DATASTAX

C/C++ Driver 2.0 for Apache Cassandra

Documentation

${ds.localized.time}

Contents

About the C/C++ driver

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

The C/C++ 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 in 2.0

Here are the new and noteworthy features of the 2.0 driver.

- Latency-aware routing
- Performance metrics
- Removal of `CassString`, `CassBytes`, and `CassDecimal` types
- Removal of Boost library dependency

Architecture

An overview of the C/C++ architecture.

The C/C++ Driver 2.0 for Apache Cassandra works exclusively with the Cassandra Query Language (CQL) version 3 and Cassandra's binary protocol which was introduced in Cassandra version 1.2.

Architectural overview

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.
- Cassandra trace handling: tracing can be set on a per-query basis and the driver provides a convenient API to retrieve the trace.
- 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).
- Configurable retry policy: a retry policy 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.
- Convenient schema access: the driver exposes a Cassandra schema in a usable way.
- Node discovery: the driver automatically discovers and uses all nodes of the Cassandra cluster, including newly bootstrapped ones.
- Paging: both automatic and manual.
- SSL support
- 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 C/C++ 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 driver is compatible with any Cassandra version from 1.2. The driver uses native protocol 1 (for Cassandra 1.2) and 2 (Cassandra 2.0+).

Building the driver

Supported operating systems

- Mac OS X 10.8 (Mavericks) and 10.9 (Yosemite)
- Windows 7 SP1
- CentOS/RHEL 5, 6, and 7
- Ubuntu 12.04 and14.04 LTS

Build dependencies

- **Libraries**
  - libuv 1.x or 0.10.x
  - OpenSSL (optional)
- **Tools**
  - CMake
  - GCC 4.1.2+, Clang 3.4+, or MSVC 2010, 2012, or 2013

Note: Utilizing the default package manager configuration to install dependencies on Unix-based operating systems may result in older versions of dependencies being installed.

Testing dependencies

- Boost 1.55+
- libssh2

Build dependencies under Linux

The driver has been built using both Clang (Ubuntu 12.04 or 14.04 LTS and Mac OS X) and GCC 4.1.2+ (Linux).

1. (Optional CentOS or RHEL 5.

   CentOS and RHEL 5 do not contain `git` in its repositories; however RepoForge (formerly RPMforge) has a RPM for this dependency.

   a. Download the appropriate RepoForge release package:

      - 32-bit
      - 64-bit

   b. Install key and RPM package:

```
$ sudo rpm --import http://apt.sw.be/RPM-GPG-KEY.dag.txt
$ sudo rpm -i rpmforge-release-0.5.3-1.el5.rf.*.rpm
```

2. Install dependencies and `libuv` library:

```
$ sudo yum install automake cmake gcc-c++ git libtool openssl-devel wget
$ pushd /tmp
$ wget http://libuv.org/dist/v1.4.2/libuv-v1.4.2.tar.gz
$ tar xzf libuv-v1.4.2.tar.gz
$ pushd libuv-v1.4.2
$ sh autogen.sh
$ ./configure
$ sudo make install
$ popd
$ popd
```

Additional dependencies under Ubuntu 12.04

Ubuntu 12.04 does not contain `libuv` in its repositories; however the LinuxJedi PPA has a backport from Ubuntu 14.04.

```
$ sudo apt-add-repository ppa:linuxjedi/ppa
$ sudo apt-get update
```

GCC

```
$ sudo apt-get install g++ make cmake libuv-dev libssl-dev
```

Clang

```
$ sudo apt-get install clang make cmake libuv-dev libssl-dev
```

Build dependencies under Mac OS X

The driver has been built and tested using the Clang compiler provided by XCode 5.1+. The dependencies were obtained using Homebrew.

```
$ brew install libuv cmake
```

Note: The driver utilizes the OpenSSL library included with XCode.

Building the driver under Linux or Mac OS X

```
$ git clone https://github.com/datastax/cpp-driver.git
$ mkdir cpp-driver/build
$ cd cpp-driver/build
$ cmake ..
$ make
```

Building the driver under Windows

The driver has been built and tested using Microsoft Visual Studio 2010, 2012 and 2013 (using both the Express and Professional versions) and Windows SDK 7.1, 8.0, and 8.1 on Windows 7 SP1. The library dependencies are automatically downloaded and built, however the following build dependencies need to be installed.

1. Download and install CMake.

   Make sure to select the option **Add CMake to the system PATH for all users** or **Add CMake to the system PATH for current user**.
2. Download and install git.

   Make sure to select the option **Use Git from Windows Command Prompt** or manually add the `git` executable to the system PATH.
3. Download and install ActiveState Perl.

   Make sure to select the option **Add Perl to PATH environment variable**.

   Note: This build dependency is required if building with OpenSSL support.
4. Download and install Python 2.7.x.

   Make sure to select and install the feature **Add python.exe to Path**.
5. A batch script has been created to detect installed versions of Visual Studio (and/or Windows SDK installations) to simplify the build process on Windows. If you have more than one version of Visual Studio (and/or Windows SDK) installed you will be prompted to select which version to use when compiling the driver.

   First you will need to open a **Command Prompt** (or Windows SDK Command Prompt) to execute the `vc_build.bat` batch script.

   ```
   Usage: VC_BUILD.BAT [OPTION...]

       --DEBUG                          Enable debug build
   ```

```
    --RELEASE                          Enable release build (default)
    --DISABLE-CLEAN                    Disable clean build
    --TARGET-COMPILER [version]        120, 110, 100, or WINSDK
    --DISABLE-OPENSSL                  Disable OpenSSL support
    --ENABLE-EXAMPLES                  Enable example builds
    --ENABLE-PACKAGES [version]        Enable package generation (*)
    --ENABLE-TESTS [boost-root-dir]    Enable test builds
    --ENABLE-ZLIB                      Enable zlib
    --GENERATE-SOLUTION                Generate Visual Studio solution (**)
    --INSTALL-DIR [install-dir]        Override installation directory
    --SHARED                           Build shared library (default)
    --STATIC                           Build static library
    --X86                              Target 32-bit build (***)
    --X64                              Target 64-bit build (***)
    --USE-BOOST-ATOMIC                 Use Boost atomic

    --HELP                             Display this message

*   Packages are only generated using detected installations of Visual
 Studio
**  Dependencies are built before generation of Visual Studio solution
*** Default target architecture is determined based on system architecture
```

To use the `vc_build.bat` script for easy inclusion into a project:

```
dos VC_BUILD.BAT --TARGET-COMPILER 120 --INSTALL-DIR
        C:\myproject\dependencies\libs\cpp-driver
```

Removal of Boost library dependency

Boost was an internal dependency in previous versions of the driver. This likely wasn't even known to most users of the driver because the required subset of the Boost source code was included in the driver itself. However, for some users this caused serious build issues because their applications also had a dependency on an older, but incompatible version of Boost. Instead of making Boost a required external dependency we've instead removed our dependency on Boost. There is still one external, but optional Boost dependency.

Before version 2.0, the driver depended on Boost's implementation for lock-free and atomic operations. Moving forwarded the driver will allow for three choices for the implementation of atomic operations: `std::atomic`, a built-in intrinsics implementation and Boost Atomic. The built-in implementation works on all supported platforms, but can be slower than using the optimized versions found in std::atomic or Boost Atomic. On compilers that support the C++11 memory model the `std::atomic` implementation is automatically enabled, however; on older compilers and systems, to achieve maximum performance, it is recommended that the Boost Atomic implementation be used.

