



Java Driver 2.1 for Apache Cassandra Documentation

`#{ds.localized.time}`

Contents

About the Java driver.....	4
Architecture.....	5
The driver and its dependencies.....	5
Writing your first client.....	7
Connecting to a Cassandra cluster.....	7
Using a session to execute CQL statements.....	9
Using bound statements.....	14
Java driver reference.....	16
Four simple rules for coding with the driver.....	16
Asynchronous I/O.....	17
Automatic failover.....	18
BATCH statements.....	19
Cluster configuration.....	21
Tuning policies.....	21
Connection options.....	22
Connection requirements.....	24
CQL data types to Java types.....	24
CQL statements.....	25
Building dynamic queries programmatically with the <code>QueryBuilder</code> API.....	26
Debugging.....	29
Exceptions.....	29
Monitoring.....	30
Enabling tracing.....	32
Enabling debug-level logging.....	37
Node discovery.....	39
Object-mapping API.....	39
Basic CRUD operations.....	39
Mapping UDTs.....	42
Accessor-annotated interfaces.....	44
Setting up your Java development environment.....	45
Tuple types.....	45
User-defined types.....	46
UDT API.....	46
Direct field manipulation.....	47
FAQ.....	49
Can I check if a conditional statement (lightweight transaction) was successful?.....	49
What is a parameterized statement and how can I use it?.....	49
Does a parameterized statement escape parameters?.....	49
What's the difference between a parameterized statement and a Prepared statement?.....	49
Can I combine Prepared statements and normal statements in a batch?.....	50
Can I get the raw bytes of a text column?.....	50
How to increment counters with <code>QueryBuilder</code> ?.....	50

Is there a way to control the batch size of the results returned from a query?.....50
What's the difference between using setFetchSize() and LIMIT?.....50

API reference.....51

Using the docs.....52

About the Java driver

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

The Java 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 2.1.6?

- Revert of JAVA-425, which marked a node down when a query timed out, and led to `NoHostAvailableExceptions` under high load (probably our most requested change!) This no longer happens, and if a node is really having problems, it will be detected by other existing mechanisms (notification from another node that detected it through gossip, failed connection heartbeat).
- Upgrade of Netty to 4.0.27 and application-side coalescing of protocol frames.
- Speculative executions to preemptively retry requests when a host takes too long to reply (the client-side equivalent to Cassandra's rapid read retry).
- Query logger that allows you to trace slow or failed queries.
- Access to the paging state and ability to re-inject it into a statement.
- New pool resizing algorithm (JAVA-419) that fixes issues with variable-sized pools (core connections != max).
- Internal optimizations around connection usage.
- EC2 address translator (Previously released in version 2.1.5), connection heartbeat, and token metadata API.

What's new in 2.1?

The driver is compatible with all versions of Cassandra 1.2 and later.

- **Cassandra 2.1 support**
 - [User-defined types](#) (UDT)
 - [Tuple type](#)
- **New features**
 - Simple [object mapping API](#)

Architecture

An overview of the Java driver architecture.

The Java Driver 2.1 for Apache Cassandra works exclusively with the Cassandra Query Language version 3 (CQL3) and Cassandra's new 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. A Mapping and a JDBC module will be added on top of that in upcoming versions of the driver.

The driver relies on [Netty](#) to provide non-blocking I/O with Cassandra for providing a fully asynchronous architecture. Multiple queries can be submitted to the driver which then will dispatch the responses to the appropriate client threads.

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.
- **Nodes discovery:** the driver automatically discovers and uses all nodes of the Cassandra cluster, including newly bootstrapped ones.
- **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).
- **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.
- **Cassandra trace handling:** tracing can be set on a per-query basis and the driver provides a convenient API to retrieve the trace.
- **Convenient schema access:** the driver exposes a Cassandra schema in a usable way.
- **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.
- **Tunability:** the default behavior of the driver can be changed or fine tuned by using tuning policies and connection options.

Queries can be executed synchronously or asynchronously, prepared statements are supported, and a query builder auxiliary class can be used to build queries dynamically.

The driver and its dependencies

The Java 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.

Architecture

Cassandra compatibility

The 2.0 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.

	Java driver 1.0.x	Java driver 2.0.x
Cassandra 1.2.x	Compatible	Compatible
Cassandra 2.0.x	Compatible for Cassandra 1.0 API and commands	Compatible

Maven dependencies

The latest release of the driver is available on Maven Central. You can install it in your application using the following Maven dependency:

```
<dependency>
  <groupId>com.datastax.cassandra</groupId>
  <artifactId>cassandra-driver-core</artifactId>
  <version>2.1.5</version>
</dependency>
```

You ought to build your project using the [Mojo Versions plug-in](#). Add the **versions:display-dependency-updates** setting to your POM file, and it lets you know when the driver you are using is out of date during the build process.

Writing your first client

This section walks you through a small sample client application that uses the Java 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

The Java 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
- Eclipse IDE
- Maven 2 Eclipse plug-in

While Eclipse and Maven are not required to use the Java driver to develop Cassandra client applications, they do make things easier. You can use Maven from the command-line, but if you wish to use a different build tool (such as Ant) to run the sample code, you will have to [set up your environment](#) accordingly.

See for more information on setting up your programming environment to do this.

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. In the Eclipse IDE, create a simple Maven project.
Use the following data:
 - groupId: com.example.cassandra
 - artifactId: simple-client
2. Right-click on the simple-client node in the **Project Viewer** and select **Maven > Add Dependency**.
 - a) Enter datastax in the search text field.
You should see several DataStax libraries listed in the Search Results list.
 - b) Expand the com.datastax.cassandra.cassandra-driver-core library and select the version you want.
3. Create a new Java class, com.example.cassandra.SimpleClient.
 - a) Add an instance field, cluster, to hold a Cluster reference.

```
private Cluster cluster;
```

- b) Add an instance method, connect, to your new class.

```
public void connect(String node) {}
```

The connect method:

- adds a contact point (node IP address) using the `Cluster.Builder` auxiliary class
- builds a cluster instance
- retrieves metadata from the cluster

Writing your first client

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

```
public void connect(String node) {
    cluster = Cluster.builder()
        .addContactPoint(node)
        .build();
    Metadata metadata = cluster.getMetadata();
    System.out.printf("Connected to cluster: %s\n",
        metadata.getClusterName());
    for ( Host host : metadata.getAllHosts() ) {
        System.out.printf("Datacenter: %s; Host: %s; Rack: %s\n",
            host.getDatacenter(), host.getAddress(), host.getRack());
    }
}
```

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

```
public void close() {
    cluster.close();
}
```

- d) In the class `main` method instantiate a `SimpleClient` object, call `connect` on it, and close it.

```
public static void main(String[] args) {
    SimpleClient client = new SimpleClient();
    client.connect("127.0.0.1");
    client.close();
}
```

4. Right-click in the `SimpleClient` class editor pane and select **Run As > 1 Java Application** to run the program.

Code listing

The complete code listing illustrates:

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

```
package com.example.cassandra;

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

public class SimpleClient {
    private Cluster cluster;

    public void connect(String node) {
        cluster = Cluster.builder()
            .addContactPoint(node)
            .build();
        Metadata metadata = cluster.getMetadata();
        System.out.printf("Connected to cluster: %s\n",
            metadata.getClusterName());
    }
}
```



```

        for ( Host host : metadata.getAllHosts() ) {
            System.out.printf("Datacenter: %s; Host: %s; Rack: %s\n",
                host.getDatacenter(), host.getAddress(), host.getRack());
        }
    }

    public void close() {
        cluster.close();
    }

    public static void main(String[] args) {
        SimpleClient client = new SimpleClient();
        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:

```

Connected to cluster: xerxes
Datacenter: datacenter1; Host: /127.0.0.1; Rack: rack1
Datacenter: datacenter1; Host: /127.0.0.2; Rack: rack1
Datacenter: datacenter1; Host: /127.0.0.3; Rack: rack1

```

Using a session to execute CQL statements

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 CQL3 schema which is described in a post on the DataStax developer blog. Reading [that post](#), could help with some of the new CQL3 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 Java 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.

```
private Session session;
```

b) Declare and implement a getter for the new session field.

```
public Session getSession() {
    return this.session;
}
```

Writing your first client

```
}
```

- c) 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 multiple connections 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, `createSchema`, to the `SimpleClient` class implementation.

