



Node.js Driver 2.1 for Apache Cassandra Documentation

`#{ds.localized.time}`

Contents

- What's new?..... 3**

- About the Node.js driver.....4**
 - The driver and its dependencies..... 4

- Writing your first client..... 6**
 - Setting up the project.....6
 - Connecting to a Cassandra cluster.....7
 - Executing CQL statements.....9

- Reference..... 14**
 - Three simple rules for coding with the driver..... 14**
 - BATCH statements..... 15
 - Client configuration..... 15
 - Address resolution..... 16
 - Client options..... 17
 - Native protocol..... 19
 - Pooling configuration..... 20
 - Tuning policies..... 21
 - Cluster and schema metadata..... 23
 - CQL data types to JavaScript types..... 24
 - Collections..... 25
 - Numerical values..... 27
 - User-defined types..... 29
 - UUID and time-based UUID data types..... 30
 - Tuples..... 31
 - Fetching large result sets..... 32
 - Parameterized queries..... 33
 - Routing queries..... 34

- FAQ..... 35**
 - Which versions of Cassandra does the driver support?..... 35
 - Which versions of CQL does the driver support?..... 35
 - How do I generate a uuid or a timebased uuid?..... 35
 - Should I create one client instance per module in my application?..... 35
 - Should I shut down the pool after executing a query?..... 35

- API reference..... 36**

- Using the docs..... 37**

What's new?

Here are the new and noteworthy features of the driver.

What's new in 2.1?

- Support for Cassandra 2.1 features
 - user-defined types
 - tuples
 - nested collections
 - native binary protocol version 3
- `AddressTranslator` interface
- Expose table metadata
- Query tracing

What's new in 2.0?

- Named parameters support
- Improved support for data types
 - ECMAScript 6 Map and Set support
 - Improved `uuid` and `timeuuid` client support
 - Improved client support for `decimal` and `varint` types
- BATCH of prepared statements
- Heartbeat at connection level: When a connection is idling for some time, it makes a background request to prevent it being automatically closed by firewalls.

About the Node.js driver

The Node.js Driver 2.1 for Apache Cassandra works exclusively with the Cassandra Query Language (CQL) version 3 and Cassandra's binary protocol which was introduced in Cassandra version 1.2.

Architectural overview

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

The driver has the following features:

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

The driver and its dependencies

The Node.js driver only supports the Cassandra Binary Protocol and CQL.

Cassandra binary protocol

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

```
start_native_transport: true
```

Then restart the node.

Cassandra compatibility

The driver is compatible with any Cassandra version from 1.2. The driver uses native protocol version 1 (for Cassandra 1.2), version 2 (Cassandra 2.0), and version 3 (Cassandra 2.1).

Build environment dependencies

The driver works with the following versions of Node.js:

- Node.js 0.10

Writing your first client

This section walks you through a small web application that implements a RESTful API and uses the Node.js driver to connect to a Cassandra cluster, create a schema and load some data, and execute some queries.

Setting up the project

This tutorial uses Node.js and Express.js to build a simple web application that exposes a REST API to access data in a Cassandra cluster.

Before you begin

This tutorial uses the following software:

- An **installed** and running **Cassandra** or DSE cluster
- **Node.js**
- **npm** (Node Packaged Modules)
- **cURL**

Procedure

1. Install the Express (web application framework) module.

```
$ npm install -g express
```

2. Create a stub Express project (called simple) in a directory of your choice.

```
$ mkdir projects
$ cd projects
$ node express simple
```

3. Edit and save the project's `package.json` file to add some information and declare the dependencies. Changes in bold.

```
{
  "name": "simple",
  "description": "Simple Cassandra client",
  "version": "0.0.1",
  "private": true,
  "dependencies": {
    "express": "4.9",
    "body-parser": "1.9",
    "async": "0.9.0",
    "cassandra-driver": "1.0.0"  }
  }
}
```

4. Install the dependencies.

```
$ cd simple
$ npm install
```

Connecting to a Cassandra cluster

The Node.js driver provides a `Client` class which is your application's entry point for connecting to a Cassandra cluster and retrieving metadata.

About this task

This tutorial implements a simple **REST** (representational state transfer) API.

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

Procedure

- Using a text editor, add the appropriate `require` statements at the beginning of the `server.js` file in the simple project.
These are the modules you installed in the previous object.

```
var express = require('express');
var bodyParser = require('body-parser');
var cassandra = require('cassandra-driver');
```

- Add statements to connect to a node in your Cassandra cluster and set up your web application to use the `express` and `body-parser` modules.
 - Instantiate a `Client` object and connect to the cluster.

```
var client = new cassandra.Client( { contactPoints :
  [ '127.0.0.1' ] } );
client.connect(function(err, result) {
  console.log('Connected.');
```

The `connect` function ensures the pool is connected. It is optional, internally the driver calls `connect` when executing a query.

- Add statements to instantiate an Express web server object, have it use the `body-parser` module for handling JSON in HTTP requests and responses, and turn on pretty printing of JSON for readability.

```
var app = express();
app.use(bodyParser.json());
app.set('json spaces', 2);
```

- Register a function and map it to a `GET` HTTP request URL.

```
app.get('/metadata', function(req, res) {
  res.send(client.hosts.slice(0).map(function (node) {
    return { address : node.address, rack : node.rack, datacenter :
      node.datacenter }
  })));
});
```

The `GET` request returns an array of objects which contain the node IP address, the rack, and the datacenter. For example:

- Save the `server.js` file.
- Run the server.

Writing your first client

```
$ node server.js
```

A single line is displayed.

```
Listening on port 3000
```

6. Call the API using cURL to create the keyspace on the cluster.

```
$ curl -H "Content-Type: application/json" -X GET http://localhost:3000/metadata
```

```
[
  {
    "address": "127.0.0.1",
    "rack": "rack1",
    "datacenter": "datacenter1"
  },
  {
    "address": "127.0.0.3",
    "rack": "rack1",
    "datacenter": "datacenter1"
  },
  {
    "address": "127.0.0.2",
    "rack": "rack1",
    "datacenter": "datacenter1"
  }
]
```

Code listing

The complete code listing illustrates:

- connecting to a cluster
- retrieving metadata and printing it out

```
var express = require('express');
var bodyParser = require('body-parser');
var cassandra = require('cassandra-driver');

var client = new cassandra.Client( { contactPoints : [ '127.0.0.1' ] } );
client.connect(function(err, result) {
  console.log('Connected.');
```

```
});

var app = express();
app.use(bodyParser.json());
app.set('json spaces', 2);

app.get('/metadata', function(req, res) {
  res.send(client.hosts.slice(0).map(function (node) {
    return { address : node.address, rack : node.rack, datacenter :
node.datacenter }
  }));
});

var server = app.listen(3000, function() {
  console.log('Listening on port %d', server.address().port);
});
```


Executing CQL statements

Once you have connected to a Cassandra cluster using a `Client` object, you execute CQL statements to read and write data.

