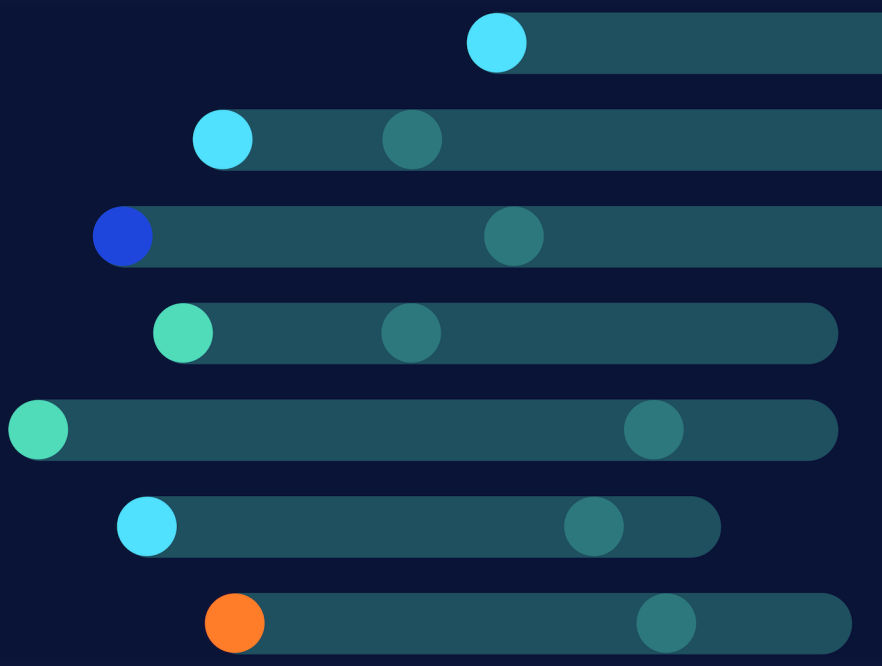
Developing applications with DataStax drivers



DATASTAX Documentation

2021-02-27-08:00

Contents

1. Developing applications with Apache Cassandra™ and DataStax Enterprise

The DataStax drivers are the primary resource for application developers creating solutions using using Cassandra or DataStax Enterprise (DSE).

This guide covers general features and access patterns common across all DataStax drivers, with links to the individual driver documentation for details on using the driver features. If you are developing applications using data stored in Cassandra or DataStax Enterprise, familiarize yourself with these techniques and features to make sure your application is performant and scalable.

Prerequisites

Before building an application, learn about the DSE architecture and review the DataStax Academy tutorials. It's especially important to understand data modeling when developing applications using Cassandra or DSE. This foundation helps ensure your application performs to its fullest potential.

Programming language options

There are DataStax drivers for a range of programming languages. The drivers have features common across all the drivers. Please consult the individual driver documentation pages for detailed information on the driver's features and API.

Table 1. Development status of DataStax drivers

Actively developed drivers	Maintenance mode drivers[1]
C/C++ driver	PHP driver
C# driver | C# DSE Graph Extension	Ruby driver
Java driver (DSE Graph Extension included)	
Node.js driver | Node.js DSE Graph Extension	
Python driver | Python Graph Extension	

DataStax drivers

As of January 2020, DataStax no longer develops separate drivers for Cassandra (OSS drivers) and for DataStax Enterprise (DSE drivers). All of the driver functionality that was

1. Supported by DataStax, but only critical bug fixes will be included in new versions.

formerly in the DSE driver only is now in the single DataStax driver. Going forward, all new driver features will be implemented in the DataStax driver. See the blog post for details.

If you currently use one of the DSE drivers, see the upgrade guide in the individual driver's documentation for details on how to migrate to the unified DataStax driver.

Table 2. Driver upgrade guides

C/C++	C#	Java	Node.js	Python

DataStax driver functionality
• CQL support • Synchronous/Asynchronous API • Address translation • Load balancing policies • Retry policies • Reconnection policies • Connection pooling • Automatic cluster discovery • SSL • Error Handling • Compression • Query builder • Object mapper • DSE Graph Fluent API • DSE Advanced Security, Unified Authentication • DSE Geometric types

DSE workloads

DataStax Enterprise supports several workload types:

- Transactional (Cassandra-only)
- Search
- Analytics

- Graph

The DataStax drivers support most of these workloads natively, simplifying applications that run Transactional, Search, and Graph queries alongside one another.

The DataStax drivers in this guide do not support DSE Analytics queries. The access patterns in analytics use cases are different than the access patterns used by the other workloads. The ODBC and JDBC drivers for DSE Analytics and the DSE Spark Cassandra Connector are typically used with the DSE drivers for analytics applications.

For more about workloads, see Working with multi-workload clusters.

Getting started

Each DataStax driver is hosted on the common distribution channel for the particular language. Visit the DataStax Downloads for direct links to download locations and the DataStax Documentation for installation information.

Table 3. Drivers for each language

C/C++	C#	Java	Node.js	PHP	Python	Ruby
Binaries hosted on datastax.com	NuGet	Maven	NPMJS	Binaries hosted on datastax.com	PyPI	RubyGems

2. Best practices for DataStax drivers

These rules and recommendations improve performance and minimize resource utilization in applications that use DataStax drivers.

Use a single session object per application

Create and reuse a single session for the entire lifetime of an application. Sessions are expensive to create because they initialize and maintain connection pools to every node in a cluster. A single driver session can handle thousands of queries concurrently. Use a single driver session to execute all the queries in an application. Using a single session per cluster allows the drivers to coalesce queries destined for the same node, which can significantly reduce system call overhead.

Note: It is not recommended that a session be created per keyspace. Applications should use fully qualified keyspaces in their query strings or explicitly set the keyspace on statement objects. Applications should use fully qualified keyspaces in their query strings or explicitly set the keyspace on statement objects.

Table 4. API references for session

C/C++	C#	Java	Node.js	PHP	Python	Ruby

Use a single cluster object per physical cluster

Create a single cluster object per physical cluster. Cluster objects are relatively expensive to create because they maintain a control connection to a given cluster. Creating more than one cluster object per physical cluster duplicates these resources unnecessarily.

Important: This rule doesn't apply to the C/C++ and OSS Java 4.X / DSE Java 2.x drivers because each session maintains its own cluster state. However, the single session per application rule still applies.

Note: The Node.js driver combines the concepts of session and cluster into a single Client interface.

Table 5. API references for cluster

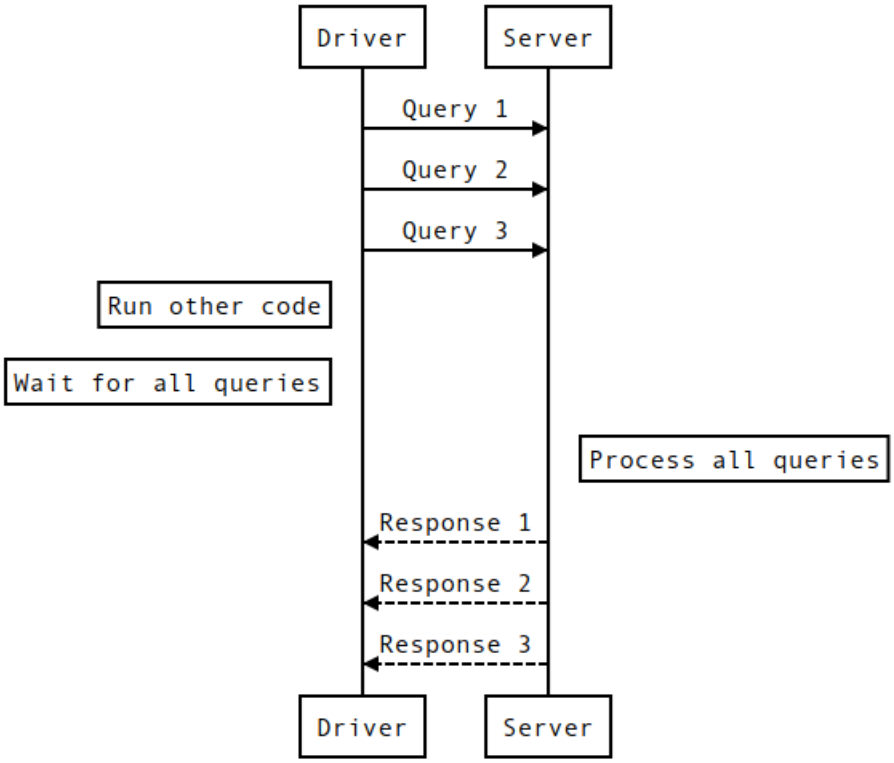
C/C++	C#	Java	Node.js	PHP	Python	Ruby

Run queries asynchronously for higher throughput

Use the driver's asynchronous APIs to achieve maximum throughput. The asynchronous APIs provide execution methods that return immediately without blocking the application's

progress, allowing a single application thread to run many queries concurrently. Asynchronous execution methods return future objects that can be used by the application to obtain query results and errors if they occur. Running many queries concurrently allows applications to optimize their query processing, improves the driver's ability to coalesce query requests, and maximizes use of server-side resources.

Figure 1. Asynchronous queries



Use prepared statements for frequently run queries

Prepare queries that are used more than once. Preparing queries allows the server and driver to reduce the amount of processing and network data required to run a query. For prepared statements, the server parses the query once and it is then cached for the lifetime of an application. The server also avoids sending response metadata after the initial prepare step, which reduces the data sent over the network and the corresponding client side processing.

Table 6. API references for prepared statement

C/C++	C#	Java	Node.js	PHP	Python	Ruby

Explicitly set the local datacenter when using a datacenter-aware load balancing

When using a datacenter-aware load balancing policy, your application should explicitly set the local datacenter instead of allowing the drivers to infer the local datacenter from the contact points. If the driver chose the wrong local datacenter, it increases cross-datacenter traffic, which is often higher latency and monetarily expensive than inter-datacenter traffic. Setting the local datacenter explicitly eliminates the chance that the driver will choose the wrong local datacenter.

When configuring a driver connection, it is easy to include contact points in remote datacenters or invalid datacenters. For example, an application might include contact points for an internal datacenter used during testing. Explicitly setting the local datacenter avoids these types of errors.

Table 7. API references for load balancing

C/C++	C#	Java	Node.js	PHP	Python	Ruby

Avoid delete workloads and writing nulls

In DSE and Cassandra, a tombstone is a marker that indicates that table data is logically deleted. DSE and Cassandra store updates to tables in immutable SSTable files to maintain throughput and avoid reading stale data. Deleted data, time-to-live (TTL) data, and null values will create tombstones, which allows the database to reconcile the logically deleted data with new queries across the cluster. While tombstones are a necessary byproduct of a distributed database, limiting the number of tombstones and avoiding tombstone creation increases database and application performance.

