# CQL for Cassandra 2.0 & 2.1

## Documentation

March 8, 2016

# Contents

# Introduction to Cassandra Query Language

The Cassandra Query Language (CQL) is the primary language for communicating with the Cassandra database. The most basic way to interact with Cassandra is using the CQL shell, cqlsh. Using cqlsh, you can create keyspaces and tables, insert and query tables, plus much more. If you prefer a graphical tool, you can use DataStax DevCenter. For production, DataStax supplies a number drivers so that CQL statements can be passed from client to cluster and back. Other administrative tasks can be accomplished using OpsCenter.

**Important:** This document assumes you are familiar with either the Cassandra 2.1 or Cassandra 2.0 documentation.

## Cassandra 2.1 features

Cassandra 2.1 new CQL features include:

- Nested user-defined types
- Improved counter columns that maintain the correct count when Cassandra replays the commit log
- Configurable counter cache
- Support for indexes on collections, including using map keys to filter query results
- Timestamps precise to the millisecond
- The new tuple type that holds fixed-length sets of typed positional fields

The cqlsh utility also has been improved:

- Capability to accept and execute a CQL statement from the operating system command line
- Support for describing types using the DESCRIBE command

DataStax Java Driver 2.0.0 supports Cassandra 2.1 with limitations. This version of the driver is incompatible with the new features.

Other notable changes:

- Cassandra rejects USING TIMESTAMP or USING TTL in the command to update a counter column, and now generates an error message when you attempt such an operation.
- In Cassandra 2.1, the CQL table property index_interval is replaced by min_index_interval and max_index_interval. The max_index_interval is 2048 by default. The default would be reached only when SSTables are infrequently-read and the index summary memory pool is full. When upgrading from earlier releases, Cassandra uses the old index_interval value for the min_index_interval.

## Cassandra 2.0.x features

Cassandra 2.0.x key features are:

- Lightweight transactions using the IF keyword in INSERT and UPDATE statements.
- Prevention of application errors by performing conditional tests for the existence of a table, keyspace, or index.

  Simply include IF EXISTS or IF NOT EXISTS in DROP or CREATE statements, such as DROP KEYSPACE or CREATE TABLE.
- Initial support for triggers that fire events executed in or out of a database cluster.
- The ALTER TABLE DROP command, which had been removed in the earlier release.
- Column aliases, similar to aliases in RDBMS SQL, in a SELECT statement.
- Indexing of the part, partition key or clustering columns, portion of a compound primary key.

DataStax drivers support Cassandra 2.0.

CQL for Cassandra 2.0 deprecated super columns. Cassandra continues to support apps that query super columns, translating super columns on the fly into CQL constructs and results.

Some changes were made to the cqlsh commands in CQL Cassandra 2.0:

- The ASSUME command has been removed.

  Use the blobAsType and typeAsBlob conversion functions instead of ASSUME.
- The COPY command now supports for collections.

Several CQL table attributes were added in CQL included with Cassandra 2.0:

- default_time_to_live
- memtable_flush_period_in_ms
- populate_io_cache_on_flush
- speculative_retry

# CQL data modeling

**Note:** DataStax Academy provides a course in Cassandra data modeling. This course presents techniques using the Chebotko method for translating a real-world domain model into a running Cassandra schema.

## Data Modeling Concepts

For a very basic explanation of CQL, see the Data model distilled in *Cassandra & DataStax Enterprise Essentials*.

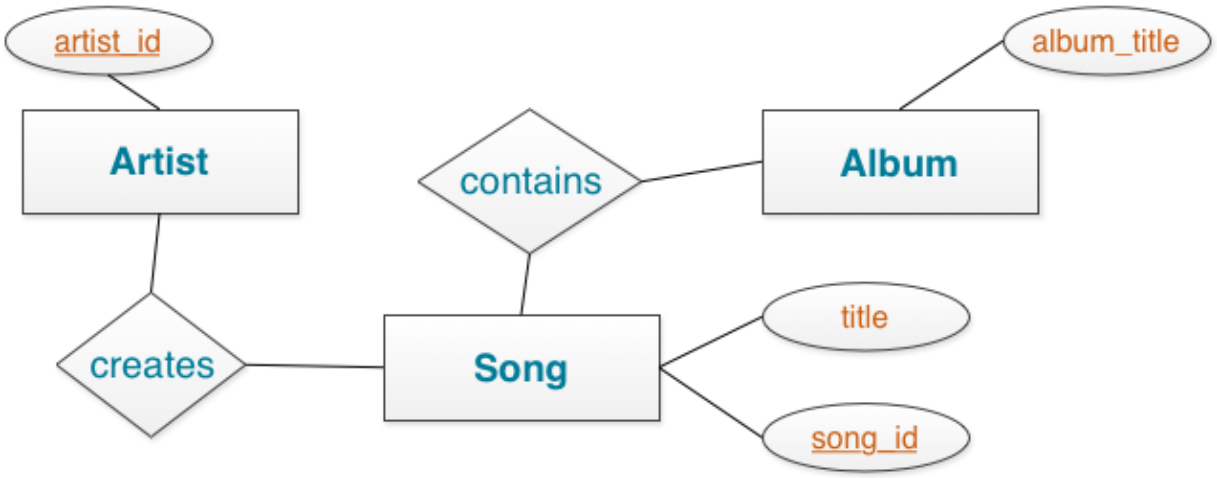
**Note:** DataStax Academy provides a course in Cassandra data modeling. This course presents techniques using the Chebotko method for translating a real-world domain model into a running Cassandra schema.

Data modeling is a process that involves identifying the entities, or items to be stored, and the relationships between entities. In addition, data modeling involves the identification of the patterns of data access and the queries that will be performed. These two ideas inform the organization and structure of how storing the data, and the design and creation of the database's tables. In some cases, indexing the data improves the performance, so judicious choices about secondary indexing must be considered.

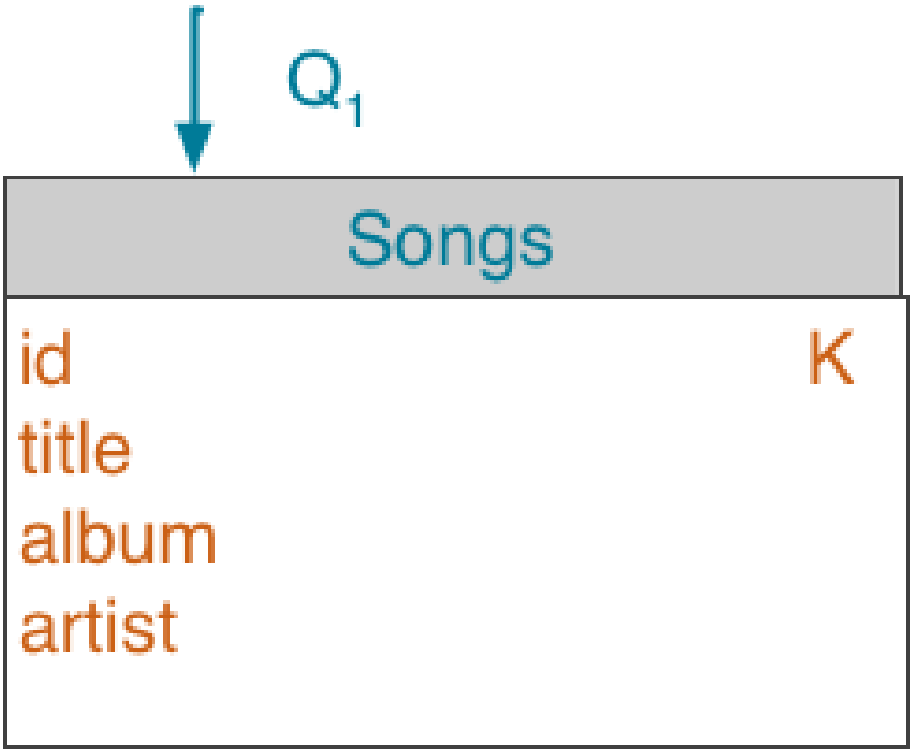
Data modeling in Cassandra uses a query-driven approach, in which specific queries are the key to organizing the data. Cassandra's database design is based on the requirement for fast reads and writes, so the better the schema design, the faster data is written and retrieved. Queries are the result of selecting data from a table; schema is the definition of how data in the table is arranged.

Cassandra's data model is a partitioned row store with tunable consistency. Rows are organized into tables; the first component of a table's primary key is the partition key; within a partition, rows are clustered by the remaining columns of the key. Other columns can be indexed separately from the primary key. Because Cassandra is a distributed database, efficiency is gained for reads and writes when data is grouped together on nodes by partition. The fewer partitions that must be queried to get an answer to a question, the faster the response. Tuning the consistency level is another factor in latency, but is not part of the data modeling process.

For this reason, Cassandra data modeling focuses on the queries. Throughout this topic, the music service example demonstrates the schema that results from modeling the Cassandra tables for specific queries.



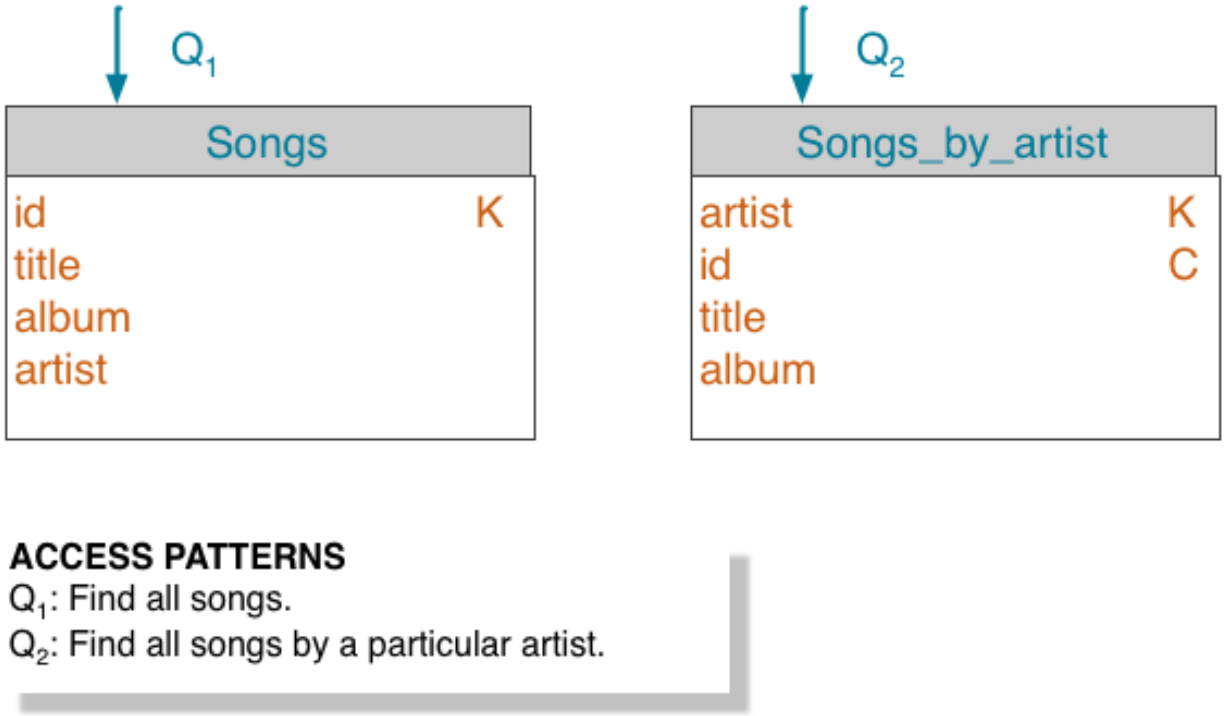

One basic query for a music service is a listing of songs, including the title, album, and artist. To uniquely identify a song in the table, an additional column id is added. For a simple query to list all songs, a table that includes all the columns identified and a partition key (K) of id is created.




**ACCESS PATTERNS**

$Q_1$: Find all songs.

A related query searches for all songs by a particular artist. For Cassandra, this query is more efficient if a table is created that groups all songs by artist. All the same columns of title, album, and artist are required, but now the primary key of the table includes the artist as the partition key (K) and groups within the partition by the id (C). This ensures that unique records for each song are created.




Notice that the key to designing the table is not the relationship of the table to other tables, as it is in relational database modeling. Data in Cassandra is often arranged as one query per table, and data is repeated amongst many tables, a process known as denormalization. The relationship of the entities is important, because the order in which data is stored in Cassandra can greatly affect the ease and speed of data retrieval.

## Data modeling example

Cassandra's data model is a partitioned row store with tunable consistency. Rows are organized into tables; the first component of a table's primary key is the partition key; within a partition, rows are clustered by the remaining columns of the key. Other columns can be indexed separately from the primary key. Tables can be created, dropped, and altered at runtime without blocking updates and queries.

The example of a music service shows the how to use compound keys, clustering columns, and collections to model Cassandra data.

**Note:** DataStax Academy provides a course in Cassandra data modeling. This course presents techniques using the Chebotko method for translating a real-world domain model into a running Cassandra schema.

## Example of a music service

This example of a social music service requires a songs table having a title, album, and artist column, plus a column (called data) for the actual audio file itself. The table uses a UUID as a primary key.

```
CREATE TABLE songs (
  id uuid PRIMARY KEY,
  title text,
  album text,
  artist text,
  data blob
```

```
  );
```

In a relational database, you would create a playlists table with a foreign key to the songs, but in Cassandra, you denormalize the data because joins are not performant in a distributed system. Later, this document covers how to use a collection to accomplish the same goal as joining the tables to tag songs. To represent the playlist data, you can create a table like this:

```
CREATE TABLE playlists (
  id uuid,
  song_order int,
  song_id uuid,
  title text,
  album text,
  artist text,
  PRIMARY KEY  (id, song_order ) );
```

The combination of the id and song_order in the playlists table uniquely identifies a row in the playlists table. You can have more than one row with the same id as long as the rows contain different song_order values.

**Note:** The UUID is handy for sequencing the data or automatically incrementing synchronization across multiple machines. For simplicity, an int song_order is used in this example.

Here's an example of inserting a single record into the playlist: inserting the example data into playlists

```
INSERT INTO playlists (id, song_order, song_id, title, artist, album)
  VALUES (62c36092-82a1-3a00-93d1-46196ee77204, 4,
  7db1a490-5878-11e2-bcfd-0800200c9a66,
  'Ojo Rojo', 'Fu Manchu', 'No One Rides for Free');
```

After inserting the remaining records, use the following SELECT query to display the table's data:

```
SELECT * FROM playlists;
```

```
id          | song_order | album        | artist         | song_id     | title
------------+------------+--------------+----------------+-------------+---------------------
62c36092... |          1 | Tres Hombres |         ZZ Top | a3e63f8f... |           La Grange
62c36092... |          2 | We Must Obey |      Fu Manchu | 8a172618... |    Moving in Stereo
62c36092... |          3 |    Roll Away | Back Door Slam | 2b09185b... | Outside Woman Blues
```

The example below illustrates how to create a query that uses the artist as a filter:

```
SELECT album, title FROM playlists WHERE artist = 'Fu Manchu';
```

Cassandra will reject this query because the query requires a sequential scan across the entire playlists dataset, because artist is not a partition key or clustering column. By creating an index on artist, Cassandra can now pull out the records.

```
CREATE INDEX ON playlists( artist );
```

Now, you can query the playlists for songs by Fu Manchu. The output looks like this:

```
album                      | title
---------------------------+-------------------
We Must Obey               | Moving in Stereo
No One Riddes for Free     | Ojo Rojo
```

## Compound keys and clustering

A compound primary key consists of the partition key and one or more additional columns that determine clustering. The partition key determines which node stores the data. It is responsible for data distribution across the nodes. The additional columns determine per-partition clustering. Clustering is a storage engine process that sorts data within the partition.

In a simple primary key, Cassandra uses the first column name as the partition key. (Note that Cassandra can use multiple columns in the definition of a partition key.) In the music service's playlists table, id is the partition key. The remaining columns can be defined as clustering columns. In the playlists table below, the song_order is defined as the clustering column column:

```
PRIMARYKEY (id, song_order));
```

The data for each partition is clustered by the remaining column or columns of the primary key definition. On a physical node, when rows for a partition key are stored in order based on the clustering columns, retrieval of rows is very efficient. For example, because the id in the playlists table is the partition key, all the songs for a playlist are clustered in the order of the remaining song_order column. The others columns are displayed in alphabetical order by Cassandra.

Insertion, update, and deletion operations on rows sharing the same partition key for a table are performed atomically and in isolation.

You can query a single sequential set of data on disk to get the songs for a playlist.

```
SELECT * FROM playlists WHERE id = 62c36092-82a1-3a00-93d1-46196ee77204
  ORDER BY song_order DESC LIMIT 50;
```

The output looks something like this:

```
id          | song_order | album                 | artist         | song_id     | title
------------+------------+-----------------------+----------------+-------------+---------------------
62c36092... |          4 | No One Rides for Free |      Fu Manchu | 7db1a490... |            Ojo Rojo
62c36092... |          3 |             Roll Away | Back Door Slam | 2b09185b... | Outside Woman Blues
62c36092... |          2 |          We Must Obey |      Fu Manchu | 8a172618... |    Moving in Stereo
62c36092... |          1 |          Tres Hombres |         ZZ Top | a3e63f8f... |           La Grange
```

Cassandra stores an entire row of data on a node by partition key. If you have too much data in a partition and want to spread the data over multiple nodes, use a composite partition key.

## Collection columns

CQL contains these collection types:

- set
- list
- map

In a relational database, to allow users to have multiple email addresses, you create an email_addresses table having a many-to-one (joined) relationship to a users table. CQL handles the classic multiple email

addresses use case, and other use cases, by defining columns as collections. Using the set collection type to solve the multiple email addresses problem is convenient and intuitive.

Another use of a collection type can be demonstrated using the music service example. Also see Using frozen in a collection.

## Adding a collection to a table

The music service example includes the capability to tag the songs. From a relational standpoint, you can think of storage engine rows as partitions, within which (object) rows are clustered. To tag songs, use a collection set. Declare the collection set using the CREATE TABLE or ALTER TABLE statements. Because the playlists table already exists from the earlier example, just alter that table to add a collection set, tags:

```
ALTER TABLE playlists ADD tags set<text>;
```

## Updating a collection

Update the playlists table to insert the tags data:

```
UPDATE playlists SET tags = tags + {'2007'}
  WHERE id = 62c36092-82a1-3a00-93d1-46196ee77204 AND song_order = 2;
UPDATE playlists SET tags = tags + {'covers'}
  WHERE id = 62c36092-82a1-3a00-93d1-46196ee77204 AND song_order = 2;
UPDATE playlists SET tags = tags + {'1973'}
  WHERE id = 62c36092-82a1-3a00-93d1-46196ee77204 AND song_order = 1;
UPDATE playlists SET tags = tags + {'blues'}
  WHERE id = 62c36092-82a1-3a00-93d1-46196ee77204 AND song_order = 1;
UPDATE playlists SET tags = tags + {'rock'}
  WHERE id = 62c36092-82a1-3a00-93d1-46196ee77204 AND song_order = 4;
```

A music reviews list and a schedule (map collection) of live appearances can be added to the table:

```
ALTER TABLE playlists ADD reviews list<text>;
ALTER TABLE playlists ADD venue map<timestamp, text>;
```

Each element of a set, list, or map is internally stored as one Cassandra column. To update a set, use the UPDATE command and the addition (+) operator to add an element or the subtraction (-) operator to remove an element. For example, to update a set:

```
UPDATE playlists
  SET tags = tags + {'punk rock'}
  WHERE id = 62c36092-82a1-3a00-93d1-46196ee77204 AND song_order = 4;
```

To update a list, a similar syntax using square brackets instead of curly brackets is used.

```
UPDATE playlists
  SET reviews = reviews + [ 'best lyrics' ]
  WHERE id = 62c36092-82a1-3a00-93d1-46196ee77204 and song_order = 4;
```

To update a map, use INSERT to specify the data in a map collection.

```
INSERT INTO playlists (id, song_order, venue)
  VALUES (62c36092-82a1-3a00-93d1-46196ee77204, 4,
  { '2013-9-22 22:00'  : 'The Fillmore',
  '2013-10-1 21:00' : 'The Apple Barrel'});

INSERT INTO playlists (id, song_order, venue)
  VALUES (62c36092-82a1-3a00-93d1-46196ee77204, 3,
  { '2014-1-22 22:00'  : 'Cactus Cafe',
```

```
   '2014-01-12 20:00' : 'Mohawk'});
```

Inserting data into the map replaces the entire map.

Selecting all the data from the playlists table at this point gives you output something like this:

```
id          | song_order | album                  | artist         | reviews          | song_id
------------+------------+------------------------+----------------+------------------+-------------
62c36092... |          1 |           Tres Hombres |         ZZ Top |             null | a3e63f8f...
62c36092... |          2 |           We Must Obey |      Fu Manchu |             null | 8a172618...
62c36092... |          3 |              Roll Away | Back Door Slam |             null | 2b09185b...
62c36092... |          4 | No One Riddes for Free |      Fu Manchu | ["best lyrics"] | 7db1a490...
```

```
song_id     | tags                    | title               | venue
------------+-------------------------+---------------------+-----------------------------------
a3e63f8f... |     {"1973", "blues"} |           La Grange |                              null
8a172618... |    {"2007", "covers"} |    Moving in Stereo |                              null
2b09185b... |                  null | Outside Woman Blues |      {2014-01-12..."Cactus Cafe"}
7db1a490... | {"punk rock", "rock"} |            Ojo Rojo | {2013-09-22..."The Apple Barrel"}
```

# Indexing a collection

In Cassandra 2.1 and later, you can index collections and query the database to find a collection containing a particular value. Continuing with the music service example, suppose you want to find songs tagged blues and that debuted at the Fillmore. Index the tags set and venue map. Query for values in the tags set and the venue map, as shown in the next section.

```
CREATE INDEX ON playlists (tags);
CREATE INDEX mymapvalues ON playlists (venue);
```

Specifying a name for the index, such as mymapindex, is optional.

## Indexing collection map keys

The last example created the index on the venue collection values by using the venue map column name to create the index. You can also create an index on map collection keys. A map key is the literal to the left of the colon in the JSON-style array. A map value is the literal to the right of the colon.

```
{ literal : literal, literal : literal ... }
```

For example, the collection keys in the venue map are the timestamps. The collection values in the venue map are 'The Fillmore' and 'Apple Barrel'.

```
artist     | venue
-----------------------------------------------------------------------------
Fu Manchu | {“2013-09-22 12:01:00-0700”: “The Fillmore”, “2013-10-01 1
```

Indexes on the keys and values of a map cannot co-exist. For example, if you created mymapindex, you would need to drop it to create an index on the map keys using the KEYS keyword and map name in nested parentheses:

```
DROP INDEX mymapvalues;
CREATE INDEX mymapkeys ON playlists (KEYS(venue));
```

# Filtering data in a collection

After adding data to the tags set collection, selecting the tags set returns the set of tags:

```
SELECT album, tags FROM playlists;
```

```
album                    | tags
-------------------------+-----------------------------
Tres Hombres             |      {"1973", "blues"}
We Must Obey             |     {"2007", "covers"}
Roll Away                |                       null
No One Rides for Free    | {"punk rock", "rock"}
```

Assuming you indexed the collection, to filter data using a set collection value, include the name of the collection column in the select expression. For example, find the row containing a particular tag, say "blues", using the CONTAINS condition in the WHERE clause.

```
SELECT album, tags FROM playlists WHERE tags CONTAINS 'blues';
```

The output is the row from the playlists table containing the blues tag.

```
album                    | tags
-------------------------+-----------------------------
Tres Hombres             |      {"1973", "blues"}
```

## Filtering by map value or map key

You can create two types of map collection indexes: an index of map values and an index of map keys. The two types cannot co-exist on the same collection. Assuming an index on map values is created, filter the data using a value in the venue map using the CONTAINS condition in the WHERE clause. The statement is the same one you use to filtering the data in a set or list:

```
SELECT artist, venue FROM playlists WHERE venue CONTAINS 'The Fillmore';
```

The output is the requested data for the song that debuted at The Fillmore.

Assuming an index on map keys is created, filter the data using a key in the venue map:

```
SELECT album, venue FROM playlists WHERE venue CONTAINS KEY '2013-09-22
 22:00:00-0700';
```

# When to use a collection

Use collections when you want to store or denormalize a small amount of data. Values of items in collections are limited to 64K. Other limitations also apply. Collections work well for storing data such as the phone numbers of a user and labels applied to an email. If the data you need to store has unbounded growth potential, such as all the messages sent by a user or events registered by a sensor, do not use collections. Instead, use a table having a compound primary key and store data in the clustering columns.

# Indexing

An index provides a means to access data in Cassandra using attributes other than the partition key. The benefit is fast, efficient lookup of data matching a given condition. The index indexes column values in a separate, hidden table from the one that contains the values being indexed. Cassandra has a number of techniques for guarding against the undesirable scenario where a data might be incorrectly retrieved during a query involving indexes on the basis of stale values in the index.

As mentioned earlier, in Cassandra 2.1 and later, you can index collection columns.

# When to use an index

Cassandra's built-in indexes are best on a table having many rows that contain the indexed value. The more unique values that exist in a particular column, the more overhead you will have, on average, to query and maintain the index. For example, suppose you had a playlists table with a billion songs and wanted to look up songs by the artist. Many songs will share the same column value for artist. The artist column is a good candidate for an index.

## When *not* to use an index

Do not use an index in these situations:

- On high-cardinality columns because you then query a huge volume of records for a small number of results. See Problems using a high-cardinality column index below.
- In tables that use a counter column
- On a frequently updated or deleted column. See Problems using an index on a frequently updated or deleted column below.
- To look for a row in a large partition unless narrowly queried. See Problems using an index to look for a row in a large partition unless narrowly queried below.

**Problems using a high-cardinality column index**

If you create an index on a high-cardinality column, which has many distinct values, a query between the fields will incur many seeks for very few results. In the table with a billion songs, looking up songs by writer (a value that is typically unique for each song) instead of by their artist, is likely to be very inefficient. It would probably be more efficient to manually maintain the table as a form of an index instead of using the Cassandra built-in index. For columns containing unique data, it is sometimes fine performance-wise to use an index for convenience, as long as the query volume to the table having an indexed column is moderate and not under constant load.

Conversely, creating an index on an extremely low-cardinality column, such as a boolean column, does not make sense. Each value in the index becomes a single row in the index, resulting in a huge row for all the false values, for example. Indexing a multitude of indexed columns having foo = true and foo = false is not useful.

**Problems using an index on a frequently updated or deleted column**

Cassandra stores tombstones in the index until the tombstone limit reaches 100K cells. After exceeding the tombstone limit, the query that uses the indexed value will fail.

**Problems using an index to look for a row in a large partition unless narrowly queried**

A query on an indexed column in a large cluster typically requires collating responses from multiple data partitions. The query response slows down as more machines are added to the cluster. You can avoid a performance hit when looking for a row in a large partition by narrowing the search, as shown in the next section.

# Using an index

Using CQL, you can create an index on a column after defining a table. In Cassandra 2.1 and later, you can index a collection column. The music service example shows how to create an index on the artists column of playlist, and then query Cassandra for songs by a particular artist:

```
CREATE INDEX artist_names ON playlists( artist );
```

An index name is optional. If you do not provide a name, Cassandra assigns a name such as artist_idx. If you provide a name, such as artist_names, the name must be unique within the keyspace. After creating an index for the artist column and inserting values into the playlists table, greater efficiency is achieved when you query Cassandra directly for artist by name, such as Fu Manchu:

```
SELECT * FROM playlists WHERE artist = 'Fu Manchu';
```

As mentioned earlier, when looking for a row in a large partition, narrow the search. This query, although a contrived example using so little data, narrows the search to a single id.

```
SELECT * FROM playlists WHERE id = 62c36092-82a1-3a00-93d1-46196ee77204 AND
 artist = 'Fu Manchu';
```

The output is:

```
id          | song_order | album                 | artist    | song_id     | title
------------+------------+-----------------------+-----------+-------------+------------------
62c36092... |          2 |          We Must Obey | Fu Manchu | 8a172618... | Moving in Stereo
62c36092... |          4 | No One Rides for Free | Fu Manchu | 7db1a490... |         Ojo Rojo
```

# Using multiple indexes

For example, you can create multiple indexes, such as on album and title columns of the playlists table, and use multiple conditions in the WHERE clause to filter the results. In a real-world situation, these columns might not be good choices, depending on their cardinality as described later:

```
CREATE INDEX album_name ON playlists ( album );
```

```
CREATE INDEX title_name ON playlists ( title );
```

```
SELECT * FROM playlists
      WHERE album = 'Roll Away' AND title = 'Outside Woman Blues'
      ALLOW FILTERING ;
```

When multiple occurrences of data match a condition in a WHERE clause, Cassandra selects the least-frequent occurrence of a condition for processing first for efficiency. For example, suppose data for Blind Joe Reynolds and Cream's versions of "Outside Woman Blues" were inserted into the playlists table. Cassandra queries on the album name first if there are fewer albums named Roll Away than there are songs called "Outside Woman Blues" in the database. When you attempt a potentially expensive query, such as searching a range of rows, Cassandra requires the ALLOW FILTERING directive.

# Building and maintaining indexes

An advantage of indexes is the operational ease of populating and maintaining the index. Indexes are built in the background automatically, without blocking reads or writes. Client-maintained *tables as indexes*

must be created manually; for example, if the artists column had been indexed by creating a table such as songs_by_artist, your client application would have to populate the table with data from the songs table.

To perform a hot rebuild of an index, use the `nodetool rebuild_index` command.

# Working with legacy applications

Internally, CQL does not change the row and column mapping from the Thrift API mapping. CQL and Thrift use the same storage engine. CQL supports the same query-driven, denormalized data modeling principles as Thrift. Existing applications do not have to be upgraded to CQL. The CQL abstraction layer makes CQL easier to use for new applications. For an in-depth comparison of Thrift and CQL, see "A Thrift to CQL Upgrade Guide" and CQL for Cassandra experts.

## Creating a legacy table

You can create legacy (Thrift/CLI-compatible) tables in CQL using the COMPACT STORAGE directive. The compact storage directive used with the CREATE TABLE command provides backward compatibility with older Cassandra applications; new applications should generally avoid it.

Compact storage stores an entire row in a single column on disk instead of storing each non-primary key column in a column that corresponds to one column on disk. Using compact storage prevents you from adding new columns that are not part of the PRIMARY KEY.

# Using a CQL query

Using CQL, you can query a legacy table. A legacy table managed in CQL includes an implicit WITH COMPACT STORAGE directive. When you use CQL to query legacy tables with no column names defined for data within a partition, Cassandra generates the names (column1 and value1) for the data. Using the RENAME clause, you can change the default column name to a more meaningful name.

```
ALTER TABLE users RENAME userid to user_id;
```

CQL supports dynamic tables created in the Thrift API, CLI, and earlier CQL versions. For example, a dynamic table is represented and queried like this:

```
CREATE TABLE clicks (
  userid uuid,
  url text,
  timestamp date
  PRIMARY KEY  (userid, url ) ) WITH COMPACT STORAGE;

SELECT url, timestamp
  FROM clicks
  WHERE  userid = 148e9150-1dd2-11b2-0000-242d50cf1fff;

SELECT timestamp
  FROM clicks
  WHERE  userid = 148e9150-1dd2-11b2-0000-242d50cf1fff
  AND  url = 'http://google.com';
```

In these queries, only equality conditions are valid.