The std::atomic implementation can also be explicitly enabled:

```
cmake -DCASS_USE_STD_ATOMIC ...
```

When Boost is available (on older per-C++11 compilers) its Atomics library can be used:

```
cmake -DCASS_USE_BOOST_ATOMIC ...
```

Note: Both of these implementations are entirely optional and the driver will work and perform well without them, but to achieve maximum performance they're recommended.

Architecture

An overview of the C/C++ architecture.

Cluster

The `CassCluster` object represents your cluster's configuration. The default cluster object is good for most clusters and only a list of contact points needs to be configured. Once a session is connected using a cluster object its configuration is constant. Modify the cluster object configuration after connecting a session doesn't change the session's configuration.

Session

A session object is used to execute queries. Internally, it also manages a pool of client connections to the cluster and uses a load-balancing policy to distribute requests across those connections. Your application should only create a single session object per keyspace because a session object is designed to be created once, reused, and shared by multiple application threads. The throughput of a session can be scaled by increasing the number of I/O threads. An I/O thread is used to handle reading and writing query request data to and from Cassandra. The number of I/O threads defaults to one per CPU core, but it can be configured using `cass_cluster_set_num_threads_io()`. It's generally better to create a single session with more I/O threads than multiple sessions with a smaller number of I/O threads.

Thread safety

`CassSession` is designed to be used concurrently from multiple threads. It is best practice to create a single session per keyspace. `CassFuture` is also thread safe. Other than these exceptions, in general, functions that modify an object's state are *not* thread safe. Objects that are immutable (marked `const`) can be read safely by multiple threads. None of the `free` functions can be called concurrently on the same instance of an object.

Memory handling

Values such as `CassString` and `CassBytes` point to memory held by the result object. The lifetimes of those values are valid as long as the result object is not freed. These values need to be copied into application memory if they need to live longer than the result object's lifetime. Primitive types such as `cass_int32_t` are copied by the driver because it can be done cheaply without incurring extra allocations.

Moving an iterator to the next value invalidates the value it previously returned.

Writing your first client

This section walks you through creating a client application that uses the C/C++ driver to connect to a Cassandra cluster, create a schema, load some data, and execute some queries.

Connecting to a Cassandra cluster

The C/C++ driver provides functions for connecting to a Cassandra cluster.

Before you begin

This tutorial uses the following software:

- An installed and running Cassandra or DSE cluster
- DataStax C/C++ driver
- A C/C++ compiler (e.g., gcc)
- A build tool (e.g., make)

About this task

In this tutorial, you create a client application project consisting of four files:

- `SimpleClient.hpp` (the header file for the SimpleClient class)
- `SimpleClient.cpp` (the implementation for the SimpleClient class)
- `simple.cpp` (a C++ program that has a main function)
- `Makefile` (the make file that contains the directives to build the simple application)

Procedure

1. Create a directory to contain your project.

```
$ mkdir simple
$ cd simple
```

2. Using a text editor, create the `SimpleClient.hpp` file.
   a) Add include directives for the driver and standard string header files.

```
#include "cassandra.h"
#include <string>
```

   b) After the include directives, declare a `SimpleClient` class within an `examples` namespace:

```
namespace example
{

class SimpleClient
{

} // end class
} // end namespace
```

   c) Add two private member fields to the class to store references to a cluster and a session object.

```
class SimpleClient
```

```
{
private:
    CassCluster* cluster;
    CassSession* session;
} // end class
```

d) Now add three public member functions: an inline getter for the session object, a `connect()`, and a `close()` function.

```
class SimpleClient
{
private:
    CassCluster* cluster;
    CassSession* session;
public:
    inline CassSession* getSession() { return session; }
    CassError connect(const std::string nodes);
    void close();
} // end class
```

3. Using a text editor, create the `SimpleClient.cpp` file.
4. Implement the `SimpleClient::connect()` function in the `SimpleClient.cpp` file.

a) Add include directives for the `iostream` and the `SimpleClient` header files.

```
#include <iostream>
#include "SimpleClient"
```

b) Declare the `examples` namespace.

```
namespace example {

} // end namespace
```

c) Declare the `connect()` function.

```
namespace example
{

CassError SimpleClient::connect(const string nodes)
{
}
} // end namespace
```

d) Add code to return the `CassError` code to the calling function and log some helpful information:

```
CassError SimpleClient::connect(const string nodes)
{
   CassError rc = CASS_OK;
   cout << "Connecting to " << nodes << "\n";

   return rc;
}
```

e) Create new cluster and session objects and store references to them in the class' appropriate private member fields.

```
CassError SimpleClient::connect(const string nodes)
{
    CassError rc = CASS_OK;
    cout << "Connecting to " << nodes << "\n";

    cluster = cass_cluster_new();
    session = cass_session_new();

    return rc;
}
```

You only need one cluster and one session object per client application. The driver library takes care of managing connections and providing load balancing.

f) Set the contact points (of one or more of the nodes in your cluster) and connect to the cluster.

```
CassError SimpleClient::connect(const string nodes)
{
    // earlier code elided
    CassFuture* connect_future = NULL;
    cass_cluster_set_contact_points(cluster, "127.0.0.1");
    connect_future = cass_session_connect(session, cluster);
    cass_future_wait(connect_future);

    return rc;
}

} // end namespace
```

The `cass_session_connect()` function returns a future object. In this example, we merely wait for the asynchronous call to return, but in your code, you could do other tasks while waiting for a callback from the future.

g) Retrieve the error code and log information appropriate to its value.

```
CassError SimpleClient::connect(const string nodes)
{
    // earlier code elided
    rc = cass_future_error_code(connect_future);
    if ( rc == CASS_OK )
    {
        cout << "Connected." << "\n";
    }
    else
    {
        return printError(rc);
    }
    return rc;
}

} // end namespace
```

h) Free up the future object and return the resulting error code.

```
CassError SimpleClient::connect(const string nodes)
{
    // earlier code elided
    cass_future_free(connect_future);

    return rc;
}
```

```
} // end namespace
```

5. Implement the `SimpleClient::close()` function in the `SimpleClient.cpp` file.

```
void SimpleClient::close()
{
    cout << "Closing down cluster connection." << "\n";
    cass_session_close(session);
    cass_cluster_free(cluster);
}
```

Before exiting your client application, close down the session you created and free up the memory allocated for the cluster object.

Code listing

`SimpleClient.hpp`

```
#ifndef __SIMPLE_CLIENT_H__
#define __SIMPLE_CLIENT_H__

#include "cassandra.h"
#include <string>

namespace example
{

class SimpleClient
{
private:
    CassCluster* cluster;
    CassSession* session;
public:
    inline CassSession* getSession() { return session; }
    CassError connect(const std::string nodes);
    void close();
} // end class

} // end namespace

#endif
```

Code listing

`SimpleClient.cpp`

```
#include <iostream>
#include "SimpleClient.hpp"

namespace example {
using namespace std;

CassError SimpleClient::connect(const string nodes)
{
    CassError rc = CASS_OK;
    cout << "Connecting to " << nodes << "\n";

    cluster = cass_cluster_new();
```

```
    session = cass_session_new();

    CassFuture* connect_future = NULL;
    cass_cluster_set_contact_points(cluster, "127.0.0.1");
    connect_future = cass_session_connect(session, cluster);
    cass_future_wait(connect_future);

    rc = cass_future_error_code(connect_future);
    if ( rc == CASS_OK )
    {
        cout << "Connected." << "\n";
    }
    else
    {
        return printError(rc);
    }
    cass_future_free(connect_future);
    return rc;
}

void SimpleClient::close()
{
    cout << "Closing down cluster connection." << "\n";
    cass_session_close(session);
    cass_cluster_free(cluster);
}

} // end namespace
```