Deletes can often be avoided through data modeling techniques. Nulls can be avoided with proper query construction. For more details on tombstones, see this article on DataStax Academy.

Heavy deletes and nulls use extra disk space and decrease performance on reads. Tombstones can cause warnings and log errors.

For example, in the following schema:

```
CREATE TABLE test_ks.my_table_compound_key (
        primary_key text,
        clustering_key text,
        regular_col text,
        PRIMARY KEY (primary_key, clustering_key)
        )
```

This query results in no tombstones for `regular_col`:

```
INSERT INTO my_table_compound_key (primary_key, clustering_key)
```

```
        VALUES ('pk1', 'ck1');
```

However this query results in a tombstone for `regular_col`:

```
INSERT INTO my_table_compound_key (primary_key, clustering_key,
 regular_col)
        VALUES ('pk1', 'ck1', null);
```

3. Connecting to Apache Cassandra™ and DSE clusters using DataStax drivers

All the DataStax drivers share common features for connecting to Cassandra and DSE clusters.

Connecting to DataStax Astra databases

The DataStax drivers connect to DataStax Astra databases, in addition to traditional DataStax Enterprise (DSE) or Apache Cassandra™ database clusters.

For detailed information about connecting to Astra, see Connecting to Astra databases using DataStax drivers.

Driver configuration

All DataStax drivers have configuration file attributes, builder methods, or constructor parameters for pointing to the secure connect bundle. The database username and password are specified through the normal driver APIs for configuring a plain text authentication provider, or through using one of the convenience methods for username and password.

When using the secure connect bundle, the DataStax drivers automatically establish mutually authenticated TLS connections to the service. You do not have to unzip the secure connect bundle or supply contact points.

After establishing a connection the resulting session is optimally configured for interacting with your Astra database. It is possible (but not required) to use other advanced features, such as Speculative query execution and Execution profiles.

The DataStax driver documentation for your language contains getting started examples and API documentation. Upgrade to the latest minor version of your language's driver to get the new Astra connection API.

Choosing which driver to use

For new applications and for testing, use the DataStax drivers for Apache Cassandra™:

Table 8. DataStax drivers for Apache Cassandra™

C/C++ (Astra API introduced in version 2.14.0)	C# (Astra API introduced in version 3.12.0)	Java version 4.x (Astra API introduced in version 4.3.0) and 3.x (Astra API introduced in version 3.8.0)	Node.js (Astra API introduced in version 4.3.0)	PHP (not supported)	Python (Astra API introduced in version 3.20.0)	Ruby (not sup-

Table 8. DataStax drivers for Apache Cassandra™ (continued)

						ported)

For existing applications that use the deprecated DSE Drivers:

Table 9. DataStax Enterprise drivers

C/C++ (Astra API introduced in version 1.10.0)	C# (Astra API introduced in version 2.9.0)	Java version 2.x (Astra API introduced in version 2.3.0) and 1.x (Astra API introduced in version 1.9.0)	Node.js (Astra API introduced in version 2.3.0)	PHP (not supported)	Python (Astra API introduced in version 2.11.0)	Ruby (not supported)

Note: All the driver functionality that was in the DSE drivers is now in the DataStax drivers for Apache Cassandra™. See the upgrade guides for each driver for information on migrating to the DataStax drivers.

Migrating applications between Apache Cassandra™, DDAC, DSE, and Astra

If you have tooling that currently works with self-managed deployments (Apache Cassandra™, DataStax Distribution of Apache Cassandra™ or DataStax Enterprise clusters), connecting it to Astra is simple:

- Upgrade to the version of the driver that supports the Astra secure connect bundle.
- Set a configuration value to point to the secure connect bundle.

The following table shows the configuration settings required to connect to a self-managed deployment or an Astra database.

Table 10. Configuration changes for self-managed deployments and Astra

Parameter	Self-managed	DataStax Astra
Contact points	**required**	unset
Secure connect bundle	unset	**required**
SSL context	optional, configured manually	automatically configured
Local datacenter	**required**	automatically configured
Database username	optional, configured manually	**required**

Table 10. Configuration changes for self-managed deployments and Astra (continued)

Parameter	Self-managed	DataStax Astra
Database password	optional, configured manually	**required**

Note: Astra enforces application best practices by preventing functionality that degrades the database. For a full list of these limits see the Astra on AWS and Astra on GCP documentation.

Authentication in DataStax drivers

The DataStax drivers support Apache Cassandra™ authentication and DSE Unified Authentication.

To simplify and standardize DataStax Enterprise security features, Unified Authentication was introduced in DSE 5.0 to give operators and developers a single, flexible security model. A single DSE server can accept multiple forms of authentication:

- internal username and password authentication
- LDAP or Active Directory authentication
- Kerberos authentication

Clients with different levels of access can use varying authentication schemes to connect to the same DSE server.

DSE's Unified Authentication provides the ability to assign users to roles and to tie access to database resources based on that role. DSE also allows users to login and execute using proxy roles. All of these features are enabled directly in the DataStax drivers, with built-in classes to enable the desired security configuration.

The DataStax drivers ship with built-in authentication providers that provide the necessary utilities to connect to a secured DSE cluster.

By default, DSE has no authentication service enabled. This makes it easy to get started but is not intended for production deployments. Before configuring the driver to use authentication, enable the desired security schemes within DSE and create users and roles in the database.

Authenticating with internal or LDAP usernames and passwords

The drivers use a plain text authentication provider to perform both DSE's internal and LDAP or Active Directory authentication. The driver will send a plain text username and password to the server, which will authenticate to the underlying configured scheme.

Because these mechanisms transmit credentials in clear text in the native protocol, they should always be used in conjunction with client-server transport encryption.

Table 11. References for plain text authentication

C/C++	C#	Java	Node.js	PHP	Python	Ruby

Authenticating with Kerberos

Each driver extends authentication providers to perform DSE's Kerberos authentication.

Kerberos uses a krb5.conf file for configuration. Credentials can reside in a ticket cache or a keytab.

Important: The C# driver uses the Microsoft security framework called SSPI, which supports Kerberos. When using the C# driver, a Kerberos ticket is obtained during system login and the driver uses that ticket to authenticate. The rest of this section does not apply to the DSE C# driver.

The krb5.conf configuration file

DSE driver authentication against a Kerberos-enabled DSE cluster requires a `krb5.conf` file containing the Kerberos configuration settings. This file may be in the node's `/etc` directory. If it is not, contact your Kerberos system administrator to locate the file.

To reference a `krb5.conf` file in a non-default location, set the `KRB5_CONFIG` environment variable to the location of `krb5.conf`. Kerberos command line tools such as `kinit`, `klist`, and `kdestroy` respect this variable.

All drivers except Java and C# respect this environment variable. Java clients must set the `java.security.krb5.conf` system property to the path to the `krb5.conf` file at startup. The C# driver uses SSPI, which doesn't use `krb5.conf`.

The Kerberos ticket cache

To use the Kerberos ticket cache, use the `kinit` command to authenticate with the Kerberos server and obtain a ticket. Verify the ticket cache contains a ticket for the successful authentication with the `klist` command. Once you verify there is a ticket in the ticket cache, an application that has been configured to use the Kerberos authentication provider is ready to run. If multiple principals have valid tickets in the ticket cache and no principal was specified in the application, the driver will arbitrarily choose one and use that ticket.

Kerberos keytabs

A keytab can be used to authenticate with Kerberos without requiring any additional credentials or a password. Keytab files must have their permissions set properly to restrict access. The permissions should be set to allow the application user to access the keytab.

Proxy login and execution

Proxy login allows users to authenticate using a fixed set of authentication credentials but allow authorization of resources based on another user role. To use proxy login, see the documentation for the individual drivers.

Table 12. References for proxy authentication

C/C++	C#	Java	Node.js	PHP	Python

Like proxy login, proxy execute allows users to authenticate using a fixed set of authentication credentials but execute requests based on another user role. To use proxy execute, see the documentation for the individual drivers.

Table 13. References for proxy execute

C/C++	C#	Java	Node.js	PHP	Python

Using SSL in DataStax drivers

DSE drivers support SSL-encrypted connections between the driver and server.

By default Apache Cassandra™ and DataStax Enterprise (DSE) are configured to communicate with clients using an unencrypted binary protocol. This is convenient for getting started but is usually not suitable for production environments, especially those with clients communicating with Cassandra or DSE over the public internet.

SSL in Cassandra and DSE can be configured different ways depending on the security requirements of the deployment. All configurations result in encrypted communication between the client and server.

SSL allows for different levels of identity verification:

- No identity verification between the client and server.
- The client verifies the identity of the server.
- The server verifies the identity of the client.

A typical SSL workflow consists of the following stages:

1. The client opens a TCP connection to the server on the configured SSL port.
2. An SSL handshake is initialized by the server, sending its public key (or certificate) to the client.

3. The client uses that public key certificate to generate an encrypted session key and sends it back to the server.
4. The server decrypts the message using its private key and retrieves the session key.
5. All communication from that point on is encrypted using that session key.

Table 14. SSL certificate instructions

C/C++	C#	Java	Node.js*	PHP	Python	Ruby

No identity verification

Clients with DataStax drivers communicating securely with Cassandra or DSE servers should validate the server's identity. While most drivers support creating SSL connections to the server without identity verification, it is not recommended for production deployments.

When a secure browser contacts a web server, the browser verifies the identity of the server before sending it requests in case an attacker is masquerading as the web server. Secure communication to a bad actor defeats the purpose of configuring secure communication between the browser and web server in the first place.

Client verifies server

To verify the identity of a server, the driver must be configured with a list of trusted certificate authorities (CAs). When the driver receives the server's SSL certificate during the SSL handshake, it checks that the certificate was signed by one of the registered CAs. If the certificate was not signed by a registered CA, the client checks that the signer was signed by one of the registered CAs. It continues through the signers until it finds one that is in the client's list of trusted CAs.

If the client doesn't find a registered CA, identify verification fails.

Server verifies client

A server is configured to verify the identity of a client by setting the `require_client_auth` to true under `client_encryption_options` in cassandra.yaml. This scenario requires clients be configured with their own certificates to send to the server upon request during the SSL handshake.

Load balancing with DataStax drivers

The DataStax drivers control the distribution of the incoming load across the cluster. The *load balancing policy* determines the node in the cluster to be the coordinator for executing a given query. Determine the load balancing policy during application development because the policy determines the nodes that will have connection pools created and maintained by the driver. For most deployments, use the default load balancing policy.

Table 15. Load balancing policy configuration

C/C++	C#	Java	Node.js	PHP	Python	Ruby

Coordinator selection

Each time a query is executed, the load balancing policy returns a query plan that determines which hosts are eligible to receive the query. The driver uses the first host on the list to execute the request, leaving the successive hosts for retry and speculative execution.

