CQL for Cassandra 2.0

Documentation

March 13, 2014

Contents

Introduction

This document includes descriptions of features in Cassandra 2.0.6 that are based on the CQL specification 3.1.1. New features are:

- Batching of conditional updates
- Static columns
- Increased control for slicing over clustering columns

Cassandra 2.0 included the following new features and changes:

- 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.
- Atomic BATCH guarantees for large sets of prepared statements.
- One-shot binding of optional variables or prepared statements and variables for server-side request parsing and execution using a BATCH message containing a list of query strings and bind variables.
- 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 a compound primary key component.

Drivers that utilize CQL are available on github:

- DataStax Java Driver 2.0.0
- DataStax Java Driver 1.0.x
- DataStax C# Driver
- DataStax Python Driver

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

Some changes have been made to the cqlsh commands:

- The ASSUME command has been removed.

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

Several CQL table attributes have been added:

- default_time_to_live
- memtable_flush_period_in_ms
- populate_io_cache_on_flush
- speculative_retry

CQL data modeling

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.

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. 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.

After inserting the example data into playlists, the output of selecting all the data looks like this:

```
SELECT * FROM playlists;
```

id	song_order	album	artist	song_id	title
62c36092...	1	Tres Hombres	ZZ Top	a3e64f8f...	La Grang
62c36092...	2	We Must Obey	Fu Manchu	8a172618...	Moving in Stere
62c36092...	3	Roll Away	Back Door Slam	2b09185b...	Outside Woman Blue

The next example illustrates how you can create a query that uses the artist as a filter. First, add a little more data to the playlist table to make things interesting for the collections examples later:

```
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');
```

With the schema as given so far, a query that includes the artist filter would require a sequential scan across the entire playlists dataset. Cassandra will reject such a query. If you first create an index on artist, Cassandra can efficiently pull out the records in question.

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

Now, you can query the playlists for songs by Fu Manchu, for example:

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

The output looks something like this:

id	song_order	album	artist	song_id	title
62c36092...	2	We Must Obey	Fu Manchu	8a172618...	Moving in Ster
62c36092...	4	No One Rides for Free	Fu Manchu	7db1a490...	Ojo Ro

Compound keys and clustering

A compound primary key includes the partition key, which determines on which node data is stored, and one or more additional columns that determine clustering. Cassandra uses the first column name in the primary key definition as the partition key. For example, in the playlists table, id is the partition key. The remaining column, or columns that are not partition keys in the primary key definition are the clustering columns. In the case of the playlists table, the song_order is the clustering column. 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.

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...	
62c36092...	3	Roll Away	Back Door Slam	2b09185b...	Outside W
62c36092...	2	We Must Obey	Fu Manchu	8a172618...	Moving
62c36092...	1	Tres Hombres	ZZ Top	a3e64f8f...	

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 introduces 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.

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 songs table already exists from the earlier example, just alter that table to add a collection set, tags:

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

Updating a collection

Update the songs table to insert the tags data:

```
UPDATE songs SET tags = tags + {'2007'}
  WHERE id = 8a172618-b121-4136-bb10-f665cfc469eb;
UPDATE songs SET tags = tags + {'covers'}
  WHERE id = 8a172618-b121-4136-bb10-f665cfc469eb;
UPDATE songs SET tags = tags + {'1973'}
  WHERE id = a3e64f8f-bd44-4f28-b8d9-6938726e34d4;
UPDATE songs SET tags = tags + {'blues'}
  WHERE id = a3e64f8f-bd44-4f28-b8d9-6938726e34d4;
UPDATE songs SET tags = tags + {'rock'}
  WHERE id = 7db1a490-5878-11e2-bcfd-0800200c9a66;
```

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

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

Each element of a map, 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 songs
  SET tags = tags + {'rock'}
  WHERE id = 7db1a490-5878-11e2-bcfd-0800200c9a66;
```

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

```
UPDATE songs
  SET reviews = reviews + [ 'hot dance music' ]
  WHERE id = 7db1a490-5878-11e2-bcfd-0800200c9a66;
```

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

```
INSERT INTO songs (id, venue)
  VALUES (7db1a490-5878-11e2-bcfd-0800200c9a66,
  { '2013-9-22 12:01'  : 'The Fillmore',
  '2013-10-1 18:00' : 'The Apple Barrel'});
```

Inserting data into the map replaces the entire map.

Querying a collection

To query a collection, include the name of the collection column in the select expression. For example, selecting the tags set returns the set of tags, sorted alphabetically in this case because the tags set is of the text data type:

```
SELECT id, tags FROM songs;
```

```
 id                                   | tags
--------------------------------------+-----------------
 7db1a490-5878-11e2-bcfd-0800200c9a66 |        {rock}
 a3e64f8f-bd44-4f28-b8d9-6938726e34d4 |  {1973, blues}
 8a172618-b121-4136-bb10-f665cfc469eb | {2007, covers}
```

```
SELECT id, venue FROM songs;
```

```
 id          | venue
-------------+------------------------------------------------------------------------------
 7db1a490... | {2013-10-01 18:00:00-0700: The Apple Barrel, 2013-09-22 12:01:00-0700: The Fil
 a3e64f8f... |
 8a172618... |
```

The collection types are described in more detail in Using collections: set, list, and map.

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.

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 . . . more
- In tables that use a counter column
- On a frequently updated or deleted column . . . more
- To look for a row in a large partition unless narrowly queried . . . more

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. 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 Stere
62c36092...	4	No One Rides for Free	Fu Manchu	7db1a490...	Ojo Roj

Using multiple indexes

For example purposes, let's say you can create multiple indexes, for example 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 earlier:

```
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 occurrances 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 utility 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. A table contains a timestamp representing the date/time that a write occurred to a columns. You use WRITETIME in the select expression to get this timestamp. For example, to get the date/times that a write occurred to the body column:

```
SELECT WRITETIME (title )
  FROM songs
  WHERE  id = 8a172618-b121-4136-bb10-f665cfc469eb;

 writetime (title )
------------------
 1353890782373000
```

The output in microseconds shows the write time of the data in the title column of the songs table.