Code listing

simple.cpp

```
#include <iostream>

#include "SimpleClient.hpp"

int main(int argc, char**)
{
    using example::SimpleClient;

    SimpleClient client;
    client.connect("127.0.0.1");
    return 0;
}
```

Code listing

Makefile

```
OBJS = simple.o bound.o SimpleClient.o BoundStatementsClient.o

CC = g++
DRIVER_DIR = path-to/cpp-driver

DEBUG = -g
CPPFLAGS = -Wall -c $(DEBUG) -I$(DRIVER_DIR)/include
LFLAGS = -Wall $(DEBUG)
```

```
LIBS = -L$(DRIVER_DIR) -luv -lssh2 -lcassandra

all: simple

simple: SimpleClient.o simple.o
 $(CC) $(LFLAGS) -o simple SimpleClient.o simple.o $(LIBS)

clean:
 rm $(OBJS) simple bound
```

Executing CQL statements

Once you have connected to a Cassandra cluster using a `Session` object, you 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

After connecting to a cluster and creating a `Session` object, you can execute CQL statements. In this tutorial, you will add code to your client for:

- printing out errors
- executing statements
  - creating a keyspace
  - creating tables
  - inserting data into those tables
  - querying the tables
  - printing the results

Procedure

1. Declare and implement an inline function, `printError()`:

```
inline CassError printError(CassError error)
{
    cout << cass_error_desc(error) << "\n";
    return error;
}
```

2. Declare and implement an inline function, `executeStatement()`:
   a) Declare the function in the class declaration in the header file (`SimpleClient.hpp`):

```
class SimpleClient
{

private:
    CassSession* session;
    CassCluster* cluster;
    inline CassError executeStatement(const char* cqlStatement, const
 CassResult** results /* = NULL */);
```

b) Declare the function in your implementation file (`SimpleClient.cpp`) and add code to hold the resulting error code and return it to the calling function and log some information to the user in the console:

```
inline CassError executeStatement(const char* cqlStatement, const
 CassResult** results /* = NULL */)
{
    CassError rc = CASS_OK;
    CassFuture* result_future = NULL;

    cout << "Executing " << cqlStatement << "\n";

    return rc;
}
```

c) After the logging statement, add code to create a statement from the specified query string and execute it on the session object:

```
inline CassError executeStatement(string cqlStatement, const
 CassResult** results /* = NULL */)
{
    // Some code elided
    cout << "Executing " << cqlStatement << "\n";

    CassString query = cass_string_init(cqlStatement.c_str());
    CassStatement* statement = cass_statement_new(query, 0);
    result_future = cass_session_execute(session, statement);

    return rc;
}
```

The driver is designed so that no function forces your application to block. Functions that would normally cause your application to block, such as connecting to a cluster or running a query, instead return a `CassFuture` object that can be waited on, polled, or used to register a callback. The driver API can also be used synchronously by immediately attempting to get the result from a future.

d) Add code that waits on the `CassFuture` object to complete:

```
inline CassError executeStatement(const char* cqlStatement, const
 CassResult** results /* = NULL */)
{
    // Some code elided

    result_future = cass_session_execute(session, statement);
    cass_future_wait(result_future);

    return rc;
}
```

e) Add code to retrieve the resulting error code and deal with it (by getting the result if `CASS_OK` or printing out the error message if not):

```
inline CassError executeStatement(const char* cqlStatement, const
 CassResult** results /* = NULL */)
{
    // Some code elided

    cass_future_wait(result_future);
```

```
    rc = cass_future_error_code(result_future);
    if (rc == CASS_OK)
    {
        cout << "Statement " << cqlStatement << " executed successully."
 << "\n";
        if ( results != NULL )
        {
            *results = cass_future_get_result(result_future);
        }
    }
    else
    {
        return printError(rc);
    }

    return rc;
}
```

f) Add code to free up the statement and future objects:

```
inline CassError executeStatement(const char* cqlStatement, const
 CassResult** results /* = NULL */)
{
    // Some code elided

    cass_statement_free(statement);
    cass_future_free(result_future);

    return rc;
}
```

Write code to create a schema on your cluster, load data into it, and query the data.

3. Declare and implement a function, `createSchema()`, that creates a keyspace with two tables in it:

a) Add the `createSchema()` function signature in the header file (`SimpleClient.hpp`):

```
class SimpleClient
{
    // Some code elided.

public:
    CassError createSchema();
};
```

b) Declare the function in your implementation file (`SimpleClient.cpp`) and add code to hold the resulting error code and return it to the calling function and log some information to the user in the console:

```
CassError SimpleClient::createSchema()
{
    CassError rc = CASS_OK;

    cout << "Creating simplex keyspace." << endl;

    return rc;
}
```

c) Add code to create a keyspace and two tables:

```
CassError SimpleClient::createSchema()
{
    // Some code elided
    rc = executeStatement(
        string("CREATE KEYSPACE IF NOT EXISTS simplex ") +
        "WITH replication = {'class':'SimpleStrategy',
 'replication_factor':3};");
    rc = executeStatement(
        string("CREATE TABLE simplex.songs (") +
              "id uuid PRIMARY KEY," +
              "title text," +
              "album text," +
              "artist text," +
              "tags set<text>," +
              "data blob" +
          ");");
    rc = executeStatement(
        string("CREATE TABLE simplex.playlists (") +
              "id uuid," +
              "title text," +
              "album text, " +
              "artist text," +
              "song_id uuid," +
              "PRIMARY KEY (id, title, album, artist)" +
              ");");

    return rc;
}
```

4. Declare and implement a function, `loadData()`, that populates the schema with some data:
   a) Add the `loadData()` function signature in the header file (`SimpleClient.hpp`):

```
class SimpleClient
{
    // Some code elided.

public:
    CassError loadData();
};
```

   b) Declare the function in your implementation file (`SimpleClient.cpp`) and add code to hold the resulting error code and return it to the calling function and log some information to the user in the console:

```
CassError SimpleClient::loadData()
{
    CassError rc = CASS_OK;

    cout << "Loading data into simplex keyspace." << endl;

    return rc;
}
```

   c) Add code to insert some data into the newly created schema:

```
CassError SimpleClient::loadData()
{
    // Some code elided.

    rc = executeStatement(
```

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

    rc = executeStatement(
        string("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'" +
        ");");
    rc = executeStatement(
        string("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'" +
        ");");
    rc = executeStatement(
        string("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'" +
        ");");

    return rc;
}
```

5. Declare and implement a function, `querySchema()`:
   a) Add the `querySchema()` function signature in the header file (`SimpleClient.hpp`):

```
class SimpleClient
{
    // Some code elided.

public:
    CassError querySchema();
};
```

   b) Declare the function in your implementation file (`SimpleClient.cpp`) and add code to hold the resulting error code and return it to the calling function and log some information to the user in the console:

```
CassError SimpleClient::querySchema()
{
    CassError rc = CASS_OK;

    cout << "Querying the simplex.playlists table." << endl;

    return rc;
}
```

   c) Add code to execute a simple query on one of the tables:

```
CassError SimpleClient::querySchema()
{
    // Some code elided.

    const CassResult* results;
    rc = executeStatement(
        string("SELECT title, artist, album FROM simplex.playlists ") +
        "WHERE id = 2cc9ccb7-6221-4ccb-8387-f22b6a1b354d;", &results);

    return rc;
}
```

   This time, you pass in a `CassResults` object to the `executeStatement()` function that will contain the results of the query.
   d) Add code to get a `CassIterator` object from the results object and check that the returned error code is `CASS_OK`:

```
CassError SimpleClient::querySchema()
{
    // Some code elided.