```
public void createSchema() { }
```

3. Add the code to create a new schema.

- a) Execute a statement that creates a new keyspace.

Add to the `createSchema` 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)" +  
    ");");
```

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

```
public void loadData() { }
```

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 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.

```

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

```

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

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

```

System.out.println(String.format("%-30s\t%-20s\t%-20s\n%s", "title",
    "album", "artist",
    "-----+-----"));
+-----+);
for (Row row : results) {
    System.out.println(String.format("%-30s\t%-20s\t%-20s",
        row.getString("title"),
        row.getString("album"), row.getString("artist")));
}
System.out.println();

```

7. In the class `main` method, add a call to the new `createSchema`, `loadData`, and `querySchema` methods.

```

public static void main(String[] args) {
    SimpleClient client = new SimpleClient();
    client.connect("127.0.0.1");
    client.createSchema();
    client.loadData();
    client.querySchema();
    client.close();
}

```

8. Add a call to close the session object in the `close` method.

```

public void close() {
    session.close();
}

```

Writing your first client

```
        cluster.close();
    }
```

Code listing

The complete code listing illustrates:

- creating a keyspace and tables
- inserting data into tables
- querying tables for data

```
package com.example.cassandra;

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

public class SimpleClient {
    private Cluster cluster;
    private Session session;

    public Session getSession() {
        return this.session;
    }

    public void connect(String node) {
        cluster = Cluster.builder()
            .addContactPoint(node)
            .build();
        Metadata metadata = cluster.getMetadata();
        System.out.printf("Connected to cluster: %s\n",
            metadata.getClusterName());
        for ( Host host : metadata.getAllHosts() ) {
            System.out.printf("Datacenter: %s; Host: %s; Rack: %s\n",
                host.getDatacenter(), host.getAddress(), host.getRack());
        }
        session = cluster.connect();
    }

    public void createSchema() {
        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)" +
        ");");
    }

    public void loadData() {
        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-f22b6alb354d," +
                "756716f7-2e54-4715-9f00-91dcbea6cf50," +
                "'La Petite Tonkinoise'," +
                "'Bye Bye Blackbird'," +
                "'Joséphine Baker'" +
            ");");
    }

    public void querySchema() {
        ResultSet results = session.execute("SELECT * FROM simplex.playlists "
+
            "WHERE id = 2cc9ccb7-6221-4ccb-8387-f22b6alb354d;");
        System.out.println(String.format("%-30s\t%-20s\t%-20s\n%s", "title",
"album", "artist",
"-----+-----"));
        for (Row row : results) {
            System.out.println(String.format("%-30s\t%-20s\t%-20s",
row.getString("title"),
row.getString("album"), row.getString("artist")));
        }
        System.out.println();
    }

    public void close() {
        session.close();
        cluster.close();
    }

    public static void main(String[] args) {
        SimpleClient client = new SimpleClient();
        client.connect("127.0.0.1");
        client.createSchema();
        client.loadData();
        client.querySchema();
        client.close();
    }
}

```

Using bound statements

The previous tutorial used simple CQL statements to read and write data, but you can also use prepared statements, which only need to be parsed once by the cluster, and then bind values to the variables and execute the bound statement you read or write data to a cluster.

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. Override the `loadData` method and implement it.

```
public void loadData() { }
```

2. Add code to prepare an `INSERT` statement.

You get a prepared statement by calling the `prepare` method on your session.

```
PreparedStatement statement = getSession().prepare(
    "INSERT INTO simplex.songs " +
    "(id, title, album, artist, tags) " +
    "VALUES (?, ?, ?, ?, ?);");
```

3. 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..

```
BoundStatement boundStatement = new BoundStatement(statement);
Set<String> tags = new HashSet<String>();
tags.add("jazz");
tags.add("2013");
getSession().execute(boundStatement.bind(
    UUID.fromString("756716f7-2e54-4715-9f00-91dcbea6cf50"),
    "La Petite Tonkinoise'",
    "Bye Bye Blackbird'",
    "Joséphine Baker",
    tags ) );
```

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

4. Add code for the other two insertions into the `simplex.songs` table.
5. Add code to create a new bound statement for inserting data into the `simplex.playlists` table.

```
statement = getSession().prepare(
    "INSERT INTO simplex.playlists " +
    "(id, song_id, title, album, artist) " +
    "VALUES (?, ?, ?, ?, ?);");
boundStatement = new BoundStatement(statement);
getSession().execute(boundStatement.bind(
```

```
UUID.fromString("2cc9ccb7-6221-4ccb-8387-f22b6a1b354d"),  
UUID.fromString("756716f7-2e54-4715-9f00-91dcbea6cf50"),  
"La Petite Tonkinoise",  
"Bye Bye Blackbird",  
"Joséphine Baker") );
```

6. Add a call in the class main method to loadData.

```
public static void main(String[] args) {  
    BoundStatementsClient client = new BoundStatementsClient();  
    client.connect("127.0.0.1");  
    client.createSchema();  
    client.loadData();  
    client.querySchema();  
    client.updateSchema();  
    client.dropSchema();  
    client.close();  
}
```

Java driver reference

Reference for the Java driver.

Four simple rules for coding with the driver

When writing code that uses the driver, there are **four simple rules** that you should follow that will also make your code efficient:

- Use one cluster instance per (physical) cluster (per application lifetime)
- Use at most one session instance per keyspace, or use a single Session and explicitly specify the keyspace in your queries
- If you execute a statement more than once, consider using a prepared statement
- You can reduce the number of network roundtrips and also have atomic operations by using batches

Cluster

A Cluster instance allows to configure different important aspects of the way connections and queries will be handled. At this level you can configure everything from contact points (address of the nodes to be contacted initially before the driver performs node discovery), the request routing policy, retry and reconnection policies, and so forth. Generally such settings are set once at the application level.

```
Cluster cluster = Cluster.builder()
    .addContactPoint("10.1.1.3", "10.1.1.4", "10.1.1.5")
    .withLoadBalancingPolicy(new DCAwareRoundRobinPolicy("US_EAST"))
    .build();
cluster.getConfiguration()
    .getProtocolOptions()
    .setCompression(ProtocolOptions.Compression.LZ4);
```

Session

While the session instance is centered around query execution, the Session it also manages the per-node connection pools. The session instance is a long-lived object, and it should not be used in a request-response, short-lived fashion. The code should share the same cluster and session instances across your application.

Prepared statements

Using prepared statements provides multiple benefits. A prepared statement is parsed and prepared on the Cassandra nodes and is ready for future execution. When binding parameters, only these (and the query id) are sent over the wire. These performance gains add up when using the same queries (with different parameters) repeatedly.

Batch operations

The **BATCH** statement combines multiple data modification statements (INSERT, UPDATE, DELETE) into a single logical operation which is sent to the server in a single request. Also batching together multiple operations ensures these are executed in an atomic way: either all succeed or none.

To make the best use of BATCH, read about **atomic batches** in Cassandra 1.2 and **static columns** and **batching of conditional updates**.

For more information see [this blog post](#) on the four simple rules.

Asynchronous I/O

Use asynchronous methods to execute CQL statements so that your client does not block.

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 `executeAsync` method.

About this task

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-cassandra-client` project. It should extend the `SimpleClient` class.

```
package com.example.cassandra;

public class AsynchronousExample extends SimpleClient {
}
```

2. Add an instance method, `getRows`, and implement it.
 - a) Implement the `getRows` method so it returns a `ResultSetFuture` object.

```
public ResultSetFuture getRows() {}
```

The `ResultSetFuture` class implements the `java.util.concurrent.Future<V>` interface. Objects which implement this interface allow for non-blocking computation. The calling code may wait for the completion of the computation or to check if it is done.

- b) Using the `QueryBuilder` class, build a `SELECT` query that returns all the rows for the `song` table for all columns.

```
Statement statement = QueryBuilder.select().all().from("simplex",
    "songs");
```

- c) Execute the query asynchronously and return the `ResultSetFuture` object.

```
return getSession().executeAsync(query);
```

3. Add a class method, `main`, to your class implementation and add calls to create the schema, load the data, and then query it using the `getRows` method.

```
public static void main(String[] args) {
    AsynchronousExample client = new AsynchronousExample();
    client.connect("127.0.0.1");
    client.createSchema();
    client.loadData();
    ResultSetFuture results = client.getRows();
    for (Row row : results.getUninterruptibly()) {
        System.out.printf( "%s: %s / %s\n",
            row.getString("artist"),
            row.getString("title"),
            row.getString("album") );
    }
    client.dropSchema("simplex");
    client.close();
}
```

```
}
```

Of course, in our implementation, the call to `getUninterruptibly` blocks until the result set future has completed execution of the statement on the session object. Functionally it is no different from executing the `SELECT` query synchronously.

AsynchronousExample code listing

```
package com.example.cassandra;

import com.datastax.driver.core.Query;
import com.datastax.driver.core.ResultSetFuture;
import com.datastax.driver.core.Row;
import com.datastax.driver.core.querybuilder.QueryBuilder;

public class AsynchronousExample extends SimpleClient {
    public AsynchronousExample() {
    }

    public ResultSetFuture getRows() {
        Query query = QueryBuilder.select().all().from("simplex", "songs");
        return getSession().executeAsync(query);
    }

    public static void main(String[] args) {
        AsynchronousExample client = new AsynchronousExample();
        client.connect("127.0.0.1");
        client.createSchema();
        client.loadData();
        ResultSetFuture results = client.getRows();
        for (Row row : results.getUninterruptibly()) {
            System.out.printf( "%s: %s / %s\n",
                row.getString("artist"),
                row.getString("title"),
                row.getString("album") );
        }
        client.dropSchema("simplex");
        client.close();
    }
}
```

Automatic failover

If a node fails to respond, the driver tries other ones in the cluster.

If a Cassandra node fails or becomes unreachable, the Java driver automatically and transparently tries other nodes in the cluster and schedules reconnections to the dead nodes in the background.

Description

How the driver handles failover is determined by which retry and reconnection policies are used when building a cluster object.