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 CQL 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

You can use CQL on the command line of a Cassandra node, from DataStax DevCenter, or programmatically using a number of APIs and drivers. 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.
- Use a DataStax driver for programmatic access.
    - DataStax Java Driver 2.0.0

      Compatible with Cassandra 2.0/CQL 3.1 specification and Cassandra 1.2/CQL 3.0 specification. Supports a smooth upgrade to Cassandra 2.0.
    - DataStax Java Driver 1.0.x

      Based on the native/binary protocol version 1, this driver accesses the CQL 3 version based on the CQL 3.0 specification, the default CQL mode in Cassandra 1.2. Use of the driver is limited for CQL applications based on the CQL 3.1 specification, the default CQL mode in Cassandra 2.0. This driver includes support for upgrading nodes in mixed-version clusters when the next generation driver is available.
    - DataStax C# Driver 1.0.x
    - DataStax Python Driver 1.0.x
- Use the set_cql_version Thrift method for programmatic access.

This document presents examples using cqlsh.

Starting cqlsh

Starting cqlsh on Linux

About this task

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

Procedure

1. Make the Cassandra bin directory your working directory.
2. Start cqlsh on the Mac OSX, for example.

   ```
   ./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.

   ```
   ./cqlsh 1.2.3.4 9160
   ```

Starting cqlsh on Windows

About this task

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 9160
   ```

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, you specify a keyspace replication strategy, SimpleStrategy or NetworkTopologyStrategy. Using SimpleStrategy is fine for evaluating Cassandra. For production use or for use with mixed workloads, use NetworkTopologyStrategy.

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. 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-3a90-4aa3-
a2f9-3e162baa4990  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.

Example of creating a keyspace

About this task

To query Cassandra, you first create and use a keyspace. In this example, dc1 is the name of a data center that has been registered in the properties file of the snitch. To use NetworkTopologyStrategy for a cluster in a single data center, there is no need to register the data center name in the properties file. Simply use the default data center name, for example datacenter1.

Procedure

1. Create a keyspace.

   ```
   cqlsh> CREATE KEYSPACE demodb
            WITH REPLICATION = { 'class' : 'NetworkTopologyStrategy', 'dc1' :
    3 };
   ```
2. Use the keyspace.

   ```
   USE demodb;
   ```

Updating the replication factor

About this task

Increasing the replication factor increases the total number of copies of keyspace data stored in a Cassandra cluster. It is particularly important to increase the replication factor of the system_auth keyspace if you are using security features because if you use the default, 1, and the node with the lone replica goes down, you will not be able to log into the cluster because the system_auth keyspace was not replicated.

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, to change a keyspace from using SimpleStrategy to NetworkTopologyStrategy:

   ```
   ALTER KEYSPACE mykeyspace WITH REPLICATION =
     { 'class' : 'NetworkTopologyStrategy', 'dc1' : 3 };
   ```
2. On each affected node, run nodetool repair.
3. Wait until repair completes on a node, then move to the next node.

Creating a table

About this task

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

About this task

Use a compound primary key when you want 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

About this task

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');
```

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 1: Columns in System Tables

Table name	Column name	Comment
schema_keyspaces	keyspace_name, durable_writes, strategy_class, strategy_options	None

Table name	Column name	Comment
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 column families.

Keyspace, table, and column information

About this task

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

About this task

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 | rack  | release_version | ring_id          |
 rpc_address | schema_version    | tokens
-----------+-------------+-------+-----------------+-----------------
+------------+----------------+-------. . .
 127.0.0.3 | datacenter1 | rack1 | 1.2.0-beta2     | 53d171bc-ff. . .|
 127.0.0.3   | 59adb24e-f3 . . .|  {3074. . .
 127.0.0.2 | datacenter1 | rack1 | 1.2.0-beta2     | 3d19cd8f-c9. . .|
 127.0.0.2   | 59adb24e-f3 . . .| {-3074. . .}

(2 rows)
```

Retrieving and sorting results

About this task

To retrieve results, use the SELECT command.

```
SELECT * FROM users WHERE first_name = 'jane' and last_name='smith';
```

Similar to a SQL query, use the WHERE clause and then the ORDER BY clause to retrieve and sort results.

Procedure

1. Retrieve and sort results in descending order.

```
cqlsh:demodb> SELECT * FROM emp WHERE empID IN (130,104) ORDER BY deptID
 DESC;

 empid | deptid | first_name | last_name
-------+--------+------------+-----------
   104 |     15 |       jane |     smith
   130 |      5 |     sughit |     singh

(2 rows)
```

2. Retrieve and sort results in ascending order.

```
cqlsh:demodb> SELECT * FROM emp where empID IN (130,104) ORDER BY deptID
 ASC;

 empid | deptid | first_name | last_name

-------+--------+------------+-----------
   130 |      5 |     sughit |     singh
   104 |     15 |       jane |     smith
```

The music service example shows how to retrieve and sort results using compound primary keys.

Slicing over partition rows

In Cassandra 2.0.6 and later, you can use a new syntax for slicing over rows of a partition if the table has more than one clustering column. Using a conditional operator, you can compare groups of clustering columns 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 does work:

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

Using the keyspace qualifier

About this task

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.songs.

```
INSERT INTO Music.songs (id, title, artist, album)
  VALUES (a3e64f8f-bd44-4f28-b8d9-6938726e34d4, 'La Grange', 'ZZ Top', 'Tres
 Hombres');
```

Adding columns to a table

About this task

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.