    CassIterator* rows = cass_iterator_from_result(results);

    if ( rc == CASS_OK )
    {
    }

    return rc;
}
```

e) Add code to retrieve the data from the result and print it out to the console:

```
CassError SimpleClient::querySchema()
{
    // Some code elided.

    CassIterator* rows = cass_iterator_from_result(results);

    if ( rc == CASS_OK )
    {
        while ( cass_iterator_next(rows) )
        {
            const CassRow* row = cass_iterator_get_row(rows);

            CassString title, artist, album;

            cass_value_get_string(cass_row_get_column(row, 0), &title);
            cass_value_get_string(cass_row_get_column(row, 1), &artist);
            cass_value_get_string(cass_row_get_column(row, 2), &album);

            cout << "title: " << title.data << ", artist: " <<
 artist.data << ", album: " << album.data << "\n";
        }
    }

    return rc;
}
```

f) Finish the implementation of the function by freeing the result and iterator objects:

```
CassError SimpleClient::querySchema()
{
    // Some code elided.

    CassIterator* rows = cass_iterator_from_result(results);

    if ( rc == CASS_OK )
    {
        // Some code elided.
        cass_result_free(results);
        cass_iterator_free(rows);
    }

    return rc;
}
```

Example

The complete code listing illustrates:

- write two inline auxiliary functions
- creating a keypsace with two tables
- loading data into your new schema
- retrieving a row from one of the tables

Code listing

SimpleClient.hpp

```
#ifndef __SIMPLE_CLIENT_H__
#define __SIMPLE_CLIENT_H__

#include "cassandra.h"

#include <string>

namespace example
{

class SimpleClient
{

private:
    CassSession* session;
    CassCluster* cluster;
    inline CassError executeStatement(std::string cqlStatement, const
 CassResult** results = NULL);
public:
    inline CassSession* getSession() { return session; }

    CassError connect(const std::string nodes);
    CassError createSchema();
    virtual CassError loadData();
    CassError querySchema();

    void close();

    SimpleClient() { }
    ~SimpleClient() { }

};

} // end namespace example

#endif
```

Code listing

SimpleClient.cpp

```
#include <iostream>
#include "SimpleClient.hpp"

namespace example {

using namespace std;

const std::string TAGS_COLUMN("tags");
const std::string TITLE_COLUMN("title");
const std::string ARTIST_COLUMN("artist");
const std::string ALBUM_COLUMN("album");

// auxiliary functions

inline CassError printError(CassError error)
```

```
{
    cout << cass_error_desc(error) << "\n";
    return error;
}

inline CassError SimpleClient::executeStatement(string cqlStatement, const
 CassResult** results /* = NULL */)
{
    CassError rc = CASS_OK;
    CassFuture* result_future = NULL;

    cout << "Executing " << cqlStatement << "\n";

    CassString query = cass_string_init(cqlStatement.c_str());
    CassStatement* statement = cass_statement_new(query, 0);
    result_future = cass_session_execute(session, statement);
    cass_future_wait(result_future);

    rc = cass_future_error_code(result_future);
    if (rc == CASS_OK)
    {
        cout << "Statement " << cqlStatement << " executed successully." <<
 "\n";
        if ( results != NULL )
        {
            *results = cass_future_get_result(result_future);
        }
    }
    else
    {
        return printError(rc);
    }
    cass_statement_free(statement);
    cass_future_free(result_future);

    return rc;
}

CassError SimpleClient::connect(const string nodes)
{
    CassError rc = CASS_OK;
    cout << "Connecting to " << nodes << "\n";
    cluster = cass_cluster_new();
    session = cass_session_new();
    CassFuture* connect_future = NULL;

    cass_cluster_set_contact_points(cluster, "127.0.0.1");

    connect_future = cass_session_connect(session, cluster);

    cass_future_wait(connect_future);
    rc = cass_future_error_code(connect_future);

    if ( rc == CASS_OK )
    {
        cout << "Connected." << "\n";
    }
    else
    {
        return printError(rc);
    }
    cass_future_free(connect_future);
    return rc;
}
```

```
CassError SimpleClient::createSchema()
{
    CassError rc = CASS_OK;

    cout << "Creating simplex keyspace." << endl;
    rc = executeStatement(string("CREATE KEYSPACE IF NOT EXISTS simplex ") +
        "WITH replication = {'class':'SimpleStrategy',
 'replication_factor':3};");
    rc = executeStatement(
        string("CREATE TABLE simplex.songs (") +
              "id uuid PRIMARY KEY," +
              "title text," +
              "album text," +
              "artist text," +
              "tags set<text>," +
              "data blob" +
          ");");
    rc = executeStatement(
        string("CREATE TABLE simplex.playlists (") +
              "id uuid," +
              "title text," +
              "album text, " +
              "artist text," +
              "song_id uuid," +
              "PRIMARY KEY (id, title, album, artist)" +
              ");");

    return rc;
}

CassError SimpleClient::loadData()
{
    CassError rc = CASS_OK;

    cout << "Loading data into simplex keyspace." << endl;

    rc = executeStatement(
        string("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'})" +
        ";");
    rc = executeStatement(
        string("INSERT INTO simplex.songs (id, title, album, artist, tags)
 ") +
        "VALUES (" +
            "f6071e72-48ec-4fcb-bf3e-379c8a696488," +
            "'Die Mösch'," +
            "'In Gold'," +
            "'Willi Ostermann'," +
            "{'kölsch', '1996', 'birds'}" +
        ");");
    rc = executeStatement(
        string("INSERT INTO simplex.songs (id, title, album, artist, tags)
 ") +
        "VALUES (" +
            "fbdf82ed-0063-4796-9c7c-a3d4f47b4b25," +
            "'Memo From Turner'," +
            "'Performance'," +
```

```
            "'Mick Jager'," +
            "{'soundtrack', '1991'}" +
        ");");

    rc = executeStatement(
        string("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'" +
        ");");
    rc = executeStatement(
        string("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'" +
        ");");
    rc = executeStatement(
        string("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'" +
        ");");

    return rc;
}

CassError SimpleClient::querySchema()
{
    CassError rc = CASS_OK;
    const CassResult* results;

    cout << "Querying the simplex.playlists table." << endl;

    rc = executeStatement(
        string("SELECT title, artist, album FROM simplex.playlists ") +
        "WHERE id = 2cc9ccb7-6221-4ccb-8387-f22b6a1b354d;", &results);

    CassIterator* rows = cass_iterator_from_result(results);

    if ( rc == CASS_OK )
    {
        while ( cass_iterator_next(rows) )
        {
            const CassRow* row = cass_iterator_get_row(rows);

            CassString title, artist, album;

            cass_value_get_string(cass_row_get_column(row, 0), &title);
            cass_value_get_string(cass_row_get_column(row, 1), &artist);
            cass_value_get_string(cass_row_get_column(row, 2), &album);
```

```
            cout << "title: " << title.data << ", artist: " << artist.data
 << ", album: " << album.data << "\n";
        }
        cass_result_free(results);
        cass_iterator_free(rows);
    }

    return rc;
}

void SimpleClient::close()
{
    cout << "Closing down cluster connection." << "\n";
    cass_session_close(session);
    cass_cluster_free(cluster);
}

} // end namespace example
```

Reference

Reference for the C/C++ driver.

Basics

Statements, results sets, futures, and schemas.

BATCH statements

Group together two or more statements and execute them as though they were a single statemen

Batches are used to group multiple mutations (`UPDATE`, `INSERT`, and `DELETE`) together into a single statement. There are three different types of batches supported by Cassandra.

- `CASS_BATCH_TYPE_LOGGED` is used to make sure that multiple mutations across multiple partitions happen atomically, that is, all the included mutations will eventually succeed. However, there is a performance penalty imposed by atomicity guarantee.
- `CASS_BATCH_TYPE_UNLOGGED` is generally used to group mutations for a single partition and do not suffer from the performance penalty imposed by logged batches, but there is no atomicity guarantee for multi-partition updates.
- `CASS_BATCH_TYPE_COUNTER` is used to group counters updates.

Note: Be careful when using batches as a performance optimization.