Examples

This code illustrates building a cluster instance with a retry policy which sometimes retries with a lower consistency level than the one specified for the query.

```
public RollYourOwnCluster() {
    Cluster cluster = Cluster.builder()
        .addContactPoints("127.0.0.1", "127.0.0.2")
        .withRetryPolicy(DowngradingConsistencyRetryPolicy.INSTANCE)
        .withReconnectionPolicy(new ConstantReconnectionPolicy(100L))
        .build();
    session = cluster.connect();
}
```

BATCH statements

Use BATCH statements to group together two or more CQL statements for execution.

Because the new data model introduced with CQL (version 3) breaks wide rows into several CQL rows, it's common for applications to require to batch multiple `INSERT` statements. Naturally, CQL comes with a `BEGIN BATCH ... APPLY BATCH` statement that allows you to group together several `INSERT` statements, so that you can build a string of such a batch request and execute it.

```
public class BatchClient extends SimpleClient {
    public void loadData() {
        getSession().execute(
            "BEGIN BATCH USING TIMESTAMP " +
            "    INSERT INTO simplex.songs (id, title, album, artist)
VALUES (" +
            UUID.randomUUID() +
            ", 'Poulaillers' Song', 'Jamais content', 'Alain
Souchon'); " +
            "    INSERT INTO simplex.songs (id, title, album, artist)
VALUES (" +
            UUID.randomUUID() +
            ", 'Bonnie and Clyde', 'Bonnie and Clyde', 'Serge
Gainsbourg'); " +
            "    INSERT INTO simplex.songs (id, title, album, artist)
VALUES (" +
            UUID.randomUUID() +
            ", 'Lighthouse Keeper', 'A Clockwork Orange', 'Erika
Eigen'); " +
            "APPLY BATCH"
        );
    }
}
```

Batching prepared statements

Prepared statements can be batched together as well as simple statements. Prepared statements are useful for queries that are frequently executed in an application with different values, because they reduce the network traffic and the overall processing time on both clients and servers by making it possible to send only the values along with the identifier of the prepared statement to the server.

```
public class BatchClient extends SimpleClient {
    public void loadData() {
        PreparedStatement insertPreparedStatement = getSession().prepare(
            "BEGIN BATCH USING TIMESTAMP " + timestamp +
```

```

        "    INSERT INTO simplex.songs (id, title, album, artist) " +
        "VALUES (?, ?, ?, ?); " +
        "    INSERT INTO simplex.songs (id, title, album, artist) " +
        "VALUES (?, ?, ?, ?); " +
        "    INSERT INTO simplex.songs (id, title, album, artist) " +
        "VALUES (?, ?, ?, ?); " +
        "APPLY BATCH"
    );

    getSession().execute(
        insertPreparedStatement.bind(
            UUID.randomUUID(), "Seaside Rendezvous", "A Night at
the Opera", "Queen",
            UUID.randomUUID(), "Entre Nous", "Permanent Waves",
            "Rush",
            UUID.randomUUID(), "Frank Sinatra", "Fashion
Nugget", "Cake"
        ));
    }
}

```

Note: You cannot refer to bind variables in batched statements by name, but by position.

This does not work for all cases where you wish to use a batched prepared statement, because you need to know the number of statements in the batch up front. Since version 2.0 of the driver, you can build batched statements using individual prepared (or simple) statements and adding them individually.

```

public class BatchClient extends SimpleClient {
    public void loadData() {
        PreparedStatement insertSongPreparedStatement =
        getSession().prepare(
            "INSERT INTO simplex.songs (id, title, album, artist) VALUES
            (?, ?, ?, ?);");
        batch.add(insertSongPreparedStatement.bind(
            UUID.randomUUID(), "Die Möschi", "In Gold", "Willi
Ostermann"));
        batch.add(insertSongPreparedStatement.bind(
            UUID.randomUUID(), "Memo From Turner", "Performance", "Mick
Jagger"));
        batch.add(insertSongPreparedStatement.bind(
            UUID.randomUUID(), "La Petite Tonkinoise", "Bye Bye
Blackbird", "Joséphine Baker"));
        getSession().execute(batch);
    }
}

```

Timestamps

Due to how Cassandra stores data, it is possible to have multiple results for a column in a row, and timestamps are used to determine which is the most current one. These timestamps are generated by default by the server or can be specified on a statement or on a batch programmatically.

```

public class BatchClient extends SimpleClient {
    public void loadData() {
        long timestamp = new Date().getTime();
        getSession().execute(
            "BEGIN BATCH USING TIMESTAMP " + timestamp +
            "    INSERT INTO simplex.songs (id, title, album, artist)
VALUES (" +
            UUID.randomUUID() +

```

```

        ", 'Poulaillers' Song', 'Jamais content', 'Alain
Souchon'); " +
        "    INSERT INTO simplex.songs (id, title, album, artist)
VALUES (" +
        UUID.randomUUID() +
        ", 'Bonnie and Clyde', 'Bonnie and Clyde', 'Serge
Gainsbourg'); " +
        "    INSERT INTO simplex.songs (id, title, album, artist)
VALUES (" +
        UUID.randomUUID() +
        ", 'Lighthouse Keeper', 'A Clockwork Orange', 'Erika
Eigen'); " +
        "APPLY BATCH"
    );
}
}

```

Cluster configuration

Configure the cluster with different tuning policies and connection options.

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

The load balancing policy interface consists of three methods:

- `HostDistance distance(Host host)`: determines the distance to the specified host. The values are `HostDistance.IGNORED`, `LOCAL`, and `REMOTE`.
- `void init(Cluster cluster, Collection<Host> hosts)`: initializes the policy. The driver calls this method only once and before any other method calls are made.
- `Iterator<Host> newQueryPlan()`: returns the hosts to use for a query. Each new query calls this method.

The policy also implements the `Host.StateListener` interface which is for tracking node events (that is add, down, remove, and up).

By default, the driver uses a round robin load balancing policy when building a cluster object. There is also a token-aware policy which allows the ability to prefer the replica for a query as coordinator. The driver includes these three policy classes:

- `DCAwareRoundRobinPolicy`
- `RoundRobinPolicy`
- `TokenAwarePolicy`

Reconnection policy

The reconnection policy determines how often a reconnection to a dead node is attempted.

Description

The reconnection policy consists of one method:

- `ReconnectionPolicy.ReconnectionSchedule newSchedule()`: creates a new schedule to use in reconnection attempts.

By default, the driver uses an exponential reconnection policy. The driver includes these two policy classes:

- `ConstantReconnectionPolicy`
- `ExponentialReconnectionPolicy`

Retry policy

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

Description

A client may send requests to any node in a cluster whether or not it is a replica of the data being queried. This node is placed into the coordinator role temporarily. Which node is the coordinator is determined by the load balancing policy for the cluster. The coordinator is responsible for routing the request to the appropriate replicas. If a coordinator fails during a request, the driver connects to a different node and retries the request. If the coordinator knows before a request that a replica is down, it can throw an `UnavailableException`, but if the replica fails after the request is made, it throws a `TimeoutException`. Of course, this all depends on the consistency level set for the query before executing it.

A retry policy centralizes the handling of query retries, minimizing the need for catching and handling of exceptions in your business code.

The retry policy interface consists of three methods:

- `RetryPolicy.RetryDecision onReadTimeout(Statement statement, ConsistencyLevel cl, int requiredResponses, int receivedResponses, boolean dataRetrieved, int nbRetry)`
- `RetryPolicy.RetryDecision onUnavailable(Statement statement, ConsistencyLevel cl, int requiredReplica, int aliveReplica, int nbRetry)`
- `RetryPolicy.RetryDecision onWriteTimeout(Statement statement, ConsistencyLevel cl, WriteType writeType, int requiredAcks, int receivedAcks, int nbRetry)`

By default, the driver uses a default retry policy. The driver includes these four policy classes:

- `DefaultRetryPolicy`
- `DowngradingConsistencyRetryPolicy`
- `FallthroughRetryPolicy`
- `LoggingRetryPolicy`

Connection options

There are three classes the driver uses to configure node connections.

Protocol options

Protocol options configure the port on which to connect to a Cassandra node and which type of compression to use.

Description

Table 1: Protocol options

Option	Description	Default
port	The port to connect to a Cassandra node on.	9042
compression	What kind of compression to use when sending data to a node: either no compression or snappy. Snappy compression is optimized for high speeds and reasonable compression.	ProtocolOptions.Compression.NONE

Pooling options

The Java driver uses connections asynchronously, so multiple requests can be submitted on the same connection at the same time.

Description

The driver only needs to maintain a relatively small number of connections to each Cassandra host. These options allow you to control how many connections are kept exactly. The defaults should be fine for most applications.

Table 2: Connection pooling options

Option	Description	Default value
coreConnectionsPerHost	The core number of connections per host.	2 for HostDistance.LOCAL, 1 for HostDistance.REMOTE
maxConnectionPerHost	The maximum number of connections per host.	8 for HostDistance.LOCAL, 2 for HostDistance.REMOTE
maxSimultaneousRequestsPerConnectionThreshold	The number of simultaneous requests on all connections to an host after which more connections are created.	128
minSimultaneousRequestsPerConnectionThreshold	The number of simultaneous requests on a connection below which connections in excess are reclaimed.	25

Socket options

Socket options configure the low-level sockets used to connect to nodes.