About expiring columns

Data in a column can have an optional expiration date called TTL (time to live). When a column is inserted, the client request can specify an optional TTL value, defined in seconds, for the data in the column. TTL columns are marked with a tombstone and after the requested amount of time has expired. Data is removed during the compaction process after gc_grace_seconds.

Use CQL to set the TTL for data in a column.

If you want to change the TTL of data, re-insert the column with a new TTL and values. In Cassandra, the insertion of a column and values is actually an insertion or update operation, depending on whether or not a previous version of the column exists. To update the TTL for data in a column, you have to read the column and then re-insert it with the new TTL value.

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.

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

Determining time-to-live for a column

About this task

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 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.

Expiring columns

About this task

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';
   ```

Dropping a table or keyspace

About this task

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

About this task

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

About this task

Using the WRITETIME function in a SELECT statement returns the date/time in microseconds that the column was written to the database. 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 WRITELINE 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

(1 rows)
```

Altering the data type of a column

About this task

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.
- 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.

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

Using the set type

About this task

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

   (1 rows)
   ```

Using the list type

About this task

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.

   ```
   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.

   ```
   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';
   ```

Using the map type

About this task

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',
     '2012-10-2 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['2012-10-2 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 (todo)
  VALUES ( { '2013-9-22 12:01'  : 'birthday wishes to Bilbo',
  '2013-10-1 18:00' : 'Check into Inn of Prancing Pony' });
```

   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['2012-9-24'] FROM users 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

About this task

You can use cqlsh to create an index on column values. 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

About this task

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’
```

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 bills SET balance=-200 WHERE user='user1' IF balance=-208;
      UPDATE bills SET paid=true
                 WHERE user='user1' AND expense_id=1
```

```
                  IF paid=false;
APPLY BATCH;

SELECT * FROM bills;

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

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).

Using a counter

About this task

A counter is a special kind of column used to store a number that incrementally counts the occurrences of a particular event or process. For example, you might use a counter column to count the number of times a page is viewed.

Counter column tables must use Counter data type. Counters may only be stored in dedicated tables. You cannot index a counter column.

You load data into a counter column using the UPDATE command instead of INSERT. To increase or decrease the value of the counter, you also use UPDATE.

Procedure

1. Create a keyspace on Linux for use in a single data center. Use the default data center name from the output of nodetool status, 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

   (1 rows)
   ```
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 tutorial

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. The Cassandra 2.0 documentation contains detailed discussions about how Cassandra distributes data and handles consistency. 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

About this task

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.

   Use this library to create, launch, and remove a Apache Cassandra cluster on a single computer. Use the instructions in the ccm README and in this document to set up the cluster.
2. Download source code from the Apache Cassandra web site or use an existing Cassandra installation.

   Subsequent steps assume you are using a newly downloaded Cassandra 2.0.2 on a Mac OSX.

3. In the `apache-cassandra-2.0.2-src` directory, use the `ant` command to build the source code.

   ```
   $ cd <install_location>
   $ ant
   ```
4. In the `conf` directory, inhibit the use of vnodes by opening the `cassandra.yaml` and setting num_nodes to 0.
5. Enter the following commands on the command line to set up alias on the local IP.

   ```
   $ 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
   ```
6. Start the ccm cluster named trace_consistency pointing to the Apache Cassandra 2.0.2 installation directory:

   ```
   $ ccm create trace_consistency --cassandra-dir=/Users/krishahn/builds/2.0.2/
   apache-cassandra-2.0.2-src
   ```
7. Use the following commands to populate and check the cluster:

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

   ```
   $ ccm node1 ring
   ```

   The output shows the status of all five nodes.

Trace reads at different consistency levels

About this task

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 replicas 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 ONE.
   ```
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: 230a0d10-5576-11e3-add5-a180b5ee3938