Token awareness

Token awareness is common across all drivers. *Token awareness* uses the primary key information for a given query and parameters to retrieve the replica nodes. By selecting replicas, this policy guarantees that the selected coordinator for the query owns the data that will be written or retrieved, thereby avoiding an extra network connection on the server side.

The key is automatically calculated for prepared statement executions to obtain accurate query routing.

Datacenter awareness

In some use cases, application requests should be limited to a given datacenter to ensure the data is returned to the user as efficiently as possible.

In a global application, users in North America should have their requests directed to a datacenter in North America. Users in Europe should have their requests routed to a datacenter in Europe. To accomplish this, specify a local datacenter in the load balancing policy so that the driver routes this query more efficiently.

If the requests to the local datacenter do not succeed, many of the drivers support using remote datacenter hosts for queries. Though this may appear to be a way to enact datacenter failover, this feature often leads to unexpected latencies and behaviors in the application. For a detailed explanation, see this "Designing Fault Tolerant Applications with DataStax and Apache Cassandra" white paper.

Default load balancing policy

The DataStax drivers integrate the best practices of token awareness and datacenter awareness into the default load balancing policy. Specifically, the default policy will retrieve the replicas for a given token and return a list of hosts containing the replicas in the local datacenter first, followed by the rest of nodes in the specified local datacenter. Using a load distributing algorithm, the default load balancing policy fairly distributes the load across the replica nodes.

Customizing load balancing

If custom routing and load balancing are required in an application, the existing load balancing interface can be extended. Custom load balancing is provided through whitelist and blacklist load balancing policies. Refer to the individual driver documentation for information on whitelist and blacklist load balancing policies. Customizing the load balancing policy is an advanced topic. Study the existing policies before implementing a custom load balancing policy.

Connection pooling

The DataStax drivers maintain a pool of connections to each of the nodes selected by the load balancing policy. By default, the driver instance creates one connection to each of the local datacenter hosts in the default load balancing policy.

Connection pooling is separate from the initial contact points. Initial contact points are supplied to the driver instance. Those contact points are used only to establish the control connection to discover the Cassandra or DSE cluster topology.

Connection pools are accessed asynchronously. Multiple requests can be submitted on a single connection simultaneously. For most workloads, it is recommended to use one long-lived connection from the driver to each DSE server. If the default connection pool settings are not adequate, the number of connections per host and the maximum number of simultaneous requests per connection are configurable. The binary protocol allows up to 32768 concurrent requests per connection.

Table 16. Connection pools configuration

C/C++	C#	Java	Node.js	PHP	Python [2]	Ruby [3]

Retry policies

Retry policies allow the DataStax drivers to automatically retry a request upon encountering specific types of server errors:

- read timeouts
- write timeouts
- unavailable exceptions

2. Does not have ability to customize connections per host, single thread/GIL is limiting factor.
3. Search page for `connections_per_local_node` and `requests_per_connection`.

In these scenarios, a node is designated as a coordinator for the request by the load balancing policy. The coordinator routes the request to the replicas and returns the response to the driver.

Each DataStax driver implements a default retry policy.

Table 17. Retry policies for drivers

C/C++	C#	Java	Node.js	PHP	Python	Ruby

Along with the driver-included retry policies, some drivers allow extended retry policies to implement custom behavior based on read timeout, write timeout, unavailable exceptions, and request errors.

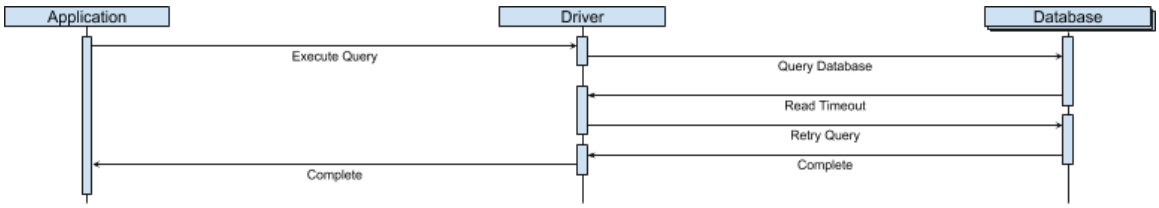
Default retry policy

The default retry policy retries a request when it is safe to do so while preserving the consistency level of the original request. Use this policy for most deployments.

Read timeout

If the number of replicas that reply is greater than or equal to the number of required responses per the consistency level, the default retry policy retries the request. In all other cases, it returns an error.

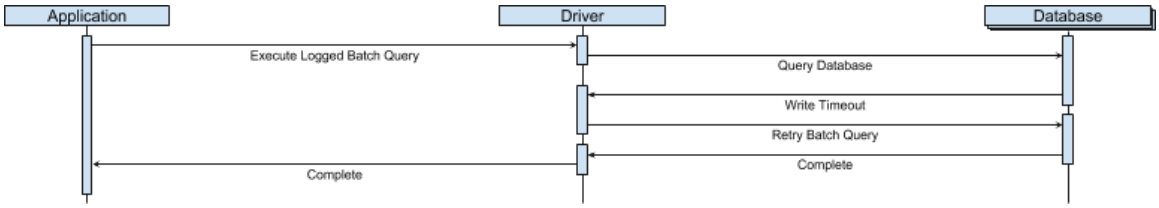
Figure 2. Read timeout



Write timeout

If the request is a logged batch request and fails to write to the batch log, the default retry policy retries the request. In all other cases, it returns an error.

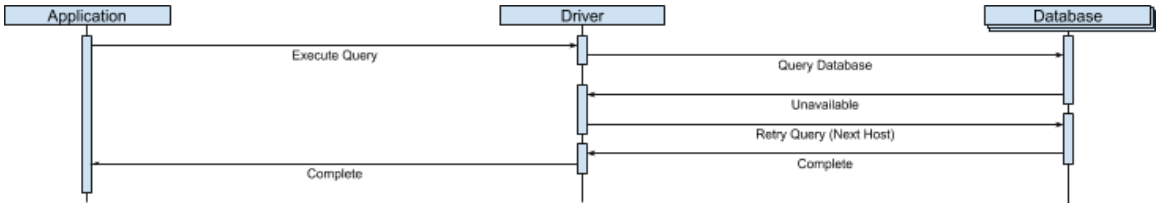
Figure 3. Write timeout



Unavailable errors

If the request encounters an unavailable error, the default retry policy retries the request using the next host in the load balancing policy.

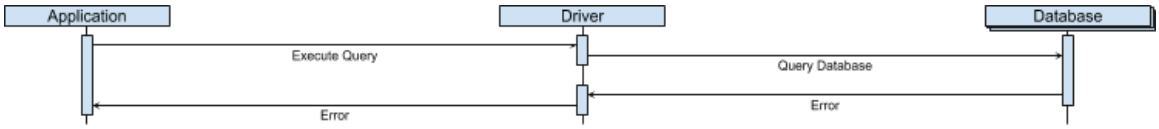
Figure 4. Unavailable errors



Fall-through retry policy

The fall-through retry policy never retries or ignores a failed request. In all cases, the fall-through retry policy returns an error. Use this policy for applications that need to implement their own business logic to handle retrying a request.

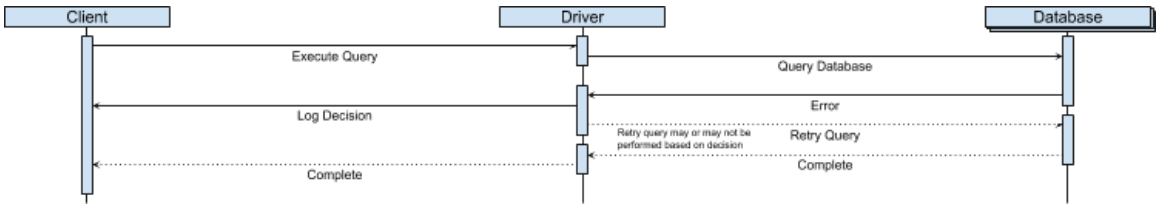
Figure 5. Fall through retry



Logging retry policy

The logging retry policy is intended to be a parent policy for another retry policy implementation and *only* logs the retry decision made by its child policy. This policy is typically used to debug driver retry behavior.

Figure 6. Logging retry policy



Reconnection policies

Reconnection policies allow the DataStax drivers to automatically reestablish a connection to a node that was previously marked as down. A node can be marked down by the

server's gossip process or as a result of an idle connection timeout. Status changes are passed along the control connection back to the driver.

All of the drivers offer a standard reconnection policy. Some drivers offer additional reconnection policies:

- **Constant**: The driver waits a constant amount of time between each reconnection attempt.
- **Exponential**: The driver waits exponentially longer between each reconnection attempt.
- **Fixed**: The driver waits a different amount of time between each reconnection attempt.

Note: Drivers that offer the exponential reconnection policy use that policy as their default. For other drivers, the constant reconnection policy is the default policy.

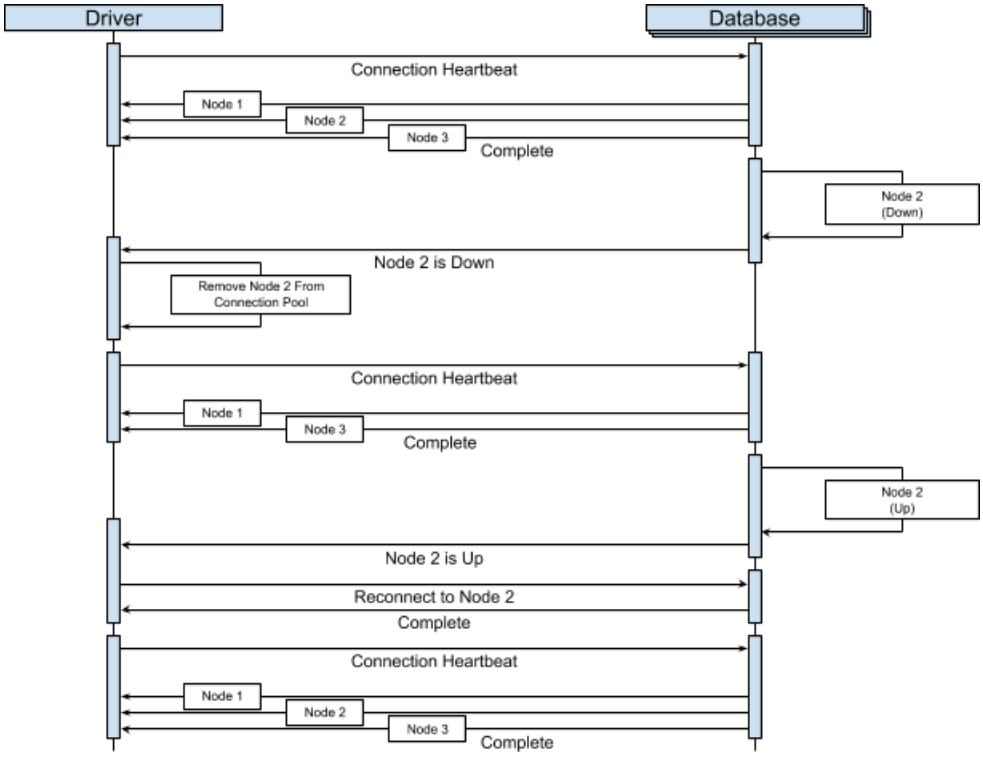
Table 18. Reconnection policies for drivers

C/C++	C#	Java	Node.js	PHP	Python	Ruby

Gossip reconnection

The control connection for the drivers listens to push notifications from the DSE server cluster. When a node is marked *up*, all scheduled reconnections are canceled and a new connection to that node is established.

