DataStax Bulk Loader 1.4.1 Latest version

Updated: 2020-02-15Z
Contents

Chapter 1. About DataStax Bulk Loader ........................................................................... 5

Chapter 2. DataStax Bulk Loader release notes ............................................................... 6
  Bulk Loader 1.4.1 release notes .................................................................................. 6
  Bulk Loader 1.4.0 release notes .................................................................................. 7
  Bulk Loader 1.3.4 release notes .................................................................................. 7
  Bulk Loader 1.3.3 release notes .................................................................................. 9
  Bulk Loader 1.3.2 release notes ................................................................................. 10
  Bulk Loader 1.2.0 release notes ................................................................................. 13
  Bulk Loader 1.1.0 release notes ................................................................................. 14
  Bulk Loader 1.0.2 release notes ............................................................................... 15
  Bulk Loader 1.0.1 release notes ............................................................................... 15

Chapter 3. Installing DataStax Bulk Loader ................................................................. 17

Chapter 4. Architecture .............................................................................................. 19

Chapter 5. Getting Started ......................................................................................... 21
  Loading data ............................................................................................................. 21
  Unloading data ......................................................................................................... 22
  Creating a configuration file for dsbulk .................................................................... 23
  Using SSL with dsbulk ............................................................................................. 24
  Printing cluster information .................................................................................... 25
  Counting data in tables ............................................................................................ 25

Chapter 6. Kerberos client authentication .................................................................... 27
  Kerberos configuration file ....................................................................................... 27
  Keytab file ................................................................................................................ 27
  Ticket Cache ............................................................................................................. 28

Chapter 7. DataStax Bulk Loader reference ............................................................... 31
  dsbulk ..................................................................................................................... 31
    Loading data examples ......................................................................................... 33
    Unloading data examples ....................................................................................... 36
    Exit codes ............................................................................................................. 38
  Common options ...................................................................................................... 38
  Connector options ................................................................................................... 41
  Count options ......................................................................................................... 46
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schema options</td>
<td>47</td>
</tr>
<tr>
<td>Batch options</td>
<td>50</td>
</tr>
<tr>
<td>Codec options</td>
<td>51</td>
</tr>
<tr>
<td>Driver options</td>
<td>55</td>
</tr>
<tr>
<td>Engine options</td>
<td>63</td>
</tr>
<tr>
<td>Executor options</td>
<td>64</td>
</tr>
<tr>
<td>Logging options</td>
<td>65</td>
</tr>
<tr>
<td>Monitoring options</td>
<td>66</td>
</tr>
</tbody>
</table>
Chapter 1. About DataStax Bulk Loader

Use DataStax Bulk Loader and its dsbulk commands to load and unload CSV or JSON data in and out of supported databases.

About this document
Welcome to the DataStax Bulk Loader documentation. To ensure that you get the best experience in using this document, take a moment to read Tips for using DataStax documentation.

About DataStax Bulk Loader
DataStax Bulk Loader is supported on Linux, macOS, and Windows platforms.
DataStax Bulk Loader efficiently and reliably loads small or large amounts of data, supporting developer and production environments. Using dsbulk commands, CSV or JSON files can be rapidly loaded or unloaded to or from the following supported databases:

- DataStax Enterprise (DSE) 4.7 and later databases
- Open source Apache Cassandra® 2.1 and later databases
- DataStax Distribution of Apache Cassandra (DDAC) databases
- DataStax Apollo cloud databases

Features in DataStax Bulk Loader
- CSV and JSON are supported formats, and optionally you can load or unload the data from/to compressed files.
- Files, directories, stdin/stdout, and web URLs can be used for either source or destination.
- Performance improvements of 2-3 times faster compared to cqlsh COPY, due to multi-threaded operation.
- Connect to a cloud-based DataStax Apollo database by including the path to the secure connect bundle, which participants in the Open Beta program can download from the DataStax Cloud console after creating an Apollo database.
- DataStax Java driver options are available directly with dsbulk commands via the datastax-java-driver prefix.
- Secure authentication via Kerberos or username/password over SSL options.
- Configurable data parsing. For instance, date formatting is configurable.
- Performance and progress reporting.
- Command line tool for both Linux and Windows:
  # Can use configuration files to simplify command line calls to dsbulk
  # Tunable parameters to optimize loading and unloading times.
  # Enhancements allow secure connections for loading and unloading data.
- Print basic information about the associated cluster when you request verbose logging on the dsbulk command. Refer to Printing cluster information.
- Diagnose issues encountered during write operations. Refer to Detection of write failures.

Also see the DataStax blog post Introducing the DataStax Bulk Loader.
DataStax Bulk Loader release notes

Release notes for DataStax Bulk Loader.

DataStax Bulk Loader can load and unload data in CSV or JSON format (or via CSV/JSON compressed files) in or out of:

- DataStax Enterprise (DSE) 4.7 and later databases
- Open source Apache Cassandra® 2.1 and later databases
- DataStax Distribution of Apache Cassandra (DDAC) databases
- DataStax Apollo cloud databases

DataStax Bulk Loader is supported on Linux, macOS, and Windows platforms.

For related information, refer to the DataStax Enterprise release notes:

- DSE 6.7 release notes
- DSE 6.0 release notes
- DSE 5.1 release notes

DataStax Bulk Loader 1.4.1 release notes

16 December 2019

DataStax Bulk Loader 1.4.1 release notes

- 1.4.1 Changes and enhancements
- 1.4.1 Resolved issue

Changes and enhancements

This release, DataStax Bulk Loader 1.4.1, adds support for using the `dsbulk load` command to write CSV/JSON data to open source Apache Cassandra™ 2.1 and later database tables. Previously, you could only use `dsbulk unload` and `dsbulk count` commands with Apache Cassandra. (DAT-519)

The new support in 1.4.1 is in addition to the existing functionality to use all `dsbulk` commands with:

- DataStax Enterprise (DSE) 4.7 and later databases
- DataStax Distribution of Apache Cassandra (DDAC) databases
- DataStax Apollo cloud databases

For details about using the commands, refer to:

- Loading data examples
- Unloading data examples
- Count options

For specific requirements with DataStax Apollo cloud databases, see Loading and unloading data with DataStax Bulk Loader.
1.4.1 Resolved issue

When exporting data, the \u0000 null character is now enclosed in quotes, so that the exported data can be loaded subsequently with the same DataStax Bulk Loader settings (DAT-516). By default, the null character is used as the comment character.

DataStax Bulk Loader 1.4.0 release notes

12 November 2019
DataStax Bulk Loader 1.4.0 release notes

Changes and enhancements

DataStax Bulk Loader 1.4.0 has been upgraded to use the latest 2.x version of the DataStax Java driver. (DAT-303)

Before upgrading to DataStax Bulk Loader 1.4.0, note that as a result of the driver enhancements, this release supports DSE 4.7 and later, and Apache Cassandra™ 2.1 and later. Prior releases of DSE and Apache Cassandra are not supported. If you are using earlier releases of DSE or Apache Cassandra, you must remain on DataStax Bulk Loader 1.3.4.

- Many new driver options are available directly with dsbulk commands via the datastax-java-driver prefix.
  
  A number of previously available options have been deprecated, as indicated in the DataStax Bulk Loader reference topics. Those prior options are still supported, but may not be supported in a subsequent release. When you can, review and adjust your command scripts and configuration files to take advantage of the new options that use the datastax-java-driver prefix.
  
  For details about the DataStax Java driver enhancements, start with the Driver options topic. Also refer to the Executor options topic. Several of the Driver and Executor options have been deprecated and replaced by settings that use the datastax.java.driver prefix. In addition, the Security options topic has been removed, with the options moved into the SSL section of the Driver options topic.

- You can connect DataStax Bulk Loader to a cloud-based DataStax Apollo database by including the path to the secure connect bundle, and by specifying the username and password entered when the database was created. For details about using the --driver.basic.cloud.secureConnectBundle option, which was implemented for Apollo in support of its Open Beta program, refer to Loading and unloading data with DataStax Bulk Loader.
  
  Open Beta participants can download the secure connect bundle from the DataStax Cloud console after creating an Apollo database. (DAT-412)
  
  Also see the examples in the dsbulk load topic.

- You can use DataStax Bulk Loader to load/unload your table data from/to compressed CSV or JSON files. For details, refer to the --connector.{csv | json}.compression parameter. (DAT-449)

DataStax Bulk Loader 1.3.4 release notes

16 July 2019
DataStax Bulk Loader 1.3.4 release notes include:

- 1.3.4 Changes and enhancements
- 1.3.4 Resolved issues
1.3.4 Changes and enhancements

After upgrading to 1.3.4, be sure to review and adjust your scripts to use the changed settings.

- The DataStax Bulk Loader Help provides an entry for --version. (DAT-383)
- Improved error message provided when a row fails to decode. (DAT-411)

In the DataStax Bulk Loader logging options, the format is: -maxErrors, --log.maxErrors (number | "N %")

We have provided an updated explanation:

The maximum number of errors to allow before aborting the entire operation. This setting may be expressed as:

# An absolute number of errors; in which case, set this value to an integer greater than or equal to zero.

# Or a percentage of the total rows processed so far; in which case, set this value to a string of the form "N %", where N is a decimal number between 0 and 100 exclusive. Example: -maxErrors "20%"

Setting this value to any negative integer disables the feature, which is not recommended.

- When a table contains static columns, it is possible that some partitions only contain static data. In this case, that data is exported as a pseudo row where all clustering columns and regular columns are null. Example:

```
create table t1 (pk int, cs int static, cc int, v int, primary key (pk, cc));
```

```
insert into t1 (pk, cs) values (1,1);
```

```
select * from t1;
```

```
<table>
<thead>
<tr>
<th>pk</th>
<th>cc</th>
<th>cs</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>null</td>
<td>1</td>
<td>null</td>
</tr>
</tbody>
</table>
```

In prior DataStax Bulk Loader releases, you could not import this type of static data, even though the query was valid. For example, the following query was rejected:

```
INSERT INTO t1 (pk, cs) values (:pk, :cs);
```

Operation LOAD_20190412-134352-912437 failed: Missing required primary key column conversation_id from schema.mapping or schema.query.

DataStax Bulk Loader now allows this valid query. (DAT-414)

- You can use the CQL date and time types with UNITS_SINCE_EPOCH, in addition to timestamp. Previously, you could only use the CQL timestamp type. On the dsbulk command, you can use codec.unit and codec.epoch to convert integers to, or from, these types. Refer to --codec.unit and --codec.epoch. (DAT-428)

- You can use a new monitoring setting, monitoring.trackBytes, to enable or disable monitoring of DataStax Bulk Loader operations in bytes per second. Because this type of monitoring can consume excessive allocation resources, and in some cases excessive CPU cycles, the setting is disabled by default. If you want monitoring in bytes per second, you must enable it with monitoring.trackBytes. Test and compare the setting in your development environment. Disabling this setting may improve the allocation rate. If leaving the
setting disabled improves throughput, consider disabling it (or keeping it disabled) in production. Or enable monitoring of bytes per second on an as-needed basis. (DAT-432)

- The default output file name format, defined by the `--connector.(csv|json).fileNameFormat string` option, no longer includes the thousands separator. The prior default output file name format was:

  ```
  # output-%0,6d.csv
  # output-%0,6d.json
  ```

  The updated default format is:

  ```
  # output-%06d.csv
  # output-%06d.json
  ```

Refer to `--connector.(csv | json).fileNameFormat string` (DAT-443)

- In load operations, you can pass the URL of a CSV or JSON data file. In cases where you have multiple URLs, DataStax Bulk Loader 1.3.4 makes this task easier by providing the following command-line options. You can point to the file that contains all the URLs for the data files:

  ```
  # --connector.csv.urlfile string
  # --connector.json.urlfile string
  ```

Refer to `--connector.(csv | json).urlfile string`. (DAT-445)

- When using `REPLICA_SET` batch mode, the server may issue query warnings if the number of statements in a single batch exceeds `unlogged_batch_across_partitions_warn_threshold`. To avoid reporting excessive warning messages in `stdout`, DataStax Bulk Loader logs only one warning at the beginning of the operation. (DAT-451)

### 1.3.4 Resolved issues

- DataStax Bulk Loader should reject CSV files containing invalid headers, such as headers that are empty or contain duplicate fields. (DAT-427)

- Logging option `-maxErrors 0` does not abort the operation. (DAT-333)

- DataStax Bulk Loader should reject invalid execution IDs. An execution ID is used to create MBean names. DataStax Bulk Loader now validates user-provided IDs to ensure, for example, that an ID does not contain a comma. (DAT-441)

### DataStax Bulk Loader 1.3.3 release notes

13 March 2019

DataStax Bulk Loader 1.3.3 release notes summarize the resolved issue.

#### 1.3.3 Resolved issue

Export of varchar column containing JSON may truncate data. (DAT-400)

Columns of type `varchar` that contain JSON are now exported “as is,” meaning DataStax Bulk Loader does not attempt to parse the JSON payload.
For example, assume you had a column col1 whose value was:

'{"foo":42}'

This was previously exported as shown below. That is, the contents of the column were parsed into a JSON node:

```
col1 = {'foo':42}
```

In DataStax Bulk Loader 1.3.3, the JSON `{"foo":42}` in a varchar column is exported as a string:

```
col1 = "{"foo":42}"
```

## DataStax Bulk Loader 1.3.2 release notes

20 February 2019

DataStax Bulk Loader 1.3.2 release notes include:

- 1.3.2 Changes and enhancements
- 1.3.2 Resolved issues

### 1.3.2 Changes and enhancements

After upgrading to 1.3.2, be sure to review and adjust scripts to use changed settings.

- Print basic information about the cluster. (DAT-340)
  - Refer to Printing cluster information.
- Unload timestamps as units since an epoch. (DAT-364)
  - Datasets containing numeric data that are intended to be interpreted as units since a given epoch require the setting `codec.timestamp=UNITS_SINCE_EPOCH`. Failing to specify this special format will result in all records being rejected due to an invalid timestamp format. Refer to Codec options.
- Provide better documentation on how to choose the best batching strategy. (DAT-353)
  - Refer to Batch options for the dsbulk command.
- Implement unload and count for materialized views. (DAT-385)
- Calculate batch size dynamically - Adaptive Batch Sizing. (DAT-352)
  - The new setting, `batch.maxSizeInBytes`, defaults to -1 (unlimited).
  - `batch.maxBatchSize` is deprecated; instead, use `batch.maxBatchStatements`.
  - `batch.bufferSize` should be a multiple of `batch.maxBatchStatements`. (DAT-389)
  - By default `batch.bufferSize` is set to 4 times `batch.maxBatchStatements` if its value is less than or equal to 0.
- Improve support for lightweight transactions. (DAT-384)
  - DataStax Bulk Loader can detect write failures due to a failed LWT write. Records that could not be inserted will appear in two new files:
    1. `paxos.bad` is a new "bad file" devoted to LWT write failures.
    2. `paxos-erros.log` is a new debug file devoted to LWT write failures.
DataStax Bulk Loader also writes any records from failed writes to a .bad file in the operation's directory, depending on when the failure occurred. For details, refer to Detection of write failures.

- Extend DataStax Bulk Loader rate limiting capability to reads. (DAT-336)
  Previously, the rate limiter used by DataStax Bulk Loader, and adjustable via the `--executor.maxPerSecond` setting, only applied to writes. DataStax Bulk Loader extends the functionality to reads by making it consider the number of received rows instead of the number of requests sent.

- Expose settings to control how to interpret empty fields in CSV files. (DAT-344)
  There are two new settings for the CSV connector:

  1. `nullValue`
  2. `emptyValue`

  Previously when reading a CSV file, the connector would emit an empty string when a field was empty and non-quoted. By default, starting with DataStax Bulk Loader 1.3.2, the CSV connector will return a null value in such situations, which may not make a difference in most cases. The only noticeable difference will be for columns of type `VARCHAR` or `ASCII`; the resulting stored value will be null instead of an empty string.

- Allow functions to appear in mapping variables. (DAT-327)
  Previously for loads only, a mapping entry could contain a function on the right side of the assignment. This functionality has been extended to unloads. For example, loads may continue to use:

  ```java
  now() = column1
  ```

  On load, the result of the `now()` function is inserted into `column1` for every row.

  For unloads, you can export the result of `now()` as `fieldA` for every row read. For example:

  ```java
  fieldA = now()
  ```

- Detect writetime variable when unloading. (DAT-367)
  You can specify a `writetime` function in a mapping definition when unloading. For example:

  ```java
  fieldA = column1, fieldB = writetime(column1)
  ```

  In this example, because the data type is detected, `fieldB` will be exported as a timestamp, not as an integer.

- Relax constraints on queries for the Count workflow. (DAT-308)
  The `schema.query` setting can contain any `SELECT` clause when counting rows.

- Automatically add token range restriction to `WHERE` clauses. (DAT-319)
  When a custom query is provided with `--schema.query`, to enable read parallelization, it is no longer necessary to provide a `WHERE` clause using the form:

  ```sql
  WHERE token(pk) > :start AND token(pk) <= :end
  ```

  If the query does not contain a `WHERE` clause, DataStax Bulk Loader will automatically generate that `WHERE` clause. However, if the query contains a `WHERE` clause, DataStax Bulk Loader will not be able to parallelize the read operations.

