C# Driver 2.5 for Apache Cassandra

Documentation

${ds.localized.time}

Contents

About the C# driver

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

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

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

What's new in 2.5?

Here are the new and noteworthy features of the C# 2.5 driver:

- Routing API more user friendly
- Manual paging
- LINQ component
  - support for *lightweight* transactions
  - collection append and prepend
  - CQL functions support (token, maxtimeuuid, mintimeuuid)
  - ignore properties
  - LINQ mapping of anonymous types
  - fluent mapping configuration
- Mapper component: a lightweight object mapper for Apache Cassandra, originally developed by Luke Tillman as separated project cqlpoco.

What's new in 2.1?

Here are the new and noteworthy features of the C# 2.1 driver:

- **Cassandra 2.1 support**
  - User-defined types (UDT)
  - Named parameters for statements
  - When using the version 3 protocol, only one connection is opened per-host, and throughput is improved due to reduced pooling overhead and lock contention.
  - Tuples
- **New features**
  - New reconnection policy `FixedReconnectionPolicy`
  - The C# driver uses the most performant socket interface and (lock-free) asynchronous I/O, resulting in a significant read / write performance improvement for highly concurrent scenarios.

Note: The C# 2.1 driver works with Apache Cassandra 2.1, 2.0, and 1.2 and DataStax Enterprise 4.5, 4.0, 3.2, and 3.1.

Architecture

The C# Driver 2.5 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 works with both Cassandra 1.2 and 2.0.

Core features

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

Architectural overview

The driver architecture is a layered one. At the bottom lies the driver core. This core, located in `Cassandra.dll` assembly, 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.

Queries can be executed synchronously or asynchronously, prepared statements are supported, and LINQ can be used to embed queries directly into C# code.

The driver contains four components that you can choose from to determine how your client interacts with Cassandra nodes.

- Core
- Mapper
- LINQ
- ADO.NET

The driver and its dependencies

The C# driver only supports the Cassandra Binary Protocol and CQL3

Cassandra binary protocol

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

```
start_native_transport: true
```

Then restart the node.

Cassandra compatibility

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

	C# driver 1.0.x	**C# driver 2.0.x**
Cassandra 1.2.x	Compatible	Compatible
Cassandra 2.0.x	Compatible for Cassandra 1.0 API and commands	Compatible

If you try to use any Cassandra 2.0 features with Cassandra 1.2, the driver throws a `NotSupportedException`.

Installing the driver with NuGet package manager

You install the latest version of the driver from the NuGet Gallery:

```
PM> Install-Package CassandraCSharpDriver
```

Writing your first client

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

Connecting to a Cassandra cluster

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

Before you begin

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

Apache Cassandra
ccm (optional)
Microsoft Visual Studio
Package Manager Console for Visual Studio
DataStax C# driver for Apache Cassandra

About this task

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

Procedure

1. In the Visual Studio IDE, create a console application project.
   In the New Project dialog, use the following data:
   - template: **Installed** > **Visual C#** > **Console Application**
   - .NET framework: 4.5
   - Name: CassandraApplication
   - Location: `<your-projects-directory>`
   - Solution: Create new solution
   - Solution name: CassandraApplication
2. In the **Package Manager Console** window install the C# driver.
   a) Select **Tools** > **Library Package Manager** > **Package Manager Console**.
      The Package Manager Console opens at the bottom of the IDE.
   b) Type the following to install the C# driver:
      Install-Package CassandraCSharpDriver

   The C# driver is installed.
3. Create a new C# class, SimpleCassandraApplication.SimpleClient.
   a) Right-click on the SimpleCassandraApplication node in the **Solution Explorer** and select **Add** > **New Item**.
   b) In the **Add New Item** dialog, add the following for your new class:
      - Select **Installed** > **Visual C# Items** > **Class**.
      - Name: SimpleClient.cs
   c) Select **Add**.

4. Create a new C# class, `CassandraApplication.SimpleClient`.

a) Add a using directive that references the Cassandra namespace.

b) Add a read-only property, `Cluster`, to hold a Cluster reference.

```
public Cluster Cluster { get; private set; }
```

c) Add an instance method member, `Connect`, to your new class.