 activity                                                                   |
 timestamp    | source    | source_elapsed
---------------------------------------------------------------------------
+--------------+-----------+----------------
                                                         execute_cql3_query |
 18:06:03,361 | 127.0.0.1 |              0
                                            Message received from /127.0.0.1 |
 18:06:03,362 | 127.0.0.4 |             51
                Parsing SELECT * FROM demo_table WHERE id = 0 LIMIT 10000; |
 18:06:03,362 | 127.0.0.1 |             63
                                                         Preparing statement |
 18:06:03,362 | 127.0.0.1 |            142
                                               Sending message to /127.0.0.4 |
 18:06:03,362 | 127.0.0.1 |            518
                            Executing single-partition query on demo_table |
 18:06:03,363 | 127.0.0.4 |            532
                                                Acquiring sstable references |
 18:06:03,363 | 127.0.0.4 |            556
                                                 Merging memtable tombstones |
 18:06:03,363 | 127.0.0.4 |            597
 Skipped 0/0 non-slice-intersecting sstables, included 0 due to tombstones |
 18:06:03,363 | 127.0.0.4 |            676
                                Merging data from memtables and 0 sstables |
 18:06:03,363 | 127.0.0.4 |            714
                                          Read 1 live and 0 tombstoned cells |
 18:06:03,363 | 127.0.0.4 |            796
                                            Enqueueing response to /127.0.0.1 |
 18:06:03,363 | 127.0.0.4 |           1044
                                               Sending message to /127.0.0.1 |
 18:06:03,363 | 127.0.0.4 |           1154
                                            Message received from /127.0.0.4 |
 18:06:03,364 | 127.0.0.1 |           2284
                                         Processing response from /127.0.0.4 |
 18:06:03,364 | 127.0.0.1 |           2401
                                                            Request complete |
 18:06:03,363 | 127.0.0.1 |           2585
```

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;

. . .
Tracing session: 298af000-5576-11e3-add5-a180b5ee3938

 activity                                                                   |
 timestamp    | source    | source_elapsed
---------------------------------------------------------------------------
+--------------+-----------+----------------
                                                         execute_cql3_query |
 18:06:14,273 | 127.0.0.1 |              0
```

```
                  Parsing SELECT * FROM demo_table WHERE id = 0 LIMIT 10000; |
 18:06:14,273 | 127.0.0.1 |             64
                                                        Preparing statement |
 18:06:14,273 | 127.0.0.1 |            144
                                              Sending message to /127.0.0.4 |
 18:06:14,273 | 127.0.0.1 |            492
                                              Sending message to /127.0.0.3 |
 18:06:14,273 | 127.0.0.1 |            528
                                           Message received from /127.0.0.1 |
 18:06:14,274 | 127.0.0.4 |             46
                                           Message received from /127.0.0.1 |
 18:06:14,274 | 127.0.0.3 |             63
                                Executing single-partition query on demo_table |
 18:06:14,274 | 127.0.0.4 |            517
                                Executing single-partition query on demo_table |
 18:06:14,274 | 127.0.0.3 |            614
                                               Acquiring sstable references |
 18:06:14,274 | 127.0.0.4 |            542
                                               Acquiring sstable references |
 18:06:14,274 | 127.0.0.3 |            640
                                                Merging memtable tombstones |
 18:06:14,274 | 127.0.0.4 |            581
                                                Merging memtable tombstones |
 18:06:14,274 | 127.0.0.3 |            719
 Skipped 0/0 non-slice-intersecting sstables, included 0 due to tombstones |
 18:06:14,274 | 127.0.0.4 |            702
 Skipped 0/0 non-slice-intersecting sstables, included 0 due to tombstones |
 18:06:14,274 | 127.0.0.3 |            822
                                    Merging data from memtables and 0 sstables |
 18:06:14,274 | 127.0.0.4 |            750
                                    Merging data from memtables and 0 sstables |
 18:06:14,274 | 127.0.0.3 |            862
                                            Read 1 live and 0 tombstoned cells |
 18:06:14,274 | 127.0.0.4 |            840
                                              Enqueuing response to /127.0.0.1 |
 18:06:14,275 | 127.0.0.4 |           1091
                                            Read 1 live and 0 tombstoned cells |
 18:06:14,275 | 127.0.0.3 |           1001
                                           Message received from /127.0.0.4 |
 18:06:14,275 | 127.0.0.1 |           2460
                                              Sending message to /127.0.0.1 |
 18:06:14,275 | 127.0.0.4 |           1207
                                              Enqueuing response to /127.0.0.1 |
 18:06:14,275 | 127.0.0.3 |           1410
                                        Processing response from /127.0.0.4 |
 18:06:14,275 | 127.0.0.1 |           2628
                                              Sending message to /127.0.0.1 |
 18:06:14,275 | 127.0.0.3 |           1643
                                           Message received from /127.0.0.3 |
 18:06:14,275 | 127.0.0.1 |           2695
                                        Processing response from /127.0.0.3 |
 18:06:14,276 | 127.0.0.1 |           2774
                                                          Request complete |
 18:06:14,275 | 127.0.0.1 |           2998
```

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;

. . .
Tracing session: 32733600-5576-11e3-add5-a180b5ee3938

 activity                                                                   |
 timestamp    | source    | source_elapsed
```

```
---------------------------------------------------------------------------
+--------------+-----------+----------------
                                                        execute_cql3_query |
 18:06:29,217 | 127.0.0.1 |              0
                                         Message received from /127.0.0.1 |
 18:06:29,217 | 127.0.0.4 |             54
                                         Message received from /127.0.0.1 |
 18:06:29,217 | 127.0.0.3 |             55
                                         Message received from /127.0.0.1 |
 18:06:29,217 | 127.0.0.5 |             54
               Parsing SELECT * FROM demo_table WHERE id = 0 LIMIT 10000; |
 18:06:29,217 | 127.0.0.1 |             62
                                                       Preparing statement |
 18:06:29,217 | 127.0.0.1 |            139
                                            Sending message to /127.0.0.4 |
 18:06:29,217 | 127.0.0.1 |            475
                                            Sending message to /127.0.0.5 |
 18:06:29,217 | 127.0.0.1 |            499
                                            Sending message to /127.0.0.3 |
 18:06:29,217 | 127.0.0.1 |            516
                           Executing single-partition query on demo_table |
 18:06:29,218 | 127.0.0.4 |            663
                                             Acquiring sstable references |
 18:06:29,218 | 127.0.0.4 |            691
                                              Merging memtable tombstones |
 18:06:29,218 | 127.0.0.4 |            745
 Skipped 0/0 non-slice-intersecting sstables, included 0 due to tombstones |
 18:06:29,218 | 127.0.0.4 |            833
                               Merging data from memtables and 0 sstables |
 18:06:29,218 | 127.0.0.4 |            886
                                        Read 1 live and 0 tombstoned cells |
 18:06:29,218 | 127.0.0.4 |            974
                                         Enqueuing response to /127.0.0.1 |
 18:06:29,219 | 127.0.0.4 |           1238
                           Executing single-partition query on demo_table |
 18:06:29,219 | 127.0.0.3 |           1812
                           Executing single-partition query on demo_table |
 18:06:29,219 | 127.0.0.5 |           1671
                                         Message received from /127.0.0.4 |
 18:06:29,219 | 127.0.0.1 |           2900
                                            Sending message to /127.0.0.1 |
 18:06:29,219 | 127.0.0.4 |           1433
                                             Acquiring sstable references |
 18:06:29,219 | 127.0.0.3 |           1847
                                             Acquiring sstable references |
 18:06:29,219 | 127.0.0.5 |           1707
                                              Merging memtable tombstones |
 18:06:29,219 | 127.0.0.3 |           1927
                                              Merging memtable tombstones |
 18:06:29,219 | 127.0.0.5 |           1790
 Skipped 0/0 non-slice-intersecting sstables, included 0 due to tombstones |
 18:06:29,219 | 127.0.0.3 |           2040
 Skipped 0/0 non-slice-intersecting sstables, included 0 due to tombstones |
 18:06:29,219 | 127.0.0.5 |           1900
                               Merging data from memtables and 0 sstables |
 18:06:29,219 | 127.0.0.5 |           1944
                                        Read 1 live and 0 tombstoned cells |
 18:06:29,219 | 127.0.0.5 |           2107
                               Merging data from memtables and 0 sstables |
 18:06:29,220 | 127.0.0.3 |           2080
                                         Enqueuing response to /127.0.0.1 |
 18:06:29,220 | 127.0.0.5 |           2534
                                      Processing response from /127.0.0.4 |
 18:06:29,220 | 127.0.0.1 |           3451
```