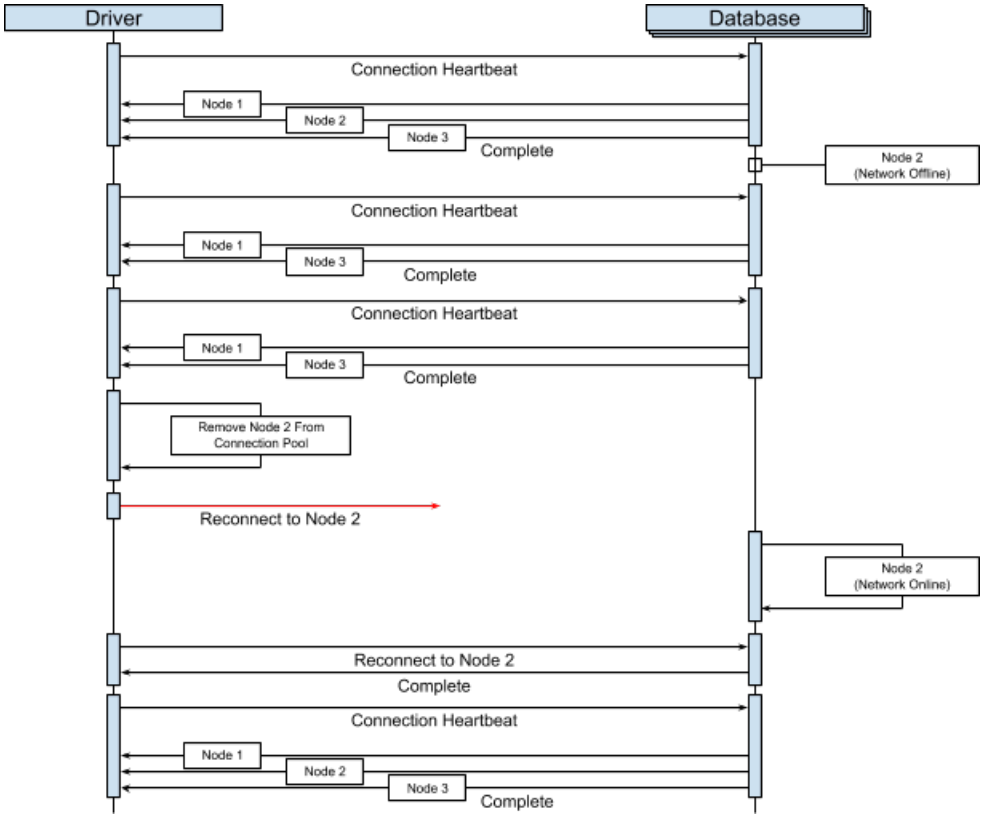
Figure 7. Gossip reconnection policy example



Idle disconnect and reconnection

To prevent intermediate network devices like routers and firewalls from disconnecting the drivers from a node, an OPTIONS request is sent to a connection at a constant interval, also known as a heartbeat. If the connection becomes idle and the node does not respond to the heartbeat in a given amount of time, the node is marked *down*. Once this occurs, the driver waits a specified amount of time based on the reconnection policy before attempting to reconnect to the node.

Figure 8. Idle disconnect and reconnection policy example



Execution profiles

Execution profiles allows a single session to run different types of query workloads, each with its own settings. An execution profile encapsulates a group of settings that can then be associated with individual queries. This provides a convenient way to group queries based on the following settings:

- Request timeout
- Consistency level
- Load balancing policy
- Retry policy
- Speculative execution policy

Note: This is not an exhaustive list and can vary by driver implementation.

Table 19. Execution policies for drivers

C/C++	C#	Java	Node.js	PHP (not supported)	Python	Ruby

Using execution profiles

Create, configure, and register an execution profile object by name using the cluster or session objects, depending on the driver. Associate the named execution profile object with a query by providing the name to the query execution method or setting it on a statement object.

The method of associating the profile name with the query execution can vary by driver implementation. See the language-specific documentation for additional details on using execution profiles.

4. Submitting queries with DataStax drivers

The ability to submit queries and receive results is the core functionality of DataStax drivers.

Working with multi-workload clusters

There are several different ways to query the data stored in DataStax Enterprise (DSE), above and beyond that of Apache Cassandra™. These access patterns are enabled by what are known as *workloads* in DSE. The DataStax drivers can query clusters with different workload types. Below are the different workloads that DSE offers to extend a multi-model experience to developers.

- DSE Core: transactional, typically through standard Cassandra CQL queries
- DSE Search: filtering, typically through Lucene queries
- DSE Analytics: computation, typically through Spark jobs
- DSE Graph: relationships, typically through TinkerPop traversals

When developing an application, each workload type requires different techniques to effectively leverage the use case covered by the workload.

Before creating applications, study and understand the DSE deployment architecture. Developers need to know which datacenters make up a DSE cluster and the supported workloads to direct the different types of queries to the appropriate datacenter.

For example, to use a solr_query in the application, the target datacenter must have DSE Search enabled. To execute graph traversals from the application, the connected datacenter must have DSE Graph enabled. Transactional queries can typically be made against any datacenter, as this core functionality is present for all workloads.

The DataStax drivers expose load balancing policies as a means to steer from the application. The load balancing policy can be supplied in the execution profiles or while configuring the driver cluster and session objects. Use the datacenter-aware policy to restrict queries to a specified datacenter.

Execution profiles

For the drivers that support execution profiles, define separate profiles for the different workloads used in the application. For example, if there are two datacenters, one with DSE Search enabled and one with DSE Core only, use a `SearchExecutionProfile` to direct the DSE Search queries to the DSE Search datacenter. For the `SearchExecutionProfile`, pass the DSE Search datacenter as the local datacenter in the `DCAware` load balancing policy. This profile can then be passed to

all execution methods that use DSE Search indexes in the queries. These queries will be directed to the datacenter that supports DSE Search workloads.

Driver instance per workload

For drivers that do not support execution profiles, use separate driver instances for the different workloads used in the application. This is similar to the execution profile mechanics except that the local datacenter is passed to the load balancing policy when creating the driver instance. For example, create a `SearchSession` with the DSE Search datacenter configured as the local datacenter in the load balancing policy. Use this `SearchSession` in the application for all queries that use DSE Search indexes.

DSE Core and DSE Search

Plain CQL queries and CQL queries that use the `solr_query` syntax for DSE Search are natively supported in the DataStax drivers through the synchronous and asynchronous execute methods.

Table 20. CQL queries for drivers

C/C++	C#	Java	Node.js	PHP	Python	Ruby

DSE Graph

DSE Graph queries are supported in the DataStax drivers through dedicated graph methods. We strongly recommend using the DSE Graph Fluent API (similar to TinkerPop's ByteCode API) to execute graph queries. The String API is also available (similar to TinkerPop's Script API) for the drivers that do not support the Fluent API.

Table 21. DSE Graph queries for drivers

C/C++	C#	Java	Node.js	PHP	Python	Ruby

DSE Analytics

DSE Analytics queries are supported by these drivers:

- Simba ODBC Driver for Apache Spark
- Simba JDBC Driver for Apache Spark
- Spark DSE Connector

Using DSE Search with the DataStax drivers

The DataStax drivers allow developers to perform DSE Search queries in their applications.

DataStax Enterprise Search enables applications to query data. Queries may use these features:

- general
- indexing
- full-text
- faceted (categorization)
- hit prioritization
- spatial and temporl filtering
- social media matchups

The DataStax drivers allow applications to access these features using solr_query syntax in the `WHERE` clause or normal CQL semantics.

Before using DSE Search capabilities in an application, enable the target DSE nodes for DSE Search. Create search indexes for the columns that will be accessed in the queries.

Note: It is a best practice to plan ahead for columns and types that are indexed because adding search indexes has a resource and performance cost.

Load balancing

DSE Search queries must be directed to a datacenter with DSE Search enabled by using the load balancing policy in the driver. Use a datacenter-aware load balancing policy with the DSE Search local datacenter. For more, see Working with multi-workload clusters.

Paging

DSE Search paging is integrated in the DataStax driver execution implementation. The drivers use cursors for deep pagination. Enable paging through the `cql_solr_query_paging` option in dse.yaml on the server, or dynamically in the application in the `solr_query` parameters.

Geospatial data types

Location-based search is a key feature for a personalized user experience. DataStax Enterprise enables this through special geospatial data types:

- `Point`
- `LineString`
- `Polygon`

See Geospatial queries for Point and LineString. There are basic and advanced examples of geospatial data type queries.

Date ranges

Filtering by date and time is a common use case in search queries. DSE Search delivers powerful filtering on single point-in-time or open bound date ranges through the CQL DateRangeType.

Submitting DSE Graph queries with the DataStax drivers

The DataStax drivers expose the String API and Fluent API for executing DSE Graph traversals.

Graph use cases are characterized by highly connected data. Traversing these connections is essential for solving modern fraud detection and personalization use cases. To address the emerging demand of the Graph database, DataStax invests heavily into the Apache TinkerPop graph computing framework that leverages Gremlin as its property graph query language and core API. The DataStax drivers expose several interfaces for executing DSE Graph traversals:

- Fluent API (analogous to TinkerPop Bytecode API)
- String API (analogous to TinkerPop Script API)
- remote traversal sources for full compatibility with TinkerPop's execution model

DataStax recommendes Fluent API as the interface for graph traversals.

Fluent API

The DataStax drivers Graph Fluent API leverages TinkerPop's Gremlin Language Variants and allows developers to programmatically construct Gremlin traversals and execute the compiled bytecode through a DSE session, similar to standard CQL queries. This interface is recommended for all new DSE Graph applications.