# Using CQL

CQL provides an API to Cassandra that is simpler than the Thrift API for new applications. The Thrift API and legacy versions of CQL expose the internal storage structure of Cassandra. CQL adds an abstraction layer that hides implementation details of this structure and provides native syntaxes for collections and other common encodings.

### Accessing CQL

Common ways to access CQL are:

- Start cqlsh, the Python-based command-line client, on the command line of a Cassandra node.
- Use DataStax DevCenter, a graphical user interface.
- For developing applications, you can use one of the official DataStax C#, Java, or Python open-source drivers.
- Use the set_cql_version Thrift method for programmatic access.

This document presents examples using cqlsh.

## Starting cqlsh

## Starting cqlsh on Linux

This procedure briefly describes how to start cqlsh on Linux. The cqlsh command is covered in detail later.

### Procedure

1. Navigate to the Cassandra installation directory.
2. Start cqlsh on the Mac OSX, for example.

   ```
   bin/cqlsh
   ```

   If you use security features, provide a user name and password.
3. Optionally, specify the IP address and port to start cqlsh on a different node.

   ```
   bin/cqlsh 1.2.3.4 9042
   ```

## Starting cqlsh on Windows

This procedure briefly describes how to start cqlsh on Windows. The cqlsh command is covered in detail later.

### Procedure

1. Open Command Prompt.
2. Navigate to the Cassandra bin directory.

3. Type the command to start cqlsh.

```
python cqlsh
```

Optionally, specify the IP address and port to start cqlsh on a different node.

```
python cqlsh 1.2.3.4 9042
```

## Using tab completion

You can use tab completion to see hints about how to complete a cqlsh command. Some platforms, such as Mac OSX, do not ship with tab completion installed. You can use easy_install to install tab completion capabilities on Mac OSX:

```
easy_install readline
```

## Creating and updating a keyspace

Creating a keyspace is the CQL counterpart to creating an SQL database, but a little different. The Cassandra keyspace is a namespace that defines how data is replicated on nodes. Typically, a cluster has one keyspace per application. Replication is controlled on a per-keyspace basis, so data that has different replication requirements typically resides in different keyspaces. Keyspaces are not designed to be used as a significant map layer within the data model. Keyspaces are designed to control data replication for a set of tables.

When you create a keyspace, specify a strategy class for replicating keyspaces. Using the SimpleStrategy class is fine for evaluating Cassandra. For production use or for use with mixed workloads, use the NetworkTopologyStrategy class.

To use NetworkTopologyStrategy for evaluation purposes using, for example, a single node cluster, specify the default data center name. To determine the default data center name, use the `nodetool status` command. On Linux, for example, in the installation directory:

```
$ bin/nodetool status
```

The output is:

```
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address    Load       Tokens  Owns (effective)  Host ID      Rack
UN  127.0.0.1  41.62 KB   256     100.0%            75dcca8f...  rack1
```

To use NetworkTopologyStrategy for production use, you need to change the default snitch, SimpleSnitch, to a network-aware snitch, define one or more data center names in the snitch properties file, and use the data center name(s) to define the keyspace; otherwise, Cassandra will fail to complete any write request, such as inserting data into a table, and log this error message:

```
Unable to complete request: one or more nodes were unavailable.
```

You cannot insert data into a table in keyspace that uses NetworkTopologyStrategy unless you define the data center names in the snitch properties file or you use a single data center named datacenter1.

# Example of creating a keyspace

To query Cassandra, create and use a keyspace. Choose an arbitrary data center name and register the name in the properties file of the snitch. Alternatively, in a cluster in a single data center, use the default data center name, for example, datacenter1, and skip registering the name in the properties file.

## Procedure

1. Create a keyspace.

```
cqlsh> CREATE KEYSPACE demodb WITH REPLICATION = { 'class' :
 'NetworkTopologyStrategy', 'datacenter1' : 3 };
```

2. Use the keyspace.

```
USE demodb;
```

# Updating the replication factor

Increasing the replication factor increases the total number of copies of keyspace data stored in a Cassandra cluster. If you are using security features, it is particularly important to increase the replication factor of the system_auth keyspace from the default (1) because you will not be able to log into the cluster if the node with the lone replica goes down. It is recommended to set the replication factor for the system_auth keyspace equal to the number of nodes in each data center.

## Procedure

1. Update a keyspace in the cluster and change its replication strategy options.

```
ALTER KEYSPACE system_auth WITH REPLICATION =
  {'class' : 'NetworkTopologyStrategy', 'dc1' : 3, 'dc2' : 2};
```

   Or if using SimpleStrategy:

```
ALTER KEYSPACE "Excalibur" WITH REPLICATION =
  { 'class' : 'SimpleStrategy', 'replication_factor' : 3 };
```

2. On each affected node, run the `nodetool repair` command.
3. Wait until repair completes on a node, then move to the next node.

# Creating a table

Tables can have single and compound primary keys. To create a table having a single primary key, use the PRIMARY KEY keywords followed by the name of the key, enclosed in parentheses.

## Procedure

1. Create and use the keyspace in the last example if you haven't already done so.

2. Create this users table in the demodb keyspace, making the user name the primary key.

```
CREATE TABLE users (
  user_name varchar,
  password varchar,
  gender varchar,
  session_token varchar,
  state varchar,
  birth_year bigint,
  PRIMARY KEY (user_name));
```

# Using a compound primary key

Use a compound primary key to create columns that you can query to return sorted results.

## Procedure

To create a table having a compound primary key, use two or more columns as the primary key.

```
CREATE TABLE emp (
  empID int,
  deptID int,
  first_name varchar,
  last_name varchar,
  PRIMARY KEY (empID, deptID));
```

The compound primary key is made up of the empID and deptID columns in this example. The empID acts as a partition key for distributing data in the table among the various nodes that comprise the cluster. The remaining component of the primary key, the deptID, acts as a clustering mechanism and ensures that the data is stored in ascending order on disk (much like a clustered index in Microsoft SQL Server).

# Inserting data into a table

In a production database, inserting columns and column values programmatically is more practical than using cqlsh, but often, being able to test queries using this SQL-like shell is very convenient.

## Procedure

To insert employee data for Jane Smith, use the INSERT command.

```
INSERT INTO emp (empID, deptID, first_name, last_name)
  VALUES (104, 15, 'jane', 'smith');
```

# Using a user-defined type

In Cassandra 2.1 and later, you can create a user-defined type to attach multiple data fields to a column. This example shows how to accomplish these tasks:

- Create the user-defined types address and fullname.

- Create a table that defines map and set collection columns of the address and fullname types, respectively.
- Include a name column, also of the fullname type, in the table.
- Insert the street, city, and postal code in the addresses column.
- Insert the first and last name of a person in the name column.
- Filter data on a column of a user-defined type.
- Insert a names of direct reports in the set collection.

## Procedure

1. Create a keyspace.

```
CREATE KEYSPACE mykeyspace WITH REPLICATION = { 'class' :
 'NetworkTopologyStrategy', 'datacenter1' : 1 };
```

2. Create a user-defined type named address.

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

3. Create a user-defined type for the name of a user.

```
CREATE TYPE mykeyspace.fullname (
  firstname text,
  lastname text
);
```

4. Create a table for storing user data in columns of type fullname and address. Use the frozen keyword in the definition of the user-defined type column.

```
CREATE TABLE mykeyspace.users (
  id uuid PRIMARY KEY,
  name frozen <fullname>,
  direct_reports set<frozen <fullname>>,     // a collection set
  addresses map<text, frozen <address>>     // a collection map
);
```

5. Insert a user's name into the fullname column.

```
INSERT INTO mykeyspace.users (id, name) VALUES
 (62c36092-82a1-3a00-93d1-46196ee77204, {firstname: 'Marie-Claude',
 lastname: 'Josset'});
```

6. Insert an address labeled home into the table.

```
UPDATE mykeyspace.users SET addresses = addresses + {'home': { street:
 '191 Rue St. Charles', city: 'Paris', zip_code: 75015, phones: {'33 6 78
 90 12 34'}}} WHERE id=62c36092-82a1-3a00-93d1-46196ee77204;
```

7. Retrieve the full name of a user.

```
SELECT name FROM mykeyspace.users WHERE
 id=62c36092-82a1-3a00-93d1-46196ee77204;
```

Using the column name of a user-defined type retrieves all fields: firstname and lastname.

```
name
------------------------------------------------
 {firstname: 'Marie-Claude', lastname: 'Josset'}
```

Using dot notation, you can retrieve a component of the user-defined type column, for example just the last name.

```
SELECT name.lastname FROM mykeyspace.users WHERE
 id=62c36092-82a1-3a00-93d1-46196ee77204;

 name.lastname
---------------
        Josset
```

8. Filter data on a column of a user-defined type. Create an index and then run a conditional query. In Cassandra 2.1.x, you need to list all components of the name column in the WHERE clause.

```
CREATE INDEX on mykeyspace.users (name);

SELECT id FROM mykeyspace.users WHERE name = {firstname: 'Marie-Claude',
 lastname: 'Josset'};
```

Output is:

```
 id
--------------------------------------
 62c36092-82a1-3a00-93d1-46196ee77204
```

9. Insert names of people who report to Marie-Claude. Use the UPDATE command. Insert the name of a person who reports to another manager using the INSERT command.

   When using the frozen keyword, you cannot update parts of a user-defined type value. The entire value must be overwritten. Cassandra treats the value of a frozen, user-defined type like a blob.

```
UPDATE mykeyspace.users SET direct_reports = { ( 'Naoko', 'Murai'),
 ( 'Sompom', 'Peh') } WHERE id=62c36092-82a1-3a00-93d1-46196ee77204;

INSERT INTO mykeyspace.users (id, direct_reports) VALUES
 ( 7db1a490-5878-11e2-bcfd-0800200c9a66, { ('Jeiranan',
 'Thongnopneua') } );
```

10. Query the table for the direct reports to Marie-Claude.

```
SELECT direct_reports FROM mykeyspace.users;

 direct_reports
-------------------------------------------------------------------------------------
                             {{firstname: 'Jeiranan', lastname:
 'Thongnopneua'}}
 {{firstname: 'Naoko', lastname: 'Murai'}, {firstname: 'Sompom', lastname:
 'Peh'}}
```

# Querying a system table

The system keyspace includes a number of tables that contain details about your Cassandra database objects and cluster configuration.

Cassandra populates these tables and others in the system keyspace.

**Table: Columns in System Tables**

| Table name | Column name | Comment |
|---|---|---|
| schema_keyspaces | keyspace_name, durable_writes, strategy_class, strategy_options | None |
| local | "key", bootstrapped, cluster_name, cql_version, data_center, gossip_generation, native_protocol_version,partitioner, rack, release_version, ring_id, schema_version, thrift_version, tokens set, truncated at map | Information a node has about itself and a superset of gossip. |
| peers | peer, data_center, rack, release_version, ring_id, rpc_address, schema_version, tokens set | Each node records what other nodes tell it about themselves over the gossip. |
| schema_columns | keyspace_name, columnfamily_name, column_name, component_index, index_name, index_options, index_type, validator | Used internally with compound primary keys. |
| schema_columnfamilies | See comment. | Inspect schema_columnfamilies to get detailed information about specific tables. |

# Keyspace, table, and column information

An alternative to the Thrift API describe_keyspaces function is querying system.schema_keyspaces directly. You can also retrieve information about tables by querying system.schema_columnfamilies and about column metadata by querying system.schema_columns.

## Procedure

Query the defined keyspaces using the SELECT statement.

```
SELECT * from system.schema_keyspaces;
```

The cqlsh output includes information about defined keyspaces.

```
 keyspace | durable_writes | name    | strategy_class | strategy_options
----------+----------------+---------+----------------
+----------------------------
  history |           True | history | SimpleStrategy |
 {"replication_factor":"1"}
  ks_info |           True | ks_info | SimpleStrategy |
 {"replication_factor":"1"}

(2 rows)
```

# Cluster information

You can query system tables to get cluster topology information. You can get the IP address of peer nodes, data center and rack names, token values, and other information. *"The Data Dictionary"* article describes querying system tables in detail.

## Procedure

After setting up a 3-node cluster using ccm on the Mac OSX, query the peers and local tables.

```
USE system;
SELECT * FROM peers;
```

Output from querying the peers table looks something like this:

```
   peer     | data_center | host_id      | preferred_ip | rack  |
 release_version | rpc_address | schema_version | tokens
-----------+-------------+-------------+--------------+-------
+-----------------+-------------+---------------+-----------
 127.0.0.3 | datacenter1 | edda8d72... |         null | rack1 |
 2.1.0 | 127.0.0.3   | 59adb24e-f3... |  {3074...
 127.0.0.2 | datacenter1 | ef863afa... |         null | rack1 |
 2.1.0 | 127.0.0.2   | 3d19cd8f-c9... | {-3074...}

(2 rows)
```

# Retrieving and sorting results

Querying tables to select data is the reason data is stored in databases. Similar to SQL, CQL can `SELECT` data using simple or complex qualifiers. At its simplest, a query selects all data in a table. At its most complex, a query delineates which data to retrieve and display.

## Procedure

- The example below illustrates how to create a query that uses first_name and last_name as a filter.

```
cqlsh> SELECT * FROM users_by_name WHERE first_name = 'jane';
```

Note that Cassandra will reject this query if first_nameand last_name are not part of the primary key, either a partition key or clustering column. Queries require a sequential retrieval across the entire users table. In a distributed database like Cassandra, this is a crucial concept to grasp; scanning all data across all nodes is prohibitively slow and thus blocked from execution. The use of partition key and clustering columns in a `WHERE` clause must result in the selection of a contiguous set of rows.

```
 first_name | last_name | age | userid
------------+-----------+-----+--------
       jane |     smith |  45 |    103
       jane |    weasley |  36 |    104
```

- You can also pick the columns to display instead of choosing all data.

```
cqlsh> SELECT first_name, age FROM users_by_name WHERE first_name =
 'jane';
```

```
 first_name | age
------------+-----
       jane |  45
       jane |  36
```

- For a large table, limit the number of rows retrieved using `LIMIT`. The default limit is 10,000 rows. To sample data, pick a smaller number. To retrieve more than 10,000 rows set `LIMIT` to a large value.

```
cqlsh> SELECT * FROM users LIMIT 10;
```

- You can fine-tune the display order using the `ORDER BY` clause, either descending or ascending. The partition key must be defined in the `WHERE` clause and the `ORDER BY` clause defines the clustering column to use for ordering.

```
cqlsh> SELECT * FROM users WHERE userID IN (102,104) ORDER BY age ASC;
```

```
 userid | age | fn    | ln
--------+-----+-------+---------
    104 |  36 | jane | weasley
    102 |  45 | dick |   tracy
```

# Retrieval using the IN keyword

Similar to a SQL query, in CQL queries the `IN` keyword can define a set of clustering columns to fetch together, supporting a "multi-get" of CQL rows. A single clustering column can be defined if all preceding columns are defined for either equality or group inclusion. Alternatively, several clustering columns may be defined to collect several rows, as long as all preceding columns are queried for equality or group inclusion. The defined clustering columns can also be queried for inequality.

Note that using both `IN` and `ORDER BY` will require turning off paging with the `PAGING OFF` command in cqlsh.

## Procedure

- Turn off paging

```
cqlsh> PAGING OFF
```

- Retrieve and sort results in descending order.

```
cqlsh> SELECT * FROM users WHERE userID IN (100,102,104) ORDER BY age
 DESC;
```

```
 userid | age | fn    | ln
--------+-----+-------+----------
    102 |  45 | dick  |    tracy
    104 |  36 | jane  | weasley
    100 |  21 |  joe  | america
```

- Retrieve rows using multiple clustering columns event_start_date and event_end_date. This example searches the partition key event_ids for several events, but the partition key can also be composed as an equality for one value.

```
cqlsh> SELECT * FROM calendar WHERE event_id IN (100, 101, 102) AND
 (event_start_date, event_end_date) IN (('2015-05-09','2015-05-31'),
('2015-05-06', '2015-05-31'));
```

```
 event_id | event_start_date | event_end_date
----------+------------------+----------------
      100 |       2015-05-06 |     2015-05-31
      100 |       2015-05-09 |     2015-05-31
```

- Retrieve rows using multiple clustering columns event_start_date and event_end_date and inequality.

```
cqlsh> SELECT * FROM calendar WHERE event_id IN (100, 101, 102) AND
 (event_start_date, event_end_date) >= ('2015-05-09','2015-05-24');
```

```
 event_id | event_start_date | event_end_date
----------+------------------+----------------
      100 |       2015-05-09 |     2015-05-31
```

## Slicing over partition rows

In Cassandra 2.0.6 and later, you can use a new syntax for slicing over rows of a partition when the table has more than one clustering column. Using a conditional operator, you can compare groups of clustering keys to certain values. For example:

```
CREATE TABLE timeline (
   day text,
   hour int,
   min int,
   sec int,
   value text,
   PRIMARY KEY (day, hour, min, sec)
   );

   INSERT INTO timeline (day, hour, min, sec, value)
   VALUES ('12 Jan 2014', 3, 43, 12, 'event1');
   INSERT INTO timeline (day, hour, min, sec, value)
   VALUES ('12 Jan 2014', 3, 52, 58, 'event2');
   INSERT INTO timeline (day, hour, min, sec, value)
   VALUES ('12 Jan 2014', 4, 37, 01, 'event3');
   INSERT INTO timeline (day, hour, min, sec, value)
   VALUES ('12 Jan 2014', 4, 37, 41, 'event3');
   INSERT INTO timeline (day, hour, min, sec, value)
   VALUES ('12 Jan 2014', 6, 00, 34, 'event4');

   SELECT * FROM timeline;
```

```
   day          | hour | min | sec | value
   -------------+------+-----+-----+--------
   12 Jan 2014 |    3 |  43 |  12 | event1
   12 Jan 2014 |    3 |  52 |  58 | event2
   12 Jan 2014 |    4 |  37 |   1 | event3
   12 Jan 2014 |    4 |  37 |  41 | event3
   12 Jan 2014 |    6 |   0 |  34 | event4
```

To retrieve events for the 12th of January 2014 between 3:50:00 and 4:37:30, use the new syntax as follows:

```
SELECT * FROM timeline WHERE day='12 Jan 2014'
   AND (hour, min) >= (3, 50)
   AND (hour, min, sec) <= (4, 37, 30);
```

```
   day          | hour | min | sec | value
   -------------+------+-----+-----+--------
   12 Jan 2014 |    3 |  52 |  58 | event2
```

```
    12 Jan 2014 |    4 |  37 |   1 | event3
```

The new syntax, in this example, uses a conditional operator to compare groups of clustering keys, such as hour, min, and sec, to certain values.

In the WHERE clause, you need to use sequential clustering columns. The sequence must match the sequence of the columns in the table definition. For example:

```
CREATE TABLE no_column_skipping
(a int, b int, c int, d int, e int,
PRIMARY KEY (a, b, c, d))
```

This WHERE clause does not work:

```
SELECT ... WHERE a=0 AND (b, d) > (1, 2)
```

This WHERE clause works:

```
SELECT ... WHERE a=0 AND (b, c) > (1, 2)
```

## Batching conditional updates to a static column

As explained in the BATCH statement reference, in Cassandra 2.0.6 and later, you can batch conditional updates. This example shows batching conditional updates combined with using static columns, also introduced in Cassandra 2.0.6. The example stores records about each purchase by user and includes the running balance of all a user's purchases.

```
CREATE TABLE purchases (
     user text,
     balance int static,
     expense_id int,
     amount int,
     description text,
     paid boolean,
     PRIMARY KEY (user, expense_id)
  );
```

Because the balance is static, all purchase records for a user have the same running balance.

The statements for inserting values into purchase records use the IF conditional clause.

```
BEGIN BATCH
  INSERT INTO purchases (user, balance) VALUES ('user1', -8) IF NOT EXISTS;
  INSERT INTO purchases (user, expense_id, amount, description, paid)
    VALUES ('user1', 1, 8, 'burrito', false);
APPLY BATCH;

BEGIN BATCH
  UPDATE purchases SET balance = -208 WHERE user='user1' IF balance = -8;
  INSERT INTO purchases (user, expense_id, amount, description, paid)
    VALUES ('user1', 2, 200, 'hotel room', false);
APPLY BATCH;
```

Because the column is static, you can provide only the partition key when updating the data. To update a non-static column, you would also have to provide a clustering key. Using batched conditional updates, you can maintain a running balance. If the balance were stored in a separate table, maintaining a running

balance would not be posssible because a batch having conditional updates cannot span multiple partitions.

```
SELECT * FROM purchases;
```

At this point, the output is:

```
 user  | expense_id | balance | amount | description | paid
-------+------------+---------+--------+-------------+-------
 user1 |          1 |    -208 |      8 |     burrito | False
 user1 |          2 |    -208 |    200 |  hotel room | False
```

You could then use a conditional batch to update records to clear the balance.

```
BEGIN BATCH
  UPDATE purchases SET balance=-200 WHERE user='user1' IF balance=-208;
  UPDATE purchases SET paid=true WHERE user='user1' AND expense_id=1 IF
 paid=false;
APPLY BATCH;

SELECT * FROM purchases;
```

```
 user  | expense_id | balance | amount | description | paid
-------+------------+---------+--------+-------------+-------
 user1 |          1 |    -200 |      8 |     burrito | True
 user1 |          2 |    -208 |    200 |  hotel room | False
```

# Using and misusing batches

Batches are often mistakenly used in an attempt to optimize performance. Unlogged batches require the coordinator to manage inserts, which can place a heavy load on the coordinator node. If other nodes own partition keys, the coordinator node needs to deal with a network hop, resulting in inefficient delivery. Use unlogged batches when making updates to the same partition key.

Using a primary key of (date, timestamp) for example, this unlogged batch resolves to only one write internally, regardless of the number of writes, assuming all have the same date value.

```
BEGIN UNLOGGED BATCH;
  INSERT INTO sensor_readings (date, time, reading) values
 (20140910,'2014-09-10T11:00:00.00+0000', 6335.2);
  INSERT INTO sensor_readings (date, time, reading) values
 (20140910,'2014-09-10T11:00:15.00+0000', 5222.2);
APPLY BATCH;
```

The coordinator node might also need to work hard to process a logged batch while maintaining consistency between tables. For example, upon receiving a batch, the coordinator node sends batch logs to two other nodes. In the event of a coordinator failure, the other nodes retry the batch. The entire cluster is affected. Use a logged batch to synchronize tables, as shown in this example:

```
BEGIN BATCH;
  UPDATE users
    SET state = 'TX'
    WHERE user_uuid = 8a172618-b121-4136-bb10-f665cfc469eb;
  UPDATE users_by_ssn
    SET state = 'TX'
    WHERE ssn = '888-99-3987';
APPLY BATCH;
```

For information about the fastest way to load data, see "Cassandra: Batch loading without the Batch keyword."

# Using the keyspace qualifier

Sometimes issuing a USE statement to select a keyspace is inconvenient. If you use connection pooling, for example, you have multiple keyspaces to juggle. To simplify tracking multiple keyspaces, use the keyspace qualifier instead of the USE statement. You can specify the keyspace using the keyspace qualifier in these statements:

- ALTER TABLE
- CREATE TABLE
- DELETE
- INSERT
- SELECT
- TRUNCATE
- UPDATE

## Procedure

To specify a table when you are not in the keyspace containing the table, use the name of the keyspace followed by a period, then the table name. For example, Music.playlists.

```
INSERT INTO Music.playlists (id, title, artist, album)
  VALUES (62c36092-82a1-3a00-93d1-46196ee77204, 'La Grange', 'ZZ Top',
 'Tres Hombres');
```

# Adding columns to a table

The ALTER TABLE command adds new columns to a table.

## Procedure

Add a coupon_code column with the varchar data type to the users table.

```
cqlsh:demodb> ALTER TABLE users ADD coupon_code varchar;
```

This creates the column metadata and adds the column to the table schema, but does not update any existing rows.

# Expiring data

Data in a column, other than a counter column, can have an optional expiration period called TTL (time to live). The client request specifies a TTL value, defined in seconds, for the data. TTL data is marked with a tombstone after the requested amount of time has expired. A tombstone exists for gc_grace_seconds. After data is marked with a tombstone, the data is automatically removed during the normal compaction and repair processes.

Use CQL to set the TTL for data.

If you want to change the TTL of expiring data, you have to re-insert the data with a new TTL. In Cassandra, the insertion of data is actually an insertion or update operation, depending on whether or not a previous version of the data exists.

TTL data has a precision of one second, as calculated on the server. Therefore, a very small TTL probably does not make much sense. Moreover, the clocks on the servers should be synchronized; otherwise reduced precision could be observed because the expiration time is computed on the primary host that receives the initial insertion but is then interpreted by other hosts on the cluster.

Expiring data has an additional overhead of 8 bytes in memory and on disk (to record the TTL and expiration time) compared to standard data.

# Expiring data example

Both the INSERT and UPDATE commands support setting a time for data in a column to expire. The expiration time (TTL) is set using CQL.

## Procedure

1. Use the INSERT command to set a password column in the users table to expire in 86400 seconds, or one day.

```
cqlsh:demodb> INSERT INTO users
                (user_name, password)
                VALUES ('cbrown', 'ch@ngem4a') USING TTL 86400;
```

2. Extend the expiration period to five days by using the UPDATE command/

```
cqlsh:demodb> UPDATE users USING TTL 432000 SET password = 'ch@ngem4a'
                WHERE user_name = 'cbrown';
```

# Determining time-to-live for a column

This procedure creates a table, inserts data into two columns, and calls the TTL function to retrieve the date/time of the writes to the columns.

## Procedure

1. Create a users table named clicks in the excelsior keyspace.

```
CREATE TABLE excelsior.clicks (
  userid uuid,
  url text,
  date timestamp,  //unrelated to WRITETIME discussed in the next section
  name text,
  PRIMARY KEY (userid, url)
);
```

2. Insert data into the table, including a date in yyyy-mm-dd format, and set that data to expire in a day (86400 seconds). Use the USING TTL clause to set the expiration period.

```
INSERT INTO excelsior.clicks (
```

```
  userid, url, date, name)
  VALUES (
    3715e600-2eb0-11e2-81c1-0800200c9a66,
    'http://apache.org',
    '2013-10-09', 'Mary')
    USING TTL 86400;
```

3. Wait for a while and then issue a SELECT statement to determine how much longer the data entered in step 2 has to live.

```
SELECT TTL (name) from excelsior.clicks
  WHERE url = 'http://apache.org' ALLOW FILTERING;
```

Output is, for example, 85908 seconds:

```
 ttl(name)
-----------
 85908

(1 rows)
```

# Removing a keyspace, schema, or data

To remove data, you can set column values for automatic removal using the TTL (time-to-expire) table attribute. You can also drop a table or keyspace, and delete keyspace column metadata.

# Dropping a table or keyspace

You drop a table or keyspace using the DROP command.

## Procedure

1. Drop the users table.

```
cqlsh:demodb> DROP TABLE users;
```

2. Drop the demodb keyspace.

```
cqlsh:demodb> DROP KEYSPACE demodb;
```

# Deleting columns and rows

CQL provides the DELETE command to delete a column or row. Deleted values are removed completely by the first compaction following deletion.

## Procedure

1. Deletes user jsmith's session token column.

```
cqlsh:demodb> DELETE session_token FROM users where pk = 'jsmith';
```

2. Delete jsmith's entire row.

```
cqlsh:demodb> DELETE FROM users where pk = 'jsmith';
```

# Determining the date/time of a write

A table contains a timestamp representing the date/time that a write occurred to a column. Using the WRITETIME function in a SELECT statement returns the date/time that the column was written to the database. The output of the function is microseconds except in the case of Cassandra 2.1 counter columns. Counter column writetime is milliseconds. This procedure continues the example from the previous procedure and calls the WRITETIME function to retrieve the date/time of the writes to the columns.

## Procedure

1. Insert more data into the table.

```
INSERT INTO excelsior.clicks (
  userid, url, date, name)
  VALUES (
    cfd66ccc-d857-4e90-b1e5-df98a3d40cd6,
    'http://google.com',
    '2013-10-11', 'Bob'
);
```

2. Retrieve the date/time that the value Mary was written to the name column of the apache.org data. Use the WRITETIME function in a SELECT statement, followed by the name of a column in parentheses:

```
SELECT WRITETIME (name) FROM excelsior.clicks
  WHERE url = 'http://apache.org' ALLOW FILTERING;
```

The writetime output in microseconds converts to Sun, 14 Jul 2013 21:57:58 GMT or to 2:57 pm Pacific time.

```
writetime(name)
------------------
1373839078327001
```

3. Retrieve the date/time of the last write to the date column for google.com data.

```
SELECT WRITETIME (date) FROM excelsior.clicks
  WHERE url = 'http://google.com' ALLOW FILTERING;
```

The writetime output in microseconds converts to Sun, 14 Jul 2013 22:03:15 GMT or 3:03 pm Pacific time.

```
writetime(date)
------------------
1373839395324001
```

# Altering the data type of a column

Using ALTER TABLE, you can change the data type of a column after it is defined or added to a table.

## Procedure

Change the coupon_code column to store coupon codes as integers instead of text by changing the data type of the column.

```
cqlsh:demodb> ALTER TABLE users ALTER coupon_code TYPE int;
```

Only newly inserted values, not existing coupon codes are validated against the new type.

# Using collections

Cassandra includes collection types that provide an improved way of handling tasks, such as building multiple email address capability into tables. Observe the following limitations of collections:

- The maximum size of an item in a collection is 64K or 2B, depending on the native protocol version.
- Keep collections small to prevent delays during querying because Cassandra reads a collection in its entirety. The collection is not paged internally.

  As discussed earlier, collections are designed to store only a small amount of data.
- Never insert more than 64K items in a collection.

  If you insert more than 64K items into a collection, only 64K of them will be queryable, resulting in data loss.

**Note:** For detailed limits on collections, see CQL limits.

You can expire each element of a collection by setting an individual time-to-live (TTL) property.

# Using the set type

A set stores a group of elements that are returned in sorted order when queried. A column of type set consists of unordered unique values. Using the set data type, you can solve the multiple email problem in an intuitive way that does not require a read before adding a new email address.

## Procedure

1. Define a set, emails, in the users table to accommodate multiple email address.

   ```
   CREATE TABLE users (
     user_id text PRIMARY KEY,
     first_name text,
     last_name text,
     emails set<text>
   );
   ```

2. Insert data into the set, enclosing values in curly brackets.

   Set values must be unique.

