

Interacting with Cassandra

Most common programming languages have drivers for interacting with Cassandra. When selecting a driver, you should look for libraries that support the CQL binary protocol, which is the latest and most efficient way to communicate with Cassandra.

Note

Note

The CQL binary protocol is a relatively new introduction; older versions of Cassandra used the Thrift protocol as a transport layer. Although Cassandra continues to support Thrift, avoid Thrift-based drivers as they are less performant than the binary protocol.

Here are the CQL binary drivers available for some popular programming languages:

Language	Driver	Available at
Java	DataStax Java Driver	https://github.com/datastax/java-driver
Python	DataStax Python Driver	https://github.com/datastax/python-driver
Ruby	DataStax Ruby Driver	https://github.com/datastax/ruby-driver
C++	DataStax C++ Driver	https://github.com/datastax/cpp-driver
C#	DataStax C# Driver	https://github.com/datastax/csharp-driver
JavaScript (Node.js)	node-cassandra-cql	https://github.com/jorgebay/node-cassandra-cql
PHP	phpbinarycql	https://github.com/rmcfrazier/phpbinarycql

While you are likely to use one of these drivers in your applications, to try out the code examples in this book, you can simply use the `cqlsh` tool, which is a command-line interface for executing CQL queries and viewing the results. To start `cqlsh` on OS X or Linux, simply type `cqlsh` into your command line; you should see something like this:

Copy

```
$ cqlsh
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.0.9 | CQL spec 3.4.0 | Native protocol v4]
Use HELP for help.
cqlsh>
```

On Windows, you can start `cqlsh` just the way you ran `nodetool`:

Copy

```
C:> cd %CASSANDRA_HOME%
C:> bin\cqlsh
```

Once you open it, you should see the same output we just saw.

DataStax Java Driver 4.2 (Earlier version) ([../index.html](#))

Native protocol

The native protocol defines the format of the binary messages exchanged between the driver and Cassandra over TCP. As a driver user, you don't need to know the fine details (although the protocol spec (<https://github.com/datastax/native-protocol/tree/1.x/src/main/resources>) is available if you're curious); the most visible aspect is that some features are only available with specific protocol versions.

Compatibility matrix

Java driver 4 supports protocol versions 3 to 5. By default, the version is negotiated with the first node the driver connects to:

Cassandra version	Negotiated protocol version with driver 4 ¹
2.1.x (DSE 4.7/4.8)	v3
2.2.x	v4
3.x (DSE 5.0/5.1)	v4
4.x ²	v5

(1) for previous driver versions, see the 3.x documentation (https://docs.datastax.com/en/developer/java-driver/3.5/manual/native_protocol/)

(2) at the time of writing, Cassandra 4 is not released yet. Protocol v5 support is still in beta, and must be enabled explicitly (negotiation will yield v4).

Controlling the protocol version

To find out which version you're currently using, use the following:

```
ProtocolVersion currentVersion = session.getContext().getProtocolVersion();
```

The protocol version cannot be changed at runtime. However, you can force a particular version in the configuration ([../configuration/](#)):

```
datastax-java-driver {
  advanced.protocol {
    version = v3
  }
}
```

If you force a version that is too high for the server, you'll get an error:

```
Exception in thread "main" com.datastax.oss.driver.api.core.AllNodesFailedException:
  All 1 node tried for the query failed (showing first 1, use getErrors() for more:
    /127.0.0.1:9042: com.datastax.oss.driver.api.core.UnsupportedProtocolVersionException:
      [/127.0.0.1:9042] Host does not support protocol version V5)
```

DataStax Java Driver 4.15 (Latest version) ([../../../../index.html](https://github.com/datastax/java-driver))

Native protocol

Quick overview

Low-level binary format. Mostly irrelevant for everyday use, only governs whether certain features are available.

- setting the version:
 - automatically negotiated during the connection (improved algorithm in driver 4, no longer an issue in mixed clusters).
 - or force with `advanced.protocol.version` in the configuration.
- reading the version: `session.getContext().getProtocolVersion()`
(<https://docs.datastax.com/en/drivers/java/4.14/com/datastax/oss/driver/api/core/detach/AttachmentPoint.html#getProtocolVersion->
-).

The native protocol defines the format of the binary messages exchanged between the driver and Cassandra over TCP. As a driver user, you don't need to know the fine details (although the protocol spec (<https://github.com/datastax/native-protocol/tree/1.x/src/main/resources>) is available if you're curious); the most visible aspect is that some features are only available with specific protocol versions.

Compatibility matrix

Java driver 4 supports protocol versions 3 to 5. By default, the version is negotiated with the first node the driver connects to:

Cassandra version	Negotiated protocol version with driver 4 ¹
2.1.x	v3
2.2.x	v4
3.x	v4
4.x ²	v5

(1) for previous driver versions, see the 3.x documentation (https://docs.datastax.com/en/developer/java-driver/3.10/manual/native_protocol/)

(2) at the time of writing, Cassandra 4 is not released yet. Protocol v5 support is still in beta, and must be enabled explicitly (negotiation will yield v4).