Before you begin

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

About this task

Getting metadata for the cluster is good, but you also want to be able to read and write data to the cluster. The Node.js driver lets you execute CQL statements using a `Client` instance. You will add code to your client for:

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

In this topic you will add the following functionality by mapping functions that execute CQL statements using the driver `Client` object.

Table 1:

URL	HTTP method	Description
<code>http://localhost:3000/keyspace</code>	POST	Creates a keyspace, <code>simplex</code> , on the cluster. (The example here uses a three node cluster running on localhost.)
<code>http://localhost:3000/tables</code>	POST	Creates two tables, <code>songs</code> and <code>playlists</code> in the <code>simplex</code> keyspace.
<code>http://localhost:3000/song</code>	POST	Insert data into the two tables. (The example here will use text files with JSON.)
<code>http://localhost:3000/song/id</code>	GET	Retrieve a song by id.

Procedure

1. Add code to your `server.js` file to create the `simplex` keyspace and the `songs` and `playlists` tables.
 - a) Register a POST URL with the `app` object to create the keyspace.

```
app.post('/keyspace', function(req, res) {
  client.execute("CREATE KEYSPACE IF NOT EXISTS simplex WITH
  replication " +
    "= {'class' : 'SimpleStrategy',
    'replication_factor' : 3};",
    afterExecution('Error: ', 'Keyspace created.', res));
});
```

Writing your first client

Note the `afterExecution` function being passed to the `execute` method. Because the driver executes CQL statements asynchronously, you need to pass in a callback function.

- b) Add the `afterExecution` function.

```
function afterExecution(errorMessage, successMessage, res) {
  return function(err) {
    if (err) {
      return res.json(errorMessage);
    } else {
      res.json(successMessage);
    }
  }
}
```

- c) Register a `POST` URL with the `app` object to create the two tables.

```
app.post('/tables', function(req, res) {
  async.parallel([
    function(next) {
      client.execute('CREATE TABLE IF NOT EXISTS simplex.songs ('
+
      'id uuid PRIMARY KEY,' +
      'title text,' +
      'album text,' +
      'artist text,' +
      'tags set<text>,' +
      'data blob' +
      ');',
      next);
    },
    function(next) {
      client.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)' +
      ');',
      next);
    }
  ], afterExecution('Error: ', 'Tables created.' , res));
});
```

2. Add code to insert data into the `simplex.songs` table.

- a) Add a global variable for the `upsert` statement with bind variables.

```
var upsertSong = 'INSERT INTO simplex.songs (id, title, album, artist,
tags, data) '
+ 'VALUES(?, ?, ?, ?, ?, ?);';
```

- b) Register a `POST` URL with the `app` object to insert data into the `simplex.songs` table.

```
app.post('/song', function(req, res) {
  var id = null;
  if (! req.body.hasOwnProperty('id')) {
    id = cassandra.types.uuid();
  }
}
```

```

    } else {
      id = req.body.id;
    }
    client.execute(upsertSong,
      [id, req.body.title, req.body.album, req.body.artist,
      req.body.tags, null],
      afterExecution('Error: ', 'Song ' + req.body.title + '
      upserted.', res));
  });

```

The function tests to see whether a UUID is provided in the request body. If not it creates one.

3. Add code to get a song (specified by id) from the `simplex.songs` table.

a) Add a global variable for the `SELECT` statement with bind variables.

```
var getSongById = 'SELECT * FROM simplex.songs WHERE id = ?';
```

b) Register a `GET` URL with the `app` object to select a song (specified by id) from the `simplex.songs` table.

```

app.get('/song/:id', function(req, res) {
  client.execute(getSongById, [ req.params.id ], function(err, result)
  {
    if (err) {
      res.status(404).send({ msg : 'Song not found.' });
    } else {
      res.json(result);
    }
  });
});

```

4. Using a text editor, create two or more songs to insert with `cURL`.

Save the following object in a file, `song001.json`.

```

{
  "id" : "756716f7-2e54-4715-9f00-91dcbea6cf50",
  "title" : "La Petite Tonkinoise",
  "album" : "Bye Bye Blackbird",
  "artist" : "Joséphine Baker",
  "tags" : [ "jazz", "2013" ]
}

```

5. Run the server, insert some songs, and retrieve a song by id.

a) Save the `server.js` file.

b) Run the server.

```
$ node server.js
```

A single line is displayed.

```
Listening on port 3000
```

c) Insert some songs with `cURL`.

```
$ curl -H "Content-Type: application/json" -X POST --data @song001.json
http://localhost:3000/song
```

d) Retrieve a song by id.

```
$ curl -H "Content-Type: application/json" -X GET http://localhost:3000/
song/756716f7-2e54-4715-9f00-91dcbea6cf50
```

Writing your first client

An excerpt of the JSON returned.

```
{
  "rows": [
    {
      "id": "756716f7-2e54-4715-9f00-91dcbea6cf50",
      "album": "Bye Bye Blackbird",
      "artist": "Joséphine Baker",
      "data": null,
      "tags": [
        "2013",
        "jazz"
      ],
      "title": "La Petite Tonkinoise"
    }
  ]
}
```

Example

The complete code listing illustrates:

- creating a keyspace with two tables
- loading data into your new schema
- retrieving a row from one of the tables

```
var upsertSong = 'INSERT INTO simplex.songs (id, title, album, artist, tags,
  data) '
  + 'VALUES(?, ?, ?, ?, ?, ?)';
var getSongById = 'SELECT * FROM simplex.songs WHERE id = ?';

var app = express();
app.use(bodyParser.json());

var client = new cassandra.Client( { contactPoints : [ '127.0.0.1' ] } );
client.connect(function(err, result) {
  console.log('Connected. ');
});

var app = express();
app.use(bodyParser.json());

app.get('/metadata', function(req, res) {
  res.send(client.hosts.slice(0).map(function (node) {
    return { address : node.address, rack : node.rack, datacenter :
      node.datacenter }
  }));
});

app.post('/keyspace', function(req, res) {
  client.execute("CREATE KEYSPACE IF NOT EXISTS simplex WITH replication "
    +
      " = { 'class' : 'SimpleStrategy', 'replication_factor' :
    3 };",
    afterExecution('Error: ', 'Keyspace created.', res));
});

app.post('/tables', function(req, res) {
  async.parallel([
    function(next) {
      client.execute('CREATE TABLE IF NOT EXISTS simplex.songs (' +
        'id uuid PRIMARY KEY, ' +
```

```

        'title text,' +
        'album text,' +
        'artist text,' +
        'tags set<text>,' +
        'data blob' +
        ');',
        next);
    },
    function(next) {
        client.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)' +
            ');',
            next);
    }
], afterExecution('Error: ', 'Tables created.' , res));
});

app.post('/song', function(req, res) {
    var id = null;
    if ( ! req.body.hasOwnProperty('id')) {
        id = cassandra.types.uuid();
    } else {
        id = req.body.id;
    }
    client.execute(upsertSong,
        [id, req.body.title, req.body.album, req.body.artist, req.body.tags,
        null],
        afterExecution('Error: ', 'Song ' + req.body.title + ' upserted.',
        res));
});

app.get('/song/:id', function(req, res) {
    client.execute(getSongById, [ req.params.id ], function(err, result) {
        if (err) {
            res.status(404).send({ msg : 'Song not found.' });
        } else {
            res.json(result);
        }
    });
});

function afterExecution(errorMessage, successMessage) {
    return function(err, result) {
        if (err) {
            return console.log(errorMessage);
        } else {
            return console.log(successMessage);
        }
    }
}

var server = app.listen(3000, function() {
    console.log('Listening on port %d', server.address().port);
});

```

Reference

Reference for the Node.js driver.

Three simple rules for coding with the driver

When writing code that uses the driver, there are three simple rules that you should follow that make your code efficient:

- Only use one `Client` instance per keyspace or use a single `Client` and explicitly specify the keyspace in your queries and reuse it in across your modules in the application lifetime.
- If you execute a statement more than once, use a prepared statement.
- You can reduce the number of network roundtrips and also have atomic operations by using batches.

Client

The `Client` instance allows you to configure different important aspects of the way connections and queries are 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 on. Generally such settings are set once at the application level.

```
var cassandra = require('cassandra-driver');
var DCAwareRoundRobinPolicy = cassandra.policies.DCAwareRoundRobinPolicy;
var client = new cassandra.Client({
  contactPoints: ['10.1.1.3', '10.1.1.4', '10.1.1.5'],
  policies: {
    loadBalancing: new DCAwareRoundRobinPolicy('US_EAST');
  }
});
```

A `Client` instance is a long-lived object, and it should not be used in a request-response, short-lived fashion.

Your code should share the same client instance 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 they (and the query id) are sent over the wire. These performance gains add up when using the same queries (with different parameters) repeatedly. Additionally, when preparing, the driver retrieves information about the parameter types which allows an accurate mapping between a JavaScript type and a CQL type.

Preparing and executing statements in the driver does not require two chained asynchronous calls. You can set the `prepare` flag in the query options and the driver handles the rest.

```
var query = 'SELECT id, name FROM users WHERE id = ?';
client.execute(query, [id], {prepare: true}, callback);
```

BATCH statements

The `BATCH` statement combines multiple data modification statements (`INSERT`, `UPDATE`, or `DELETE`) into a single logical operation that is sent to the server in a single request. Batching together multiple

operations also ensures that they are executed in an atomic way, (that is, 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.

Starting with Cassandra 2.0, prepared statements can be used in batch operations.

```
var queries = [
  { query: 'UPDATE user_profiles SET email=? WHERE key=?',
    params: [emailAddress, 'hendrix']},
  { query: 'INSERT INTO user_track (key, text, date) VALUES (?, ?, ?)',
    params: ['hendrix', 'Changed email', new Date()]}
];
var queryOptions = { prepare: true, consistency:
  cassandra.types.consistencies.quorum };
client.batch(queries, queryOptions, function(err) {
  assert.ifError(err);
  console.log('Data updated on cluster');
});
```

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

BATCH statements

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

It's common for applications to require atomic batching of multiple `INSERT`, `UPDATE`, or `DELETE` statements, even in different partitions or column families. Thanks to the Cassandra protocol changes introduced in Cassandra 2.0, the driver allows you to execute multiple statements efficiently without the need to concatenate multiple queries.

The method `batch()` accepts the queries as first parameter:

```
var query1 = 'UPDATE user_profiles SET email = ? WHERE key = ?';
var query2 = 'INSERT INTO user_track (key, text, date) VALUES (?, ?, ?)';
var queries = [
  { query: query1, params: [emailAddress, 'hendrix'] },
  { query: query2, params: ['hendrix', 'Changed email', new Date()] }
];
client.batch(queries, { prepare: true}, function (err) {
  // All queries have been executed successfully
  // Or none of the changes have been applied, check err
});
```

By preparing your queries, you will get the best performance and your JavaScript parameters correctly mapped to Cassandra types. The driver will prepare each query once on each host and execute the batch every time with the different parameters provided.

Client configuration

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

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

Address resolution

The driver auto-detects new Cassandra nodes when they are added to the cluster by means of server-side push notifications and checking the system tables.

For each node, the address the driver receives the address set as `rpc_address` in the node's `cassandra.yaml` file. In most cases, this is the correct value, however, sometimes the addresses received in this manner are either not reachable directly by the driver or are not the preferred address to use. A common such scenario is a multi-datacenter deployment with a client connecting using the private IP address to the local datacenter (to reduce network costs) and the public IP address for the remote datacenter nodes.

The AddressTranslator interface

The `AddressTranslator` interface allows you to deal with such cases, by transforming the address sent by a Cassandra node to another address to be used by the driver for connection.

```
function MyAddressTranslator() {
}

util.inherits(MyAddressTranslator, AddressTranslator);
MyAddressTranslator.prototype.translate = function (address, port, callback)
{
  // Your custom translation logic.
};
```

You then configure the driver to use your `AddressTranslator` implementation in the client options.

```
var client = new Client({
  contactPoints: ['1.2.3.4'],
  policies: {
    addressResolution: new MyAddressTranslator()
  }
});
```

Note: The contact points provided while creating the `Client` are not translated, only addresses retrieved from or sent by Cassandra nodes are.