   ```
   INSERT INTO users (user_id, first_name, last_name, emails)
   ```

```
  VALUES('frodo', 'Frodo', 'Baggins', {'f@baggins.com',
 'baggins@gmail.com'});
```

3. Add an element to a set using the UPDATE command and the addition (+) operator.

```
UPDATE users
  SET emails = emails + {'fb@friendsofmordor.org'} WHERE user_id =
 'frodo';
```

4. Retrieve email addresses for frodo from the set.

```
SELECT user_id, emails FROM users WHERE user_id = 'frodo';
```

When you query a table containing a collection, Cassandra retrieves the collection in its entirety; consequently, keep collections small enough to be manageable, or construct a data model to replace collections that can accommodate large amounts of data.

Cassandra returns results in an order based on the type of the elements in the collection. For example, a set of text elements is returned in alphabetical order. If you want elements of the collection returned in insertion order, use a list.

```
 user_id | emails
---------
+-----------------------------------------------------------------
 frodo   |
 {"baggins@caramail.com","f@baggins.com","fb@friendsofmordor.org"}
```

5. Remove an element from a set using the subtraction (-) operator.

```
UPDATE users
  SET emails = emails - {'fb@friendsofmordor.org'} WHERE user_id =
 'frodo';
```

6. Remove all elements from a set by using the UPDATE or DELETE statement.

A set, list, or map needs to have at least one element; otherwise, Cassandra cannot distinguish the set from a null value.

```
UPDATE users SET emails = {} WHERE user_id = 'frodo';

DELETE emails FROM users WHERE user_id = 'frodo';
```

A query for the emails returns null.

```
SELECT user_id, emails FROM users WHERE user_id = 'frodo';
```

```
 user_id | emails
---------+---------
 frodo   | null
```

# Using the list type

When the order of elements matters, which may not be the natural order dictated by the type of the elements, use a list. Also, use a list when you need to store same value multiple times. List values are returned according to their index value in the list, whereas set values are returned in alphabetical order, assuming the values are text.

Using the list type you can add a list of preferred places for each user in a users table, and then query the database for the top x places for a user.

## Procedure

1. Add a list declaration to a table by adding a column top_places of the list type to the users table.

```
ALTER TABLE users ADD top_places list<text>;
```

2. Use the UPDATE command to insert values into the list.

```
UPDATE users
  SET top_places = [ 'rivendell', 'rohan' ] WHERE user_id = 'frodo';
```

3. Prepend an element to the list by enclosing it in square brackets, and using the addition (+) operator.

```
UPDATE users
  SET top_places = [ 'the shire' ] + top_places WHERE user_id = 'frodo';
```

4. Append an element to the list by switching the order of the new element data and the list name in the UPDATE command.

```
UPDATE users
  SET top_places = top_places + [ 'mordor' ] WHERE user_id = 'frodo';
```

   These update operations are implemented internally without any read-before-write. Appending and prepending a new element to the list writes only the new element.

5. Add an element at a particular position using the list index position in square brackets

```
UPDATE users SET top_places[2] = 'riddermark' WHERE user_id = 'frodo';
```

   When you add an element at a particular position, Cassandra reads the entire list, and then writes only the updated element. Consequently, adding an element at a particular position results in greater latency than appending or prefixing an element to a list.

6. Remove an element from a list using the DELETE command and the list index position in square brackets. For example, remove mordor, leaving the shire, rivendell, and riddermark.

```
DELETE top_places[3] FROM users WHERE user_id = 'frodo';
```

7. Remove all elements having a particular value using the UPDATE command, the subtraction operator (-), and the list value in square brackets. For example, remove riddermark.

```
UPDATE users
  SET top_places = top_places - ['riddermark'] WHERE user_id = 'frodo';
```

   The former, indexed method of removing elements from a list requires a read internally. Using the UPDATE command as shown here is recommended over emulating the operation client-side by reading the whole list, finding the indexes that contain the value to remove, and then removing those indexes. This emulation would not be thread-safe. If another thread/client prefixes elements to the list between the read and the write, the wrong elements are removed. Using the UPDATE command as shown here does not suffer from that problem.

8. Query the database for a list of top places.

```
SELECT user_id, top_places FROM users WHERE user_id = 'frodo';
```

Results show:

```
 user_id | top_places
---------+----------------------------
   frodo | ['the shire', 'rivendell']
```

# Using the map type

As its name implies, a map maps one thing to another. A map is a name and a pair of typed values. Using the map type, you can store timestamp-related information in user profiles. Each element of the map is internally stored as one Cassandra column that you can modify, replace, delete, and query. Each element can have an individual time-to-live and expire when the TTL ends.

## Procedure

1. Add a todo list to every user profile in an existing users table using the CREATE TABLE or ALTER statement, specifying the map collection and enclosing the pair of data types in angle brackets.

```
ALTER TABLE users ADD todo map<timestamp, text>;
```

2. Set or replace map data, using the INSERT or UPDATE command, and enclosing the timestamp and text values in a map collection: curly brackets, separated by a colon.

```
UPDATE users
  SET todo =
  { '2012-9-24' : 'enter mordor',
  '2014-10-2 12:00' : 'throw ring into mount doom' }
  WHERE user_id = 'frodo';
```

3. Set a specific element using the UPDATE command, enclosing the timestamp of the element in square brackets, and using the equals operator to map the value to that timestamp.

```
UPDATE users SET todo['2014-10-2 12:00'] = 'throw my precious into mount
 doom'
  WHERE user_id = 'frodo';
```

4. Use INSERT to specify data in a map collection.

```
INSERT INTO users (user_id, todo) VALUES ('frodo', { '2013-9-22 12:01' :
 'birthday wishes to Bilbo', '2013-10-1 18:00': 'Check into Inn of Pracing
 Pony'}) ;
```

In Cassandra 2.1.1 and later, you can add map elements using this syntax:

```
UPDATE users SET todo = todo + { '2013-9-22 12:01' : 'birthday wishes
 to Bilbo', '2013-10-1 18:00': 'Check into Inn of Pracing Pony'} WHERE
 user_id='frodo';
```

Inserting this data into the map replaces the entire map.

5. Delete an element from the map using the DELETE command and enclosing the timestamp of the element in square brackets:

```
DELETE todo['2013-9-22 12:01'] FROM users WHERE user_id = 'frodo';
```

In Cassandra 2.1.1 and later, you can delete multiple map elements using this syntax:

```
UPDATE users SET todo=todo - {'2013-9-22 12:01','2013-10-01
 18:00:00-0700'} WHERE user_id='frodo';
```

6. Retrieve the todo map.

```
SELECT user_id, todo FROM users WHERE user_id = 'frodo';
```

The order of the map output depends on the type of the map.

7. Compute the TTL to use to expire todo list elements on the day of the timestamp, and set the elements to expire.

```
UPDATE users USING TTL <computed_ttl>
  SET todo['2012-10-1'] = 'find water' WHERE user_id = 'frodo';
```

# Indexing a column

You can use cqlsh to create an index on column values. In Cassandra 2.1 and later, you can index collection columns. Indexing can impact performance greatly. Before creating an index, be aware of when and when not to create an index.

## Procedure

1. Creates an index on the state and birth_year columns in the users table.

```
cqlsh:demodb> CREATE INDEX state_key ON users (state);
cqlsh:demodb> CREATE INDEX birth_year_key ON users (birth_year);
```

2. Query the columns that are now indexed.

```
cqlsh:demodb> SELECT * FROM users
                WHERE gender = 'f' AND
                state = 'TX' AND
                birth_year > 1968
                ALLOW FILTERING;
```

# Using lightweight transactions

INSERT and UPDATE statements using the IF clause, support lightweight transactions, also known as Compare and Set (CAS).

## Procedure

1. Register a new user.

```
INSERT INTO users (login, email, name, login_count)
  VALUES ('jdoe', 'jdoe@abc.com', 'Jane Doe', 1)
  IF NOT EXISTS;
```

2. Perform a CAS operation against a row that does exist by adding the predicate for the operation at the end of the query. For example, reset Jane Doe's password.

```
UPDATE users
  SET email = ‘janedoe@abc.com’
  WHERE login = 'jdoe'
  IF email = ‘jdoe@abc.com’;
```

# Paging through unordered partitioner results

When using the RandomPartitioner or Murmur3Partitioner, Cassandra rows are ordered by the hash of their value and hence the order of rows is not meaningful. Using CQL, you can page through rows even when using the random partitioner or the murmur3 partitioner using the token function as shown in this example:

```
SELECT * FROM test WHERE token(k) > token(42);
```

The ByteOrdered partitioner arranges tokens the same way as key values, but the RandomPartitioner and Murmur3Partitioner distribute tokens in a completely unordered manner. The token function makes it possible to page through these unordered partitioner results. Using the token function actually queries results directly using tokens. Underneath, the token function makes token-based comparisons and does not convert keys to tokens (not k > 42).

You can use the TOKEN function to express a conditional relation on a partition key column. In this case, the query returns rows based on the token of the partition key rather than on the value.

# Using a counter

A counter is a special column used to store a number that is changed in increments. For example, you might use a counter column to count the number of times a page is viewed.

Cassandra 2.1 counter column improves the implementation of counters and provides a number of configuration options to tune counters. In Cassandra 2.1 and later, you can configure how long the coordinator can wait for counter writes to complete, the size of the counter cache in memory, how long Cassandra waits before saving counter cache keys, the number of keys to save, and concurrent_counter_writes. You set the options in the `cassandra.yaml` file. The replicate_on_write table property used by the Cassandra 2.0.x counter implementation has been removed from Cassandra 2.1.

Define a counter in a dedicated table only and use the counter data type. You cannot index, delete, or re-add a counter column. All non-counter columns in the table must be defined as part of the primary key.

To load data into a counter column, or to increase or decrease the value of the counter, use the UPDATE command. Cassandra rejects USING TIMESTAMP or USING TTL in the command to update a counter column.

## Procedure

1. Create a keyspace. For example, on Linux create a keyspace for use in a single data center having a replication factor of 3. Use the default data center name from the output of the `nodetool status` command, for example datacenter1.

```
CREATE KEYSPACE counterks WITH REPLICATION =
{ 'class' : 'NetworkTopologyStrategy', 'datacenter1' : 3 };
```

2. Create a table for the counter column.

```
CREATE TABLE counterks.page_view_counts
  (counter_value counter,
  url_name varchar,
  page_name varchar,
  PRIMARY KEY (url_name, page_name)
);
```

3. Load data into the counter column.

```
UPDATE counterks.page_view_counts
 SET counter_value = counter_value + 1
 WHERE url_name='www.datastax.com' AND page_name='home';
```

4. Take a look at the counter value.

```
SELECT * FROM counterks.page_view_counts;
```

Output is:

```
 url_name         | page_name | counter_value
------------------+-----------+---------------
 www.datastax.com |      home |             1
```

5. Increase the value of the counter.

```
UPDATE counterks.page_view_counts
  SET counter_value = counter_value + 2
  WHERE url_name='www.datastax.com' AND page_name='home';
```

6. Take a look at the counter value.

```
 url_name         | page_name | counter_value
------------------+-----------+---------------
 www.datastax.com |      home |             3
```

# Tracing consistency changes

In a distributed system such as Cassandra, the most recent value of data is not necessarily on every node all the time. The client application configures the consistency level per request to manage response time versus data accuracy. By tracing activity on a five-node cluster, this tutorial shows the difference between these consistency levels and the number of replicas that participate to satisfy a request:

- ONE

  Returns data from the nearest replica.
- QUORUM

  Returns the most recent data from the majority of replicas.
- ALL

  Returns the most recent data from all replicas.

Follow instructions to setup five nodes on your local computer, trace reads at different consistency levels, and then compare the results.

# Setup to trace consistency changes

To setup five nodes on your local computer, trace reads at different consistency levels, and then compare the results.

## Procedure

1. Get the ccm library of scripts from github.

   You will use this library in subsequent steps to perform the following actions:

   - Download Apache Cassandra source code.
   - Create and launch an Apache Cassandra cluster on a single computer.

   Refer to the ccm README for prerequisites.
2. Set up loopback aliases. For example, enter the following commands on the command line to set up the alias on the Mac. On some platforms, you can probably skip this step.

   ```
   $ sudo ifconfig lo0 alias 127.0.0.2 up
   $ sudo ifconfig lo0 alias 127.0.0.3 up
   $ sudo ifconfig lo0 alias 127.0.0.4 up
   $ sudo ifconfig lo0 alias 127.0.0.5 up
   ```
3. Download Apache Cassandra source code, version 2.1.0 for example, into the /.ccm/repository, and start the ccm cluster named trace_consistency.

   ```
   $ ccm create trace_consistency -v 2.1.0
   ```

   ```
   Downloading http://archive.apache.org/dist/cassandra/2.1.0/
   apache-cassandra-2.1.0-src.tar.gz to /var/folders/9k/
   ywsprd8n14s7hzb5qnztgb5h0000gq/T/ccm-d7fGAN.tar.gz (15.750MB)
     16514874  [100.00%]
   Extracting /var/folders/9k/ywsprd8n14s7hzb5qnztgb5h0000gq/T/ccm-
   d7fGAN.tar.gz as version 2.1.0 ...
   Compiling Cassandra 2.1.0 ...
   Current cluster is now: trace_consistency
   ```
4. Use the following commands to populate and check the cluster:

   ```
   $ ccm populate -n 5
   $ ccm start
   ```
5. Check that the cluster is up:

   ```
   $ ccm node1 ring
   ```

   The output shows the status of all five nodes.

**Related information**

Cassandra 2.0 cassandra.yaml

Cassandra 2.1 cassandra.yaml

# Trace reads at different consistency levels

After performing the setup steps, run and trace queries that read data at different consistency levels. The tracing output shows that using three replicas on a five-node cluster, a consistency level of ONE processes responses from one of three replicas, QUORUM from two of three replicas, and ALL from three of three replicas.

## Procedure

1. Connect cqlsh to the first node in the ring.

```
$ ccm node1 cqlsh
```

2. On the cqlsh command line, create a keyspace that specifies using three replica for data distribution in the cluster.

```
cqlsh> CREATE KEYSPACE demo_cl WITH replication =
 {'class':'SimpleStrategy', 'replication_factor':3};
```

3. In the three-replica keyspace, create a table, and insert some values:

```
cqlsh> USE demo_cl;
cqlsh:demo_cl> CREATE TABLE demo_table ( id int PRIMARY KEY, col1 int,
 col2 int );
cqlsh:demo_cl> INSERT INTO demo_table (id, col1, col2) VALUES (0, 0, 0);
```

4. Turn on tracing and use the CONSISTENCY command to check that the consistency level is ONE, the default.

```
cqlsh:demo_cl> TRACING on;
cqlsh:demo_cl> CONSISTENCY;
```

   The output should be:

```
Current consistency level is 1.
```

5. Query the table to read the value of the primary key.

```
cqlsh:demo_cl> SELECT * FROM demo_table WHERE id = 0;
```

   The output includes tracing information:

```
 id | col1 | col2
----+------+------
  0 |    0 |    0

(1 rows)

Tracing session: 0f5058d0-6761-11e4-96a3-fd07420471ed

 activity
                              | timestamp                  | source    |
 source_elapsed
-------------------------------------------------------------------------------------------
+----------------------------+-----------+----------------

    Execute CQL3 query | 2014-11-08 08:05:29.437000 | 127.0.0.1 |
     0
                Parsing SELECT * FROM demo_table WHERE id = 0 LIMIT 10000;
 [SharedPool-Worker-1] | 2014-11-08 08:05:29.438000 | 127.0.0.1 |
   820
                                                      Preparing statement
 [SharedPool-Worker-1] | 2014-11-08 08:05:29.438000 | 127.0.0.1 |
  1637
                                                   Sending message
 to /127.0.0.3 [WRITE-/127.0.0.3] | 2014-11-08 08:05:29.439000 | 127.0.0.1
 |           2211
```

```
                                                     Sending message
to /127.0.0.4 [WRITE-/127.0.0.4] | 2014-11-08 08:05:29.439000 | 127.0.0.1
|           2237
                                                   Message received
from /127.0.0.1 [Thread-10] | 2014-11-08 08:05:29.441000 | 127.0.0.3 |
         75
                             Executing single-partition query on demo_table
[SharedPool-Worker-1] | 2014-11-08 08:05:29.441000 | 127.0.0.4 |
  818
                                                 Acquiring sstable references
[SharedPool-Worker-1] | 2014-11-08 08:05:29.441000 | 127.0.0.4 |
  861
                                                  Merging memtable tombstones
[SharedPool-Worker-1] | 2014-11-08 08:05:29.441000 | 127.0.0.4 |
  915
Skipped 0/0 non-slice-intersecting sstables, included 0 due to tombstones
[SharedPool-Worker-1] | 2014-11-08 08:05:29.442000 | 127.0.0.4 |
  999
                                  Merging data from memtables and 0 sstables
[SharedPool-Worker-1] | 2014-11-08 08:05:29.442000 | 127.0.0.4 |
 1018
                                                  Merging memtable tombstones
[SharedPool-Worker-1] | 2014-11-08 08:05:29.442000 | 127.0.0.3 |
 1058
Skipped 0/0 non-slice-intersecting sstables, included 0 due to tombstones
[SharedPool-Worker-1] | 2014-11-08 08:05:29.442000 | 127.0.0.3 |
 1146
                                  Merging data from memtables and 0 sstables
[SharedPool-Worker-1] | 2014-11-08 08:05:29.442000 | 127.0.0.3 |
 1165
                                       Read 1 live and 0 tombstoned cells
[SharedPool-Worker-1] | 2014-11-08 08:05:29.442000 | 127.0.0.3 |
 1223
                                         Enqueuing response to /127.0.0.1
[SharedPool-Worker-1] | 2014-11-08 08:05:29.442001 | 127.0.0.3 |
 1504
                                                   Message received
from /127.0.0.4 [Thread-7] | 2014-11-08 08:05:29.443000 | 127.0.0.1 |
      6399
                                                    Sending message
to /127.0.0.1 [WRITE-/127.0.0.1] | 2014-11-08 08:05:29.443000 | 127.0.0.3
|           1835
                                                   Message received
from /127.0.0.3 [Thread-8] | 2014-11-08 08:05:29.443000 | 127.0.0.1 |
      6449
                                     Processing response from /127.0.0.4
[SharedPool-Worker-2] | 2014-11-08 08:05:29.443000 | 127.0.0.1 |
 6623
                                     Processing response from /127.0.0.3
[SharedPool-Worker-3] | 2014-11-08 08:05:29.443000 | 127.0.0.1 |
 6635

     Request complete | 2014-11-08 08:05:29.443897 | 127.0.0.1 |
 6897
```

6. Change the consistency level to QUORUM and run the SELECT statement again.

```
cqlsh:demo_cl> CONSISTENCY quorum;
cqlsh:demo_cl> SELECT * FROM demo_table WHERE id = 0;
```

```
 id | col1 | col2
----+------+------
  0 |    0 |    0
```

```
(1 rows)


Tracing session: 3bbae430-6761-11e4-96a3-fd07420471ed

 activity
                            | timestamp                  | source    |
 source_elapsed
-------------------------------------------------------------------------------------------
+---------------------------+----------+----------------

    Execute CQL3 query | 2014-11-08 08:06:43.955000 | 127.0.0.1 |
     0
               Parsing SELECT * FROM demo_table WHERE id = 0 LIMIT 10000;
 [SharedPool-Worker-1] | 2014-11-08 08:06:43.955000 | 127.0.0.1 |
    71
                                                      Preparing statement
 [SharedPool-Worker-1] | 2014-11-08 08:06:43.955000 | 127.0.0.1 |
   267
                                                    Sending message
 to /127.0.0.4 [WRITE-/127.0.0.4] | 2014-11-08 08:06:43.956000 | 127.0.0.1
 |           1628
                                                    Sending message
 to /127.0.0.5 [WRITE-/127.0.0.5] | 2014-11-08 08:06:43.956000 | 127.0.0.1
 |           1690
                                                  Message received
 from /127.0.0.1 [Thread-9] | 2014-11-08 08:06:43.957000 | 127.0.0.5 |
         95
                             Executing single-partition query on demo_table
 [SharedPool-Worker-2] | 2014-11-08 08:06:43.957000 | 127.0.0.4 |
   229
                                                 Acquiring sstable references
 [SharedPool-Worker-2] | 2014-11-08 08:06:43.957000 | 127.0.0.4 |
   249
                                                  Merging memtable tombstones
 [SharedPool-Worker-2] | 2014-11-08 08:06:43.957000 | 127.0.0.4 |
   299
 Skipped 0/0 non-slice-intersecting sstables, included 0 due to tombstones
 [SharedPool-Worker-2] | 2014-11-08 08:06:43.957000 | 127.0.0.4 |
   387
                                Merging data from memtables and 0 sstables
 [SharedPool-Worker-2] | 2014-11-08 08:06:43.957000 | 127.0.0.4 |
   408
                                       Read 1 live and 0 tombstoned cells
 [SharedPool-Worker-2] | 2014-11-08 08:06:43.957000 | 127.0.0.4 |
   469
                                         Enqueuing response to /127.0.0.1
 [SharedPool-Worker-2] | 2014-11-08 08:06:43.957001 | 127.0.0.4 |
   734
                                                    Sending message
 to /127.0.0.1 [WRITE-/127.0.0.1] | 2014-11-08 08:06:43.957001 | 127.0.0.4
 |            894
                                                  Message received
 from /127.0.0.4 [Thread-7] | 2014-11-08 08:06:43.958000 | 127.0.0.1 |
       3383
                                      Processing response from /127.0.0.4
 [SharedPool-Worker-2] | 2014-11-08 08:06:43.958000 | 127.0.0.1 |
  3612
                             Executing single-partition query on demo_table
 [SharedPool-Worker-1] | 2014-11-08 08:06:43.959000 | 127.0.0.5 |
  1462
```

```
                                                    Acquiring sstable references
 [SharedPool-Worker-1] | 2014-11-08 08:06:43.959000 | 127.0.0.5 |
  1509
                                                   Merging memtable tombstones
 [SharedPool-Worker-1] | 2014-11-08 08:06:43.959000 | 127.0.0.5 |
  1569
 Skipped 0/0 non-slice-intersecting sstables, included 0 due to tombstones
 [SharedPool-Worker-1] | 2014-11-08 08:06:43.959000 | 127.0.0.5 |
  1662
                                 Merging data from memtables and 0 sstables
 [SharedPool-Worker-1] | 2014-11-08 08:06:43.959000 | 127.0.0.5 |
  1681
                                           Read 1 live and 0 tombstoned cells
 [SharedPool-Worker-1] | 2014-11-08 08:06:43.959000 | 127.0.0.5 |
  1760
                                             Enqueuing response to /127.0.0.1
 [SharedPool-Worker-1] | 2014-11-08 08:06:43.959001 | 127.0.0.5 |
  2104
                                                          Message received
 from /127.0.0.5 [Thread-10] | 2014-11-08 08:06:43.960000 | 127.0.0.1 |
        5330
                                                    Sending message
 to /127.0.0.1 [WRITE-/127.0.0.1] | 2014-11-08 08:06:43.960000 | 127.0.0.5
 |           2423
                                               Processing response from /127.0.0.5
 [SharedPool-Worker-2] | 2014-11-08 08:06:43.960000 | 127.0.0.1 |
  5519

      Request complete | 2014-11-08 08:06:43.960947 | 127.0.0.1 |
  5947
```

7. Change the consistency level to ALL and run the SELECT statement again.

```
cqlsh:demo_cl> CONSISTENCY ALL;
cqlsh:demo_cl> SELECT * FROM demo_table WHERE id = 0;
```

```
 id | col1 | col2
----+------+------
  0 |    0 |    0

(1 rows)


Tracing session: 4da75ca0-6761-11e4-96a3-fd07420471ed

 activity
                             | timestamp                  | source    |
 source_elapsed
-------------------------------------------------------------------------------------------
+----------------------------+-----------+----------------

    Execute CQL3 query | 2014-11-08 08:07:14.026000 | 127.0.0.1 |
     0
                  Parsing SELECT * FROM demo_table WHERE id = 0 LIMIT 10000;
 [SharedPool-Worker-1] | 2014-11-08 08:07:14.026000 | 127.0.0.1 |
    73
                                                       Preparing statement
 [SharedPool-Worker-1] | 2014-11-08 08:07:14.026000 | 127.0.0.1 |
   271
                                                    Sending message
 to /127.0.0.4 [WRITE-/127.0.0.4] | 2014-11-08 08:07:14.027000 | 127.0.0.1
 |            978
```

```
                                                              Message received
from /127.0.0.1 [Thread-9] | 2014-11-08 08:07:14.027000 | 127.0.0.5 |
        56
                                                            Sending message
to /127.0.0.5 [WRITE-/127.0.0.5] | 2014-11-08 08:07:14.027000 | 127.0.0.1
|           1012
                                Executing single-partition query on demo_table
[SharedPool-Worker-2] | 2014-11-08 08:07:14.027000 | 127.0.0.3 |
  253
                                                            Sending message
to /127.0.0.3 [WRITE-/127.0.0.3] | 2014-11-08 08:07:14.027000 | 127.0.0.1
|           1054
                                                         Acquiring sstable references
[SharedPool-Worker-2] | 2014-11-08 08:07:14.027000 | 127.0.0.3 |
  275
                                                          Merging memtable tombstones
[SharedPool-Worker-2] | 2014-11-08 08:07:14.027000 | 127.0.0.3 |
  344
Skipped 0/0 non-slice-intersecting sstables, included 0 due to tombstones
[SharedPool-Worker-2] | 2014-11-08 08:07:14.028000 | 127.0.0.3 |
  438
                                                         Acquiring sstable references
[SharedPool-Worker-1] | 2014-11-08 08:07:14.028000 | 127.0.0.5 |
  461
                                                          Merging memtable tombstones
[SharedPool-Worker-1] | 2014-11-08 08:07:14.028000 | 127.0.0.5 |
  525
Skipped 0/0 non-slice-intersecting sstables, included 0 due to tombstones
[SharedPool-Worker-1] | 2014-11-08 08:07:14.028000 | 127.0.0.5 |
  622
                                  Merging data from memtables and 0 sstables
[SharedPool-Worker-1] | 2014-11-08 08:07:14.028000 | 127.0.0.5 |
  645
                                           Read 1 live and 0 tombstoned cells
[SharedPool-Worker-1] | 2014-11-08 08:07:14.028000 | 127.0.0.5 |
  606
                                             Enqueuing response to /127.0.0.1
[SharedPool-Worker-1] | 2014-11-08 08:07:14.028001 | 127.0.0.5 |
 1125
                                                              Message received
from /127.0.0.3 [Thread-8] | 2014-11-08 08:07:14.029000 | 127.0.0.1 |
      3224
                                                            Sending message
to /127.0.0.1 [WRITE-/127.0.0.1] | 2014-11-08 08:07:14.029000 | 127.0.0.5
|           1616
                                         Processing response from /127.0.0.3
[SharedPool-Worker-3] | 2014-11-08 08:07:14.029000 | 127.0.0.1 |
 3417
                                                           Message received
from /127.0.0.5 [Thread-10] | 2014-11-08 08:07:14.029000 | 127.0.0.1 |
       3454
                                                              Message received
from /127.0.0.4 [Thread-7] | 2014-11-08 08:07:14.029000 | 127.0.0.1 |
      3516
                                         Processing response from /127.0.0.5
[SharedPool-Worker-2] | 2014-11-08 08:07:14.029000 | 127.0.0.1 |
 3627
                                         Processing response from /127.0.0.4
[SharedPool-Worker-2] | 2014-11-08 08:07:14.030000 | 127.0.0.1 |
 3688

     Request complete | 2014-11-08 08:07:14.030347 | 127.0.0.1 |
 4347
```

## How consistency affects performance

Changing the consistency level can affect read performance. The tracing output shows that as you change the consistency level from ONE to QUORUM to ALL, performance degrades in from 2585 to 2998 to 5219 microseconds, respectively. If you follow the steps in this tutorial, it is not guaranteed that you will see the same trend because querying a one-row table is a degenerate case, used for example purposes. The difference between QUORUM and ALL is slight in this case, so depending on conditions in the cluster, performance using ALL might be faster than QUORUM.

Under the following conditions, performance using ALL is worse than QUORUM:

- The data consists of thousands of rows or more.
- One node is slower than others.
- A particularly slow node was not selected to be part of the quorum.

### Tracing queries on large datasets

You can use probabilistic tracing on databases having at least ten rows, but this capability is intended for tracing through much more data. After configuring probabilistic tracing using the `nodetool settraceprobability` command, you query the system_traces keyspace.

```
SELECT * FROM system_traces.events;
```

# CQL reference

## Introduction

All of the commands included in the CQL language are available on the cqlsh command line. There are a group of commands that are available on the command line, but are not support by the CQL language. These commands are called cqlsh commands. You can run cqlsh commands from the command line only. You can run CQL commands in a number of ways.

This reference covers CQL and cqlsh based on the CQL specification 3.1.0-3.1.6. CQL 2 is deprecated and removal is planned for Cassandra 3.0.

## CQL lexical structure

CQL input consists of statements. Like SQL, statements change data, look up data, store data, or change the way data is stored. Statements end in a semicolon (;).

For example, the following is valid CQL syntax:

```
SELECT * FROM MyTable;

UPDATE MyTable
  SET SomeColumn = 'SomeValue'
  WHERE columnName = B70DE1D0-9908-4AE3-BE34-5573E5B09F14;
```

This is a sequence of two CQL statements. This example shows one statement per line, although a statement can usefully be split across lines as well.

# Uppercase and lowercase

Keyspace, column, and table names created using CQL are case-insensitive unless enclosed in double quotation marks. If you enter names for these objects using any uppercase letters, Cassandra stores the names in lowercase. You can force the case by using double quotation marks. For example:

```
CREATE TABLE test (
  Foo int PRIMARY KEY,
  "Bar" int
);
```

The following table shows partial queries that work and do not work to return results from the test table:

**Table: What Works and What Doesn't**

| Queries that Work | Queries that Don't Work |
|---|---|
| SELECT foo FROM . . . | SELECT "Foo" FROM . . . |
| SELECT Foo FROM . . . | SELECT "BAR" FROM . . . |
| SELECT FOO FROM . . . | SELECT bar FROM . . . |
| SELECT "foo" FROM . . . | SELECT Bar FROM . . . |
| SELECT "Bar" FROM . . . | |

SELECT "foo" FROM ... works because internally, Cassandra stores foo in lowercase. The double-quotation mark character can be used as an escape character for the double quotation mark.

Case sensitivity rules in earlier versions of CQL apply when handling legacy tables.

CQL keywords are case-insensitive. For example, the keywords SELECT and select are equivalent. This document shows keywords in uppercase.

# Valid characters for data types and keyspace/table/column names

Keyspace, column, and table names created using CQL can only contain alphanumeric and underscore characters. User-defined data type names and field names, user-defined function names, and user-defined aggregate names created using CQL can only contain alphanumeric and underscore characters. If you enter names for these objects using anything other than alphanumeric characters or underscores, Cassandra will issue an invalid syntax message and fail to create the object.