- Should allow JSON array mapping with UDTs. (DAT-316)
Previously, when loading User Defined Types (UDTs) it was required that the input be a JSON object to allow for field-by-field mapping. Starting with DataStax Bulk Loader 1.3.2, a JSON array can also be mapped to UDTs, in which case the mapping is based on field order.

- Improve `WHERE` clause token range restriction detection. (DAT-372)

  When you provide a custom query for unloading, the token range restriction variables can have any name, not only `start` and `end`. For example, the following is valid:

  ```
  SELECT * FROM table1 WHERE token(pk) > :foo AND token(pk) <= :bar
  ```

- Remove record location URI. (DAT-370)

  DataStax Bulk Loader previously provided a record's URI to uniquely identify the record. However, the URI was very long and difficult to read. You can instead identify a failed record by looking into the record's source statement or row.

- Allow columns and fields to be mapped more than once. (DAT-373)

  It is possible to map a field/column more than once. The following rules apply:

  - When loading, a field can be mapped to 2 or more columns, but a column cannot be mapped to 2 or more fields. Thus the following mapping is correct: `fieldA = column1, fieldA = column2`.
  - When unloading, a column can be mapped to 2 or more fields, but a field cannot be mapped to 2 or more columns. Thus the following mapping is correct: `fieldA = column1, fieldB = column1`.

- UDT and tuple codecs should respect `allowExtraFields` and `allowMissingFields`. (DAT-315)

  The settings `schema.allowMissingFields` and `schema.allowExtraFields` apply to UDTs and tuples. For example, if a tuple has three elements, but the JSON input only has two elements, this scenario results in an error if `schema.allowMissingFields` is `false`. However, this scenario is accepted if `schema.allowMissingFields` is `true`. The missing element in this example is assigned as null.

- Add support for DataStax Enterprise 4.8 and lower. (DAT-312)

  DataStax Bulk Loader is compatible with C* 1.2 and later releases, and DataStax Enterprise 3.2 and later releases. All protocol versions are supported. Some features might not be available depending on the protocol version and server version.

  The `schema.splits` (default: `8C`) setting was added to compensate for the absence of paging in C* 1.2. The token ring is split into small chunks and is controlled by this setting.

  For example:

  ```
  $ bin/dsbulk unload -url myData.csv --driver.pooling.local.connections 8 \
  --driver.pooling.local.requests 128 --driver.pooling.remote.requests 128 \
  --schema.splits 0.5C -k test -t test
  ```

  On `--schema.splits`, you can optionally use special syntax, `nC`, to specify a number that is a multiple of the available cores, resulting in a calculated number of splits. If the number of cores is 8, `--schema.splits 0.5C = 0.5 * 8`, which results in 4 splits. Refer to `--schema.splits number`.

- Add support for keyspace-qualified UDFs in mappings. (DAT-378)

  If needed, you can qualify a user-defined function (UDF) with a keyspace name. For example: `fieldA = ks1.func1(column1, column2)`

- Allow fields to appear as function parameters on the left side of mapping entries. (DAT-379)

  When loading, a mapping entry can contain a function on the left side that references fields of the dataset. For example, consider the case where:

  - A dataset has two fields, `fieldA` and `fieldB`
  - A table with three columns: `colA`, `colB` and `colSum`
# A user-define function: sum(int, int)

The following mapping works:

```cql
fieldA = colA, fieldB = colB, sum(fieldA, fieldB)=colSum
```

This will store the sum of `fieldA` and `fieldB` into `colSum`.

- Improve handling of search queries. (DAT-309)
  You can supply a DataStax Enterprise search predicate using the `solr_query` mechanism. For example, assume you create a search index on the `dsbulkblog.iris_with_id` table:

```cql
$ cqlsh -e "CREATE SEARCH INDEX IF NOT EXISTS ON dsbulkblog.iris_with_id"
```

You can issue a query for just the `Iris-setosa` rows:

```cql
$ dsbulk unload -query "SELECT id, petal_length, petal_width, 
sepal_length, sepal_width, species FROM dsbulkblog.iris_with_id 
WHERE solr_query = '{"q": \"species:Iris-setosa\"}"
```

- Ability to hard-limit the number of concurrent continuous paging sessions. (DAT-380)
  DataStax Bulk Loader adds a new setting: `executor.continuousPaging.maxConcurrentQueries` (Default: 60). It sets the maximum number of concurrent continuous paging queries that should be carried in parallel. Set this number to a value equal to or less than the value configured server-side for `continuous_paging.max_concurrent_sessions` in the `cassandra.yaml` configuration file, which is also 60 by default. Otherwise some requests may be rejected. You can disable `executor.continuousPaging.maxConcurrentQueries` by assigning any negative value or 0.

- Ability to skip unloading or loading the `solr_query` column. (DAT-365)
  DataStax Bulk Loader will skip the `solr_query` column when loading and unloading.

### 1.3.2 Resolved issues

- Setting `executor.maxInFlight` to a negative value triggers fatal error. (DAT-392)
- Murmur3TokenRangeSplitter should allow long overflows when splitting ranges. (DAT-334)
- CSV connector trims trailing white space when reading data. (DAT-339)
- Avoid overflows in `CodecUtils.numberToInstant`. (DAT-368)
- Call to `ArrayBackedRow.toString()` causes fatal NPE. (DAT-369)

### DataStax Bulk Loader 1.2.0 release notes

1 August 2018

DataStax Bulk Loader 1.2.0 release notes include:

- **1.2.0 Changes and enhancements**
- **1.2.0 Resolved issues**

#### 1.2.0 Changes and enhancements
DataStax Bulk Loader release notes

After upgrade to 1.2.0, be sure to review and adjust scripts to use changed settings.

- Improve range split algorithm in multi-DC and vnodes environments. (DAT-252)
- Support simplified notation for JSON arrays and objects in collection fields. (DAT-317)

1.2.0 Resolved issues:

- CSVWriter trims leading/trailing whitespace in values. (DAT-302)
- CSV connector fails when the number of columns in a record is greater than 512. (DAT-311)
- Bulk Loader fails when mapping contains a primary key column mapped to a function. (DAT-326)

DataStax Bulk Loader 1.1.0 release notes

18 June 2018

DataStax Bulk Loader 1.1.0 release notes include:

- 1.1.0 Changes and enhancements
- 1.1.0 Resolved issues

1.1.0 Changes and enhancements

After upgrade to 1.1.0, be sure to review and adjust scripts to use changed settings.

- Combine batch.mode and batch.enabled into a single setting: batch.mode. If you are using the batch.enabled setting in scripts, change to batch.mode with value DISABLED. (DAT-287)
- Improve handling of Univocity exceptions. (DAT-286)
- Logging improvements. (DAT-290)
  
  # Log messages are logged only to operation.log. Logging does not print to stdout.
  
  # Configurable logging levels with the log.verbosity setting.
  
  # The setting log.ansiEnabled is changed to log.ansiMode.
- New count workflow. (DAT-291, DAT-299)
  
  # Supports counting rows in a table.
  
  # Configurable counting mode.
  
  # When mode = partitions, configurable number of partitions to count. Support to count the number of rows for the $n$ biggest partitions in a table.
- Counter tables are supported for load and unload. (DAT-292)
- Improve validation to include user-supplied queries and mappings. (DAT-294)
- The codec.timestamp CQL_DATE_TIME setting is renamed to CQL_TIMESTAMP. Adjust scripts to use the new setting. (DAT-298)

1.1.0 Resolved issues:

- Generated query does not contain all token ranges when a range wraps around the ring. (DAT-295)
• Empty map values do not work when loading using dsbulk. (DAT-297)
• DSBulk cannot handle columns of type list<timestamp>. (DAT-288)
• Generated queries do not respect indexed mapping order. (DAT-289)
• DSBulk fails to start with Java 10+. (DAT-300)

DataStax Bulk Loader 1.0.2 release notes
5 June 2018
DataStax Bulk Loader 1.0.2 release notes include:

1.0.2 Changes and enhancements

• DataStax Bulk Loader 1.0.2 is bundled with DSE 6.0.1. (DSP-16206)
• Configure whether to use ANSI colors and other escape sequences in log messages printed to standard output and standard error. (DAT-249)

DataStax Bulk Loader 1.0.1 release notes
17 April 2018
DataStax Bulk Loader 1.0.1 release notes include:

1.0.1 Changes and enhancements

• DataStax Bulk Loader (dsbulk) version 1.0.1 is automatically installed with DataStax Enterprise, and can also be installed as a standalone tool. DataStax Bulk Loader 1.0.1 is supported for use with DSE 5.0 and later. (DSP-13999, DSP-15623)
• Support to manage special characters on the command line and in the configuration file. (DAT-229)
• Improve error messages for incorrect mapping. (DAT-235)
• Improved monitoring options. (DAT-238)
• Detect console width on Windows. (DAT-240)
• Null words are supported by all connectors. The schema.nullStrings is changed to codec.nullWords. Renamed the convertTo and convertFrom methods. See Codec options and Schema options. (DAT-241)
• Use Logback to improve filtering to make stack traces more readable and useful. On ANSI-compatible terminals, the date prints in green, the hour in cyan, the level is blue (INFO) or red (WARN), and the message prints in black. (DAT-242)
• Improved messaging for completion with errors. (DAT-243)
• Settings schema.allowExtraFields and schema.allowMissingFields are added to reference.conf. (DAT-244)
• Support is dropped for using :port to specify the port to connect to. Specify the port for all hosts only with driver.port. (DAT-245)
1.0.1 Resolved issues

- Numeric overflows should display the original input that caused the overflow. (DAT-237)
- Null words are not supported by all connectors. (DAT-241)
- Addresses might not be properly translated when cluster has custom native port. (DAT-245)
Chapter 3. Installing DataStax Bulk Loader 1.4.1

DataStax Bulk Loader lets you efficiently and reliably load and unload CSV/JSON data in and out of:

- DataStax Enterprise (DSE) 4.7 and later databases
- Open source Apache Cassandra® 2.1 and later databases
- DataStax Distribution of Apache Cassandra (DDAC) databases
- DataStax Apollo cloud databases

DataStax recommends using the latest dsbulk version, which is currently 1.4.1.

DataStax Bulk Loader is supported on Linux, macOS, and Windows platforms.

You can use DataStax Bulk Loader as a standalone tool that connects remotely to a cluster. The tool is not required to run locally on a cluster node, but can be used in this configuration.

Before upgrading to DataStax Bulk Loader 1.4.0 and later releases, note that 1.4.0 added support for the latest 2.x version of the DataStax Java driver. Many new driver options are available directly with dsbulk commands via the datastax-java-driver prefix. If you are using a pre-4.7 DSE release, the new driver options are not supported and you must use or remain on DataStax Bulk Loader 1.3.4.

For open source Apache Cassandra 2.1 and later databases, DataStax Bulk Loader 1.4.1 added support for load and count operations; previous DataStax Bulk Loader releases supported unload operations only.

End User License Agreement (EULA). By downloading this DataStax product, you agree to the terms of the EULA.

1. Download the tarball or zip file from the DataStax Bulk Loader download page. Select the package for your OS: A tar file is provided for Linux and macOS; a zip file is provided for Windows.

2. If you agree, enable the Terms checkbox and click the Download button.

3. Unpack the distribution. Linux example:

   ```
   $ tar -xzvf dsbulk-1.4.1.tar.gz
   ```

   The files are downloaded and extracted into the current directory.

What’s next:

If you previously used a package install of DSE or DDAC on the node where you just installed dsbulk, a prior version of dsbulk was included, such as 1.2.0 or 1.3.0. After unpacking the latest version of dsbulk from the standalone tarball, update your PATH so that it points to the new version.
For example, on a macOS node, edit your $HOME/.bashrc file, adding a command such as:

```
$ export PATH=path-to-unpacked-location/dsbulk-1.4.1/bin:$PATH
```

From the command line, execute your updated .bashrc, and verify the dsbulk version. Example:

```
$ source ~/.bashrc

$ dsbulk --version

DataStax Bulk Loader v1.4.1
```

Next, learn how to get started with dsbulk.
Chapter 4. Architecture

The DataStax Workflow Engine is the component responsible for the orchestration of loading and unloading operations. The main features are:

- **Configuration**: The engine collects user-supplied settings, merges them with default values and configures the loading/unloading operation to run.

- **Connection**: The engine handles the driver connection to:
  - DataStax Enterprise (DSE) 4.7 and later databases
  - Open source Apache Cassandra® 2.1 and later databases
  - DataStax Distribution of Apache Cassandra (DDAC) databases
  - DataStax Apollo cloud databases

  The engine manages the driver-specific settings, as well as supports authentication and SSL encryption.

- **Conversion**: The engine handles data type conversions, e.g. boolean, number, date conversions from anything (typically, strings or raw bytes as emitted by a connector) to appropriate internal representations (typically, Java Temporal or Number objects). It also handles **NULL** and **UNSET** values.

- **Mapping**: The engine analyzes metadata gathered from the driver and infers the appropriate **INSERT** or **SELECT** prepared statement, then checks this information against user-supplied information about the data source, to infer the bound variables to use.

- **Monitoring**: The engine reports metrics about all its internal components, mainly the connector and the bulk executor.

- **Error Handling**: The engine handles errors from both connectors and the bulk executor, and reports read, parse, and write failures. These are redirected to a configurable “bad file” that contains sources that could not be loaded.

**Figure 1: Loading Workflow**
Figure 2: Unloading Workflow

Architecture
Chapter 5. Getting Started

To help you get started, this topic describes the key features of DataStax Bulk Loader and its `dsbulk` command.

**Prerequisites:**

Obtain the following information and resources:

- Download and install DataStax Bulk Loader.
- Create DataStax tables using a tool like CQLSH. `dsbulk` commands load, unload, and count data, but it does not create keyspaces or tables.

**Key features**

- **Load data** without a configuration file
- **Unload data** without a configuration file
- **Create a configuration file** for use with loading or unloading
- **Use SSL** with `dsbulk`
- **Print cluster information to the console** with the `dsbulk` command
- **Count data** in tables

**Loading data without a configuration file**

The `dsbulk` command examples often show a parameter such as `-url filename.csv` or `-url filename.json`. Optionally, you can load or unload data from/to compressed CSV or JSON files. For details, refer to the `--connector.(csv|json).compression` option.

Load CSV or JSON data with a `dsbulk load` command.

**Load data from a local file**

Load data from a local file `export.csv` with headers into keyspace `ks1` and table `table1`:

Apollo

```
$ dsbulk load -url export.csv -k ks1 -t table1 -b "path/to/secure-connect-database_name.zip" -u database_user -p database_password -header true
```

DSE/DDAC

```
$ dsbulk load -url export.csv -k ks1 -t table1 -h '10.200.1.3, 10.200.1.4' -header true
```

*url* can designate the path to a resource, such as a local file, or a web URL from which to read/write data.

**Specify an external data source**

Specify an external source of data. For DSE/DDAC, you can indicate a port for the cluster hosts:
Getting Started

Apollo

$ dsbulk load -url https://svr/data/export.csv -k ks1 -t table1 \ -b "path/to/secure-connect-database_name.zip" -u database_user -p database_password

DSE/DDAC

$ dsbulk load -url https://svr/data/export.csv -k ks1 -t table1 -h '10.200.1.3, 10.200.1.4' -port 9876

Specify a file with URLs

Specify a file that contains a list of multiple, well-formed URLs for the CSV or JSON data files to load:

Apollo

$ dsbulk load --connector.json.urlfile "my/local/multiple-input-data-urls.txt" -k ks1 -t table1 \ -b "path/to/secure-connect-database_name.zip" -u database_user -p database_password

DSE/DDAC

$ dsbulk load --connector.json.urlfile "my/local/multiple-input-data-urls.txt" -k ks1 -t table1 -h '10.200.1.3'

Load CSV data from stdin

Load CSV data from stdin as it is generated from a loading script generate_data. The data is loaded to the keyspace ks1 and table table1. If not specified, the field names are read from a header row in the input file.

Apollo

$ generate_data | dsbulk load -url stdin:/ -k ks1 -t table1 \ -b "path/to/secure-connect-database_name.zip" -u database_user -p database_password

DSE/DDAC

$ generate_data | dsbulk load -url stdin:/ -k ks1 -t table1

Unloading data without a configuration file

Unload CSV or JSON data with a dsbulk unload command.

Unload data from an external file

Specify the external file to write the data to from keyspace ks1 and table table1:
Creating a configuration file for DataStax Bulk Loader

The configuration file for setting DataStax Bulk Loader and related DataStax Java Driver values are written in a simple format, one option per line.

Load a configuration from a local file

```
####### my-application.conf #######

dsbulk {
    # The name of the connector to use
    connector.name = "csv"
    # CSV field delimiter
    connector.csv.delimiter = "|"
    # The keyspace to connect to
    schema.keyspace = "myKeyspace"
    # The table to connect to
    schema.table = "myTable"
    # The field-to-column mapping
    schema.mapping = "0=name, 1=age, 2=email"
}

datastax-java-driver {
    advanced {
        auth-provider {
            class = "PlainTextAuthProvider"
            username = "myAccount4"
            password = "dse#r0cks!"
        }
    }
}
```

For information about using SSL with DataStax Bulk Loader, refer to Using SSL with dsbulk.