```
                                              Read 1 live and 0 tombstoned cells |
       18:06:29,220 | 127.0.0.3 |           2231
                                                   Sending message to /127.0.0.1 |
       18:06:29,220 | 127.0.0.5 |           2826
                                                Enqueuing response to /127.0.0.1 |
       18:06:29,220 | 127.0.0.3 |           2691
                                                   Sending message to /127.0.0.1 |
       18:06:29,221 | 127.0.0.3 |           3176
                                                Message received from /127.0.0.5 |
       18:06:29,221 | 127.0.0.1 |           4027
                                             Processing response from /127.0.0.5 |
       18:06:29,221 | 127.0.0.1 |           4534
                                                Message received from /127.0.0.3 |
       18:06:29,221 | 127.0.0.1 |           4602
                                             Processing response from /127.0.0.3 |
       18:06:29,221 | 127.0.0.1 |           4821
                                                                Request complete |
       18:06:29,222 | 127.0.0.1 |           5219
```

How consistency affects performance

About this task

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 utility, 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 3.1.1.

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 2: 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 "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.

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.

Valid literals

Valid literal consist of these kinds of values:

- blob

  hexadecimal
- boolean

  true or false, case-insensitive, and in CQL, enclosure in single quotation marks is not required prior to Cassandra 1.2.2. In 1.2.2 and later, using quotation marks is not allowed.
- 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+])?`
- identifier

  A letter followed by any sequence of letters, digits, or the underscore. Names of tables, columns, and other objects are identifiers. Enclose them 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

(1 rows)
```

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 3: 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
BATCH	yes
BEGIN	yes
BIGINT	no
BLOB	no
BOOLEAN	no
BY	yes
CLUSTERING	no
COLUMNFAMILY	yes
COMPACT	no
CONSISTENCY	no
COUNT	no

Keyword	Reserved
COUNTER	no
CREATE	yes
CUSTOM	no
DECIMAL	no
DELETE	yes
DESC	yes
DISTINCT (CQL 3.1.1)	no
DOUBLE	no
DROP	yes
EACH_QUORUM	yes
FILTERING	no
FLOAT	no
FROM	yes
GRANT	yes
IN	yes
INDEX	yes
INET	yes
INFINITY	yes
INSERT	yes
INT	no
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

Keyword	Reserved
OF	yes
ON	yes
ONE	
ORDER	yes
PASSWORD	yes
PERMISSION	no
PERMISSIONS	no
PRIMARY	yes
QUORUM	yes
RENAME	yes
REVOKE	yes
SCHEMA	yes
SELECT	yes
SET	yes
STORAGE	no
SUPERUSER	no
TABLE	yes
TEXT	no
TIMESTAMP	no
TIMEUUID	no
THREE	yes
TO	yes
TOKEN	yes
TRUNCATE	yes
TTL	no
TWO	yes
TYPE	no
UNLOGGED	yes
UPDATE	yes
USE	yes
USER	no
USERS	no
USING	yes
UUID	no

Keyword	Reserved
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 4: 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)
decimal	integers, floats	Variable-precision decimal
double	integers	64-bit IEEE-754 floating point
float	integers, floats	32-bit IEEE-754 floating point
inet	strings	IP address string in IPv4 or IPv6 format*
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
uuid	uuids	A UUID in standard UUID format
timeuuid	uuids	Type 1 UUID only

CQL Type	Constants	Description
varchar	strings	UTF-8 encoded string
varint	integers	Arbitrary-precision integer

*Used by python-cql driver and CQL native protocols.

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.

Blob

Cassandra 1.2.3 still supports blobs as string constants for input (to allow smoother transition to blob constant). Blobs as strings are now deprecated and will not be supported in the near future. If you were using strings as blobs, update your client code to switch to blob constants. A blob constant is an hexadecimal number defined by 0[xX](hex)+ where hex is an hexadecimal character, such as [0-9a-fA-F]. For example, 0xcafe.

Blob conversion functions

A number of functions convert the native types into binary data (blob). For every <native-type> 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. For example, bigintAsBlob(3) is 0x0000000000000003 and blobAsBigint(0x0000000000000003) is 3.