Description

All but one of these socket options comes from the Java runtime library's `SocketOptions` class. The `connectTimeoutMillis` option though is from Netty's `ChannelConfig` class.

Table 3: Pooling options

Option	Corresponds to	Description
connectTimeoutMillis	org.jboss.netty.channel.ChannelConnector. "connectTimeoutMillis"	The connect timeout in milliseconds for the underlying Netty channel.
keepAlive	java.net.SocketOptions.SO_KEEPALIVE	
receiveBufferSize	java.net.SocketOptions.SO_RCVBUF	Hint on the size of the buffer used to receive data.
reuseAddress	java.net.SocketOptions.SO_REUSEADDR	Whether to allow the same port to be bound to multiple times.
sendBufferSize	java.net.SocketOptions.SO_SNDBUF	Hint on the size of the buffer used to send data.
soLinger	java.net.SocketOptions.SO_LINGER	When specified, disables the immediate return from a call to close() on a TCP socket.
tcpNoDelay	java.net.SocketOptions.TCP_NODELAY	Disables Nagle's algorithm on the underlying socket.

Connection requirements

Requirements for connecting to a cluster.

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 Java types

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

Description

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

Table 4: Java classes to CQL data types

CQL3 data type	Java type
ascii	java.lang.String
bigint	long

CQL3 data type	Java type
blob	java.nio.ByteBuffer
boolean	boolean
counter	long
decimal	java.math.BigDecimal
double	double
float	float
inet	java.net.InetAddress
int	int
list	java.util.List<T>
map	java.util.Map<K, V>
set	java.util.Set<T>
text	java.lang.String
timestamp	java.util.Date
timeuuid	java.util.UUID
tuple	com.datastax.driver.core.TupleType
uuid	java.util.UUID
varchar	java.lang.String
varint	java.math.BigInteger

CQL statements

CQL statements are represented by the `Statement` class.

You can configure a `Statement` by.

- enabling or disabling tracing
- setting
 - the consistency level
 - retry policy
 - fetch size
 - routing key

Examples

You set these options on the statement object before execution. The following example shows how to set the consistency level and the fetch size on two statements before they are executed.

```
public void querySchema() {
    // setting the consistency level on a statement
    Statement statement = new SimpleStatement("SELECT * FROM simplex.songs "
+
    "WHERE id = 2cc9ccb7-6221-4ccb-8387-f22b6a1b354d;");
```

```

System.out.printf( "Default consistency level = %s\n",
statement.getConsistencyLevel() );
statement.setConsistencyLevel(ConsistencyLevel.ONE);
System.out.printf( "New consistency level = %s\n",
statement.getConsistencyLevel() );
for ( Row row : getSession().execute(statement) ) {
    System.out.printf( "%s %s\n", row.getString("artist"),
row.getInt("title"));
}
// setting the fetch size on a statement
// the following SELECT statement returns over 5K results
statement = new SimpleStatement("SELECT * FROM lexicon.concordance;");
statement.setFetchSize(100);
results = getSession().execute(statement);
for ( Row row : results ) {
    System.out.printf( "%s: %d\n", row.getString("word"),
row.getInt("occurrences"));
}
}

```

Building dynamic queries programmatically with the `QueryBuilder` API

Overview of the `QueryBuilder` API

The `QueryBuilder` API allows you to create dynamic CQL queries programmatically. The `QueryBuilder` API encapsulates the syntax of CQL queries, providing classes and methods to construct and execute queries without having to write CQL. This is more secure, as the queries are not subject to injection attacks.

The `QueryBuilder` API classes are located in the `com.datastax.driver.core.querybuilder` package.

Creating a query using the `QueryBuilder` API

To create a query, create a `com.datastax.driver.core.Statement` object by calling one of the `com.datastax.driver.core.querybuilder.QueryBuilder` methods. These methods return subclasses of `Statement`.

Table 5: `QueryBuilder` methods

Method	Description
<code>QueryBuilder.select</code>	Equivalent to the CQL <code>SELECT</code> statement. Returns a <code>Select</code> object.
<code>QueryBuilder.insert</code>	Equivalent to the CQL <code>INSERT</code> statement. Returns an <code>Insert</code> object.
<code>QueryBuilder.update</code>	Equivalent to the CQL <code>UPDATE</code> statement. Returns an <code>Update</code> object.
<code>QueryBuilder.delete</code>	Equivalent to the CQL <code>DELETE</code> statement. Returns a <code>Delete</code> object.

The `Statement` object can then be configured with the conditions of the query.

For example, the following line returns a `Select` object for a particular keyspace and table.

```
Select select = QueryBuilder.select().from("keyspace_name", "table_name");
```

The statement can then be executed similar to a hand-coded CQL query string using a `Session` instance's `execute` method, with the results returned as a `ResultSet` object.

```
Cluster cluster = Cluster.builder()
    .addContactPoint("localhost")
    .build();
Session session = cluster.connect();
Select select = QueryBuilder.select()
    .all()
    .from("music", "playlist");
ResultSet results = session.execute(select);
```

Setting query options on Statement instances

`Statement` objects represent a particular query. The `Statement` class has methods that allow you to set query options such as the consistency level, tracing options, fetch size, or the retry policy of the query.

To enable tracing on the query, call `Statement.enableTracing()`. `Statement.disableTracing()` disables tracing of the query.

The consistency level of the query can be set by calling `Statement.setConsistencyLevel()` and setting it to one of the enums of `com.datastax.driver.core.ConsistencyLevel`, which correspond to the [consistency levels of Cassandra](#).

```
Statement statement = ...
statement.setConsistencyLevel(ConsistencyLevel.LOCAL_QUORUM);
```

The serial consistency level of the query, set with `Statement.setSerialConsistencyLevel()`, is similar to the consistency level, but only used in queries that have conditional updates. That is, it is only used in queries that have the equivalent of the `IF` condition in CQL.

The fetch size, or number of rows returned simultaneously by a select query, can be controlled by calling `Select.setFetchSize()` and passing in the number of rows.

```
Select statement = ...
statement.setFetchSize(100);
```

Setting the fetch size is typically needed when queries return extremely large numbers of rows. Setting the fetch size to small values is discouraged, as it results in poor performance.

To disable paging of the results, set the fetch size to `Integer.MAX_VALUE`. If the fetch size is set to a value less than or equal to 0, the default fetch size will be used.

The retry policy defines the default behavior of queries when they encounter a `TimeoutException` or `UnavailableException`. If not explicitly set, the retry policy is set to the default retry policy for the target cluster, returned by calling `Policies.getRetryPolicy()` in the cluster configuration. To explicitly set the retry policy, call `Statement.setRetryPolicy()` and passing in one of the classes that implement `com.datastax.driver.core.policies.RetryPolicy`.

Table 6: Retry policies

Retry policy	Description
<code>DefaultRetryPolicy</code>	The query will be retried for read requests if enough replica nodes replied but no data was retrieved, or on write requests if there is a timeout while writing the distributed log used by batch statements.
<code>DowngradingConsistencyRetryPolicy</code>	The query may be retried with a lower consistency level than the one initially requested. This policy will retry queries in the same circumstances

Retry policy	Description
	as <code>DefaultRetryPolicy</code> , as well as a few other cases. For a read request, if the number of replica nodes is more than one but lower than the number required by the consistency level, the query is rerun with the lower consistency level. For write requests, if the write type is <code>WriteType.UNLOGGED_BATCH</code> and at least one replica node acknowledged the write, the query is retried at the lower consistency level. If the query fails due to an <code>UnavailableException</code> , and at least one replica node is available, the query is retried at the lower consistency level.
<code>FallthroughRetryPolicy</code>	This policy does not retry queries, but allows the client business logic to implement retries.
<code>LoggingRetryPolicy</code>	This policy allows you to wrap other retry policies in order to log the decision of the policy.

Query builder examples

The query builder class helps to create executable CQL statements dynamically without resorting to building query strings by hand.

Select all example

Here is an example that creates a select all statement from the specified schema and returns all the rows.

```
public List<Row> getRows(String keyspace, String table) {
    Statement statement = QueryBuilder
        .select()
        .all()
        .from(keyspace, table);
    return getSession()
        .execute(statement)
        .all();
}
```

Select particular rows example

This example selects all rows that match some conditions.

```
Select select = QueryBuilder.select()
    .all()
    .distinct()
    .from("addressbook", "contact")
    .where(eq("type", "Friend"))
    .and(eq("city", "San Francisco"));
ResultSet results = session.execute(select);
```

Insert example

This example demonstrates how to insert data into a row.

```
Insert insert = QueryBuilder
    .insertInto("addressbook", "contact")
```

```

        .value("firstName", "Dwayne")
        .value("lastName", "Garcia")
        .value("email", "dwayne@example.com");
    ResultSet results = session.execute(insert);

```

Update example

This example demonstrates how to update the data in a particular row.

```

    Update update = QueryBuilder.update("addressbook", "contact")
        .with(set("email", "dwayne.garcia@example.com"))
        .where(eq("username", "dgarcia"));
    ResultSet results = session.execute(update);

```

Delete example

This example demonstrates how to delete a particular row.

```

    Delete delete = QueryBuilder.delete()
        .from("addressbook", "contact")
        .where(eq("username", "dgarcia"));

```

Debugging

Various options for debugging your client application.

You have several options to help in debugging your application.