Starting in DataStax Bulk Loader 1.4.0, all driver settings need to be prefixed with datastax-java-driver. On the command line, you can shorten this prefix to driver. All other DataStax Bulk Loader settings (that are not driver related) must use the dsbulk prefix in configuration files. However, on the command line, you can omit the dsbulk prefix. To avoid confusion, configuration files are formatted with the following equivalent Human-Optimized Config Object Notation (HOCON) syntax: dsbulk { connector.name = "csv" ... }. For information about HOCON, refer to this specification.

DataStax recommends specifying username and password credentials in a configuration file instead of on the command line. When done in a configuration file, the credentials are not displayed in subsequent commands such as ps -ef | grep cqlsh.
Getting Started

To use the configuration file, specify `-f filename`, where `filename` is the configuration file. Example:

```
$ dsbulk load -f my-application.conf -url export.csv -k ksi -t table1
```

Using SSL with dsbulk

To use SSL with DataStax Bulk Loader, first refer to the DSE Security docs to set up SSL. While the SSL options can be specified on the command line, using a configuration file is recommended.

As explained in Creating a configuration file for DataStax Bulk Loader, be sure to enclose the Java driver configuration options in the `datastax-java-driver { ... }` block.

The `datastax-java-driver` prefix was introduced with the DataStax Bulk Loader 1.4.0 release, as a result of upgrading DataStax Bulk Loader to use the latest 2.x version of the DataStax Java driver. For related information, refer to Driver options.

```yaml
dsbulk {
    connector.name = "csv"
    connector.csv.delimiter = "|"
    schema.keyspace = "myKeyspace"
    schema.table = "myTable"
    schema.mapping = "0=name, 1=age, 2=email"
}

datastax-java-driver {
    advanced {
        ssl-engine-factory {
            keystore-password = "cassandra"
            keystore-path = "/Users/myaccount/tmp/ssl/keystore.node0"
            class = DefaultSslEngineFactory
            truststore-password = "dse#r0cks!"
            truststore-path = "/Users/myaccount/tmp/ssl/truststore.node0"
        }
    }
}
```
Enclose passwords that contain special characters in quotes, as shown in the example:

```
truststore-password = "dse#r0cks!"
```

Optionally, you can create a separate configuration file such as `driver.conf` that contains all the `datastax-java-driver` properties. Then reference that separate file in your DataStax Bulk Loader configuration file, by using a statement such as:

```
include classpath("driver.conf")
```

On the command line, use the `-f` switch to specify your config file. Example:

```
$ dsbulk load -f my-application.conf -url file1.csv -k ks1 -t table1
```

### Printing cluster information

When you enable verbose logging by using `--log.verbosity 2` on the `dsbulk` command, DataStax Bulk Loader prints basic information about the associated cluster. These data points often help when you are investigating any load or unload issues.

- **Partitioner:** name-of-partitioner
- **Total number of hosts:** number
- **DataCenters:** list-of-datacenter-names
- **Hosts:** list-of-hosts: address, dseVersion, cassandraVersion, dataCenter

The output is sorted by ascending IP address. If there are more than 100 nodes comprising the cluster, the other nodes are not listed. Instead, DataStax Bulk Loader prints (Other nodes omitted).

Log messages are only logged to the main log file, `operation.log`, and to standard error. Nothing from the log is printed to `stdout`. For information about log levels, refer to Logging Options.

### Counting data in tables

Use the `dsbulk count` command to return information about the loaded data.

For example, the following command returns information about the partition data used in the `cycling.comments` table. The results are organized as follows:

1. Left column: partition key value
2. Middle column: number of rows using that partition key value
3. Right column: the partition’s percentage of rows compared to the total number of rows in the table

```
dsbulk count -k cycling -t comments --stats.modes partitions --stats.numPartitions 50
```

```
Operation directory: /home/automaton/cycling/logs/
COUNT_20190424-213840-954894

<table>
<thead>
<tr>
<th>total</th>
<th>failed</th>
<th>rows/s</th>
<th>mb/s</th>
<th>kb/row</th>
<th>p50ms</th>
<th>p99ms</th>
<th>p999ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
<td>74</td>
<td>0.00</td>
<td>0.02</td>
<td>27.59</td>
<td>31.33</td>
<td>31.33</td>
</tr>
</tbody>
</table>

Operation COUNT_20190424-213840-954894 completed successfully in 0
seconds.
```

DataStax Bulk Loader 1.4.1 Latest version
Additional options are provided with the `dsbulk count` command. Refer to Count options.
Chapter 6. Setting up Kerberos client authentication

When loading data into a Kerberos enabled DataStax Enterprise cluster or DataStax Distribution of Apache Cassandra™ cluster, DataStax Bulk Loader must provide Kerberos credentials using one of the following methods:

- Keytab file
- Ticket Cache

Configuring the location of the Kerberos Configuration file

Authenticating with Kerberos credentials using a keytab file or ticket cache requires the Kerberos configuration file (`krb5.conf`). Typically, this file is in the `/etc` directory. If it is not, obtain one from your Kerberos system administrator.

To use a location other than `/etc`, set the environment variables for Kerberos command line tools, such as `kinit`, `klist`, and `kdestroy` and DS Bulk Loader.

If the Kerberos configuration file is not the default location (`/etc`), set the path to the file using the environment variables:

1. Set the `KRB5_CONFIG` environment variable to the location of `krb5.conf`.
   
   The following shows an example of the file location: `~JAVA_HOME/lib/security/krb5.conf`.
   
   ```
   $ EXPORT KRB5_CONFIG=$JAVA_HOME/lib/security/krb5.conf
   ```

2. Add the path to `DSBULK_JAVA_OPTS`.
   
   The following shows an example of the file location: `~JAVA_HOME/lib/security/krb5.conf`.
   
   ```
   $ EXPORT DSBULK_JAVA_OPTS=$DSBULK_JAVA_OPTS -Djava.security.krb5.conf=$JAVA_HOME/lib/security/krb5.conf
   ```

Using a Kerberos Keytab file for authentication

To use a Kerberos keytab file, first use the `kadmin` command to create the keytab file and get a ticket.

Prerequisites:

These steps require MIT Kerberos tools:

- `kinit`
- `kadmin`

1. Create a keytab file with `kadmin`.
Setting up Kerberos client authentication

1. Start `kadmin`:

   
   ```shell
   $ kadmin
   ```

2. Create file:

   
   ```shell
   $ ktadd -k file_name principal_name
   ```

3. Login using `kinit`:

   
   ```shell
   $ kinit -k -t file_name principal_name
   ```

2. Authenticate from Bulk Loader using the ticket:

   • To configure Bulk Loader to use the keytab file, in the `application.conf` set:

     ```
     # driver.auth.provider to DseGSSAPIAuthProvider.
     # driver.auth.principal to the principal name.
     # driver.auth.keyTab to keytab file using the full path.
     
     If multiple principals may have valid tickets in the ticket cache, DSBulk arbitrarily chooses one to use. Specify the principal explicitly by setting the `driver.auth.principal` to the principal name.
     
     For example:
     ```
     
     ```
     """
     ############ MyConfFile.conf ############
     
     dsbulk { 
     # The name of the connector to use 
     connector.name = "csv"
     # CSV field delimiter 
     connector.csv.delimiter = "|"
     # The keyspace to connect to 
     schema.keyspace = "myKeyspace"
     # The table to connect to 
     schema.table = "myTable"
     # The field-to-column mapping 
     schema.mapping = "0=name, 1=age, 2=email"
     # The authentication configuration for Kerberos 
     driver.auth.provider=DseGSSAPIAuthProvider
     driver.auth.principal="principal_name"
     driver.auth.keyTab="file_path"
     }
     """
     
     Additional command line parameters are not required when using this option.

   • Specify Kerberos options on the command line:

     ```shell
     $ dsbulk load -k ks -t t1 -url ~/data.csv \ --driver.auth.provider DseGSSAPIAuthProvider \ --driver.auth.principal dsbulk_principal_name \ --driver.auth.keyTab file_path
     ```

Using a Kerberos Ticket Cache for authentication

To use the Kerberos ticket cache, first use the `kinit` command to authenticate with the Kerberos server and obtain a ticket.
Prerequisites:
These steps require MIT Kerberos tools:

- kinit
- klist

1. Get a Kerberos ticket:
   a. Authenticate with the Kerberos server and obtain a ticket:

   $ kinit principal_name@REALM

   b. Verify the ticket and expiration:

   $ klist

   One or more tickets display in the list with the expiration time.

   Ticket cache: FILE:/tmp/krb5cc_1002
   Default principal: principal_name@REALM
   Valid starting       Expires              Service principal
   02/14/2019 21:53:51  02/15/2019 07:53:51  krbtgt/host@REALM
   renew until 02/15/2019 21:53:49

2. Authenticate from Bulk Loader using the ticket:

   - To configure Bulk Loader to use a ticket in the cache, in the application.conf set the
     driver.auth.provider to DseGSSAPIAuthProvider.

     If multiple principals may have valid tickets in the ticket cache, DSBulk arbitrarily chooses one to use.
     Specify the principal explicitly by setting the driver.auth.principal to the principal name.

     For example:

     ############ MyConfFile.conf ############

     dsbulk {
       # The name of the connector to use
       connector.name = "csv"
       # CSV field delimiter
       connector.csv.delimiter = "\""
       # The keyspace to connect to
       schema.keyspace = "myKeyspace"
       # The table to connect to
       schema.table = "myTable"
       # The field-to-column mapping
       schema.mapping = "0-name, 1-age, 2-email"
       # The authentication provider for Kerberos
       driver.auth.provider="DseGSSAPIAuthProvider"
       driver.auth.principal="principal_name"
     }

     Additional command line parameters are not required when using this option.

     - Specify Kerberos options on the command line:
Setting up Kerberos client authentication

# Use any cached ticket:

```
$ dsbulk load -k ks -t t1 -url ~/data.csv --driver.auth.provider DseGSSAPIAuthProvider
```

# Use a specific principal when more than one ticket is cached:

```
$ dsbulk load -k ks -t t1 -url ~/data.csv --driver.auth.provider DseGSSAPIAuthProvider --driver.auth.principal dsbulk_principal_name
```
dsbulk

DataStax Bulk Loader dsbulk can be used for loading and unloading data to or from:

- DataStax Enterprise (DSE) 4.7 and later databases
- Open source Apache Cassandra® 2.1 and later databases
- DataStax Distribution of Apache Cassandra (DDAC) databases
- DataStax Apollo cloud databases

Two subcommands, load and unload, are straightforward. Both subcommands require the options keyspace and table, or a schema.query, plus a data source.

A wide variety of options are also available to help you tailor how DataStax Bulk Loader operates. These options have defined default values or values inferred from the input data, if the operation is loading, or from the database data, if the operation is unloading. The options described here are grouped functionally, so that additional requirements can be noted. For example, if loading or unloading CSV data, the connector.csv.url option must be set, specifying the path or URL of the CSV data file used for loading or unloading.

The standalone tool is launched using the command dsbulk from within the bin directory of your distribution. The tool also provides inline help for all settings. A configuration file specifying option values can be used, or options can be specified on the command line. Options specified on the command line will override the configuration file option settings.

**Synopsis**

```
$ dsbulk ( load | unload | count ) [options] (( -k | --keyspace ) keyspace_name ( -t | --table ) table_name) | ( --schema.query string ) [ help | --help ]
```

| Table 1: Legend |
|-----------------|-----------------|
| **Syntax conventions** | **Description** |
| *italics* | Variable value. Replace with a user-defined value. |
| [] | Optional. Square brackets ([ ]) surround optional command arguments. Do not type the square brackets. |
| {} | Group. Parentheses ({ }) identify a group to choose from. Do not type the parentheses. |
| | Or. A vertical bar (|) separates alternative elements. Type any one of the elements. Do not type the vertical bar. |
| [ -- ] | Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options. |
General use
Get general help about dsbulk and the common options:

```
$ dsbulk help
```

Get help about particular dsbulk options, such as connector.csv options using the help subcommand:

```
$ dsbulk help connector.csv
```

Run dsbulk -c csv with --help option to see its short options, along with the general help:

```
$ dsbulk -c csv --help
```

Display the version number:

```
$ dsbulk --version
```

Escaping and Quoting Command Line Arguments
When supplied via the command line, all option values are expected to be in valid HOCON syntax. For example, control characters, the backslash character, and the double-quote character all need to be properly escaped. For example, \t is the escape sequence that corresponds to the tab character, whereas \\ is the escape sequence for the backslash character:

```
$ dsbulk load -delim '\t' -url 'C:\Users\My Folder'
```

In general, string values containing special characters also need to be properly quoted with double-quotes, as required by the HOCON syntax:

```
$ dsbulk load -url "C:\Users\My Folder"
```

However, when the expected type of an option is a string, it is possible to omit the surrounding double-quotes, for convenience. Thus, note the absence of the double-quotes in the first example. Similarly, when an argument is a list, it is possible to omit the surrounding square brackets; making the following two lines equivalent:

```
$ dsbulk load --codec.nullStrings 'NIL, NULL' dsbulk load --codec.nullStrings '[NIL, NULL]
```

The same applies for arguments of type map: it is possible to omit the surrounding curly braces, making the following two lines equivalent:

```
$ dsbulk load --connector.json.deserializationFeatures '{ USE_BIG_DECIMAL_FOR_FLOATS : true }'
$ dsbulk load --connector.json.deserializationFeatures 'USE_BIG_DECIMAL_FOR_FLOATS : true'
```

This syntactic sugar is only available for command line arguments of type string, list or map: all other option types, as well as all options specified in a configuration file must be fully compliant with HOCON syntax, and it is the user's responsibility to ensure that such options are properly escaped and quoted.

Detection of write failures
In the Cassandra documentation, you may have encountered one or more of the following terms, all of which have the same meaning:
• Lightweight Transactions (LWT), used in this topic
• Compare-And-Set (CAS)
• Paxos protocol

DataStax Bulk Loader detects any failures due to failed LWT write operations. In 1.3.2 or later, records that could not be inserted are shown in two files:

• `paxos.bad` is the "bad file" devoted to LWT write failures.
• `paxos-erros.log` is the debug file devoted to LWT write failures.

DataStax Bulk Loader also writes any failed records to one of the following files in the operation's directory, depending on when the failure occurred:

• If while parsing data, the records are written to `connector.bad`.
• If while mapping data to the supported DSE, DDAC, DataStax Apollo, Apache Cassandra databases, the records are written to `mapping.bad`.
• If while inserting data into any of those supported databases, the records are written to `load.bad`.

The operation's directory is the `logs` subdirectory under the location from which you ran the `dsbulk` command.

Loading data examples

This topic presents examples of using the `dsbulk load` command to write CSV or JSON data to a database table.

Databases supported by DataStax Bulk Loader

DataStax Bulk Loader supports the use of the `dsbulk load`, `dsbulk unload`, and `dsbulk count` commands with:

• DataStax Enterprise (DSE) 4.7 and later databases
• Open source Apache Cassandra® 2.1 and later databases
• DataStax Distribution of Apache Cassandra (DDAC) databases
• DataStax Apollo cloud databases
### dsbulk load examples

Load data from CSV data read from stdin to table `table1` in keyspace `ks1`:

```bash
$ dsbulk load -k ks1 -t table1
```

Load a configuration file from `/tmp/dsbulk_load.conf` to use for loading the file `export.csv` to table `table1` in keyspace `ks1`:

```bash
$ dsbulk load -f /tmp/dsbulk_load.conf --connector.csv.url export.csv -k ks1 -t table1
```

Load the file `export.csv` to table `table1` in keyspace `ks1` using the short form option for `url`:

```bash
$ dsbulk load -url export.csv -k ks1 -t table1
```

Load the file `export.csv` to table `table1` in keyspace `ks1` using the short form option for `url` and the tab character as a field delimiter:

```bash
$ dsbulk load -k ks1 -t table1 -url export.csv -delim '	'
```

Load the file `/tmp/export.csv` to table `table1` in keyspace `ks1` using the long form option for `url`:

```bash
$ dsbulk load --connector.csv.url file:///tmp/export.csv -k ks1 -t table1
```

Note that `file:///tmp/export.csv` is loading from localhost, hence the empty host in the `file://` designation.
Load table `table1` in keyspace `ks1` from a gzipped CSV file by unzipping it to stdout and piping to stdin of `dsbulk`:

```
$ gzcat table1.csv.gz | dsbulk load -k ks1 -t table1
```

Specify a few hosts (initial contact points) that belong to the desired cluster and load from a local file, without headers. Map field indices of the input to table columns with `-m`:

```
$ dsbulk load -url ~/export.csv -k ks1 -t table1 -h '10.200.1.3, 10.200.1.4' -header false -m '0=col1,1=col3'
```

Specify port 9876 for the cluster hosts and load from an external source URL:

```
$ dsbulk load -url https://192.168.1.100/data/export.csv -k ks1 -t table1 -h '10.200.1.3,10.200.1.4' -port 9876
```

Load all csv files from a directory. The files do not have a header row, `-header false`. Map field indices of the input to table columns with `-m`:

```
$ dsbulk load -url ~/export-dir -k ks1 -t table1 -header false -m '0=col1,1=col3'
```

Specify a file that contains a list of multiple, well-formed URLs for the CSV or JSON data files to load:

```
$ dsbulk load --connector.json.urlfile "my/local/multiple-input-data-urls.txt" -k ks1 -t table1 -h '10.200.1.3'
```

Load data using a CQL query and include TTL values from the input data:

```
$ dsbulk load -query "INSERT INTO dsbulkblog.iris_with_id (sepal_length,sepal_width,petal_length,petal_width,species,id) VALUES (:sepal_length,:sepal_width,:petal_length,:petal_width,:species,:id) USING TTL :ttl_to_use"
```

Load data using a field-to-column mapping. The example also shows how a function call can be used to apply a function to one or more fields before mapping to a single column named `odd_column`:

```
$ dsbulk load -url /tmp/dsbulkblog/iris.csv -k dsbulkblog -t iris_with_id --schema.mapping "sepal_length = sepal_length, sepal_width = sepal_width, petal_length = petal_length, petal_width = petal_width,"
```
DataStax Bulk Loader reference

```
petal_width = petal_width, species = species, id = id, max(sepal_length, petal_length) = odd_column
```

With default port for cluster hosts, keyspace, table, and mapping set in `conf/application.conf`:

```
$ dbulk load -url https://192.168.1.100/data/export.csv -h '10.200.1.3,10.200.1.4'
```

Load table `table1` in keyspace `ks1` from a CSV file, where double-quote characters in fields are escaped with a double-quote; for example, "f1", "value with ""quotes"" and more" is a line in the CSV file:

```
$ dbulk load -url ~/export.csv -k ks1 -t table1 -escape '"'
```

Loading collections to a table has some specific helpful simplification. Collections inside a CSV file can contain valid JSON or simpler non-compliant JSON. For example, for the following table:

```
CREATE TABLE t1 (col1 set<int> PRIMARY KEY, col2 list<int>, col3 map<text,int>);
```

This pipe-delimited CSV file contains valid JSON and could be loaded:

```
$ col1|col2|col3 1,2,3|1,2,3|{"key1":1,"key2":2}
```

Or, you could load the same CSV file as follows:

```
$ col1|col2|col3 1,2,3|1,2,3|"key1":1,"key2":2
```

Notice that the surrounding brackets and braces are omitted from the valid JSON in the third column.