Figure 9. DSE Graph Fluent API

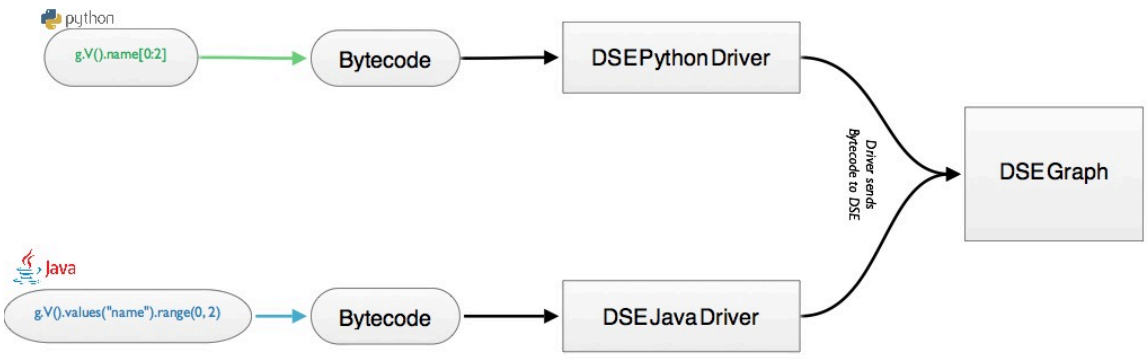


Table 22. Fluent API for drivers

C/C++ (not supported)	C#	Java	Node.js	PHP (not supported)	Python	Ruby (not supported)

String API

The String API is a more limited interface than the Fluent API. The String API simply passes Gremlin Groovy strings through the DataStax Driver to the DSE Graph server.

Table 23. String API for drivers

C/C++	C#	Java	Node.js	PHP	Python	Ruby

Remote traversal source

The DataStax drivers allow a TinkerPop `GraphTraversalSource` to be remotely connected to DSE Graph. This source lends full compatibility with TinkerPop types and uses an implicit execution model through the TinkerPop terminal steps.

Note: The results for a `GraphTraversalSource` are detached from the server. Modifications to the remote elements do not directly affect the data stored in DSE Graph.

Table 24. Remote traversal source for drivers

C/C++ (not supported)	C#	Java	Node.js	PHP (not supported)	Python	Ruby (not supported)

Domain Specific Languages

Domain Specific Languages (DSLs) simplify code and provide concise APIs for DSE Graph applications. DSLs allow the developer to abstract the underlying Gremlin code that is traversing the DSE property graph into usable methods that are tailored to the application.

Table 25. Domain Specific Languages for drivers

C/C++ (not supported)	C#	Java	Node.js* (not supported)	PHP (not supported)	Python	Ruby (not supported)

User-defined IDs

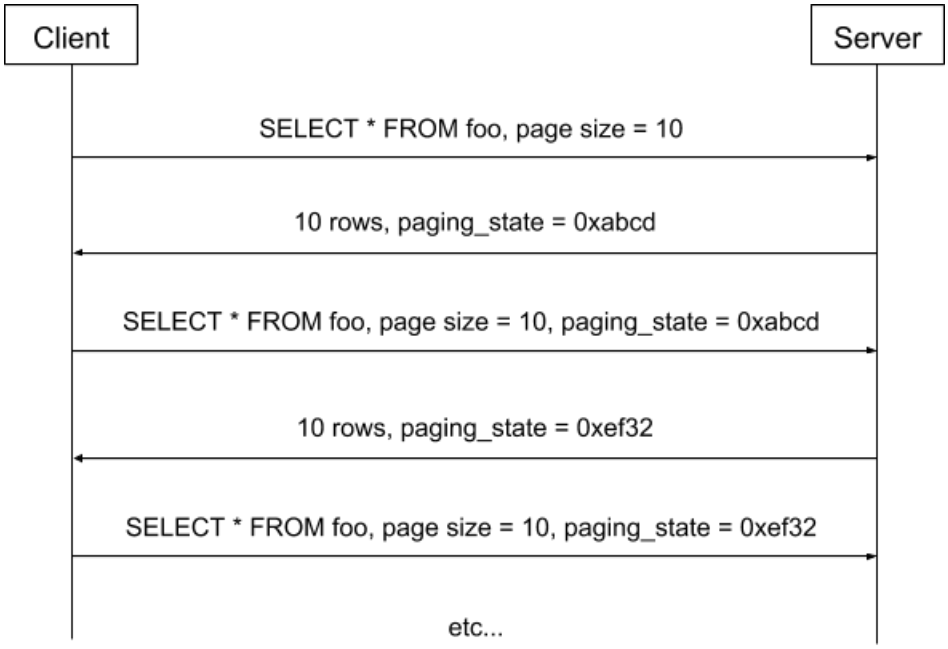
Partition and clustering keys in DSE Core extend to DSE Graph. Use partition and clustering keys when creating vertex labels. Vertex labels more effectively distribute the data throughout the cluster and gives the user control over where the data is distributed.

Result paging with DataStax drivers

Large result sets can be divided into multiple pages that the client will fetch in separate network requests.

The *page size* specifies how many rows will be returned at a single time from the server. With each response, the server returns a *paging state* which is a binary token for the next request to indicate where to restart from.

Figure 10. Result paging example



While the paging API is specific to each driver, they all share a common set of features:

- Paging is enabled by default with options that can be configured.
- The page size can be overridden per query, or paging can be disabled for individual queries.
- Result objects provide a way to fetch the next page directly without manipulating the paging state or re-executing the query explicitly.
- The paging state can be extracted from a given result object and reinjected in a query later. This option is useful if you need to store the state across executions. For example, in a stateless REST web service, the paging state can be encoded in the link to the next page to seamlessly navigate to where the user left off.

Note: The paging state can be forged to access different partitions, so it should not be exposed in plain text in unsafe environments.

Some drivers (C#, Java, Node.js and Python) provide a way to traverse the whole result set transparently by triggering background fetches as the iteration crosses page boundaries.

Table 26. Result paging for drivers

C/C++	C#	Java	Node.js	PHP	Python	Ruby

Synchronous and asynchronous query execution

Queries can be executed against the database synchronously or asynchronously. The correct execution paradigm to use depends on the application.

Synchronous execution

Synchronous query execution is *blocking*, meaning nothing else in the application proceeds until the result from the query is returned. The application blocks for the entire round trip, from when the query is first sent to the database until the results are retrieved and returned to the application.

The advantage of synchronous queries is that it is simple to tell when a query completes, so the execution logic of the application is easy to follow. However, synchronous queries cause poor application throughput.

Table 27. Synchronous query execution for drivers

C/C++	C#	Java	Node.js	PHP	Python	Ruby

Asynchronous execution

Asynchronous query execution is more complex. An asynchronous query execute call does not block for results. Instead, a *future* is immediately returned from the asynchronous execute call. A future is a placeholder object that stands in for the result until the result is returned from the database. Depending on the driver and feature set of the language, this future can facilitate asynchronous processing of results. This typically allows high throughput.

Figure 11. Asynchronous execution example

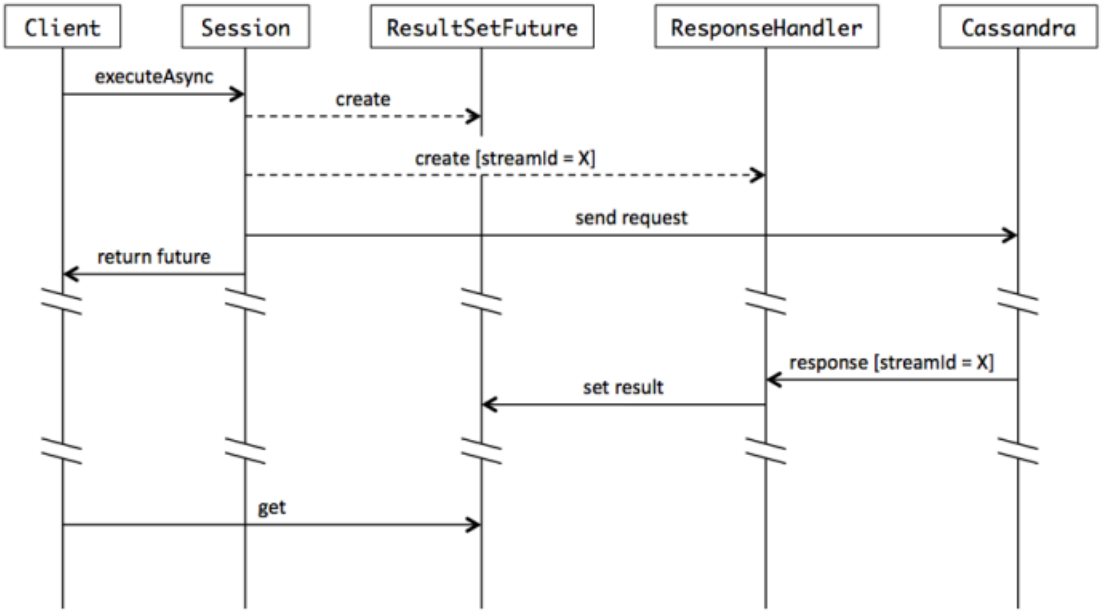


Table 28. Asynchronous query execution for drivers

C/C++	C#	Java	Node.js	PHP	Python	Ruby

Managing concurrency in asynchronous query execution

The DataStax drivers support sending multiple concurrent requests on a single connection to improve overall query performance. This is also known as *request pipelining*. These requests are processed by the server concurrently and responses are sent back to the client driver without strict ordering, allowing improved overall performance when a single operation is slow but the rest of the operations can be processed without any delay. For example, a query that involves consolidating data from multiple partitions will be much slower than a query that only retrieves data from a single partition.

Cassandra or DSE deployments should be planned and provisioned to support the maximum number of parallel requests required for the desired latency of an application. For a given deployment, introducing more load to the system above a minimum threshold will increase overall latency.

On the client side, the driver limits the amount of *in-flights requests* (or simultaneous requests that haven't completed yet) to between 1024 and 2048 per connection by default, depending on the driver language. Above that limit, the driver will immediately throw an exception indicating that the connections to the cluster are busy. You may reach this limit

as a result of handling incoming load to your application. If your application is hitting the limit of in-flight requests add additional capacity to your DSE cluster.

Note: Increasing the limit of in-flight requests can be set using the driver configuration settings, but it's not recommended. A server node usually takes less than a millisecond to fulfil a request. Exceeding the driver side limit would mean trying to support millions of requests per second with a few server nodes.

Limiting simultaneous requests in your application code

When submitting several requests in parallel, the requests are queued at one of three levels: on the driver side, on the network stack, or on the server side. Excessive queueing on any of these levels affects the total time it takes each operation to complete. Adjust the *concurrency level*, or number of simultaneous requests, to reduce the amount of queuing and get high throughput and low latency.

The optimal concurrency level depends on both the client and server hardware specifications as well as other factors like:

- the server cluster size
- the number of instances of the application accessing the database
- the complexity of the queries

When implementing an application, launch a fixed number of asynchronous operations using the concurrency level as the maximum. As each operation completes, add a new one. This ensures your application's asynchronous operations will not exceed the concurrency level.