On the client side, you can:

- use the Eclipse debugger
- monitor different metrics
 - some by default
 - others by implementing your own metric objects
- log using a SLF4J-compliant logging library

On the server side, you can:

- enable tracing
- adjusting the log4j configuration for finer logging granularity

If you are using DataStax Enterprise, you can [store the log4j messages](#) into a Cassandra cluster.

Exceptions

Handling exceptions.

Overview

Many of the exceptions for the Java driver are runtime exceptions, meaning that you do not have to wrap your driver-specific code in try/catch blocks or declared thrown exceptions as part of your method signatures.

Example

The code in this listing illustrates catching the various exceptions thrown by a call to execute on a session object.

```
public void querySchema() {
    Statement statement = new SimpleStatement(
        "INSERT INTO simplex.songs " +
        "(id, title, album, artist) " +
        "VALUES (da7c6910-a6a4-11e2-96a9-4db56cdc5fe7," +
        "'Golden Brown', 'La Folie', 'The Stranglers' " +
        ");");
    try {
        getSession().execute(statement);
    } catch (NoHostAvailableException e) {
        System.out.printf("No host in the %s cluster can be contacted to
execute the query.\n",
            getSession().getCluster());
    } catch (QueryExecutionException e) {
        System.out.println("An exception was thrown by Cassandra because it
cannot " +
            "successfully execute the query with the specified consistency
level.");
    } catch (QueryValidationException e) {
        System.out.printf("The query %s \nis not valid, for example,
incorrect syntax.\n",
            statement.getQueryString());
    } catch (IllegalStateException e) {
        System.out.println("The BoundStatement is not ready.");
    }
}
```

Monitoring

The Java driver uses the Metrics library (and JMX) to allow for monitoring of your client application.

Description

The following links are for your reference:

- [Metrics library](#)
- [Java Management Extensions \(JMX\)](#)

The Metrics object exposes the following metrics:

Table 7: Metrics

Metric	Description
connectedToHosts	A gauge that presents the number of hosts that are currently connected to (at least once).
errorMetrics	An Error.Metrics object which groups errors which have occurred.
knownHosts	A gauge that presents the number of nodes known by the driver, regardless of whether they are currently up or down).
openConnections	A gauge that presents the total number of connections to nodes.

Metric	Description
requestsTimer	A timer that presents metrics on requests executed by the user.

In addition, you can register your own Metrics objects by extending the `com.codahale.metrics.Metric` interface and registering it with the metric registry. Metrics are enabled by default, but you can also disable them.

Examples

You retrieve the metrics object from your cluster.

```
public void printMetrics() {
    System.out.println("Metrics");
    Metrics metrics = getSession().getCluster().getMetrics();
    Gauge<Integer> gauge = metrics.getConnectedToHosts();
    Integer numberOfHosts = gauge.value();
    System.out.printf("Number of hosts: %d\n", numberOfHosts);
    Metrics.Errors errors = metrics.getErrorMetrics();
    Counter counter = errors.getReadTimeouts();
    System.out.printf("Number of read timeouts: %d\n", counter.count());
    Timer timer = metrics.getRequestsTimer();
    Timer.Context context = timer.time();
    try {
        long numberUserRequests = timer.getCount();
        System.out.printf("Number of user requests: %d\n",
            numberUserRequests);
    } finally {
        context.stop();
    }
}
```

You can create and register your own Metrics objects:

```
public void createNumberInsertsGauge() {
    Metric ourMetric = getSession()
        .getCluster()
        .getMetrics()
        .getRegistry()
        .getMetrics()
        .get(MetricRegistry.name(getClass(), "numberInserts"));
    System.out.printf("Number of insert statements executed: %5d\n",
        ((Gauge<?>) ourMetric).value());
}
```

The Metrics library exposes its metrics objects as JMX managed beans (MBeans). You can gather metrics from your client application from another application for reporting purposes. This example presumes that a user-defined metric has been registered by the client as in the previous example.

```
public void connectMBeanServer() {
    try {
        JMXServiceURL url = new JMXServiceURL("service:jmx:rmi:///jndi/
            rmi://:9999/jmxrmi");
        JMXConnector jmxc = JMXConnectorFactory.connect(url, null);
        MBeanServerConnection mbsConnection = jmxc.getMBeanServerConnection();
        ObjectName objectName = new ObjectName("cluster1-metrics:" +
            "name=com.example.cassandra.MetricsExample.numberInserts");
    }
}
```

```
        JmxGaugeMBean mBean = JMX.newMBeanProxy(mbsConnection, objectName,
JmxGaugeMBean.class);
        System.out.printf("Number of inserts: %5d\n", mBean.getValue());
    } catch (MalformedURLException mue) {
        mue.printStackTrace();
    } catch (IOException ioe) {
        ioe.printStackTrace();
    } catch (NullPointerException npe) {
        npe.printStackTrace();
    }
}
```

When running the previous example, you must pass the following properties to the JVM running your client application:

```
-Dcom.sun.management.jmxremote.port=9999 \
-Dcom.sun.management.jmxremote.authenticate=false \
-Dcom.sun.management.jmxremote.ssl=false
```

Enabling tracing

To help you to understand how Cassandra functions when executing statements, you enable tracing.

About this task

Tracing is only enabled on a per-query basis. The sample client described here extends the SimpleClient class and then enables tracing on two queries: an INSERT of data and a SELECT that returns all the rows in a table.

Procedure

1. Add a new class, TracingExample, to your simple-cassandra-client project. It should extend the SimpleClient class.

```
package com.example.cassandra;

public class TracingExample extends SimpleClient {
}
```

2. Add an instance method, traceInsert and implement it.
 - a) Build an INSERT query and enable tracing on it.

```
/* INSERT INTO simplex.songs
 * (id, title, album, artist)
 * VALUES (da7c6910-a6a4-11e2-96a9-4db56cdc5fe7,
 *         'Golden Brown', 'La Folie', 'The Stranglers'
 * );
 */
Statement insert = QueryBuilder.insertInto("simplex", "songs")
    .value("id", UUID.randomUUID())
    .value("title", "Golden Brown")
    .value("album", "La Folie")
    .value("artist", "The Stranglers")
    .setConsistencyLevel(ConsistencyLevel.ONE).enableTracing();
```

The code illustrates using the QueryBuilder class and method chaining.

- b) Execute the INSERT query and retrieve the execution tracing information.

```
ResultSet results = getSession().execute(insert);
ExecutionInfo executionInfo = results.getExecutionInfo();
```

- c) Print out the information retrieved.

```
System.out.printf( "Host (queried): %s\n",
executionInfo.getQueriedHost().toString() );
for (Host host : executionInfo.getTriedHosts()) {
    System.out.printf( "Host (tried): %s\n", host.toString() );
}
QueryTrace queryTrace = executionInfo.getQueryTrace();
System.out.printf("Trace id: %s\n\n", queryTrace.getTraceId());
System.out.println("-----
+-----+-----+-----");
for (QueryTrace.Event event : queryTrace.getEvents()) {
    System.out.printf("%38s | %12s | %10s | %12s\n",
event.getDescription(),
millis2Date(event.getTimestamp()),
event.getSource(), event.getSourceElapsedMicros());
}
```

The code prints out information similar to that which cqlsh does when tracing has been enabled. The difference is that in cqlsh tracing is enabled and all subsequent queries prints out tracing information. In the Java driver, a session is basically stateless with respect to tracing which must be enabled on a per-query basis.

- d) Implement a private instance method to format the timestamp to be human-readable.

```
private SimpleDateFormat format = new
SimpleDateFormat("HH:mm:ss.SSS");

private Object millis2Date(long timestamp) {
    return format.format(timestamp);
}
```

3. Add another instance method, traceSelect, and implement it.

```
public void traceSelect() {
    Statement scan = new SimpleStatement("SELECT * FROM
simplex.songs;");
    ExecutionInfo executionInfo =
getSession().execute(scan.enableTracing()).getExecutionInfo();
    System.out.printf( "Host (queried): %s\n",
executionInfo.getQueriedHost().toString() );
    for (Host host : executionInfo.getTriedHosts()) {
        System.out.printf( "Host (tried): %s\n", host.toString() );
    }
    QueryTrace queryTrace = executionInfo.getQueryTrace();
    System.out.printf("Trace id: %s\n\n", queryTrace.getTraceId());
    System.out.printf("%-38s | %-12s | %-10s | %-12s\n", "activity",
"timestamp", "source", "source_elapsed");
    System.out.println("-----
+-----+-----+-----");
    for (QueryTrace.Event event : queryTrace.getEvents()) {
        System.out.printf("%38s | %12s | %10s | %12s\n",
event.getDescription(),
millis2Date(event.getTimestamp()),
```

```

        event.getSource(), event.getSourceElapsedMicros());
    }
    scan.disableTracing();
}

```

Instead of using `QueryBuilder`, this code uses `SimpleStatement`.