Simplification of the JSON data will not work for nested collections. If your table has a column `col4` of type `list<list<int>>`, only the outermost structure can omit the surrounding characters. For example:

```
$ col1,col2,col3,col4 1,2,3|1,2,3|{"key1":1,"key2":2}|[[1,2,3],[4,5,6]]
```

You can simplify the prior example, as follows:

```
$ col1,col2,col3,col4 1,2,3|1,2,3|"key1":1,"key2":2|[1,2,3],[4,5,6]
```

However, the inner-list items cannot be simplified further.

**Unloading data examples**

Unloading is the process of using the `dbulk unload` command to extract data from a database table into a CSV or JSON file. Many `dbulk` options used in loading and unloading are the same.

The `dbulk` command examples often show a parameter such as `-url filename.csv` or `-url filename.json`. Optionally, you can load or unload data from/to compressed CSV or JSON files. For details, refer to the `--connector.(csv|json).compression` option.

**Databases supported by DataStax Bulk Loader**

DataStax Bulk Loader supports the use of the `dbulk load`, `dbulk unload`, and `dbulk count` commands with:

- DataStax Enterprise (DSE) 4.7 and later databases
- Open source Apache Cassandra® 2.1 and later databases
dsbulk unload examples

Unload data to stdout from the `ks1.table1` table in a cluster with a localhost contact point. Column names in the table map to field names in the data. Field names must be emitted in a header row in the output:

```
$ dsbulk unload -k ks1 -t table1
```

Unload data to stdout from the `ks1.table1` table and gzip the result:

```
$ dsbulk unload -k ks1 -t table1 | gzip > table1.gz
```

Unload data to a local directory (which may not yet exist):

```
$ dsbulk unload -url ~/data-export -k ks1 -t table1
```

Unload data on a cluster with authentication and SSL enabled:

```
$ dsbulk unload -h '
["fe80::f861:3eff:feld:9d7a"]'
-u myaccount
-p mypassword
--driver.auth.provider DsePlainTextAuthProvider
--driver.ssl.provider JDK
--driver.ssl.keystore.path /etc/dse/keystores/client.keystore
--driver.ssl.keystore.password mysslkspass
--driver.ssl.truststore.path /etc/dse/keystores/client.truststore
--driver.ssl.truststore.password myssltrustpass
-k mykeyspace
-t mytable
-url mytable_backup
```

For more details, refer to SSL options.

Unload and count data from a materialized view named `health_data_mv` to a local directory:

```
$ dsbulk unload count -k healthcare
-t health_data_mv
-url ~/export-mv
```

In DataStax Bulk Loader 1.3.2 and later, you can use `dsbulk unload count` and specify a materialized view on the `-t` or `--table` option. Previously, you could only use `dsbulk unload count` with tables. For related information, refer to Creating a materialized view.

Unload data using a search index:

```
dsbulk unload -query
"SELECT id, petal_length, petal_width, sepal_length, sepal_width, species
FROM dsbulkblog.iris_with_id
WHERE solr_query = "escape:\"species:Iris-setosa\"\""
--executor.continuousPaging.enabled false
```

where a search index exists for the table `iris_with_id`. Continuous Paging must be disabled because DSE Search does not work with that feature. The `solr_query` requires double-escaping of the double-quotes in the `solr_query` string.

Faceted queries are not supported.
Unload data using a CQL query using function calls:

```bash
$ dbulk unload -query "SELECT id, species, writetime(species) AS TIMESTAMP, ttl(species) AS TTL FROM dbulkblog.iris_with_id"
```

Exit codes

The `dbulk` command has exit codes that are returned to a calling process. The following values link the integer value returned with the status:

<table>
<thead>
<tr>
<th>Integer value</th>
<th>Status value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>STATUS_OK</td>
</tr>
<tr>
<td>1</td>
<td>STATUS_COMPLETED_WITH_ERRORS</td>
</tr>
<tr>
<td>2</td>
<td>STATUS_ABORTED_TOO_MANY_ERRORS</td>
</tr>
<tr>
<td>3</td>
<td>STATUS_ABORTED_FATAL_ERROR</td>
</tr>
<tr>
<td>4</td>
<td>STATUS_INTERRUPTED</td>
</tr>
<tr>
<td>5</td>
<td>STATUS_CRASHED</td>
</tr>
</tbody>
</table>

Common options

Some options are commonly required to use `dbulk`. In the following list, required options are designated.

The options can be used in short form (`-k keyspace_name`) or long form (`--schema.keyspace keyspace_name`).

--version

Show the program's version number and exit.

Default: unspecified

-f filename

Load options from the given file rather than from `dbulk_home/conf/application.conf`.

Default: unspecified

-c,--connector.name { csv | json }

The name of the connector to use.

Supported: `dbulk load` and `dbulk unload` operations.

Default: csv

-b,--driver.basic.cloud.secureConnectBundle secure-connect-database-name.zip

Specifies the path to a secure connect bundle used to connect with a DataStax Apollo database. The specified location must be a path on the local filesystem or a valid URL. Download the secure connect bundle for a DataStax Apollo database from the DataStax Cloud console.

The following examples show different methods of indicating the path to the secure connect bundle:

- Path on *Nix systems
  - `/path/to/secure-connect-database-name.zip`
  - `./path/to/secure-connect-database-name.zip`
  - `~/path/to/secure-connect-database-name.zip`

- Path on *Nix relative to working directory
  - `@/secure-connect-database-name.zip`
  - `@/secure-connect-database-name.zip`
  - `@/secure-connect-database-name.zip`

- Path on Microsoft Windows systems
  - `C:\path\to\secure-connect-database-name.zip`

- You must escape backslashes in HOCON
  - `file:/path/to/secure-connect-database-name.zip`
  - `http://host.com/secure-connect-database-name.zip`

If a secure connect bundle is specified using this parameter, any of the following options are ignored and a warning is logged:
• Contact points
• Consistency level other than LOCAL_QUORUM (only for loading operations)
• SSL configurations

Default: none

-k, --schema.keyspace string

Keyspace used for loading, unloading, or counting data. Keyspace names should not be quoted and are case-sensitive. MyKeyspace will match a keyspace named MyKeyspace but not mykeyspace. Required option if schema.query is not specified; otherwise, optional.

Default: unspecified

-t, --schema.table string

Table used for loading, unloading, or counting data. Table names should not be quoted and are case-sensitive. MyTable will match a table named MyTable but not mytable. Required option if schema.query is not specified; otherwise, optional.

Default: unspecified

-m, --schema.mapping string

The field-to-column mapping to use. Applies to loading and unloading. If not specified, DataStax Bulk Loader applies a strict one-to-one mapping between the source fields and the database table. If that is not your intention, you must supply an explicit mapping. Mappings should be specified as a map of the following form:

- Indexed data sources: $0 = col1, 1 = col2, 2 = col3$, where 0, 1, 2, are the zero-based indices of fields in the source data; and col1, col2, col3 are bound variable names in the insert statement.
- A shortcut to map the first n fields is to simply specify the destination columns: col1, col2, col3.
- Mapped data sources: $fieldA = col1$, $fieldB = col2$, $fieldC = col3$, where fieldA, fieldB, fieldC are field names in the source data; and col1, col2, col3 are bound variable names in the insert statement.
- A shortcut to map fields named like columns is to simply specify the destination columns: col1, col2, col3.

To specify that a field should be used as the timestamp (write time) or as ttl (time to live) of the inserted row, use the specially named fake columns _ttl and _timestamp: $fieldA = _timestamp$, $fieldB = _ttl$. Timestamp fields can be parsed as CQL timestamp columns and must use the format specified in either codec.timestamp or codec.unit + codec.epoch; the latter is an integer representing the number of units specified by codec.unit since the specified epoch. TTL fields are parsed as integers representing duration in seconds and must use the format specified in codec.number.

To specify that a column should be populated with the result of a function call for loading operations, specify the function call as the input field (e.g. $now() = c4$). Similarly, to specify that a field should be populated with the result of a function call for unloading operations, specify the function call as the input column (e.g. $field1=now()$). Function calls can also be qualified by a keyspace name: $field1 = keyspace1.max(c1, c2)$.

In addition, for mapped data sources, it is also possible to specify that the mapping be partly auto-generated and partly explicitly specified. For example, if a source row has fields c1, c2, c3, and c5, and the table has columns c1, c2, c3, c4, one can map all like-named columns and specify that c5 in the source maps to c4 in the table as follows: $* = *, c5 = c4$.

To specify that all like-named fields be mapped, except for c2, use: $* = -c2$. To skip c2 and c3, use: $* = [-c2, -c3]$.

Any identifier, field, or column, that is not strictly alphanumeric (that is, not matching [a-zA-Z0-9_]+) must be surrounded by double-quotes, just like you would do in CQL: "Field ""A"""" = ""Column 2"" (to escape a double-quote, simply double it).

Unlike CQL grammar, unquoted identifiers will not be lowercased by DataStax Bulk Loader. An identifier such as MyColumn1 will match a column named MyColumn1, but will not match mycolumn1.
The exact type of mapping to use depends on the connector being used. Some connectors can only produce indexed records; others can only produce mapped ones, while others are capable of producing both indexed and mapped records at the same time. Refer to the connector’s documentation to know which kinds of mapping it supports.

Default: null

-url, --connector.{csv | json}.url string
The URL or path of the resources to read from or write to. Possible options are - (representing stdin for reading and stdout for writing) and file (filepath).

File URLs can also be expressed as simple paths without the file prefix. A directory of files can also be specified. The following examples provide descriptions of using this parameter in different ways:

Specify a few hosts (initial contact points) that belong to the desired cluster and load from a local file, without headers. Map field indices of the input to table columns with -m:

```
$ dsbulk load -url ~/export.csv -k ks1 -t table1 -h '10.200.1.3, 10.200.1.4' -header false -m '0=col1,1=col3'
```

Specify port 9876 for the cluster hosts and load from an external source URL:

```
$ dsbulk load -url https://192.168.1.100/data/export.csv -k ks1 -t table1 -h '10.200.1.3,10.200.1.4' -port 9876
```

Load all csv files from a directory. The files do not have a header row, -header false. Map field indices of the input to table columns with -m:

```
$ dsbulk load -url ~/export-dir -k ks1 -t table1 -header false -m '0=col1,1=col3'
```

See Loading data examples for more examples.

Default: -

-delim, --connector.csv.delimiter string
The character to use as field delimiter.
Default: , (a comma)

-header, --connector.csv.header { true | false }
Enable or disable whether the files to read or write begin with a header line. If enabled for loading, the first non-empty line in every file will assign field names for each record column, in lieu of schema.mapping, fieldA = col1, fieldB = col2, fieldC = col3. If disabled for loading, records will not contain fields names, only field indexes, 0 = col1, 1 = col2, 2 = col3. For unloading, if this setting is enabled, each file will begin with a header line, and if disabled, each file will not contain a header line.

This option will apply to all files loaded or unloaded.

Default: true

-h, --driver.basic.contact-points, --datastax-java-driver.basic.contact-points host_name(s)
The contact points to use for the initial connection to the cluster. This must be a list of strings with each contact point specified as host or host:port. If the host is specified without a port, the default port specified in basic.default-port will be used. Apache Cassandra 3.0 and earlier and DataStax Enterprise (DSE) 6.7 and earlier require all nodes in a cluster to share the same port.

If the host is a DNS name that resolves to multiple A-records, all the corresponding addresses will be used. Do not use localhost as a host-name (because it resolves to both IPv4 and IPv6 addresses on some platforms). The port for all hosts must be specified with driver.port.
Be sure to enclose address strings that contain special characters in quotes, as shown in these examples:

$ dcbulk unload -h "[fe80::f861:3eff:fe1d:9d7a]" -query "SELECT * from foo.bar;"

$ dcbulk unload -h "[fe80::f861:3eff:fe1d:9d7b","fe80::f861:3eff:fe1d:9d7c]" -query "SELECT * from foo1.bar1;"

Default: 127.0.0.1

-port, --driver.basic.default-port, --datastax-java-driver.basic.default-port port_number

The port to use for basic.contact-points, when a host is specified without a port. All nodes in a cluster must accept connections on the same port number.

Default: 9042

**Connector options**

Connectors allow different types of data to be loaded and unloaded using dcbulk. The general format for connector options is:

```
--connector.name.option { string | number }
```

For example:

```
$ dcbulk load -k ks1 -t table1 --connector.json.urlfile "my/local/multiple-input-data-urls.txt"
```

The available URL protocols depend on which URL stream handlers have been installed. At a minimum, the file protocol is supported for reading and writing, and the HTTP/HTTPS protocols are supported for reading.

The file protocol can be used with all supported file systems, local or remote.

- When reading: the URL can point to a single file or to an existing directory. In the case of a directory, you can specify the `--connector.{csv | json}.fileNamePattern` option to filter the files to be read. And you can use the `--connector.{csv | json}.recursive` option to control whether the connector should also look for files in subdirectories.
- When writing: the URL is treated as a directory. If it doesn't exist, DataStax Bulk Loader attempts to create it. If successful, CSV files are created in this directory, and you can set the file names to be used with the `--connector.{csv | json}.fileNameFormat` option.

If the value specified does not have a protocol, it is assumed to be a file protocol. Relative URLs will be resolved against the current working directory. Also, for your convenience, if the path begins with a tilde (~), that symbol will be expanded to the current user’s home directory.

- If you have URLs of multiple CSV or JSON data files to load, you can create a file that contains the list of well-formed URLs, and specify the single file with `--connector.{csv | json}.urlfile`.
- Another option is to use `--connector.{csv | json}.compressed` to load or unload data from/to a compressed file.

The options can be used in short form (`-k keyspace_name`) or long form (`--schema.keyspace keyspace_name`).

-c,--connector.name { csv | json }

The name of the connector to use.
Supported: dsbulk load and dsbulk unload operations.
Default: csv

Common to both CSV and JSON connectors
--connector.(csv | json).compression string
Specify the compression type to use when loading or unloading data from/to a compressed file. Two examples:

```
$ dsbulk unload -k test -t table1 --connector.json.compression gzip -url mydir
```

```
$ dsbulk load -k test -t table1 --connector.json.compression gzip -url mydir
```

The default output from the dsbulk unload command, with compression and the first counter, is output-000001.csv.gz. Refer to --connector.(csv | json).fileNameFormat string for details on dsbulk unload output file naming.

Supported compressed file types for dsbulk load and dsbulk unload operations:

- bzip2
- deflate
- gzip
- lzma
- lz4
- snappy
- xz
- zstd

Supported compressed file types for only dsbulk load:

- brotli
- deflate64
- z

DataStax Bulk Loader automatically adjusts the default file pattern while searching for the file to load, by appending the default compression file extension (such as .gz, for a gzip compression type) to the defined filename pattern. Also, when unloading data, the default file extension for a given compression is automatically appended to the value of the fileNameFormat option, if the compressed file extension ends with .csv or .json.

Supported: dsbulk load and dsbulk unload operations.
Default: none

--connector.(csv | json).fileNameFormat string
The file name format to use when writing during unloading. This option is ignored when reading and for any URL that is not a file. The file name must comply with the formatting rules of String.format(), and must contain a %d format specifier that will be used to increment file name counters.
Supported: Only dsbulk unload operations.
Default: output-%06d.(csv | json)

--connector.(csv | json).filePattern string
The glob pattern to use when searching for files to read. The syntax to use is the glob syntax, as described in java.nio.file.FileSystem.getPathMatcher(). This option is ignored when writing and
for URLs that are not files. Only applicable when the --connector.{csv | json}.url option points to a directory on a known file system; ignored otherwise.

