval`	`Numeric`	How often a heartbeat is sent to determine if a connection is alive. Only one heartbeat request is ever outstanding

Key	Type	Description
		on a given connection. Each heatbeat is sent in at least `:heartbeat_interval` seconds after the last request has been sent on a given connection. Default: 30.
`:idle_timeout`	`Numeric`	Period of inactivity after which a connection is considered dead. This value should be at least a few times larger than `:heartbeat_interval`. Default: 60.
`:username`	`String`	The username to use for authentication with Cassandra. You must specify a `:password` option as well. Default: `none`.
`:password`	`String`	The password to use for authentication with Cassandra. You must specify a `:username` option as well. Default: `none`.
`:ssl`	`Boolean` or `OpenSSL::SSL::SSLContext`	Enable the default SSL authentication if `true` (not recommended). Also accepts an initialized `OpenSSL::SSL::SSLContext`. This option is ignored if `:server_cert`, `:client_cert`, `:private_key`, or `:passphrase` are given. Default: `false`.
`:server_cert`	`String`	Path to server certificate or certificate authority file. Default: `none`.
`:client_cert`	`String`	Path to client certificate file. This option is only required when encryption is configured to require client authentication. Default: `none`.
`:private_key`	`String`	Path to client private key. This option is only required when encryption is configured to require client authentication. Default: `none`.

Key	Type	Description
`:passphrase`	`String`	Passphrase for private key. Default: `none`.
`:compression`	`Symbol`	Compression to use. This option must be either `:snappy` or `:lz4`. In order for compression to work, you must install the `snappy` or `lz4-ruby` gems. Default: `none`.
`:load_balancing_policy`	`LoadBalancing::Policy`	Default: token-aware datacenter-aware round robin policy.
`:address_resolution`	`Symbol`	A pre-configured address resolver to use. The value of this option must be either `:none` or `:ec2_multi_region`. Default: `none`.
`:reconnection_policy`	`Reconnection::Policy`	Default: `Reconnection::Policies::Exponent` The default policy is configured with `(0.5, 30, 2)`.
`:retry_policy`	`Retry::Policy`	Default: `Retry::Policies::Default`.
`:listeners`	`Enumerable<Listener>`	A list of initial cluster state listeners. Note that a `:load_balancing_policy` is automatically registered with the cluster. Default: `none`.
`:logger`	`Logger`	a `Logger` instance from the standard library or any object responding to standard log methods (`#debug`, `#info`, `#warn`, `#error`, and `#fatal`). Default: `none`.
`:consistency`	`Symbol`	default consistency to use for all requests. The value of this option must be one of CONSISTENCIES. Default: `:one`.
`:trace`	`Boolean`	Whether or not to trace all requests. Default: `false`.
`:page_size`	`Integer`	The page size for all select queries. Default: `nil`.
`:credentials`	`Hash{String => String}`	A hash of credentials to be used with credentials

Key	Type	Description
		authentication in cassandra 1.2. If you also specify the `:username` and `:password` options, these credentials are configured automatically. Default: `none`.
`:auth_provider`	`Auth::Provider`	A custom auth provider to be used with SASL authentication in cassandra 2.0.If you have also specified the `:username` and `:password` options, a `Auth::Providers::Password` instance is used automatically. Default: `none`.
`:compressor`	`Cassandra::Compressor`	A custom compressor. If you have also specified the `:compression` option, an appropriate compressor is provided automatically. Default: `none`.
`:address_resolution_policy`	`AddressResolution::Policy`	Default: `AddressResolution::Policies::Non` a custom address resolution policy. If you havealso specified `:address_resolution`, an appropriate address resolution policy ise provided automatically.
`:futures_factory`	`Object<#all, #error,#value, #promise>`	A custom futures factory to assist with integration into existing futures library. Promises returned by this object must conform to the `Promise` API, which is not yet public. Things may change, use at your own risk. Default: `none`.

Connection requirements

In order to ensure that the driver can connect to the Cassandra or DSE cluster, please check the following requirements.

- the cluster is running Apache Cassandra 1.2+ or DSE 3.2+
- you have configured the following in the `cassandra.yaml` you have :

```
start_native_transport : true
rpc_address : IP address or hostname reachable from the client
```

- machines in the cluster can accept connections on port 9042

Note: The client port can be configured using the native_transport_port in `cassandra.yaml`.

CQL data types to Ruby types

A summary of the mapping between CQL data types and Ruby data types is provided.

Apache Cassandra supports a variety of datatypes. The driver transparently maps each of these datatypes to a specific Ruby type.

Description

When retrieving the value of a column from a Row object, you use a getter based on the type of the column.

Table 2: Ruby classes to CQL data types

CQL data type	Ruby type
ascii	String
bigint	Numeric
blob	String[1]
boolean	Boolean
counter	Numeric
decimal	BigDecimal
double	Float
float	Float
inet	IPAddr
int	Numeric
list	Array
map	Hash
set	Set
text	String[1]
timestamp	Time
timeuuid	Cassandra::TimeUuid
uuid	Cassandra::Uuid
varchar	String[1]
varint	Numeric

Using strings

This example uses this schema:

[1] CQL Data types that map to `String` can have different encodings.

```
CREATE KEYSPACE simplex WITH replication = {'class': 'SimpleStrategy',
 'replication_factor': 3};
USE simplex;
CREATE TABLE mytable (
    a int PRIMARY KEY,
    b ascii,
    c blob,
    d text,
    e varchar,
);
INSERT INTO mytable (a, b, c, d, e)
VALUES (
    0,
    'ascii',
    0x626c6f62,
    'text',
    'varchar'
)
```

