Java Driver 1.0 for Apache Cassandra

Documentation

October 8, 2013

Contents

Architectural overview

The Java Driver 1.0 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.

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:

- connection pooling
- node discovery
- automatic failover
- load balancing

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.

Cassandra compatibility

As explained above the driver is compatible with Cassandra 1.2 or greater. Cassandra 1.2.0 can be used with this driver, but it is strongly advised to use Cassandra 1.2.4 or higher as several limitations in prepared statements such as null support have been removed in it.

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>1.0.3</version>
</dependency>
```

Note: For DSE cleints, use the following Maven dependency instead:

```
<dependency>
  <groupId>com.datastax.cassandra</groupId>
  <artifactId>cassandra-driver-core</artifactId>
  <version>1.0.3-dse</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.

You can download the examples in this tutorial from GitHub.

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.

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.

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

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.Build auxiliary class
   - enables SSL for client to node encryption (optional)
   - builds a cluster instance
   - retrieves metadata from the cluster
   - 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)
         // .withSSL() // Uncomment if using client to node encryption
         .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.shutdown();
}
```

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("Datatacenter: %s; Host: %s; Rack: %s
\n",
```

```
                host.getDatacenter(), host.getAddress(),
 host.getRack());
      }
   }

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

   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
Datatacenter: datacenter1; Host: /127.0.0.1; Rack: rack1
Datatacenter: datacenter1; Host: /127.0.0.2; Rack: rack1
Datatacenter: 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.

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.

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

1. Modify your SimpleClient class.
   a) Add a Session instance field.

   ```
   private Session session;
   ```

   b) Get a session from your cluster and store the reference to it.
   Add the following line to the end of the connect method:

   ```
   session = cluster.connect();
   ```

   c) Add a getSession method which returns the session instance field.

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

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 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 simplex.songs (" +
            "id uuid PRIMARY KEY," +
            "title text," +
            "album text," +
            "artist text," +
            "tags set<text>," +
            "data blob" +
            ");");
session.execute(
      "CREATE TABLE simplex.playlists (" +
            "id uuid," +
            "title text," +
            "album text, " +
            "artist text," +
            "song_id uuid," +
            "PRIMARY KEY (id, title, album, artist)" +
            ");");
```

   c) In the class main method, add a call to the new createSchema method.

```
client.createSchema();
```

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

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

   a) 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'" +
                  ");");
```

b) In the class main method, add a call to the new loadData method.

```
client.createSchema();
```

5. 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();
```

6. In the class main method, add a call to the new querySchema method.

```
client.querySchema();
```

7. Add a dropSchema method and add code to drop the keyspace.

```
public void dropSchema(String keyspace) {
   getSession().execute("DROP KEYSPACE " + keyspace);
   System.out.println("Finished dropping " + keyspace + " keyspace.");
}
```

8. In the class main method, add a call to the new dropSchema method.

```
client.dropSchema();
```

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.

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

1. Create a new class, BoundStatementsClient, which extends the SimpleClient class.

```
public class BoundStatementsClient extends SimpleClient {
}
```

2. Override the loadData method and implement it.

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

3. 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 (?, ?, ?, ?, ?);");
```

4. 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.

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.dropSchema();
   client.close();
}
```

Java driver reference

Reference for the Java driver.

Connection requirements

In order to ensure that the Java 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`.

Asynchronous I/O

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.

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

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.

```
Query query = 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 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();
}
```

Debugging

You have several options to help in debugging your application.

On the client side, besides using the Eclipse debugger, you can monitor different metrics that the Java driver collects by default, (for example, connections to hosts). You can also implement your own metric objects and register them with the driver.

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 Yammer (and JMX) to allow for some monitoring using the Metrics class.

Description

The following links are for your reference:

Yammer metrics library
Java Management Extensions (JMX)

The Metrics object exposes the following metrics:

Table 1: 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.
requestsTimer	A timer that presents metrics on requests executed by the user.