Supported: Only dsbulk load operations.

Default: **/*.{csv | json}

-encoding,--connector.{csv | json}.encoding **string**

The file encoding to use for all read or written files.

Supported: dsbulk load and dsbulk unload operations.

Default: UTF-8

-maxConcurrentFiles,--connector.{csv | json}.maxConcurrentFiles **string**

The maximum number of files that can be written simultaneously. This option is ignored when loading and when the output URL is anything other than a directory on a filesystem. The special syntax NC can be used to specify a number of threads that is a multiple of the number of available cores, e.g. if the number of cores is 8, then 0.5C = 0.5 * 8 = 4 threads.

Supported: Only dsbulk unload operations.

Default: 0.25C

-maxRecords,--connector.{csv | json}.maxRecords **number**

The maximum number of records to read from or write to each file. When reading, all records past this number will be discarded. When writing, a file will contain at most this number of records; if more records remain to be written, a new file will be created using --connector.{csv | json}.fileNameFormat.

Note that when writing to anything other than a directory, this option is ignored. For CSV, --connector.csv.maxRecords takes into account --connector.csv.header. If a file begins with a header line, that line is not counted as a record. This feature is disabled by default.

Supported: dsbulk load and dsbulk unload operations.

Default: -1

--connector.{csv | json}.recursive { true | false }

Enable or disable scanning for files in the root’s subdirectories. Only applicable when url is set to a directory on a known filesystem.

Supported: Only dsbulk load operations.

Default: false

-skipRecords,--connector.{csv | json}.skipRecords **number**

The number of records to skip from each input file before the parser can begin to execute. Note that if the file contains a header line (for CSV), that line is not counted as a valid record.

Supported: Only dsbulk load operations.

Default: 0

-url,--connector.{csv | json}.url **string**

The URL or path of the resource(s) to read from or write to. Possible options are - (representing stdin for reading and stdout for writing) and file (filepath). File URLs can also be expressed as simple paths without the file prefix. A directory of files can also be specified.

Supported: dsbulk load and dsbulk unload operations.

Default: -

--connector.{csv | json}.urlfile **string**

The URL or path of the local file that contains a list of CSV or JSON data files from which to read during dsbulk load operations.

This --connector.{csv | json}.urlfile option and the --connector.{csv | json}.url option are mutually exclusive. If both are defined and not empty, the --connector.{csv | json}.urlfile option takes precedence.

In the file with URLs:

- Encode in UTF-8.
- Each line should contain one path and a valid URL to load.
- You do not need to escape characters in the path.
DataStax Bulk Loader reference

- The format rules documented in this topic, including rules for `fileNamePattern`, `recursive`, and `fileNameFormat`, also apply to `connector.{csv | json}.urlfile`.
- You can use the `#` character to comment out a line.
- DataStax Bulk Loader removes any leading or trailing white space from each line.

Supported: Only `dsbulk load` operations.

Do not use `connector.{csv | json}.urlfile` with `dsbulk unload`; doing so results in a fatal error.

Default: unspecified.

CSV connector options

- `--connector.csv.comment string`
  
  The character that represents a line comment when found in the beginning of a line of text. Only one character can be specified.
  
  Supported: `dsbulk load` and `dsbulk unload` operations.
  
  Default: disabled by default and indicated with a null character value `"\u0000"`

- `--connector.csv.delimiter string`
  
  The character to use as field delimiter.
  
  Supported: `dsbulk load` and `dsbulk unload` operations.
  
  Default: `,` (a comma)

- `--connector.csv.emptyValue string`
  
  Sets the string representation of an empty value. When loading, if the parser does not read any character from the quoted input, the value of this option is used.
  
  Supported: Only `dsbulk load` operations.
  
  Default: `""` (empty string)

- `--connector.csv.escape string`
  
  The character used for escaping quotes inside an already quoted value. Only one character can be specified. Note that this option applies to all files to be read or written.
  
  Supported: Only `dsbulk load` operations.
  
  Default: `\`

- `--connector.csv.header { true | false }`
  
  Whether the files to read or write will begin with a header line. If enabled for loading, the first non-empty line in every file will assign field names for each record column, in lieu of `schema.mapping`, `fieldA = col1, fieldB = col2, fieldC = col3`. If disabled for loading, records will not contain fields names, only field indexes, `0 = col1, 1 = col2, 2 = col3`. For unloading, if this option is enabled, each file will begin with a header line, and if disabled, each file will not contain a header line.
  
  Supported: `dsbulk load` and `dsbulk unload` operations.
  
  Default: `true`

- `--connector.csv.newline { auto | string }`
  
  The string of one or two characters that represents a line ending or default value of `auto`. If set to the default value, the system’s line separator (in Java, `System.lineSeparator()` determines the strings) is used for writing, and auto-detection of line endings are enabled for reading. Typical line separator characters need to be escaped. For example, the common line ending on Microsoft Windows is a carriage return followed by a newline, or `\r\n`.
  
  Supported: `dsbulk load` and `dsbulk unload` operations.
  
  Default: `auto`

- `--connector.csv.nullValue string`
  
  Sets the string representation of a `null` value. The value of this option is used when loading, where the parser does not read any character in the input, or when unloading, where the writer has a `null` object written to the output. The default value will emit a `null` when loading, and will not write any character to the output when unloading.
  
  Supported: `dsbulk load` and `dsbulk unload` operations.
  
  Default: `null`
--connector.csv.normalizeLineEndingsInQuotes boolean
Defines whether the system's line separator (in Java, System.lineSeparator() determines the strings) is replaced by a normalized line separator \n in quoted values.
On Microsoft Windows, the detection mechanism for line endings may not function properly when this option is false, due to a defect in the CSV parsing library. If problems arise, set this value to true.
Supported: dsbulk load and dsbulk unload operations.
Default: false

--connector.csv.ignoreLeadingWhitespaces boolean
Defines whether to skip leading whitespaces from values being read or written. Used for both loading and unloading.
Supported: Only dsbulk load operations.
Default: false

--connector.csv.ignoreLeadingWhitespacesInQuotes boolean
Defines whether to skip leading whitespaces in quoted values being read.
Supported: Only dsbulk load operations.
Default: false

--connector.csv.ignoreTrailingWhitespaces boolean
Defines whether to skip trailing whitespaces from values being read or written in files.
Supported: dsbulk load and dsbulk unload operations.
Default: false

--connector.csv.ignoreTrailingWhitespacesInQuotes boolean
Defines whether to skip trailing whitespaces in quoted values being read.
Supported: Only dsbulk load operations.
Default: false

--connector.csv.maxCharsPerColumn number
The maximum number of characters that a field can contain. This option is used to size internal buffers and to avoid out-of-memory problems. If set to -1, internal buffers will be resized dynamically. While convenient, this can lead to memory problems. It could also hurt throughput, if some large fields require constant resizing; if this is the case, set this value to a fixed positive number that is big enough to contain all field values.
Supported: dsbulk load and dsbulk unload operations.
Default: 4096

--connector.csv.maxColumns number
The maximum number of columns that a record can contain. This option is used to size internal buffers and to avoid out-of-memory (OOM) problems.
Supported: dsbulk load and dsbulk unload operations.
Default: 512

--connector.csv.quote character
The character used for quoting fields when the field delimiter is part of the field value. Only one character can be specified. Note that this option applies to all files to be read or written.
Supported: dsbulk load and dsbulk unload operations.
Default: \n
JSON connector options
--connector.json.mode { SINGLE_DOCUMENT | MULTI_DOCUMENT }
The mode for loading and unloading JSON documents. Valid values are:

- MULTI_DOCUMENT: Each resource may contain an arbitrary number of successive JSON documents to be mapped to records. For example the format of each JSON document is a single document: {doc1}. The root directory for the JSON documents can be specified with url and the documents can be read recursively by option connector.json.recursive to true.
• **SINGLE_DOCUMENT**: Each resource contains a root array whose elements are JSON documents to be mapped to records. For example, the format of the JSON document is an array with embedded JSON documents:

```
[ {doc1}, {doc2}, {doc3} ]
```

Supported: dsbulk load and dsbulk unload operations.
Default: MULTI_DOCUMENT

**--connector.json.parserFeatures map**

JSON parser features to enable. Valid values are all the enum constants defined in com.fasterxml.jackson.core.JsonParser.Feature. For example, a value of `{ ALLOW_COMMENTS : true, ALLOW_SINGLE_QUOTES : true }` will configure the parser to allow the use of comments and single-quoted strings in JSON data.

Supported: Only dsbulk load operations.
Default: `{ }`

**--connector.json.generatorFeatures map**

JSON generator features to enable. Valid values are all the enum constants defined in com.fasterxml.jackson.core.JsonGenerator.Feature. For example, a value of `{ ESCAPE_NON_ASCII : true, QUOTE_FIELD_NAMES : true }` will configure the generator to escape all characters beyond 7-bit ASCII and quote field names when writing JSON output.

Supported: Only dsbulk unload operations.
Default: `{ }`

**--connector.json.serializationFeatures map**

A map of JSON serialization features to set. Map keys should be enum constants defined in com.fasterxml.jackson.databind.SerializationFeature.

Supported: Only dsbulk unload operations.
Default: `{ }`

**--connector.json.deserializationFeatures map**

A map of JSON deserialization features to set. Map keys should be enum constants defined in com.fasterxml.jackson.databind.deserialization.Feature. The default value is the only way to guarantee that floating point numbers will not have their precision truncated when parsed, but can result in slightly slower parsing.

Supported: Only dsbulk load operations.
Default: `{ USE_BIG_DECIMAL_FOR_FLOATS : true }`

**--connector.json.serializationStrategy string**

The strategy for filtering out entries when formatting output. Valid values are enum constants defined in com.fasterxml.jackson.annotation.JsonInclude.Include.

The `CUSTOM` strategy is not supported.

Supported: Only dsbulk unload operations.
Default: ALWAYS

**--connector.json.prettyPrint { true | false }**

Enable or disable pretty printing. When enabled, JSON records are written with indents.

Using this option results in much bigger records.

Supported: Only dsbulk unload operations.
Default: false

---

**Count options**

Specify options for the `dsbulk count` command. These options specify how counting will be accomplished by DataStax Bulk Loader.

**Databases supported by DataStax Bulk Loader**

DataStax Bulk Loader supports the use of the `dsbulk load`, `dsbulk unload`, and `dsbulk count` commands with:
DataStax Enterprise (DSE) 4.7 and later databases

• Open source Apache Cassandra® 2.1 and later databases

• DataStax Distribution of Apache Cassandra (DDAC) databases

• DataStax Apollo cloud databases

--stats.modes { global | ranges | hosts | partitions }

Kind(s) of statistics to compute. Only applicable for count, ignored otherwise. Valid values are:

• global: Count the total number of rows in the table.

• ranges: Count the total number of rows per token range in the table.

• hosts: Count the total number of rows per hosts in the table.

• partitions: Count the total number of rows in the N biggest partitions in the table. Choose how many partitions to track with **stats.numPartitions** option. For partitions, the results are organized as follows:
  1. Left column: partition key value
  2. Middle column: number of rows using that partition key value
  3. Right column: the partition’s percentage of rows compared to the total number of rows in the table

Default: global

--stats.numPartitions number

The number of distinct partitions for which to count rows. Only applicable for count, ignored otherwise.
Default: 10

Schema options

Specify schema options for the dsbulk command.

The options can be used in short form (**-k keyspace_name**) or long form (**--schema.keyspace keyspace_name**).

- **-k, --schema.keyspace string**
  Keyspace used for loading, unloading, or counting data. Keyspace names should not be quoted and are case-sensitive. MyKeyspace will match a keyspace named **MyKeyspace** but not **mykeyspace**. Required option if schema.query is not specified; otherwise, optional.
  Default: unspecified

- **-t, --schema.table string**
  Table used for loading, unloading, or counting data. Table names should not be quoted and are case-sensitive. MyTable will match a table named **MyTable** but not **mytable**. Required option if schema.query is not specified; otherwise, optional.
  Default: unspecified

- **-m, --schema.mapping string**
  The field-to-column mapping to use. Applies to loading and unloading. If not specified, DataStax Bulk Loader applies a strict one-to-one mapping between the source fields and the database table. If that is not your intention, you must supply an explicit mapping. Mappings should be specified as a map of the following form:

  • Indexed data sources: 0 = col1, 1 = col2, 2 = col3, where 0, 1, 2, are the zero-based indices of fields in the source data; and col1, col2, col3 are bound variable names in the insert statement.

  • A shortcut to map the first n fields is to simply specify the destination columns: col1, col2, col3.
**DataStax Bulk Loader reference**

- **Mapped data sources**: fieldA = col1, fieldB = col2, fieldC = col3, where fieldA, fieldB, fieldC are field names in the source data; and col1, col2, col3 are bound variable names in the insert statement.

- A shortcut to map fields named like columns is to simply specify the destination columns: col1, col2, col3.

To specify that a field should be used as the timestamp (write time) or as ttl (time to live) of the inserted row, use the specially named fake columns __ttl and __timestamp: fieldA = __timestamp, fieldB = __ttl. Timestamp fields can be parsed as CQL timestamp columns and must use the format specified in either codec.timestamp or codec.unit + codec.epoch; the latter is an integer representing the number of units specified by codec.unit since the specified epoch. TTL fields are parsed as integers representing duration in seconds and must use the format specified in codec.number.

To specify that a column should be populated with the result of a function call for loading operations, specify the function call as the input field (e.g. now() = c4). Similarly, to specify that a field should be populated with the result of a function call for unloading operations, specify the function call as the input column (e.g. field1=now()). Function calls can also be qualified by a keyspace name: field1 = keyspace1.max(c1, c2).

In addition, for mapped data sources, it is also possible to specify that the mapping be partly auto-generated and partly explicitly specified. For example, if a source row has fields c1, c2, c3, and c5, and the table has columns c1, c2, c3, c4, one can map all like-named columns and specify that c5 in the source maps to c4 in the table as follows: * = *, c5 = c4.

To specify that all like-named fields be mapped, except for c2, use: * = -c2. To skip c2 and c3, use: * = [-c2, -c3].

Any identifier, field, or column, that is not strictly alphanumeric (that is, not matching [a-zA-Z0-9_]+) must be surrounded by double-quotes, just like you would do in CQL: "Field ""A""" = "Column 2" (to escape a double-quote, simply double it).

Unlike CQL grammar, unquoted identifiers will not be lowercased by DataStax Bulk Loader. An identifier such as MyColumn1 will match a column named MyColumn1, but will not match mycolumn1.

The exact type of mapping to use depends on the connector being used. Some connectors can only produce indexed records; others can only produce mapped ones, while others are capable of producing both indexed and mapped records at the same time. Refer to the connector’s documentation to know which kinds of mapping it supports.

Default: null

**--schema.nullToUnset { true | false }**

Specify whether to map null input values to "unset" in the database; that is, don’t modify a potentially pre-existing value of this field for this row. Only applicable for loading; otherwise ignored. Note that setting to false creates tombstones to represent null. When the protocol version in use does not support unset values (all protocol versions less than 4), this setting is forced to false and a warning will be logged. Setting ignored for counting.

This setting is applied after the codec.nullStrings setting, and may intercept nulls produced by that setting.

Default: true

**--schema.allowExtraFields { true | false }**

Specify whether to accept records that contain extra fields that are not declared in the mapping. For example, assume a record contains three fields A, B, and C, but the mapping declares only fields A and B. When this option is true, C is silently ignored and the record is considered valid. When set to false, the record is rejected. Only applicable for loading; ignored otherwise.

Default: true

**--schema.allowMissingFields { true | false }**

Specify whether to accept records that are missing fields declared in the mapping. For example, assume the mapping declares three fields A, B, and C, but a record contains only fields A and B. When this option is true, C is silently assigned null and the record is considered valid. When set to false, the
record is rejected. Also applies to user-defined types (UDTs) and tuples, with element replacing field in explanation. If the missing field is mapped to a primary key column, the record is always rejected because the database would reject the record. Only applicable for loading; ignored otherwise.
Default: false

- --schema.query \textit{string}

The query to use. If not specified, then \texttt{schema.keyspace} and \texttt{schema.table} must be specified, and \texttt{dsbulk} will infer the appropriate statement based on the table's metadata, using all available columns. If \texttt{schema.keyspace} is provided, the query need not include the keyspace to qualify the table reference.

For loading, the statement can be any \texttt{INSERT}, \texttt{UPDATE} or \texttt{DELETE} statement. \texttt{INSERT} statements are preferred for most load operations, and bound variables should correspond to mapped fields; for example, \texttt{INSERT INTO table1 (c1, c2, c3) VALUES (:fieldA, :fieldB, :fieldC)}. \texttt{UPDATE} statements are required if the target table is a counter table, and the columns are updated with incremental operations (\texttt{SET coll = coll + :fieldA}) where \texttt{:fieldA} is a column in the input data. A \texttt{DELETE} statement will remove existing data during the load operation.