Collection types

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
```

Currently, you cannot create an index on a column of type map, set, or list.

UUID and timeuuid

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 Type 1 UUID. A type 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.

Timeuuid functions

Functions for use with the timeuuid type are:

- 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()

  Generates a new unique timeuuid when the statement is executed. 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 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.

TimestampType

Values for the TimestampType 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 TimestampType can be entered as an integer for CQL input, or as a string literals 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.

TimestampType 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.

Counter type

To use counter types, see the DataStax blog about counters and the "Using a counter" section. Do not assign this type to a column that serves as the primary key. Also, do not use the counter type in a table that contains anything other than counter types (and 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.

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 5: CQL properties

CQL property	Description	Default
bloom_filter_fp_chance	Desired false-positive probability for SSTable Bloom filters. . . . more	0.01 for `SizeTieredCompactionStrategy`, 0.1 for `LeveledCompactionStrategy`
caching	Optimizes the use of cache memory without manual tuning. Set caching to one of the following values: • all • keys_only • rows_only • none Cassandra weights the cached data by size and access frequency. Use this parameter to specify a key or row cache instead of a table cache, as in earlier versions.	keys_only
comment	A human readable comment describing the table. . . . more	N/A
compaction	Sets the compaction strategy for the table. . . . more	`SizeTieredCompactionStrategy`
compression	The compression algorithm to use. Valid values are `LZ4Compressor`, `SnappyCompressor`, and `DeflateCompressor`. . . . more	`LZ4Compressor`
dclocal_read_repair_chance	Specifies the probability of read repairs being invoked over all replicas in the current data center.	0.0
default_time_to_live	The default expiration time in seconds for a table. Used in MapReduce/Hive scenarios when you have no control of TTL.	0
gc_grace_seconds	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.	864000 [10 days]
index_interval	Configure sample frequency of the partition summary by changing the index_interval property. After changing the value of index_interval,	128

CQL property	Description	Default
	SSTables are written to disk with new value. . . . more	
memtable_flush_period_in_ms	Forces flushing of the memtable after the specified time in milliseconds elapses.	0
populate_io_cache_on_flush	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. See also the global option, compaction_preheat_key_cache.	false
read_repair_chance	Specifies the probability with which read repairs should be invoked on non-quorum reads. The value must be between 0 and 1.	0.1
replicate_on_write	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.	true
speculative_retry	Overrides normal read timeout when read_repair_chance is not 1.0, sending another request to read. Choices are: • 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. more . . .	99percentile Cassandra 2.0.2 and later

Bloom filter

When data is requested, the Bloom filter checks if the row exists before doing disk I/O.

- Valid range: 0 to 1.0
- Valid values

  0 Enables the unmodified (effectively the largest possible) Bloom filter

  1.0 disable the Bloom Filter
- Recommended setting: 0.1.

A higher value yields diminishing returns.

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 available compaction strategies are:

- `SizeTieredCompactionStrategy`: The default compaction strategy and the only compaction strategy available in releases earlier than Cassandra 1.0. This strategy triggers a minor compaction whenever there are a number of similar sized SSTables on disk (as configured by the subproperty, min_threshold. Using this strategy causes bursts in I/O activity while a compaction is in process, followed by longer and longer lulls in compaction activity as SSTable files grow larger in size. These I/O bursts can negatively effect read-heavy workloads, but typically do not impact write performance. Watching disk capacity is also important when using this strategy, as compactions can temporarily double the size of SSTables for a table while a compaction is in progress.
- `LeveledCompactionStrategy`: The leveled compaction strategy creates SSTables of a fixed, relatively small size (5 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 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. Articles When to Use Leveled Compaction and Leveled Compaction in Apache Cassandra provide more details.

Also use the compaction subproperties.

compression

Use an empty string ('') to disable 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". Also use the compression subproperties.

index_interval

The index_interval property controls the sampling of entries from the primary row index. 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. 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.

speculative retry

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

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

Or:

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

Subproperties of compaction

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

Table 6: CQL Compaction Subproperties

Compaction Subproperties	Description	Default	Supported Strategy
bucket_high	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].	1.5	SizeTieredCompactionStrategy
bucket_low	Same as above.	0.5	SizeTieredCompactionStrategy
cold_reads_to_omit	The maximum percentage of reads/sec that ignored SSTables may account for. The recommended range of values is 0.0 and 1.0. . . .. . . more	0.0 (disabled)	SizeTieredCompactionStrategy
max_threshold	In `SizeTieredCompactionStrategy`, sets the maximum number of SSTables to allow in a minor compaction. In `LeveledCompactionStrategy`, it applies to L0 when L0 gets behind, that is, when L0 accumulates more than `MAX_COMPACTING_L0` SSTables.	32	SizeTieredCompactionStrategy
min_threshold	In SizeTieredCompactionStrategy sets the minimum number of SSTables to trigger a minor compaction.	4	SizeTieredCompactionStrategy

Compaction Subproperties	Description	Default	Supported Strategy
min_sstable_size	The `SizeTieredCompactionStrategy` groups 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.	50MB	SizeTieredCompactionStrategy
sstable_size_in_mb	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.	160MB	LeveledCompactionStrategy
tombstone_compaction_interval	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.	1 day	all
tombstone_threshold	A ratio of garbage-collectable tombstones to all contained columns, which if exceeded by the SSTable triggers compaction (with no other SSTables) for the	0.2	all

Compaction Subproperties	Description	Default	Supported Strategy
	purpose of purging the tombstones.		

To disable background compactions, use nodetool disableautocompaction/enableautocompaction instead of setting min/max compaction thresholds to 0.

cold_reads_to_omit

In Cassandra 2.0.3 and later, you can enable the cold_reads_to_omit property to tune performace per table. 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.

Subproperties of compression

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

Table 7: CQL Compression Subproperties

Compression Subproperties	Description	Default
sstable_compression	The compression algorithm to use. Valid values are `LZ4Compressor` available in Cassandra 1.2.2 and later), `SnappyCompressor`, and `DeflateCompressor`. . . . more	`SnappyCompressor`
chunk_length_kb	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.	64KB
crc_check_chance	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	1.0

Compression Subproperties	Description	Default
	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 read.	

sstable_compression

Use an empty string ('') to disable 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
- Timeuuid functions
- Token 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 depend 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.

Prepared statements

CQL supports prepared statements. Using a prepared statement, Cassandra parses a query only once, but executes it multiple times with different concrete values.

In a statement, each time a column value is expected in the data manipulation and query statements, a bind variable marker can be used instead. A statement with bind variables must then be prepared. During execution of the prepared statement, concrete values for bind variables are provided in the order the bind variables are defined in the query string.

Use bind variables, a ? in these examples, to represent these values:

- INSERT statement values