```
/* This logged batch will makes sure that all the mutations eventually
 succeed */
CassBatch* batch = cass_batch_new(CASS_BATCH_TYPE_LOGGED);

/* Statements can be immediately freed after being added to the batch */

{
   CassStatement* statement
      = cass_statement_new(cass_string_init("INSERT INTO example1(key,
 value) VALUES ('a', '1')"), 0);
   cass_batch_add_statement(batch, statement);
   cass_statement_free(statement);
}

{
   CassStatement* statement
      = cass_statement_new(cass_string_init("UPDATE example2 set value = '2'
 WHERE key = 'b'"), 0);
   cass_batch_add_statement(batch, statement);
   cass_statement_free(statement);
}

{
   CassStatement* statement
      = cass_statement_new(cass_string_init("DELETE FROM example3 WHERE key
 = 'c'"), 0);
   cass_batch_add_statement(batch, statement);
   cass_statement_free(statement);
}

CassFuture* batch_future = cass_session_execute_batch(session, batch);
```

```
/* Batch objects can be freed immediately after being executed */
cass_batch_free(batch);

/* This will block until the query has finished */
CassError rc = cass_future_error_code(batch_future);

printf("Batch result: %s\n", cass_error_desc(rc));

cass_future_free(batch_future);
```

Futures

A future is a kind of proxy for a result that is initially unknown, because the computation of its value is yet incomplete.

Futures are returned from any driver call that would have required your application to block. This allows your application to continue processing and/or also submitting several queries at once. Although the driver has an asynchronous design it can be used synchronously by immediately attempting to get result or explicitly waiting for the future.

Waiting for results

You can wait indefinitely for futures results by either calling a wait function or by attempting to get the result. You can also wait for results for an specified amount of time or you can poll without waiting.

Waiting synchronously

```
CassFuture* future = /* Some operation */

/* Block until a result or error is set */
cass_future_wait(future);

cass_future_free(future);
```

Waiting synchronously for the result

```
CassFuture* future = cass_session_execute(session, statement);

/* Blocks and gets a result */
const CassResult* result = cass_future_get_result(future);

/* If there was an error then the result won't be available */
if (result == NULL) {
   /* The error code and message will be set instead */
   CassError error_code = cass_future_error_code(future);
   CassString error_message = cass_future_error_message(future);

   /* Handle error */
}

cass_future_free(future);
```

Timed waiting

```
CassFuture* future = /* Some operation */
```

```
cass_duration_t microseconds = 30 * 1000000; /* 30 seconds */

/* Block for a fixed amount of time */
if (cass_future_wait_timed(future, microseconds) {
   /* A result or error was set during the wait call */
} else {
   /* The operation hasn't completed yet */
}

cass_future_free(future);
```

Polling

```
CassFuture* future = /* Some operation */

/* Poll to see if the future is ready */
while (!cass_future_ready(future)) {
   /* Run other application logic or wait */
}

/* A result or error was set */

cass_future_free(future);
```

Using callbacks

You can set a callback on a future, and it notifies your application when a request has completed. Using a future callback is the lowest latency way of notification when waiting for several asynchronous operations.

Note: The driver may run the callback on thread that's different from the application's calling thread. Any data accessed in the callback must be immutable or synchronized with a mutex, semaphore, etc.

Example

```
void on_result(CassFuture* future, void* data) {
   /* This result will now return immediately */
   CassError rc = cass_future_error_code(future);
   printf("%s\n", cass_error_desc(rc));
}

/* Code elided ... */

CassFuture* future = /* Some operation */;

/* Set a callback instead of waiting for the result to be returned */
cass_future_set_callback(on_result, NULL);

/* The application's reference to the future can be freed immediately */
cass_future_free(future);

/* Run other application logic */
```

Handling results

A result object contains the results of a SELECT statement.

The `CassResult` object is generally only returned for `SELECT` statements. For mutations (`INSERT`, `UPDATE`, and `DELETE`) only a status code is present and is accessed using `cass_future_error_code()`. However, when using lightweight transactions a result object is available to check the status of the transaction. The result object is obtained from the query's future object.

Note: Rows, column values, collections, decimals, strings, and bytes objects are all invalidated when the result object is freed. All of these objects point to memory held by the result. This allows the driver to avoid unnecessarily copying data.

```
const CassResult* result = cass_future_get_result(future);

/* Process result */

cass_result_free(result);
```

Note:

The result object is immutable and may be accessed by multiple threads concurrently.

Rows and columns values

The result object represents a collection of rows. The first row, if present, can be obtained using `cass_result_first_row()`. Multiple rows are accessed using a `CassIterator` object. Once a row is retrieved the column values are retrieved from a row by either index or by name. A `CassIterator` object is used for enumerated column values.

Column value by index example

```
const CassRow* row = cass_result_first_row(result);

/* Get the first column value using the index */
const CassValue* column1 = cass_row_get_column(row, 0);
```

Column value by name example

```
const CassRow* row = cass_result_first_row(result);

/* Get the value of the column named "column1" */
const CassValue* column1 = cass_row_get_column_by_name(row, "column1");
```

Retrieving a column value

Once you have the column values you retrieve the actual value of the data in the column.

```
cass_int32_t int_value;
cass_value_get_int32(column1, &int_value);

cass_int64_t timestamp_value;
cass_value_get_int32(column2, &timestamp_value);

CassString string_value;
cass_value_get_string(column3, &string_value);
```

Iterators

Iterators are used to iterate over the rows in a result, the columns in a row, or the values in a collection.

Note: `cass_iterator_next()` invalidates values retrieved by the previous iteration.

```
const CassResult* result = cass_future_get_result(future);

CassIterator* iterator = cass_iterator_from_result(result);

while (cass_iterator_next(iterator)) {
   const CassRow* row = cass_iterator_get_row(iterator);
   /* Retrieve and use values from the row */
}

cass_iterator_free(iterator);
```

All the iterators use the same pattern, but are instantiated differently and have different retrieval functions. Iterating over a map collection is slightly different because it has two values per entry, but it uses the same basic pattern.

```
/* Execute SELECT query where a map collection is returned */

const CassResult* result = cass_future_get_result(future);

const CassRow* row = cass_result_first_row(result);

const CassValue* map = cass_row_get_column(row, 0);

CassIterator* iterator = cass_iterator_from_map(map);

while (cass_iterator_next(iterator)) {
   /* A separate call is used to get the key and the value */
   const CassValue* key = cass_iterator_get_map_key(iterator);
   const CassValue* value = cass_iterator_get_map_value(iterator);

   /* Use key/value pair */
}

cass_iterator_free(iterator);
```

Paging

Large result sets can be divided into multiple pages automatically using paging. The result object keeps track of the pagination state for the sequence of paging queries. When paging through a result set the result object is checked to see if more pages exist and then attached to the statement before re-executing the query to get the next page.