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

The Connect method:

- adds a contact point (node IP address) using the `Builder` class
- builds a cluster instance
- retrieves metadata from the cluster:
  - the name of the cluster
  - the datacenter, host name or IP address, and rack for each of the nodes in the cluster

```
public void Connect(String node)
{
    Cluster = Cluster.Builder()
             .AddContactPoint(node)
             .Build();
    Console.WriteLine("Connected to cluster: " +
 Cluster.Metadata.ClusterName.ToString());
    foreach (var host in Cluster.Metadata.AllHosts())
    {
        Console.WriteLine("Data Center: " + host.Datacenter + ", " +
            "Host: " + host.Address + ", " +
            "Rack: " + host.Rack);
    }
}
```

d) Add an instance method member, `Close`, to shut down the cluster instance once you are finished with it.

```
public void Close()
{
    Cluster.Shutdown();
}
```

e) In the CassandraApplication.Program class `Main` method instantiate a `SimpleClient` object, call `Connect` on it, and then `Close`.

```
static void Main(String[] args)
{
   SimpleClient client = new SimpleClient();
   client.Connect("127.0.0.1");
   Console.ReadKey(); // pause the console before exiting
   client.Close();
}
```

Code listing

The complete code listing illustrates:

- connecting to a cluster

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

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using Cassandra;

namespace CassandraExamples
{
    public class SimpleClient
    {
        public Cluster Cluster { get; private set; }

        public void Connect(String node)
        {
            cluster = Cluster.Builder()
                   .AddContactPoint(node)
                   .Build();
            Console.WriteLine("Connected to cluster: " +
 Cluster.Metadata.ClusterName.ToString());
            foreach (var host in Cluster.Metadata.AllHosts())
            {
                Console.WriteLine("Data Center: " + host.Datacenter + ", " +
                    "Host: " + host.Address + ", " +
                    "Rack: " + host.Rack);
            }
        }

        public void Close()
        {
            Cluster.Shutdown();
        }

        public static void Main(String[] args)
        {
            SimpleClient client = new SimpleClient();
            client.Connect("127.0.0.1");
            Console.ReadKey(); // pause the console before exiting
            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
```

Press any key to quit your application.

Using a session to execute CQL statements

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

Before you begin

This tutorial uses a CQL3 schema which is described in a post on the DataStax developer blog. Reading that post, could help with some of the new CQL3 concepts used here.

About this task

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

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

Procedure

1. Modify your `SimpleClient` class.
   a) Add an `ISession` read-only property.

      ```
      public ISession Session { get; private set; }
      ```

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

      ```
      Session = Cluster.Connect();
      ```

   You can execute statements 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)" +
            ");");
```

4. Add a virtual instance method, `LoadData`, to the `SimpleCient` class implementation.

   ```
   public virtual void LoadData() { }
   ```

   Declare the `LoadData` method to be virtual because it will be overridden in the `BoundStatementClient` later in this tutorial.
5. Add the code to insert data into the new schema.

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

6. In the `CassandraApplication.Program` class `Main` method, add calls to the `CreateSchema` and `LoadData` methods (after the `Connect` method).

```
client.CreateSchema();
client.LoadData();
```

7. To the `SimpleClient` class), add an instance method, `QuerySchema`, that executes a SELECT statement on the tables and then prints out the results.

a) Add a virtual instance method, `QuerySchema`, to the `SimpleCient` class implementation.

```
public virtual void QuerySchema() { }
```

Declare the `QuerySchema` method to be virtual because it will be overridden in the `BoundStatementClient` and `AsynchronousClient` classes later in this document.

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

```
RowSet results = _session.Execute("SELECT * FROM playlists " +
        "WHERE id = 2cc9ccb7-6221-4ccb-8387-f22b6a1b354d;");
```

The `Execute` method returns a `RowSet` object that holds rows returned by the SELECT statement.

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

```
Console.WriteLine(String.Format("{0, -30}\t{1, -20}\t{2, -20}\t{3,
 -30}",
    "title", "album", "artist", "tags"));
Console.WriteLine("-------------------------------
+-----------------------+--------------------
+-------------------------------");
foreach (Row row in results.GetRows())
{
    Console.WriteLine(String.Format("{0, -30}\t{1, -20}\t{2, -20}\t{3}",
        row.GetValue<String>("title"), row.GetValue<String>("album"),
        row.GetValue<String>("artist"),
 row.GetValue<List<String>>("tags").ToString()));
}
```

8. Add a new instance method, `DropSchema` method, that drops a keyspace from the schema.

```
public void DropSchema()
{
    Session.Execute ( "DROP KEYSPACE " + keyspace );
    Console.WriteLine ( "Finished dropping " + keyspace + " keyspace." );
}
```

9. Add a new instance method, `Close` that closes the cluster and disposes of the session.

```
public void Close()
{
    Cluster.Shutdown();
    Session.Dispose();
}
```

10. To the `CassandraApplication.Program` class `Main` method, add calls to the new `QuerySchema`, `DropSchema`, and `Close` methods.

```
client.QuerySchema();
```

Using bound statements

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

About this task

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

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

Procedure

1. Create a new class, `BoundStatementsClient` which extends `SimpleClient`, to the `CassandraApplication` solution.
2. Add a using directive for `Cassandra`.

   ```
   using Cassandra;
   ```

3. Add a new method, `PrepareStatements`, and implement it.

   a) Add two properties, `InsertSongPreparedStatement` and `InsertPlaylistPreparedStatement`, to hold references to the two prepared statements you will use in the `LoadData` method.