  ```
  INSERT INTO playlists (id, song_order, song_id, title, artist, album)
      VALUES (62c36092-82a1-3a00-93d1-46196ee77204, 1,
      a3e64f8f-bd44-4f28-b8d9-6938726e34d4, '?', '?', '?');
  ```
- Values on the right side of the = operator in the SET and clauses of the UPDATE statement

```
UPDATE users
  SET name = '?',
  email = '?'
  WHERE user_uuid = ?;
```

- Values on the right side of the IN or = operators in the WHERE clause of the DELETE statement

```
DELETE email, phone
    FROM users
    USING TIMESTAMP 1318452291034
    WHERE user_name = '?';
```

- Values for the TIMESTAMP and TTL properties of the UPDATE statement

```
UPDATE myTable USING TTL ? SET v = 2 WHERE k = 'foo';
```

- Value of the LIMIT property in a SELECT statement

```
SELECT * FROM myTable LIMIT ?;
```

The exact procedure to prepare a statement and execute a prepared statement depends on the CQL driver used and is beyond the scope of this document. The Java Driver documentation includes an example.

Cassandra 2.0 and later support a query that uses bind variables even though the statement is not prepared. In Cassandra 2.0.1 (CQL 3.1.1) and later, you can use named bind variables (:name instead of ?).

Prepared statement batch message

Description

Using a CQL native protocol version 2 connection, you can execute a BATCH message containing a list of prepared DML statements. Construct a message having a body in this format:

```
<type><n><query_1>...<query_n><consistency>
```

where:

- *type* is of data type byte and indicates the type of batch to use: If type is 0, the batch is logged. This is equivalent to a normal CQL batch statement. If type is 1, the batch is unlogged. If type is 2, the batch is a counter batch (and non-counter statements are rejected).
- *n* is a [short] indicating the number of following queries.
- *query_1...query_n* are the queries to execute.

A query_i must be of the form:

```
<kind><string_or_id><n><value_1>...<value_n>
```

where:

- *kind* is of type byte indicating whether the following query is a prepared one or not. The value is either 0 or 1.
- *string_or_id* depends on the value of kind:

  If kind is 0, string_or_id is a query string of type long string and might contain bind markers. Otherwise, if kind is 1, string_or_id is of type short bytes representing a prepared query ID.
- *n* is of type short indicating the number of (possibly 0) of following values.
- *value_1...value_n* are the bytes to use for bound variables.

Ensure that there is no newline at end of file containing the prepared query.

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.

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

The cqlsh utility uses the Thrift transport.

Requirements

By default, Cassandra 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.

Options

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

--debug
: Show additional debugging information.

-f file_name, --file=file_name
: Execute commands from FILE, 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.

cqlshrc options

You can create a `cqlshrc` file that resides in the hidden `.cassandra` directory in your home directory. You configure the `cqlshrc` file by setting these options in the [authenication], [ui], or [ssl] sections of the file.

[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 of the timestamp type. For example, a yyyy-mm-dd HH:mm:ssZ formatting produces this timestamp: 2014-01-01 12:00:00GMT. Default = '%Y-%m-%d %H:%M:%S%z'.

[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.

[ssl] options are covered in the Cassandra documentation.

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 9160 -u jdoe -p mypassword
Connected to trace_consistency at 1.2.3.4:9160.
[cqlsh 4.1.1 | Cassandra 2.0.6 | CQL spec 3.1.1 | Thrift protocol 19.39.0]
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.

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.

Creating and using a `cqlshrc` file

When present, the `cqlshrc` 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. 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.

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

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

Description

Providing an argument to the CONSISTENCY command overrides the default consistency level and configures the consistency level for future requests.

Providing no argument shows the current consistency level.

Example

```
CONSISTENCY
```

Assuming the consistency level is ONE, 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 1.1.3 and higher.

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

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

Description

Using the COPY options in a WITH clause, you can change the CSV format. This table describes these options:

Table 8: COPY options

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.
ENCODING	UTF8	Set the COPY TO command to output unicode strings.
NULL	an empty string	Represents the absence of a value.

The ENCODING option cannot be used in the COPY FROM 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.

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 is intended for importing small datasets (a few million rows or less) into Cassandra. For importing larger datasets, use Cassandra bulk loader or the sstable2json/json2sstable2 utility

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

(1 rows)
```

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. In version 1.1.6 and later, 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.

Examples

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' :
 3 };

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, '.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.

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

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
     MiG-23 Flogger | Russian-made | 1967 | 2.35

(4 rows)
```

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 )
```

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 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 CLUSTER

  Output is the information about the connected Cassandra cluster, such as the cluster name, and the partitioner and snitch in use. When you are connected to a non-system keyspace, this command also shows endpoint-range ownership information for the Cassandra ring.
- DESCRIBE FULL SCHEMA

  Output is the list of CQL commands that could be used to recreate the entire user-created schema and system schema.
- DESCRIBE SCHEMA

  Output is a list of CQL commands that could be used to recreate the entire user-created schema. Works as though DESCRIBE KEYSPACE <k> was invoked for each keyspace k.
- DESCRIBE KEYSPACES

  Output is a list of all keyspace names.
- DESCRIBE KEYSPACE *keyspace_name*

  Output is a list of CQL commands that could be used to recreate the given keyspace, and the tables in it. In some cases, as the CQL interface matures, there will be some metadata about a keyspace that is not representable with CQL. That metadata will not be shown.

The <keyspacename> argument can be omitted when using a non-system keyspace; in that case, the current keyspace is described.

- DESCRIBE TABLES

  Output is a list of the names of all tables in the current keyspace, or in all keyspaces if there is no current keyspace.
- DESCRIBE TABLE table_name

  Output is a list of CQL commands that could be used to recreate the given table. In some cases, there might be table metadata that is not representable and it is not shown.