```
CassString query = cass_string_init("SELECT * FROM table1");
CassStatement* statement = cass_statement_new(query, 0);

/* Return a 100 rows every time this statement is executed */
cass_statement_set_paging_size(statement, 100);

cass_bool_t has_more_pages = cass_true;

while (has_more_pages) {
   CassFuture* query_future = cass_session_execute(session, statement);
```

```
   const CassResult* result = cass_future_get_result(future);

   if (result == NULL) {
      /* Handle error */
      cass_future_free(query_future);
      break;
   }

   /* Get values from result... */

   /* Check to see if there are more pages remaining for this result */
   has_more_pages = cass_result_has_more_pages(result);

   if (has_more_pages) {
      /* If there are more pages we need to set the position for the next
 execute */
      cass_statement_set_paging_state(statement, result);
   }

   cass_result_free(result);
}
```

Keyspaces

There are several strategies for using keyspaces with a session.

Setting keyspaces at connection time

It's best to only create a single session per keyspace. Specify the keyspace when initially connecting a session.

```
CassFuture* connect_future
   = cass_session_connect_keyspace(session, cluster, "keyspace1");
```

Changing the keyspace with a USE statement

To change the keyspace of an already connected session, execute a `USE` statement.

Using a single session with multiple keyspaces

To use multiple keyspaces from a single session, fully qualify the table names in your queries, (for example, `keyspace_name.table_name`.

```
SELECT * FROM keyspace_name.table_name WHERE ...;
INSERT INTO keyspace_name.table_name (...) VALUES (...);
```

Prepared statements

Prepare a statement once (on the server) and pass values to bind to its variables before executing it.

Prepared statements can be used to improve the performance of frequently executed queries. Preparing the query caches it on the Cassandra cluster and only needs to be performed once. Once created, prepared statements should be reused with different bind variables. Prepared queries use the ? marked to denote bind variables in the query string.

```
/* Prepare the statement on the Cassandra cluster */
CassFuture* prepare_future
```

```
   = cass_session_prepare(session, "INSERT INTO example (key, value) VALUES
 (?, ?)");

/* Wait for the statement to prepare and get the result */
CassError rc = cass_future_error_code(prepare_future);

printf("Prepare result: %s\n", cass_error_desc(rc));

if (rc != CASS_OK) {
   /* Handle error */
   cass_future_free(prepare_future);
   return -1;
}

/* Get the prepared object from the future */
const CassPrepared* prepared = cass_future_get_prepared(prepared_future);

/* The future can be freed immediately after getting the prepared object */
cass_future_free(prepare_future);

/* The prepared object can now be used to create statements that can be
 executed */
CassStatement* statement = cass_prepared_bind(prepared);

/* Bind variables by name this time (this can only be done with prepared
 statements)*/
cass_statement_bind_string_by_name(statement, "key",
 cass_string_init("abc"));
cass_statement_bind_int32_by_name(statement, "value", 123);

/* Execute statement (same a the non-prepared code) */

/* The prepared object must be freed */
cass_prepared_free(prepared);
```

Bound parameters

The driver supports two kinds of bound parameters: by marker and by name.

Binding parameters

The ? marker is used to denote the bind variables in a query string. This is used for both regular and prepared parameterized queries. In addition to adding the bind marker to your query string, your application must also provide the number of bind variables to `cass_statement_new()` when constructing a new statement. If a query doesn't require any bind variables then 0 can be used. The `cass_statement_bind_*()` functions are then used to bind values to the statement's variables.

Bind variables can be bound by the marker index or by name.

Bind by marker index example

```
CassString query = cass_string_init("SELECT * FROM table1 WHERE column1
 = ?");

/* Create a statement with a single parameter */
CassStatement* statement = cass_statement_new(query, 1);

cass_statement_bind_string(statement, 0, cass_string_init("abc"));

/* Execute statement */
```

```
cass_statement_free(statement);
```

Bind by marker name example

Variables can only be bound by name for prepared statements. This limitation exists because query metadata provided by Cassandra is required to map the variable name to the variable's marker index.

```
/* Prepare statement */

/* The prepared query allocates the correct number of parameters
 automatically */
CassStatement* statement = cass_prepared_bind(prepared);

/* The parameter can now be bound by name */
cass_statement_bind_string_by_name(statement, "column1",
 cass_string_init("abc"));

/* Execute statement */

cass_statement_free(statement);
```

Binding large values

The data of values bound to statements are copied into the statement. That means that values bound to statements can be freed immediately after being bound. However, this might be problematic for large values so the driver provides `cass_statement_bind_custom()` which allocates a buffer where the large value is directly stored avoiding an extra allocation and copy.

```
CassString query =
   cass_string_init("INSERT INTO table1 (column1, column2) VALUES (?, ?)");

/* Create a statement with two parameters */
CassStatement* statement = cass_statement_new(query, 2);

cass_statement_bind_string(statement, 0, cass_string_init("abc"));

cass_byte_t* bytes;
cass_statement_bind_custom(statement, 1, 8 * 1024 * 1024, &bytes);

/* 'bytes' then can be used in the application to store a large value */

/* Execute statement */
```

Constructing connections

Collections are supported using `CassCollection` objects, which supports the `list`, `map`, and `set` CQL types.

Note: Values appended to the collection can be freed immediately afterward because the values are copied.

List example

This shows how to construct a list collection. A set can be constructed in a very similar way. The difference is the type `CASS_COLLECTION_TYPE_SET` is used to create the collection instead of `CASS_COLLECTION_TYPE_LIST`.

```
CassStatement* statement = cass_statement_new(query, 1);

CassCollection* list = cass_collection_new(CASS_COLLECTION_TYPE_LIST, 3);

cass_collection_append_string(list, cass_string_init("123"));
cass_collection_append_string(list, cass_string_init("456"));
cass_collection_append_string(list, cass_string_init("789"));

cass_statement_bind_collection(statement, 0, list);

/* The collection can be freed after binding */
cass_collection_free(list);
```

Map example

Maps are built similarly, but the key and value need to be interleaved as they are appended to the collection.

```
CassStatement* statement = cass_statement_new(query, 1);

CassCollection* map = cass_collection_new(CASS_COLLECTION_TYPE_MAP, 2);

/* map["abc"] = 123 */
cass_collection_append_string(map, cass_string_init("abc"));
cass_collection_append_int32(map, 123);

/* map["def"] = 456 */
cass_collection_append_string(map, cass_string_init("def"));
cass_collection_append_int32(map, 456);

cass_statement_bind_collection(statement, 0, map);

/* The collection can be freed after binding */
cass_collection_free(map);
```

Schema metadata

Metadata about a session object's keyspace and tables is provided by the driver.

The driver provides access to keyspace and table metadata. This schema metadata is monitored by the control connection and automatically kept up-to-date.