The following code examples show how to launch asynchronous operations in a loop and controlling the concurrency level.

C/C++	C#	Java	Node.js	Python

Using specialized tools to avoid problems in custom applications

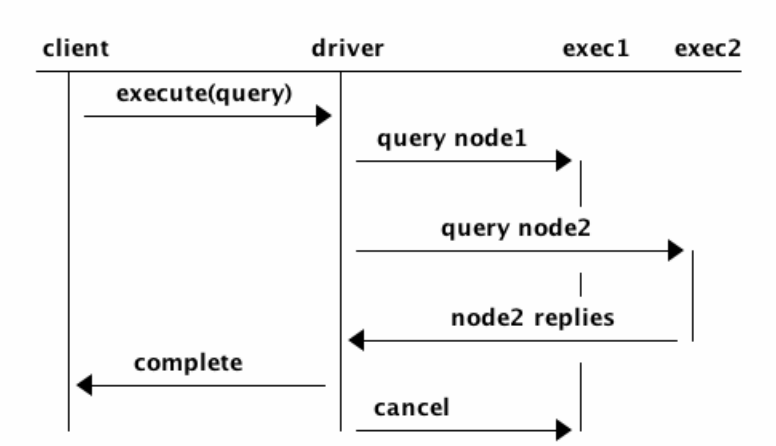
Unbounded concurrency issues often arise when performing bulk operations in custom code. Avoid them by using the appropriate tool for the task. If you are importing data from other sources use dsbulk. If you are performing transformations from external data sources use Apache Spark.

Speculative query execution

Speculative queries are used to preemptively start a second execution of a query against another node, before the first node has replied or returned an error. Sometimes a node may be slow to respond. Queries sent to that node will experience increased latency.

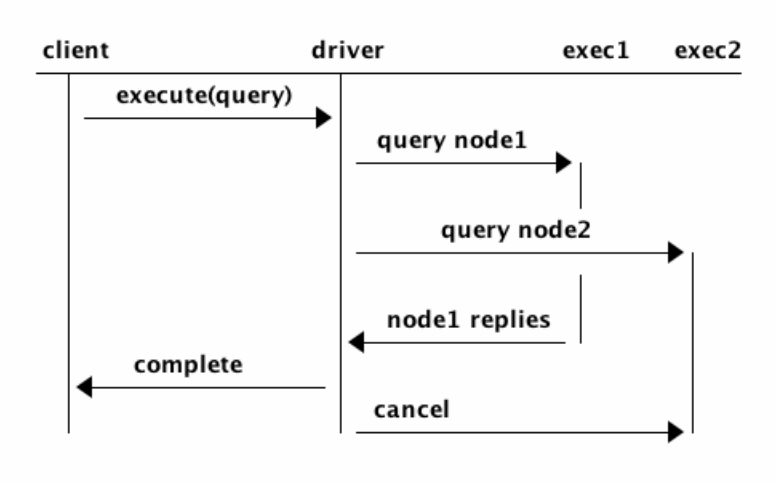
To mitigate this situation, use speculative execution to preemptively start a second execution of the query against another node, before the first node has replied or returned an error. If the second node replies faster, that response is returned to the client and the first execution is cancelled. If the first execution is cancelled, the driver ignores the response, but the request still interacts with the server.

Figure 12. Speculative execution with the second node responding first



The first node might reply just after the second execution call was started. In this case, the second preemptive execution is cancelled. For applications that use speculative execution, the data from whichever node replies faster is returned to the client.

Figure 13. Speculative execution with the first node responding first



Configuration

Speculative execution is disabled by default in the DataStax drivers. See the driver-specific links for how to enable and configure speculative execution.

Table 29. Speculative execution for drivers

C/C++	C#	Java	Node.js	PHP (not supported)	Python	Ruby (not supported)

Tuning and practical details

The goal of speculative execution is to improve the overall latency of an application. However, too many speculative executions increase the load on the cluster. If speculative executions are used to avoid sending queries to unhealthy nodes, a healthy node should rarely reach resource limits. Drivers provide a configurable delay threshold at which speculative executions will be sent. In order to determine an appropriate threshold for your application, benchmark the healthy platform state (all nodes are up and under a normal load), monitoring the response latencies. Based on these results, use the latency at a high percentile (p99.9) as the speculative executions threshold.

Alternatively, when low latency is the highest priority and the cluster can handle the increased throughput, set the threshold to 0, which effectively always performs speculative executions.

Most drivers surface metrics for speculative executions that can be used to observe the speculative executions frequency.

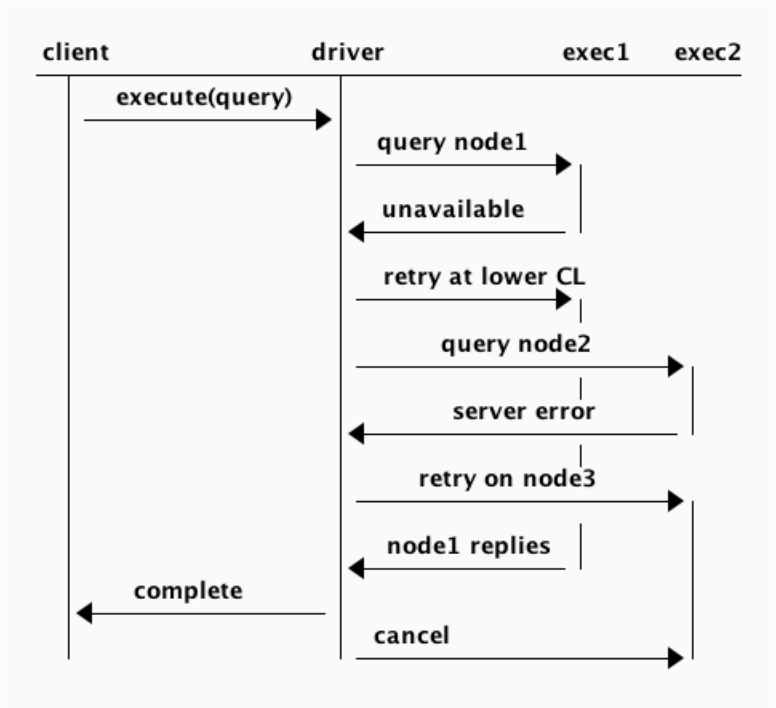
Query idempotence

If a query is not idempotent, the driver will never schedule speculative executions for the query, because there is no way to guarantee that only one coordinator will apply the mutation.

Retries

Turning on speculative executions doesn’t change the driver’s retry behavior. Each parallel execution triggers retries independently. The only impact is that all executions of the same query always share the same query plan, so each node will be used by no more than one execution.

Figure 14. Query plan used in speculative execution



Stream ID exhaustion

One effect of speculative executions is that many requests get cancelled, which can lead to a phenomenon called *stream ID exhaustion*. Each TCP connection can handle multiple simultaneous requests, identified by a unique number called a *stream ID*. See Connection pooling. When a request gets cancelled, the stream ID for that request cannot be used immediately because the response for that request may be returned later from the server once it fulfills the request. If this happens often, the number of available stream IDs diminishes over time. When the available stream IDs goes below a given threshold, the connection is closed and a new connection is created. If requests are often cancelled, connections will be recycled at a high rate.

In practice, exhausting all stream IDs on a connection should not occur. Each connection can reference 32768 stream IDs. Most driver implementations are configured by default to only send 1000 requests per connection.

Most drivers provide a metric for observing the number of inflight requests for a node. Some drivers provide a metric for observing the number of orphaned stream IDs. Monitor these metrics to ensure that stream ID exhaustion is not occurring.

If stream ID exhaustion is occurring, the typical solution is to add more capacity to the cluster, or adjust the system settings of the nodes to avoid resource limits.

Request ordering

Ordering issues are only a problem when using server-side timestamps. All recent versions of the DataStax drivers use client-side timestamps with exception of the C++, PHP, and Ruby drivers. Unless the driver is explicitly configured to use server-side timestamps, this section does not apply. See Query timestamps for details.

For example, the following query is run with speculative execution and server-side timestamps enabled.

```
INSERT INTO my_table (k, v) VALUES (1, 1);
```

When the first execution is slow, a second execution is triggered. Finally, the first execution completes, so the second execution is cancelled. However, cancelling an execution only means that the driver stops waiting for the server's response. The request could still be active.

Suppose that while the second request is still active after the driver canceled the execution, the following query is run and completes successfully.

```
DELETE from my_table where k = 1;
```

The second request of the `INSERT` query finally reaches its target node, which applies the `INSERT`. The row that was successfully deleted is back, despite the driver canceling the second request.

Important: To avoid this scenario, use client-side timestamps.

Query idempotence

A CQL query is *idempotent* if it can be applied multiple times without changing the result of the initial application.

```
UPDATE my_table SET list_col = [1] WHERE pk = 1
```

This query is idempotent because no matter how many times it is executed, `list_col` will always end up with the value `[1]`.

```
UPDATE my_table SET list_col = [1] + list_col WHERE pk = 1
```

This query is not idempotent because if `list_col` was initially empty, it will contain `[1]` after the first execution, `[1, 1]` after the second.

By default, all DataStax Drivers consider queries to be non-idempotent. It is the user's responsibility to mark queries as idempotent to leverage features such as retry and speculative execution.

C/C++	C#	Java	Node.js (see `isIdempotent`)	PHP (not supported)	Python	Ruby

Non-idempotent examples

Queries that insert the result of a non-deterministic functional call (for example `now()` and `uuid()`) are not idempotent.

The following query is not idempotent because `now()` produces a value based on the current time on the DSE coordinator responsible for handling the request, so successive invocations will produce different values if done at different times.

```
UPDATE my_table SET v = now() WHERE pk = 1
```

Counter updates are not idempotent. Each application of a counter update changes the accumulated value of the counter.

```
UPDATE my_table SET counter_value = counter_value + 1 WHERE pk = 1;
```

Prepend, append, or deletion operations on lists are not idempotent.

Prepend and append add elements to the list on each invocation. Delete removes values at a position which may vary depending on the state of the list when the query is invoked. Note that update, insert, and delete operations on set, map, tuples and user defined types are idempotent.

```
UPDATE my_table SET list_col = [1] + list_col WHERE pk = 1
```

Lightweight transactions should be considered non-idempotent if linearizability is a concern. For example:

```
UPDATE my_table SET v = 4 WHERE k = 1 IF v = 1
```

If this statement is executed twice, the `IF` condition will fail on the second execution. In this case, the second execution will do nothing and `v` will still have the value 4. The problem appears when multiple clients execute the query with retries enabled.

1. `v` has the value 1.
2. Client 1 executes the query above, performing a compare and set operation from 1 to 4.
3. Client 1's connection drops, but the query completes successfully. `v` now has the value 4.
4. Client 2 executes a compare and set operation on `v` from 4 to 2.