**Table: What Works and What Doesn't**

| Creations that Work | Creations that Don't Work |
|---|---|
| CREATE TABLE foo ... | CREATE TABLE foo!$% ... |
| CREATE TABLE foo_bar ... | CREATE TABLE foo[]"90 ... |
| ALTER TABLE foo5 ... | ALTER TABLE foo5$$ |
| CREATE FUNCTION foo5 ... | CREATE FUNCTION foo5$$ |
| CREATE AGGREGATE foo5 ... | CREATE AGGREGATE foo5$$ |
| CREATE TYPE foo5 (bar9 text, ... | CREATE TYPE foo5$$ (bar#9 int ... |

# Escaping characters

Column names that contain characters that CQL cannot parse need to be enclosed in double quotation marks in CQL.

Dates, IP addresses, and strings need to be enclosed in single quotation marks. To use a single quotation mark itself in a string literal, escape it using a single quotation mark.

# Valid literals

Valid literal consist of these kinds of values:

- blob

  hexadecimal defined as 0[xX](hex)+
- boolean

  true or false, case-insensitive, not enclosed in quotation marks
- numeric constant

  A numeric constant can consist of integers 0-9 and a minus sign prefix. A numeric constant can also be float. A float can be a series of one or more decimal digits, followed by a period, ., and one or more decimal digits. There is no optional + sign. The forms .42 and 42 are unacceptable. You can use leading or trailing zeros before and after decimal points. For example, 0.42 and 42.0. A float constant, expressed in E notation, consists of the characters in this regular expression:

  ```
  '-'?[0-9]+('.'[0-9]*)?([eE][+-]?[0-9+])?
  ```

  NaN and Infinity are floats.
- identifier

  A letter followed by any sequence of letters, digits, or the underscore. Names of tables, columns, and other objects are identifiers and enclosed in double quotation marks.
- integer

  An optional minus sign, -, followed by one or more digits.
- string literal

  Characters enclosed in single quotation marks. To use a single quotation mark itself in a string literal, escape it using a single quotation mark. For example, use '' to make dog plural: dog''s.
- uuid

  32 hex digits, 0-9 or a-f, which are case-insensitive, separated by dashes, -, after the 8th, 12th, 16th, and 20th digits. For example: 01234567-0123-0123-0123-0123456789ab
- timeuuid

  Uses the time in 100 nanosecond intervals since 00:00:00.00 UTC (60 bits), a clock sequence number for prevention of duplicates (14 bits), plus the IEEE 801 MAC address (48 bits) to generate a unique identifier. For example: d2177dd0-eaa2-11de-a572-001b779c76e3
- whitespace

  Separates terms and used inside string literals, but otherwise CQL ignores whitespace.

# Exponential notation

Cassandra supports exponential notation. This example shows exponential notation in the output from a cqlsh command.

```
CREATE TABLE test(
  id varchar PRIMARY KEY,
```

```
  value_double double,
  value_float float
);

INSERT INTO test (id, value_float, value_double)
  VALUES ('test1', -2.6034345E+38, -2.6034345E+38);

SELECT * FROM test;
```

```
 id    | value_double | value_float
-------+--------------+-------------
 test1 |  -2.6034e+38 | -2.6034e+38
```

# CQL code comments

You can use the following notation to include comments in CQL code:

- Double hypen

  ```
  -- Single-line comment
  ```

- Double forward slash

  ```
  //Single-line comment
  ```

- Forward slash asterisk

  ```
  /* Multi-line comment */
  ```

# CQL Keywords

This table lists keywords and whether or not the words are reserved. A reserved keyword cannot be used as an identifier unless you enclose the word in double quotation marks. Non-reserved keywords have a specific meaning in certain context but can be used as an identifier outside this context.

**Table: Keywords**

| Keyword | Reserved |
|---|---|
| ADD | yes |
| ALL | no |
| ALLOW | yes |
| ALTER | yes |
| AND | yes |
| ANY | yes |
| APPLY | yes |
| AS | no |
| ASC | yes |
| ASCII | no |
| AUTHORIZE | yes |

| Keyword | Reserved |
|---|---|
| BATCH | yes |
| BEGIN | yes |
| BIGINT | no |
| BLOB | no |
| BOOLEAN | no |
| BY | yes |
| CLUSTERING | no |
| COLUMNFAMILY | yes |
| COMPACT | no |
| CONSISTENCY | no |
| COUNT | no |
| COUNTER | no |
| CREATE | yes |
| CUSTOM | no |
| DECIMAL | no |
| DELETE | yes |
| DESC | yes |
| DISTINCT | no |
| DOUBLE | no |
| DROP | yes |
| EACH_QUORUM | yes |
| EXISTS | no |
| FILTERING | no |
| FLOAT | no |
| FROM | yes |
| FROZEN | no |
| GRANT | yes |
| IF | yes |
| IN | yes |
| INDEX | yes |
| INET | yes |
| INFINITY | yes |
| INSERT | yes |
| INT | no |

| Keyword | Reserved |
|---|---|
| INTO | yes |
| KEY | no |
| KEYSPACE | yes |
| KEYSPACES | yes |
| LEVEL | no |
| LIMIT | yes |
| LIST | no |
| LOCAL_ONE | yes |
| LOCAL_QUORUM | yes |
| MAP | no |
| MODIFY | yes |
| NAN | yes |
| NORECURSIVE | yes |
| NOSUPERUSER | no |
| NOT | yes |
| OF | yes |
| ON | yes |
| ONE | yes |
| ORDER | yes |
| PASSWORD | yes |
| PERMISSION | no |
| PERMISSIONS | no |
| PRIMARY | yes |
| QUORUM | yes |
| RENAME | yes |
| REVOKE | yes |
| SCHEMA | yes |
| SELECT | yes |
| SET | yes |
| STATIC | no |
| STORAGE | no |
| SUPERUSER | no |
| TABLE | yes |
| TEXT | no |

| Keyword | Reserved |
|---|---|
| TIMESTAMP | no |
| TIMEUUID | no |
| THREE | yes |
| TO | yes |
| TOKEN | yes |
| TRUNCATE | yes |
| TTL | no |
| TUPLE | no |
| TWO | yes |
| TYPE | no |
| UNLOGGED | yes |
| UPDATE | yes |
| USE | yes |
| USER | no |
| USERS | no |
| USING | yes |
| UUID | no |
| VALUES | no |
| VARCHAR | no |
| VARINT | no |
| WHERE | yes |
| WITH | yes |
| WRITETIME | no |

# CQL data types

CQL defines built-in data types for columns. The counter type is unique.

**Table: CQL Data Types**

| CQL Type | Constants | Description |
|---|---|---|
| ascii | strings | US-ASCII character string |
| bigint | integers | 64-bit signed long |
| blob | blobs | Arbitrary bytes (no validation), expressed as hexadecimal |
| boolean | booleans | true or false |
| counter | integers | Distributed counter value (64-bit long) |

| CQL Type | Constants | Description |
|---|---|---|
| decimal | integers, floats | Variable-precision decimal<br>Java type<br>**Note:** When dealing with currency, it is a best practice to have a currency class that serializes to and from an int or use the Decimal form. |
| double | integers, floats | 64-bit IEEE-754 floating point<br>Java type |
| float | integers, floats | 32-bit IEEE-754 floating point<br>Java type |
| frozen | tuples, collections, user-defined types | A frozen value serializes multiple components into a single value. Non-frozen types allow updates to individual fields. Cassandra treats the value of a frozen type as a blob. The entire value must be overwritten.<br>**Note:** Cassandra 2.1.0 to 2.1.2 requires using frozen for tuples:<br>`frozen <tuple <int, tuple<text, double>>>`<br>Cassandra 2.1.3+ does not require this keyword for tuples. |
| inet | strings | IP address string in IPv4 or IPv6 format, used by the python-cql driver and CQL native protocols |
| int | integers | 32-bit signed integer |
| list | n/a | A collection of one or more ordered elements |
| map | n/a | A JSON-style array of literals: { literal : literal, literal : literal ... } |
| set | n/a | A collection of one or more elements |
| text | strings | UTF-8 encoded string |
| timestamp | integers, strings | Date plus time, encoded as 8 bytes since epoch |
| timeuuid | uuids | Type 1 UUID only |
| tuple | n/a | Cassandra 2.1 and later. A group of 2-3 fields. |
| uuid | uuids | A UUID in standard UUID format |
| varchar | strings | UTF-8 encoded string |
| varint | integers | Arbitrary-precision integer<br>Java type |

In addition to the CQL types listed in this table, you can use a string containing the name of a JAVA class (a sub-class of AbstractType loadable by Cassandra) as a CQL type. The class name should either be fully qualified or relative to the org.apache.cassandra.db.marshal package.

Enclose ASCII text, timestamp, and inet values in single quotation marks. Enclose names of a keyspace, table, or column in double quotation marks.

### Java types

The Java types, from which most CQL types are derived, are obvious to Java programmers. The derivation of the following types, however, might not be obvious:

**Table: Derivation of selective CQL types**

| CQL type | Java type |
|---|---|
| decimal | java.math.BigDecimal |
| float | java.lang.Float |
| double | java.lang.Double |
| varint | java.math.BigInteger |

## Blob type

The Cassandra blob data type represents a constant hexadecimal number defined as 0[xX](hex)+ where hex is an hexadecimal character, such as [0-9a-fA-F]. For example, 0xcafe. The maximum theoretical size for a blob is 2GB. The practical limit on blob size, however, is less than 1 MB, ideally even smaller. A blob type is suitable for storing a small image or short string.

### Blob conversion functions

These functions convert the native types into binary data (blob):

- typeAsBlob(type)
- blobAsType

For every native, nonblob type supported by CQL, the typeAsBlob function takes a argument of type type and returns it as a blob. Conversely, the blobAsType function takes a 64-bit blob argument and converts it to a bigint value.

This example shows how to use bitintAsBlob:

```
CREATE TABLE bios ( user_name varchar PRIMARY KEY,
   bio blob
 );

INSERT INTO bios (user_name, bio) VALUES ('fred', bigintAsBlob(3));

SELECT * FROM bios;

 user_name | bio
-----------+--------------------
      fred | 0x0000000000000003
```

This example shows how to use blobAsBigInt.

```
ALTER TABLE bios ADD id bigint;

INSERT INTO bios (user_name, id) VALUES ('fred',
 blobAsBigint(0x0000000000000003));

SELECT * FROM bios;
```

```
 user_name | bio                 | id
-----------+---------------------+----
      fred | 0x0000000000000003 |  3
```

# Collection type

A collection column is declared using the collection type, followed by another type, such as int or text, in angle brackets. For example, you can create a table having a list of textual elements, a list of integers, or a list of some other element types.

```
list<text>
list<int>
```

Collection types cannot currently be nested. For example, you cannot define a list within a list:

```
list<list<int>>     // not allowed
```

In Cassandra 2.1 and later, you can create an index on a column of type map, set, or list.

## Using frozen in a collection

A frozen value serializes multiple components into a single value. Non-frozen types allow updates to individual fields. Cassandra treats the value of a frozen type as a blob. The entire value must be overwritten.

**Note:** You cannot use non-frozen collections for primary key columns. However, you can use frozen collections for primary key columns.

```
column_name <collection_type><cql_type, frozen<column_name>>
```

For example:

```
CREATE TABLE mykeyspace.users (
  id uuid PRIMARY KEY,
  name frozen <fullname>,
  direct_reports set<frozen <fullname>>,     // a collection set
  addresses map<text, frozen <address>>     // a collection map
);
```

# Counter type

A counter column value is a 64-bit signed integer. You cannot set the value of a counter, which supports two operations: increment and decrement.

Use counter types as described in the "Using a counter" section. Do not assign this type to a column that serves as the primary key or partition key. Also, do not use the counter type in a table that contains anything other than counter types and the primary key. To generate sequential numbers for surrogate keys, use the timeuuid type instead of the counter type. You cannot create an index on a counter column or set data in a counter column to expire using the Time-To-Live (TTL) property.

# UUID and timeuuid types

The UUID (universally unique id) comparator type is used to avoid collisions in column names. Alternatively, you can use the timeuuid.

Timeuuid types can be entered as integers for CQL input. A value of the timeuuid type is a Version 1 UUID. A Version 1 UUID includes the time of its generation and are sorted by timestamp, making them ideal for use in applications requiring conflict-free timestamps. For example, you can use this type to identify a column (such as a blog entry) by its timestamp and allow multiple clients to write to the same partition key simultaneously. Collisions that would potentially overwrite data that was not intended to be overwritten cannot occur.

A valid timeuuid conforms to the timeuuid format shown in valid literals.

## uuid and Timeuuid functions

Cassandra 2.0.7 and later includes the uuid() function. This function takes no parameters and generates a random Type 4 UUID suitable for use in INSERT or SET statements.

Several Timeuuid functions are designed for use with the timeuuid type:

- dateOf()

  Used in a SELECT clause, this function extracts the timestamp of a timeuuid column in a resultset. This function returns the extracted timestamp as a date. Use unixTimestampOf() to get a raw timestamp.
- now()

  In the coordinator node, generates a new unique timeuuid in milliseconds when the statement is executed. The timestamp portion of the timeuuid conforms to the UTC (Universal Time) standard. This method is useful for inserting values. The value returned by now() is guaranteed to be unique.
- minTimeuuid() and maxTimeuuid()

  Returns a UUID-like result given a conditional time component as an argument. For example:

  ```
  SELECT * FROM myTable
     WHERE t > maxTimeuuid('2013-01-01 00:05+0000')
     AND t < minTimeuuid('2013-02-02 10:00+0000')
  ```
- unixTimestampOf()

  Used in a SELECT clause, this functions extracts the timestamp in milliseconds of a timeuuid column in a resultset. Returns the value as a raw, 64-bit integer timestamp.

The min/maxTimeuuid example selects all rows where the timeuuid column, t, is strictly later than 2013-01-01 00:05+0000 but strictly earlier than 2013-02-02 10:00+0000. The t >= maxTimeuuid('2013-01-01 00:05+0000') does not select a timeuuid generated exactly at 2013-01-01 00:05+0000 and is essentially equivalent to t > maxTimeuuid('2013-01-01 00:05+0000').

The values returned by minTimeuuid and maxTimeuuid functions are not true UUIDs in that the values do not conform to the Time-Based UUID generation process specified by the RFC 4122. The results of these functions are deterministic, unlike the now function.

## Timestamp type

Values for the timestamp type are encoded as 64-bit signed integers representing a number of milliseconds since the standard base time known as the epoch: January 1 1970 at 00:00:00 GMT. A timestamp type can be entered as an integer for CQL input, or as a string literal in any of the following ISO 8601 formats:

```
yyyy-mm-dd HH:mm
yyyy-mm-dd HH:mm:ss
yyyy-mm-dd HH:mmZ
yyyy-mm-dd HH:mm:ssZ
yyyy-mm-dd'T'HH:mm
yyyy-mm-dd'T'HH:mmZ
yyyy-mm-dd'T'HH:mm:ss
```

```
yyyy-mm-dd'T'HH:mm:ssZ
yyyy-mm-dd
yyyy-mm-ddZ
```

where Z is the RFC-822 4-digit time zone, expressing the time zone's difference from UTC. For example, for the date and time of Jan 2, 2003, at 04:05:00 AM, GMT:

```
2011-02-03 04:05+0000
2011-02-03 04:05:00+0000
2011-02-03T04:05+0000
2011-02-03T04:05:00+0000
```

If no time zone is specified, the time zone of the Cassandra coordinator node handing the write request is used. For accuracy, DataStax recommends specifying the time zone rather than relying on the time zone configured on the Cassandra nodes.

If you only want to capture date values, the time of day can also be omitted. For example:

```
2011-02-03
2011-02-03+0000
```

In this case, the time of day defaults to 00:00:00 in the specified or default time zone.

Timestamp output appears in the following format by default:

```
yyyy-mm-dd HH:mm:ssZ
```

You can change the format by setting the time_format property in the [ui] section of the cqlshrc file.

# Tuple type

Cassandra 2.1 introduces the tuple type that holds fixed-length sets of typed positional fields. You can use a tuple as an alternative to a user-defined type when you don't need to add new fields. A tuple can accommodate many fields (32768), more than you can prudently use. Typically, you create a tuple having only a few fields.

In the table creation statement, use angle brackets and a comma delimiter to declare the tuple component types. Surround tuple values in parentheses to insert the values into a table, as shown in this example.

```
CREATE TABLE collect_things (
  k int PRIMARY KEY,
  v <tuple<int, text, float>>
);

INSERT INTO collect_things (k, v) VALUES(0, (3, 'bar', 2.1));

SELECT * FROM collect_things;

 k | v
---+-----------------
 0 | (3, 'bar', 2.1)
```

**Note:** Cassandra 2.1.0 to 2.1.2 requires using frozen for tuples:

```
frozen <tuple <int, tuple<text, double>>>
```

Cassandra 2.1.3+ does not require this keyword for tuples.

You can filter a selection using a tuple.

```
CREATE INDEX on collect_things (v);
```

```
SELECT * FROM collect_things WHERE v = (3, 'bar', 2.1);

 k | v
---+-----------------
 0 | (3, 'bar', 2.1)
```

You can nest tuples as shown in the following example:

```
create table nested (k int PRIMARY KEY, t frozen <tuple <int, tuple<text,
 double>>>);

INSERT INTO nested (k, t) VALUES (0, (3, ('foo', 3.4)));
```

# User-defined type

Cassandra 2.1 supports user-defined types. A user-defined type facilitates handling multiple fields of related information in a table. Applications that required multiple tables can be simplified to use fewer tables by using a user-defined type to represent the related fields of information instead of storing the information in a separate table. The address type example demonstrates how to use a user-defined type.

You can create, alter, and drop a user-defined type using these commands:

- CREATE TYPE
- ALTER TYPE
- DROP TYPE

The cqlsh utility includes these commands for describing a user-defined type or listing all user-defined types:

- DESCRIBE TYPE
- DESCRIBE TYPES

The scope of a user-defined type is the keyspace in which you define it. Use dot notation to access a type from a keyspace outside its scope: keyspace name followed by a period followed the name of the type. Cassandra accesses the type in the specified keyspace, but does not change the current keyspace; otherwise, if you do not specify a keyspace, Cassandra accesses the type within the current keyspace.

# CQL keyspace and table properties

The CQL WITH clause specifies keyspace and table properties in these CQL commands:

- ALTER KEYSPACE
- ALTER TABLE
- CREATE KEYSPACE
- CREATE TABLE

## CQL keyspace properties

CQL supports setting the following keyspace properties in addition to naming data centers.

- class

  The name of the replication strategy: SimpleStrategy or NetworkTopologyStrategy. You set the replication factor independently for each data center.
- replication_factor

  The replication_factor property is used only when specifying the SimpleStrategy, as shown in CREATE KEYSPACE examples. The replication factor value is the total number of replicas across the cluster.

For production use or for use with mixed workloads, create the keyspace using NetworkTopologyStrategy. SimpleStrategy is fine for evaluation purposes. NetworkTopologyStrategy is recommended for most deployments because it is much easier to expand to multiple data centers when required by future expansion.

You can also configure the durable writes property when creating or altering a keyspace.

# Table properties

CQL supports Cassandra table properties, such as comments and compaction options, listed in the following table.

In CQL commands, such as CREATE TABLE, you format properties in either the name-value pair or collection map format. The name-value pair property syntax is:

```
name = value AND name = value
```

The collection map format, used by compaction and compression properties, is:

```
{ name : value, name : value, name : value ... }
```

Enclose properties that are strings in single quotation marks.

See CREATE TABLE for examples.

**Table: CQL properties**

| CQL property | Default | Description |
|---|---|---|
| bloom_filter_fp_chance | 0.01 for `SizeTieredCompactionStrategy` and DateTieredCompactionStrategy, 0.1 for `LeveledCompactionStrategy` | Desired false-positive probability for SSTable Bloom filters. See Bloom filter below. |
| caching | Cassandra 2.1:<br>ALL for keys<br>NONE for rows_per_partition<br>Cassandra 2.0.x: keys_only | Optimizes the use of cache memory without manual tuning. See caching below. |
| comment | N/A | A human readable comment describing the table. See comments below. |
| compaction | `SizeTieredCompactionStrategy` | Sets the compaction strategy for the table. See compaction below. |
| compression | `LZ4Compressor` | The compression algorithm to use. Valid values are `LZ4Compressor`), `SnappyCompressor`, and `DeflateCompressor`. See compression below. |
| dclocal_read_repair_chance | 0.1 (Cassandra 2.1, Cassandra 2.0.9 and later)<br>0.0 (Cassandra 2.0.8 and earlier) | Specifies the probability of read repairs being invoked over all replicas in the current data center. |

| CQL property | Default | Description |
|---|---|---|
| default_time_to_live | 0 | The default expiration time in seconds for a table. Used in MapReduce scenarios when you have no control of TTL. |
| gc_grace_seconds | 864000 [10 days] | Specifies the time to wait before garbage collecting tombstones (deletion markers). The default value allows a great deal of time for consistency to be achieved prior to deletion. In many deployments this interval can be reduced, and in a single-node cluster it can be safely set to zero. |
| min_index_interval, max_index_interval (Cassandra 2.1.x) or index_interval (Cassandra 2.0.x) | min_index_interval 128 and max_index_interval 2048, or index_interval 128 | To control the sampling of entries from the partition index, configure the sample frequency of the partition summary by changing these properties. See min_index_interval and max_index_interval below. |
| memtable_flush_period_in_ms | 0 | Forces flushing of the memtable after the specified time in milliseconds elapses. |
| populate_io_cache_on_flush (Cassandra 2.0.x only) | false | Adds newly flushed or compacted SSTables to the operating system page cache, potentially evicting other cached data to make room. Enable when all data in the table is expected to fit in memory. You can also configure the global compaction_preheat_key_cache option in the cassandra.yaml file. |
| read_repair_chance | 0.0 (Cassandra 2.1, Cassandra 2.0.9 and later) 0.1 (Cassandra 2.0.8 and earlier) | Specifies the basis for invoking read repairs on reads in clusters. The value must be between 0 and 1. |
| replicate_on_write (Cassandra 2.0.x only) | true | Removed in Cassandra 2.1. Applies only to counter tables. When set to true, replicates writes to all affected replicas regardless of the consistency level specified by the client for a write request. For counter tables, this should always be set to true. |
| speculative_retry | 99percentile Cassandra 2.0.2 and later | Overrides normal read timeout when read_repair_chance is not 1.0, sending another request to read. See speculative retry below. |

## Bloom filter

The Bloom filter property is the desired false-positive probability for SSTable Bloom filters. When data is requested, the Bloom filter checks if the row exists before doing disk I/O. Bloom filter property value ranges from 0 to 1.0. The effects of the minimum and maximum values are:

0 Enables the unmodified, effectively the largest possible, Bloom filter

1.0 Disables the Bloom Filter

The recommended setting is 0.1. A higher value yields diminishing returns.

## caching

Caching optimizes the use of cache memory without manual tuning. You set table properties to configure caching when you create or alter the table. Cassandra weights the cached data by size and access frequency. After configuring the caching table property, configure the global caching properties in the

`cassandra.yaml` file. For information about global caching properties, see Cassandra 2.1 documentation or Cassandra 2.0 documentation.

**Cassandra 2.1**

Configure the cache by creating a property map of values for the caching property. Options are:

- keys: ALL or NONE
- rows_per_partition: number of CQL rows, NONE, or ALL

For example:

```
CREATE TABLE DogTypes (
  block_id uuid,
  species text,
  alias text,
  population varint,
  PRIMARY KEY (block_id)
) WITH caching = { 'keys' : 'NONE', 'rows_per_partition' : '120' };
```

**Cassandra 2.0**

Configure the cache using one of these caching property options:

- all
- keys_only
- rows_only
- none

You can specify a key or row cache, or specify both key and row caches using the options. For example:

```
// Cassandra 2.0.x only

CREATE TABLE DogTypes (
  block_id uuid,
  species text,
  alias text,
  population varint,
  PRIMARY KEY (block_id)
) WITH caching = 'keys_only';
```

**Important:** In Cassandra 2.0.x, use row caching with caution.

## comments

Comments can be used to document CQL statements in your application code. Single line comments can begin with a double dash (--) or a double slash (//) and extend to the end of the line. Multi-line comments can be enclosed in /* and */ characters.

## compaction

The compaction property defines the compaction strategy class to use. Choose the compaction strategy that best fits your data and environment:

**Note:** For more guidance, see the When to Use Leveled Compaction, Leveled Compaction in Apache Cassandra blog.

- `SizeTieredCompactionStrategy (STCS)`: The default compaction strategy. This strategy triggers a minor compaction when there are a number of similar sized SSTables on disk as configured by the table subproperty, min_threshold. A minor compaction does not involve all the tables in a keyspace. Also see STCS compaction subproperties.

- `DateTieredCompactionStrategy (DTCS)`: Available in Cassandra 2.0.11 and 2.1.1 and later. This strategy is particularly useful for time series data. DateTieredCompactionStrategy stores data written within a certain period of time in the same SSTable. For example, Cassandra can store your last hour of data in one SSTable *time window*, and the next 4 hours of data in another time window, and so on. Compactions are triggered when the min_threshold (4 by default) for SSTables in those windows is reached. The most common queries for time series workloads retrieve the last hour/day/month of data. Cassandra can limit SSTables returned to those having the relevant data. Also, Cassandra can store data that has been set to expire using TTL in an SSTable with other data scheduled to expire at approximately the same time. Cassandra can then drop the SSTable without doing any compaction. Also see DTCS compaction subproperties and DateTieredCompactionStrategy: Compaction for Time Series Data.

  **Note:** When using DTCS disabling read repair is recommended. Use full repair as necessary.
- `LeveledCompactionStrategy (LCS)`: The leveled compaction strategy creates SSTables of a fixed, relatively small size (160 MB by default) that are grouped into levels. Within each level, SSTables are guaranteed to be non-overlapping. Each level (L0, L1, L2 and so on) is 10 times as large as the previous. Disk I/O is more uniform and predictable on higher than on lower levels as SSTables are continuously being compacted into progressively larger levels. At each level, row keys are merged into non-overlapping SSTables. This can improve performance for reads, because Cassandra can determine which SSTables in each level to check for the existence of row key data. This compaction strategy is modeled after Google's leveldb implementation. Also see LCS compaction subproperties.

Hybrid (leveled and size-tiered) compaction improvements to the leveled compaction strategy reduce the performance overhead on read operations when compaction cannot keep pace with write-heavy workload. When using the LCS, if Cassandra cannot keep pace with the workload, the compaction strategy switches to STCS until Cassandra catches up. For this reason, it is a best practice to configure the max_threshold subproperty for a table to use when the switch occurs.

You can specify a custom strategy. Use the full class name as a string constant.

## compression

To configure compression, choose the `LZ4Compressor`, `SnappyCompressor`, or `DeflateCompressor` property to use in creating or altering a table. Use an empty string ('') to disable compression, as shown in the example of how to use subproperties. Choosing the right compressor depends on your requirements for space savings over read performance. LZ4 is fastest to decompress, followed by Snappy, then by Deflate. Compression effectiveness is inversely correlated with decompression speed. The extra compression from Deflate or Snappy is not enough to make up for the decreased performance for general-purpose workloads, but for archival data they may be worth considering. Developers can also implement custom compression classes using the `org.apache.cassandra.io.compress.ICompressor` interface. Specify the full class name enclosed in single quotation marks. Also use the compression subproperties.

## min_index_interval and max_index_interval

The index_interval (Cassandra 2.0.x) property or the min_index_interval and max_index_interval (Cassandra 2.1) properties control the sampling of entries from the primary row index, configure sample frequency of the partition summary by changing the index interval. After changing the index interval, SSTables are written to disk with new information. The interval corresponds to the number of index entries that are skipped between taking each sample. By default Cassandra samples one row key out of every 128. The larger the interval, the smaller and less effective the sampling. The larger the sampling, the more effective the index, but with increased memory usage. In Cassandra 2.0.x, generally, the best trade off between memory usage and performance is a value between 128 and 512 in combination with a large table key cache. However, if you have small rows (many to an OS page), you may want to increase the sample size, which often lowers memory usage without an impact on performance. For large rows, decreasing the sample size may improve read performance.

In Cassandra 2.1, the name index_interval is replaced by min_index_interval and max_index_interval. The max_index_interval is 2048 by default. The default would be reached only when SSTables are infrequently-read and the index summary memory pool is full. When upgrading from earlier releases, Cassandra uses the old index_interval value for the min_index_interval.

### speculative retry

To override normal read timeout when read_repair_chance is not 1.0, sending another request to read, choose one of these values and use the property to create or alter the table:

- ALWAYS: Retry reads of all replicas.
- Xpercentile: Retry reads based on the effect on throughput and latency.
- Yms: Retry reads after specified milliseconds.
- NONE: Do not retry reads.