   ```
   private PreparedStatement InsertSongPreparedStatement;
   private PreparedStatement InsertPlaylistPreparedStatement;
   ```

   b) In the `PrepareStatements` method body add the following code:

   ```
   InsertSongPreparedStatement = Session.Prepare(
       "INSERT INTO simplex.songs " +
       "(id, title, album, artist, tags) " +
       "VALUES (?, ?, ?, ?, ?);");
   InsertPlaylistPreparedStatement = Session.Prepare(
       "INSERT INTO simplex.playlists " +
       "(id, song_id, title, album, artist) " +
       "VALUES (?, ?, ?, ?, ?);");
   ```

   Note: You only need to prepare a statement once per session.

4. Add a new method, `LoadData`, and implement it.

   ```
   public override void LoadData() { }
   ```

5. Add a collection to hold the tags which will be bound to the prepared statement.

   ```
   HashSet<String> tags = new HashSet<String>();
   tags.Add("jazz");
   tags.Add("2013");
   ```

6. 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 = InsertSongPreparedStatement.Bind(
   new Guid("756716f7-2e54-4715-9f00-91dcbea6cf50"),
   "La Petite Tonkinoise'",
   "Bye Bye Blackbird'",
   "Joséphine Baker",
   tags);
Session.Execute(boundStatement);
```

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

```
public override void LoadData()
{
   HashSet<String> tags = new HashSet<String>();
   tags.Add("jazz");
   tags.Add("2013");
   BoundStatement boundStatement = InsertSongPreparedStatement.Bind(
      new Guid("756716f7-2e54-4715-9f00-91dcbea6cf50"),
      "La Petite Tonkinoise'",
      "Bye Bye Blackbird'",
      "Joséphine Baker",
      tags);
   Session.Execute(boundStatement);
   boundStatement = InsertPlaylistPreparedStatement.Bind(
      new Guid("2cc9ccb7-6221-4ccb-8387-f22b6a1b354d"),
      new Guid("756716f7-2e54-4715-9f00-91dcbea6cf50"),
      "La Petite Tonkinoise",
      "Bye Bye Blackbird",
      "Joséphine Baker");
   Session.Execute(boundStatement);
}
```

8. In the `Main` method of the `CassandraApplication.Program` class, replace the line which instantiates a client from the `SimpleClient` class to the `BoundStatementsClient` and add a call to the `PrepareStatements` method after you have created the schema.

```
static void Main(String[] args)
{
   BoundStatementsClient client = new BoundStatementsClient();
   client.Connect("127.0.0.1");
   client.CreateSchema();
   client.PrepareStatements();
   client.LoadData();
   client.QuerySchema();
   Console.ReadKey();
   client.DropSchema("simplex");
   client.Close();
   client.Close();
}
```

C# driver reference

Reference for the C# driver.

Asynchronous statement execution

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.

The 1.0 version of the driver used a different pattern for asynchronous execution of CQL statements: by calling `BeginExecute` and `EndExecute` methods. Now the driver has added an `ExecuteAsync` method to its API. This uses the .NET 4.5 framework TAP (Task-based Asynchronous Pattern).

Calling `ExecuteAsync` returns a `Task<RowSet>` object immediately without blocking the calling method. In the example code below, a lambda expression is passed to the `ContinuesWith` method. When the CQL statement has finished execution, the anonymous `Func` is called and the `RowSet` is printed out.

Example code

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using Cassandra;

namespace CassandraApplication
{
    class AsynchronousClient : SimpleClient
    {
        public AsynchronousClient() { }

        public override void QuerySchema()
        {
            Statement statement = new SimpleStatement("SELECT * FROM
 simplex.songs;");
            var task = Session.ExecuteAsync(statement);
            task.ContinueWith((asyncTask) =>
            {
                Console.WriteLine(String.Format("{0, -30}\t{1, -20}\t{2,
 -20}\t{3, -30}",
                    "title", "album", "artist", "tags"));
                Console.WriteLine("-------------------------------
+-----------------------+--------------------
+-------------------------------");
                foreach (var row in asyncTask.Result)
                {
                    Console.WriteLine(String.Format("{0, -30}\t{1, -20}\t{2,
 -20}\t{3}",
                        row.GetValue<String>("title"),
 row.GetValue<String>("album"),
                        row.GetValue<String>("artist"),
 Prettify(row.GetValue<List<String>>("tags")
                        )));
                }
            });
            task.Wait();
        }


        internal String Prettify(IEnumerable<String> collection)
        {
            StringBuilder result = new StringBuilder("[ ");
            foreach (var item in collection)
            {
                result.Append(item);
                result.Append(" ");
            }
            result.Append("]");
            return result.ToString();
        }
    }
}
```