5. Client 2's transaction succeeds. `v` now has the value 2.

6. Since Client 1 lost its connection, it considers the query as failed, and transparently retries the compare and set operation on `v` from 1 to 4. Because `v` now has a value of 2, it receives a "not applied" response.

One important aspect of lightweight transactions is linearizability. Given a set of concurrent operations on a column from different clients, there must be a way to reorder them to yield a sequential history that is correct. In the above example, from the client's point of view there were two operations.

- Client 1 executed a compare and set operation on `v` from 1 to 4 that was not applied.
- Client 2 executed a compare and set operation on `v` from 4 to 2 that was applied.

Overall the column changed from 1 to 2. There is no ordering of the two operations that can explain the change, and linearizability was broken by the transparent retry at step 6.

Driver metrics

DataStax drivers expose metrics through different libraries and APIs depending on the language.

Whether in the process of developing an application or deploying the solution in production, it is critical to have systems in place that provide insights into the performance of the application. Many modern applications are critical to maintaining a business's value. It is vitally important to effectively monitor and alert operators when systems are degrading. A framework for the application monitoring allows for greater ease when tracing the source of performance issues.

For example, an organization uses DSE OpsCenter to manage a DataStax Enterprise deployment. The operators receive an alert that there is a spike in latency on the server side. If application monitoring is also in place, operators investigating the issue could narrowed the latency to a single DataStax driver instance, and then evaluate how to fix the latency problem.

Table 30. Driver metrics

C/C++	C#	Java	Node.js	PHP (not supported)	Python	Ruby (not supported)

C/C++

The C/C++ driver tracks its metrics through an internal object called CassMetrics. This object contains information about requests (latency and throughput), stats (connections), and errors (timeouts). The DSE C/C++ Driver also exposes information for speculative executions through a CassSpeculativeExecutionMetrics object.

Java

The Java driver delivers its internal measurements through the Dropwizard Metrics library. For all versions of the Java Driver, metrics are exposed through a `MetricRegistry`. The reporter options include JMX, JSON (via a servlet), stdout, CSV files, SLF4J logs, and Graphite. See the Dropwizard Documentation for more details.

C#

The C# driver offers an extension package based on the App.Metrics library. Application developers can use to export metrics to a monitoring tool.

Node.js

The Node.js driver exposes several internal driver metrics in the form of counters in 2 different ways:

- A default implementation which leverages the Node.js events API to expose different counter increments and push it in your existing application metrics toolkit.
- A `ClientMetrics` interface that can be used by metrics libraries, service providers and the community to implement support for existing toolkits like metrics, datadog, prometheus, and measured.

Python

The Python driver uses the scales library for its metrics. Metrics collection is not enabled by default in the Python Driver. To use these metrics, create the `Cluster` object with `metrics_enabled` set to `True`. To view the reported statistics, use a simple HTTP server for spot checking the metrics. A more robust solution for collecting and reporting to Graphite is also supported via a GraphitePusher.

Object mappers in DSE drivers

The C#, Java, Node.js and Python drivers each provide an object mapper. These mappers are tools for generating, executing, and consuming the results of queries.

The object mapper APIs intentionally do not implement all the CQL features. The mapper APIs differ for each language, as each language requires a different set of patterns. Examine the API documentation for each language to make sure it fits with your application.

The following concepts are common across the mapper APIs:

- A *model class* is a class that represents a Cassandra or DSE table. These classes have member variables that map to columns in that table.
- Instances of model classes are *data objects*.

Table 31. Object mapper API documentation for drivers

C/C++ (not supported)	C#	Java	Node.js	PHP (not supported)	Python	Ruby (not supported)

C# object mapper

In the C# object mapper, model classes are normal classes with setters and getters, also known as plain old CLR objects (POCOs). By default, the mapper automatically discovers the object member to DSE column name mapping. The mapping can also be explicitly configured by using the `Define` method.

Because model classes are POCOs, all queries must be mediated by the mapper object. Mappers wrap `Session` instances and provide methods, such as `First` and `Fetch`, that execute statements and define the types into which the mapper reads data. Similarly, mappers provide methods, such as `Insert` and `Update`, that take in a data object and use it to generate a write statement. All of these methods are generic and can take a parameterized query and arguments or a data object.

By default, writing null values deletes or unsets the corresponding value in the database. Writing null values creates tombstones that can impact query performance and the overall health of the database. Writing nulls can be controlled on a per-operation basis by setting insertNulls when using the `Insert` or `InsertAsync` methods.

The C# driver also includes a C# LINQ API available that will not be covered in detail in this guide.

Java object mapper: 3.x version

The Java 3.x mapper relies on annotated classes that are processed at runtime using reflection. Model classes are created by annotating classes with @Table. By default, the mapper automatically discovers the object member to DSE column name mapping. The mapping can also be explicitly configured using the @Column annotation.

To execute queries with the mapper, create a data object that defines the query and pass it using CRUD operations on the mapper. For read queries, the data object arguments are used as filters on the Cassandra or DSE columns. If custom queries are needed, the mapper extends functionality via Accessors. It is also possible to map regular ResultSets to data objects. For write requests, the data object values are used as values in the insert query.

By default, writing null values deletes or unsets the corresponding value in the database. Writing null values creates tombstones that can impact query performance and the overall health of the database. Writing nulls can be controlled by default or on a per-operation basis by setting the saveNullFields option.

Java object mapper: 4.x version

The Java 4.x mapper, like the 3.x mapper, relies on annotations to configure mapped entities and queries. However, there are a few notable differences:

- It uses compile-time annotation processing instead of runtime reflection.
- The “mapper” and “accessor” concepts have been unified into a single “DAO” component, that handles both pre-defined CRUD patterns and user-provided queries.

See the driver manual for more on enabling annotation processor hooks.

Model classes are created by annotating classes with @Entity. The mapper automatically discovers the object member to DSE column name mapping using the default NamingConvention. The default mapping can be changed by specifying a different naming convention with the @NamingStrategy annotation, or by providing your own implementation of NameConverter (not both). In addition, you can override entity and column name mappings individually with the @CqlName annotation.

Once your Entities are designed, you will need to create Data Access Object (DAO) interfaces annotated with @Dao. Your DAOs should define all needed query methods, each marked with the appropriate query annotation for performing your CRUD operations.

Executing queries requires an instance of a DAO, which you can get from the mapper. For each DAO, your mapper interface should define a DaoFactory method. An instance of a mapper can be obtained from the auto-generated Mapper Builder. Simply build an instance of the mapper from the mapper builder and invoke the desired DaoFactory method to get an instance of a DAO. From the DAO, invoke your desired query method to execute the query.

By default, writing null values will **not** delete or unset the corresponding value in the database. Writing null values creates tombstones that can impact query performance and the overall health of the database. You can change this behavior by specifying a different NullSavingStrategy in the @DefaultNullSavingStrategy annotation at the DAO level, specific query methods, or both if you want some query methods of a DAO to have a different strategy than the rest.

Node.js object mapper

For details on the Node.js object mapper, see this DataStax Academy blog post.

Python object mapper (cqlengine)

For the Python object mapper, model classes are created by sub-classing cassandra.cqlengine.model.Model. By default, the mapper automatically discovers the mapping of object attributes created from `cassandra.cqlengine.columns.Column` to DSE column names. This mapping can also be explicitly defined using the db_field kwarg to Column subclass initializers.

It is safest to create the tables for model objects outside the scope of the mapper, though the Python driver does allow for making schema changes with the object mapper. Creating tables within the mapper can result in concurrent schema modifications, which are not recommended.

The Python mapper provides class methods for reading and writing data objects. In addition, queries can be executed by directly calling methods on the model class. The

mapper read query methods return collections of data objects that have instance methods for CRUD operations.

Passing None corresponds to a DELETE operation on the value in the corresponding row. This creates tombstones that can impact query performance and the overall health of the database.

Mapper connections are maintained in a connection registry that can be used to access the sessions that connect to the database.

Query timestamps

Timestamps determine the order of precedence for operations on the same column value from different queries. In Apache Cassandra™ and DataStax Enterprise (DSE), each mutation—update, insert, delete—is assigned a microsecond-precision timestamp to order operations relative to each other. The order of precedence for operations on the same column value is:

1. Data with the latest timestamp.
2. If the operations have the same timestamp, deletes have priority over inserts and updates.
3. Otherwise, the lexically larger value of data has priority. For example, 2 is chosen over 1.

Timestamps can be assigned by the driver client or the server-side node coordinating the request. All recent versions of the DataStax drivers use client-generated timestamps by default for Cassandra versions 2.1 and later and DSE versions 4.7 and later. Older versions of Cassandra and DSE do not support client timestamps, as they were introduced in the CQL native protocol version 3.

Client-side timestamp generation is the default to keep order of operations predictable from the perspective of a single client. Through monotonically increasing client-side timestamps, the driver ensures that all operations are written in the sequential order that they were executed within the scope of that instance.

Without client timestamps, the client is at the whim of timestamps assigned by coordinating nodes. Coordinating nodes assign timestamps based on their internal system clock. It is difficult to keep the different nodes system clock synchronized in a distributed system. Each node is subject to clock drifts ranging from tens of milliseconds to seconds, even when the nodes use NTP or other clock synchronization software.

For example, consider the following scenario where server timestamps are used.

1. A client executes the following query:

   ```
   DELETE FROM tbl_a WHERE key = 0
   ```

   The query is sent to Node A, which creates a delete mutation with timestamp 10.

2. The client then executes:

   ```
   UPDATE tbl_a SET x = ‘hello’ where key = 0
   ```

   The query is sent to Node B, which creates an update mutation with timestamp 9.

3. The client executes:

   ```
   SELECT x from tbl_a where key = 0
   ```

   and receives a result set with 0 rows.

It should be surprising that no rows were returned from the `SELECT` query in step 3. Even though the `DELETE` operation in step 1 was executed before the `UPDATE` operation in step 2, it takes precedence because the largest timestamp (10) was assigned to it. This scenario is avoided completely by using client timestamps.

Configuring timestamp generation in the drivers

Client timestamp generation can be configured or disabled in each of the drivers. See the individual driver documentation for more information on each driver:

Table 32. Client timestamp generation for drivers

C/C++	C#	Java	Node.js	PHP	Python	Ruby

When to use server timestamps

One possible downside to using client timestamps is that the number of client application servers often outnumber DataStax Enterprise nodes in production environments. It’s not unusual for different applications using the same DSE cluster to be managed by different teams. In these cases, it may be operationally challenging to keep the clocks synchronized between many different client application servers.

Out-of-sync client application server clocks is an issue only when there are clients making updates to the same partition values as other clients within a window that would be smaller than the expected clock drift between client nodes. Even in this case, it may not be important that updates made in this window be properly ordered in the sequence in which they were executed. It is possible that these updates were made by different parties who are not aware of one another. If it is important, consider using lightweight transactions.