Since version 4.5.0, the driver can also use DSE protocols when all nodes are running a version of DSE. The table below shows the protocol matrix for these cases:

DSE version	Negotiated protocol version with driver 4
4.7/4.8	v3
5.0	v4
5.1	DSE_V1 ³
6.0/6.7/6.8	DSE_V2 ³

(3) DSE Protocols are chosen before other Cassandra native protocols.

DataStax Node.js Driver 4.6 (Latest version)

(.././../index.html)

Connection pooling

The driver maintains one or more connections opened to each Apache 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 the version of the Apache Cassandra cluster. When using the driver to connect to modern server versions (Apache Cassandra 2.1 and above), the driver uses one connection per host.

Setting the number of connections per host

If needed, you can set the number of connections per host depending on the distance, relative to the driver instance, in the `pooling` configuration:

```
const cassandra = require('cassandra-driver');
const distance = cassandra.types.distance;

const options = {
  contactPoints,
  localDataCenter,
  pooling: {
    coreConnectionsPerHost: {
      [distance.local]: 2,
      [distance.remote]: 1
    }
  }
};

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

Simultaneous requests per connection

The driver limits the amount of concurrent requests per connection to `2048` with modern protocol versions and `128` with older versions of the protocol (v1 and v2).

You can throttle requests by setting the `maxRequestsPerConnection` value in the `poolingOptions`.

When the limit is reached for all connections to a host, the driver will move to the next host according to the query plan. When the query plan is exhausted, the driver will yield a `NoHostAvailableError` containing `BusyConnectionError` instances per each host in the `innerErrors` property.

Get status of the connection pool

You can use `getState()` method to get a point-in-time information of the state of the connections pools to each host.

```
const state = client.getState();
for (let host of state.getConnectedHosts()) {
  console.log('Host %s: open connections = %d; in flight queries = %d',
    host.address, state.getOpenConnections(host), state.getInFlightQueries(host));
}
```

DataStax Java Driver 4.15 (Latest version)

(.././../index.html)

Connection pooling

Quick overview

One connection pool per node. **Many concurrent requests** per connection (don't tune like a JDBC pool).

- `advanced.connection` in the configuration: `max-requests-per-connection` , `pool.local.size` , `pool.remote.size` .
 - **metrics (per node)**: `pool.open-connections` , `pool.in-flight` , `pool.available-streams` , `pool.orphaned-streams` .
 - **heartbeat**: driver-level keepalive, prevents idle connections from being dropped; `advanced.heartbeat` in the configuration.
-

Basics

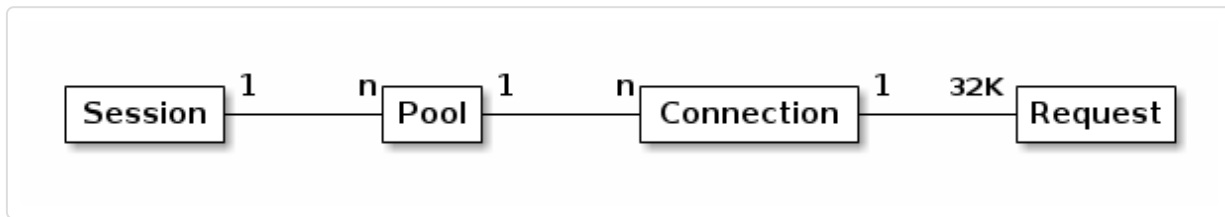
The driver communicates with Cassandra over TCP, using the Cassandra binary protocol. This protocol is asynchronous, which allows each TCP connection to handle multiple simultaneous requests:

- when a query gets executed, a *stream id* gets assigned to it. It is a unique identifier on the current connection;
- the driver writes a request containing the stream id and the query on the connection, and then proceeds without waiting for the response (if you're using the asynchronous API, this is when the driver will send you back a `java.util.concurrent.CompletionStage`). Once the request has been written to the connection, we say that it is *in flight*;
- at some point, Cassandra will send back a response on the connection. This response also contains the stream id, which allows the driver to trigger a callback that will complete the corresponding query (this is the point where your `CompletionStage` will get completed).

You don't need to manage connections yourself. You simply interact with a `CqlSession` (<https://docs.datastax.com/en/drivers/java/4.14/com/datastax/oss/driver/api/core/CqlSession.html>) object, which takes care of it.

For a given session, there is one connection pool per connected node (a node is connected when it is up and not ignored by the load balancing policy (`../load_balancing/`)).

The number of connections per pool is configurable (this will be described in the next section). There are up to 32768 stream ids per connection.



Configuration

Pool sizes are defined in the `connection` section of the configuration (`../configuration/`). Here are the relevant options with their default values:

```
datastax-java-driver.advanced.connection {
  max-requests-per-connection = 1024
  pool {
    local.size = 1
    remote.size = 1
  }
}
```

Do not change those values unless informed by concrete performance measurements; see the Tuning section at the end of this page.

Unlike previous versions of the driver, pools do not resize dynamically. However you can adjust the options at runtime, the driver will detect and apply the changes.

Heartbeat

If connections stay idle for too long, they might be dropped by intermediate network devices (routers, firewalls...). Normally, TCP keepalive should take care of this; but tweaking low-level keepalive settings might be impractical in some environments.

The driver provides application-side keepalive in the form of a connection heartbeat: when a connection does not receive incoming reads for a given amount of time, the driver will simulate activity by writing a dummy request to it. If that request fails, the

connection is trashed and replaced.

This feature is enabled by default. Here are the default values in the configuration:

```
datastax-java-driver.advanced.heartbeat {  
  interval = 30 seconds  
  
  # How long the driver waits for the response to a heartbeat. If this timeout  
  # is considered failed.  
  timeout = 500 milliseconds  
}
```

Both options can be changed at runtime, the new value will be used for new connections created after the change.

Monitoring

The driver exposes node-level metrics (`./metrics/`) to monitor your pools (note that all metrics are disabled by default, you'll need to change your configuration to enable them):


```

datastax-java-driver {
  advanced.metrics.node.enabled = [
    # The number of connections open to this node for regular requests (exposed
    # Gauge<Integer>).
    #
    # This includes the control connection (which uses at most one extra connection
    # node in the cluster).
    pool.open-connections,

    # The number of stream ids available on the connections to this node (exposed
    # Gauge<Integer>).
    #
    # Stream ids are used to multiplex requests on each connection, so this is
    # how many more requests the node could handle concurrently before becoming
    # that this is a driver-side only consideration, there might be other limitations
    # server that prevent reaching that theoretical limit).
    pool.available-streams,

    # The number of requests currently executing on the connections to this node
    # Gauge<Integer>). This includes orphaned streams.
    pool.in-flight,

    # The number of "orphaned" stream ids on the connections to this node (exposed
    # Gauge<Integer>).
    #
    # See the description of the connection.max-orphan-requests option for more
    pool.orphaned-streams,
  ]
}

```

In particular, it's a good idea to keep an eye on those two metrics:

- `pool.open-connections`: if this doesn't match your configured pool size, something is preventing connections from opening (either configuration or network issues, or a server-side limitation – see CASSANDRA-8086 (<https://issues.apache.org/jira/browse/CASSANDRA-8086>));
- `pool.available-streams`: if this is often close to 0, it's a sign that the pool is getting saturated. Consider adding more connections per node.

Tuning

The driver defaults should be good for most scenarios.

Number of requests per connection

In our experience, raising `max-requests-per-connection` above 1024 does not bring any significant improvement: the server is only going to service so many requests at a time anyway, so additional requests are just going to pile up.

Lowering the value is not a good idea either. If your goal is to limit the global throughput of the driver, a throttler (`../throttling`) is a better solution.

Number of connections per node

1 connection per node (`pool.local.size` or `pool.remote.size`) is generally sufficient. However, it might become a bottleneck in very high performance scenarios: all I/O for a connection happens on the same thread, so it's possible for that thread to max out its CPU core. In our benchmarks, this happened with a single-node cluster and a high throughput (approximately 80K requests / second / connection).


It's unlikely that you'll run into this issue: in most real-world deployments, the driver connects to more than one node, so the load will spread across more I/O threads. However if you suspect that you experience the issue, here's what to look out for:


- the driver throughput plateaus but the process does not appear to max out any system resource (in particular, overall CPU usage is well below 100%);
- one of the driver's I/O threads maxes out its CPU core. You can see that with a profiler, or OS-level tools like `pidstat -tu` on Linux. By default, I/O threads are named `<session_name>-io-<n>`.


Try adding more connections per node. Thanks to the driver's hot-reload mechanism, you can do that at runtime and see the effects immediately.


How do I set the maximum pool size in gocql?

Asked 3 months ago Modified 3 months ago Viewed 86 times


2







I wanted to know how can we explicitly set max pool size in `gocql` ?

I mean changing the number of queries which will be run simultaneously.

We are having `200ms` avg response time in Grafana, but we are seeing `24ms` avg read latency in Cassansra exporter panels. Can it be related to pool size?

P.S. I think that's the function in Java driver:
`poolingOptions.setMaxRequestsPerConnection(num)` . What is the equivalent function in gocql?

go cassandra gocql

Share Improve this question
Follow

edited Feb 27 at 5:53



Erick Ramirez

13.3k ● 1 ● 18 ● 23

asked Feb 25 at 17:11




Mojtaba Arezoomand


2,048 ● 8 ● 21


1 Answer


Sorted by:


Highest score (default)


2









I don't think there is an equivalent of the Cassandra Java driver's `setMaxRequestsPerConnection()` in gocql.

The gocql default is 2 connections per host ([NumConns: 2 in cluster.go](#)) and on the server-side, Cassandra will accept up to 128 concurrent requests ([native_transport_max_threads: 128 in cassandra.yaml](#)). Cheers!

Share Improve this answer

Follow

edited Feb 27 at 8:22

Erick Ramirez

13.3k ● 1 ● 18 ● 23

answered Feb 27 at 6:00

NumConns is actually the number connection to db. The requests will be sent through these connections simultaneously. So I think increasing the number of connections would do the job, thanks.

– [Mojtaba Arezoomand](#) Feb 27 at 7:21