Automatic failover

Description

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

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();
}
```

Binary protocol frame change: 2-byte stream id

Starting with Cassandra 2.1, the binary protocol (version 3) the frame stream id is two bytes, allowing for up to 32K requests without waiting for a response. This is the same level of concurrency that was previously possible using multiple connections to the same host.

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 Initialize(Cluster cluster)`: initializes the policy. The driver calls this method only once and before any other method calls are made.
- `IEnumerable<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:

- `IReconnectionSchedule NewSchedule()`: creates a new schedule to use in reconnection attempts.

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

- `ConstantReconnectionPolicy`
- `ExponentialReconnectionPolicy`
- `FixedReconnectionPolicy`

The new `FixedReconnectionPolicy` allows you to specify custom delays for each reconnection attempt. This policy is specially suitable for small clusters where you want to have fine-grained control over the reconnection delays.

FixedReconnectionPolicy sample

```
// When building a cluster, set the reconnection policy to
// Don’t wait to reconnect the first attempt (0 ms)
// Wait 5 seconds for the seconds reconnection attempt (5000 ms)
// Wait 2 minutes for the third (2 * 60000 ms)
// Wait 1 hour for the following attempts (60 * 60000 ms)
Cluster.Builder()
   .WithReconnectionPolicy(new FixedReconnectionPolicy(0, 5000, 2 * 60000,
 60 * 60000)
```

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:

- `RetryDecision OnReadTimeout(Query query, ConsistencyLevel cl, int requiredResponses, int receivedResponses, boolean dataRetrieved, int nbRetry)`
- `RetryDecision OnUnavailable(Query query, ConsistencyLevel cl, int requiredReplica, int aliveReplica, int nbRetry)`
- `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 1: Protocol options

Option	Description	Default
Port	The port to connect to a Cassandra node on.	9042

Option	Description	Default
Compression	What kind of compression to use when sending data to a node: either no compression or snappy. Snappy compression is optimized for high speeds and reasonable compression.	CompressionType.NoCompression

Pooling options

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

Description

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

Table 2: Connection pooling options

Option	Description	Default value
CoreConnectionsPerHost	The core number of connections per host.	2 for HostDistance.LOCAL, 1 for HostDistance.REMOTE
MaxConnectionPerHost	The maximum number of connections per host.	8 for HostDistance.LOCAL, 2 for HostDistance.REMOTE
MaxSimultaneousRequestsPerConnectionTreshold	The number of simultaneous requests on all connections to an host after which more connections are created.	128
MinSimultaneousRequestsPerConnectionTreshold	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

These properties represent low-level socket options.

Table 3: Pooling options

Option	Description
ConnectTimeoutMillis	The connect timeout in milliseconds for the underlying I/O channel.
KeepAlive	The amount of time before you send your peer a keepalive probe packet with no data in it and the ACK flag turned on.
ReceiveBufferSize	A hint on the size of the buffer used to receive data.

Option	Description
ReuseAddress	Whether to allow the same port to be bound to multiple times.
SendBufferSize	A hint on the size of the buffer used to send data.
SoLinger	When specified, disables the immediate return from a call to close() on a TCP socket.
TcpNoDelay	Disables Nagle's algorithm on the underlying socket.

CQL data types to C# types

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

Description

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

Table 4: C# classes to CQL data types

CQL3 data type	C# type
ascii	Encoding.ASCII string
bigint	long
blob	byte[]
boolean	bool
counter	long
custom	Encoding.UTF8 string
decimal	float
double	double
float	float
inet	IPEndPoint
int	int
list	IEnumerable<T>
map	IDictionary<K, V>
set	IEnumerable<T>
text	Encoding.UTF8 string
timestamp	System.DateTimeOffset
timeuuid	System.Guid
uuid	System.Guid
varchar	Encoding.UTF8 string

CQL3 data type	C# type
varint	System.Numerics.BigInteger (.NET 4.0), byte[] (.NET 3.5)

Debugging

You have several options to help in debugging your application.

On the client side, you can use your IDE's debugging feature.

If you are using DataStax Enterprise, you can store the log4j messages into a Cassandra cluster.

Driver components

The driver contains four different components that you can choose to interact with Cassandra nodes.

Core component

The core component is responsible for maintaining a pool of connections to the cluster and executes the statements based on client configuration.

The core component is responsible for maintaining a pool of connections to the cluster and executes the statements based on client configuration.

Even though the core component allows low-level fine tuning, (for example, load-balancing policies to determine which node to use for each query), you interact using high-level objects like the `Session` that represents a pool of connections to the Cassandra cluster.

The other three components use the core component to execute statements and to handle the encoding and decoding of data.

The core component allows for fine tuning for performance, provides both sync and async APIs, and is used by all the others driver components.

Example