4. Add a class method `main` and implement it.

```

public static void main(String[] args) {
    TracingExample client = new TracingExample();
    client.connect("127.0.0.1");
    client.createSchema();
    client.traceInsert();
    client.traceSelect();
    client.resetSchema();
    client.close();
}

```

Code listing

Complete code listing which illustrates:

- Enabling tracing on:
 - INSERT
 - SELECT
- Retrieving execution information and printing it out.

```

package com.example.cassandra;

import java.text.SimpleDateFormat;
import java.util.UUID;

import com.datastax.driver.core.ConsistencyLevel;
import com.datastax.driver.core.ExecutionInfo;
import com.datastax.driver.core.Host;
import com.datastax.driver.core.QueryTrace;
import com.datastax.driver.core.ResultSet;
import com.datastax.driver.core.SimpleStatement;
import com.datastax.driver.core.Statement;
import com.datastax.driver.core.querybuilder.QueryBuilder;

public class TracingExample extends SimpleClient {
    private SimpleDateFormat format = new SimpleDateFormat("HH:mm:ss.SSS");

    public TracingExample() {
    }

    public void traceInsert() {
        /* INSERT INTO simplex.songs
        *   (id, title, album, artist)
        *   VALUES (da7c6910-a6a4-11e2-96a9-4db56cdc5fe7,
        *           'Golden Brown', 'La Folie', 'The Stranglers'
        *   );
        */
        Statement insert = QueryBuilder.insertInto("simplex", "songs")
            .value("id", UUID.randomUUID())
            .value("title", "Golden Brown")
            .value("album", "La Folie")
            .value("artist", "The Stranglers")

```

```

        .setConsistencyLevel(ConsistencyLevel.ONE).enableTracing();
        ResultSet results = getSession().execute(insert);
        ExecutionInfo executionInfo = results.getExecutionInfo();
        System.out.printf( "Host (queried): %s\n",
executionInfo.getQueriedHost().toString() );
        for (Host host : executionInfo.getTriedHosts()) {
            System.out.printf( "Host (tried): %s\n", host.toString() );
        }
        QueryTrace queryTrace = executionInfo.getQueryTrace();
        System.out.printf("Trace id: %s\n\n", queryTrace.getTraceId());
        System.out.printf("%-38s | %-12s | %-10s | %-12s\n", "activity",
"timestamp", "source", "source_elapsed");
        System.out.println("-----
+-----+-----+-----");
        for (QueryTrace.Event event : queryTrace.getEvents()) {
            System.out.printf("%38s | %12s | %10s | %12s\n",
event.getDescription(),
                millis2Date(event.getTimestamp()),
                event.getSource(), event.getSourceElapsedMicros());
        }
        insert.disableTracing();
    }

    public void traceSelect() {
        Statement scan = new SimpleStatement("SELECT * FROM simplex.songs;");
        ExecutionInfo executionInfo =
getSession().execute(scan.enableTracing()).getExecutionInfo();
        System.out.printf( "Host (queried): %s\n",
executionInfo.getQueriedHost().toString() );
        for (Host host : executionInfo.getTriedHosts()) {
            System.out.printf( "Host (tried): %s\n", host.toString() );
        }
        QueryTrace queryTrace = executionInfo.getQueryTrace();
        System.out.printf("Trace id: %s\n\n", queryTrace.getTraceId());
        System.out.printf("%-38s | %-12s | %-10s | %-12s\n", "activity",
"timestamp", "source", "source_elapsed");
        System.out.println("-----
+-----+-----+-----");
        for (QueryTrace.Event event : queryTrace.getEvents()) {
            System.out.printf("%38s | %12s | %10s | %12s\n",
event.getDescription(),
                millis2Date(event.getTimestamp()),
                event.getSource(), event.getSourceElapsedMicros());
        }
        scan.disableTracing();
    }

    private Object millis2Date(long timestamp) {
        return format.format(timestamp);
    }

    public static void main(String[] args) {
        TracingExample client = new TracingExample();
        client.connect("127.0.0.1");
        client.createSchema();
        client.traceInsert();
        client.traceSelect();
        client.close();
    }
}

```

Output:

```

Connected to cluster: xerxes
Simplex keyspace and schema created.
Host (queried): /127.0.0.1
Host (tried): /127.0.0.1
Trace id: 96ac9400-a3a5-11e2-96a9-4db56cdc5fe7

```

activity	timestamp	source
source_elapsed		
28	12:17:16.736	/127.0.0.1
199	12:17:16.736	/127.0.0.1
348	12:17:16.736	/127.0.0.1
788	12:17:16.736	/127.0.0.1
805	12:17:16.736	/127.0.0.1
828	12:17:16.736	/127.0.0.1
848	12:17:16.736	/127.0.0.1
900	12:17:16.736	/127.0.0.1
34	12:17:16.737	/127.0.0.2
25	12:17:16.737	/127.0.0.3
672	12:17:16.737	/127.0.0.2
525	12:17:16.737	/127.0.0.3
692	12:17:16.737	/127.0.0.2
541	12:17:16.737	/127.0.0.3
741	12:17:16.737	/127.0.0.2
583	12:17:16.737	/127.0.0.3
751	12:17:16.737	/127.0.0.3
950	12:17:16.738	/127.0.0.2
178	12:17:16.738	/127.0.0.1
1189	12:17:16.738	/127.0.0.2
249	12:17:16.738	/127.0.0.1
345	12:17:16.738	/127.0.0.1
377	12:17:16.738	/127.0.0.1

```

Connected to cluster: xerxes
Host (queried): /127.0.0.3

```

```

Host (tried): /127.0.0.3
Trace id: da7c6910-a6a4-11e2-96a9-4db56cdc5fe7

activity                                     | timestamp      | source         |
source_elapsed                               +-----+-----+
-----+-----+-----+
35          Parsing statement                | 12:49:34.497  | /127.0.0.3    |
191         Peparing statement                | 12:49:34.497  | /127.0.0.3    |
342        Determining replicas to query     | 12:49:34.497  | /127.0.0.3    |
1561       Sending message to /127.0.0.2    | 12:49:34.498  | /127.0.0.3    |
37         Message received from /127.0.0.3 | 12:49:34.499  | /127.0.0.2    |
2880       Message received from /127.0.0.2 | 12:49:34.499  | /127.0.0.3    |
Executing seq scan across 0 sstables
for [min(-9223372036854775808),
min(-9223372036854775808)]                | 12:49:34.499  | /127.0.0.2    |
580        Scanned 0 rows and matched 0     | 12:49:34.499  | /127.0.0.2    |
648        Enqueuing response to /127.0.0.3 | 12:49:34.499  | /127.0.0.2    |
670        Sending message to /127.0.0.3    | 12:49:34.499  | /127.0.0.2    |
767        Processing response from /127.0.0.2 | 12:49:34.500  | /127.0.0.3    |
3237

```

Enabling debug-level logging

The driver uses the Simple Logging Facade for Java ([SLF4J](#)) which works with most common logging frameworks such as Apache [Log4j](#) and [logback](#). Enabling debugging depends on which framework your application uses. For example, if your driver client application uses Log4j, you enable debugging for the driver by adding the following to your driver client `log4j.properties` file:

```
log4j.logger.com.datastax.driver=DEBUG
```

Logging example

The `DEBUG` messages that result have additional information such as what contact points are used, what nodes were found, what nodes can and cannot be connected to, etc.

About this task

The following example uses the `SimpleClient` application you developed in the [Writing your first client](#) tutorial. It uses the `Log4j` library and configures it so that the driver logs `DEBUG` messages, while the `SimpleClient` application logs only `INFO` messages.

Procedure

1. Open the `simple-client` project in Eclipse.
2. Add the `slf4j-log4j12` JAR file as a Maven dependency.
 - a) Right-click on the `simple-client` project node and select **Maven > Add Dependency > .** The **Add Dependency** dialog displays.
 - b) Enter `slf4j-log4j12` in the search textfield.

- c) Expand the `org.apache.directory.studio` node and choose 1.7.2 [jar].
 - d) Click OK to dismiss the dialog.
3. Add a Log4j configuration file to the project.
- a) Right-click on the `src/main/resources` directory node in the package explorer and select **New > File**.
The **New File** dialog displays.
 - b) Enter `log4j.properties` in the **File name** textfield.
 - c) Configure Log4j to turn on debug-level logging for the driver and information-level logging for the `SimpleClient` application.

```
log4j.logger.com.datastax.driver=DEBUG, A1
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
```

```
log4j.logger.com.example.cassandra.SimpleClient=INFO, A2
log4j.appender.A2=org.apache.log4j.ConsoleAppender
log4j.appender.A2.layout=org.apache.log4j.PatternLayout
```

- d) Save the properties file.

The Log4j library looks for a `log4j.xml` or `log4j.properties` file on the classpath. Files stored in `src/main/resources` directory are automatically included on the project's classpath.