Using the speculative retry property, you can configure rapid read protection in Cassandra 2.0.2 and later. Use this property to retry a request after some milliseconds have passed or after a percentile of the typical read latency has been reached, which is tracked per table. For example:

```
ALTER TABLE users WITH speculative_retry = '10ms';
```

Or:

```
ALTER TABLE users WITH speculative_retry = '99percentile';
```

**Related information**

Cassandra 2.0 cassandra.yaml

Cassandra 2.1 cassandra.yaml

## Compaction subproperties

Using CQL, you can configure a table to use SizeTieredCompactionStrategy (STCS), DateTieredCompactionStrategy (DTCS), and LeveledCompactionStrategy (LCS). DateTieredCompactionStrategy is available in Cassandra 2.0.11 and later. You construct a map of the compaction property and the following subproperties:

**CQL Compaction Subproperties for STCS**

| Compaction Subproperties | Default | Description |
|---|---|---|
| bucket_high | 1.5 | Size-tiered compaction considers SSTables to be within the same bucket if the SSTable size diverges by 50% or less from the default bucket_low and default bucket_high values: [average-size × bucket_low, average-size × bucket_high]. |
| bucket_low | 0.5 | Same as above. |
| cold_reads_to_omit | 0.05 | The maximum percentage of reads/sec that ignored SSTables may account for. The recommended range of values is 0.0 and 1.0. See cold_reads_to_omit. |
| enabled | true | Enables background compaction. See Enabling and disabling background compaction. |
| max_threshold | 32 | Sets the maximum number of SSTables to allow in a minor compaction. |

| Compaction Subproperties | Default | Description |
|---|---|---|
| min_threshold | 4 | Sets the minimum number of SSTables to trigger a minor compaction. |
| min_sstable_size | 50MB | STCS roups SSTables for compaction into buckets. The bucketing process groups SSTables that differ in size by less than 50%. This results in a bucketing process that is too fine grained for small SSTables. If your SSTables are small, use min_sstable_size to define a size threshold (in bytes) below which all SSTables belong to one unique bucket. |
| tombstone_compaction_interval | 1 day | The minimum time to wait after an SSTable creation time before considering the SSTable for tombstone compaction. Tombstone compaction is the compaction triggered if the SSTable has more garbage-collectable tombstones than tombstone_threshold. |
| tombstone_threshold | 0.2 | A ratio of garbage-collectable tombstones to all contained columns, which if exceeded by the SSTable triggers compaction (with no other SSTables) for the purpose of purging the tombstones. |
| unchecked_tombstone_compaction | false | True enables more aggressive than normal tombstone compactions. A single SSTable tombstone compaction runs without checking the likelihood of success. Cassandra 2.0.9 and later. |

**CQL Compaction Subproperties for DTCS (Cassandra 2.0.11 and 2.1.1)**

| Compaction Subproperties | Default | Description |
|---|---|---|
| base_time_seconds | 3600 (1 hour) | Set to the size of the first time window. |
| enabled | true | Set to enable background compaction. See Enabling and disabling background compaction. |
| max_sstable_age_days | 1000 | Stop compacting SSTables only having data older than these specified days. Fractional days can be set. This parameter is deprecated in Casandra3.2. |
| max_window_size_seconds | 86,4000 | Set the maximum window size in seconds. The default is 1 day. |
| max_threshold | 32 | Set the maximum number of SSTables to allow in a minor compaction. |
| min_threshold | 4 | Set the minimum number of SSTables to trigger a minor compaction. |
| timestamp_resolution | MICROSECONDS | Set to MICROSECONDS or MILLISECONDS, depending on the timestamp unit of the data you insert |
| tombstone_compaction_interval | 1 day | Set to the minimum time to wait after an SSTable creation time before considering the SSTable for tombstone compaction. Tombstone compaction is the compaction triggered if the SSTable has more garbage-collectable tombstones than tombstone_threshold. |

| Compaction Subproperties | Default | Description |
|---|---|---|
| tombstone_threshold | 0.2 | Set the ratio of garbage-collectable tombstones to all contained columns, which if exceeded by the SSTable triggers compaction (with no other SSTables) for the purpose of purging the tombstones. |
| unchecked_tombstone_compaction | false | Set to True enables more aggressive than normal tombstone compactions. A single SSTable tombstone compaction runs without checking the likelihood of success. Cassandra 2.0.9 and later. |

**CQL Compaction Subproperties for LCS**

| Compaction Subproperties | Default | Description |
|---|---|---|
| enabled | true | Set to enable background compaction. See cold_reads_to_omit below. |
| sstable_size_in_mb | 160MB | Set the target size for SSTables that use the leveled compaction strategy. Although SSTable sizes should be less or equal to sstable_size_in_mb, it is possible to have a larger SSTable during compaction. This occurs when data for a given partition key is exceptionally large. The data is not split into two SSTables. |
| tombstone_compaction_interval | 1 day | Set the minimum time to wait after an SSTable creation time before considering the SSTable for tombstone compaction. Tombstone compaction is the compaction triggered if the SSTable has more garbage-collectable tombstones than tombstone_threshold. |
| tombstone_threshold | 0.2 | Set a ratio of garbage-collectable tombstones to all contained columns, which if exceeded by the SSTable triggers compaction (with no other SSTables) for the purpose of purging the tombstones. |
| unchecked_tombstone_compaction | false | Set to True enables more aggressive than normal tombstone compactions. A single SSTable tombstone compaction runs without checking the likelihood of success. |

## cold_reads_to_omit

Using SizeTieredCompactionStrategy, you can configure the maximum percentage of reads/sec that ignored SSTables may account for. The recommended range of values is 0.0 and 1.0. In Cassandra 2.1 and later, Cassandra ignores the coldest 5% of SSTables. In Cassandra 2.0.3 and later, the cold_reads_to_omit is (0.0) by default: all SSTables are compacted.

You can increase the cold_reads_to_omit property value to tune performace per table. A value of 1.0 completely disables compaction. The "Optimizations around Cold SSTables" blog includes detailed information tuning performance using this property, which avoids compacting cold SSTables. Use the `ALTER TABLE` command to configure cold_reads_to_omit.

### Enabling and disabling background compaction

The following example uses the enable property to disable background compaction:

```
ALTER TABLE mytable WITH COMPACTION = {'class':
 'SizeTieredCompactionStrategy', 'enabled': 'false'}
```

Disabling background compaction can prove harmful, as disk space is never regained and zombies, tombstones that rewrite data as new data, are possible. In most cases, although compaction uses I/O, it is better to leave it enabled, rather than disabling it.

## Compression subproperties

Using CQL, you can configure compression for a table by constructing a map of the compaction property and the following subproperties:

**Table: CQL Compression Subproperties**

| Compression Subproperties | Default | Description |
|---|---|---|
| sstable_compression | `SnappyCompressor` | The compression algorithm to use. Valid values are `LZ4Compressor`, `SnappyCompressor`, and `DeflateCompressor`. See sstable_compressionbelow. |
| chunk_length_kb | 64KB | On disk, SSTables are compressed by block to allow random reads. This subproperty of compression defines the size (in KB) of the block. Values larger than the default value might improve the compression rate, but increases the minimum size of data to be read from disk when a read occurs. The default value is a good middle-ground for compressing tables. Adjust compression size to account for read/write access patterns (how much data is typically requested at once) and the average size of rows in the table. |
| crc_check_chance | 1.0 | When compression is enabled, each compressed block includes a checksum of that block for the purpose of detecting disk bitrot and avoiding the propagation of corruption to other replica. This option defines the probability with which those checksums are checked during read. By default they are always checked. Set to 0 to disable checksum checking and to 0.5, for instance, to check them on every other read. |

### sstable_compression

The compression algorithm to use. Valid values are `LZ4Compressor` `SnappyCompressor`, and `DeflateCompressor`. Use an empty string ('') to disable compression:

```
ALTER TABLE mytable WITH COMPRESSION = {'sstable_compression': ''};
```

Choosing the right compressor depends on your requirements for space savings over read performance. LZ4 is fastest to decompress, followed by Snappy, then by Deflate. Compression effectiveness is inversely correlated with decompression speed. The extra compression from Deflate or Snappy is not enough to make up for the decreased performance for general-purpose workloads, but for archival data they may be worth considering. Developers can also implement custom compression classes using the

`org.apache.cassandra.io.compress.ICompressor` interface. Specify the full class name as a "string constant".

# Functions

CQL supports several functions that transform one or more column values into a new value. Aggregation functions are not supported.

- Blob conversion functions
- UUID and Timeuuid functions
- Token function
- WRITETIME function
- TTL function

Use the token function to compute the token for a given partition key. The exact signature of the token function depends on the table and partitioner used by the cluster. The type of the arguments to the token function depends on the type of the partition key columns. The return type depends on the partitioner in use:

- Murmur3Partitioner, bigint
- RandomPartitioner, varint
- ByteOrderedPartitioner, blob

For instance, in a cluster using the default Murmur3Partitioner, the token function that computes the token for the partition key of this table takes a single argument of type text. The partition key is userid. There is no clustering column so the partition key is the same as the primary key, and the return type is bigint.

```
CREATE TABLE users (
userid text PRIMARY KEY,
username text,
...
)
```

Regardless of the partitioner in use, Cassandra does not support non-equal conditional operations on the partition key. Use the token function for range queries on the partition key.

# CQL limits

Observe the following upper limits:

- Columns in a partition: 2B ($2^{31}$); single column value size: 2 GB (*x* MB is recommended)
- Clustering column value, length of: 65535 ($2^{16}$-1)
- Key length: 65535 ($2^{16}$-1)
- Query parameters in a query: 65535 ($2^{16}$-1)
- Statements in a batch: 65535 ($2^{16}$-1)
- Fields in a tuple: 32768 ($2^{15}$) (just a few fields, such as 2-10, are recommended)
- Collection (List): collection size: 2B ($2^{31}$); values size: 65535 ($2^{16}$-1) (Cassandra 2.1 and later, using native protocol v3)
- Collection (Set): collection size: 2B ($2^{31}$); values size: 65535 ($2^{16}$-1) (Cassandra 2.1 and later, using native protocol v3)
- Collection (Map): collection size: 2B ($2^{31}$); number of keys: 65535 ($2^{16}$-1); values size: 65535 ($2^{16}$-1) (Cassandra 2.1 and later, using native protocol v3)
- Blob size: 2 GB ( less than 1 MB is recommended)

**Note:** The limits specified for collections are for non-frozen collections.

# cqlsh commands

# cqlsh

Start the CQL interactive terminal.

## Synopsis

```
cqlsh [options] [host [port]]
```

```
python cqlsh [options] [host [port]]
```

## Description

The Cassandra installation includes the cqlsh utility, a python-based command line client for executing Cassandra Query Language (CQL) commands. The cqlsh command is used on the Linux or Windows command line to start the cqlsh utility. On Windows, the keyword *python* is used if the PATH environment variable does not point to the python installation.

You can use cqlsh to execute CQL commands interactively. cqlsh supports tab completion. You can also execute cqlsh commands, such as `TRACE`.

## Requirements

In Cassandra 2.1, the cqlsh utility uses the native protocol. In Cassandra 2.1, which uses the Datastax python driver, the default cqlsh listen port is 9042.

In Cassandra 2.0, the cqlsh utility uses the Thrift transport. In Cassandra 2.0.x, the default cqlsh listen port is 9160. By default, Cassandra 2.0.x and earlier enables Thrift by configuring start_rpc to true in the cassandra.yaml file. The cqlsh utility uses the Thrift RPC service. Also, firewall configuration to allow access through the Thrift port might be required.

For more information about configuration, see the Cassandra 2.1 cassandra.yaml or Cassandra 2.0 cassandra.yaml file.

## Options

**-C, --color**
Always use color output.

**--debug**
Show additional debugging information.

**--cqlshrc path**
Use an alternative cqlshrc file location, path. (Cassandra 2.1.1)

**-e cql_statement, --execute cql_statement**
Accept and execute a CQL command in Cassandra 2.1 and later. Useful for saving CQL output to a file.

**-f file_name, --file=file_name**
Execute commands from file_name, then exit.

**-h, --help**
Show the online help about these options and exit.

**-k keyspace_name**
Use the given keyspace. Equivalent to issuing a USE *keyspace* command immediately after starting cqlsh.

**--no-color**
Never use color output.

**-p password**
Authenticate using password. Default = cassandra.

**-t transport_factory_name, --transport=transport_factory_name**
Use the provided Thrift transport factory function.

**-u user_name**
Authenticate as user. Default = cassandra.

**--version**
Show the cqlsh version.

## Creating and using `cqlshrc` file

You can create `cqlshrc` file that resides in the hidden `.cassandra` directory in your home directory. When present, the file can pass default configuration information to cqlsh. A sample file looks like this:

```
; Sample ~/.cassandra/cqlshrc file.
[authentication]
username = fred
password = !!bang!!
```

The Cassandra installation includes a `cqlshrc.sample` file in the `conf` directory of a tarball installation. On Windows, in Command Prompt, create this file by copying the `cqlshrc.sample` file from the `conf` directory to the hidden `.cassandra` folder your user home folder, and renaming it to `cqlshrc`.

You can use a `cqlshrc` file to configure SSL encryption instead of overriding the SSL_CERTFILE environmental variables repeatedly. Cassandra internal authentication must be configured before users can use the authentication options.

## cqlshrc options

You configure the `cqlshrc` file by setting these options in the [authentication], [ui], or [ssl] sections of the file.

[authentication] options are:

**keyspace**
Use the given keyspace. Equivalent to issuing a USE keyspace command immediately after starting cqlsh.

**password**
Authenticate using password.

**username**
Authenticate as user.

[connection] option (Cassandra 2.1.1) is:

**client_timeout**
Configures the cqlsh timeout in seconds. Set to None or the number of seconds of inactivity that triggers timeout.

[csv] option (Cassandra 2.0.15+, 2.1.5+) is:

**field_size_limit**
Configures the cqlsh field size. Set to a particular field size. For instance, field_size_limit (1000000000).

[ssl] options are covered in the Cassandra documentation.

[ui] options are:

**color**
Always use color output.

**completekey**
Use this key for autocompletion of a cqlsh shell entry. Default is the tab key.

**float_precision**
Use this many decimal digits of precision. Default = 5.

**time_format**
Configure the output format of database objects using Python strftime syntax.

## Using CQL commands

On startup, cqlsh shows the name of the cluster, IP address, and the port used for connection to the cqlsh utility. The cqlsh prompt initially is cqlsh>. After you specify a keyspace to use, the prompt includes the name of the keyspace. For example:

```
cqlsh 1.2.3.4 9042 -u jdoe -p mypassword
Connected to trace_consistency at 1.2.3.4:9042.
[cqlsh 5.0.1 | Cassandra 2.1.0 | CQL spec 3.2 | Native protocol v3]
Use HELP for help.
cqlsh>USE mykeyspace;
cqlsh:mykeyspace>
```

At the cqlsh prompt, type CQL commands. Use a semicolon to terminate a command. A new line does not terminate a command, so commands can be spread over several lines for clarity.

```
cqlsh> USE demo_cl;
cqlsh:demo_cl> SELECT * FROM demo_table
           ... WHERE id = 0;
```

If a command is sent and executed successfully, results are sent to standard output.

```
 id | col1 | col2
----+------+------
  0 |    0 |    0

(1 rows)
```

The lexical structure of commands, covered earlier in this reference, includes how upper- and lower-case literals are treated in commands, when to use quotation marks in strings, and how to enter exponential notation.

## Saving CQL output in a file

Using the -e option to the `cqlsh` command followed by a CQL statement, enclosed in quotation marks, accepts and executes the CQL statement. For example, to save the output of a SELECT statement to `myoutput.txt`:

```
cqlsh -e "SELECT * FROM mytable" > myoutput.txt
```

## Using files as input

To execute CQL commands in a file, use the -f option and the path to the file on the operating system command line. Or, after you start cqlsh, use the `SOURCE` command and the path to the file on the cqlsh command line.

### cqlsh environment variables

You can override the default cqlsh host and listen port by setting the CQLSH_HOST and CQLSH_PORT environment variables. You set the CQLSH_HOST to a host name or IP address. When configured, cqlsh uses the variables instead of the default values of localhost and port 9042 (Cassandra 2.1 or later) or 9160 (Cassandra 2.0.x). A host and port number given on the command line takes precedence over configured variables.

**Related information**

Cassandra 2.0 cassandra.yaml

Cassandra 2.1 cassandra.yaml

## CAPTURE

Captures command output and appends it to a file.

### Synopsis

```
CAPTURE ('<file>' | OFF )
```

### Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

### Description

To start capturing the output of a query, specify the path of the file relative to the current directory. Enclose the file name in single quotation marks. The shorthand notation in this example is supported for referring to $HOME.

### Examples

```
CAPTURE '~/mydir/myfile.txt'
```

Output is not shown on the console while it is captured. Only query result output is captured. Errors and output from cqlsh-only commands still appear.

To stop capturing output and return to normal display of output, use CAPTURE OFF.

To determine the current capture state, use CAPTURE with no arguments.

## CONSISTENCY

Shows the current consistency level, or given a level, sets it.

### Synopsis

```
CONSISTENCY level
```

### Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

### Description

Providing an argument to the CONSISTENCY command overrides the default consistency level of ONE, and configures the consistency level for future requests. Valid values are: ANY, ONE, TWO, THREE, QUORUM, ALL, LOCAL_QUORUM, EACH_QUORUM, SERIAL and LOCAL_SERIAL.

Providing no argument shows the current consistency level.

### Example

```
CONSISTENCY
```

If you haven't changed the default, the output of the CONSISTENCY command with no arguments is:

```
Current consistency level is ONE.
```

## COPY

Imports and exports CSV (comma-separated values) data to and from Cassandra.

### Synopsis

```
COPY table_name ( column, ...)
FROM ( 'file_name' | STDIN )
WITH option = 'value' AND ...

COPY table_name ( column , ... )
TO ( 'file_name' | STDOUT )
WITH option = 'value' AND ...
```

### Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

### Description

Using the COPY options in a WITH clause, you can change the CSV format. These tables describe the available options:

**Table: Common COPY options for TO and FROM**

| COPY Options | Default Value | Use To |
|---|---|---|
| DELIMITER | comma (,) | Set the character that separates fields having newline characters in the file. |
| QUOTE | quotation mark (") | Set the character that encloses field values. |
| ESCAPE | backslash (\) | Set the character that escapes literal uses of the QUOTE character. |
| HEADER | false | Set true to indicate that first row of the file is a header. |
| NULL | an empty string | Represents the absence of a value. |
| DATETIMEFORMAT | '%Y-%m-%d %H:%M:%S%z' | Set the time format for reading or writing CSV time data. The timestamp uses the strftime format. If not set, the default value is set to the time_format value in the cqlshrc file. |
| MAX ATTEMPTS | 5 | Set the maximum number of attempts for errors. |
| REPORTFREQUENCY | 0.25 | Set the frequency with which status is displayed, in seconds |
| DECIMALSEP | period (.) | Set a separator for decimal values |
| THOUSANDSSEP | None | Set a separator for thousands digit groups |
| BOOLSTYLE | True, False | Set a representation for boolean values for True and False. The values are case insensitive.,Example: yes,no or 1,0 |
| NUMPROCESSES | Number of cores - 1 | Set the number of worker processes. Maximum value is 16. |
| CONFIGFILE | None | Specify a configuration file with the same format as the cqlshrc file to set WITH options. The following sections can be specified: [copy], [copy-to], [copy-from], [copy:ks.table], [copy-to:ks.table], [copy-from:ks.table], where <ks> is the keyspace name and <table> is the tablename. Options are read from these sections, in the order specified above. Command line options always override options in configuration files. Depending on the COPY direction, only the relevant copy-from or copy-to sections are used. If no configuration file is specified, the cqlshrc file is searched instead. |
| RATEFILE | None | Specify a file for printing output statistics. |

**Table: COPY FROM options**

| COPY Options | Default Value | Use To |
|---|---|---|
| CHUNKSIZE | 1,000 | Set the size of chunks passed to worker processes. |
| INGESTRATE | 100,000 | Set an approximate ingest rate in rows per second. Must be set to a greater value than chunk size. |
| MAXBATCHSIZE | 20 | Set the maximum size of an import batch. |

| COPY Options | Default Value | Use To |
|---|---|---|
| MINBATCHSIZE | 2 | Set the minimum size of an import batch. |
| MAXROWS | -1 | Set the maximum number of rows. "-1" sets no maximum. |
| SKIPROWS | 0 | The number of rows to skip. |
| SKIPCOLS | None | Set a comma-separated list of column names to skip. |
| MAXPARSEERRORS | -1 | Set the maximum global number of parsing errors. "-1" sets no maximum. |
| MAXINSERTERRORS | -1 | Set the maximum global number of insert errors. "-1" sets no maximum. |
| ERRFILE | None | Set a file to store all rows that are not imported. If no value is set, the information is stored in import_ks_table.err where <ks> is the keyspace and <table> is the table name. |
| TTL | 3600 | Set the time to live in seconds. By default ,data will not expire. |

**Table: COPY TO options**

| COPY Options | Default Value | Use To |
|---|---|---|
| ENCODING | UTF8 | Set the COPY TO command to output unicode strings. |
| PAGESIZE | 1,000 | Set the page size for fetching results. |
| PAGETIMEOUT | 10 | Set the page timeout for fetching results. |
| BEGINTOKEN | None | Set the minimum token string for exporting data. |
| ENDTOKEN | None | Set the maximum token string for exporting data. |
| MAXREQUESTS | 6 | Set the maximum number of requests each worker process can work on in parallel. |
| MAXOUTPUTSIZE | -1 | Set the maximum size of the output file, measured in number of lines. If a value is set, the output file will be split into segment when the value is exceeded. "-1" sets no maximum. |

The `ENCODING` option is available only for the `COPY TO` command. This table shows that, by default, Cassandra expects the CSV data to consist of fields separated by commas (,), records separated by line separators (a newline, \r\n), and field values enclosed in double-quotation marks (""). Also, to avoid ambiguity, escape a literal double-quotation mark using a backslash inside a string enclosed in double-quotation marks ("\""). By default, Cassandra does not expect the CSV file to have a header record on the first line that consists of the column names. `COPY TO` includes the header in the output if `HEADER=TRUE`. COPY FROM ignores the first line if `HEADER=TRUE`.

You cannot copy data to or from counter tables.

## COPY FROM a CSV file

By default, when you use the `COPY FROM` command, Cassandra expects every row in the CSV input to contain the same number of columns. The number of columns in the CSV input is the same as the number

of columns in the Cassandra table metadata. Cassandra assigns fields in the respective order. To apply your input data to a particular set of columns, specify the column names in parentheses after the table name.

`COPY FROM` loads rows from a CVS file in a parallel non-deterministic order. Empty data for a column is assumed by default as NULL value and will override a value; if the data is overwritten, unexpected results can occur. Data can be successfully loaded but with non-deterministic random results if there is more than one row in the CSV file with the same primary key. Having more than one row with the same primary key is not explicitly checked, so unintended results can occur.

COPY FROM is intended for importing small datasets (a few million rows or less) into Cassandra. For importing larger datasets, use the Cassandra bulk loader.

## COPY TO a CSV file

For example, assume you have the following table in CQL:

```
cqlsh> SELECT * FROM test.airplanes;
```

```
 name           | mach | manufacturer | year
----------------+------+--------------+------
 P38-Lightning |  0.7 |     Lockheed | 1937
```

After inserting data into the table, you can copy the data to a CSV file in another order by specifying the column names in parentheses after the table name:

```
COPY airplanes
(name, mach, year, manufacturer)
 TO 'temp.csv'
```

## Specifying the source or destination files

Specify the source file of the CSV input or the destination file of the CSV output by a file path. Alternatively, you can use the STDIN or STDOUT keywords to import from standard input and export to standard output. When using stdin, signal the end of the CSV data with a backslash and period ("\.") on a separate line. If the data is being imported into a table that already contains data, COPY FROM does not truncate the table beforehand. You can copy only a partial set of columns. Specify the entire set or a subset of column names in parentheses after the table name in the order you want to import or export them. By default, when you use the COPY TO command, Cassandra copies data to the CSV file in the order defined in the Cassandra table metadata. You can also omit listing the column names when you want to import or export all the columns in the order they appear in the source table or CSV file.

## Roundtrip copying of a simple table

**Copy a table to a CSV file.**

1. Using CQL, create a table named airplanes and copy it to a CSV file.

```
CREATE KEYSPACE test
  WITH REPLICATION = { 'class' : 'NetworkTopologyStrategy',
 'datacenter1' : 1 };

USE test;

CREATE TABLE airplanes (
  name text PRIMARY KEY,
  manufacturer ascii,
  year int,
```

```
  mach float
);

INSERT INTO airplanes
  (name, manufacturer, year, mach)
  VALUES ('P38-Lightning', 'Lockheed', 1937, 0.7);

COPY airplanes (name, manufacturer, year, mach) TO 'temp.csv';
```

```
1 rows exported in 0.004 seconds.
```

2. Clear the data from the airplanes table and import the data from the temp.csv file.

```
TRUNCATE airplanes;

COPY airplanes (name, manufacturer, year, mach) FROM 'temp.csv';
```

```
1 rows imported in 0.087 seconds.
```

**Copy data from standard input to a table.**

1. Enter data directly during an interactive cqlsh session, using the COPY command defaults.

```
TRUNCATE airplanes;

COPY airplanes (name, manufacturer, year, mach) FROM STDIN;
```

The output is:

```
[Use \. on a line by itself to end input]
[copy]
```

2. At the [copy] prompt, enter the following data:

```
'F-14D Super Tomcat', Grumman, 1987, 2.34
'MiG-23 Flogger', Russian-made, 1964, 2.35
'Su-27 Flanker', U.S.S.R., 1981, 2.35
\.
```

3. Query the airplanes table to see data imported from STDIN:

```
SELECT * FROM airplanes;
```

Output is:

```
 name                | manufacturer | year | mach
---------------------+--------------+------+-------------
 F-14D Super Tomcat |      Grumman | 1987 | 2.35
      P38-Lightning |     Lockheed | 1937 | 0.7
      Su-27 Flanker |     U.S.S.R. | 1981 | 2.35
```

## Copying collections

Cassandra supports round-trip copying of collections to and from CSV files. To perform this example, download the sample code now.

1. Unzip the downloaded file named `cql_collections.zip`.
2. Copy/paste all the CQL commands from the `cql_collections.txt` file to the cqlsh command line.

3. Take a look at the contents of the songs table. The table contains a map of venues, a list of reviews, and a set of tags.

```
cqlsh> SELECT * FROM music.songs;
```

```
 id          |album|artist|data|reviews            |tags              |
title|venue
------------+-----+------+----+-------------------+------------------+-----
+-----------------------------------------------------------------------
 7db1a490...| null| null |null|['hot dance music']|         {'rock'}|
 null|{'2013-09-22...': 'The Fillmore', '2013-10-01...': 'The Apple
 Barrel'}
 a3e64f8f...| null| null |null|               null|{'1973', 'blues'}|
 null|null
 8a172618...| null| null |null|               null|'2007', 'covers'}|
 null|null
```

4. Copy the music.songs table to a CSV file named `songs-20140603.csv`.

```
cqlsh> COPY music.songs to 'songs-20140603.csv';
```

```
3 rows exported in 0.006 seconds.
```

5. Check that the copy operation worked.

```
cqlsh> exit;

$ cat songs-20140603.csv
7db1a490...,,,,,['hot dance music'],{'rock'},,"{'2013-09-22...': 'The
 Fillmore', '2013-10-01....': 'The Apple Barrel'}"
a3e64f8f...,,,,,,"{'1973', 'blues'}",,
8a172618...,,,,,,"{'2007', 'covers'}",,
```

6. Start cqlsh again, and create a table definition that matches the data in the `songs-204140603` file.

```
cqlsh> CREATE TABLE music.imported_songs (
        id uuid PRIMARY KEY,
        album text,
        artist text,
        data blob,
        reviews list<text>,
        tags set<text>,
        title text,
        venue map<timestamp, text>
       );
```

7. Copy the data from the CSV file into the imported_songs table.

```
cqlsh> COPY music.imported_songs from 'songs-20140603.csv';
```

```
3 rows imported in 0.004 seconds.
```

# DESCRIBE

Provides information about the connected Cassandra cluster, or about the data objects stored in the cluster.

## Synopsis

```
DESCRIBE FULL ( CLUSTER | SCHEMA )
| KEYSPACES
| ( KEYSPACE keyspace_name )
| TABLES
| ( TABLE table_name )
| TYPES
| ( TYPE user_defined_type )
| INDEX
| ( INDEX index_name )
```

## Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

## Description

The DESCRIBE or DESC command outputs information about the connected Cassandra cluster, or about the data stored on it. To query the system tables directly, use SELECT.

The keyspace and table name arguments are case-sensitive and need to match the upper or lowercase names stored internally. Use the DESCRIBE commands to list objects by their internal names. Use DESCRIBE FULL SCHEMA if you need the schema of system_* keyspaces.

DESCRIBE functions in the following ways:

| DESCRIBE commands | Example | Description |
|---|---|---|
| DESCRIBE CLUSTER | DESCRIBE CLUSTER; | Output information about the connected Cassandra cluster. Cluster name, partitioner, and snitch are output. For non-system keyspace, the endpoint-range ownership information is also shown. |
| DESCRIBE KEYSPACES | | Output a list of all keyspace names. |
| DESCRIBE KEYSPACE <keyspace_name> | DESCRIBE KEYSPACE cycling; | Output CQL commands for the given keyspace. These CQL commands can be used to recreate the keyspace and tables. |
| DESCRIBE [FULL] SCHEMA | | Output CQL commands for entire non-system keyspace and table schema. Use the FULL option to also include system keyspaces. |
| DESCRIBE TABLES | | Output all tables in the current keyspace, or in all keyspaces if there is not current keyspace. |
| DESCRIBE TABLE <table_name> | DESCRIBE TABLE upcoming_calendar; | Output CQL commands for the given table. This CQL command can be used to recreate the table. |
| DESCRIBE INDEX <index_name> | DESCRIBE INDEX team_entry; | Output CQL command for the given index. This CQL command can be used to recreate the index. |
| DESCRIBE TYPES | | Output list of all user-defined types in the current keyspace. |

| DESCRIBE commands | Example | Description |
|---|---|---|
| DESCRIBE TYPE <type_name> | DESCRIBE TYPE basic_info; | Output CQL command for the given user-defined type. This CQL command can be used to recreate the index. |

# EXPAND

Formats the output of a query vertically.

## Synopsis

```
EXPAND ( ON | OFF )
```

## Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

## Description

This command lists the contents of each row of a table vertically, providing a more convenient way to read long rows of data than the default horizontal format. You scroll down to see more of the row instead of scrolling to the right. Each column name appears on a separate line in column one and the values appear in column two.

Sample output of EXPAND ON is:

```
cqlsh:my_ks> EXPAND ON
             Now printing expanded output
cqlsh:my_ks> SELECT * FROM users;
```

```
@ Row 1
------------+---------------------------------------------
 userid     | samwise
 emails     | {'samwise@gmail.com', 's@gamgee.com'}
 first_name | Samwise
 last_name  | Gamgee
 todo       | {'2013-09-22 12:01:00-0700': 'plant trees'}
 top_scores | null

@ Row 2
------------+---------------------------------------------
 userid     | frodo
 emails     | {'baggins@gmail.com', 'f@baggins.com'}
 first_name | Frodo
 last_name  | Baggins
 todo       | {'2012-10-02 12:00:00-0700': 'throw my precious into mount
 doom'}
 top_scores | null

(2 rows)
```

# EXIT

Terminates cqlsh.

## Synopsis

```
EXIT | QUIT
```

## Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

# PAGING

Enables or disables query paging.

## Synopsis

```
PAGING ( ON | OFF )
```

## Synopsis Legend

In the synopsis section of each statement, formatting has the following meaning:

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

## Description

In Cassandra 2.1.1, you can use query paging in cqlsh. Paging provides the output of queries in 100-line chunks followed by the *more* prompt. To get the next chunk, press the space bar. Turning paging on enables query paging for all further queries. Using no ON or OFF argument with the command shows the current query paging status.

# SHOW

Shows the Cassandra version, host, or tracing information for the current cqlsh client session.

## Synopsis

```
SHOW VERSION
| HOST
| SESSION tracing_session_id
```

### Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

### Description

A SHOW command displays this information about the current cqlsh client session:

- The version and build number of the connected Cassandra instance, as well as the CQL mode for cqlsh and the native protocol version used by the connected Cassandra instance.
- The host information of the Cassandra node that the cqlsh session is currently connected to.
- Tracing information for the current cqlsh session.

The SHOW SESSION command retrieves tracing session information, which is available for 24 hours. After that time, the tracing information time-to-live expires.

These examples show how to use the commands.

```
cqlsh:my_ks> SHOW version
[cqlsh 5.0.1 | Cassandra 2.1.0 | CQL spec 3.2 | Native protocol v3]

cqlsh:my_ks> SHOW host
Connected to Test Cluster at 127.0.0.1:9042.

cqlsh:my_ks> SHOW SESSION d0321c90-508e-11e3-8c7b-73ded3cb6170
```

Sample output of SHOW SESSION is:

```
Tracing session: d0321c90-508e-11e3-8c7b-73ded3cb6170

 activity
                                                                                  | timestamp
 | source    | source_elapsed
-----------------------------------------------------------------------------------------------
+--------------+-----------+----------------

                                              execute_cql3_query | 12:19:52,372
 | 127.0.0.1 |              0
 Parsing CREATE TABLE emp (\n  empID int,\n  deptID int,\n  first_name
 varchar,\n  last_name varchar,\n  PRIMARY KEY (empID, deptID)\n); |
 12:19:52,372 | 127.0.0.1 |            153

                                                 Request complete | 12:19:52,372
 | 127.0.0.1 |            650
. . .
```

# SOURCE

Executes a file containing CQL statements.