Running this code:

```
require 'cassandra'

cluster = Cassandra.cluster
session = cluster.connect("simplex")

row = session.execute("SELECT * FROM mytable").first

puts "Ascii: #{row['b']}"
puts "Blob: #{row['c']}"
puts "Text: #{row['d']}"
puts "Varchar: #{row['e']}"
```

Produces this output:

```
Ascii: ascii
Blob: blob
Text: text
Varchar: varchar
```

Using numbers

This example uses this schema:

```
CREATE KEYSPACE simplex WITH replication = {'class': 'SimpleStrategy',
 'replication_factor': 3};
USE simplex;
CREATE TABLE mytable (
    a int PRIMARY KEY,
    b bigint,
    c decimal,
    d double,
    e float,
    f int,
    g varint
);
INSERT INTO mytable (a, b, c, d, e, f, g)
VALUES (
    0,
    765438000,
```

```
    1313123123.234234234234234234123,
    3.141592653589793,
    3.14,
    4,
    67890656781923123918798273492834712837198237
)
```

Running this code:

```
require 'cassandra'

cluster = Cassandra.cluster
session = cluster.connect("simplex")

row = session.execute("SELECT * FROM mytable").first

puts "Bigint: #{row['b']}"
puts "Decimal: #{row['c']}"
puts "Double: #{row['d']}"
puts "Float: #{row['e']}"
puts "Integer: #{row['f']}"
puts "Varint: #{row['g']}"
```

Produces this output:

```
Bigint: 765438000
Decimal: 0.131312312323423423423423423423123E10
Double: 3.141592653589793
Float: 3.140000104904175
Integer: 4
Varint: 67890656781923123918798273492834712837198237
```

Using UUIDs, booleans, and IP addresses

This example uses this schema:

```
CREATE KEYSPACE simplex WITH replication = {'class': 'SimpleStrategy',
 'replication_factor': 3};
USE simplex;
CREATE TABLE mytable (
    a int PRIMARY KEY,
    b boolean,
    c inet,
    d timestamp,
    e timeuuid,
    f uuid
);
INSERT INTO mytable (a, b, c, d, e, f)
VALUES (
    0,
    true,
    '200.199.198.197',
    '2013-12-11 10:09:08+0000',
    FE2B4360-28C6-11E2-81C1-0800200C9A66,
    00b69180-d0e1-11e2-8b8b-0800200c9a66
)
```

Running this code:

```
require 'cassandra'

cluster = Cassandra.cluster
session = cluster.connect("simplex")

row = session.execute("SELECT * FROM mytable").first

puts "Boolean: #{row['b']}"
puts "Inet: #{row['c'].class.name} - #{row['c']}"
puts "Timestamp: #{row['d'].httpdate}"
puts "Timeuuid: #{row['e']}"
puts "Uuid: #{row['f']}"
```

Produces this output:

```
Boolean: true
Inet: IPAddr - 200.199.198.197
Timestamp: Wed, 11 Dec 2013 10:09:08 GMT
Timeuuid: fe2b4360-28c6-11e2-81c1-0800200c9a66
Uuid: 00b69180-d0e1-11e2-8b8b-0800200c9a66
```

Using lists, maps, and sets

This example uses this schema:

```
CREATE KEYSPACE simplex WITH replication = {'class': 'SimpleStrategy',
 'replication_factor': 3};
USE simplex;
CREATE TABLE user (
    id int PRIMARY KEY,
    logins List<timestamp>,
    locations Map<timestamp, double>,
    ip_addresses Set<inet>
);
INSERT INTO user (id, logins, locations, ip_addresses)
VALUES (
    0,
    ['2014-09-11 10:09:08+0000', '2014-09-12 10:09:00+0000'],
    {'2014-09-11 10:09:08+0000': 37.397357},
    {'200.199.198.197', '192.168.1.15'}
)
```

Running this code:

```
require 'cassandra'

cluster = Cassandra.cluster
session = cluster.connect("simplex")

row = session.execute("SELECT * FROM user").first

puts "Logins: #{row['logins'].map(&:httpdate)}"
puts "Location at #{row['locations'].first.first.httpdate}:
 #{row['locations'].first.last}"
puts "IP addresses: #{row['ip_addresses'].inspect}"
```

Produces output like this:

```
Logins: ["Thu, 11 Sep 2014 10:09:08 GMT", "Fri, 12 Sep 2014 10:09:00 GMT"]
```

```
Location at Thu, 11 Sep 2014 10:09:08 GMT: 37.397357
Ip Addresses: #<Set: {#<IPAddr: IPv4:192.168.1.15/255.255.255.255>, #<IPAddr:
 IPv4:200.199.198.197/255.255.255.255>}>
```

Using tuples

This example uses this schema:

```
CREATE KEYSPACE simplex WITH replication = {'class': 'SimpleStrategy',
 'replication_factor': 3};
USE simplex;
CREATE TABLE user (
    id int PRIMARY KEY,
    name frozen <tuple<varchar, varchar>>
);
INSERT INTO user (id, name)
VALUES (0, ('John', 'Smith'))
```

Running this code:

```
require 'cassandra'

cluster = Cassandra.cluster
session = cluster.connect("simplex")

row = session.execute("SELECT * FROM user").first

puts "Name: #{row['name']}"

update = session.prepare("UPDATE user SET name=? WHERE id=0")
session.execute(update, arguments: [Cassandra::Tuple.new('Jane', 'Doe')])

row = session.execute("SELECT * FROM user").first
puts "Name: #{row['name']}"

session.execute("INSERT INTO user (id, name) VALUES (1, (?, ?))", arguments:
 ['Agent', 'Smith'])
row = session.execute("SELECT * FROM user WHERE id=1").first
puts "Name: #{row['name']}"

insert = session.prepare("INSERT INTO user (id, name) VALUES (?, ?)")
session.execute(insert, arguments: [2, Cassandra::Tuple.new('Apache',
 'Cassandra')])
row = session.execute("SELECT * FROM user WHERE id=2").first

puts "Name: #{row['name']}"

begin
    session.execute(update, arguments: [Cassandra::Tuple.new('Jane', 'Doe',
 'Extra')])
    rescue => e
    puts "#{e.class.name}: #{e.message}"
end
```