4. Run the `SimpleClient` application.

```
com.datastax.driver.NEW_NODE_DELAY_SECONDS is undefined, using default value
1
com.datastax.driver.NON_BLOCKING_EXECUTOR_SIZE is undefined, using default
value 4
com.datastax.driver.NOTIF_LOCK_TIMEOUT_SECONDS is undefined, using default
value 60
Cannot find Snappy class, you should make sure the Snappy library is in the
classpath if you intend to use it. Snappy compression will not be available
for the protocol.
Cannot find LZ4 class, you should make sure the LZ4 library is in the
classpath if you intend to use it. LZ4 compression will not be available
for the protocol.
Starting new cluster with contact points [/127.0.0.1:9042]
Connection[/127.0.0.1:9042-1, inFlight=0, closed=false] Transport
initialized and ready
[Control connection] Refreshing node list and token map
[Control connection] Refreshing schema
[Control connection] Refreshing node list and token map
[Control connection] Successfully connected to /127.0.0.1:9042
Using data-center name 'datacenter1' for DCAwareRoundRobinPolicy (if
this is incorrect, please provide the correct datacenter name with
DCAwareRoundRobinPolicy constructor)
New Cassandra host /127.0.0.3:9042 added
New Cassandra host /127.0.0.2:9042 added
New Cassandra host /127.0.0.1:9042 added
New Cassandra host /127.0.0.4:9042 added
Connected to cluster: darius
Connection[/127.0.0.1:9042-2, inFlight=0, closed=false] Transport
initialized and ready
...
Received event EVENT DROPPED TABLE simplex.songs, scheduling delivery
Received event EVENT DROPPED TABLE simplex.playlists, scheduling delivery
Received event EVENT DROPPED KEYSPACE simplex, scheduling delivery
Finished dropping simplex keyspace.
Connection[/127.0.0.3:9042-1, inFlight=0, closed=true] closing connection
Connection[/127.0.0.2:9042-1, inFlight=0, closed=true] closing connection
Connection[/127.0.0.1:9042-2, inFlight=0, closed=true] closing connection
```

```

Connection[/127.0.0.4:9042-1, inFlight=0, closed=true] closing connection
Shutting down
Connection[/127.0.0.1:9042-1, inFlight=0, closed=true] closing connection

```

Node discovery

The Java 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 the `Host.StateListener` interface.

Object-mapping API

Map table data to objects.

A common use case in Java applications is to transform query results into custom Java classes modeling domain objects. The new object-mapping API is distributed in the `cassandra-driver-mapping` JAR file. To use the object-mapping API, you must add the `cassandra-driver-mapping` JAR to your classpath or as a Maven dependency.

Version 2.1 of the driver introduces a new object mapping API with the following features:

Basic CRUD operations

Using specially annotated Java POJOs allows your application to perform basic CRUD operations (for example, save, delete and simple get) with the `Mapper` class.

For example:

```

CREATE TYPE complex.address (
    street text,
    city text,
    zipCode int,
    phones list<text>
);

CREATE TABLE complex.accounts (
    email text PRIMARY KEY,
    name text,
    addr frozen<address>
);

```

You annotate the Java class with `Table`, passing in the required elements specifying the keyspace and the table name. As long as the object's fields and the table's column's have the same name, you do not need any further annotations. The `email` column which is the primary key for the `accounts` table is annotated with `PartitionKey`.

```

package com.example.cassandra;

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

```

```
import com.google.common.base.Objects;

@Table(keyspace = "complex", name = "accounts")
public class Account {
    @PartitionKey
    private String email;
    private String name;
    @Column (name = "addr")
    @Frozen
    private Address address;

    public Account() {
    }

    public Account(String name, String email, Address address) {
        this.name = name;
        this.email = email;
        this.address = address;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }

    @Override
    public boolean equals(Object other) {
        if (other instanceof Account) {
            Account that = (Account) other;
            return Objects.equal(this.name, that.name) &&
                Objects.equal(this.email, that.email);
        }
        return false;
    }

    @Override
    public int hashCode() {
        return Objects.hashCode(name, email);
    }
}
```

Using a `Mapper` instance, you can perform basic CRUD operations on your object.


```

Mapper<Account> mapper = new
    MappingManager(getSession()).mapper(Account.class);
Phone phone = new Phone("home", "707-555-3537");
List<Phone> phones = new ArrayList<Phone>();
phones.add(phone);
Address address = new Address("25800 Arnold Drive", "Sonoma", 95476,
    phones);
Account account = new Account("John Doe", "jd@example.com", address);
mapper.save(account);
Account whose = mapper.get("jd@example.com");
System.out.println("Account name: " + whose.getName());
mapper.delete(account);

```

The `Mapper` object also has an asynchronous `get` method that returns a `com.google.common.util.concurrent.ListenableFuture<T>`.

```

ListenableFuture<User> userFuture = mapper.getAsync(userId);
ListenableFuture<Void> saveFuture = mapper.saveAsync(user);
ListenableFuture<Void> deleteAsync(userId);

```

Field-column mismatches

If an object's field has a different name from the corresponding column, you must annotate it, specifying the name. For example:

```

private int zipCode;

@Column( name = "zip_code" )
public int getZipCode() {
    return zipCode;
}

```

The Transient annotation

To prevent a field from being mapped, use the `Transient` annotation.

```

CREATE TABLE complex.minuscules (
    name text PRIMARY KEY,
    size int
);

@Table(keyspace = "complex", name = "minuscules")
public class Little {
    @PartitionKey
    private String name;
    @Transient
    private String secret;
    private int size;

    // etc.
}

```

The Enumerated annotation

If your class contains an enum type field, you use the `Enumerated` annotation.

```
enum Gender { FEMALE, MALE };

// ...

// FEMALE will be persisted as 'FEMALE'
@Enumerated(EnumType.STRING)
private Gender gender;

// FEMALE will be persisted as 0, MALE as 1
@Enumerated(EnumType.ORDINAL)
private Gender gender
```

Mapping UDTs

In your application, you can map your UDTs to application entities. For example, given the following UDT:

```
CREATE TYPE complex.address (
  street text,
  city text,
  zipCode int,
  phones list<text>
);
```

You create a Java class to map it to and annotate:

```
@UDT (keyspace = "complex", name = "address")
public class Address {
  private String street;
  private String city;
  private int zipCode;
  private List<Phone> phones;

  public Address() {
  }

  public String getStreet() {
    return street;
  }

  public void setStreet(String street) {
    this.street = street;
  }

  public String getCity() {
    return city;
  }

  public void setCity(String city) {
    this.city = city;
  }

  public int getZipCode() {
    return zipCode;
  }
}
```

```

    public void setZipCode(int zipCode) {
        this.zipCode = zipCode;
    }

    public List<Phone> getPhones() {
        return phones;
    }

    public void setPhones(List<Phone> phones) {
        this.phones = phones;
    }
}

```

You map the UDT to its corresponding Java class:

```

UDTMapper<Address> mapper = new MappingManager(getSession())
    .udtMapper(Address.class);

```

Once you have a mapper instance, you use it during your application's lifetime. When you retrieve a column of your UDT from a result set, it is an instance of `UDTValue`. You use your mapper to map it to your class.

```

ResultSet results = getSession().execute("SELECT * FROM complex.users " +
    "WHERE id = 756716f7-2e54-4715-9f00-91dcbea6cf50;");
for (Row row : results) {
    System.out.println(row.getString("name"));
    Map<String, UDTValue> addresses = row.getMap("addresses", String.class,
        UDTValue.class);
    for (String key : addresses.keySet()) {
        Address address = mapper.fromUDT(addresses.get(key));
        System.out.println(key + " address: " + address);
    }
}

```

CQL column and Java class fields mismatch

For the automatic mapping to work, the table column names and the class properties must match (case does not count). For example, in the UDT and the Java class examples above were changed:

```

CREATE TYPE address (
    street text,
    city text,
    zip_code int,
    phones list<text>
);

```

You use the `Field` annotation to provide the name of the field in your Java class that differs from that in your UDT:

```

public class Address {
    // all other fields and their getters and setters the same as above
    @Field (name = "zip_code")
    private int zipCode;

    public int getZipCode() {
        return zipCode;
    }
}

```

```
    public void setZipCode(int zipCode) {
        this.zipCode = zipCode;
    }
}
```

Accessor-annotated interfaces

If you need to use queries that are more complex than CRUD methods, you can use `Accessor`-annotated interfaces. The query strings are specific, but most of the boilerplate can still be handled automatically. You generate an object that implements the interface with the `MappingManager`.

Query are bound either by name or position.

An example using the `User` table above:

```
package com.example.cassandra;

import java.util.UUID;

import com.datastax.driver.core.ResultSet;
import com.datastax.driver.mapping.Result;
import com.datastax.driver.mapping.annotations.Accessor;
import com.datastax.driver.mapping.annotations.Param;
import com.datastax.driver.mapping.annotations.Query;

@Accessor
public interface UserAccessor {
    @Query("SELECT * FROM complex.users WHERE id = :id")
    User getUserNamed(@Param("userId") UUID id);

    @Query("SELECT * FROM complex.users WHERE id = ?")
    User getOnePosition(UUID userId);

    @Query("UPDATE complex.users SET addresses[:name]=:address WHERE id = :id")
    ResultSet addAddress(@Param("id") UUID id, @Param("name") String addressName, @Param("address") Address address);

    @Query("SELECT * FROM complex.users")
    public Result<User> getAll();

    @Query("SELECT * FROM complex.users")
    public ListenableFuture<Result<User>> getAllAsync();
}
```

Once you have your accessor written, you create an instance implementing the interface:

```
MappingManager manager = new MappingManager (getSession());
Address address = new Address();
UserAccessor userAccessor = manager.createAccessor(UserAccessor.class);
Result<User> users = userAccessor.getAll();
for(User user : users) {
    System.out.println(user.getName());
}
```

Table 8: Possible return types

Return type	Description
T	A mapped class. In which case the first row is mapped.
ResultSet	An <code>Iterable<Row></code> object.
Result<T>	An <code>Iterable<T></code> object that maps all the rows to objects.
ListenableFuture<R>	A Guava future, <code>com.google.common.util.concurrent.ListenableFuture</code> of one of the other return types. Returning this type means the method is asynchronous.
ResultSetFuture	Same as <code>ResultSet</code> , but the query will be executed asynchronously.
Statement	The method does not execute anything, but returns a <code>BoundStatement</code> ready for execution. This is useful if you want to add the statement to a batch.