### Synopsis

```
SOURCE 'file'
```

### Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

### Description

To execute the contents of a file, specify the path of the file relative to the current directory. Enclose the file name in single quotation marks. The shorthand notation in this example is supported for referring to $HOME:

### Examples

```
SOURCE '~/mydir/myfile.txt'
```

The output for each statement, if there is any, appears in turn, including any error messages. Errors do not abort execution of the file.

Alternatively, use the --file option to execute a file while starting CQL.

## TRACING

Enables or disables request tracing.

### Synopsis

```
TRACING ( ON | OFF )
```

### Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

### Description

To turn tracing read/write requests on or off, use the TRACING command. After turning on tracing, database activity creates output that can help you understand Cassandra internal operations and troubleshoot performance problems. For example, using the tracing tutorial you can see how different consistency levels affect operations. Some tracing messages refer to internals of the database that users cannot understand, but can provide to the Cassandra team for troubleshooting.

For 24 hours, Cassandra saves the tracing information in sessions and events tables in the system_traces keyspace, which you query when using probabilistic tracing. For information about probabilistic tracing, see Cassandra 2.1 documentation or Cassandra 2.0 documentation.

```
$ CREATE TABLE sessions (
  session_id uuid PRIMARY KEY,
```

```
  coordinator inet,
  duration int,
  parameters map<text, text>,
  request text,
  started_at timestamp
);

CREATE TABLE events (
 session_id uuid,
 event_id timeuuid,
 activity text,
 source inet,
 source_elapsed int,
 thread text,
 PRIMARY KEY (session_id, event_id)
);
```

The source_elapsed column stores the elapsed time in microseconds before the event occurred on the source node.

To keep tracing information, copy the data in sessions and event tables to another location. Alternatively, use the tracing session id to retrieve session information using SHOW SESSION. Tracing session information expires after one day.

## Tracing a write request

This example shows tracing activity on a 3-node cluster created by ccm on Mac OSX. Using a keyspace that has a replication factor of 3 and an employee table similar to the one in "Using a compound primary key on page 21," the tracing shows that the coordinator performs the following actions:

- Identifies the target nodes for replication of the row.
- Writes the row to the commitlog and memtable.
- Confirms completion of the request.

```
TRACING ON

INSERT INTO emp (empID, deptID, first_name, last_name)
  VALUES (104, 15, 'jane', 'smith');
```

Cassandra provides a description of each step it takes to satisfy the request, the names of nodes that are affected, the time for each step, and the total time for the request.

```
Tracing session: 740b9c10-3506-11e2-0000-fe8ebeead9ff

 activity                                  | timestamp    | source    |
 source_elapsed
-------------------------------------------+--------------+-----------
+----------------
                    execute_cql3_query | 16:41:00,754 | 127.0.0.1 |
   0
                     Parsing statement | 16:41:00,754 | 127.0.0.1 |
  48
                   Preparing statement | 16:41:00,755 | 127.0.0.1 |
 658
    Determining replicas for mutation | 16:41:00,755 | 127.0.0.1 |
 979
     Message received from /127.0.0.1 | 16:41:00,756 | 127.0.0.3 |
  37
      Acquiring switchLock read lock | 16:41:00,756 | 127.0.0.1 |
 1848
       Sending message to /127.0.0.3 | 16:41:00,756 | 127.0.0.1 |
 1853
```

```
              Appending to commitlog | 16:41:00,756 | 127.0.0.1 |
 1891
      Sending message to /127.0.0.2 | 16:41:00,756 | 127.0.0.1 |
 1911
              Adding to emp memtable | 16:41:00,756 | 127.0.0.1 |
 1997
     Acquiring switchLock read lock | 16:41:00,757 | 127.0.0.3 |
 395
   Message received from /127.0.0.1 | 16:41:00,757 | 127.0.0.2 |
  42
              Appending to commitlog | 16:41:00,757 | 127.0.0.3 |
 432
     Acquiring switchLock read lock | 16:41:00,757 | 127.0.0.2 |
 168
              Adding to emp memtable | 16:41:00,757 | 127.0.0.3 |
 522
              Appending to commitlog | 16:41:00,757 | 127.0.0.2 |
 211
              Adding to emp memtable | 16:41:00,757 | 127.0.0.2 |
 359
   Enqueuing response to /127.0.0.1 | 16:41:00,758 | 127.0.0.3 |
 1282
   Enqueuing response to /127.0.0.1 | 16:41:00,758 | 127.0.0.2 |
 1024
      Sending message to /127.0.0.1 | 16:41:00,758 | 127.0.0.3 |
 1469
      Sending message to /127.0.0.1 | 16:41:00,758 | 127.0.0.2 |
 1179
   Message received from /127.0.0.2 | 16:41:00,765 | 127.0.0.1 |
 10966
   Message received from /127.0.0.3 | 16:41:00,765 | 127.0.0.1 |
 10966
 Processing response from /127.0.0.2 | 16:41:00,765 | 127.0.0.1 |
 11063
 Processing response from /127.0.0.3 | 16:41:00,765 | 127.0.0.1 |
 11066
                   Request complete | 16:41:00,765 | 127.0.0.1 |
 11139
```

## Tracing a sequential scan

Due to the log structured design of Cassandra, a single row is spread across multiple SSTables. Reading one row involves reading pieces from multiple SSTables, as shown by this trace of a request to read the employee table, which was pre-loaded with 10 rows of data.

```
SELECT * FROM emp;
```

Output is:

```
 empid | deptid | first_name | last_name
-------+--------+------------+-----------
   110 |     16 |      naoko |     murai
   105 |     15 |       john |     smith
   111 |     15 |       jane |     thath
   113 |     15 |       lisa |     amato
   112 |     20 |       mike |     burns
   107 |     15 |     sukhit |       ran
   108 |     16 |        tom |     brown
   109 |     18 |        ann |     green
   104 |     15 |       jane |     smith
```

```
   106 |     15 |        bob |     jones
```

The tracing output of this read request looks something like this (a few rows have been truncated to fit on this page):

```
Tracing session: bf5163e0-350f-11e2-0000-fe8ebeead9ff

 activity                                                  | timestamp    | source
 | source_elapsed
-----------------------------------------------------------+--------------+-----------
+----------------
                                  execute_cql3_query | 17:47:32,511 | 127.0.0.1
 |              0
                                   Parsing statement | 17:47:32,511 | 127.0.0.1
 |             47
                                 Preparing statement | 17:47:32,511 | 127.0.0.1
 |            249
                      Determining replicas to query | 17:47:32,511 | 127.0.0.1
 |            383
                      Sending message to /127.0.0.2 | 17:47:32,512 | 127.0.0.1
 |            883
                 Message received from /127.0.0.1 | 17:47:32,512 | 127.0.0.2
 |             33
  Executing seq scan across 0 sstables for . . . | 17:47:32,513 | 127.0.0.2
 |            670
                 Read 1 live cells and 0 tombstoned | 17:47:32,513 | 127.0.0.2
 |            964
                 Read 1 live cells and 0 tombstoned | 17:47:32,514 | 127.0.0.2
 |           1268
                 Read 1 live cells and 0 tombstoned | 17:47:32,514 | 127.0.0.2
 |           1502
                 Read 1 live cells and 0 tombstoned | 17:47:32,514 | 127.0.0.2
 |           1673
                       Scanned 4 rows and matched 4 | 17:47:32,514 | 127.0.0.2
 |           1721
                    Enqueuing response to /127.0.0.1 | 17:47:32,514 | 127.0.0.2
 |           1742
                      Sending message to /127.0.0.1 | 17:47:32,514 | 127.0.0.2
 |           1852
                    Message received from /127.0.0.2 | 17:47:32,515 | 127.0.0.1
 |           3776
                   Processing response from /127.0.0.2 | 17:47:32,515 | 127.0.0.1
 |           3900
                      Sending message to /127.0.0.2 | 17:47:32,665 | 127.0.0.1
 |         153535
                 Message received from /127.0.0.1 | 17:47:32,665 | 127.0.0.2
 |             44
  Executing seq scan across 0 sstables for . . . | 17:47:32,666 | 127.0.0.2
 |           1068
                 Read 1 live cells and 0 tombstoned | 17:47:32,667 | 127.0.0.2
 |           1454
                 Read 1 live cells and 0 tombstoned | 17:47:32,667 | 127.0.0.2
 |           1640
                       Scanned 2 rows and matched 2 | 17:47:32,667 | 127.0.0.2
 |           1694
                    Enqueuing response to /127.0.0.1 | 17:47:32,667 | 127.0.0.2
 |           1722
                      Sending message to /127.0.0.1 | 17:47:32,667 | 127.0.0.2
 |           1825
                    Message received from /127.0.0.2 | 17:47:32,668 | 127.0.0.1
 |         156454
                   Processing response from /127.0.0.2 | 17:47:32,668 | 127.0.0.1
 |         156610
```

```
 Executing seq scan across 0 sstables for . . . | 17:47:32,669 | 127.0.0.1
|         157387
            Read 1 live cells and 0 tombstoned | 17:47:32,669 | 127.0.0.1
|         157729
            Read 1 live cells and 0 tombstoned | 17:47:32,669 | 127.0.0.1
|         157904
            Read 1 live cells and 0 tombstoned | 17:47:32,669 | 127.0.0.1
|         158054
            Read 1 live cells and 0 tombstoned | 17:47:32,669 | 127.0.0.1
|         158217
                     Scanned 4 rows and matched 4 | 17:47:32,669 | 127.0.0.1
|         158270
                                  Request complete | 17:47:32,670 | 127.0.0.1
|         159525
```

The sequential scan across the cluster shows:

- The first scan found 4 rows on node 2.
- The second scan found 2 more rows found on node 2.
- The third scan found the 4 rows on node 1.

**Related information**

Cassandra 2.1 probabilistic tracing
Cassandra 2.0 probabilistic tracing

# CQL commands

# ALTER KEYSPACE

Change property values of a keyspace.

## Synopsis

```
ALTER ( KEYSPACE | SCHEMA ) keyspace_name
WITH REPLICATION = map
| ( WITH DURABLE_WRITES = ( true | false ))
AND ( DURABLE_WRITES = ( true | false))
```

map is a map collection, a JSON-style array of literals:

{ literal : literal *, literal : literal, ...* }

## Synopsis legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

### Description

ALTER KEYSPACE changes the map that defines the replica placement strategy and/or the DURABLE_WRITES value. You can also use the alias ALTER SCHEMA. Use these properties and values to construct the map. To set the replica placement strategy, construct a map of properties and values, as shown in the table of map properties on the CREATE KEYSPACE reference page. CQL property map keys must be lower case. For example, class and replication_factor are correct.

You cannot change the name of the keyspace.

### Example

Change the definition of the mykeyspace to use the NetworkTopologyStrategy in a single data center. Use the default data center name in Cassandra and a replication factor of 3.

```
ALTER KEYSPACE "Excalibur" WITH REPLICATION =
  { 'class' : 'NetworkTopologyStrategy', 'datacenter1' : 3 };
```

## ALTER TABLE

Modify the column metadata of a table.

### Synopsis

```
ALTER TABLE keyspace_name.table_name instruction
```

instruction is:

```
ALTER column_name TYPE cql_type
| ( ADD column_name cql_type )
| ( DROP column_name )
| ( RENAME column_name TO column_name )
| ( WITH property AND property ... )
```

cql_type is compatible with the original type and is a CQL type, other than a collection or counter. Exceptions: ADD supports a collection type and also, if the table is a counter, a counter type.

property is a CQL table property and value, such as speculative_retry = '10ms'. Enclose a string property in single quotation marks.

### Synopsis legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

### Description

ALTER TABLE manipulates the table metadata. You can change the data storage type of columns, add new columns, drop existing columns, and change table properties. No results are returned. You can also use the alias ALTER COLUMNFAMILY.

First, specify the name of the table to be changed after the ALTER TABLE keywords, followed by the type of change: ALTER, ADD, DROP, RENAME, or WITH. Next, provide the rest of the needed information, as explained in the following sections.

You can qualify table names by keyspace. For example, to alter the addamsFamily table in the monsters keyspace:

```
ALTER TABLE monsters.addamsFamily ALTER lastKnownLocation TYPE uuid;
```

## Changing the type of a column

To change the storage type for a column, the type you are changing from must be compatible with the type you are changing to. You can change an ascii type to text. You cannot change text (varchar) to ascii because every UTF8 string is not ascii. You can convert text to blobs. You cannot convert a blob to text because not every blob is not a UTF8 string. For example, change this type of the bio column in the users table from ascii to text, and then from text to blob.

```
CREATE TABLE users (
  user_name varchar PRIMARY KEY,
  bio ascii,
  );
ALTER TABLE users ALTER bio TYPE text;
ALTER TABLE users ALTER bio TYPE blob;
```

The column must already exist in current rows. The bytes stored in values for that column remain unchanged, and if existing data cannot be deserialized according to the new type, your CQL driver or interface might report errors.

These changes to a column type are not allowed:

- Changing the type of a clustering column.
- Changing columns on which an index is defined.

Altering the type of a column after inserting data can confuse CQL drivers/tools if the new type is incompatible with the data.

## Adding a column

To add a column, other than a column of a collection type, to a table, use ALTER TABLE and the ADD keyword in the following way:

```
ALTER TABLE addamsFamily ADD gravesite varchar;
```

To add a column of the collection type:

```
ALTER TABLE users ADD top_places list<text>;
```

No validation of existing data occurs.

These additions to a table are not allowed:

- Adding a column having the same name as an existing column
- A static column

## Dropping a column

To drop a column from the table, use ALTER TABLE and the DROP keyword. Dropping a column removes the column from the table.

```
ALTER TABLE addamsFamily DROP gender;
```

ALTER DROP removes the column from the table definition, removes data pertaining to that column, and eventually reclaims the space formerly used by the column. The column is unavailable for querying immediately after being dropped. The actual data removal occurs during compaction; data is not included in SSTables in the future. To force the removal of dropped columns before compaction occurs, use the nodetool upgradesstables command followed by an ALTER TABLE statement, which updates the table metadata to register the drop.

After re-adding a dropped column, a query does not return values written before the column was last dropped. Do not re-add a dropped column to a table using client-supplied timestamps, which is not a Cassandra-generated write time.

You cannot drop columns from tables defined with the COMPACT STORAGE option.

## Renaming a column

The main purpose of the RENAME clause is to change the names of CQL-generated primary key and column names that are missing from a legacy table. Primary key columns can be renamed. You cannot rename an indexed column or a static column, which is supported in Cassandra 2.0.6 and later.

## Modifying table properties

To change the table storage properties established during creation of the table, use one of the following formats to alter a table:

- ALTER TABLE and a WITH directive that includes the property name and value
- ALTER TABLE and a property map, shown in the next section

Using a WITH directive, for example, you can modify the read_repair_chance property, which configures the basis for invoking read repairs on reads in clusters configured for a non-quorum consistency.

To change multiple properties, use AND as shown in this example:

```
ALTER TABLE addamsFamily
  WITH comment = 'A most excellent and useful table'
  AND read_repair_chance = 0.2;
```

Enclose a text property value in single quotation marks. You cannot modify properties of a table having compact storage.

## Modifying compression and compaction

Use a property map to alter a compression or compaction option.

```
ALTER TABLE addamsFamily WITH compression =
  { 'sstable_compression' : 'DeflateCompressor', 'chunk_length_kb' : 64 };

ALTER TABLE mykeyspace.mytable
  WITH compaction = {'class': 'SizeTieredCompactionStrategy',
 'cold_reads_to_omit': 0.05};
```

Change the values of the caching property. For example, change the keys option from ALL, the default, to NONE and change the rows_per_partition to 15.

## Changing caching

In Cassandra 2.1, you create and change the caching options using a property map.

```
//Cassandra 2.1 only
```

```
ALTER TABLE users WITH caching = { 'keys' : 'NONE', 'rows_per_partition' :
 '15' };
```

Next, change just the rows_per_partition to 25.

```
//Cassandra 2.1 only

ALTER TABLE users WITH caching = { 'rows_per_partition' : '25' };
```

Finally, take a look at the table definition.

```
//Cassandra 2.1 only

DESCRIBE TABLE users;

CREATE TABLE mykeyspace.users (
    user_name text PRIMARY KEY,
    bio blob
) WITH bloom_filter_fp_chance = 0.01
    AND caching = '{"keys":"NONE", "rows_per_partition":"25"}'
    AND comment = ''
    AND compaction = {'min_threshold': '4', 'class':
 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy',
 'max_threshold': '32'}
    AND compression = {'sstable_compression':
 'org.apache.cassandra.io.compress.LZ4Compressor'}
    AND default_time_to_live = 0
    AND gc_grace_seconds = 864000
    AND max_index_interval = 2048
    AND memtable_flush_period_in_ms = 0
    AND min_index_interval = 128
    AND read_repair_chance = 0.1
    AND speculative_retry = '99.0PERCENTILE';
```

In Cassandra 2.0.x, you alter the caching options using the WITH directive.

```
//Cassandra 2.0.x only
      ALTER TABLE users WITH caching = "keys_only;
```

**Important:** Use row caching in Cassandra 2.0.x with caution.

# ALTER TYPE

Modify a user-defined type. Cassandra 2.1 and later.

## Synopsis

```
ALTER TYPE field_name instruction
```

instruction is:

```
ALTER field_name TYPE new_type
| ( ADD field_name new_type )
| ( RENAME field_name TO field_name )
   ( AND field_name TO field_name ) ...
```

field_name is an arbitrary identifier for the field.

new_type is an identifier other than the reserved type names.

## Synopsis legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

## Description

ALTER TYPE can change a user-defined type in the following ways:

- Change the type of an existing field.
- Append a new field to an existing type.
- Rename a field defined by the type.

First, after the ALTER TYPE keywords, specify the name of the user-defined type to be changed, followed by the type of change: ALTER, ADD, or RENAME. Next, provide the rest of the needed information, as explained in the following sections.

## Changing the type of a field

To change the type of a field, the field must already exist in type definition and its type should be compatible with the new type. You can change an ascii type to text. You cannot change text (varchar) to ascii because every UTF8 string is not ascii. You can convert text to blobs. You cannot convert a blob to text because not every blob is not a UTF8 string. This example shows changing the type of the model field from ascii to text and then to blob.

```
CREATE TYPE version (
  model ascii,
  version_number int
);
```

```
ALTER TYPE version ALTER model TYPE text;
ALTER TYPE version ALTER model TYPE blob;
```

You cannot change the type of these columns:

- Clustering columns because doing so induces the on-disk ordering of rows
- Indexed columns

## Adding a field to a type

To add a new field to a type, use ALTER TYPE and the ADD keyword.

```
ALTER TYPE version ADD release_date timestamp;
```

To add a collection map field called point_release in this example that represents the release date and decimal designator, use this syntax:

```
ALTER TYPE version ADD point_release map<timestamp, decimal>;
```

### Renaming a field of a type

To change the name of a field of a user-defined type, use the RENAME old_name TO new_name syntax. You can't use different keyspaces prefixes for the old and new names. Make multiple changes to field names of a type by appending AND old_name TO new_name to the command.

```
ALTER TYPE version RENAME model TO sku;
ALTER TYPE version RENAME sku TO model AND version_number TO num
```

## ALTER USER

Alter existing user options.

### Synopsis

```
ALTER USER user_name
  WITH PASSWORD 'password' ( NOSUPERUSER | SUPERUSER )
```

### Synopsis legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

### Description

Superusers can change a user's password or superuser status. To prevent disabling all superusers, superusers cannot change their own superuser status. Ordinary users can change only their own password. Enclose the user name in single quotation marks if it contains non-alphanumeric characters. Enclose the password in single quotation marks.

### Examples

```
ALTER USER moss WITH PASSWORD 'bestReceiver';
```

## BATCH

Write multiple DML statements.

### Synopsis

Cassandra 2.1 and later

```
BEGIN UNLOGGED  BATCH
  USING TIMESTAMP timestamp
  dml_statement;
  dml_statement;
  ...
APPLY BATCH;
```

Cassandra 2.0.x

```
BEGIN ( UNLOGGED | COUNTER ) BATCH
  USING TIMESTAMP timestamp
  dml_statement;
  dml_statement;
  ...
APPLY BATCH;
```

dml_statement is:

- INSERT
- UPDATE
- DELETE

## Synopsis legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

## Description

A BATCH statement combines multiple data modification language (DML) statements (INSERT, UPDATE, DELETE) into a single logical operation, and sets a client-supplied timestamp for all columns written by the statements in the batch. Batching multiple statements can save network exchanges between the client/server and server coordinator/replicas. However, because of the distributed nature of Cassandra, spread requests across nearby nodes as much as possible to optimize performance. Using batches to optimize performance is usually not successful, as described in Using and misusing batches section. For information about the fastest way to load data, see "Cassandra: Batch loading without the Batch keyword."

Batches are atomic by default. In the context of a Cassandra batch operation, atomic means that if any of the batch succeeds, all of it will. To achieve atomicity, Cassandra first writes the serialized batch to the batchlog system table that consumes the serialized batch as blob data. When the rows in the batch have been successfully written and persisted (or hinted) the batchlog data is removed. There is a performance penalty for atomicity. If you do not want to incur this penalty, prevent Cassandra from writing to the batchlog system by using the UNLOGGED option: BEGIN UNLOGGED BATCH

Although an atomic batch guarantees that if any part of the batch succeeds, all of it will, no other transactional enforcement is done at the batch level. For example, there is no batch isolation. Clients are able to read the first updated rows from the batch, while other rows are still being updated on the server. However, transactional row updates within a partition key are isolated: clients cannot read a partial update.

Statement order does not matter within a batch; Cassandra applies all rows using the same timestamp. Use client-supplied timestamps to achieve a particular order.

## Using a timestamp

BATCH supports setting a client-supplied timestamp, an integer, in the USING clause with one exception: if a DML statement in the batch contains a compare-and-set (CAS) statement, such as the following statement, do not attempt to use a timestamp:

```
INSERT INTO users (id, lastname) VALUES (999, 'Sparrow')  IF NOT EXISTS
```

The timestamp applies to all statements in the batch. If not specified, the current time of the insertion (in microseconds) is used. The individual DML statements inside a BATCH can specify a timestamp if one is not specified in the USING clause.

For example, specify a timestamp in an INSERT statement.

```
BEGIN BATCH
  INSERT INTO purchases (user, balance) VALUES ('user1', -8) USING TIMESTAMP
 19998889022757000;
  INSERT INTO purchases (user, expense_id, amount, description, paid)
    VALUES ('user1', 1, 8, 'burrito', false);
APPLY BATCH;
```

Verify that balance column has the client-provided timestamp.

```
SELECT balance, WRITETIME(balance) FROM PURCHASES;
```

```
 balance | writetime_balance
---------+-------------------
      -8 | 19998889022757000
```

## Batching conditional updates

In Cassandra 2.0.6 and later, you can batch conditional updates introduced as lightweight transactions in Cassandra 2.0. Only updates made to the same partition can be included in the batch because the underlying Paxos implementation works at the granularity of the partition. You can group updates that have conditions with those that do not, but when a single statement in a batch uses a condition, the entire batch is committed using a single Paxos proposal, as if all of the conditions contained in the batch apply. This example shows batching of conditional updates:

The statements for inserting values into purchase records use the IF conditional clause.

```
BEGIN BATCH
  INSERT INTO purchases (user, balance) VALUES ('user1', -8) IF NOT EXISTS;
  INSERT INTO purchases (user, expense_id, amount, description, paid)
    VALUES ('user1', 1, 8, 'burrito', false);
APPLY BATCH;

BEGIN BATCH
  UPDATE purchases SET balance = -208 WHERE user='user1' IF balance = -8;
  INSERT INTO purchases (user, expense_id, amount, description, paid)
    VALUES ('user1', 2, 200, 'hotel room', false);
APPLY BATCH;
```

A continuation of this example shows how to use a static column with conditional updates in batch.

## Batching counter updates

In Cassandra 2.1 and later, batches of counters should use UNLOGGED because, unlike other writes in Cassandra, counter updates are not an idempotent operation.

In Cassandra 2.0, use BEGIN COUNTER BATCH in a batch statement for batched counter updates.

## Cassandra 2.1 Example

```
BEGIN UNLOGGED BATCH
  UPDATE UserActionCounts SET total = total + 2 WHERE keyalias = 523;
  UPDATE AdminActionCounts SET total = total + 2 WHERE keyalias = 701;
APPLY BATCH;
```

### Cassandra 2.0 Example

```
BEGIN COUNTER BATCH
  UPDATE UserActionCounts SET total = total + 2 WHERE keyalias = 523;
  UPDATE AdminActionCounts SET total = total + 2 WHERE keyalias = 701;
APPLY BATCH;
```

# CREATE INDEX

Define a new index on a single column of a table.

## Synopsis

```
CREATE CUSTOM INDEX IF NOT EXISTS index_name
ON keyspace_name.table_name ( KEYS ( column_name ) )
(USING class_name) (WITH OPTIONS = map)
```

Restrictions: *Using class_name* is allowed only if CUSTOM is used and class_name is a string literal containing a java class name.

index_name is an identifier, enclosed or not enclosed in double quotation marks, excluding reserved words.

map is a map collection, a JSON-style array of literals:

```
{ literal : literal, literal : literal ... }
```

## Synopsis legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

## Description

CREATE INDEX creates a new index on the given table for the named column. Attempting to create an already existing index will return an error unless the IF NOT EXISTS option is used. If you use the option, the statement will be a no-op if the index already exists. Optionally, specify a name for the index itself before the ON keyword. Enclose a single column name in parentheses. It is not necessary for the column to exist on any current rows. The column and its data type must be specified when the table is created, or added afterward by altering the table.

You can use dot notation to specify a keyspace for the table: keyspace name followed by a period followed the name of the table. Cassandra creates the table in the specified keyspace, but does not change the current keyspace; otherwise, if you do not use a keyspace name, Cassandra creates the index for the table within the current keyspace.

If data already exists for the column, Cassandra indexes the data during the execution of this statement. After the index is created, Cassandra indexes new data for the column automatically when new data is inserted.

Cassandra supports creating an index on most columns, including a clustering column of a compound primary key or on the partition (primary) key itself. Cassandra 2.1 and later supports creating an index on

a collection or the key of a collection map. Cassandra rejects an attempt to create an index on both the collection key and value.

Indexing can impact performance greatly. Before creating an index, be aware of when and when not to create an index.

Counter columns cannot be indexed.

Cassandra supports creating a custom index, which is primarily for internal use, and options that apply to the custom index. For example:

```
CREATE CUSTOM INDEX ON users (email) USING 'path.to.the.IndexClass';
CREATE CUSTOM INDEX ON users (email) USING 'path.to.the.IndexClass' WITH
 OPTIONS = {'storage': '/mnt/ssd/indexes/'};
```

## Creating an index on a column

Define a table and then create an index on two of its columns:

```
CREATE TABLE myschema.users (
   userID uuid,
   fname text,
   lname text,
   email text,
   address text,
   zip int,
   state text,
   PRIMARY KEY (userID)
 );

CREATE INDEX user_state
   ON myschema.users (state);

CREATE INDEX ON myschema.users (zip);
```

## Creating an index on a clustering column

Define a table having a composite partition key, and then create an index on a clustering column.

```
CREATE TABLE mykeyspace.users (
   userID uuid,
   fname text,
   lname text,
   email text,
   address text,
   zip int,
   state text,
  PRIMARY KEY ((userID, fname), state)
);

CREATE INDEX ON mykeyspace.users (state);
```

## Creating an index on a collection

In Cassandra 2.1 and later, create an index on a collection column as you would any other column. Enclose the name of the collection column in parentheses at the end of the CREATE INDEX statement. For example, add a collection of phone numbers to the users table to index the data in the phones set.

```
ALTER TABLE users ADD phones set<text>;
```

```
CREATE INDEX ON users (phones);
```

If the collection is a map, Cassandra creates an index on map values. Assume the users table contains this map data from the example of a todo map:

```
{'2014-10-2 12:10' : 'die' }
```

The map value is located to the right of the colon, 'die'. The map key, the timestamp, is located to the left of the colon. You can also create an index on map keys using a slightly different syntax. If an index of the map keys of the collection exists, drop that index before creating an index on the map collection values.

### Creating an index on map keys

In Cassandra 2.1 and later, you can create an index on map collection keys. If an index of the map values of the collection exists, drop that index before creating an index on the map collection keys.

To index map keys, you use the KEYS keyword and map name in nested parentheses. For example, index the collection keys, the timestamps, in the todo map in the users table:

```
CREATE INDEX todo_dates ON users (KEYS(todo));
```

To query the table, you can use CONTAINS KEY in WHERE clauses.

## CREATE KEYSPACE

Define a new keyspace and its replica placement strategy.

### Synopsis

```
CREATE ( KEYSPACE | SCHEMA ) IF NOT EXISTS keyspace_name
WITH REPLICATION = map
AND DURABLE_WRITES = ( true | false )
```

map is a map collection, a JSON-style array of literals:

```
{ literal : literal, literal : literal ... }
```

### Synopsis legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

### Description

CREATE KEYSPACE creates a top-level namespace and sets the keyspace name, replica placement strategy class, replication factor, and DURABLE_WRITES options for the keyspace. For information about the replica placement strategy, see Cassandra 2.1 replica placement strategy or Cassandra 2.0 replica placement strategy.

When you configure NetworkTopologyStrategy as the replication strategy, you set up one or more virtual data centers. Alternatively, you use the default data center. Use the same names for data centers as

those used by the snitch. For information about the snitch, see Cassandra 2.1 snitch documentation or Cassandra 2.0 snitch documentation.

You assign different nodes, depending on the type of workload, to separate data centers. For example, assign Hadoop nodes to one data center and Cassandra real-time nodes to another. Segregating workloads ensures that only one type of workload is active per data center. The segregation prevents incompatibility problems between workloads, such as different batch requirements that affect performance.

A map of properties and values defines the two different types of keyspaces:

```
{ 'class' : 'SimpleStrategy', 'replication_factor' : <integer> };

{ 'class' : 'NetworkTopologyStrategy'[, '<data center>' : <integer>, '<data
 center>' : <integer>] . . . };
```

**Table: Table of map properties and values**

| Property | Value | Value Description |
|---|---|---|
| 'class' | 'SimpleStrategy' or 'NetworkTopologyStrategy' | Required. The name of the replica placement strategy class for the new keyspace. |
| 'replication_factor' | <number of replicas> | Required if class is SimpleStrategy; otherwise, not used. The number of replicas of data on multiple nodes. |
| '<first data center>' | <number of replicas> | Required if class is NetworkTopologyStrategy and you provide the name of the first data center. This value is the number of replicas of data on each node in the first data center. Example |
| '<next data center>' | <number of replicas> | Required if class is NetworkTopologyStrategy and you provide the name of the second data center. The value is the number of replicas of data on each node in the data center. |
| . . . | . . . | More replication factors for optional named data centers. |

CQL property map keys must be lower case. For example, class and replication_factor are correct. Keyspace names are 48 or fewer alpha-numeric characters and underscores, the first of which is an alpha character. Keyspace names are case-insensitive. To make a name case-sensitive, enclose it in double quotation marks.