EC2 multi-region

The `EC2MultiRegionTranslator` class is provided out of the box. It helps optimize network costs when your infrastructure (both Cassandra nodes and clients) is distributed across multiple Amazon EC2 regions:

- a client communicating with a Cassandra node in the same EC2 region should use the node's private IP address (which is less expensive)
- a client communicating with a node in a different region should use the public IP address

To use this implementation, provide an instance when initializing the `Client` object.

```
var cassandra = require('cassandra-driver');
var addressResolution = cassandra.policies.addressResolution;
var client = new Client({
  contactPoints: ['1.2.3.4'],
  policies: {
    addressResolution: new addressResolution.EC2MultiRegionTranslator()
  }
});
```


The `Client` class performs a reverse DNS lookup of the origin address to find the domain name of the target instance. Then it performs a forward DNS lookup of the domain name; the EC2 DNS does the private to public switch automatically based on location.

Client options

Client options are configured when instantiating a `Client` object.

Table 2: Client Options

Name	Type	Description
<code>contactPoints</code>	Array	Array of addresses or host names of the nodes to add as contact point.
<code>policies</code>	Object	Properties: <ul style="list-style-type: none"> <code>loadBalancing</code>: the <code>LoadBalancingPolicy</code> instance to be used to determine the coordinator per query. Default: <code>TokenAwarePolicy</code> with <code>DCAwareRoundRobinPolicy</code> as it child. <code>retry</code>: the <code>RetryPolicy</code> to be used. Default: <code>RetryPolicy</code>. <code>reconnection</code>: the <code>ReconnectionPolicy</code> to be used. Default: <code>new reconnection.ExponentialReconnect(10 * 60 * 1000, false)</code>
<code>queryOptions</code>	<code>QueryOptions</code>	
<code>pooling</code>	Object	Properties: <ul style="list-style-type: none"> <code>heartbeatInterval</code>: the amount of idle time in milliseconds that has to pass before the driver issues a request on an active connection to avoid idle time disconnections;. Default: 30000 <code>coreConnectionsPerHost</code>: an associative array containing amount of connections per host distance.
<code>protocolOptions</code>	Object	Properties: <ul style="list-style-type: none"> <code>port</code>: the port Number to use to connect to the Cassandra host. Default: 9042. <code>maxSchemaAgreementWaitSeconds</code>: the maximum time in seconds to wait for schema agreement

Reference

Name	Type	Description
		between nodes before returning from a DDL query. Default: 10.
socketOptions	Object	Properties: <ul style="list-style-type: none">• <code>connectTimeout</code>: connection timeout in Number of milliseconds. Default: 5000.• <code>keepAlive</code>: whether to enable TCP keepalive on the socket. Default: true.• <code>keepAliveDelay</code>: TCP keepalive delay in milliseconds. Default: 0.
authProvider	AuthProvider	Provider to be used to authenticate to an auth-enabled host. Default: null.
sslOptions	Object	Client-to-node ssl options: when set the driver will use the secure layer. You can specify cert, ca, ... options named after the Node.js <code>tls.connect</code> options.
encoding	Object	Properties: <ul style="list-style-type: none">• <code>map</code>: a map constructor to use for Cassandra map types encoding and decoding. Default: Javascript Object with map keys as property names.• <code>set</code>: a set constructor to use for Cassandra set types encoding and decoding. Default: Javascript Array.

The user can provide the options when creating a new instance of client:

```
var options = {
  policies: {
    loadBalancing: new cassandra.loadBalancing.DCAwareRoundRobinPolicy()
  },
  authProvider: new cassandra.auth.PlainTextAuthProvider('usr1',
  'p@ssword1')
}

var client = new Client(options);
```

Query options

Name	Type	Description
consistency	Number	Consistency level.

Name	Type	Description
fetchSize	Number	Amount of rows to retrieve per page.
prepare	Boolean	Determines if the query must be executed as a prepared statement.
autoPage	Boolean	Determines if the driver must retrieve the next pages.
routingKey	Buffer or Array	Partition key(s) to determine which coordinator should be used for the query.
routingIndexes	Array	Index of the parameters that are part of the partition key to determine the routing.
routingNames	Array	Array of the parameters names that are part of the partition key to determine the routing.
hints	Array or Array<Array>	Type hints for parameters given in the query, ordered as for the parameters. For batch queries, an array of such arrays, ordered as with the queries in the batch.
pageState	Buffer or String	Buffer or string token representing the paging state. Useful for manual paging, if provided, the query will be executed starting from a given paging state.
retry	RetryPolicy	Retry policy for the query. This property can be used to specify a different retry policy to the one specified in the <code>ClientOptions.policies</code> .

Native protocol

Examples

The native protocol defines the format of the binary messages exchanged between the driver and Cassandra over TCP. As a driver user what you need to be aware of is that some Cassandra features are only available with a specific protocol version, but if you are interested in the technical details you can check the specification in the [Cassandra codebase](#).

Compatibility matrix

	Cassandra 1.2.x (DSE 3.2)	Cassandra 2.0.x (DSE 4.0 – 4.6)	Cassandra 2.1.x (DSE 4.7)
Driver 1.0.x	Native protocol version 1	Native protocol version 1	Native protocol version 1
Driver 2.0.x	Native protocol version 1	Native protocol version 2	Native protocol version 2

	Cassandra 1.2.x (DSE 3.2)	Cassandra 2.0.x (DSE 4.0 – 4.6)	Cassandra 2.1.x (DSE 4.7)
Driver 2.1.x	Native protocol version 1	Native protocol version 2	Native protocol version 3

For example, if you use version 2.1.0 of the driver to connect to Cassandra 2.0.9, the maximum version you can use (and the one you'll get by default) is native protocol version 2 (last line, middle column). If you use the same version to connect to Cassandra 2.1.4, you can use native protocol version 3.

Controlling the protocol version

By default, the driver uses the highest protocol version supported by the driver and the Cassandra cluster. If you want to limit the protocol version to use, you do so in the protocol options.

```
var client = new Client({
    contactPoints: ['1.2.3.4'],
    protocolOptions: { maxVersion: 2}
});
```

Mixed cluster versions and rolling upgrades

The protocol version used between the client and the Cassandra cluster is negotiated upon establishing the first connection. For clusters with nodes running mixed versions of Cassandra and during rolling upgrades this could represent an issue that could lead to limited availability.