Produces output like this:

```
Name: (John, Smith)
Name: (Jane, Doe)
Name: (Agent, Smith)
Name: (Apache, Cassandra)
```

```
ArgumentError: argument for "name" must be tuple<varchar, varchar>, (Jane,
 Doe, Extra) give
```

User-defined data types

The following example use this schema:

```
CREATE KEYSPACE simplex WITH replication = {'class': 'SimpleStrategy',
 'replication_factor': 3};
USE simplex;
CREATE TYPE address (street text, zipcode int);
CREATE TYPE check_in (
   location frozen <address>,
   time timestamp,
   data frozen <tuple<int, text, float>>
);
CREATE TABLE users (id int PRIMARY KEY, location frozen<address>);
```

Using UDTs with prepared statements

Running this code:

```
require 'cassandra'
cluster = Cassandra.cluster
session = cluster.connect('simplex')
insert  = session.prepare('INSERT INTO users (id, location) VALUES (?, ?)')
session.execute(insert, arguments: [0, Cassandra::UDT.new(street: '123 Main
 St.', zipcode: 78723)])
session.execute('SELECT * FROM users').each do |row|
    location = row['location']
    puts "Location: #{location}"
end
```

Produces this output:

```
ocation: { street: "123 Main St.", zipcode: 78723 }
```

Using UDTs with raw CQL

Running this code:

```
require 'cassandra'
cluster = Cassandra.cluster
session = cluster.connect('simplex')
session.execute("INSERT INTO users (id, location) VALUES (0, {street: '123
 Main St.', zipcode: 78723})")
session.execute('SELECT * FROM users').each do |row|
    location = row['location']
    puts "Location: #{location}"
end
```

Contains this output:

```
Location: { street: "123 Main St.", zipcode: 78723 }
```

Inspecting UDTs

Running this code:

```
require 'cassandra'
cluster = Cassandra.cluster
cluster.keyspace('simplex').each_type do |type|
    puts type.to_cql
end
```

Produces this output:

```
CREATE TYPE simplex.address (
    street varchar,
    zipcode int
);
CREATE TYPE simplex.check_in (
    location frozen <address>,
    time timestamp,
    data frozen <tuple<int, varchar, float>>
);
```

Inserting a partially-complete UDT

Running this code:

```
require 'cassandra'
cluster = Cassandra.cluster
session = cluster.connect('simplex')
insert  = session.prepare('INSERT INTO users (id, location) VALUES (?, ?)')
session.execute(insert, arguments: [0, Cassandra::UDT.new(zipcode: 78723)])
session.execute('SELECT * FROM users').each do |row|
    location = row['location']
    puts "Location: #{location}"
end
```

Produces this output:

```
Location: { street: nil, zipcode: 78723 }
```

Nesting a UDT

Running this code:

```
require 'cassandra'
cluster = Cassandra.cluster
session = cluster.connect('simplex')
session.execute("CREATE TABLE registration (id int PRIMARY KEY, info
 frozen<check_in>)")
insert  = session.prepare('INSERT INTO registration (id, info) VALUES
 (?, ?)')
location = Cassandra::UDT.new(street: '123 Main St.', zipcode: 78723)
tuple = Cassandra::Tuple.new(42, 'math', 3.14)
input = Cassandra::UDT.new(location: location, time: Time.at(1358013521,
 123000), data: tuple)
session.execute(insert, arguments: [0, input])
session.execute('SELECT * FROM registration').each do |row|
    info = row['info']
```

```
    puts "Info: {street: #{info.location.street}, zipcode:
 #{info.location.zipcode}}, #{info.time.httpdate}, #{info.data}"
end
```

Contains this output:

```
Info: {street: 123 Main St., zipcode: 78723}, Sat, 12 Jan 2013 17:58:41 GMT,
 (42, math, 3.140000104904175)
```

Debugging

You have several options to help in debugging your application.

Execution information

Every result contains useful execution information (`Cassandra::Execution::Info`).

This example extends the `SimpleClient` class in the Quick start section by adding a method.

```
def getExecutionInformation()
    execution = @session.execute("SELECT * FROM simplex.songs",
 consistency: :one).execution_info
    puts "coordinator: #{execution.hosts.last.ip}"
    puts "cql: #{execution.statement.cql}"
    puts "requested consistency: #{execution.options.consistency}"
    puts "actual consistency: #{execution.consistency}"
    puts "number of retries: #{execution.retries}"
end
```

Example output from calling the `getExecutionInformation` method.

```
coordinator: 127.0.0.3
cql: SELECT * FROM simplex.songs
requested consistency: one
actual consistency: one
number of retries: 0
```

Execution information reflects retry decision

Given a file, `retrying_at_a_given_consistency_policy.rb`, with:

```
class RetryingAtAGivenConsistencyPolicy
    include Cassandra::Retry::Policy

    def initialize(consistency_to_use)
        @consistency_to_use = consistency_to_use
    end

    def read_timeout(statement, consistency_level, required_responses,
                     received_responses, data_retrieved, retries)
        try_again(@consistency_to_use)
    end

    def write_timeout(statement, consistency_level, write_type,
                      acks_required, acks_received, retries)
        try_again(@consistency_to_use)
    end
```

```
    def unavailable(statement, consistency_level, replicas_required,
                    replicas_alive, retries)
        try_again(@consistency_to_use)
    end
end
```

And the following example:

```
require 'cassandra'
require_relative 'retrying_at_a_given_consistency_policy'

cluster   = Cassandra.cluster(retry_policy:
 RetryingAtAGivenConsistencyPolicy.new(:one))
session   = cluster.connect("simplex")
execution = session.execute("SELECT * FROM songs",
 consistency: :all).execution_info

puts "Requested consistency: #{execution.options.consistency}"
puts "Actual consistency: #{execution.consistency}"
puts "Number of retries: #{execution.retries}"
```

Output:

```
Requested consistency: all
Actual consistency: one
Number of retries: 1
```

Request tracing

This example uses the `simplex` schema from the Quick start section by adding a method.