You can use the alias CREATE SCHEMA instead of CREATE KEYSPACE. Attempting to create an already existing keyspace will return an error unless the IF NOT EXISTS option is used. If the option is used, the statement will be a no-op if the keyspace already exists.

## Example of setting the SimpleStrategy class

To construct the CREATE KEYSPACE statement, first declare the name of the keyspace, followed by the WITH REPLICATION keywords and the equals symbol. The name of the keyspace is case insensitive unless enclosed in double quotation marks. Next, to create a keyspace that is not optimized for multiple

data centers, use SimpleStrategy for the class value in the map. Set replication_factor properties, separated by a colon and enclosed in curly brackets. For example:

```
CREATE KEYSPACE Excelsior
  WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' :
 3 };
```

Using SimpleStrategy is fine for evaluating Cassandra. For production use or for use with mixed workloads, use NetworkTopologyStrategy.

## Example of setting the NetworkToplogyStrategy class

Using NetworkTopologyStrategy is also fine for evaluating Cassandra. To use NetworkTopologyStrategy for evaluation purposes using, for example, a single node cluster, specify the default data center name of the cluster. To determine the default data center name, use `nodetool status`.

```
$ nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address    Load       Tokens  Owns    Host ID
    Rack
UN  127.0.0.1  46.59 KB   256     100.0%  dd867d15-6536-4922-b574-
e22e75e46432  rack1
```

Cassandra uses datacenter1 as the default data center name. Create a keyspace named NTSkeyspace on a single node cluster, for example:

```
CREATE KEYSPACE NTSkeyspace WITH REPLICATION = { 'class' :
 'NetworkTopologyStrategy', 'datacenter1' : 1 };
```

To use NetworkTopologyStrategy with data centers in a production environment, you need to change the default snitch, SimpleSnitch, to a network-aware snitch, define one or more data center names in the snitch properties file, and use those data center name(s) to define the keyspace; otherwise, Cassandra will fail to find a node, to complete a write request, such as inserting data into a table.

After configuring Cassandra to use a network-aware snitch, such as the PropertyFileSnitch, you define data center and rack names in the `cassandra-topology.properties` file.

Construct the CREATE KEYSPACE statement using NetworkTopologyStrategy for the class value in the map. Set one or more key-value pairs consisting of the data center name and number of replicas per data center, separated by a colon and enclosed in curly brackets. For example:

```
CREATE KEYSPACE "Excalibur"
  WITH REPLICATION = {'class' : 'NetworkTopologyStrategy', 'dc1' : 3,
 'dc2' : 2};
```

This example sets three replicas for a data center named dc1 and two replicas for a data center named dc2. The data center name you use depends on the cluster-configured snitch you are using. There is a correlation between the data center name defined in the map and the data center name as recognized by the snitch you are using. The `nodetool status` command prints out data center names and rack locations of your nodes if you are not sure what they are.

### Setting DURABLE_WRITES

You can set the DURABLE_WRITES option after the map specification of the CREATE KEYSPACE command. When set to false, data written to the keyspace bypasses the commit log. Be careful using this option because you risk losing data. Do not set this attribute on a keyspace using the SimpleStrategy.

```
CREATE KEYSPACE Risky
  WITH REPLICATION = { 'class' : 'NetworkTopologyStrategy',
  'datacenter1' : 3 } AND DURABLE_WRITES = false;
```

### Checking created keyspaces

Check that the keyspaces were created:

```
SELECT * FROM system.schema_keyspaces;
```

```
keyspace_name  | durable_writes | strategy_class
           | strategy_options
---------------+----------------
+-------------------------------------------------------
+-----------------------------
     excelsior |           True |
 org.apache.cassandra.locator.SimpleStrategy | {"replication_factor":"3"}
     Excalibur |           True |
 org.apache.cassandra.locator.NetworkTopologyStrategy |
 {"dc2":"2","dc1":"3"}
         risky |          False |
 org.apache.cassandra.locator.NetworkTopologyStrategy |
 {"datacenter1":"1"}
        system |           True |
 org.apache.cassandra.locator.LocalStrategy |                         {}
 system_traces |           True |
 org.apache.cassandra.locator.SimpleStrategy | {"replication_factor":"1"}

(5 rows)
```

Cassandra converted the excelsior keyspace to lowercase because quotation marks were not used to create the keyspace and retained the initial capital letter for the Excalibur because quotation marks were used.

**Related information**

Cassandra 2.1 replication strategy
Cassandra 2.0 replication strategy
Cassandra 2.1 snitch configuraition
Cassandra 2.0 snitch configuration
Cassandra 2.1 property file snitch
Cassandra 2.0 property file snitch

## CREATE TABLE

Define a new table.

### Synopsis

```
CREATE TABLE IF NOT EXISTS keyspace_name.table_name
( column_definition, column_definition, ...)
```

```
WITH property AND property ...
```

Cassandra 2.1.3+ column_definition is:

```
column_name cql_type STATIC PRIMARY KEY
| column_name <tuple<tuple_type> tuple<tuple_type>... > PRIMARY KEY
| column_name frozen<user-defined_type> PRIMARY KEY
| column_name frozen<collection_name><collection_type>... PRIMARY KEY
| ( PRIMARY KEY ( partition_key ) )
```

**Note:** You cannot use non-frozen collections for primary key columns. However, you can use frozen collections for primary key columns.

Cassandra 2.1.0 to 2.1.2 column_definition is:

```
column_name cql_type STATIC PRIMARY KEY
| column_name frozen<tuple<tuple_type> tuple<tuple_type>... > PRIMARY KEY
| column_name frozen<user-defined_type> PRIMARY KEY
| column_name frozen<collection> PRIMARY KEY
| ( PRIMARY KEY ( partition_key ) )
```

Cassandra 2.0.x column_definition is:

```
column_name cql_type STATIC PRIMARY KEY
| ( PRIMARY KEY ( partition_key ) )
```

Restrictions:

- There should always be exactly one primary key definition.
- cql_type of the primary key must be a CQL data type or a user-defined type.
- cql_type of a collection uses this syntax:

  ```
  LIST<cql_type>
  | SET<cql_type>
  | MAP<cql_type, cql_type>
  ```

- In Cassandra 2.1 only, tuple and user-defined types require the *frozen* keyword followed by the type in angle brackets.

PRIMARY KEY is:

```
column_name
| ( column_name1, column_name2, column_name3 ...)
| ((column_name4, column_name5), column_name6, column_name7 ...)
```

column_name1 is the partition key.

column_name2, column_name3 ... are clustering columns.

column_name4, column_name5 are partitioning keys.

column_name6, column_name7 ... are clustering columns.

property is a CQL table property, enclosed in single quotation marks in the case of strings, or one of these directives:

```
COMPACT STORAGE
| ( CLUSTERING ORDER BY (clustering_column ( ASC) | DESC ), ...) )
```

## Synopsis legend

- Uppercase means literal

- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

## Description

CREATE TABLE creates a new table under the current keyspace. You can also use the alias CREATE COLUMNFAMILY.

Cassandra 2.1.1 and later supports the IF NOT EXISTS syntax for creating a trigger. Attempting to create an existing table returns an error unless the IF NOT EXISTS option is used. If the option is used, the statement if a no-op if the table already exists.

Valid table names are strings of alphanumeric characters and underscores, which begin with a letter. You can use dot notation to specify a keyspace for the table: keyspace name followed by a period followed the name of the table, Cassandra creates the table in the specified keyspace, but does not change the current keyspace; otherwise, if you do not use a keyspace name, Cassandra creates the table within the current keyspace.

In Cassandra 2.0.6 and later, you can use a static column to store the same data in multiple clustered rows of a partition, and then retrieve that data with a single SELECT statement.

You can add a counter column, which has been improved in Cassandra 2.1, to a table.

## Defining a column

You assign a type to columns during table creation. Column types, other than collection-type columns, are specified as a parenthesized, comma-separated list of column name and type pairs.

This example shows how to create a table that includes collection-type columns: map, set, and list.

```
CREATE TABLE users (
  userid text PRIMARY KEY,
  first_name text,
  last_name text,
  emails set<text>,
  top_scores list<int>,
  todo map<timestamp, text>
);
```

## Defining columns of the user-defined, tuple types, and collections

To support future capabilities, a column definition of a user-defined or tuple type requires the frozen keyword. Cassandra serializes a frozen value having multiple components into a single value. For examples and usage information, see "Using a user-defined type", "Tuple type", and Collection type on page 57.

**Note:** Cassandra 2.1.0 to 2.1.2 requires using frozen for tuples:

```
frozen <tuple <int, tuple<text, double>>>
```

Cassandra 2.1.3+ does not require this keyword for tuples.

## Ordering results

You can order query results to make use of the on-disk sorting of columns. You can order results in ascending or descending order. The ascending order will be more efficient than descending. If you need

results in descending order, you can specify a clustering order to store columns on disk in the reverse order of the default. Descending queries will then be faster than ascending ones.

The following example shows a table definition that changes the clustering order to descending by insertion time.

```
CREATE TABLE timeseries (
  event_type text,
  insertion_time timestamp,
  event blob,
  PRIMARY KEY (event_type, insertion_time)
)
WITH CLUSTERING ORDER BY (insertion_time DESC);
```

## Using compact storage

When you create a table using compound primary keys, for every piece of data stored, the column name needs to be stored along with it. Instead of each non-primary key column being stored such that each column corresponds to one column on disk, an entire row is stored in a single column on disk. If you need to conserve disk space, use the WITH COMPACT STORAGE directive that stores data in the legacy (Thrift) storage engine format.

```
CREATE TABLE sblocks (
  block_id uuid,
  subblock_id uuid,
  data blob,
  PRIMARY KEY (block_id, subblock_id)
)
WITH COMPACT STORAGE;
```

Using the compact storage directive prevents you from defining more than one column that is not part of a compound primary key. A compact table using a primary key that is not compound can have multiple columns that are not part of the primary key.

A compact table that uses a compound primary key must define at least one clustering column. Columns cannot be added nor removed after creation of a compact table. Unless you specify WITH COMPACT STORAGE, CQL creates a table with non-compact storage.

## Setting a table property

Using the optional WITH clause and keyword arguments, you can configure caching, compaction, and a number of other operations that Cassandra performs on new table. Use the WITH clause to specify the properties of tables listed in Setting a table property on page 106. Enclose a string property in single quotation marks.

## Creating a table WITH ID

If a table is accidentally dropped with DROP TABLE, this option can be used to recreate the table and run a commitlog replay to retrieve the data.

```
CREATE TABLE users (
        userid text PRIMARY KEY,
        emails set<text>
        ) WITH ID='5a1c395e-b41f-11e5-9f22-ba0be0483c18';
```

## Defining a primary key column

The only schema information that must be defined for a table is the primary key and its associated data type. Unlike earlier versions, CQL does not require a column in the table that is not part of the primary key. A primary key can have any number (1 or more) of component columns.

If the primary key consists of only one column, you can use the keywords, PRIMARY KEY, after the column definition:

```
CREATE TABLE users (
  user_name varchar PRIMARY KEY,
  password varchar,
  gender varchar,
  session_token varchar,
  state varchar,
  birth_year bigint
);
```

Alternatively, you can declare the primary key consisting of only one column in the same way as you declare a compound primary key. Do not use a counter column for a key.

## Setting a table property

Using the optional WITH clause and keyword arguments, you can configure caching, compaction, and a number of other operations that Cassandra performs on new table. You can use the WITH clause to specify the properties of tables listed in CQL table properties, including caching, table comments, compression, and compaction. Format the property as either a string or a map. Enclose a string property in single quotation marks. For example, to embed a comment in a table, you format the comment as a string property:

```
CREATE TABLE MonkeyTypes (
  block_id uuid,
  species text,
  alias text,
  population varint,
  PRIMARY KEY (block_id)
)
WITH comment='Important biological records'
AND read_repair_chance = 1.0;
```

To configure compression and compaction, you use property maps:

```
CREATE TABLE DogTypes (
  block_id uuid,
  species text,
  alias text,
  population varint,
  PRIMARY KEY (block_id)
) WITH compression =
    { 'sstable_compression' : 'DeflateCompressor', 'chunk_length_kb' : 64 }
  AND compaction =
    { 'class' : 'SizeTieredCompactionStrategy', 'min_threshold' : 6 };
```

To specify using compact storage or clustering order use the WITH clause.

To configure caching in Cassandra 2.1, you also use a property map.

```
// Cassandra 2.1
```

```
CREATE TABLE DogTypes (
             ...   block_id uuid,
             ...   species text,
             ...   alias text,
             ...   population varint,
             ...   PRIMARY KEY (block_id)
             ... ) WITH caching = '{ 'keys' : 'NONE', 'rows_per_partition' :
 '120' }';
```

To configure caching in Cassandra 2.0.x, you do not use a property map. Simply set the caching property to a value:

```
// Cassandra 2.0.x only

CREATE TABLE DogTypes (
  block_id uuid,
  species text,
  alias text,
  population varint,
  PRIMARY KEY (block_id)
) WITH caching = 'keys_only';
```

**Important:** In Cassandra 2.0.x, use row caching with caution.

## Using a compound primary key

As shown in the music service example, a compound primary key consists of more than one column and treats the first column declared in a definition as the partition key. To create a compound primary key, use the keywords, PRIMARY KEY, followed by the comma-separated list of column names enclosed in parentheses.

```
CREATE TABLE emp (
  empID int,
  deptID int,
  first_name varchar,
  last_name varchar,
  PRIMARY KEY (empID, deptID)
);
```

## Using a composite partition key

A composite partition key is a partition key consisting of multiple columns. You use an extra set of parentheses to enclose columns that make up the composite partition key. The columns within the primary key definition but outside the nested parentheses are clustering columns. These columns form logical sets inside a partition to facilitate retrieval.

```
CREATE TABLE Cats (
  block_id uuid,
  breed text,
  color text,
  short_hair boolean,
  PRIMARY KEY ((block_id, breed), color, short_hair)
);
```

For example, the composite partition key consists of block_id and breed. The clustering columns, color and short_hair, determine the clustering order of the data. Generally, Cassandra will store columns having the same block_id but a different breed on different nodes, and columns having the same block_id and breed on the same node.

## Using clustering order

You can order query results to make use of the on-disk sorting of columns. You can order results in ascending or descending order. The ascending order will be more efficient than descending. If you need results in descending order, you can specify a clustering order to store columns on disk in the reverse order of the default. Descending queries will then be faster than ascending ones.

The following example shows a table definition that changes the clustering order to descending by insertion time.

```
create table timeseries (
  event_type text,
  insertion_time timestamp,
  event blob,
  PRIMARY KEY (event_type, insertion_time)
)
WITH CLUSTERING ORDER BY (insertion_time DESC);
```

## Sharing a static column

In a table that uses clustering columns, non-clustering columns can be declared static in the table definition. Static columns are only static within a given partition.

```
CREATE TABLE t (
  k text,
  s text STATIC,
  i int,
  PRIMARY KEY (k, i)
);
INSERT INTO t (k, s, i) VALUES ('k', 'I''m shared', 0);
INSERT INTO t (k, s, i) VALUES ('k', 'I''m still shared', 1);
SELECT * FROM t;
```

Output is:

```
 k |                  s | i
---------------------------
k  | "I'm still shared" | 0
k  | "I'm still shared" | 1
```

**Restrictions**

- A table that does not define any clustering columns cannot have a static column. The table having no clustering columns has a one-row partition in which every column is inherently static.
- A table defined with the COMPACT STORAGE directive cannot have a static column.
- A column designated to be the partition key cannot be static.

You can batch conditional updates to a static column.

In Cassandra 2.0.9 and later, you can use the DISTINCT keyword to select static columns. In this case, Cassandra retrieves only the beginning (static column) of the partition.

# CREATE TRIGGER

Registers a trigger on a table.

## Synopsis

```
CREATE TRIGGER IF NOT EXISTS trigger_name ON table_name
```

```
USING 'java_class'
```

### Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

### Description

The implementation of triggers includes the capability to register a trigger on a table using the familiar CREATE TRIGGER syntax. The Trigger API is semi-private and subject to change.

```
CREATE TRIGGER myTrigger
  ON myTable
  USING 'org.apache.cassandra.triggers.AuditTrigger'
```

In Cassandra 2.1 and later, you need to enclose trigger names that use uppercase characters in double quotation marks. The logic comprising the trigger can be written in any Java (JVM) language and exists outside the database. The Java class in this example that implements the trigger is named `org.apache.cassandra.triggers` and defined in an Apache repository. The trigger defined on a table fires before a requested DML statement occurs to ensures the atomicity of the transaction.

Place the custom trigger code (JAR) in the `triggers` directory on every node. The custom JAR loads at startup. The location of triggers directory depends on the installation:

- Cassandra 2.0.x tarball: *`install_location`*`/lib/triggers`
- Cassandra 2.1.x tarball: *install_location*/conf/triggers
- Datastax Enterprise 4.5 and later: Installer-No Services and tarball: *`install_location`*`/resources/cassandra/conf/triggers`
- Datastax Enterprise 4.5 and later: Installer-Services and packages: `/etc/dse/cassandra/triggers`

Cassandra 2.1.1 and later supports lightweight transactions for creating a trigger. Attempting to create an existing trigger returns an error unless the IF NOT EXISTS option is used. If the option is used, the statement is a no-op if the table already exists.

## CREATE TYPE

Create a user-defined type. Cassandra 2.1 and later.

### Synopsis

```
CREATE TYPE IF NOT EXISTS keyspace.type_name
( field, field, ...)
```

type_name is a type identifier other than the reserved type names.

field is:

```
field_name type
```

field_name is an arbitrary identifier for the field.

type is a CQL collection or non-collection type other than a counter type.

### Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

### Description

A user-defined type is one or more typed fields. A user-defined type facilitates handling multiple fields of related information, such as address information: street, city, and postal code. Attempting to create an already existing type will return an error unless the IF NOT EXISTS option is used. If the option is used, the statement will be a no-op if the type already exists.

To create a user-defined type, use the CREATE TYPE command followed by the name of the type and a list of fields delimited by commas and enclosed in parentheses.

Choose a name for the user-defined type other than reserved type names, such as:

- byte
- smallint
- complex
- enum
- date
- interval
- macaddr
- bitstring

If you are in the system keyspace, which is the keyspace when you launch cqlsh, you need to specify a keyspace for the type. You can use dot notation to specify a keyspace for the type: keyspace name followed by a period followed the name of the type. Cassandra creates the type in the specified keyspace, but does not change the current keyspace; otherwise, if you do not specify a keyspace, Cassandra creates the type within the current keyspace.

### Example

This example creates a user-defined type, address, that consists of address and phone number information.

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

After defining the address type, you can create a table having a column of that type. CQL collection columns and other columns support the use of user-defined types, as shown in Using CQL examples.

## CREATE USER

Create a new user.

## Synopsis

```
CREATE USER IF NOT EXISTS user_name WITH PASSWORD 'password'
( NOSUPERUSER | SUPERUSER )
```

## Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

## Description

CREATE USER defines a new database user account. By default users accounts do not have superuser status. Only a superuser can issue CREATE USER requests.

User accounts are required for logging in under internal authentication and authorization.

Enclose the user name in single quotation marks if it contains non-alphanumeric characters. You cannot recreate an existing user. To change the superuser status or password, use ALTER USER.

## Creating internal user accounts

You need to use the WITH PASSWORD clause when creating a user account for internal authentication. Enclose the password in single quotation marks.

```
CREATE USER spillman WITH PASSWORD 'Niner27';
CREATE USER akers WITH PASSWORD 'Niner2' SUPERUSER;
CREATE USER boone WITH PASSWORD 'Niner75' NOSUPERUSER;
```

If internal authentication has not been set up, you do not need the WITH PASSWORD clause:

```
CREATE USER test NOSUPERUSER;
```

## Creating a user account conditionally

In Cassandra 2.0.9 and later, you can test that the user does not have an account before attempting to create one. Attempting to create an existing user results in an invalid query condition unless the IF NOT EXISTS option is used. If the option is used, the statement will be a no-op if the user exists.

```
$ bin/cqlsh -u cassandra -p cassandra
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 2.1.0 | CQL spec 3.2.0 | Native protocol v3]
Use HELP for help.

cqlsh> CREATE USER newuser WITH PASSWORD 'password';

cqlsh> CREATE USER newuser WITH PASSWORD 'password';
code=2200 [Invalid query] message="User newuser already exists"

cqlsh> CREATE USER IF NOT EXISTS newuser WITH PASSWORD 'password';
cqlsh>
```

# DELETE

Removes entire rows or one or more columns from one or more rows.

## Synopsis

```
DELETE column_name, ... | ( column_name term )
FROM keyspace_name.table_name
USING TIMESTAMP integer
WHERE row_specification
( IF ( EXISTS | ( condition(  AND condition ) . . .  ) ) )
```

term is:

```
[ list_position ] | key_value
```

row_specification is one of:

```
primary_key_name = key_value
primary_key_name IN ( key_value, key_value, ...)
```

condition is:

```
column_name = key_value
| column_name [list_position] = key_value
```

## Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

## Description

A DELETE statement removes one or more columns from one or more rows in a table, or it removes the entire row if no columns are specified. Cassandra applies selections within the same partition key atomically and in isolation.

## Deleting columns or a row

After the DELETE keyword, optionally list column names, separated by commas.

```
DELETE col1, col2, col3 FROM Planeteers WHERE userID = 'Captain';
```

When no column names are specified, the entire row(s) specified in the WHERE clause are deleted.

```
DELETE FROM MastersOfTheUniverse WHERE mastersID IN ('Man-At-Arms',
 'Teela');
```

When a column is deleted, it is not removed from disk immediately. The deleted column is marked with a tombstone and then removed after the configured grace period has expired. The optional timestamp defines the new tombstone record.

### Conditionally deleting columns

In Cassandra 2.0.7 and later, you can conditionally delete columns using IF or IF EXISTS. Deleting a column is similar to making an insert or update conditionally. Conditional deletions incur a non-negligible performance cost and should be used sparingly.

### Specifying the table

The table name follows the list of column names and the keyword FROM.

### Deleting old data

You can identify the column for deletion using a timestamp.

```
DELETE email, phone
  FROM users
  USING TIMESTAMP 1318452291034
  WHERE user_name = 'jsmith';
```

The TIMESTAMP input is an integer representing microseconds. The WHERE clause specifies which row or rows to delete from the table.

```
DELETE col1 FROM SomeTable WHERE userID = 'some_key_value';
```

This form provides a list of key names using the IN notation and a parenthetical list of comma-delimited key names.

```
DELETE col1 FROM SomeTable WHERE userID IN (key1, key2);
      DELETE phone FROM users WHERE user_name IN ('jdoe', 'jsmith');
```

In Cassandra 2.0 and later, CQL supports an empty list of values in the IN clause, useful in Java Driver applications when passing empty arrays as arguments for the IN clause.

### Using a collection set, list or map

To delete an element from the map, use the DELETE command and enclose the key of the element in square brackets:

```
DELETE todo ['2012-9-24'] FROM users WHERE user_id = 'frodo';
```

To remove an element from a list, use the DELETE command and the list index position in square brackets:

```
DELETE top_places[3] FROM users WHERE user_id = 'frodo';
```

To remove all elements from a set, you can use the DELETE statement:

```
DELETE emails FROM users WHERE user_id = 'frodo';
```

## DESCRIBE

Provides information about the connected Cassandra cluster, or about the data objects stored in the cluster.

### Synopsis

```
DESCRIBE FULL ( CLUSTER | SCHEMA )
| KEYSPACES
```

```
| ( KEYSPACE keyspace_name )
| TABLES
| ( TABLE table_name )
| TYPES
| ( TYPE user_defined_type )
| INDEX
| ( INDEX index_name )
```

## Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

## Description

The DESCRIBE or DESC command outputs information about the connected Cassandra cluster, or about the data stored on it. To query the system tables directly, use SELECT.

The keyspace and table name arguments are case-sensitive and need to match the upper or lowercase names stored internally. Use the DESCRIBE commands to list objects by their internal names. Use DESCRIBE FULL SCHEMA if you need the schema of system_* keyspaces.

DESCRIBE functions in the following ways:

| DESCRIBE commands | Example | Description |
|---|---|---|
| DESCRIBE CLUSTER | DESCRIBE CLUSTER; | Output information about the connected Cassandra cluster. Cluster name, partitioner, and snitch are output. For non-system keyspace, the endpoint-range ownership information is also shown. |
| DESCRIBE KEYSPACES | | Output a list of all keyspace names. |
| DESCRIBE KEYSPACE <keyspace_name> | DESCRIBE KEYSPACE cycling; | Output CQL commands for the given keyspace. These CQL commands can be used to recreate the keyspace and tables. |
| DESCRIBE [FULL] SCHEMA | | Output CQL commands for entire non-system keyspace and table schema. Use the FULL option to also include system keyspaces. |
| DESCRIBE TABLES | | Output all tables in the current keyspace, or in all keyspaces if there is not current keyspace. |
| DESCRIBE TABLE <table_name> | DESCRIBE TABLE upcoming_calendar; | Output CQL commands for the given table. This CQL command can be used to recreate the table. |
| DESCRIBE INDEX <index_name> | DESCRIBE INDEX team_entry; | Output CQL command for the given index. This CQL command can be used to recreate the index. |
| DESCRIBE TYPES | | Output list of all user-defined types in the current keyspace. |
| DESCRIBE TYPE <type_name> | DESCRIBE TYPE basic_info; | Output CQL command for the given user-defined type. This CQL command can be used to recreate the index. |

# DROP INDEX

Drop the named index.

## Synopsis

```
DROP INDEX IF EXISTS keyspace.index_name
```

## Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

## Description

A DROP INDEX statement removes an existing index. If the index was not given a name during creation, the index name is <table_name>_<column_name>_idx. If the index does not exists, the statement will return an error, unless IF EXISTS is used in which case the operation is a no-op. You can use dot notation to specify a keyspace for the index you want to drop: keyspace name followed by a period followed the name of the index. Cassandra drops the index in the specified keyspace, but does not change the current keyspace; otherwise, if you do not use a keyspace name, Cassandra drops the index for the table within the current keyspace.

## Example

```
DROP INDEX user_state;

DROP INDEX users_zip_idx;

DROP INDEX myschema.users_state;
```

# DROP KEYSPACE

Remove the keyspace.

## Synopsis

```
DROP ( KEYSPACE | SCHEMA ) IF EXISTS keyspace_name
```

## Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

### Description

A DROP KEYSPACE statement results in the immediate, irreversible removal of a keyspace, including all tables and data contained in the keyspace. You can also use the alias DROP SCHEMA. If the keyspace does not exists, the statement will return an error unless IF EXISTS is used, in which case the operation is a no-op.

Cassandra takes a snapshot of the keyspace before dropping it. In Cassandra 2.0.4 and earlier, the user was responsible for removing the snapshot manually.

### Example

```
DROP KEYSPACE MyTwitterClone;
```

## DROP TABLE

Remove the named table.

### Synopsis

```
DROP TABLE IF EXISTS keyspace_name.table_name
```

### Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

### Description

A DROP TABLE statement results in the immediate, irreversible removal of a table, including all data contained in the table. You can also use the alias DROP COLUMNFAMILY.

### Example

```
DROP TABLE worldSeriesAttendees;
```

## DROP TRIGGER

Removes registration of a trigger.

### Synopsis

```
DROP TRIGGER IF EXISTS trigger_name ON table_name
```

### Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

### Description

The DROP TRIGGER statement removes the registration of a trigger created using CREATE TRIGGER. Cassandra 2.1.1 and later supports the IF EXISTS syntax for dropping a trigger. Cassandra checks for the existence of the trigger before dropping it.

The Trigger API is semi-private and subject to change.

## DROP TYPE

Drop a user-defined type. Cassandra 2.1 and later.

### Synopsis

```
DROP TYPE IF EXISTS type_name
```

type_name is the name of a user-defined type.

### Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

### Description

This statement immediately and irreversibly removes a type. To drop a type, use ALTER TYPE and the DROP keyword as shown in the following example. Attempting to drop a type that does not exist will return an error unless the IF EXISTS option is used. If the option is used, the statement will be a no-op if the type already exists. Dropping a user-defined type that is in use by a table or another type is not allowed.

```
DROP TYPE version;
```

## DROP USER

Remove a user.

### Synopsis

```
DROP USER IF EXISTS user_name
```

### Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

### Description

DROP USER removes an existing user. In Cassandra 2.0.9 and later, you can test that the user exists. Attempting to drop a user that does not exist results in an invalid query condition unless the IF EXISTS option is used. If the option is used, the statement will be a no-op if the user does not exist. You have to be logged in as a superuser to issue a DROP USER statement. Users cannot drop themselves.

Enclose the user name in single quotation marks only if it contains non-alphanumeric characters.

## GRANT

Provide access to database objects.

### Synopsis

```
GRANT permission_name PERMISSION
| ( GRANT ALL PERMISSIONS ) ON resource TO user_name
```

permission_name is one of these:

- ALL
- ALTER
- AUTHORIZE
- CREATE
- DROP
- MODIFY
- SELECT

resource is one of these:

- ALL KEYSPACES
- KEYSPACE keyspace_name
- TABLE *keyspace_name.*table_name

### Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

## Description

Permissions to access all keyspaces, a named keyspace, or a table can be granted to a user. Enclose the user name in single quotation marks if it contains non-alphanumeric characters.

This table lists the permissions needed to use CQL statements:

**Table: CQL Permissions**

| Permission | CQL Statement |
|---|---|
| ALL | All statements |
| ALTER | ALTER KEYSPACE, ALTER TABLE, CREATE INDEX, DROP INDEX |
| AUTHORIZE | GRANT, REVOKE |
| CREATE | CREATE KEYSPACE, CREATE TABLE |
| DROP | DROP KEYSPACE, DROP TABLE |
| MODIFY | INSERT, DELETE, UPDATE, TRUNCATE |
| SELECT | SELECT |

To be able to perform SELECT queries on a table, you have to have SELECT permission on the table, on its parent keyspace, or on ALL KEYSPACES. To be able to CREATE TABLE you need CREATE permission on its parent keyspace or ALL KEYSPACES. You need to be a superuser or to have AUTHORIZE permission on a resource (or one of its parents in the hierarchy) plus the permission in question to be able to GRANT or REVOKE that permission to or from a user. GRANT, REVOKE and LIST permissions check for the existence of the table and keyspace before execution. GRANT and REVOKE check that the user exists.