```
var cluster = Cluster.Builder()
   .AddContactPoints("host1", "host2", "host3")
   .Build();
var session = cluster.Connect("sample_keyspace");
var rs = session.Execute("SELECT * FROM sample_table");
foreach (var row in rs)
{
   var value = row.GetValue<int>("sample_int_column");
   //do something with the value
}
```

Mapper component

The Mapper component handles the mapping of CQL table columns to fields in your classes.

The Mapper component handles the mapping of CQL table columns to fields in your classes.

The Mapper component (previously known as cqlpoco) is a lightweight object mapper for Apache Cassandra. It lets you write queries with CQL, while it takes care of mapping rows returned from Cassandra to your classes. It was inspired by PetaPoco, NPoco, Dapper.NET and the cqlengine project.

To use the Mapper:

1. Add the following `using` statement to your class:

   ```
   using Cassandra.Mapping;
   ```
2. Retrieve an `ISession` instance in the usual way and reuse that session within all the classes in your client application.
3. Instantiate a Mapper object using its constructor:

   ```
   IMapper mapper = new Mapper(session);
   ```

New Mapper instances can be created each time they are needed, as short-lived instances, as long as you are reusing the same Session instance.

The Mapper works by mapping the column names in your CQL statement to the property names on your classes.

For example:

```
public class User
{
   public Guid UserId { get; set; }
   public string Name { get; set; }
}

// Get a list of users from Cassandra
IEnumerable<User> users = mapper.Fetch<User>("SELECT userid, name FROM
 users");
IEnumerable<User> users = mapper.Fetch<User>("SELECT * FROM users WHERE name
 = ?", someName);
```

Simple scenarios such as this are possible without doing any further mapping configuration. When using parameters, use query markers (?) instead of hardcoded stringified values, this improves serialization performance and lower memory consumption.

The Mapper will create new instances of your classes using the parameterless constructor.

Configuring mappings

In many scenarios, you need more control over how your class maps to a CQL table. You have two ways of configuring the Mapper:

- decorate your classes with attributes
- define mappings in code using the fluent interface

An example using the fluent interface:

```
MappingConfiguration.Global.Define(
   new Map<User>()
      .TableName("users")
      .PartitionKey(u => u.UserId)
      .Column(u => u.UserId, cm => cm.WithName("id")));
```

You can also create a class to group all your mapping definitions.

```
public class MyMappings : Mappings
{
   public MyMappings()
   {
       // Define mappings in the constructor of your class
```

```
         // that inherits from Mappings
         For<User>()
            .TableName("users")
            .PartitionKey(u => u.UserId)
            .Column(u => u.UserId, cm => cm.WithName("id")));
         For<Comment>()
            .TableName("comments");
      }
   }
```

Then, you can assign the mappings class in your configuration.

```
MappingConfiguration.Global.Define<MyMappings>();
```

Mapper API example

A simple query example is great, but the `Mapper` has many other methods for doing things like Inserts, Updates, Deletes, selecting a single record and more. And all methods have async counterparts. Here's a quick sampling.

```
// All query methods (Fetch, Single, First, etc.) will auto generate
// the SELECT and FROM clauses if not specified.
IEnumerable<User> users = mapper.Fetch<User>();
IEnumerable<User> users = mapper.Fetch<User>("FROM users WHERE name = ?",
 someName);
IEnumerable<User> users = mapper.Fetch<User>("WHERE name = ?", someName);

// Single and SingleOrDefault for getting a single record
var user = mapper.Single<User>("WHERE userid = ?", userId);
var user = mapper.SingleOrDefault<User>("WHERE userid = ?", userId);

// First and FirstOrDefault for getting first record
var user = mapper.First<User>("SELECT * FROM users");
var user = mapper.FirstOrDefault<User>("SELECT * FROM users");

// All query methods also support "flattening" to just the column's value
 type when
// selecting a single column
Guid userId = mapper.First<Guid>("SELECT userid FROM users");
IEnumerable<string> names = mapper.Fetch<string>("SELECT name FROM users");

// Insert a POCO var newUser = new User { UserId = Guid.NewGuid(), Name =
 "SomeNewUser" };
mapper.Insert(newUser);

// Update with POCO someUser.Name = "A new name!";
mapper.Update(someUser);

// Update with CQL (will prepend table name to CQL)
mapper.Update<User>("SET name = ? WHERE id = ?", someNewName, userId);

// Delete with POCO
mapper.Delete(someUser);

// Delete with CQL (will prepend table name to CQL)
mapper.Delete<User>("WHERE id = ?", userId);
```

LINQ component

The LINQ component of the driver is an implementation of LINQ `IQueryProvider` and `IQueryable<T>` interfaces that allows you to write CQL queries in LINQ and read the results using your object model.

The LINQ component of the driver is an implementation of LINQ `IQueryProvider` and `IQueryable<T>` interfaces that allows you to write CQL queries in LINQ and read the results using your object model.

When you execute a LINQ statement, the component translates language-integrated queries into CQL and sends them to the cluster for execution. When the cluster returns the results, the LINQ component translates them back into objects that you can work with in C#.

1. Add a `using` statement to your class:

   ```
   using Cassandra.Data.Linq;
   ```

2. Retrieve an `ISession` instance in the usual way and reuse that session within all the classes in your client application.
3. You can get an `IQueryable` instance of using the `Table` constructor:

   ```
   var users = new Table<User>(session);
   ```

New `Table<T> (IQueryable)` instances can be created each time they are needed, as short-lived instances, as long as you are reusing the same Session instance.

For example:

```
public class User
{
   public Guid UserId { get; set; }
   public string Name { get; set; }
   public string Group { get; set; }
}

// Get a list of users from Cassandra using a Linq query
IEnumerable<User> adminUsers =
      (from user in users where user.Group == "admin" select
 user).Execute();

You can also write your queries using lambda syntax

IEnumerable<User> adminUsers = users
      .Where(u => u.Group == "admin")
      .Execute();
```

The LINQ component creates new instances of your classes using its parameterless constructor.

Configuring mappings

In many scenarios, you need more control over how your class maps to a CQL table. You have two ways of configuring the LINQ:

- decorate your classes with attributes
- define mappings in code using the fluent interface

An example using the fluent interface:

```
MappingConfiguration.Global.Define(
   new Map<User>()
      .TableName("users")
      .PartitionKey(u => u.UserId)
```

```
       .Column(u => u.UserId, cm => cm.WithName("id")));
```

You can also create a class to group all your mapping definitions.

```
public class MyMappings : Mappings
{
   public MyMappings()
   {
       // Define mappings in the constructor of your class
       // that inherits from Mappings
       For<User>()
          .TableName("users")
          .PartitionKey(u => u.UserId)
          .Column(u => u.UserId, cm => cm.WithName("id")));
       For<Comment>()
          .TableName("comments");
   }
}
```

Then, you can assign the mappings class in your configuration.

```
MappingConfiguration.Global.Define<MyMappings>();
```

LINQ API examples

The simple query example is great, but the LINQ component has a lot of other methods for doing Inserts, Updates, Deletes, and even Create table. Including LINQ operations `Where()`, `Select()`, `OrderBy()`, `OrderByDescending()`, `First()`, `Count()`, and `Take()`, it translates into the most efficient CQL query possible, trying to retrieve as less data possible.

For example, the following query only retrieves the username from the cluster to fill in a lazy list of `string` on the client side.

```
IEnumerable<string> userNames = ( from user in users where user.Group ==
 "admin" select user.Name).Execute();
```

Some other examples:

```
// First row or null using a query
User user = (
   from user in users where user.Group == "admin"
   select user.Name).FirstOrDefault().Execute();

// First row or null using lambda syntax
User user = users.Where(u => u.UserId == "john")
      .FirstOrDefault()
      .Execute();

// Use Take() to limit your result sets server side
var userAdmins = (
   from user in users where user.Group == "admin"
   select user.Name).Take(100).Execute();

// Use Select() to project to a new form server side
var userCoordinates = (
   from user in users where user.Group == "admin"
   select new Tuple(user.X, user.Y)).Execute();
```

```
// Delete
users.Where(u => u.UserId == "john")
      .Delete()
      .Execute();

// Delete If (Cassandra 2.1+)
users.Where(u => u.UserId == "john")
      .DeleteIf(u => u.LastAccess == value)
      .Execute();

// Update
users.Where(u => u.UserId == "john")
      .Select(u => new User { LastAccess = TimeUuid.NewId()})
      .Update()
      .Execute();
```

ADO.NET

Implementation of the ADO.NET interfaces and abstract classes common present in the `System.Data` namespace of the .NET Framework: `IDbConnection`, `IDbCommand`, and `IDbDataAdapter`.

Implementation of the ADO.NET interfaces and abstract classes common present in the `System.Data` namespace of the .NET Framework: `IDbConnection`, `IDbCommand`, and `IDbDataAdapter`.

It allows users to interact with a Cassandra cluster using a common .NET data access pattern.

Example

```
var connection  = new CqlConnection(connectionString);
connection.Open();
try
{
   var command = connection.CreateCommand();
   command.CommandText = "UPDATE tbl SET val = 'z' WHERE id = 1";
   command.ExecuteNonQuery();
}
finally
{
   connection.Close();
}
```

Mapping UDTs

You map UDTs to C# types.

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

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

User-defined types are declared at the keyspace level.

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

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

You create a C# class that maps to the UDT using the [data type conversions]:

```
public class Address
{
   public string Street { get; set; }
   public string City { get; set; }
   public int Zip { get; set; }
   public IEnumerable<string> Phones { get; set;}
}
```

You declare the mapping once at the session level:

```
session.UserDefinedTypes.Define(
   UdtMap.For<Address>()
);
```

Once declared the mapping will be available for the lifetime of the application:

```
var results = session.Execute("SELECT id, name, address FROM users where id
 = 756716f7-2e54-4715-9f00-91dcbea6cf50");
var row = results.First();
// You retrieve the field as a value of type Address
var userAddress = row.GetValue<Address>("address");
Console.WriteLine("The user lives on {0} Street", userAddress.Street);
```

CQL column and C# class properties mismatch

For the automatic mapping to work, the table column names and the class properties must match (note that column-to-field matching is **case-insensitive**). For example, in the UDT and the C# class examples aboved were changed:

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

```
public class Address
{
   public string Street { get; set; }
   public string City { get; set; }
   public int ZipCode { get; set; }
   public IEnumerable<string> Phones { get; set;}
}
```

You can also define the properties manually:

```
session.UserDefinedTypes.Define(
   UdtMap.For<Address>()
      .Map(a => a.Street, "street")
      .Map(a => a.City, "city")
      .Map(a => a.Zip, "zip")
      .Map(a => a.Phones, "phones")
);
```

You can still use automatic mapping, but you must add a call to the `Map` method. For example:

```
session.UserDefinedTypes.Define(
   UdtMap.For<Address>()
      .Automap()
      .Map(a => a.Zipcode, "zip_code")
);
```

Nesting User-defined types In CQL

UDTs can be nested relatively arbitrarily. For the C# driver you have to define the mapping to all the user-defined types used.

Based on the previous example, let's change the phones column from `set<text>` to a `set<phone>`, where `phone` contains an alias, a number and a country code.

Phone UDT

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

Address UDT

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

Now we can update the Address class to use the Phone class:

```
public class Address
{
   public string Street { get; set; }
   public string City { get; set; }
   public int ZipCode { get; set; }
   public IEnumerable<Phone> Phones { get; set;}
}
```

You have to define the mapping for both classes

```
session.UserDefinedTypes.Define(
   UdtMap.For<Phone>(),
   UdtMap.For<Address>()
      .Automap()
```

```
        .Map(a => a.ZipCode, "zip_code")
);
```

After that, you can reuse the mapping within your application.

```
var userAddress = row.GetValue<Address>("address");
var mainPhone = userAddress.Phones.First();
Console.WriteLine(“User main phone is {0}”, mainPhone.Alias);
```

Node discovery

Description

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

The driver discovers the nodes that constitute a cluster by querying the contact points used in building the cluster object. After this, the driver keeps track of the nodes in the cluster by registering to Cassandra events from it.

Paging results

The driver automatically pages results by default or you can manually page through the rows in a RowSet.

Automatic paging

You can iterate indefinitely over the `RowSet`, having the rows fetched block by block until the rows available on the client side are exhausted.

```
var ps = session.Prepare("SELECT * from tbl1 WHERE key = ?");
// Set the page size at statement level.
var statement = ps.Bind(key).SetPageSize(1000);
var rs = session.Execute(statement);
foreach (var row in rs)
{
   // The enumerator will yield all the rows from Cassandra.
   // Retrieving them in the back in blocks of 1000.
}
```

Manual paging

If you want to retrieve the next page of results only when you ask for it (for example, in a webpager), use the `PagingState` property in the `RowSet` to execute the following statement.

```
var ps = session.Prepare("SELECT * from tbl1 WHERE key = ?");
// Disable automatic paging.
var statement = ps
   .Bind(key)
   .SetAutoPage(false)
   .SetPageSize(pageSize);
var rs = session.Execute(statement);
// Store the paging state
var pagingState = rs.PagingState;
```

```
// Later in time ...
// Retrieve the following page of results.
var statement2 = ps
   .Bind(key)
   .SetAutoPage(false)
   .SetPagingState(pagingState)
var rs2 = Session.Execute(statement2);
```

Note: The `PagingState` property is not encrypted and can be used to inject values to retrieve other partitions, so be careful not to expose it to the end user.

Automatic paging in both LINQ and Mapper components

Both LINQ and Mapper queries support automatic paging: as you iterate through the mapped results, it fetches the following pages. If you want to manually page, you can use LINQ's `ExecutePaged()` method, Mapper's `FetchPage()`, or their async counterparts.

A LINQ paging example:

```
// Providing page size.
IPage<User> adminUsers = users
   .Where(u => u.Group == "admin")
   .SetPageSize(pageSize)
   .ExecutePaged();


// Providing paging state (following pages).
IPage<User> adminUsers = users
   .Where(u => u.Group == "admin")
   .SetPageSize(pageSize)
   .SetPagingState(pagingState)
   .ExecutePaged();
```

A Mapper paging example:

```
IPage<User> users = mapper.FetchPage<User>(pageSize, pagingState, query,
 parameters);

// Or using query options

IPage<User> authors = mapper.FetchPage<User>(
      Cql.New(query, parameters).WithOptions(opt =>
            opt.SetPageSize(pageSize).SetPagingState(state)));
```

Parameterized queries (positional and named)

You can bind the values of parameters in a `BoundStatement` or `SimpleStatement` either by position or by using named markers.

Positional parameters example