By default, request tracing is disabled.

```
require 'cassandra'

cluster   = Cassandra.cluster
session   = cluster.connect("simplex")
execution = session.execute("SELECT * FROM songs").execution_info

at_exit { cluster.close }

if execution.trace
    puts "Failure."
else
    puts "Success."
end
```

Output:

```
Success.
```

Enabling tracing

Enable tracing by setting the :trace option to true when executing a statement. You retrieve trace information by calling execution_info method on your result.

```
require 'cassandra'
```

```
cluster   = Cassandra.cluster
session   = cluster.connect("simplex")
execution = session.execute("SELECT * FROM songs", :trace =>
 true).execution_info()
trace     = execution.trace

at_exit { cluster.close() }

puts "Coordinator: #{trace.coordinator}"
puts "Started at: #{trace.started_at}"
puts "Total events: #{trace.events.size}"
puts "Request: #{trace.request}"
```

Example output:

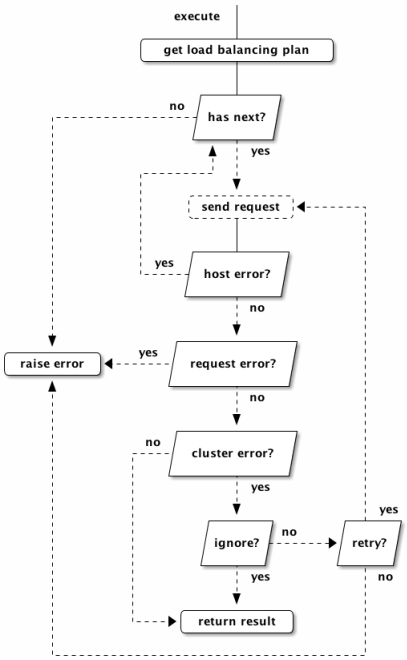
```
Coordinator: 127.0.0.3
Started at: 2014-11-03 07:44:20 -0800
Total events: 26
Request: Execute CQL3 query
```

Error handling

Handling errors in a distributed system is a complex and complicated topic. Ideally, you must understand all of the possible sources of failure and determine appropriate actions to take. This section is intended to explain various known failure modes.

Request execution errors

Requests resulting in host errors are automatically retried on other hosts. If no other hosts are present in the load-balancing plan, a `Cassandra::Errors::NoHostsAvailable` error is raised that contains a map of host to host error that were seen during the request.



Additionally, if an empty load-balancing plan is returned by the load-balancing policy, the request is not attempted on any hosts.

Whenever a cluster error occurs, the retry policy is used to decide whether to re-raise the error, retry the request at a different consistency, or ignore the error and return empty result.

Finally, all other request errors, such as validation errors, are returned to the application without retries.

Table 3: Top-level error classes

Event type	Class
Host errors	`Cassandra::Errors::ServerError`
	`Cassandra::Errors::OverloadedError`
	`Cassandra::Errors::InternalError`
	`Cassandra::Errors::IsBootstrappingError`
Cluster errors	`Cassandra::Errors::WriteTimeoutError`
	`Cassandra::Errors::ReadTimeoutError`
	`Cassandra::Errors::UnavailableError`
Rerquest errors	`Cassandra::Errors::ValidationError`
	`Cassandra::Errors::ClientError`
	`Cassandra::Errors::TruncateError`

Connection heartbeat

In addition to the request execution errors and timeouts, the driver performs a periodic heartbeat of each open connection to detect network outages and prevent stale connections from gathering.

The default heartbeat interval is quite conservative at 30 seconds with an idle timeout of one minute, but these numbers can be changed when constructing a cluster.

Upon detecting a stale connection, the driver automatically closes it and all outstanding requests fail with a host-level error, which forces the requests to be retried on other hosts as part of a normal request execution.

Example schema

```
CREATE KEYSPACE simplex
    WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 3};

USE simplex;

CREATE TABLE songs (
    id uuid PRIMARY KEY,
    title text,
    album text,
    artist text,
    tags set<text>,
    data blob
);

CREATE TABLE simplex.playlists (
    id uuid,
    title text,
    album text,
    artist text,
```

```
    song_id uuid,
    PRIMARY KEY (id, title, album, artist)
);

INSERT INTO songs (id, title, album, artist, tags)
VALUES (
    756716f7-2e54-4715-9f00-91dcbea6cf50,
    'La Petite Tonkinoise',
    'Bye Bye Blackbird',
    'Joséphine Baker',
    {'jazz', '2013'})
;

INSERT INTO songs (id, title, album, artist, tags)
VALUES (
    f6071e72-48ec-4fcb-bf3e-379c8a696488,
    'Die Mösch',
    'In Gold',
    'Willi Ostermann',
    {'kölsch', '1996', 'birds'}
);

INSERT INTO songs (id, title, album, artist, tags)
VALUES (
    fbdf82ed-0063-4796-9c7c-a3d4f47b4b25,
    'Memo From Turner',
    'Performance',
    'Mick Jager',
    {'soundtrack', '1991'}
);

INSERT INTO simplex.playlists (id, song_id, title, album, artist)
VALUES (
    2cc9ccb7-6221-4ccb-8387-f22b6a1b354d,
    756716f7-2e54-4715-9f00-91dcbea6cf50,
    'La Petite Tonkinoise',
    'Bye Bye Blackbird',
    'Joséphine Baker'
);

INSERT INTO simplex.playlists (id, song_id, title, album, artist)
VALUES (
    2cc9ccb7-6221-4ccb-8387-f22b6a1b354d,
    f6071e72-48ec-4fcb-bf3e-379c8a696488,
    'Die Mösch',
    'In Gold',
    'Willi Ostermann'
);

INSERT INTO simplex.playlists (id, song_id, title, album, artist)
VALUES (
    3fd2bedf-a8c8-455a-a462-0cd3a4353c54,
    fbdf82ed-0063-4796-9c7c-a3d4f47b4b25,
    'Memo From Turner',
    'Performance',
    'Mick Jager'
);
```