To exemplify the above, consider a mixed cluster having nodes running either Cassandra 2.1 or 2.0. Assuming the driver version is 2.1.0 or greater:

- The first contact point is a 2.1 host, so the driver negotiates native protocol version 3
- While connecting to the rest of the cluster, the driver contacts a 2.0 host using native protocol version 3, which fails; an error is logged and this host will be permanently ignored

For these scenarios, mixed version clusters and rolling upgrades, it is strongly recommended to set the maximum protocol version when initializing the client:

```
var client = new Client({
    contactPoints: ['1.2.3.4'],
    protocolOptions: { maxVersion: 2}
});
```

And switching it to the highest protocol version once the upgrade is completed, by leaving the maximum protocol version unspecified:

```
var client = new Client({ contactPoints: ['1.2.3.4'] });
```

Pooling configuration

The driver maintains one or more connections opened to each Cassandra node selected by the load-balancing policy. The amount of connections per host is defined in the [pooling configuration](#).

Default pooling configuration

The default number of connections per host depends on which Cassandra version the driver connects to.

Cassandra versions 1.2 and 2.0 allow the clients to send up to 128 requests without waiting for a response per connection. Higher versions of Cassandra (that is 2.1 or greater) allow clients to send up to 32768 requests without waiting for a response.

By default, the driver maintains two open connections to each host in the local datacenter and one to each host in a remote datacenter for Cassandra 1.2 or 2.0 and one connection to each host (local or remote) for Cassandra 2.1 or greater.

Setting the number of connections per host

You set the number of connections per host in the pooling configuration:

```
var cassandra = require('cassandra-driver');
var distance = cassandra.types.distance;
var options = {
  contactPoints: ['1.2.3.4'],
  pooling: {
    coreConnectionsPerHost: {}
  }
};
options.pooling.coreConnectionsPerHost[distance.local] = 4;
options.pooling.coreConnectionsPerHost[distance.remote] = 1;
var client = new Client(options);
```

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:

- `#distance(Host host)`: determines the distance to the specified host. The values are `distance.ignored`, `distance.local`, and `distance.remote`.
- `#init(client, hosts, callback)`: initializes the policy. The driver calls this method only once and before any other method calls are made.
- `#newQueryPlan(keyspace, queryOptions, callback)`: executes a callback with the iterator of hosts to use for a query. Each new query calls this method.

The policies are responsible for yielding a group of nodes in an specific order for the driver to use (if the first node fails, it uses the next one). There are three load-balancing policies implemented in the driver:

- `DCAwareRoundRobinPolicy`: a datacenter-aware, round-robin, load-balancing policy. This policy provides round-robin queries over the node of the local datacenter. It also includes in the query plans returned a configurable number of hosts in the remote data centers, but those are always tried after the local nodes.
- `RoundRobinPolicy`: a policy that yields nodes in a round-robin fashion.
- `TokenAwarePolicy`: a policy that yields replica nodes for a given partition key and keyspace. The token-aware policy uses a child policy to retrieve the next nodes in case the replicas for a partition key are not available.

Default load-balancing policy

The default load-balancing policy is the `TokenAwarePolicy` with `DCAwareRoundRobinPolicy` as a child policy. It may seem complex but it actually isn't: The policy yields local replicas for a given key and, if not available, it yields nodes of the local datacenter in a round-robin manner.

Setting the load-balancing policy

To use a load-balancing policy, you pass it in as a `clientOptions` object to the `Client` constructor.

```
// You can specify the local dc relatively to the node.js app
var localDatacenter = 'us-east';
var loadBalancingPolicy = new
  cassandra.policies.loadBalancing.DCAwareRoundRobinPolicy(localDatacenter);
var clientOptions = {
  policies : {
    loadBalancing : loadBalancingPolicy
  }
};
var client = new cassandra.Client(clientOptions);
```

Implementing a custom load-balancing policy

The built-in policies in the Node.js driver cover most common use cases. In the rare case that you need to implement your own policy you can do it by inheriting from one of the existent policies or the abstract `LoadBalancingPolicy` class.

You have to take into account that the same policy is used for all queries in order to yield the hosts in correct order.

The load-balancing policies are implemented using the Iterator Protocol, a convention for lazy iteration allowing to produce only the next value in the series without producing a full Array of values. Under ECMAScript 6 (Harmony), it enables you to use the new generators.

Example: A policy that selects every node except a specific one. Note that this policy is a sample and it is not intended for production use. Use multiple datacenter Cassandra clusters instead.

```
function BlackListPolicy(blackListedHost, childPolicy) {
  this.blackListedHost = blackListedHost;
  this.childPolicy = childPolicy;
}

util.inherits(BlackListPolicy, LoadBalancingPolicy);

BlackListPolicy.prototype.init = function (client, hosts, callback) {
  this.client = client;
  this.hosts = hosts;
  //initialize the child policy
  this.childPolicy.init(client, hosts, callback);
};

BlackListPolicy.prototype.getDistance = function (host) {
  return this.childPolicy.getDistance(host);
};

BlackListPolicy.prototype.newQueryPlan = function (keyspace, queryOptions,
  callback) {
  var self = this;
  this.childPolicy.newQueryPlan(keyspace, queryOptions, function (iterator)
  {
    callback(self.filter(iterator));
  });
}

BlackListPolicy.prototype.filter = function (childIterator) {
  var self = this;
  return {
```

```

    next: {
      var item = childIterator.next();
      if (item.address === self.blackListedHost) {
        //use the next one
        return this.next();
      }
      return item;
    }
  }
}

```

Reconnection policy

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

Description

The reconnection policy consists of one method:

- `#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:

- `#onReadTimeout(requestInfo, consistency, received, blockFor, isDataPresent)`
- `#onUnavailable(requestInfo, consistency, required, alive)`
- `#onWriteTimeout(requestInfo, consistency, received, blockFor, writeType)`

In version 1 of the driver, a default and base retry policy is included.

Cluster and schema metadata

Retrieving cluster topology and schema metadata from the driver.

You can retrieve the cluster topology and the schema metadata information. from the driver

After establishing the first connection, the driver retrieves the cluster topology details and exposes these through properties of the client object. This information is kept up to date using Cassandra event notifications.