```
var statement = session.Prepare("SELECT * FROM table where a = ? and b
 = ?");
// Bind parameter by marker position
session.Execute(statement.Bind("aValue", "bValue"));
```

Named parameters example

```
var statement = session.Prepare("SELECT * FROM table where a = :a and b
 = :b");
// Bind by name using anonymous types
session.Execute(statement.Bind( new { a = "aValue", b = "bValue" }));
```

You can declare the named markers in your queries and use as parameter names when binding.

Prepared statements

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

The examples show the use in-list for prepared statements.

In-list examples

In-list example 1

```
var statement2 = Session.Prepare(
    "SELECT * FROM DynamicTimeUUIDTable WHERE id = :RowKey AND ColumnName
 IN :names");
List<string> nameslist = new List<string>();
nameslist.Add("colName1");
nameslist.Add("colName2");
RowSet results2 = Session.Execute(statement2.Bind( new { RowKey = 1,
 names=nameslist } ));
System.Console.WriteLine ("Here we are. The count is: ");
System.Console.WriteLine (results2.Count());
```

In-list example 2

```
var statement1 = Session.Prepare(
    "SELECT * FROM DynamicTimeUUIDTable WHERE id = :RowKey AND ColumnName IN
 (:names1, :names2)");
RowSet results1 = Session.Execute(
    statement1.Bind(new { RowKey = 1, names1 = "colName1", names2 =
 "colName2"})
    );
System.Console.WriteLine("Here we are. The count is: ");
System.Console.WriteLine(results1.Count());
```

Routing queries

When using the `TokenAwarePolicy`, the driver uses the `RoutingKey` to determine which nodes is used as coordinator for a given statement.

Prepared statements

When using prepared statements, the driver will determine which of the query parameters compose the partition key based on the prepared statement metadata.

Consider a table users that has a single partition key, id.

```
PreparedStatement prepared = session.Prepare(
      "INSERT INTO users (id, name) VALUES (?, ?)");
```

When binding the parameters, the driver knows which parameter corresponds to the partition key.

```
BoundStatement bound = prepared.Bind(Guid.NewGuid(), "Franz Ferdinand");
session.Execute(bound);
```

As a rule of thumb, use prepared statements, and the driver does all the routing for you.

There is one scenario when using prepared statements where the driver will not be able to determine which parameters form the partition key: when using named parameters and the parameter names do not match the column names.

Routing key generation for the following query with named parameters will work:

```
PreparedStatement prepared = session.Prepare(
      "INSERT INTO users (id, name) VALUES (:id, :name)");
```

But generation for the following query will not work:

```
PreparedStatement prepared = session.Prepare(
      "INSERT INTO users (id, name) VALUES (:id_changed_name, :name)");
```

In these cases, either use the same parameter name as the column name or specify the parameter names that compose the partition key yourself:

```
prepared.SetRoutingNames("id_changed_name");
```

Simple statements

If you want to use a simple statement with routing, you must specify the routing values.

```
var id = Guid.NewGuid();
var query = new SimpleStatement(
      "INSERT INTO users (id, name) VALUES (?, ?)", id, "Franz Ferdinand");
// You must specify the values that compose the partition key
query.SetRoutingValues(id);
session.Execute(query);
```

BATCH statements

If you want to enable routing for BATCH statements, you must specify the routing values.

```
var partitionKey = Guid.NewGuid();
var batch = new Batch();
// ... Add statements to the query
// You must specify the values that compose the partition key
batch.SetRoutingValues(partitionKey);
session.Execute(batch);
```

Tuples

The tuple type holds fixed-length sets of typed positional fields.

The new Cassandra tuple data types are now mapped to CLR `Tuple` generic type.

FAQ

Should I create multiple ISession instances in my client application?

Normally you should use one `ISession` instance per application. You should share that instance between classes within your application.

How can I enable tracing in the driver?

Code example

```
// Specify the minimum trace level you want to see
Cassandra.Diagnostics.CassandraTraceSwitch.Level = TraceLevel.Info;
// Add a standard .NET trace listener
Trace.Listeners.Add(new ConsoleTraceListener());
```

What is the recommended number of queries that BATCH size should contain?

Code example

It depends on the size of the requests and the number of column families affected by the BATCH. Large BATCHes can cause a lot of stress on the coordinator. Consider that Cassandra BATCHes are not suitable for bulk loading, there are dedicated tools for that. BATCHes allow you to group related updates in a single request, so keep the BATCH size small (in the order of tens).

Starting from Cassandra version 2.0.8, the node issues a warning if the batch size is greater than 5K.

What is the best way to retrieve multiple rows that contain large-sized blobs?

You can decrease the number of rows retrieved per page. By using the `SetPageSize()` method on a statement, you instruct the driver to retrieve fewer rows per request (the default is 5000).

API reference

DataStax C# driver for Apache Casandra

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