Node discovery

The Ruby driver automatically discovers and uses all of the nodes in a Cassandra cluster, including newly bootstrapped ones.

Description

The driver discovers the nodes that constitute a cluster by querying the contact points used in building the cluster object. After this it is up to the cluster's load balancing policy to keep track of node events (that is add, down, remove, or up) by its implementation of a listener.

Membership change listener example

A state listener which is notified on cluster membership changes.

`membership_change_printing_listener.rb` listing:

```
class MembershipChangePrintingListener
    def initialize(io)
        @out = io
    end

    def host_found(host)
        @out.puts("Host #{host.ip} is found")
        @out.flush
    end

    def host_lost(host)
        @out.puts("Host #{host.ip} is lost")
        @out.flush
    end

    def host_up(host)
        @out.puts("Host #{host.ip} is up")
        @out.flush
    end

    def host_down(host)
        @out.puts("Host #{host.ip} is down")
        @out.flush
    end
end
```

`membership_change_example.rb` listing:

```
require_relative 'membership_change_printing_listener'
require 'cassandra'

listener = MembershipChangePrintingListener.new($stderr)
cluster  = Cassandra.cluster(hosts: ['127.0.0.1'])

cluster.register(listener)

$stdout.puts("=== START ===")
$stdout.flush
$stdin.gets
$stdout.puts("=== STOP ===")
$stdout.flush
```

1. Run `membership_change_example.rb` against a three-node Cassandra cluster using the example schema.
2. Stop and restart a node.
3. Add a fourth node and start it.
4. Remove the fourth node.

Output:

```
$ ruby membership_change_example.rb
=== START ===
Host 127.0.0.2 is down
Host 127.0.0.2 is up
Host 127.0.0.4 is found
Host 127.0.0.4 is up
Host 127.0.0.4 is down
Host 127.0.0.4 is lost

=== STOP ===
```

Schema change listener example

A state listener which is notified on schema changes. There are three types of changes:

- `keyspace_created`
- `keyspace_changed`
- `keyspace_dropped`

All these changes are communicated to a state listener using its accordingly named methods with a Keyspace instance as an argument.

`schema_change_example.rb` listing:

```
require 'cassandra'
require_relative 'schema_change_printing_listener'

listener = SchemaChangePrintingListener.new($stderr)
cluster  = Cassandra.cluster(hosts: ['127.0.0.1'])

cluster.register(listener)

$stdout.puts("=== START ===")
$stdout.flush
$stdin.gets
$stdout.puts("=== STOP ===")
$stdout.flush
```

1. Run `schema_change_example.rb` against a three-node Cassandra cluster.
2. Create the simplex keyspace from the example schema.
3. Drop the simplex schema from the cluster.

Output:

```
$ ruby membership_change_example.rb
=== START ===
Keyspace "simplex" created
Keyspace "simplex" changed
Keyspace "simplex" changed
Keyspace "simplex" dropped
```

```
=== STOP ===
```

Prepared statements

Prepared statements are used to prepare a write query only once and execute it multiple times with different values. A bind variable marker (that is, ?) is used to represent a dynamic value in the statement.

An INSERT prepared statement

This example uses this schema:

```
CREATE KEYSPACE simplex WITH replication = {'class': 'SimpleStrategy',
 'replication_factor': 3};
USE simplex;
CREATE TABLE playlists (
    id uuid,
    title text,
    album text,
    artist text,
    song_id uuid,
    PRIMARY KEY (id, title, album, artist)
);
```

Running this code:

```
require 'cassandra'

cluster = Cassandra.cluster
session = cluster.connect("simplex")
insert  = session.prepare(
          "INSERT INTO playlists (id, song_id, title, artist, album) " \
          "VALUES (62c36092-82a1-3a00-93d1-46196ee77204, ?, ?, ?, ?)"
        )

songs = [
    {
      :id     =>
 Cassandra::Uuid.new('756716f7-2e54-4715-9f00-91dcbea6cf50'),
      :title  => 'La Petite Tonkinoise',
      :album  => 'Bye Bye Blackbird',
      :artist => 'Joséphine Baker'
    },
    {
      :id     => Cassandra::Uuid.new('f6071e72-48ec-4fcb-
bf3e-379c8a696488'),
      :title  => 'Die Mösch',
      :album  => 'In Gold',
      :artist => 'Willi Ostermann'
    },
    {
      :id     => Cassandra::Uuid.new('fbdf82ed-0063-4796-9c7c-
a3d4f47b4b25'),
      :title  => 'Memo From Turner',
      :album  => 'Performance',
      :artist => 'Mick Jager'
    }
]
songs.each do |song|
```

```
    session.execute(insert, arguments: [song[:id], song[:title],
 song[:artist], song[:album]])
end

session.execute("SELECT * FROM playlists").each do |row|
    puts("#{row["artist"]}: #{row["title"]} / #{row["album"]}")
end
```