In addition, you can register your own Yammer metric 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();
   System.out.printf("Number of user requests: %d %s\n", timer.count(),
 timer.eventType());
}
```

You can create and register your own Yammer metric objects:

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

Yammer 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("\"com.example.cassandra
\":type=\"MetricsExample\"," +
            "name=\"com.example.cassandra.numberInserts\"");
      JmxReporter.GaugeMBean mBean = JMX.newMBeanProxy(mbsConnection,
 objectName, JmxReporter.GaugeMBean.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.

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.

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'
 *    );
 */
Query 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() {
     Query 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.Query;
import com.datastax.driver.core.QueryTrace;
import com.datastax.driver.core.ResultSet;
import com.datastax.driver.core.SimpleStatement;
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'
          *    );
          */
      Query 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() {
      Query 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
------------------------------------------+--------------
+------------+--------------
                     Parsing statement | 12:17:16.736
 | /127.0.0.1 |           28
                    Peparing statement | 12:17:16.736
 | /127.0.0.1 |          199
     Determining replicas for mutation | 12:17:16.736
 | /127.0.0.1 |          348
         Sending message to /127.0.0.3 | 12:17:16.736
 | /127.0.0.1 |          788
         Sending message to /127.0.0.2 | 12:17:16.736
 | /127.0.0.1 |          805
        Acquiring switchLock read lock | 12:17:16.736
 | /127.0.0.1 |          828
               Appending to commitlog | 12:17:16.736
 | /127.0.0.1 |          848
             Adding to songs memtable | 12:17:16.736
 | /127.0.0.1 |          900
      Message received from /127.0.0.1 | 12:17:16.737
 | /127.0.0.2 |           34
      Message received from /127.0.0.1 | 12:17:16.737
 | /127.0.0.3 |           25
        Acquiring switchLock read lock | 12:17:16.737
 | /127.0.0.2 |          672
        Acquiring switchLock read lock | 12:17:16.737
 | /127.0.0.3 |          525
               Appending to commitlog | 12:17:16.737
 | /127.0.0.2 |          692
               Appending to commitlog | 12:17:16.737
 | /127.0.0.3 |          541
             Adding to songs memtable | 12:17:16.737
 | /127.0.0.2 |          741
             Adding to songs memtable | 12:17:16.737
 | /127.0.0.3 |          583
      Enqueuing response to /127.0.0.1 | 12:17:16.737
 | /127.0.0.3 |          751
      Enqueuing response to /127.0.0.1 | 12:17:16.738
 | /127.0.0.2 |          950
      Message received from /127.0.0.3 | 12:17:16.738
 | /127.0.0.1 |          178
         Sending message to /127.0.0.1 | 12:17:16.738
 | /127.0.0.2 |         1189
      Message received from /127.0.0.2 | 12:17:16.738
 | /127.0.0.1 |          249
   Processing response from /127.0.0.3 | 12:17:16.738
 | /127.0.0.1 |          345
   Processing response from /127.0.0.2 | 12:17:16.738
 | /127.0.0.1 |          377


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
------------------------------------------+--------------
+------------+--------------
```

```
                     Parsing statement | 12:49:34.497
  | /127.0.0.3 |           35
                    Peparing statement | 12:49:34.497
  | /127.0.0.3 |          191
          Determining replicas to query | 12:49:34.497
  | /127.0.0.3 |          342
          Sending message to /127.0.0.2 | 12:49:34.498
  | /127.0.0.3 |         1561
       Message received from /127.0.0.3 | 12:49:34.499
  | /127.0.0.2 |           37
       Message received from /127.0.0.2 | 12:49:34.499
  | /127.0.0.3 |         2880
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
```

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.

Cluster configuration

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

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

Tuning policies

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

Load balancing policy

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

Description

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(Query query, ConsistencyLevel cl, int requiredResponses, int receivedResponses, boolean dataRetrieved, int nbRetry)`
- `RetryPolicy.RetryDecision onUnavailable(Query query, ConsistencyLevel cl, int requiredReplica, int aliveReplica, int nbRetry)`
- `RetryPolicy.RetryDecision onWriteTimeout(Query query, 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 2: 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 isoptimized 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 3: 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 4: Pooling options

Option	Corresponds to	Description
connectTimeoutMillis	org.jboss.netty.channel.ChannelConfig "connectTineoutMillis"	The connect timeout in milliseconds for the underlying Netty channel.
keepAlive	java.net.SocketOptions.SO_KEEPALIVE	
receiveBufferSize	java.net.SocketOptions.SO_RCVBUF	A 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	A 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.TCPNODELAY	Disables Nagle's algorithm on the underlying socket.

Query builder

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

Examples

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

```
public ResultSet getRows(String keyspace, String table) {
   Query query = QueryBuilder
      .select()
      .all()
      .from(keyspace, table);
   return getSession()
      .execute(query)
      .getAll();
}
```

CQL3 data types to Java types

A summary of the mapping between CQL3 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 5: Java classes to CQL3 data types

CQL3 data type	Java type
ascii	java.lang.String
bigint	long
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
uuid	java.util.UUID
varchar	java.lang.String
varint	java.math.BigInteger

Setting up your Java development environment

While the tutorials here 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 Java driver.

Java driver dependencies

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

- `cassandra-driver-core-1.0.3.jar`
- `netty-3.6.3-Final.jar`
- `guava-14.0.1.jar`
- `cassandra-thrift-1.2.3.jar`
- `commons-lang-2.4.jar`
- `lib-thrift-0.7.0.jar`
- `servlet-api-2.5.jar`

- httpclient-4.0.1.jar
- httpcore-4.0.1.jar
- commons-logging-1.1.1.jar
- jackson-core-asl.1.9.2.jar
- cassandra-all-1.2.3.jarsnappy-java-1.0.4.1.jar
- lz4-1.1.0.jar
- compress-lzf-0.8.4.jar
- commons-cli-1.1.jar
- commons-codec-1.2.jar
- concurrentlinkedhashmap-lru-1.3.jar
- antlr-3.2.jar
- antlr-runtime-3.2.jar
- antlr-2.7.7.jar
- avro-1.4.0-cassandra-1.jar
- jetty-6.1.22.jar
- jetty-util-6.1.22.jar
- servlet-api-2.5-20081211.jar
- jackson-mapper-asl-1.9.2.jar
- jline-1.0.jar
- json-simple-1.1.jar
- highscale-lib-1.1.2.jar
- snakeyaml-1.6.jar
- snaptree-0.1.jar
- jbcrypt-0.3m.jar
- log4j-1.2.16.jar
- slf4j-log4j12-1.7.2.jar
- jamm-0.2.5.jar

API reference

DataStax Java Driver for Apache Cassandra.