```
/* Create session */

/* Get snapshot of the schema */
const CassSchema* schema = cass_session_get_schema(session);

/* Get information about the "keyspace1" schema */
const CassSchemaMeta* keyspace1_meta
   = cass_schema_get_keyspace(schema, "keyspace1");

if (keyspace1_meta == NULL) {
   /* Handle error */
}

/* Get the key-value filed for "strategy_class" */
const CassSchemaMetaField* strategy_class_field
   = cass_schema_meta_get_field(keyspace1_meta, "strategy_class");

if (strategy_class_field == NULL) {
```

```
   /* Handle error */
}

/* Get the value part of the field */
const CassValue* strategy_class_value
   = cass_schema_meta_field_value(strategy_class_field);

/* Fields values use the existing cass_value*() API */
CassString strategy_class;
cass_value_get_string(strategy_class_value, &strategy_class);

/* Do something with strategy_class */

/* All values derived from the schema are cleaned up */
cass_schema_free(schema);
```

The snapshot obtained by `cass_session_get_schema()` does not see schema changes that happened after the call. A new snapshot needs to be obtained to see subsequent updates to the schema.

Client configuration

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

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

The configuration of a client 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).

CQL data types to C/C++ types

A summary of the mapping between CQL data types and C/C++ data types is provided.

Description

When retrieving the value of a column from a `Row` object, the value is typed according to the following table.

Table 1: C/C++ types to CQL data types

CQL data type	C/C++ type
ascii	CassString
bigint	cass_int64_t
blob	CassBytes
boolean	cass_bool_t
counter	cass_int64_t
decimal	CassDecimal
double	cass_double_t
float	cass_float_t
inet	CassInet
int	cass_int32_t

CQL data type	C/C++ type
list	CassCollection
map	CassCollection
set	CassCollection
text	CassString
timestamp	cass_int64_t
timeuuid	CassUuid
uuid	CassUuid
varchar	CassString
varint	CassBytes

Removal of `CassString`, `CassBytes`, and `CassDecimal` types

The previous version of the driver used wrapper types to represent strings, bytes and decimal types. This provided a convenient way to encapsulate an array of data with its length. However, its use throughout the API was inconsistent, especially when it came to strings. Some functions would accept strings as null-terminated char arrays and others used `CassString`. These extra types also obfuscated the lifetime's of the memory pointed to by these objects. This release improves the API by removing these wrapper types and consistently uses two parameters in their place, a pointer to the data and the size of that data. For functions that take strings there are always two versions of that function, one that accepts a null-terminated char array, and one that makes the lengths explicit (these functions have the suffix _n).

Functions that accepted `CassStrings` before look like this:

```
CassString query = cass_string_init("SELECT keyspace_name "
                                    "FROM system.schema_keyspaces");
CassStatement* statement = cass_statement_new(query, 0);

/* Use statement */

cass_statement_free(statement);
```

Now, instead functions accept the string literal directly:

```
CassStatement* statement = cass_statement_new("SELECT keyspace_name "
                                              "FROM
 system.schema_keyspaces", 0);

/* Use statement */

cass_statement_free(statement);
```

They can also be used with an explicit lengths:

```
const char* query = "SELECT keyspace_name "
                    "FROM system.schema_keyspaces";
CassStatement* statement = cass_statement_new_n(query, strlen(query), 0);

/* Use statement */
```

```
cass_statement_free(statement);
```

Latency-aware routing

Reroute queries from poorly-performing nodes to better-performing ones.

Latency-aware routing tracks the latency of queries to avoid sending new queries to poorly performing Cassandra nodes. It can be used in conjunction with other load-balancing and routing policies.

```
/* Disable latency-aware routing (this is the default setting) */
cass_cluster_set_latency_aware_routing(cluster, cass_false);

/* Enable latency-aware routing */
cass_cluster_set_latency_aware_routing(cluster, cass_true);

/*
 * Configure latency-aware routing settings
 */

/* Up to 2 times the best performing latency is okay */
cass_double_t exclusion_threshold = 2.0;

 /* Use the default scale */
cass_uint64_t scale_ms = 100;

/* Retry a node after 10 seconds even if it was performing poorly before */
cass_uint64_t retry_period_ms = 10000;

/* Find the best performing latency every 100 milliseconds */
cass_uint64_t update_rate_ms = 100;

/* Only consider the average latency of a node after it's been queried 50
 times */
cass_uint64_t min_measured = 50;

cass_cluster_set_latency_aware_routing_settings(cluster,
                                                exclusion_threshold,
                                                scale_ms,
                                                retry_period_ms,
                                                update_rate_ms,
                                                min_measured);
```

Logging

Use the driver's logging mechanism to record different kinds of information about your client.

The driver's logging system uses `stderr` by default and the log level `CASS_LOG_WARN`. Both of these settings are changed using the driver's `cass_log_*()` configuration functions.

Note: Logging configuration must be done before calling any other driver function.

Log level

To update the log level use `cass_log_set_level()`.

```
cass_log_set_level(CASS_LOG_INFO);
/* Create cluster and connect session */
```

Custom logging callback

The use of a logging callback allows an application to log messages to a file, syslog, or any other logging mechanism. This callback must be thread-safe, because it is possible for it to be called from multiple threads concurrently. The data parameter allows custom resources to be passed to the logging callback.

```
void on_log(const CassLogMessage* message, void* data) {
   /* Handle logging */
}

// Code elided ...

void* log_data = custom-log-resource;
cass_log_set_callback(on_log, log_data);
cass_log_set_level(CASS_LOG_INFO);

/* Create cluster and connect session */
```

Logging cleanup

Resources passed to a custom logging callback should be cleaned up by calling `cass_log_cleanup()`. This shuts down the logging system and ensures that the custom callback will no longer be called.

```
/* Close any sessions */

cass_log_cleanup();

/* Free custom logging resources */
```

Performance metrics

Retrieve performance metrics and other diagnostic information from the driver.

Performance metrics and diagnostic information can be obtained from the driver using `cass_session_get_metrics()`. The resulting `CassMetrics` object contains several useful metrics for accessing request performance or debugging issues.

```
CassMetrics metrics;

/* Get a snapshot of the driver's metrics */
cass_session_get_metrics(session, &metrics);
```

Requests

The `requests` field contains information about request latency and throughput. All latency times are in microseconds and throughput numbers are in requests per seconds.

Statistics

The `stats` field contains information about connections and backpressure markers. If the number of `available_connections` is less than the number of `total_connections` this could mean the number of I/O threads or number of connections may need to be increased. The same is true for `exceeded_pending_requests_water_mark` and `exceeded_write_bytes_water_mark` metrics. It could also mean the Cassandra cluster is unable to handle the current request load.

Errors

The `errors` field contains information about the occurrence of requests and connection timeouts. Request timeouts occur when a request fails to get a timely response (default: 12 seconds). Pending request timeouts occur whens a request waits too long to be serviced by an assigned host. This can occur when too many requests are in-flight for a single host. Connection timeouts occur when the process of establishing new connections is unresponsive (default: 5 seconds).

Security

The driver currently supports plain text authentication and SSL (via OpenSSL).

The driver currently supports plain text authentication and SSL (via OpenSSL).

Authentication

Authentication allows your application to supply credentials that are used to control access to Cassandra resources. The driver currently only supports plain text authentication, so it is best used in conjunction with SSL. A future release may relax this constraint with the use of SASL authentication.

Credentials are passed using the following:

```
CassCluster* cluster = cass_cluster_new();

const char* username = "username1";
const char* password = "password1";

cass_cluster_set_credentials(cluster, username, password);

/* Connect session object */

cass_cluster_free(cluster);
```

SSL

This example uses a self-signed certificate, but most steps are similar for certificates generated by a certificate authority (CA). First, generate a public and private key pair for your Cassandra nodes and configure them to use the generated certificate.

Keystore and truststore may be used interchangeably. These can and oftentimes are the same file. This example uses the same file for both (that is, `keystore.jks`). The difference is keystores generally hold private keys and truststores hold public keys/certificate chains. Variable fields in example need to be replaced with values specific to your environment.

SSL can be rather cumbersome to setup. If you run into issues or have trouble configuring SSL please use the mailing list or IRC.

Generating the keys

The most secure method of setting up SSL is to verify that domain name or the IP address used to connect to the server matches identity information found in the SSL certificate. This helps to prevent man-in-the-middle attacks. Cassandra uses IP addresses internally so that’s the only supported information for identity verification. That means that the IP address of the Cassandra server where the certificate is installed needs to be present in either the certificate’s common name (CN) or one of its subject alternative names (SANs). It’s possible to create the certificate without either, but then it will not be possible to verify the server’s identity. Although this is not as secure, it eases the deployment of SSL by allowing the same certificate to be deployed across the entire Cassandra cluster.

To generate a public-private key pair with the IP address in the CN field use the following:

```
keytool -genkeypair -noprompt -keyalg RSA -validity 36500 \
   -alias node \
   -keystore keystore.jks \
   -storepass keystore-password \
   -keypass key-password \
   -dname "CN=IP-address-goes-here, OU=Drivers and Tools, O=DataStax Inc.,
 L=Santa Clara, ST=California, C=US"
```

If you would prefer to use a SAN use this command:

```
keytool -genkeypair -noprompt -keyalg RSA -validity 36500 \
   -alias node \
   -keystore keystore.jks \
   -storepass keystore-password \
   -keypass key-password \
   -ext SAN="IP-address-goes-here" \
   -dname "CN=node1.datastax.com, OU=Drivers and Tools, O=DataStax Inc.,
 L=Santa Clara, ST=California, C=US"
```

Enabling client-to-node encryption

Copy the generated keystore from the previous step to your Cassandra node(s), and add the following to your `cassandra.yaml` file.