Contains this output:

```
Joséphine Baker: La Petite Tonkinoise / Bye Bye Blackbird
```

An INSERT prepared statement with named parameters

This example uses this schema:

```
CREATE KEYSPACE simplex WITH replication = {'class': 'SimpleStrategy',
 'replication_factor': 3};
USE simplex;
CREATE TABLE playlists (
    id uuid,
    title text,
    album text,
    artist text,
    song_id uuid,
    PRIMARY KEY (id, title, album, artist)
);
```

Running this code:

```
require 'cassandra'

cluster = Cassandra.cluster
session = cluster.connect("simplex")
insert  = session.prepare(
          "INSERT INTO playlists (id, song_id, title, artist, album) " \
          "VALUES (62c36092-82a1-3a00-93d1-46196ee77204, :a, :b, :c, :d)"
        )

songs = [
    {
      :id     =>
 Cassandra::Uuid.new('756716f7-2e54-4715-9f00-91dcbea6cf50'),
      :title  => 'La Petite Tonkinoise',
      :album  => 'Bye Bye Blackbird',
      :artist => 'Joséphine Baker'
    },
    {
      :id     => Cassandra::Uuid.new('f6071e72-48ec-4fcb-
bf3e-379c8a696488'),
      :title  => 'Die Mösch',
      :album  => 'In Gold',
      :artist => 'Willi Ostermann'
    },
    {
      :id     => Cassandra::Uuid.new('fbdf82ed-0063-4796-9c7c-
a3d4f47b4b25'),
      :title  => 'Memo From Turner',
      :album  => 'Performance',
      :artist => 'Mick Jager'
    }
```

```
]

songs.each do |song|
    session.execute(insert, arguments: {:a => song[:id], :b =>
 song[:title], :c => song[:artist], :d => song[:album]})
end

session.execute("SELECT * FROM playlists").each do |row|
    puts("#{row["artist"]}: #{row["title"]} / #{row["album"]}")
end
```

Contains this output:

```
Joséphine Baker: La Petite Tonkinoise / Bye Bye Blackbird
Willi Ostermann: Die Mösch / In Gold
Mick Jager: Memo From Turner / Performance
```

A SELECT prepared statement with parameterized LIMIT clause

This example uses this schema:

```
CREATE KEYSPACE simplex WITH replication = {'class': 'SimpleStrategy',
 'replication_factor': 3};
USE simplex;
CREATE TABLE playlists (
    id uuid,
    title text,
    album text,
    artist text,
    song_id uuid,
    PRIMARY KEY (id, title, album, artist)
);
INSERT INTO playlists (id, song_id, title, album, artist)
VALUES (
    2cc9ccb7-6221-4ccb-8387-f22b6a1b354d,
    756716f7-2e54-4715-9f00-91dcbea6cf50,
    'La Petite Tonkinoise',
    'Bye Bye Blackbird',
    'Joséphine Baker'
);
INSERT INTO playlists (id, song_id, title, album, artist)
VALUES (
    2cc9ccb7-6221-4ccb-8387-f22b6a1b354d,
    f6071e72-48ec-4fcb-bf3e-379c8a696488,
    'Die Mösch',
    'In Gold',
    'Willi Ostermann'
);
INSERT INTO playlists (id, song_id, title, album, artist)
VALUES (
    3fd2bedf-a8c8-455a-a462-0cd3a4353c54,
    fbdf82ed-0063-4796-9c7c-a3d4f47b4b25,
    'Memo From Turner',
    'Performance',
    'Mick Jager'
);
INSERT INTO playlists (id, song_id, title, album, artist)
VALUES (
    3fd2bedf-a8c8-455a-a462-0cd3a4353c54,
    756716f7-2e54-4715-9f00-91dcbea6cf50,
```

```
    'La Petite Tonkinoise',
    'Bye Bye Blackbird',
    'Joséphine Baker'
);
```

Running this code:

```
require 'cassandra'

cluster = Cassandra.cluster
session = cluster.connect("simplex")
select  = session.prepare("SELECT * FROM playlists LIMIT ?")

limits  = [1, 2, 3]

limits.each do |limit|
    rows = session.execute(select, arguments: [limit])
    puts "selected #{rows.size} row(s)"
end
```

Contains this output:

```
selected 1 row(s)
selected 2 row(s)
selected 3 row(s)
```

Security

Authentication

By default, the driver supports Cassandra's internal authentication mechanism. It is also possible to provide a custom authenticator implementation, refer to `Cassandra::Auth` module for more information.

SSL encryption

Cassandra supports client-to-node encryption and even trusted clients, starting with version 1.2.3.

Setting up SSL on Cassandra

Setting up SSL encryption on Cassandra seems a difficult task for someone unfamiliar with the process.

1. Install the Java Cryptography Extension.
2. Set up the Cassandra keystore.
3. Extract the server certificate for peer verification.
4. Enable SSL authentication and trusted clients.

Installing the Java Cryptography extension

First you must install the Java Cryptography Extension (JCE). (Skip this step if you already have the JCE installed.) Without the JCE, Cassandra processes will fail to start after enabling encryption.

1. Download the JCE from Oracle.
2. Extract the files from the downloaded archive.
3. Copy `local_policy.jar` and `US_export_policy.jar` to the `$JAVA_HOME/jre/lib/security` directory.