## Examples

Give spillman permission to perform SELECT queries on all tables in all keyspaces:

```
GRANT SELECT ON ALL KEYSPACES TO spillman;
```

Give akers permission to perform INSERT, UPDATE, DELETE and TRUNCATE queries on all tables in the field keyspace.

```
GRANT MODIFY ON KEYSPACE field TO akers;
```

Give boone permission to perform ALTER KEYSPACE queries on the forty9ers keyspace, and also ALTER TABLE, CREATE INDEX and DROP INDEX queries on all tables in forty9ers keyspace:

```
GRANT ALTER ON KEYSPACE forty9ers TO boone;
```

Give boone permission to run all types of queries on ravens.plays table.

```
GRANT ALL PERMISSIONS ON ravens.plays TO boone;
```

Grant access to a keyspace to just one user, assuming nobody else has ALL KEYSPACES access.

```
GRANT ALL ON KEYSPACE keyspace_name TO user_name;
```

# INSERT

Add or update columns.

## Synopsis

```
INSERT INTO keyspace_name.table_name
  ( identifier, column_name...)
  VALUES ( value, value ... ) IF NOT EXISTS
  USING option AND option
```

Value is one of:

- a literal
- a set

```
{ literal, literal, . . . }
```

- a list

```
[ literal, literal, . . . ]
```

- a map collection, a JSON-style array of literals

```
{ literal : literal, literal : literal, . . . }
```

option is one of:

- TIMESTAMP microseconds
- TTL seconds

## Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

## Description

An INSERT writes one or more columns to a record in a Cassandra table atomically and in isolation. No results are returned. You do not have to define all columns, except those that make up the key. Missing columns occupy no space on disk.

If the column exists, it is updated. The row is created if none exists. Use IF NOT EXISTS to perform the insertion only if the row does not already exist. Using IF NOT EXISTS incurs a performance hit associated with using Paxos internally. For information about Paxos, see Cassandra 2.1 documentation or Cassandra 2.0 documentation.

You can qualify table names by keyspace. INSERT does not support counters, but UPDATE does. Internally, insert and update operations are identical.

## Specifying TIMESTAMP and TTL

- Time-to-live (TTL) in seconds

- Timestamp in microseconds

```
INSERT INTO Hollywood.NerdMovies (user_uuid, fan)
  VALUES (cfd66ccc-d857-4e90-b1e5-df98a3d40cd6, 'johndoe')
  USING TTL 86400;
```

TTL input is in seconds. TTL column values are automatically marked as deleted (with a tombstone) after the requested amount of time has expired. TTL marks the inserted values, not the column itself, for expiration. Any subsequent update of the column resets the TTL to the TTL specified in the update. By default, values never expire. You cannot set data in a counter column to expire.

The TIMESTAMP input is in microseconds. If not specified, the time (in microseconds) that the write occurred to the column is used.

```
INSERT INTO cycling.calendar (race_id, race_name, race_start_date,
 race_end_date)
VALUES (200, 'placeholder', '2015-05-27', '2015-05-27')
USING TIMESTAMP 123456789;
```

### Using a collection set or map

To insert data into a collection, enclose values in curly brackets. Set values must be unique. For example:

```
INSERT INTO users (userid, first_name, last_name, emails)
  VALUES('frodo', 'Frodo', 'Baggins', {'f@baggins.com',
 'baggins@gmail.com'});
```

Insert a map named todo to insert a reminder, 'die' on October 2 for user frodo.

```
INSERT INTO users (userid, todo )
  VALUES('frodo', {'2014-10-2 12:10' : 'die' } );
```

Values of items in collections are limited to 64K.

To insert data into a collection column of a user-defined type, enclose components of the type in parentheses within the curly brackets, as shown in "Using a user-defined type."

**Related information**

Cassandra 2.1 tunable consistency

Cassandra 2.0 tunable consistency

## Example of inserting data into playlists

The "Example of a music service" section described the playlists table. This example shows how to insert data into that table.

### Procedure

Use the INSERT command to insert UUIDs for the compound primary keys, title, artist, and album data of the playslists table.

```
INSERT INTO playlists (id, song_order, song_id, title, artist, album)
  VALUES (62c36092-82a1-3a00-93d1-46196ee77204, 1,
  a3e64f8f-bd44-4f28-b8d9-6938726e34d4, 'La Grange', 'ZZ Top', 'Tres
 Hombres');

INSERT INTO playlists (id, song_order, song_id, title, artist, album)
  VALUES (62c36092-82a1-3a00-93d1-46196ee77204, 2,
```

```
  8a172618-b121-4136-bb10-f665cfc469eb, 'Moving in Stereo', 'Fu Manchu',
 'We Must Obey');

INSERT INTO playlists (id, song_order, song_id, title, artist, album)
  VALUES (62c36092-82a1-3a00-93d1-46196ee77204, 3,
  2b09185b-fb5a-4734-9b56-49077de9edbf, 'Outside Woman Blues', 'Back Door
 Slam', 'Roll Away');
```

# LIST PERMISSIONS

List permissions granted to a user.

## Synopsis

```
LIST permission_name PERMISSION
| ( LIST ALL PERMISSIONS )
    ON resource OF user_name
    NORECURSIVE
```

permission_name is one of these:

- ALTER
- AUTHORIZE
- CREATE
- DROP
- MODIFY
- SELECT

resource is one of these:

- ALL KEYSPACES
- KEYSPACE keyspace_name
- TABLE *keyspace_name.*table_name

## Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

## Description

Permissions checks are recursive. If you omit the NORECURSIVE specifier, permission on the requests resource and its parents in the hierarchy are shown.

- Omitting the resource name (ALL KEYSPACES, keyspace, or table), lists permissions on all tables and all keyspaces.
- Omitting the user name lists permissions of all users. You need to be a superuser to list permissions of all users. If you are not, you must add

  ```
  OF <myusername>
  ```

- Omitting the NORECURSIVE specifier, lists permissions on the resource and its parent resources.

- Enclose the user name in single quotation marks only if it contains non-alphanumeric characters.

After creating users in and granting the permissions in the GRANT examples, you can list permissions that users have on resources and their parents.

## Example

Assuming you completed the examples in Examples, list all permissions given to akers:

```
LIST ALL PERMISSIONS OF akers;
```

Output is:

```
 username | resource           | permission
----------+--------------------+------------
    akers | <keyspace field>   |     MODIFY
```

List permissions given to all the users:

```
LIST ALL PERMISSIONS;
```

Output is:

```
 username | resource              | permission
----------+-----------------------+------------
    akers |     <keyspace field> |     MODIFY
    boone | <keyspace forty9ers> |      ALTER
    boone | <table ravens.plays> |     CREATE
    boone | <table ravens.plays> |      ALTER
    boone | <table ravens.plays> |       DROP
    boone | <table ravens.plays> |     SELECT
    boone | <table ravens.plays> |     MODIFY
    boone | <table ravens.plays> |  AUTHORIZE
 spillman |      <all keyspaces> |     SELECT
```

List all permissions on the plays table:

```
LIST ALL PERMISSIONS ON ravens.plays;
```

Output is:

```
 username | resource              | permission
----------+-----------------------+------------
    boone | <table ravens.plays> |     CREATE
    boone | <table ravens.plays> |      ALTER
    boone | <table ravens.plays> |       DROP
    boone | <table ravens.plays> |     SELECT
    boone | <table ravens.plays> |     MODIFY
    boone | <table ravens.plays> |  AUTHORIZE
 spillman |      <all keyspaces> |     SELECT
```

List all permissions on the ravens.plays table and its parents:

Output is:

```
LIST ALL PERMISSIONS ON ravens.plays NORECURSIVE;
```

```
 username | resource              | permission
----------+-----------------------+------------
```

```
    boone | <table ravens.plays> |     CREATE
    boone | <table ravens.plays> |      ALTER
    boone | <table ravens.plays> |       DROP
    boone | <table ravens.plays> |     SELECT
    boone | <table ravens.plays> |     MODIFY
    boone | <table ravens.plays> |  AUTHORIZE
```

# LIST USERS

List existing users and their superuser status.

## Synopsis

```
LIST USERS
```

## Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

## Description

Assuming you use internal authentication, created the users in the CREATE USER examples, and have not yet changed the default user, the following example shows the output of LIST USERS.

## Example

LIST USERS;

Output is:

```
 name      | super
-----------+-------
 cassandra |  True
     boone | False
     akers |  True
  spillman | False
```

# LOGIN

## Synopsis

```
cqlsh> LOGIN user_name password
```

## Synopsis Legend

In the synopsis section of each statement, formatting has the following meaning:

- Uppercase means literal

- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

### Description

Use this command to change login information without requiring `cqlsh` to restart. Login using a specified username. If the the password is specified, it will be used. Otherwise, you will be prompted to enter the password.

### Examples

Login as the user cutie with the password patootie.

```
LOGIN cutie patootie
```

## REVOKE

Revoke user permissions.

### Synopsis

```
REVOKE ( permission_name PERMISSION )
| ( REVOKE ALL PERMISSIONS )
ON resource FROM user_name
```

permission_name is one of these:

- ALL
- ALTER
- AUTHORIZE
- CREATE
- DROP
- MODIFY
- SELECT

resource is one of these:

- ALL KEYSPACES
- KEYSPACE keyspace_name
- TABLE *keyspace_name.*table_name

### Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

### Description

Permissions to access all keyspaces, a named keyspace, or a table can be revoked from a user. Enclose the user name in single quotation marks if it contains non-alphanumeric characters.

The table in GRANT lists the permissions needed to use CQL statements:

### Example

```
REVOKE SELECT ON ravens.plays FROM boone;
```

The user boone can no longer perform SELECT queries on the ravens.plays table. Exceptions: Because of inheritance, the user can perform SELECT queries on ravens.plays if one of these conditions is met:

- The user is a superuser.
- The user has SELECT on ALL KEYSPACES permissions.
- The user has SELECT on the ravens keyspace.

## SELECT

Retrieve data from a Cassandra table.

### Synopsis

```
SELECT select_expression
  FROM keyspace_name.table_name
  WHERE relation AND relation ...
  ORDER BY ( clustering_column ( ASC | DESC )...)
  LIMIT n
  ALLOW FILTERING
```

select expression is:

```
selection_list
| DISTINCT selection_list
| ( COUNT ( * | 1 ) )
```

selection_list is one of:

- A list of partition keys (used with DISTINCT)
- ```
  selector AS alias, selector AS alias, ...| *
  ```

  alias is an alias for a column name.

selector is:

```
column name
| ( WRITETIME (column_name) )
| ( TTL (column_name) )
| (function (selector , selector, ...) )
```

function is a timeuuid function, a token function, or a blob conversion function.

relation is:

```
column_name op term
| ( column_name, column_name, ... ) op term-tuple
| column_name IN ( term, ( term ... ) )
| ( column_name, column_name, ... ) IN ( term-tuple, ( term-tuple ... ) )
```
```

```
| TOKEN (column_name, ...) op ( term )
```

op is = | < | > | <= | > | = | CONTAINS | CONTAINS KEY

term-tuple (Cassandra 2.1 and later) is:

```
( term, term, ... )
```

term is

- a constant: string, number, uuid, boolean, hex
- a bind marker (?)
- a function
- set:

  ```
  { literal, literal, ... }
  ```

- list:

  ```
  [ literal, literal, ... ]
  ```

- map:

  ```
  { literal : literal, literal : literal, ... }
  ```

## Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

## Description

A SELECT statement reads one or more records from a Cassandra table. The input to the SELECT statement is the select expression. The output of the select statement depends on the select expression:

**Table: Select Expression Output**

| Select Expression | Output |
|---|---|
| Column of list of columns | Rows having a key value and collection of columns |
| COUNT aggregate function | One row with a column that has the value of the number of rows in the resultset |
| DISTINCT partition key list | Values of columns that are different from other column values |
| WRITETIME function | The date/time that a write to a column occurred |
| TTL function | The remaining time-to-live for a column |

## Specifying columns

The SELECT expression determines which columns, if any, appear in the result. Using the asterisk specifies selection of all columns:

```
SELECT * from People;
```

Columns in big data applications duplicate values. Use the DISTINCT keyword to return only distinct (different) values of partition keys.

## Counting returned rows

A SELECT expression using COUNT(*) returns the number of rows that matched the query. Alternatively, you can use COUNT(1) to get the same result.

Count the number of rows in the users table:

```
SELECT COUNT(*) FROM users;
```

The capability to use an alias for a column name is particularly useful when using a function call on a column, such as dateOf(created_at), in the select expression.

```
SELECT event_id, dateOf(created_at), blobAsText(content) FROM timeline;
```

## Using a column alias

You can define an alias on columns using the AS keyword.

```
SELECT event_id,
  dateOf(created_at) AS creation_date,
  blobAsText(content) AS content
FROM timeline;
```

In the output, columns assume the aesthetically-pleasing name.

```
 event_id                | creation_date              | content
-------------------------+----------------------------+----------------
 550e8400-e29b-41d4-a716 | 2013-07-26 10:44:33+0200 | Some stuff
```

## Specifying rows returned using LIMIT

Using the LIMIT option, you can specify that the query return a limited number of rows.

```
SELECT COUNT(*) FROM big_table LIMIT 50000;
SELECT COUNT(*) FROM big_table LIMIT 200000;
```

The output of these statements if you had 105,291 rows in the database would be: 50000, and 105,291. The cqlsh shell has a default row limit of 10,000. The Cassandra server and native protocol do not limit the number of rows that can be returned, although a timeout stops running queries to protect against running malformed queries that would cause system instability.

## Specifying the table using FROM

The FROM clause specifies the table to query. Optionally, specify a keyspace for the table followed by a period, (.), then the table name. If a keyspace is not specified, the current keyspace is used.

For example, count the number of rows in the IndexInfo table in the system keyspace:

```
SELECT COUNT(*) FROM system."IndexInfo";
```

## Filtering data using WHERE

The WHERE clause specifies which rows to query. In the WHERE clause, refer to a column using the actual name, not an alias. Columns in the WHERE clause need to meet *one of* these requirements:

- The partition key definition includes the column.
- A column that is indexed using CREATE INDEX.

The primary key in the WHERE clause tells Cassandra to race to the specific node that has the data. Put the name of the column to the left of the = or IN operator. Put the column value to the right of the operator. For example, empID and deptID columns are included in the partition key definition in the following table, so you can query all the columns using the empID in the WHERE clause:

```
CREATE TABLE emp (
  empID int,
  deptID int,
  first_name varchar,
  last_name varchar,
  PRIMARY KEY (empID, deptID));

SELECT deptid FROM emp WHERE empid = 104;
```

Cassandra supports these conditional operators in the WHERE clause: CONTAINS, CONTAINS KEY, IN, =, >, >=, <, or <=, but not all in certain situations.

**Restrictions on the use of conditions**

- Non-equal conditional operations on the partition key

  Regardless of the partitioner in use, Cassandra does not support non-equal conditional operations on the partition key. Use the token function for range queries on the partition key.
- Using the IN condition with a compound or composite partition key

  The IN condition is allowed on the last column of the partition key only if you query all preceding columns of key for equality. "Using the IN filter condition" presents examples of using the IN operation.
- Querying an indexed table

  A query on an indexed table must have at least one equality condition on the indexed column, as shown in "Indexing a column".
- Range queries

  Cassandra supports greater-than and less-than comparisons, but for a given partition key, the conditions on the clustering column are restricted to the filters that allow Cassandra to select a contiguous ordering of rows.

  For example:

```
CREATE TABLE ruling_stewards (
  steward_name text,
  king text,
  reign_start int,
  event text,
  PRIMARY KEY (steward_name, king, reign_start)
```

```
);
```

This query constructs a filter that selects data about stewards whose reign started by 2450 and ended before 2500. If king were not a component of the primary key, you would need to create an index on king to use this query:

```
SELECT * FROM ruling_stewards
  WHERE king = 'Brego'
  AND reign_start >= 2450
  AND reign_start < 2500 ALLOW FILTERING;
```

The output is:

```
 steward_name | king  | reign_start | event
--------------+-------+-------------+--------------------
      Boromir | Brego |        2477 |   Attacks continue
       Cirion | Brego |        2489 | Defeat of Balchoth

(2 rows)
```

To allow Cassandra to select a contiguous ordering of rows, you need to include the king component of the primary key in the filter using an equality condition. The ALLOW FILTERING clause is also required. ALLOW FILTERING provides the capability to query the clustering columns using any condition if performance is not an issue.

## ALLOW FILTERING clause

When you attempt a potentially expensive query, such as searching a range of rows, this prompt appears:

```
Bad Request: Cannot execute this query as it might involve data
filtering and thus may have unpredictable performance. If you want
to execute this query despite the performance unpredictability,
use ALLOW FILTERING.
```

To run the query, use the ALLOW FILTERING clause. Imposing a limit using the LIMIT n clause is recommended to reduce memory used. For example:

```
Select * FROM ruling_stewards
  WHERE king = 'none'
  AND reign_start >= 1500
  AND reign_start < 3000 LIMIT 10 ALLOW FILTERING;
```

Critically, LIMIT doesn't protect you from the worst liabilities. For instance, what if there are no entries with no king? Then you have to scan the entire list no matter what LIMIT is.

ALLOW FILTERING will probably become less strict as we collect more statistics on our data. For example, if we knew that 90% of entries have no king we would know that finding 10 such entries should be relatively inexpensive.

## Using the IN filter condition

Use IN, an equals condition operator, in the WHERE clause to specify multiple possible values for a column. For example, select two columns, first_name and last_name, from three rows having employee ids (primary key) 105, 107, or 104:

```
SELECT first_name, last_name FROM emp WHERE empID IN (105, 107, 104);
```

Format values for the IN conditional test as a comma-separated list. The list can consist of a range of column values.

#### Using IN to filter on a compound or composite primary key

The IN condition is recommended on the last column of the partition key only if you query all preceding columns of key for equality. For example:

```
CREATE TABLE parts (part_type text, part_name text, part_num int, part_year
 text, serial_num text, PRIMARY KEY ((part_type, part_name), part_num,
 part_year));

SELECT * FROM parts WHERE part_type='alloy' AND part_name='hubcap' AND
 part_num=1249 AND part_year IN ('2010', '2015');
```

The IN condition will not work on any clustering column except the last one. The following query will not work, since part_year is the final clustering column, not part_num:

```
SELECT * FROM parts WHERE part_type='alloy' AND part_name='hubcap' AND
 part_num IN (1249, 3456);
```

You can omit the equality test for clustering columns other than the last when using IN, but such a query might involve data filtering and thus may have unpredictable performance. Such a query requires use of ALLOW FILTERING. For example:

```
SELECT * FROM parts WHERE part_num=123456 AND part_year IN ('2010', '2015')
 ALLOW FILTERING;
```

CQL supports an empty list of values in the IN clause, useful in Java Driver applications when passing empty arrays as arguments for the IN clause.

### When *not* to use IN

The recommendations about when not to use an index apply to using IN in the WHERE clause. Under most conditions, using IN in the WHERE clause is not recommended. Using IN can degrade performance because usually many nodes must be queried. For example, in a single, local data center cluster with 30 nodes, a replication factor of 3, and a consistency level of LOCAL_QUORUM, a single key query goes out to two nodes, but if the query uses the IN condition, the number of nodes being queried are most likely even higher, up to 20 nodes depending on where the keys fall in the token range.

## Comparing clustering columns

In Cassandra 2.0.6 and later, you can group the partition key and clustering columns and compare the tuple to values for slicing over rows in a partition. For example:

```
SELECT * FROM ruling_stewards WHERE (steward_name, king) = ('Boromir',
 'Brego');
```

The syntax used in the WHERE clause compares records of steward_name and king as a tuple against the Boromir, Brego tuple.

## Paging through unordered results

The TOKEN function can be used with a condition operator on the partition key column to query. The query selects rows based on the token of their partition key rather than on their value. The token of a key depends on the partitioner in use. Use with the RandomPartitioner or Murmur3Partitioner will not give you a meaningful order.

For example, assume you defined this table:

```
CREATE TABLE periods (
  period_name text,
  event_name text,
```

```
  event_date timestamp,
  weak_race text,
  strong_race text,
  PRIMARY KEY (period_name, event_name, event_date)
);
```

After inserting data, this query uses the TOKEN function to find the data using the partition key.

```
SELECT * FROM periods
  WHERE TOKEN(period_name) > TOKEN('Third Age')
  AND TOKEN(period_name) < TOKEN('Fourth Age');
```

## Using compound primary keys and sorting results

ORDER BY clauses can select a single column only. That column has to be the second column in a compound PRIMARY KEY. This also applies to tables with more than two column components in the primary key. Ordering can be done in ascending or descending order, default ascending, and specified with the ASC or DESC keywords.

In the ORDER BY clause, refer to a column using the actual name, not the aliases.

For example, set up the playlists table, which uses a compound primary key, insert the example data, and use this query to get information about a particular playlist, ordered by song_order. You do not need to include the ORDER BY column in the select expression.

```
SELECT * FROM playlists WHERE id = 62c36092-82a1-3a00-93d1-46196ee77204
  ORDER BY song_order DESC LIMIT 50;
```

Output is:

```
id           | song_order | album                  | artist          | song_id     | titl
-------------+------------+------------------------+-----------------+-------------+-----
62c36092... |          4 | No One Rides for Free |      Fu Manchu | 7db1a490... |
62c36092... |          3 |             Roll Away | Back Door Slam | 2b09185b... | Outs
62c36092... |          2 |          We Must Obey |      Fu Manchu | 8a172618... |    M
62c36092... |          1 |          Tres Hombres |         ZZ Top | a3e63f8f... |
```

Or, create an index on playlist artists, and use this query to get titles of Fu Manchu songs on the playlist:

```
CREATE INDEX ON playlists(artist)

SELECT album, title FROM playlists WHERE artist = 'Fu Manchu';
```

Output is:

```
album                  | title
-----------------------+------------------
We Must Obey           | Moving in Stereo
No One Riddes for Free | Ojo Rojo
```

## Filtering a collection set, list, or map

You can query a table containing a collection to retrieve the collection in its entirety. You can also index the collection column, and then use the CONTAINS condition in the WHERE clause to filter the data for a particular value in the collection. Continuing with the music service example, after adding the collection of

tags to the playlists table, adding some tag data, and then indexing the tags, you can filter on 'blues' in the tags set.

```
SELECT album, tags FROM playlists WHERE tags CONTAINS 'blues';
```

```
album                | tags
---------------------+-------------------------
Tres Hombres         |     {"1973", "blues"}
```

After indexing the music venue map, you can filter on map values, such as 'The Fillmore':

```
SELECT * FROM playlists WHERE venue CONTAINS 'The Fillmore';
```

After indexing the collection keys in the venues map, you can filter on map keys.

```
SELECT * FROM playlists WHERE venue CONTAINS KEY '2014-09-22 22:00:00-0700';
```

### Retrieving the date/time a write occurred

Using WRITETIME followed by the name of a column in parentheses returns date/time in microseconds that the column was written to the database.

Retrieve the date/time that a write occurred to the first_name column of the user whose last name is Jones:

```
SELECT WRITETIME (first_name) FROM users WHERE last_name = 'Jones';

 writetime(first_name)
-----------------------
 1353010594789000
```

The writetime output in microseconds converts to November 15, 2012 at 12:16:34 GMT-8

## TRUNCATE

Remove all data from a table.

### Synopsis

```
TRUNCATE keyspace_name.table_name
```

### Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

### Description

A `TRUNCATE` statement results in the immediate, irreversible removal of all data in the named table.

**Note:** The consistency level must be set to `ALL` prior to performing a `TRUNCATE` operation. All replicas must remove the data.

### Examples

Set the consistency level to `ALL` first. `TRUNCATE` or `TRUNCATE TABLE` can be used to remove all data from a named table.

```
CONSISENTENCY ALL;
TRUNCATE user_activity;
```

```
CONSISTENCY ALL;
TRUNCATE TABLE menu_item;
```

## UPDATE

Update columns in a row.

### Synopsis

```
UPDATE keyspace_name.table_name
USING option AND option
SET assignment, assignment, ...
WHERE row_specification
IF column_name = literal AND column_name = literal . . .
IF EXISTS
```

option is one of:

- TIMESTAMP microseconds
- TTL seconds

assignment is one of:

```
column_name = value
set_or_list_item = set_or_list_item ( + | - ) ...
map_name = map_name ( + | - ) ...
map_name = map_name ( + | - ) { map_key : map_value, ... }
column_name [ term ] = value
counter_column_name = counter_column_name ( + | - ) integer
```

set is:

```
{ literal, literal, . . . }
```

list is:

```
[ literal, literal ]
```

map is:

```
{ literal : literal, literal : literal, . . . }
```

term is:

```
[ list_index_position | [ key_value ]
```

row_specification is:

```
primary key name = key_value
primary key name IN (key_value ,...)
```

## Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

## Description

An UPDATE writes one or more column values for a given row to a Cassandra table. No results are returned. A statement begins with the UPDATE keyword followed by a Cassandra table name.

The row is created if none existed before, and updated otherwise. Specify the row to update in the WHERE clause by including all columns composing the partition key. The IN relation is supported only for the last column of the partition key. The UPDATE SET operation is not valid on a primary key field. Specify other column values using SET. To update multiple columns, separate the name-value pairs using commas.

You can invoke a lightweight transaction using UPDATE:

```
UPDATE customer_account
SET customer_email='lauras@gmail.com'
If customer_email='laurass@gmail.com';
```

Use the IF keyword followed by a condition to be met for the update to succeed. Using an IF condition incurs a performance hit associated with using Paxos internally to support linearizable consistency. In an UPDATE statement, all updates within the same partition key are applied atomically and in isolation.

To update a counter column value in a counter table, specify the increment or decrement to the current value of the counter column. Unlike the INSERT command, the UPDATE command supports counters. Otherwise, the update and insert operations are identical internally.

```
UPDATE UserActionCounts SET total = total + 2 WHERE keyalias = 523;
```

In an UPDATE statement, you can specify these options on columns that are not counter columns:

- TTL seconds
- Timestamp microseconds

TTL input is in seconds. TTL column values are automatically marked as deleted (with a tombstone) after the requested amount of time has expired. TTL marks the inserted values, not the column itself, for expiration. Any subsequent update of the column resets the TTL to the TTL specified in the update. By default, values never expire.

The TIMESTAMP input is an integer representing microseconds. If not specified, the time (in microseconds) that the write occurred to the column is used. Each update statement requires a precise set of primary keys to be specified using a WHERE clause. You need to specify all keys in a table having compound and clustering columns. For example, update the value of a column in a table having a compound primary key, userid and url:

```
UPDATE excelsior.clicks USING TTL 432000
  SET user_name = 'bob'
  WHERE userid=cfd66ccc-d857-4e90-b1e5-df98a3d40cd6 AND
    url='http://google.com';
UPDATE Movies SET col1 = val1, col2 = val2 WHERE movieID = key1;
UPDATE Movies SET col3 = val3 WHERE movieID IN (key1, key2, key3);
UPDATE Movies SET col4 = 22 WHERE movieID = key4;
```

CQL supports an empty list of values in the IN clause, useful in Java Driver applications when passing empty arrays as arguments for the IN clause.

## Examples of updating a column

Update a column in several rows at once:

```
UPDATE users
  SET state = 'TX'
  WHERE user_uuid
  IN (88b8fd18-b1ed-4e96-bf79-4280797cba80,
    06a8913c-c0d6-477c-937d-6c1b69a95d43,
    bc108776-7cb5-477f-917d-869c12dfffa8);
```

Update several columns in a single row:

```
UPDATE users
  SET name = 'John Smith',
  email = 'jsmith@cassie.com'
  WHERE user_uuid = 88b8fd18-b1ed-4e96-bf79-4280797cba80;
```

## Updating a counter column

You can increase or decrease the value of a counter column by an arbitrary numeric value though the assignment of an expression that adds or substracts the value. To update the value of a counter column, use the syntax shown in the following example:

```
UPDATE counterks.page_view_counts
  SET counter_value = counter_value + 2
  WHERE url_name='www.datastax.com' AND page_name='home';
```

To use a lightweight transaction on a counter column to ensure accuracy, put one or more counter updates in the batch statement.

## Updating a collection set

To add an element to a set, use the UPDATE command and the addition (+) operator:

```
UPDATE users
  SET emails = emails + {'fb@friendsofmordor.org'} WHERE user_id = 'frodo';
```

To remove an element from a set, use the subtraction (-) operator.

```
UPDATE users
  SET emails = emails - {'fb@friendsofmordor.org'} WHERE user_id = 'frodo';
```

To remove all elements from a set, you can use the UPDATE statement:

```
UPDATE users SET emails = {} WHERE user_id = 'frodo';
```

## Updating a collection map

To set or replace map data, you can use the UPDATE command. Enclose the timestamp and text values in map collection syntax: strings in curly brackets, separated by a colon.

```
UPDATE users
  SET todo = { '2012-9-24' : 'enter mordor',
    '2012-10-2 12:00' : 'throw ring into mount doom' }
```

```
  WHERE user_id = 'frodo';
```

You can also update or set a specific element using the UPDATE command. For example, update a map named todo to insert a reminder, 'die' on October 2 for user frodo.

```
UPDATE users SET todo['2014-10-2 12:10'] = 'die'
  WHERE user_id = 'frodo';
```

You can set the a TTL for each map element:

```
UPDATE users USING TTL <ttl value>
 SET todo['2012-10-1'] = 'find water' WHERE user_id = 'frodo';
```

In Cassandra 2.1.1 and later, you can update the map by adding one or more elements separated by commas:

```
UPDATE users SET todo = todo + { '2012-10-1': 'find water', '2014-12-15':
 'buy presents' } where user_id = 'frodo';
```

You can remove elements from a map in the same way using - instead of +.

### Using a collection list

To insert values into the list.

```
UPDATE users
  SET top_places = [ 'rivendell', 'rohan' ] WHERE user_id = 'frodo';
```

To prepend an element to the list, enclose it in square brackets, and use the addition (+) operator:

```
UPDATE users
  SET top_places = [ 'the shire' ] + top_places WHERE user_id = 'frodo';
```

To append an element to the list, switch the order of the new element data and the list name in the UPDATE command:

```
UPDATE users
  SET top_places = top_places + [ 'mordor' ] WHERE user_id = 'frodo';
```

To add an element at a particular position, use the list index position in square brackets:

```
UPDATE users SET top_places[2] = 'riddermark' WHERE user_id = 'frodo';
```

To remove all elements having a particular value, use the UPDATE command, the subtraction operator (-), and the list value in square brackets:

```
UPDATE users
  SET top_places = top_places - ['riddermark'] WHERE user_id = 'frodo';
```

To update data in a collection column of a user-defined type, enclose components of the type in parentheses within the curly brackets, as shown in "Using a user-defined type."

## USE

Connect the client session to a keyspace.

## Synopsis

```
USE keyspace_name
```

## Synopsis Legend

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange ( and ) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

## Description

A USE statement identifies the keyspace that contains the tables to query for the current client session. All subsequent operations on tables and indexes are in the context of the named keyspace, unless otherwise specified or until the client connection is terminated or another USE statement is issued.

To use a case-sensitive keyspace, enclose the keyspace name in double quotation marks.

## Example

```
USE PortfolioDemo;
```

Continuing with the example of checking created keyspaces:

```
USE "Excalibur";
```