Setting up your Java development environment

If you are not using Maven how to set up your environment.

Java driver dependencies

While the tutorials in this document were written using Eclipse and Maven, you can use any IDE or text editor and any build tool to develop client applications that use the driver.

The following JAR files and versions are required on your build environment classpath to use the Java driver:

- `cassandra-driver-core-2.1.5.jar`
- `guava-14.0.1.jar`
- `metrics-core-3.0.2.jar`
- `slf4j-api-1.7.10.jar`

Tuple types

A tuple is a fixed-length set of typed positional fields.

Cassandra 2.1 introduced the **tuple type** for CQL. For the following table: A tuple is a fixed-length set of typed positional fields.

```
CREATE TABLE tuple_test (
  the_key int PRIMARY KEY,
  the_tuple frozen<tuple<int, text, float>>)
```

You write to the tuple column in Java like this:

```
TupleType theType = TupleType.of(DataType.cint(), DataType.text(),
  DataType.cfloat());
TupleValue theValue = theType.newValue();
```

```
theValue.setInt (0, 1);
theValue.setString (1, "abc");
theValue.setFloat (2, 1.0f);
getSession().execute("INSERT INTO complex.tuple_test(the_key, the_tuple)
VALUES (?, ?)", 2, theValue);
```

You read from a tuple column like this:

```
PreparedStatement preparedStatement = session.prepare("SELECT the_tuple FROM
complex.tuple_test WHERE id = ?");
Row row = getSession().execute(preparedStatement.bind(2)).one();
TupleValue theValue1 = row.getTupleValue("the_tuple");
// As there are no names for tuple fields, access them by position
float theFloat = theValue1.getFloat(0);
String theText = theValue1.getString(1);
TupleValue theValue2 = session.execute("SELECT * FROM tuple_test WHERE
the_key = 2").one().getTupleValue("the_tuple");
String s = theValue2.getString(1);
```

User-defined types

How UDTs map to Java data types.

Cassandra 2.1 introduces support for **User-defined types** (UDT). A user-defined type simplifies handling a group of related properties.

A quick example is a user account table that contains address details described through a set of columns: street, city, zip code. With the addition of UDTs, you can define this group of properties as a type and access them as a single entity or separately.

User-defined types are declared at the keyspace level.

UDT API

You access UDTs as you do other metadata from your session instance. For example, given the following schema:

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

CREATE TYPE complex.phone (
    alias text,
    number text
);

CREATE TYPE complex.address (
    street text,
    zip_code int,
    phones list<phone>
);

CREATE TABLE complex.users (
    id int PRIMARY KEY,
    name text,
    addresses frozen<address>
);
```

UDTs are represented by instances of `UserType`, and you create new values by using the `UDTValue` class. Here is an example that uses the schema above.

```

PreparedStatement insertUserPreparedStatement
    = getSession().prepare("INSERT INTO complex.users (id, name, addresses)
    VALUES (?, ?, ?);");
PreparedStatement selectUserPreparedStatement
    = getSession().prepare("SELECT * FROM complex.users WHERE id = ?;");

UserType addressUDT = getSession().getCluster()
    .getMetadata().getKeyspace("complex").getUserType("address");
UserType phoneUDT = getSession().getCluster()
    .getMetadata().getKeyspace("complex").getUserType("phone");

UDTValue phone1 = phoneUDT.newValue()
    .setString("alias", "home")
    .setString("number", "1-707-555-1234");
UDTValue phone2 = phoneUDT.newValue()
    .setString("alias", "work")
    .setString("number", "1-800-555-9876");

UDTValue addresses = addressUDT.newValue()
    .setString("street", "123 Arnold Drive")
    .setInt("zip_code", 95476)
    .setList("phones", ImmutableList.of(phone1, phone2));

Map<String, UDTValue> addresses = new HashMap<String, UDTValue>();
addresses.put("Work", address);

UUID userId = UUID.fromString("fbdf82ed-0063-4796-9c7c-a3d4f47b4b25");
getSession().execute(insertUserPreparedStatement.bind(userId, "G. Binary",
    addresses));

Row row =
    getSession().execute(selectUserPreparedStatement.bind(userId)).one();
for ( UDTValue addr : row.getMap("addresses", String.class,
    UDTValue.class).values() ) {
    System.out.println("Zip: " + addr.getInt("zip_code"));
}

```

Direct field manipulation

Reading UDT fields

You can access a field within a UDT from a `SELECT` statement.

Given the following schema:

```

CREATE TABLE complex.customers (
    email text PRIMARY KEY,
    phone_number frozen<phone>);

CREATE TYPE complex.phone (
    alias text,
    number text);

```

You can:

```
ResultSet results = getSession()
    .execute("SELECT phone_number.number FROM " +
        "complex.customers WHERE email = 'grex@example.com'");
String number = results.one().getString("phone_number.number");
System.out.println("Phone number: " + number);
```

Writing UDT fields

You can change a field within a UDT from an `UPDATE` statement.

With the same schema above.

```
PreparedStatement preparedStatement = getSession()
    .prepare("UPDATE complex.customers SET phone_number = " +
        "{ number : ? } WHERE email = 'grex@example.com'");
getSession().execute(preparedStatement.bind("510-555-1209"));
results = getSession()
    .execute("SELECT * FROM " +
        "complex.customers WHERE email = 'grex@example.com'");
UDTValue value = results.one().getUDTValue("phone_number");
System.out.println("Phone number: " + value.getString("number"));
```


FAQ

- Can I check if a conditional (lightweight transaction) was successful?
- What is a parameterized statement and how can I use it?
- Does a parameterized statement escape parameters?
- What is the difference between a parameterized statement and a Prepared statement?
- Can I combine Prepared statements and normal statements in a batch?
- Can I get the raw bytes of a text column?
- Is there a way to control the batch size of the results returned from a query?
- What's the difference between using `setFetchSize()` and `LIMIT`?

Can I check if a conditional statement (lightweight transaction) was successful?

When executing a **conditional statements** the `ResultSet` will contain a single `Row` with a column named `applied` of type `boolean`. This tells whether the conditional statement was successful or not:

```
ResultSet rset = session.execute(conditionalStatement);
Row row = rset.one(); // if this is true then the statement was successful
row.getBool(0);      // this is equivalent row.getBool("applied")
```

What is a parameterized statement and how can I use it?

Starting with Cassandra 2.0, normal statements (that is non-prepared statement) do not need to concatenate parameter values inside a query string. Instead you can use `?` markers and provide the values separately:

```
session.execute( "INSERT INTO contacts (email, firstname, lastname)
VALUES (?, ?, ?)", "clint.barton@hawkeye.com", "Barney", "Barton");
```

Does a parameterized statement escape parameters?

A parameterized statement sends the values of parameters separate from the query (similarly to the way a `PreparedStatement`) as bytes so there is no need to escape parameters.

What's the difference between a parameterized statement and a Prepared statement?

The only similarity between a parameterized statement and a prepared statement is in the way that the parameters are sent. The difference is that a prepared statement:

- is already known on the cluster side (it has been compiled and there is an execution plan available for it) which leads to better performance
- sends only the statement id and its parameters (thus reducing the amount of data sent to the cluster)

Can I combine Prepared statements and normal statements in a batch?

Yes. A batch can include both bound statements and simple statements:

```
PreparedStatement ps = session.prepare( "INSERT INTO contacts (email,
    firstname, lastname)
    VALUES (?, ?, ?)"); BatchStatement batch = new BatchStatement();
batch.add(ps.bind(...));
batch.add(ps.bind(...));
// here's a simple statement
batch.add(new SimpleStatement( "INSERT INTO contacts (email, firstname,
    lastname) VALUES (?, ?, ?)", ...));
session.execute(batch);
```

Can I get the raw bytes of a text column?

If you need to access the raw bytes of a text column, call the `Row.getBytesUnsafe("columnName")` method.

Trying to using `Row.getBytes("columnName")` for the same purpose results in an exception as the `getBytes` method is used to retrieve a BLOB value.

How to increment counters with QueryBuilder?

Considering the following query:

```
UPDATE clickstream SET clicks = clicks + 1 WHERE userid = id;
```

To do this using QueryBuilder:

```
Statement query = QueryBuilder.update("clickstream")
    .with(incr("clicks", 1))
    // Use incr for counters
    .where(eq("userid", id));
```

Is there a way to control the batch size of the results returned from a query?

Use the `setFetchSize()` method on your `Statement` object. The fetch size controls how many resulting rows are retrieved simultaneously (the goal being to avoid loading too many results in memory for queries yielding large result sets).

What's the difference between using setFetchSize() and LIMIT?

Basically, `LIMIT` controls the maximum number of results done on the Cassandra side, while the `setFetchSize()` method controls the amount of data transferred between Cassandra and the client.










API reference

DataStax Java 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](#)