If your `$JAVA_HOME` envirronmental variable is not set, you can determine the installation location of Java as follows:

Linux

Set the `$JAVA_HOME` variable by executing the following command.

```
JAVA_HOME = $(readlink -f /usr/bin/javac | sed  "s:/bin/javac::" )
```

Mac OS X

Set the `$JAVA_HOME` by running the `java_home` utility.

```
JAVA_HOME = $(/usr/libexec/java_home )
```

Setting up the Cassandra keystore

1. Create a server certificate by running the following script.

```
server_alias=node1
keystore_file=conf/.keystore
keystore_pass="some very long and secure password"

CN="Cassandra Node 1"
OU="Drivers and Tools"
O="DataStax Inc."
L="Santa Clara"
ST="California"
C="US"

keytool -genkey -keyalg RSA -noprompt \
    -alias "$server_alias" \
    -keystore "$keystore_file" \
    -storepass "$keystore_pass" \
    -dname "CN=$CN, OU=$OU, O=$O, L=$L, ST=$ST, C=$C"
```

   You should use a different certificate for every node that you want to secure communication with using SSL encryption.
2. After you have run this script for all nodes in our cluster, configure the Cassandra servers to use their respective `.keystore` files by adding the following to the `cassandra.yaml` file:

```
client_encryption_options:
    enabled: true
    keystore: conf/.keystore
    keystore_password: "some very long and secure password"
```

3. Note that the values of *keystore* and *keystore_password* above must be the same as the values of *$keystore_file* and *$keystore_pass* used in the shell script.
4. Restart your Cassandra processes.

At this point you already have SSL enabled and can even connect to the servers using the Ruby driver:

```
require 'cassandra'

cluster = Cassandra.connect(ssl: true)
```

Note: This is like having no security at all since the driver won't be able to verify the identity of the server.

Extracting the server certificate for peer verification

There are several ways to have your client verify the server's identity. In this example, you extract a PEM certificate of the server, which is suitable for use with the OpenSSL library that the driver uses, and give it to the client for verification.

1. Export a DER certificate of the server:

```
# values same as above
server_alias=node1
keystore_file=conf/.keystore
keystore_pass="some very long and secure password"

keytool -export \
-alias "$server_alias" \
-keystore "$keystore_file" \
-storepass "$keystore_pass" \
-file "$server_alias.der"
```

   Note: The values of *$server_alias*, *$keystore_file*, and *$keystore_pass* must be the same as in the script that we used to generate the keystore file.
2. Now that we have our DER certificate, Run `openssl` to transform the DER certificate created in the previous step into a PEM file:

```
$ openssl x509 -out "$server_alias.pem" -outform pem -in "$server_alias.der"
 -inform der
```

   This creates a PEM certificate out of your DER source certificate that was extracted from the keystore. This process has to be repeated for each unique keystore that was created previously.
3. Bundle all the PEM certificates for your client to use.

```
$ cat node1.pem node2.pem node3.pem > server.pem
```

   Note: You can skip this step if you only have one node in your cluster.
4. You specify the combined PEM file when connecting to the cluster in your client.
5. give it to the client to verify our server's identity:

```
require 'cassandra'

cluster = Cassandra.cluster(server_cert: 'path_to/server.pem')
```

   Your client can now verify the identity of the server, and, combined with Standard Authentication, this provides enough security to be useful.

Enabling SSL authentication and trusted clients

Even better than the previous example using PEM certificates to identify the server is to enable SSL authentication and trusted clients. Enabling SSL authentication means explicitly adding certificates of all clients and peers to a list of trusted certificates of each Cassandra server.

1. Be sure that all of your server nodes can talk to each other using SSL authentication by running the following Ruby script:

```
servers = [
    {
        :alias          => 'node1',
        :keystore       => 'node1/conf/.keystore',
        :kestore_pass   => "some very long and secure password",
```

```
        :truststore      => 'node1/conf/.truststore',
        :truststore_pass => "another very long and secure password"
    },
    {
        :alias           => 'node2',
        :keystore        => 'node2/conf/.keystore',
        :kestore_pass    => "some very long and secure password",
        :truststore      => 'node2/conf/.truststore',
        :truststore_pass => "another very long and secure password"
    },
    {
        :alias           => 'node3',
        :keystore        => 'node3/conf/.keystore',
        :kestore_pass    => "some very long and secure password",
        :truststore      => 'node3/conf/.truststore',
        :truststore_pass => "another very long and secure password"
    },
]

# we'll iterate over each server and add all other server certificates it
 its truststore
servers.each do |server|
    truststore = server[:truststore]
    storepass  = server[:truststore_pass]

    servers.each do |peer|
        next if peer == server # skip self

        peer_alias     = peer[:alias]
        peer_keystore  = peer[:keystore]
        peer_storepass = peer[:keystore_pass]

        # export .der certificate from this peer's keystore if we haven't
 already
        unless File.exists?("#{peer_alias}.der")
            system("keytool -export -alias \"#{peer_alias}\" -
keystore \"#{peer_keystore}\" -storepass \"#{peer_storepass}\" -file
 \"#{peer_alias}.der\"")
        end

        # now we can import extracted peer's DER certificate into server's
 truststore
        system("keytool -import -noprompt -alias \"#{peer_alias}\"
 -keystore \"#{truststore}\" -storepass \"#{storepass}\" -file
 \"#{peer_alias}.der\"")
    end
end
```

Be sure that all the data in the above script is correct: paths to keystores and truststores as well as passwords and aliases.

2. Save this file to `generate_truststores.rb` and run it:

```
ruby generate_truststores.rb
```

All your servers trusting each other.