Lightweight transactions and client timestamps

When executing lightweight transactions (LWTs), any client timestamp assigned to those operations is discarded. This is because DSE maintains a separate timestamp generator that ensures the timestamp assigned is monotonically increased across all LWTs.

One common mistake users make is mixing the use of LWTs and other mutation operations on a single table. This is not recommended, especially since the timestamp mechanism used for normal operations is different than the one used by LWTs, even when using server timestamps.

Keeping clocks in sync across servers

No matter the timestamp strategy, DataStax strongly recommends using a service like NTP to keep the system clocks synchronized across all machines in the data ecosystem. DataStax also recommends organizations measure and understand the degree of clock drift among all the servers in their production environment to understand the time windows that may exist between nodes. Use utilities and commands, such as `clockdiff`, `ntpdate -q`, and `ntp -q`, to measure clock differences between servers.

5. Error handling

When using the drivers with a DataStax Enterprise or Cassandra cluster, various errors and exceptions may be encountered. The correct way to handle these error conditions often depends on the requirements of the application utilizing the driver. In this section, types of errors are covered as well as causes and common remediation of each.

At the broadest level, there are two types of errors:

- Server-originated errors: Server errors are returned directly from the coordinator to the driver and are identical across all of the drivers.
- Client-side errors: Client-side errors are specific to issues that occur in the driver itself and vary from driver to driver.

For specifics, refer to the individual driver error documentation.

Table 33. Driver error documentations

C/C++	C#	Java	Node.js	PHP	Python	Ruby

Server errors

Server errors originate at the server and are sent back to the client. Additional information about all of these errors is available in the Apache Cassandra native_protocol document, section 9. Error Codes.

Authentication errors

Description

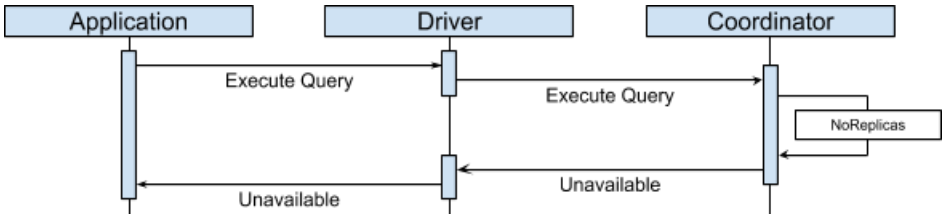
Authentication was required by the server and failed. The possible reason for failing depends on the authenticator in use and the error message may or may not contain more detail about the failure.

Remediation

Investigate the authentication mechanisms used by Cassandra or DSE and the application. Double check username and passwords. See Authentication in DataStax driversand review https://docs.datastax.com/en/security/6.8/security/secDSEUnifiedAuthAbout.html for more information.

Unavailable exceptions

Figure 15. Unavailable exception



Description

This error indicates that the consistency level of the query is higher than the number of available replicas to serve that query. This exception contains 3 parts.

- `CL`: The consistency level of the query that triggered the exception.
- `Required`: An integer representing the number of nodes that must be alive to honor the CL.
- `Alive`: An integer representing the number of replicas that were known to be alive when the request had been processed.

Remediation

Ensure that a sufficient number of replicas are available for your consistency level. This error often signals that nodes are down or lacking connectivity to the coordinator. Another possible cause is that the Cassandra or DSE cluster is in the middle of a rolling restart or upgrade. When operators are performing a rolling upgrade or restart, ensure that the previous node is fully up and ready to receive query requests before restarting the next node in the procedure.

Overloaded exceptions

Description

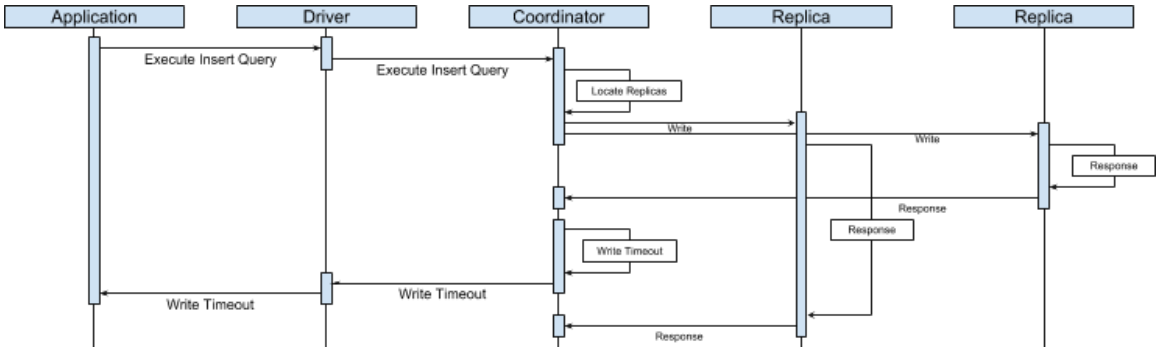
The request cannot be processed because the coordinator node is overloaded by requests.

Remediation

Overloaded exceptions signal that the cluster can not handle the incoming traffic from clients. This can be triggered during spikes in traffic or due to expensive queries exhausting the node's resources. This typically indicates an under provisioned cluster.

Write timeouts

Figure 16. Write timeout exceptions



Description

Write timeouts signal that a server side timeout exception occurred during the write request. This error contains 4 parts.

- `CL`: The consistency level of the query that triggered the write timeout.
- `received`: The number of nodes that acknowledged the request.
- `blockfor`: An integer that represents the number of replicas required to satisfy the consistency level.
- `writetype`: A string that describes the type of write that timed out. Below are the different types of writes.
  - `SIMPLE`
  - `BATCH`
  - `BATCH_LOG`
  - `UNLOGGED_BATCH`
  - `COUNTER`
  - `CAS`
  - `VIEW (MV)`
  - `CDC`

Remediation

When this happens on a non-idempotent write, such as incrementing a counter, caution must be exercised by the client, as the data may or may not have been written to the table

by the node. With an idempotent write, the write can simply be retried. Only batchlog writes are retried by the driver's default retry policy. A query's idempotence can be defined in the application. See the individual driver documentation for the API specifics.

Depending on the SLAs and application requirements, the default server side write timeout may not be adequate and this value can be adjusted in the cassandra.yaml. One common case when write timeouts surface is when batches are large or span multiple partitions. To address this, consider decreasing the batch size and limiting batch writes to a single partition. See this blog post for more details on correctly handling this error.

Read timeouts

Description

Read timeouts signal that a server side timeout exception occurred during the read request. This error contains 4 parts.

- `CL`: The consistency level of the query that triggered the read timeout.
- `received`: The number of nodes that acknowledged the request.
- `blockfor`: An integer that represents the number of replicas required to satisfy the consistency level.
- `data present`: If this value is 0 it means the replica that was asked for the data did not respond. Otherwise the value is not 0. The coordinator will only ask a single node for the data and uses a checksum from the other nodes to determine if the data is consistent.

Remediation

Read timeouts can occur for a variety of reasons. Some possible causes are if the query is requesting a very large amount of data at all once or if there are long server side garbage collection events occurring. This typically indicates issues with the data model or query patterns that are causing poor performance on the server. To debug, first verify in the server logs that garbage collection times are acceptable and then examine the data model and access patterns. The server side read timeout can be altered in cassandra.yaml if no other underlying cause of the timeouts can be diagnosed.

Read failures

Description

A read failure is a non-timeout exception encountered during a read request. This error contains 5 parts.

- `CL`: The consistency level of the query that triggered the error.

- `received`: The number of nodes that acknowledged the request.
- `blockfor`: An integer that represents the number of replicas required to satisfy the consistency level.
- `reasonmap`: A map of endpoint to failure reason codes. This maps the endpoints of the replica nodes that failed executing the request to the code representing the reason for the failure.
- `data present`: If this value is 0 it means the replica that was asked for the data did not respond. Otherwise the value is not 0. The coordinator will only ask a single node for the data and uses a checksum from the other nodes to determine if the data is consistent.

Remediation

This error is rarely encountered. Investigate the reason map to find to the root cause. The most common cause for this type of error is when too many tombstones are read during the request.

Write failures

Description

A write failure is a non-timeout exception encountered during a write request. This error contains 5 parts.

- `CL`: The consistency level of the query that triggered the error.
- `received`: The number of nodes that acknowledged the request.
- `blockfor`: An integer that represents the number of replicas required to satisfy the consistency level.
- `reasonmap`: A map of endpoint to failure reason codes. This maps the endpoints of the replica nodes that failed executing the request to the code representing the reason for the failure.
- `writeType`: A string that describes the type of write that failed. The value of the string will describe the type of write that failed. Below are the different types of writes.
  - `SIMPLE`
  - `BATCH`
  - `BATCH_LOG`
  - `UNLOGGED_BATCH`
  - `COUNTER`

- `CAS`
- `VIEW (MV)`
- `CDC`

Remediation

This error is rarely encountered. Examine the reason map to find to the root cause. The most common cause for this type of error is when batch sizes are too large.

Function failures

Description

A user defined function (UDF) failed during execution. The error message contains the following information.

- `keyspace`: The keyspace of the failed function.
- `function`: The name of the failed function.
- `arg_types`: A list of argument types of the failed function.

Remediation

It is likely that something is logically wrong with the user defined function, such as an infinite loop or syntax error. Scrutinize the UDF definition to find the issue.

Syntax errors

Description

The submitted query contains invalid syntax.

Remediation

Ensure the CQL has correct syntax.

Invalid errors

Description

The submitted query is syntactically correct, but is not a valid query.

Remediation

Ensure the query is valid. Examples of syntactically correct but invalid queries include trying to set the keyspace to a nonexistent keyspace or querying a table that does not exist.

Already exists errors

Description

The query attempted to create a keyspace or table that already exists. This error contains 2 parts.

- `ks`: The keyspace associated with the keyspace or table that already exists.
- `table`: The name of the table that already exists. If no table is involved this is empty.

Remediation

Make sure the keyspace or table does not exist before trying to create it or use the `IF NOT EXISTS` CQL syntax.

Unprepared errors

Description

The execution of a prepared statement was attempted when the statement was not prepared in advance.

Remediation

Prepare the statement before executing it.

Client errors

Client errors originate from problems with the driver itself. These vary from driver to driver depending on implementation, execution model, and ecosystem. All drivers however have the concept of a driver timeout.

Driver timeout

Description

This error name varies from driver to driver but indicates that there was a timeout on the client side. This means that the client side timeout was hit before any response was received from the server side coordinator.

Remediation

The client side timeout is commonly configured as either part of the execution profile or on the query execution method. The value for the client side timeout should be set higher than that of the server side write and read timeouts. This error is typically encountered when a read query is requesting a large amount of data or when batch write sizes are large or span multiple partitions.