For unloading and counting, the statement can be any regular \texttt{SELECT} statement. If the statement does not contain a \texttt{WHERE} clause, the engine will generate a token range restriction clause of the form: \texttt{token(...) > :start and token(...) <= :end}, and the engine will generate as many statements as there are token ranges in the cluster, thus allowing parallelization of reads while at the same time targeting coordinators that are also replicas. However, if the \texttt{SELECT} statement does contain a \texttt{WHERE} clause, the engine will only be able to parallelize the operation if that \texttt{WHERE} clause also includes a \texttt{token(...) > :start and token(...) <= :end} relation (the bound variables can have any name).

Statements can use both positional and named bound variables. Named bound variables are preferred. Named bound variables usually have names matching those of the columns in the destination table, but this is not a strict requirement; it is, however, required that their names match those of fields specified in the mapping. Positional variables will be named after their corresponding column in the destination table.

Positional variable cannot be used in \texttt{SELECT} statements that use \texttt{WHERE} clause restrictions using \texttt{token()}, \texttt{USING TIMESTAMP} clause, or \texttt{USING TTL} clause.

See \texttt{schema.mapping} setting for more information.
Default: null

--schema.queryTimestamp \textit{string}

The timestamp of inserted/updated cells during load; otherwise, the current time of the system running the tool is used. Only applicable to loading; ignored otherwise. Express the value in \texttt{ISO_ZONED_DATE_TIME} format. DSE sets the query timestamps to the nearest microsecond and truncates sub-microseconds; any sub-microsecond information specified is lost. For more information, see the \texttt{CQL Reference}.

Default: unspecified

--schema.queryTtl \textit{number}

The Time-To-Live (TTL) of inserted/updated cells during load (seconds); a value of -1 means there is no TTL. Not applicable to unloading. For more information, see the \texttt{CQL Reference}, \texttt{Setting the time-to-live (TTL) for value}, and \texttt{Expiring data with time-to-live}.
Default: -1

--schema.splits \textit{number}

The number of token range splits in which to divide the token ring. This setting determines how many read requests are generated when reading an entire table. Only used when unloading and counting; ignored otherwise.

The actual number of splits may be slightly greater or less than the number specified, depending on the actual cluster topology and token ownership. Also, it is not possible to generate fewer splits than the total number of primary token ranges in the cluster. Thus, the actual number of splits is always equal to or greater than that number.

Set the \texttt{--schema.splits} value to higher values if you experience timeouts when reading from DSE, DDAC, DataStax Apollo, or Apache Cassandra tables, especially if paging is disabled. You can optionally use the special syntax \texttt{n<core_count>} to specify a number that is a multiple of the available cores, resulting in a
calculated number of splits. For example, if the number of cores is 8, \(--\text{schema.splits} \ 0.5C \ = \ 0.5 \times 8\),
which results in 4 splits.
Default: 8C

Batch options
Specify batch options for the dsbulk command. Batch options specify how statements are grouped before writing
for loading. These options are not applicable for unloading.

\(--\text{batch.bufferSize} \ \text{number}\)
The buffer size to use for flushing batched statements. This option should be set to a multiple of
maxBatchStatements, such as 2 or 4 times its value. Higher values consume more memory and usually
do not result in any noticeable performance gain. When set to less than or equal to zero, the buffer size
is implicitly set to 4 times maxBatchStatements.
Default: -1

\(--\text{batch.maxBatchSize} \ \text{number}\)
This option is deprecated. Instead use \(--\text{batch.maxSizeInBytes} \ \text{and} \ --\text{batch.maxBatchStatements}\).

\(--\text{batch.maxSizeInBytes} \ \text{number}\)
The maximum data size that a batch can hold. This is the number of bytes required to encode all the
data to be persisted, without counting the overhead generated by the native protocol (headers, frames,
and so on). The value specified should be less than or equal to the value that has been configured
server-side for the option batch_size_fail_threshold_in_kb in cassandra.yaml.

The heuristic used to compute data sizes is not 100% accurate and sometimes underestimates the
actual size. For more information, refer to the topic Cassandra.yaml configuration file.
When set to a value less than or equal to zero, the maximum data size is considered unlimited. At least
one of maxBatchStatements or maxSizeInBytes must be set to a positive value when batching is
enabled.
Default: -1

\(--\text{batch.maxBatchStatements} \ \text{number}\)
The maximum number of statements that a batch can contain. The ideal value depends on two factors:

1. The data being loaded: the larger the data, the smaller the batches should be.
2. The batch mode: when PARTITION_KEY is used, larger batches are acceptable, whereas when
   REPLICA_SET is used, smaller batches usually perform better.

When set to a value less than or equal to zero, the maximum number of statements is considered
unlimited. At least one of maxBatchStatements or maxSizeInBytes must be set to a positive value when batching is
enabled.
Default: 32

\(--\text{batch.mode} \ \text{string}\)
The grouping mode. Valid values are:

- **DISABLED**: Disables statement batching.
- **PARTITION_KEY**: Groups together statements that share the same partition key. This is the default
  mode, and the preferred one.
- **REPLICA_SET**: Groups together statements that share the same replica set. This mode might yield
  better results for small clusters and lower replication factors, but tends to perform equally well or
  worse than PARTITION_KEY for larger clusters or high replication factors.

When tuning DataStax Bulk Loader for batching, the recommended approach is:

1. Start with **PARTITION_KEY**.
2. If the average batch size is close to 1, try increasing `bufferSize`.

3. If increasing `bufferSize` does not help, switch to REPLICA_SET and set `maxBatchStatements` or `maxSizeInBytes` to low values, which may avoid timeouts or errors.

4. To improve throughput, increase `maxBatchStatements` or `maxSizeInBytes`.

Default: `PARTITION_KEY`

**Codec options**

Specify codec options for the `dsbulk` command, which determine how record fields are parsed for loading or how row cells are formatted for unloading.

The options can be used in short form (`-locale string`) or long form (`--codec.locale string`).

```
-locale,--codec.locale string
  The locale to use for locale-sensitive conversions.
  Default: en_US
```

```
-timeZone,--codec.timeZone string
  The time zone to use for temporal conversions. When loading, the time zone will be used to obtain a timestamp from inputs that do not convey any explicit time zone information. When unloading, the time zone will be used to format all timestamps.
  Default: UTC
```

```
-nullStrings,--codec.nullStrings string
  Comma-separated list of strings that should be mapped to `null`. For loading, when a record field value exactly matches one of the specified strings, the value is replaced with `null` before writing to DSE. For unloading, this setting is only applicable for string-based connectors, such as the CSV connector: the first string specified will be used to change a row cell containing `null` to the specified string when written out. By default, no strings are mapped to `null`.
  Regardless of this setting, DataStax Bulk Loader will always convert empty strings to `null` when the target CQL type is not textual; that is, when the target is not `text`, `varchar`, or `ascii`.
  This setting is applied before `schema.nullToUnset`, hence any `null` produced by a null-string can still be left unset if required.
  Default: [ ] (no strings mapped to `null`) 
```

```
--codec.booleanStrings [ true_value:false_value, ... ]
  Specify how true and false representations can be used by DataStax Bulk Loader. Each representation is of the form `true_value:false_value`, case-insensitive. For loading, all representations are honored. For unloading, the first representation will be used and all others ignored.
  Default: ["1:0","Y:N","T:F","YES:NO","TRUE:FALSE"]
```

```
--codec.booleanNumbers [ true_value, false_value ]
  Set how true and false representations of numbers are interpreted. The representation is of the form `true_value,false_value`. The mapping is reciprocal, so that numbers are mapping to Boolean and vice versa. All numbers unspecified in this setting are rejected.
  Default: [1, 0]
```

```
--codec.number string
  The `DecimalFormat` pattern to use for conversion between `String` and CQL numeric types. See `java.text.DecimalFormat` for details about the pattern syntax to use. Most inputs are recognized: optional localized thousands separator, localized decimal separator, or optional exponent. Using `locale` en_US, 1234, 1,234, 1234.5678, 1,234.5678 and 1,234.5678E2 are all valid. For unloading and formatting, rounding may occur and cause precision loss. See `codec.formatNumbers` and `codec.roundingStrategy`.
  Default: #,##0.##
```

```
--codec.formatNumbers ( true | false )
  DataStax Bulk Loader 1.4.1 Latest version
```
Whether or not to use the `codec.number` pattern to format all numeric output. When set to `true`, the numeric pattern defined by `codec.number` will be applied. This allows for nicely-formatted output, but may result in rounding (see `codec.roundingStrategy`), or alteration of the original decimal's scale. When set to `false`, numbers will be stringified using the `toString` method, and will never result in rounding or scale alteration. Only applicable when unloading, and only if the connector in use requires stringification, because the connector, such as the CSV connector, does not handle raw numeric data; ignored otherwise.

Default: `false`

`--codec.roundingStrategy string`

The rounding strategy to use for conversions from CQL numeric types to String. Valid choices: any `java.math.RoundingMode` enum constant name, including: CEILING, FLOOR, UP, DOWN, HALF_UP, HALF_EVEN, HALF_DOWN, and UNNECESSARY. The precision used when rounding is inferred from the numeric pattern declared under `codec.number`. For example, the default `codec.number (#,##0.##)` has a rounding precision of 2, and the number 123.456 would be rounded to 123.46 if the `codec.roundingStrategy` was set to `UP`. The default value will result in infinite precision, and ignore the `codec.number` setting. Only applicable when unloading, if `codec.formatNumbers` is `true` and if the connector in use requires stringification, because the connector, such as the CSV connector, does not handle raw numeric data; ignored otherwise.

Default: `UNNECESSARY`

`--codec.overflowStrategy string`

This setting can mean one of three possibilities:

- The value is outside the range of the target CQL type. For example, trying to convert 128 to a CQL tinyint (max value of 127) results in overflow.
- The value is decimal, but the target CQL type is integral. For example, trying to convert 123.45 to a CQL int results in overflow.
- The value's precision is too large for the target CQL type. For example, trying to insert 0.1234567890123456789 into a CQL double results in overflow, because there are too many significant digits to fit in a 64-bit double.

Valid choices:

- **REJECT**: overflows are considered errors and the data is rejected. This is the default value.
- **TRUNCATE**: the data is truncated to fit in the target CQL type.

  The truncation algorithm is similar to the narrowing primitive conversion defined in The Java Language Specification, Section 5.1.3, with the following exceptions: (1) If the value is too big or too small, it is rounded up or down to the maximum or minimum value allowed, rather than truncated at bit level. For example, 128 would be rounded down to 127 to fit in a byte, whereas Java would have truncated the exceeding bits and converted to -127 instead. (2) If the value is decimal, but the target CQL type is integral, it is first rounded to an integral using the defined rounding strategy, then narrowed to fit into the target type. This can result in precision loss and should be used with caution.

Only applicable for loading, when parsing numeric inputs; it does not apply for unloading, since formatting never results in overflow.

Default: `REJECT`

`--codec.timestamp { formatter | string }`

The temporal pattern to use for String to CQL timestamp conversion. Valid choices:

- A date-time pattern
- A pre-defined formatter such as `ISO_ZONED_DATE_TIME` or `ISO_INSTANT`, or any other public static field in `java.time.format.DateTimeFormatter`
- The special formatter `CQL_TIMESTAMP`, which is a special parser that accepts all valid CQL literal formats for the `timestamp` type.
• The special formatter `UNITS_SINCE_EPOCH` is required for datasets containing numeric data that are intended to be interpreted as units since a given epoch. Once set, DataStax Bulk Loader uses the `codec.unit` and `codec.epoch` settings to determine which unit and epoch to use.

For more information on patterns and pre-defined formatters, see Patterns for Formatting and Parsing in Oracle Java documentation. For more information about CQL date, time and timestamp literals, see Date, time, and timestamp format.

When parsing, `CQL_TIMESTAMP_FORMAT` recognizes most CQL temporal literals:

<table>
<thead>
<tr>
<th>Type</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local dates</td>
<td>2012-01-01</td>
</tr>
<tr>
<td>Local times</td>
<td>12:34</td>
</tr>
<tr>
<td></td>
<td>12:34:56</td>
</tr>
<tr>
<td></td>
<td>12:34:56.123</td>
</tr>
<tr>
<td></td>
<td>12:34:56.12356</td>
</tr>
<tr>
<td></td>
<td>12:34:56.12356789</td>
</tr>
<tr>
<td>Local date-times</td>
<td>2012-01-01T12:34</td>
</tr>
<tr>
<td></td>
<td>2012-01-01T12:34:56</td>
</tr>
<tr>
<td></td>
<td>2012-01-01T12:34:56.123</td>
</tr>
<tr>
<td></td>
<td>2012-01-01T12:34:56.12356</td>
</tr>
<tr>
<td></td>
<td>2012-01-01T12:34:56.12356789</td>
</tr>
<tr>
<td>Zoned date-times</td>
<td>2012-01-01T12:34+01:00</td>
</tr>
<tr>
<td></td>
<td>2012-01-01T12:34:56+01:00</td>
</tr>
<tr>
<td></td>
<td>2012-01-01T12:34:56.123+01:00</td>
</tr>
<tr>
<td></td>
<td>2012-01-01T12:34:56.123:56+01:00</td>
</tr>
<tr>
<td></td>
<td>2012-01-01T12:34:56.123:56789+01:00</td>
</tr>
<tr>
<td></td>
<td>2012-01-01T12:34:56.123:56789+01:00[Europe/Paris]</td>
</tr>
</tbody>
</table>

When the input is a local date, the timestamp is resolved at midnight using the specified `timeZone`. When the input is a local time, the timestamp is resolved using the time zone specified under `timeZone`, and the date is inferred from the instant specified under `epoch` (by default, January 1st 1970). When formatting, this format uses the `ISO_OFFSET_DATE_TIME` pattern, which is compliant with both CQL and ISO-8601.

Default: `CQL_TIMESTAMP`

```--codec.date { formatter | string }```

The temporal pattern to use for `String` to CQL date conversion. Valid choices:

• A date-time pattern

• A pre-defined formatter such as `ISO_LOCAL_DATE`

For more information on patterns and pre-defined formatters, see Patterns for Formatting and Parsing in Oracle Java documentation. For more information about CQL date, time and timestamp literals, see Date, time, and timestamp format.

Default: `ISO_LOCAL_DATE`

```--codec.time { formatter | string }```

The temporal pattern to use for `String` to CQL time conversion. Valid choices:

• A date-time pattern, such as `HH:mm:ss`.

• A pre-defined formatter such as `ISO_LOCAL_TIME`

For more information on patterns and pre-defined formatters, see Patterns for Formatting and Parsing in Oracle Java documentation. For more information about CQL date, time and timestamp literals, see Date, time, and timestamp format.

Default: `ISO_LOCAL_TIME`

```--codec.unit```
If `codec.timestamp`, `codec.date`, or `codec.time` is set to `UNITS_SINCE_EPOCH`, the time unit specified here is used to convert numeric data to and from temporals for the following cases:

- Target column is of CQL `timestamp`, `date`, or `time` type
- Loading data with a `USING TIMESTAMP`, `USING DATE`, or `USING TIME` clause
- Unloading data with a `WRITETIME()` function call

For example, if the input is 123 and the time unit is `SECONDS`, the input will be interpreted as 123 seconds since `codec.epoch`. When loading, and the target CQL type is `numeric`, but the input is `alphanumeric` and represents a temporal literal, the time unit specified will be used to convert the parsed temporal into a numeric value. For example, if the input is `2019-02-03T19:32:45Z` and the time unit specified is `SECONDS`, the parsed temporal will be converted into the number of seconds since `codec.epoch`.

All `TimeUnit` enum constants are valid choices.

Default: `MILLISECONDS`  

--codec.epoch

If `codec.timestamp`, `codec.date`, or `codec.time` is set to `UNITS_SINCE_EPOCH`, the epoch specified here determines the relative point in time to use when converting numeric data to and from temporals for the following cases:

- Target column is of CQL `timestamp`, `date`, or `time` type
- Loading data with a `USING TIMESTAMP`, `USING DATE`, or `USING TIME` clause
- Unloading data with a `WRITETIME()` function call

For example, if the input is 123 and the epoch is `2000-01-01T00:00:00Z`, the input will be interpreted as `N` `codec.unit` since January 1st 2000.

When loading, and the target CQL type is `numeric`, but the input is `alphanumeric` and represents a temporal literal, the `codec.epoch` and `codec.unit` values will be used to convert the parsed temporal into a numeric value. For example, if the input is `2019-02-03T19:32:45Z` and the epoch specified is `2000-01-01T00:00:00Z`, the parsed timestamp will be converted to `N` `codec.units` since January 1st 2000.

When parsing temporal literals, if the input does not contain a date part, then the date part of the instant specified here will be used. For example, if the input is `19:32:45` and the epoch specified is `2000-01-01T00:00:00Z`, then the input will be interpreted as `2000-01-01T19:32:45Z`.

The value must be expressed in `ISO_ZONED_DATE_TIME`, as covered in the Oracle Java documentation.

Default: "1970-01-01T00:00:00Z"

--codec.uuidStrategy { RANDOM | FIXED | MIN | MAX }

Strategy to use when generating time-based (version 1) UUIDs from timestamps. Clock sequence and node ID parts of generated UUIDs are determined on a best-effort basis and are not fully compliant with RFC 4122. Valid values are:

- `RANDOM`: Generates UUIDs using a random number in lieu of the local clock sequence and node ID. This strategy will ensure that the generated UUIDs are unique, even if the original timestamps are not guaranteed to be unique.
- `FIXED`: Preferred strategy if original timestamps are guaranteed unique, since it is faster. Generates UUIDs using a fixed local clock sequence and node ID.
- `MIN`: Generates the smallest possible type 1 UUID for a given timestamp. This strategy doesn't guarantee uniquely generated UUIDs and should be used with caution.
- `MAX`: Generates the biggest possible type 1 UUID for a given timestamp.
This strategy doesn't guarantee uniquely generated UUIDs and should be used with caution.

Default: RANDOM

### Driver options

This topic describes a commonly-used subset of DataStax Java driver options that you can specify with the `dsbulk` command. Many additional options exist. Be sure to read the DataStax Java driver configuration reference documentation. Also refer to the driver matrix.

The options can be used in short form (`-h host_name`) or long form (`--datastax-java-driver.basic.contact-point host_name`).

DataStax Java driver configuration settings start with the prefix `datastax-java-driver`. On the `dsbulk` command line, you can abbreviate this prefix to `driver`, if you prefer.

### General options

Specify general options for using `dsbulk` with the DataStax Java driver. Use these options to define the contact points and port number for the initial connection. Additionally, define policy options pertaining to the DataStax Java driver load balancing policy settings, pooling options, query options, and socket connections.

- `-h`, `--driver.basic.contact-points`, `--datastax-java-driver.basic.contact-points` *host_name(s)*
  
  The contact points to use for the initial connection to the cluster. This must be a list of strings with each contact point specified as `host:port`. If the host is specified without a port, the default port specified in `basic.default-port` will be used. Apache Cassandra 3.0 and earlier and DataStax Enterprise (DSE) 6.7 and earlier require all nodes in a cluster to share the same port.

  If the host is a DNS name that resolves to multiple A-records, all the corresponding addresses will be used. Do not use localhost as a host-name (because it resolves to both IPv4 and IPv6 addresses on some platforms). The port for all hosts must be specified with `driver.port`.

  Be sure to enclose address strings that contain special characters in quotes, as shown in these examples:

  ```
  $ dsbulk unload -h ['"fe80::f861:3eff:fe1d:9d7a"] -query "SELECT * from foo.bar;"
  
  $ dsbulk unload -h ['"fe80::f861:3eff:fe1d:9d7b","fe80::f861:3eff:fe1d:9d7c"]' -query "SELECT * from foo1.bar1;"
  ```

  Default: 127.0.0.1

- `-port`, `--driver.basic.default-port`, `--datastax-java-driver.basic.default-port` *port_number*
  
  The port to use for `basic.contact-points`, when a host is specified without a port. All nodes in a cluster must accept connections on the same port number.

  Default: 9042

- `-b`, `--driver.basic.cloud.secure-connect-bundle`, `--datastax-java-driver.basic.cloud.secure-connect-bundle` *string*
  
  The location of the secure bundle used to connect to a cloud-based DataStax Apollo database. This setting must be a path on the local filesystem or a valid URL. Examples:

  ```
  "/path/to/bundle.zip"  # path on Linux or macOS
  
  
  
  "file:/a/path/to/bundle.zip"  # URL with file protocol
  ```

  Default: 127.0.0.1

- `-c`, `--datastax-java-driver.cloud.security.module` *string*

  The location of the cloud security module. This setting must be a path on the local filesystem or a valid URL.

  Default: datastax-java-driver.cloud.security.module
Apollo Open Beta participants can download the secure connect bundle from the DataStax Cloud console after creating an Apollo database. The secure-connect-bundle option is only for Apollo databases. Do not use the following options when connecting to cloud-based Apollo deployments:

- `datastax-java-driver.basic.contact-points`
- `datastax-java-driver.basic.request.consistency`
- `datastax-java-driver.advanced.ssl-engine-factory.*`

Default: `null`

`-cl,--driver.basic.request.consistency, --datastax-java-driver.basic.request.consistency string`

The consistency level to use for all queries. Note that stronger consistency levels usually result in reduced throughput. In addition, any level higher than `ONE` will automatically disable continuous paging, which can dramatically reduce read throughput.

Valid values are: `ANY`, `LOCAL_ONE`, `ONE`, `TWO`, `THREE`, `LOCAL_QUORUM`, `QUORUM`, `EACH_QUORUM`, `ALL`.

On cloud deployments, the only accepted consistency level when writing is `LOCAL_QUORUM`. Therefore, the default value is `LOCAL_ONE`, except when loading in cloud deployments, in which case the default is automatically changed to `LOCAL_QUORUM`.

Default: `LOCAL_ONE`

`--driver.basic.request.timeout, --datastax-java-driver.basic.request.timeout "string"`

How long the DataStax Java driver waits for a request to complete. This is a global limit on the duration of a `session.execute()` call, including any internal retries the driver might do. By default, this value is set very high because DataStax Bulk Loader is optimized for good throughput, rather than good latencies.

Default: "60 seconds"

`--driver.basic.request.default-idempotence, --datastax-java-driver.basic.request.default-idempotence {true | false}`

The default idempotence for all queries executed in DataStax Bulk Loader. Setting this option to `false` causes all unload failures to not be retried.

Default: `true`

`--driver.basic.request.serial-consistency, --datastax-java-driver.basic.request.serial-consistency string`

The serial consistency level to use during unload operations. Possible options are `LOCAL_SERIAL` or `SERIAL`.

Default: `LOCAL_SERIAL`

`--driver.basic.request.page-size, --datastax-java-driver.basic.request.page-size number`

The page size. This controls how many rows will be retrieved simultaneously in a single network roundtrip (the goal being to avoid loading too many results in memory at the same time). If there are more results, additional requests will be used to retrieve them (either automatically if you iterate with the sync API, or explicitly with the async API's `fetchNextPage` method). If the value is `0` or negative, it will be ignored and the request will not be paged.

Default: `5000`

`--driver.basic.load-balancing-policy.class, --datastax-java-driver.basic.load-balancing-policy.class string`

The load balancing policy class to use. If not qualified, the DataStax Java driver assumes that it resides in the package `com.datastax.oss.driver.internal.core.loadbalancing`. DataStax Bulk Loader uses a special policy that infers the local datacenter from the contact points. You can also specify a custom class that implements `LoadBalancingPolicy` and has a public constructor with two arguments: the `DriverContext` and a `String` representing the profile name.

Default: "com.datastax.dse.driver.internal.core.loadbalancing.DseDcInferringLoadBalancingPolicy"
**--driver.basic.load-balancing-policy.filter.class, --datastax-java-driver.basic.load-balancing-policy.filter.class** *string*

An optional custom filter to include or exclude nodes. If present, the option must be the fully-qualified name of a class that implements `java.util.function.Predicate<Node>`, and has a public constructor taking a single `DriverContext` argument. The predicate's `test(Node)` method will be invoked each time the policy processes a topology or state change. If the method returns `false`, the node will be set at distance `IGNORED`, which means the Java driver will not ever connect to it, and the node is never included in any query plan.

Default: `null`

**-dc, --driver.basic.load-balancing-policy.local-datacenter, --datastax-java-driver.basic.load-balancing-policy.local-datacenter** *string*

The datacenter that is considered local. The default load balancing policy only includes nodes from this datacenter in its query plans. Set this to a value if you want to declare the local datacenter; otherwise, the `DseDcInferringLoadBalancingPolicy` that DataStax Bulk Loader uses by default infers the local datacenter from the provided contact points.

Default: unspecified

**--driver.advanced.retry-policy.max-retries, --datastax-java-driver.advanced.retry-policy.max-retries** *number*

How many times to retry a failed query. Only valid for use with the DataStax Bulk Loader default retry policy (`MultipleRetryPolicy`).

Default: `10`

**Authorization options**

Specify authorization options for using dbulk with the DataStax Java driver.

**--driver.advanced.auth-provider.class, --datastax-java-driver.advanced.auth-provider.class** *arg*

The class of the authentication provider. If it is not qualified, the Java driver assumes that it resides in one of the following packages:

- `com.datastax.oss.driver.internal.core.auth`
- `com.datastax.dse.driver.internal.core.auth`

The DSE driver provides three implementations out of the box:

- `PlainTextAuthProvider`: uses plain-text credentials. It requires the `username` and `password` options. Should be used only when authenticating against Apache Cassandra™ clusters; not recommended when authenticating against DSE clusters.

- `DsePlainTextAuthProvider`: provides SASL authentication using the PLAIN mechanism for DSE clusters secured with `DseAuthenticator`. It requires the `username` and `password` options, and optionally, an `authorization-id`.

- `DseGssApiAuthProvider`: provides GSSAPI authentication for DSE clusters secured with `DseAuthenticator`. For detailed instructions, read the Javadoc.

You can also specify a custom class that implements `AuthProvider` and has a public constructor with a `DriverContext` argument; to simplify this step, the Java driver provides two abstract classes that can be extended: `DsePlainTextAuthProviderBase` and `DseGssApiAuthProviderBase`.

Default: `null`

**-u, --driver.advanced.auth-provider.username, --datastax-java-driver.advanced.auth-provider.username** *string*

The username to use. Providers that accept this setting:

- `PlainTextAuthProvider`
- `DsePlainTextAuthProvider`
DataStax recommends specifying username and password credentials in a configuration file, instead of on the command line. For an example, refer to [Creating a configuration file for DataStax Bulk Loader](#).

Default: null

-\p, --driver.advanced.auth-provider.password, --datastax-java-driver.advanced.auth-provider.password

string

The password to use. Providers that accept this setting:

- PlainTextAuthProvider
- DsePlainTextAuthProvider

Default: null

--driver.advanced.auth-provider.authorization-id, --datastax-java-driver.advanced.auth-provider.authorization-id

string

An authorization ID allows the currently authenticated user to act as a different user (proxy authentication). Providers that accept this setting:

- DsePlainTextAuthProvider
- DseGssApiAuthProvider

Default: null

**SSL options**

Specify SSL encryption options for using dsbulk with the DataStax Java driver. For additional information on SSL, see the [Oracle Java Guide on SSL](#).

--driver.advanced.ssl-engine-factory.class, --datastax-java-driver.advanced.ssl-engine-factory.class

string

The class of the SSL engine factory. If not qualified, the DataStax Java driver assumes that it resides in the package `com.datastax.oss.driver.internal.core.ssl`. The DataStax Java driver provides a single implementation `DefaultSslEngineFactory`, which uses the JDK's built-in SSL implementation.

You can also specify a custom class that implements `SslEngineFactory` and has a public constructor with a `DriverContext` argument.

Default: null

--driver.advanced.ssl-engine-factory.hostname-validation, --datastax-java-driver.advanced.ssl-engine-factory.hostname-validation

boolean

Whether to require validation that the hostname of the server certificate's common name matches the hostname of the server being connected to. This setting is only required when using the default SSL factory. If not set, defaults to true.

Default: true

--driver.advanced.ssl-engine-factory.truststore-path, --datastax-java-driver.advanced.ssl-engine-factory.truststore-path

string

The locations used to access truststore contents. If either `truststore-path` or `keystore-path` are specified, the DataStax Java driver builds an SSLContext from these files. This setting is only required when using the default SSL factory. If neither option is specified, the default SSLContext is used, which is based on system property configuration.

Default: null

--driver.advanced.ssl-engine-factory.truststore-password, --datastax-java-driver.advanced.ssl-engine-factory.truststore-password

string
The password used to access truststore contents. This setting is only required when using the default SSL factory.
Default: null

--driver.advanced.ssl-engine-factory.keystore-path, --datastax-java-driver.advanced.ssl-engine-factory.keystore-path string
The locations used to access keystore contents. If either truststore-path or keystore-path are specified, the DataStax Java driver builds an SSLContext from these files. This setting is only required when using the default SSL factory. If neither option is specified, the default SSLContext is used, which is based on system property configuration.
Default: null

--driver.advanced.ssl-engine-factory.keystore-password, --datastax-java-driver.advanced.ssl-engine-factory.keystore-password string
The password used to access keystore contents. This setting is only required when using the default SSL factory.
Default: null

Continuous paging options
Continuous paging options only take effect if continuous paging is globally enabled, which can be done with the executor option dsbulk.executor.continuousPaging.enabled.

--driver.advanced.continuous-paging.pages-size, --datastax-java-driver.advanced.continuous-paging.page-size number
Set the page size. The value can be interpreted in number of rows or in number of bytes, depending on the page-size-in-bytes boolean value. This page size option controls how many rows (or how much data) is retrieved simultaneously in a single network roundtrip. The goal is to avoid loading too many results in memory at the same time. If there are more results, additional requests are used to retrieve them automatically (if you iterate with the sync API), or explicitly with the async API's fetchNextPage method. The default is the same as the driver's normal request page size: 5000 (rows).
Default: 5000

--driver.advanced.continuous-paging.pages-size-in-bytes, --datastax-java-driver.advanced.continuous-paging.page-size-in-bytes {true | false}
Whether the page-size option should be interpreted in number of rows or bytes. The default of false means page size is interpreted as the number of rows.
Default: false

--driver.advanced.continuous-paging.max-pages, --datastax-java-driver.advanced.continuous-paging.max-pages number
The maximum number of pages to return. The default of zero means retrieve all pages.
Default: 0

--driver.advanced.continuous-paging.max-pages-per-second, --datastax-java-driver.advanced.continuous-paging.max-pages-per-second number
Sets the maximum number of pages per second. The default of zero means no limit.
Default: 0

--driver.advanced.continuous-paging.max-enqueued-pages, --datastax-java-driver.advanced.continuous-paging.max-enqueued-pages number
The maximum number of pages that can be stored in the local queue. This value must be positive.
Default: 4

--driver.advanced.continuous-paging.timeout.first-page, --datastax-java-driver.advanced.continuous-paging.timeout.first-page "string"
How long to wait for the DataStax Bulk Loader coordinator to the first page.
Default: "60 seconds"

--driver.advanced.continuous-paging.timeout.other-pages, --datastax-java-driver.advanced.continuous-paging.timeout.other-pages "string"
DataStax Bulk Loader reference

How long to wait for the DataStax Bulk Loader coordinator to send subsequent pages.
Default: "120 seconds"

Advanced options
Specify advanced options for using dsbulk with the DataStax Java driver.

--driver.advanced.protocol.version, --datastax-java-driver.advanced.protocol.version string
The native protocol version to use. If not set, the DataStax Java driver looks up the versions of the nodes at startup (by default, system.peers.release_version) and chooses the highest common protocol version.

For example, if you have a mixed cluster with Apache Cassandra 2.1 nodes (protocol v3) and Apache Cassandra 3.0 nodes (protocol v3 and v4), the driver chooses protocol v3. If the nodes do not have a common protocol version, initialization fails. If this option is set, the given version is used for all connections without any negotiation or downgrading. If any of the contact points do not support the protocol version, that contact point is skipped. Once the protocol version is set, it cannot change for the duration of the driver's session. If an incompatible node joins the cluster later, the connection will fail and the driver will not try to reconnect to the node.
Default: null

--driver.advanced.protocol.compression, --datastax-java-driver.advanced.protocol.compression string
The name of the algorithm used to compress protocol frames. Possible values are: lz4, snappy or none. Default: none

--driver.advanced.connection.pool.local.size, --datastax-java-driver.advanced.connection.pool.local.size number
The number of connections in the pool for nodes considered as local.
Default: 8

--driver.advanced.connection.pool.remote.size, --datastax-java-driver.advanced.connection.pool.remote.size number
The number of connections in the pool for nodes considered as remote. The default load balancing policy used by DataStax Bulk Loader does not consider remote nodes. As a result, this setting has no effect when using the default load balancing policy.
Default: 8

--driver.advanced.connection.max-requests-per-connection, --datastax-java-driver.advanced.connection.max-requests-per-connection number
The maximum number of requests that can be executed concurrently on a connection. Applies to local or remote connections. Must be a number between 1 and 32768.
Default: 32768

--driver.advanced.resolve-contact-points, --datastax-java-driver.advanced.resolve-contact-points {true | false}
Whether to resolve the addresses passed to basic.contact-points.

- If true, addresses are created with InetSocketAddress(String, int). The host name is resolved the first time, and the driver will use the resolved IP address for all subsequent connection attempts.

- If false, addresses are created with InetSocketAddress.createUnresolved(). The host name will be resolved again every time the driver opens a new connection. This is useful for containerized environments where DNS records are more likely to change over time.

  JVM and OS have their own DNS caching mechanisms, so you might need additional configuration beyond the driver.

This option only applies to the contact points specified in the configuration. It has no effect on dynamically discovered peers. The driver relies on Cassandra system tables, which expose raw IP addresses. Use a custom address translator (see advanced.address-translator.class) to convert them to unresolved addresses; if you're in a containerized environment, you probably already need address translation.
Default: true
--driver.advanced.address-translator.class, --datastax-java-driver.advanced.address-translator.class
"string"

The class of the microsecond timestamp generator. If it is not qualified, the driver assumes that it resides in the package com.datastax.oss.driver.internal.core.time. The driver provides the following implementations out of the box:

- AtomicTimestampGenerator: timestamps are guaranteed to be unique across all client threads.
- ThreadLocalTimestampGenerator: timestamps that are guaranteed to be unique within each thread only.
- ServerSideTimestampGenerator: do not generate timestamps, let the server assign them.

You can also specify a custom class that implements TimestampGenerator and has a public constructor with two arguments: the DriverContext and a String representing the profile name.

Default: "AtomicTimestampGenerator"

--driver.advanced.timestamp-generator.class, --datastax-java-driver.advanced.timestamp-generator.class
"string"

The class of the translator. If not qualified, the DataStax Java driver assumes that it resides in the package com.datastax.oss.driver.internal.core.addresstranslation. The DataStax Java driver driver provides the PassThroughAddressTranslator implementation, which returns all addresses unchanged.

You can also specify a custom class that implements AddressTranslator and has a public constructor with a DriverContext argument.

Default: "PassThroughAddressTranslator"

--driver.advanced.heartbeat.interval, --datastax-java-driver.advanced.heartbeat.interval