Reference

The following example outputs hosts information about your cluster:

```
client.hosts.forEach(function (host) {
  console.log(host.address, host.datacenter, host.rack);
});
```

Additionally, the keyspaces information is already loaded into the **Metadata object**:

```
client.hosts.forEach(function (host) {
  console.log(host.address, host.datacenter, host.rack);
});
```

To retrieve the definition of a table, use the **Metadata#getTable()** method:

```
client.metadata.getTable('ks1', 'table1', function (err, tableInfo) {
  if (!err) {
    console.log('Table %s', table.name);
    table.columns.forEach(function (column) {
      console.log('Column %s with type %j', column.name, column.type);
    });
  }
});
```

When retrieving the same table definition concurrently, the driver queries once and invokes all callbacks with the retrieved information.

CQL data types to JavaScript types

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

Description

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

Table 3: JavaScript types to CQL data types

CQL data type	JavaScript type
ascii	String
bigint	Long
blob	Buffer
boolean	Boolean
counter	Long
decimal	BigDecimal
double	Number
float	Number
inet	InetAddress
int	Number

CQL data type	JavaScript type
list	Array
map	Object / ECMAScript 6 Map
set	Array / ECMAScript 6 Set
text	String
timestamp	Date
timeuuid	String
uuid	String
varchar	String
varint	Integer

Note: There are few data types defined in the ECMAScript standard for JavaScript. This usually represents a problem when you are trying to deal with data types that come from other systems.

Encoding data

When encoding data, on a normal execute with parameters, the driver tries to guess the target type based on the input type. Values of type `Number` will be encoded as `double` (because `Number` is double or IEEE 754 value).

Consider the following example:

```
var key = 1000;
client.execute('SELECT * FROM table1 where key = ?', [key], callback);
```

If the key column is of type `int`, the execution fails. There are two possible ways to avoid this type of problem:

Prepare the data (recommended)

Using prepared statements provides multiple benefits. Prepared statements are parsed and prepared on the Cassandra nodes and are ready for future execution. Also, the driver retrieves information about the parameter types which allows an accurate mapping between a JavaScript type and a Cassandra type.

Using the previous example, setting the prepare flag in the `queryOptions` will fix it:

```
// prepare the query before execution
client.execute('SELECT * FROM table1 where key = ?', [key], { prepare : true },
  callback);
```

When using prepared statements, the driver prepares the statement once on each host to execute multiple times.

Hinting the target types

Providing hints in the query options is another way around it.

```
// Hint: the first parameter is an integer
client.execute('SELECT * FROM table1 where key = ?', [key], { hints : ['int'] },
  callback);
```

Collections

List and Set

When reading columns with CQL `list` or `set` data types, the driver exposes them as native `Arrays`. When writing values to a `list` or `set` column, you can pass in a `Array`.

```
client.execute('SELECT list_val, set_val, double_val FROM tbl', function
  (err, result) {
    assert.ifError(err);
    console.log(Array.isArray(result.rows[0]['list_val'])); // true
    console.log(Array.isArray(result.rows[0]['set_val'])); // true
  });
```

Map

JavaScript objects are used to represent the CQL `map` data type in the driver, because JavaScript objects are associative arrays.

```
client.execute('SELECT map_val FROM tbl', function (err, result) {
  assert.ifError(err);
  console.log(JSON.stringify(result.rows[0]['map_val'])); //
  {"key1":1,"key2":2}
});
```

When using CQL maps, the driver needs a way to determine that the object instance passed as a parameter must be encoded as a map. Inserting a map as an object will fail:

```
var query = 'INSERT INTO tbl (id, map_val) VALUES (?, ?)';
var params = [id, {key1: 1, key2: 2}];
client.execute(query, params, function (err) {
  console.log(err) // TypeError: The target data type could not be guessed
});
```

To overcome this limitation, you should prepare your queries. Preparing and executing statements in the driver does not require chaining two asynchronous calls, you can set the `prepare` flag in the query options and the driver will handle the rest. The previous query, using the `prepare` flag, will succeed:

```
client.execute(query, params, { prepare: true }, callback);
```

ECMAScript Map and Set support

The new built-in types in ECMAScript 6, `Map` and `Set`, can be used to represent CQL `map` and `set` values. To enable this option, you should specify the constructors in the client options.

```
var options = {
  contactPoints: contactPoints,
  encoding: {
    map: Map,
    set: Set
  }
};
var client = new cassandra.Client(options);
```

This way, when encoding or decoding `map` or `set` values, the driver uses those constructors:

```
client.execute('SELECT map_val FROM tbl', function (err, result) {
```

```

    assert.ifError(err);
    console.log(result.rows[0]['map_val'] instanceof Map); // true
  });

```

Numerical values

The driver provides support for all the CQL numerical data types, such as `int`, `float`, `double`, `bigint`, `varint`, and `decimal`. There is only one numerical data type in ECMAScript standard, `Number`, which is a type representing a double-precision 64-bit value. It is used by the driver to handle `double`, `float`, and `int` values.

int, float, and double data types

JavaScript provides methods to operate with `Numbers` (that is, IEEE 754 double-precision floats) and built-in operators (sum, subtraction, division, bitwise, etc), making it a good fit for CQL data types `int`, `float`, and `double`.

When decoding any of these data type values, it is returned as a `Number`.

```

client.execute('SELECT int_val, float_val, double_val FROM tbl', function
(err, result) {
  assert.ifError(err);
  console.log(typeof result.rows[0]['int_val']); // Number
  console.log(typeof result.rows[0]['float_val']); // Number
  console.log(typeof result.rows[0]['double_val']); // Number
});

```

When encoding the data, the driver tries to encode a `Number` as `double` because it can not automatically determine if is dealing with an `int`, a `float`, or a `double`.

Inserting a `Number` value as a `double` succeeds:

```

var query = 'INSERT INTO tbl (id, double_val) VALUES (?, ?)';
client.execute(query, [id, 1.2], callback);

```

But doing the same with a `float` fails:

```

var query = 'INSERT INTO tbl (id, float_val) VALUES (?, ?)';
client.execute(query, [id, 1.2], function (err) {
  console.log(err) // ResponseError: Expected 4 or 0 byte value for a
float (8)
});

```

Trying to do the same with an `int`, also fails because Cassandra expects a `float` or an `int`, and the driver sent a 64-bit `double`.

```

var query = 'INSERT INTO tbl (id, int_val) VALUES (?, ?)';
client.execute(query, [id, 1], function (err) {
  console.log(err) // ResponseError: Expected 4 or 0 byte int (8)
});

```

To overcome this limitation, you should prepare your queries. Because preparing and executing statements in the driver does not require chaining two asynchronous calls, you can set the `prepare` flag in the query options and the driver handles the rest.