```
client_encryption_options:
  enabled: true
  keystore: path-to-keystore/keystore.jks
  keystore_password: keystore-password ## The password you used when
 generating the keystore.
  truststore: path-to-keystore/keystore.jks
  truststore_password: keystore-password
  require_client_auth: true-or-false
```

Note: This example uses the same file for both the keystore and truststore.

Setting up SSL

Create an SSL object and configure it:

```
#include <cassandra.h>

void setup_ssl(CassCluster* cluster) {
  CassSsl* ssl = cass_ssl_new();

  // Configure SSL object...

  // To enable SSL attach it to the cluster object
  cass_cluster_set_ssl(cluster, ssl);

  // You can detach your reference to this object once it's
  // added to the cluster object
  cass_ssl_free(ssl);
```

Exporting and loading the public key

The default setting of the driver is to verify the certificate sent during the SSL handshake. For the driver to properly verify the Cassandra certificate the driver needs either the public key from the self-signed public key or the CA certificate chain used to sign the public key. To have this work, extract the public key from the Cassandra keystore generated in the previous steps. This exports a PEM formatted certificate which is required by the driver.

```
keytool -exportcert -rfc -noprompt \
   -alias node \
   -keystore keystore.jks \
   -storepass keystore-password \
   -file cassandra.pem
```

The trusted certificate is loaded using the following code:

```
c int load_trusted_cert_file(const char* file, CassSsl* ssl) {
   CassError rc;
   char* cert;
   long cert_size;

   FILE *in = fopen(file, “rb”);
   if (in == NULL)
   {
      fprintf(stderr, “Error loading certificate file ’%s’\n”, file);
      return 0;
   }

   fseek(in, 0, SEEK_END); cert_size = ftell(in); rewind(in);

   cert = (char*)malloc(cert_size);
   fread(cert, sizeof(char), cert_size, in);
   fclose(in);

   // Add the trusted certificate (or chain) to the driver
   rc = cass_ssl_add_trusted_cert(ssl, cass_string_init2(cert, cert_size));
   if (rc != CASS_OK)
   {
      fprintf(stderr, “Error loading SSL certificate: %s\n”,
         cass_error_desc(rc));
      free(cert);
      return 0;
   }

   free(cert);
   return 1;
}
```

It is possible to load multiple self-signed certificates or CA certificate chains. In the case where you’re using self-signed certificates with unique IP addresses this is required. It is possible to disable the certificate verification process, but it is not recommended.

```
// Disable certificate verification
cass_ssl_set_verify_flags(ssl,CASS_SSL_VERIFY_NONE);
```

Enabling identity verification

If you've generated a unique certificate for each Cassandra node with the IP address in the CN or SAN fields you'll need to also enable identity verification. This is disabled by default.

```
// Add identity verification flag: CASS_SSL_VERIFY_PEER_IDENTITY
cass_ssl_set_verify_flags(ssl, CASS_SSL_VERIFY_PEER_CERT |
 CASS_SSL_VERIFY_PEER_IDENTITY);
```

Using with client-side certificates

Client-side certificates allow Cassandra to authenticate the client using public key cryptography and chains of trust. This is same process as above but in reverse. The client has a public and private key and the Cassandra node has a copy of the private key or the CA chain used to generate the pair.

Generating and loading the client-side certificate

A new public-private key pair needs to be generated for client authentication.

```
keytool -genkeypair -noprompt -keyalg RSA -validity 36500 \
   -alias driver \
   -keystore keystore-driver.jks \
   -storepass keystore password \
   -keypass key-password
```

The public and private key then need to be extracted and converted to the PEM format. To extract the public:

```
keytool -exportcert -rfc -noprompt \
   -alias driver \
   -keystore keystore-driver.jks \
   -storepass keystore password \
   -file driver.pem
```

To extract and convert the private key:

```
keytool -importkeystore -noprompt -srcalias certificatekey -deststoretype
 PKCS12 \
  -srcalias driver \
  -srckeystore keystore-driver.jks \
  -srcstorepass keystore password \
  -storepass key-password \
  -destkeystore keystore-driver.p12

openssl pkcs12 -nomacver -nocerts \
  -in keystore-driver.p12 \
  -password pass:key-password \
  -passout pass:key-password \
  -out driver-private.pem
```

Now you can load the PEM formatted public and private key. The files can be loaded using the same code from above in `load_trusted_cert_file()`.

```
CassError rc = CASS_OK;

char* cert = NULL;
```

```
size_t cert_size = 0;

// Load PEM-formatted certificate data and size into cert and cert_size...

rc = cass_ssl_set_cert(ssl, cass_string_init2(cert, cert_size));
if (rc != CASS_OK) {
   // Handle error
}

char* key = NULL;
size_t key_size = 0;

// A password is required when the private key is encrypted. If the private
 key
// is NOT password protected use NULL.
const char* key_password = "<key password>";

// Load PEM-formatted private key data and size into key and key_size...

rc = cass_ssl_set_private_key(ssl, cass_string_init2(key, key_size),
 key_password);
if (rc != CASS_OK) {
   // Handle error
}
```

Setting up client authentication on the cluster

The driver's public key or the CA chain used to sign the driver's certificate will need to be added to Cassandra's truststore. If using self-signed certificate then the public key needs to be extracted from the driver's keystore generated in the previous steps.

Extract the public key from the driver's keystore and add it to Cassandra's truststore.

```
keytool -exportcert -noprompt \
   -alias driver \
   -keystore keystore-driver.jks \
   -storepass cassandra \
   -file cassandra-driver.crt

keytool -import -noprompt \
   -alias truststore \
   -keystore keystore.jks \
   -storepass cassandra \
   -file cassandra-driver.crt
```

Enable client authentication in the `cassandra.yaml` file.

```
require_client_auth:true
```

FAQ

Which versions of Cassandra does the driver support?

Version 2.0 of the driver supports any Cassandra version greater than 1.2 and 2.0. Version 2.1 is not supported.

Which versions of CQL does the driver support?

It supports CQL version 1 and 2. CQL version 3 is not supported..

How do I generate a uuid or a timebased uuid?

Create a `CassUuidGen` object and call the `cass_uuid_gen_random()` for a `uuid` and `cass_uuid_gen_time()` for a `timeuuid`.

Should I create one client instance per module in my application?

Normally you should use one client instance per application. You should share that instance between modules within your application.

Should I shut down the pool after executing a query?

No. You should only close a session once when your application is terminating.

API reference

DataStax C/C++ Driver 2.0 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