3. Create a certificate for our client and add it to the servers' truststores.

   a. Create a new keystore for the Driver to use:

```
driver_alias=driver
keystore_file=driver.keystore
keystore_pass="some very long and secure password"
```

```
CN="Ruby Driver"
OU="Drivers and Tools"
O="DataStax Inc."
L="Santa Clara"
ST="California"
C="US"

keytool -genkey -keyalg RSA -noprompt \
    -alias "$driver_alias" \
    -keystore "$keystore_file" \
    -storepass "$keystore_pass" \
    -dname "CN=$CN, OU=$OU, O=$O, L=$L, ST=$ST, C=$C"
```

Note: Be sure to change the data in this example script.

b. Check that the `driver.keystore` file (or whatever the value of *$keystore_file* was) exists.

c. Export its DER certificate:

```
# values same as above
driver_alias=driver
keystore_file=driver.keystore
keystore_pass="some very long and secure password"

keytool -export \
    -alias "$driver_alias" \
    -keystore "$keystore_file" \
    -storepass "$keystore_pass" \
    -file "$server_alias.der"
```

d. Add the driver DER certificate to the truststores of our servers by using the following Ruby script:

```
servers = [
    {
        :alias           => 'node1',
        :keystore        => 'node1/conf/.keystore',
        :kestore_pass    => "some very long and secure password",
        :truststore      => 'node1/conf/.truststore',
        :truststore_pass => "another very long and secure password"
    },
    {
        :alias           => 'node2',
        :keystore        => 'node2/conf/.keystore',
        :kestore_pass    => "some very long and secure password",
        :truststore      => 'node2/conf/.truststore',
        :truststore_pass => "another very long and secure password"
    },
    {
        :alias           => 'node3',
        :keystore        => 'node3/conf/.keystore',
        :kestore_pass    => "some very long and secure password",
        :truststore      => 'node3/conf/.truststore',
        :truststore_pass => "another very long and secure password"
    }
]

driver_alias = "driver"
driver_cert  = "driver.pem"

servers.each do |server|
    truststore = server[:truststore]
```

```
    storepass  = server[:truststore_pass]

    system("keytool -import -noprompt -alias \"#{driver_alias}\"
  -keystore \"#{truststore}\" -storepass \"#{storepass}\" -file
  \"#{driver_cert}\"")
 end
```

e. Save it to a file called `add_to_truststores.rb`.

f. Run the `add_to_truststores.rb` script.

```
$ ruby add_to_truststores.rb
```

g. Run the `openssl` utility to convert the exported DER certificate to a PEM one:

```
$ openssl x509 -in driver.der -inform der -out driver.pem -outform pem
```

At this point you have your client PEM key, or a public key,. to communicate securely with.

4. Create a private key, to be used to encrypt all communication, by running the following Java program to extract this information from `driver.keystore` file:

```
/* DumpKey.java
 * Copyright (c) 2007 by Dr. Herong Yang, http://www.herongyang.com/
 */
import java.io.*;
import java.security.*;

public class DumpKey {
    static public void main(String[] a) {
        if (a.length<5) {
            System.out.println("Usage:");
            System.out.println(
            "java DumpKey jks storepass alias keypass out");
            return;
        }
        String jksFile = a[0];
        char[] jksPass = a[1].toCharArray();
        String keyName = a[2];
        char[] keyPass = a[3].toCharArray();
        String outFile = a[4];

        try {
            KeyStore jks = KeyStore.getInstance("jks");
            jks.load(new FileInputStream(jksFile), jksPass);
            Key key = jks.getKey(keyName, keyPass);
            System.out.println("Key algorithm: "+key.getAlgorithm());
            System.out.println("Key format: "+key.getFormat());
            System.out.println("Writing key in binary form to "
            +outFile);

            FileOutputStream out = new FileOutputStream(outFile);
            out.write(key.getEncoded());
            out.close();
        } catch (Exception e) {
            e.printStackTrace();
            return;
        }
    }
}
```

a. Save the Java code to a file called `DumpKey.java`.

b. Run the following script:

```
# values same as above
driver_alias=driver
keystore_file=driver.keystore
keystore_pass="some very long and secure password"

javac DumpKey.java
java DumpKey "$keystore_file" "$keystore_pass" "$driver_alias"
 "$keystore_pass" driver_bin.key
```

This should drop the contents of `driver.keystore` private key into `driver_bin.key`.

c. Add the PEM standard header and footer by running the following script:

```
# prepend PEM header
echo "-----BEGIN PRIVATE KEY-----" | cat - driver_bin.key > driver.key
# append PEM footer
echo "-----END PRIVATE KEY-----" >> driver.key
```

d. Set up a passphrase for your private key to ensure that only our client application can use it:

```
$ ssh-keygen -p -f driver.key
```

The script prompts you to enter a secure passphrase for the key which you'll use below.

5. Enable SSL authentication by adding the following to `cassandra.yaml` on each server:

```
client_encryption_options:
    enabled: true
    keystore: conf/.keystore
    keystore_password: "some very long and secure password"
    require_client_auth: true
    truststore: conf/.truststore
    truststore_password: "another very long and secure password"
```

Be sure to update the `cassandra.yaml` file on each server with correct data from previous steps.

6. Use the client certificate and key to connect to your Cassandra cluster:

```
cluster = Cassandra.connect(
    server_cert:  '/path/to/server.pem',
    client_cert:  '/path/to/driver.pem',
    private_key:  '/path/to/driver.key',
    passphrase:   'the passphrase you picked for the key'
)
```

API reference

DataStax Ruby Driver for Apache Cassandra.

Tips for using DataStax documentation

Navigating the documents

To navigate, use the table of contents or search in the left navigation bar. Additional controls are:

	Hide or display the left navigation.
« »	Go back or forward through the topics as listed in the table of contents.
	Toggle highlighting of search terms.
	Print page.
	See doc tweets and provide feedback.
	Grab to adjust the size of the navigation pane.
¶	Appears on headings for bookmarking. Right-click the ¶ to get the link.
	Toggles the legend for CQL statements and nodetool options.

Other resources

You can find more information and help at:

- Documentation home page
- Datasheets
- Webinars
- Whitepapers
- Developer blogs
- Support