Reference

The previous query, using the prepare flag, succeeds no matter if it is an int, float, or double:

```
var query = 'INSERT INTO tbl (id, int_val) VALUES (?, ?)';
client.execute(query, [id, 1], { prepare: true }, callback);
```

decimal data type

The `BigDecimal` class provides support for representing the CQL decimal data type, because JavaScript has no built-in arbitrary precision decimal representation.

```
var BigDecimal = require('cassandra-driver').types.BigDecimal;
var value1 = new BigDecimal(153, 2);
var value2 = BigDecimal.fromString('1.53');
console.log(value1.toString()); // 1.53
console.log(value2.toString()); // 1.53
console.log(value1.equals(value2)); // true
```

The driver decodes CQL decimal data type values as instances of `BigDecimal`.

```
client.execute('SELECT decimal_val FROM users', function (err, result) {
  assert.ifError(err);
  console.log(result.rows[0]['decimal_val'] instanceof BigDecimal); //
  true
});
```

bigint data type

The `Long` class provides support for representing the CQL bigint data type, because JavaScript has no built-in 64-bit integer representation.

```
var Long = require('cassandra-driver').types.Long;
var value1 = Long.fromNumber(101);
var value2 = Long.fromString('101');
console.log(value1.toString()); // 101
console.log(value2.toString()); // 101
console.log(value1.equals(value2)); // true
console.log(value1.add(value2).toString()); // 202
```

The driver decodes CQL bigint data type values as instances of `Long`.

```
client.execute('SELECT bigint_val FROM users', function (err, result) {
  assert.ifError(err);
  console.log(result.rows[0]['bigint_val'] instanceof Long); // true
});
```

varint data type

The `Integer` class, originally part of the Google Closure math library, provides support for representing CQ: varint data type values, because JavaScript has no arbitrarily large signed integer representation.

```
var Integer = require('cassandra-driver').types.Integer;
var value1 = Integer.fromNumber(404);
var value2 = Integer.fromString(404);
```

```

console.log(value1.toString());           // 404
console.log(value2.toString());           // 404
console.log(value1.equals(value2));        // true
console.log(value1.add(value2).toString()); // 808

```

The driver decodes CQL `varint` data type values as instances of `Integer`.

```

client.execute('SELECT varint_val FROM users', function (err, result) {
  assert.ifError(err);
  console.log(result.rows[0]['varint_val'] instanceof Integer); // true
});

```

User-defined types

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

An 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.

With the Node.js driver, you can retrieve and store UDTs using JavaScript objects.

For example, given the following UDT and table:

```

CREATE TYPE address (
  street text,
  city text,
  state text,
  zip int,
  phones set<text>
);

CREATE TABLE users (
  name text PRIMARY KEY,
  email text,
  address frozen<address>
);

```

You retrieve the user address details as a regular JavaScript object.

```

var query = 'SELECT name, email, address FROM users WHERE id = ?';
client.execute(query, [name], { prepare: true }, function (err, result) {
  var row = result.first();
  var address = row.address;
  console.log('User lives in %s - %s', address.city, address.state);
});

```

You modify the address using JavaScript objects as well:

```

var address = {
  city: 'Santa Clara',
  state: 'CA',
  street: '3975 Freedom Circle',
  zip: 95054,
  phones: ['650-389-6000']
};

```

Reference

```
};
var query = 'UPDATE users SET address = ? WHERE id = ?';
client.execute(query, [address, name], { prepare: true }, callback);
```

Setting the `prepare` flag is recommended because it helps the driver to accurately map the UDT fields from and to object properties.

You can provide JavaScript objects as parameters without setting the `prepare` flag, but you must provide the parameter hint (`udt<address>`) and initially the driver makes an extra roundtrip to the cluster to retrieve the UDT metadata.

Nesting user-defined types in CQL

User defined types can be nested arbitrarily. Here is an example based on the schema used in the previous example, but with the `phones` column changed from `set<text>` to a `set<frozen<phone>>`. The phone UDT contains an alias, a `phone_number` and a `country_code`.

```
CREATE TYPE phone (
  alias text,
  phone_number text,
  country_code int
);

CREATE TYPE address (
  street text,
  city text,
  state text,
  zip int,
  phones set<frozen<phone>>
);

CREATE TABLE users (
  name text PRIMARY KEY,
  email text,
  address frozen<address>
);
```

You access the UDT fields in the same way, but as nested JavaScript objects.

```
var query = 'SELECT name, email, address FROM users WHERE id = ?';
client.execute(query, [name], { prepare: true }, function (err, result) {
  var row = result.first();
  var address = row.address;
  // phones is an Array of Objects
  address.phones.forEach(function (phone) {
    console.log('Phone %s: %s', phone.alias, phone.phone_number);
  });
});
```

UUID and time-based UUID data types

The driver provides ways to generate and decode UUIDs and time-based UUID.

Uuid

The `Uuid` class provides support for representing Cassandra `uuid` data type. To generate a version 4 unique identifier, use the `Uuid` static method `random()`:

```
var Uuid = require('cassandra-driver').types.Uuid;
var id = Uuid.random();
```

The driver decodes Cassandra `uuid` data type values as an instances of `Uuid`.

```
client.execute('SELECT id FROM users', function (err, result) {
  assert.ifError(err);
  console.log(result.rows[0].id instanceof Uuid); // true
  console.log(result.rows[0].id.toString());      // xxxxxxxx-xxxx-xxxx-
  xxx-xxxxxxxxxxxxx
});
```

You can also parse a string representation of a `uuid` into a `Uuid` instance:

```
var id = Uuid.fromString(stringValue);
console.log(id instanceof Uuid);          // true
console.log(id.toString() === stringValue); // true
```

TimeUuid

The `TimeUuid` class provides support for representing Cassandra `timeuuid` data type. To generate a time-based identifier, you can use the `now()` and `fromDate()` static methods:

```
var TimeUuid = require('cassandra-driver').types.TimeUuid;
var id1 = TimeUuid.now();
var id2 = TimeUuid.fromDate(new Date());
```

The driver decodes CQL `timeuuid` data type values as instances of `TimeUuid`.

```
client.execute('SELECT id, timeid FROM sensor', function (err, result) {
  assert.ifError(err);
  console.log(result.rows[0].timeid instanceof TimeUuid); // true
  console.log(result.rows[0].timeid instanceof Uuid); // true, it inherits
  from Uuid
  console.log(result.rows[0].timeid.toString());          // <- xxxxxxxx-xxxx-
  xxx-xxxx-xxxxxxxxxxxxx
  console.log(result.rows[0].timeid.getDate());          // <- Date stored in
  the identifier
});
```

You can specify the other parts of the identifier, such as the node and the clock sequence, or the 100-nanosecond precision value of the date using the optional parameters in `fromDate()` method.

```
var ticks = 9123; // A number from 0 to 9999
var id = TimeUuid.fromDate(new Date(), ticks, node, clock);
```

Tuples

Cassandra 2.1 introduced a `tuple` type for CQL.