"string"

The heartbeat interval. If a connection stays idle for that duration (there are no reads), the DataStax Java driver sends a dummy message on it to make sure it's still alive. If not, the connection is closed and replaced.

Default: "30 seconds"

--driver.advanced.heartbeat.timeout, --datastax-java-driver.advanced.heartbeat.timeout
"string"

How long the DataStax Java driver waits for the response to a heartbeat. If this timeout occurs, the heartbeat is considered failed.

Default: "60 seconds"

Deprecated options

--driver.socket.readTimeout string

Deprecated. The correct option to use is --datastax-java-driver.basic.request.timeout.

--driver.addressTranslator string

Deprecated. The correct option to use is --datastax-java-driver.advanced.address-translator.class.

--driver.timestampGenerator { AtomicMonotonicTimestampGenerator | ThreadLocalTimestampGenerator | ServerSideTimestampGenerator }

Deprecated. The correct option to use is --datastax-java-driver.advanced.timestamp-generator.class.

-lbp, --driver.policy.lbp.name { dse | dcAwareRoundRobin | roundRobin | whiteList | tokenAware }

Deprecated. The correct option to use is --datastax-java-driver.basic.load-balancing-policy.class.

--driver.policy.lbp.dcAwareRoundRobin.allowRemoteDCsForLocalConsistencyLevel {true | false}

Deprecated. There is no equivalent for this obsolete option.

--driver.policy.lbp.dcAwareRoundRobin.localDc string

Deprecated. The correct option to use is --datastax-java-driver.basic.load-balancing-policy.local-datacenter.

--driver.policy.lbp.dcAwareRoundRobin.usedHostsPerRemoteDc number

Deprecated. There is no equivalent for this obsolete option.

--driver.policy.lbp.dse.childPolicy { dse | dcAwareRoundRobin | roundRobin | whiteList | tokenAware }

Deprecated. There is no equivalent for this obsolete option.
Deprecated. There is no equivalent for this obsolete option.

`--driver.policy.lbp.tokenAware.childPolicy { dse | dcAwareRoundRobin | roundRobin | whiteList | tokenAware }`

Deprecated. There is no equivalent for this obsolete option.

`--driver.policy.lbp.tokenAware.shuffleReplicas { true | false }

Deprecated. There is no equivalent for this obsolete option.

`--driver.policy.lbp.whiteList.childPolicy { dse | dcAwareRoundRobin | roundRobin | whiteList | tokenAware }

Deprecated. There is no equivalent for this obsolete option.

`--driver.policy.lbp.whiteList.hosts string`

Deprecated. The correct option to use is `--datastax-java-driver.basic.load-balancing-policy.filter.class`.

`--datastax-java-driver.pooling.heartbeat string`

Deprecated. The correct option to use is `--datastax-java-driver.advanced.heartbeat.interval`.

`--driver.pooling.local.connections number`

Deprecated. The correct option to use is `--datastax-java-driver.advanced.connection.pool.local.size`.

`--driver.pooling.remote.connections number`

Deprecated. The correct option to use is `--datastax-java-driver.advanced.connection.pool.remote.size`.

`--datastax-java-driver.pooling.local.requests number`

Deprecated. The correct option to use is `--datastax-java-driver.advanced.connection.max-requests-per-connection`.

`--driver.pooling.remote.requests number`

Deprecated. The correct option to use is `--datastax-java-driver.advanced.connection.max-requests-per-connection`.

`--driver.protocol.compression string`

Deprecated. The correct option to use is `--datastax-java-driver.advanced.protocol.compression`.

`--driver.query.idempotence {true | false}`

Deprecated. The correct option to use is `--datastax-java-driver.basic.request.default-idempotence`.

`--driver.query.serialConsistency string`

Deprecated. The correct option to use is `--datastax-java-driver.basic.request.serial-consistency`.

`-u,--driver.auth.username string`

Deprecated. The correct option to use is `--datastax-java-driver.advanced.auth-provider.username`.

`-p,--driver.auth.password string`

Deprecated. The correct option to use is `--datastax-java-driver.advanced.auth-provider.password`.

`-maxRetries,--driver.policy.maxRetries number`

Deprecated. The correct option to use is `--datastax-java-driver.advanced.retry-policy.max-retries`.

`--driver.ssl.cipherSuites list`

Deprecated. The correct option to use is `--datastax-java-driver.advanced.ssl-engine-factory.class` and related `datastax-java-driver.advanced.ssl-engine-factory.*` options.

`--driver.ssl.keystore.algorithm { SunX509 | NewSunX509 }

Deprecated. The correct option to use is `--datastax-java-driver.advanced.ssl-engine-factory.class` and related `datastax-java-driver.advanced.ssl-engine-factory.*` options.

`--driver.ssh.key.password string`

Deprecated. The correct option to use is `--datastax-java-driver.advanced.ssl-engine-factory.keystore-password`. 
Engine options

Specify engine options for the dsbulk command.

The options can be used in short form (-k keyspace_name) or long form (--schema.keyspace keyspace_name).

--driver.ssl.keystore.path string
   Deprecated. The correct option to use is --datastax-java-driver.advanced.ssl-engine-factory.keystore-path.

--driver.ssl.openssl.keyCertChain string
   Deprecated. The correct option to use is --datastax-java-driver.advanced.ssl-engine-factory.class and related datastax-java-driver.advanced.ssl-engine-factory.* options.

--driver.ssl.openssl.privateKey string
   Deprecated. The correct option to use is --datastax-java-driver.advanced.ssl-engine-factory.class and related datastax-java-driver.advanced.ssl-engine-factory.* options.

--driver.ssl.provider { None | JDK | OpenSSL }
   Deprecated. The correct option to use is --datastax-java-driver.advanced.ssl-engine-factory.class and related datastax-java-driver.advanced.ssl-engine-factory.* options.

--driver.ssl.truststore.algorithm { PKIX | SunX509 }
   Deprecated. The correct option to use is --datastax-java-driver.advanced.ssl-engine-factory.class and related datastax-java-driver.advanced.ssl-engine-factory.* options.

--driver.ssl.truststore.password string
   Deprecated. The correct option to use is --datastax-java-driver.advanced.ssl-engine-factory.truststore-password.

--driver.ssl.truststore.path string
   Deprecated. The correct option to use is --datastax-java-driver.advanced.ssl-engine-factory.truststore-path.

--driver.query.fetchSize number
   Deprecated. The correct option to use is --datastax-java-driver.basic.request.page-size.

-clip--driver.query.consistency { ANY | LOCAL_ONE | ONE | TWO | THREE | LOCAL_QUORUM | QUORUM | EACH_QUORUM | ALL }
   Deprecated. The correct option to use is --datastax-java-driver.basic.request.consistency.

--driver.auth.provider { None | PlainTextAuthProvider | DsePlainTextAuthProvider | DSEGSSAPIAuthProvider }
   Deprecated. The correct option to use is --datastax-java-driver.advanced.auth-provider.class.

--driver.auth.authorizationId string
   Deprecated. The correct option to use is --datastax-java-driver.advanced.auth-provider.authorization-id.

--driver.auth.keyTab string
   Deprecated. The correct options to use are --datastax-java-driver.advanced.auth-provider.class and related --datastax-java-driver.advanced.auth-provider.* settings.

--driver.auth.principal email
   Deprecated. The correct options to use are --datastax-java-driver.advanced.auth-provider.class and related --datastax-java-driver.advanced.auth-provider.* settings.

--driver.auth.saslService string
   Deprecated. The correct options to use are --datastax-java-driver.advanced.auth-provider.class and related --datastax-java-driver.advanced.auth-provider.* settings.
DataStax Bulk Loader reference

-dryRun,--engine.dryRun { true | false }
Enable or disable dry-run mode, a test mode that runs the command but does not load data. Not applicable for unloading.
Default: false

--engine.executionId string
A unique identifier to attribute to each execution. When unspecified or empty, the engine will automatically generate identifiers of the following form: workflow_timestamp, where:

- workflow stands for the workflow type (LOAD, UNLOAD, etc.);
- timestamp is the current timestamp formatted as uuuuMMdd-HHmmss-SSSSSS (see Patterns for Formatting and Parsing in Oracle Java documentation) in UTC, with microsecond precision if available, and millisecond precision otherwise.

When this identifier is user-supplied, it is important to guarantee its uniqueness; failing to do so may result in execution failures. It is also possible to provide templates here. Any format compliant with the formatting rules of String.format() is accepted, and can contain the following parameters:

- %1$s: the workflow type (LOAD, UNLOAD, etc.);
- %2$s: the current time (with microsecond precision if available, and millisecond precision otherwise);
- %3$s: the JVM process PID (this parameter might not be available on some operating systems; if its value cannot be determined, a random integer will be inserted instead).

Default: unspecified

Executor options
Specify executor options for the dsbulk command. These options define how DataStax Bulk Loader executes queries, and allow users to control throughput, throttling, concurrency, and query method execution.

--executor.maxPerSecond, --dsbulk.executor.maxPerSecond number
The maximum number of concurrent operations per second. This acts as a safeguard to prevent more requests than the cluster can handle. Batch statements are counted by the number of statements included. Reduce this setting when the latencies get too high and a remote cluster cannot keep up with throughput, as dsbulk requests will eventually time out. This setting applies to all operations: when writing to the database, it applies to the number of statements executed; when reading from the database, it applies to the number of rows retrieved. Setting this option to any negative value will disable it.
Default: -1

--executor.maxInFlight, --dsbulk.executor.maxInFlight number
The maximum number of "in-flight" requests, or maximum number of concurrent requests waiting for a response from the server. This acts as a safeguard to prevent more requests than the cluster can handle. Batch statements count as one request. Reduce this value if you are facing out-of-memory errors. Setting this option to any negative value will disable it.
Default: 1024

--executor.continuousPaging.enabled, --dsbulk.executor.continuousPaging.enabled {true | false}
Enable or disable continuous paging. If the target cluster does not support continuous paging, or if driver.query.consistency is not ONE or LOCAL_ONE, traditional paging is used regardless of this setting. Used for unloading only.
Default: true

--executor.continuousPaging.maxConcurrentQueries, --dsbulk.executor.continuousPaging.maxConcurrentQueries number
The maximum number of concurrent continuous paging queries that should be carried in parallel. Set this number to a value equal to, or lesser than, the value configured server-side for
continuous_paging.max_concurrent_sessions in the cassandra.yaml configuration file. If not set as noted above, some requests may be rejected. Setting this option to any negative value or zero will disable it.

Default: 60

--executor.continuousPaging.maxPages, --dsbulk.executor.continuousPaging.maxPages \( \text{number} \)

Deprecated. The correct option to use is \(--\text{datastax-java-driver.advanced.continuous-paging.max-pages}\).

--executor.continuousPaging.maxPagesPerSecond, --dsbulk.executor.continuousPaging.maxPagesPerSecond \( \text{number} \)

Deprecated. The correct option to use is \(--\text{datastax-java-driver.advanced.continuous-paging.max-pages-per-second}\).

--executor.continuousPaging.pageSize, --dsbulk.executor.continuousPaging.pageSize \( \text{number} \)

Deprecated. The correct option to use is \(--\text{datastax-java-driver.advanced.continuous-paging.page-size}\).

--executor.continuousPaging.pageUnit, --dsbulk.executor.continuousPaging.pageUnit \{ROWS | BYTES\}

Deprecated. The correct option to use is \(--\text{datastax-java-driver.advanced.continuous-paging.page-size-in-bytes}\).

Logging options

Specify logging and error options for the dsbulk command. Log messages are only logged to the main log file, operation/log, and standard error, and nothing is printed to stdout.

The options can be used in short form (\(-k \text{keyspace}_\text{name}\)) or long form (\(--\text{schema.keyspace keyspace}_\text{name}\)).

-ansiMode, --log.ansiMode \{normal | force | disable\}

Whether or not to use ANSI colors and other escape sequences in log messages printed to the console. By default, dsbulk uses colored output (normal) when the terminal is: (1) compatible with ANSI escape sequences; all common terminals on "nix and BSD systems, including MacOS, are ANSI-compatible, and some popular terminals for Windows (Mintty, MinGW) or (2) a standard Windows DOS command prompt (ANSI sequences are translated on the fly). The force value will cause dsbulk to use ANSI colors even for non ANSI-compatible terminals detected. There should be no reason to disable ANSI escape sequences, but if, for some reason, colored messages are not desired or not printed correctly, this option allows disabling ANSI support altogether. For Windows: ANSI support works best with the (Microsoft Visual C++ 2008 SP1 Redistributable Package) installed.

Default: normal

-maxErrors, --log.maxErrors \{number | "N%"\}

The maximum number of errors to tolerate before aborting the entire operation. Set to either a number or a string of the form \(N\%\) where \(N\) is a decimal number between 0 and 100. Setting this value to \(-1\) disables this feature (not recommended).

Default: 10

-logDir, --log.directory \path_to_directory\n
The writable directory where all log files will be stored; if the directory specified does not exist, it will be created. URLs are not acceptable (not even file:// URLs). Log files for a specific run, or execution, will be located in a sub-directory under the specified directory. Each execution generates a sub-directory identified by an "execution ID". See engine.executionId for more information about execution IDs. Relative paths will be resolved against the current working directory. Also, for convenience, if the path begins with a tilde (\~), that symbol will be expanded to the current user’s home directory.

Default: ./logs

-log.stmt.level \{ ABRIDGED | NORMAL | EXTENDED \}

The desired log level for printing to log files. Valid values are:
• **ABRIDGED**: Print only basic information in summarized form.

• **NORMAL**: Print basic information in summarized form, and the statement's query string, if available. For batch statements, this verbosity level also prints information about the batch's inner statements.

• **EXTENDED**: Print full information, including the statement's query string, if available, and the statement's bound values, if available. For batch statements, this verbosity level also prints all information available about the batch's inner statements.

**Default**: EXTENDED

--log.stmt.maxBoundValueLength number
The maximum length for a bound value. Bound values longer than this value will be truncated.

Setting this value to -1 disables this feature (not recommended).

**Default**: 50

--log.stmt.maxBoundValues number
The maximum number of bound values to print. If the statement has more bound values than this limit, the exceeding values will not be printed.

Setting this value to -1 disables this feature (not recommended).

**Default**: 50

--log.stmt.maxInnerStatements number
The maximum number of inner statements to print for a batch statement. Only applicable for batch statements, ignored otherwise. If the batch statement has more children than this value, the exceeding child statements will not be printed.

Setting this value to -1 disables this feature (not recommended).

**Default**: 10

--log.stmt.maxQueryStringLength number
The maximum length for a query string. Query strings longer than this value will be truncated.

Setting this value to -1 disables this feature (not recommended).

**Default**: 500

-verbosity, --log.verbosity { 0 | 1 | 2 }
Desired level of verbosity. Valid values are:

• **0** (quiet): Only log `WARN` and `ERROR` messages.

• **1** (normal): Log `INFO`, `WARN` and `ERROR` messages.

• **2** (verbose): Log `DEBUG`, `INFO`, `WARN` and `ERROR` messages.

**Default**: 1

**Monitoring options**

Specify monitoring options for the `dsbulk` command.

Monitored throughput is often measured as operations per second, where an operation is a single write event or a single read event. But this unit of measurement can vary greatly, depending on the size of the row being written or read. DataStax recommends that when you monitor your data, consider using mb/sec as a different measure of throughput, to avoid the irregularity of measuring throughout as operations/sec.

In a load work flow, a typical report shows:

```
```
The options can be used in short form (-k keyspace_name) or long form (--schema.keyspace keyspace_name).

-**reportRate**, **--monitoring.reportRate string**
  The report interval for the console reporter. The console reporter will print useful metrics about the ongoing operation at this rate. Durations lesser than one second will be rounded up to 1 second. Default: 5 seconds

-**monitoring.csv { true | false }**
  Enable or disable CSV reporting. If enabled, CSV files containing metrics will be generated in the designated log directory. Default: false

-**monitoring.durationUnit string**
  The time unit used when printing latency durations. Valid values: all TimeUnit enum constants. Default: MILLISECONDS

-**monitoring.expectedReads number**
  The expected total number of reads. Optional, but if set, the console reporter will also print the overall achievement percentage. Setting this value to -1 disables this feature. Default: -1

-**monitoring.expectedWrites number**
  The expected total number of writes. Optional, but if set, the console reporter will also print the overall achievement percentage. Setting this value to -1 disables this feature. Default: -1

-**jmx, --monitoring.jmx { true | false }**
  Enable or disable JMX reporting. Note that to enable remote JMX reporting, several properties must also be set in the JVM during launch. This is accomplished via the DSBulk_JAVA_OPTS environment variable. Default: true

-**monitoring.rateUnit string**
  The time unit used when printing throughput rates. Valid values: all TimeUnit enum constants. Default: SECONDS