A tuple is a fixed-length set of typed positional fields. With the driver, you retrieve and store tuples using the `Tuple` class. For example, given the following table to represent the value of the exchange between two currencies.

Reference

```
CREATE TABLE forex (  
  name text,  
  time timeuuid,  
  currencies frozen<tuple<text, text>>,  
  value decimal,  
  PRIMARY KEY (name, time)  
);
```

You retrieve the Tuple value.

```
var query = 'SELECT name, time, currencies, value FROM forex where name  
= ?';  
client.execute(query, [name], { prepare: true }, function (err, result) {  
  result.rows.forEach(function (row) {  
    console.log('%s to %s: %s', row.currencies.get(0),  
      row.currencies.get(1), row.value);  
  });  
});
```

You use the `get(index)` method to obtain the value at any position or the `values()` method to obtain an Array representation of the Tuple.

To create a new Tuple, you use the constructor providing the values as parameters.

```
var Tuple = require('cassandra-driver').types.Tuple;  
// Create a new instance of a Tuple.  
var currencies = new types.Tuple('USD', 'EUR');  
var query = 'INSERT INTO forex (name, time, currencies, value) VALUES  
(?, ?, ?, ?)';  
var params = ['market1', TimeUuid.now(), currencies, new BigDecimal(1, 0)];  
client.execute(query, params, { prepare: true }, callback);
```

Fetching large result sets

The single-threaded nature of Node.js should be taken into consideration when dealing with retrieving large amount of rows. If an asynchronous method uses large buffers, it could cause large memory consumption and a poor response time.

Because of this, the driver exposes the `#eachRow()` and `#stream()` methods. When using these methods, the driver will parse the rows and yield them to the user as they come through the network. All methods use a default `fetchSize` of 5000 rows, retrieving only the first page of results up to a maximum of 5000 rows.

Automatic paging

If you want to retrieve the next pages, you can do it by setting `autoPage: true` in the `queryOptions` to automatically request the next page, once it finished parsing the previous page.

```
// Imagine a column family with millions of rows.  
var query = 'SELECT * FROM targetable';  
client.eachRow(query, [], { autoPage : true }, function(n, row) {  
  // This function will be called per each of the rows in all the table.  
  }, callback);
```


Manual paging

If you want to retrieve the next page of results only when you ask for it (that is, a web pager), you use the `pageState` made available by the end callback in `#eachRow()`. A non-null `pageState` value means that there are additional result pages.

```
client.eachRow(query, [], { prepare : 1 , fetchSize : 1000 }, function (n,
  row) {
    // Row callback.
  }, function (err, result) {
    // End callback.
    // Store the paging state.
    pageState = result.pageState;
  }
);
```

In the following kinds of requests, use the page state.

```
// Use the pageState in the queryOptions to continue where you left it.
client.eachRow(query, [], { pageState : pageState, prepare : 1 ,
  fetchSize : 1000 }, function (n, row) {
  // Row callback.
}, function (err, result) {
  // End callback.
  // Store the next paging state.
  pageState = result.pageState;
}
);
```

Parameterized queries

Bind values to parameters either by position or by named markers.

You can bind the values of parameters in a prepared statement either by *position* or by using *named* markers.

Positional parameterized query

When using positional parameters, the query parameters must be provided as an Array.

```
var query = 'INSERT INTO artists (id, name) VALUES (?, ?)';
// Parameters by marker position
var params = ['krichards', 'Keith Richards'];
client.execute(query, params, { prepare: true }, callback);
```

Named parameterized query

You declare the named markers in your queries and use a JavaScript object properties to define the parameters, with the Object property names matching the parameters names.

```
var query = 'INSERT INTO artists (id, name) VALUES (:id, :name)';
// Parameters by marker name
var params = { id: 'krichards', name: 'Keith Richards' };
client.execute(query, params, { prepare: true }, callback);
```

Reference

Defining named markers in your queries is supported in Cassandra 2.0 or greater for prepared statements and only Cassandra 2.1 or greater for non-prepared statements.

Routing queries

Specify the parameter values that compose the partition key of the nodes to be used as coordinators for the query.

When using the `TokenAwarePolicy`, you can specify the parameter values that compose the partition key to tell the driver which nodes should be used as coordinators for the query.

Examples

Consider a table `users` that has a single partition key, `id`. You can indicate the position of the parameter that forms the partition key.

```
var query = 'INSERT INTO users (id, name) VALUES (?, ?)';
var params = [ Uuid.random(), 'Paul'];
// The parameter at index 0 is the partition key
client.execute(query, params, { routingIndexes: [0] }, callback);
```

In the same way, for a table that has multiple partition keys:

```
var query = 'INSERT INTO temperature (name, slot, value, date) VALUES
  (?, ?, ?, ?)';
var params = [ 'Sensor 01', '201501', temperatureValue, new Date()];
// The parameters at index 0 and 1 form the partition key
client.execute(query, params, { routingIndexes: [0, 1] }, callback);
```

When using `named parameters`, you can also specify the routing parameters by name.

```
var query = 'INSERT INTO temperature (value, name, slot, date) VALUES
  (:value, :name, :slot, :date)';
var params = {value: temperatureValue, name: 'Sensor 01', slot: '201501',
  date: new Date()};
// The parameters 'name' and 'slot' form the partition key
client.execute(query, params, { routingNames: ['name', 'slot'] }, callback);
```

FAQ

Which versions of Cassandra does the driver support?

Version 2.1 of the driver supports any Cassandra version greater than 1.2 and above.

Which versions of CQL does the driver support?

It supports [CQL version 3](#).

How do I generate a uuid or a timebased uuid?

Use the `Uuid` and `TimeUuid` classes inside the `types` module. For example:

```
var id = cassandra.types.Uuid.random();
```

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

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

Should I shut down the pool after executing a query?

No, only call `client.shutdown()` once in your application's lifetime.










API reference

DataStax Node.js 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](#)