Examples

```
DESCRIBE CLUSTER;
```

```
DESCRIBE KEYSPACES;
```

```
DESCRIBE KEYSPACE PortfolioDemo;
```

```
DESCRIBE TABLES;
```

```
DESCRIBE TABLE Stocks;
```

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

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

SHOW

Shows the Cassandra version, host, or data type assumptions for the current cqlsh client session.

Synopsis

```
SHOW VERSION
| HOST
| ASSUMPTIONS
| 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

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

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 Thrift protocol used by the connected Cassandra instance.
- The host information of the Cassandra node that the cqlsh session is currently connected to.
- The data type assumptions for the current cqlsh session as specified by the ASSUME command.

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.

```
SHOW VERSION;

SHOW HOST;

SHOW ASSUMPTIONS;

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

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

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.

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 (see nodetool setprobability):

```
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)
);
```

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 Creating a table, the tracing shows:

- The coordinator 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

(10 rows)
```

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.

For examples of tracing indexed queries and diagnosing performance problems using tracing, see "Request tracing in Cassandra 1.2".

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 a keyspace that uses SimpleStrategy to use NetworkTopologyStrategy.

```
ALTER KEYSPACE "Excalibur" WITH REPLICATION =
  { 'class' : 'NetworkTopologyStrategy', 'dc1' : 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 caching = 'all'. 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 to and from must be compatible. For example, change the 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 password TYPE blob;
```

Altering the type of a column after inserting data can confuse CQL drivers/tools if the new type is incompatible with the data. 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.

The column, in this example bio, must already exist in current rows.

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.
- Adding columns to tables defined with COMPACT STORAGE.
- 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 upgradesstables utility 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 dropped columns to tables using client-supplied timestamps, which are not 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.

Modifying table properties

To change the table storage properties established during creation of the table, use ALTER TABLE and a WITH directive specifying the property and value.

In Cassandra 2.0 and later, you can modify the index_interval, which determines the sample size used in the partition summary, using ALTER TABLE.

```
ALTER TABLE addamsFamily
WITH index_interval = 512;
```

You can probably increase the index_interval to 512 without seeing degradation. Subsequently written SStables written use the new index_interval.

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;
```

The CQL table properties list the properties you can define. Enclose a property in single quotation marks. You cannot modify properties of a table having compact storage.

Modifying the compression or compaction setting

Changing any compaction or compression option erases all previous compaction or compression settings.

```
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};
```

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

```
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 saves network exchanges between the client/server and server coordinator/replicas.

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. Other clients are able to read the first updated rows from the batch, while other rows are in progress. However, transactional row updates within a single partition are isolated: a partial partition update cannot be read.

Using a timestamp

BATCH supports setting a client-supplied timestamp, an integer, in the USING clause. The timestamp applies to all statements in the batch. If not specified, the current time of the insertion (in microseconds) is used.

Individual DML statements inside a BATCH cannot specify a timestamp. If you do not specify a batch-level timestamp, you can specify a timestamp in the individual DML statements.

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 were 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 a batch.

Batching counter updates

Use BEGIN COUNTER BATCH in a batch statement for batched counter updates. Unlike other writes in Cassandra, counter updates are not idempotent.

Example

```
BEGIN BATCH
  INSERT INTO users (userID, password, name) VALUES ('user2', 'ch@ngem3b',
 'second user')
  UPDATE users SET password = 'ps22dhds' WHERE userID = 'user2'
  INSERT INTO users (userID, password) VALUES ('user3', 'ch@ngem3c')
  DELETE name FROM users WHERE userID = 'user2'
  INSERT INTO users (userID, password, name) VALUES ('user4', 'ch@ngem3c',
 'Andrew')
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 ( column_name )
USING (class_name) (WITH OPTIONS = map)
```

Restrictions: *Using class_name* is only allowed 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 it is used, 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.

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.

In this release, 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 does not yet support indexes on collections. 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.

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 a custom index

Cassandra supports creating a custom index using options introduced in Cassandra 2.0.6. Primarily for internal use.

```
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/'};
```

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.

When you configure NetworkTopologyStrategy as the replication strategy, you set up one or more virtual data centers. Use the same names for data centers as those used by the snitch. 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.

To set up the replica placement strategy and data centers, construct a map of properties and values.

Table 9: 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'	An integer	Required if class is SimpleStrategy; otherwise, not used. The number of replicas of data on multiple nodes.
'<data center name>'	A varchar	Required if class is NetworkTopologyStrategy; otherwise, not used. The data center name and number of replicas of data on each node in the data center. Example.
'<data center name>'	A varchar	Optional if class is NetworkTopologyStrategy. The data center name and number of replicas of data on each node in the data center.
. . .	. . .	More optional replication factors for additional named data centers.

CQL property map keys must be lower case. For example, class and replication_factor are correct. Keyspace names are 32 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

Construct the CREATE KEYSPACE statement by first declaring the name of the keyspace, followed by the WITH REPLICATION keywords and the equals symbol. 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`.

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, as previously mentioned, Cassandra will fail to find a node to complete any 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 one replica 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',
  'dc1' : 1 } 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 |
 {"dc1":"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.

CREATE TABLE

Define a new table.

Synopsis

```
CREATE TABLE IF NOT EXISTS keyspace_name.table_name
( column_definition, column_definition, ...)
WITH property AND property ...
```

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 custom type or a CQL data type.
- cql_type of a collection uses this syntax:

```
LIST<cql_type>
| SET<cql_type>
| MAP<cql_type, cql_type>
```

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.

Attempting to create an already existing table 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 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.

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.

Using a compound primary key

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.

Defining a column

You assign columns a type during table creation. Column types, other than collection-type columns, are specified as a parenthesized, comma-separated list of column name and type pairs. See CQL data types for the available types.

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>
);
```

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 CQL table properties. Enclose a string property in single quotation marks. For example:

```
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;
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 };
```

You can specify using compact storage or clustering order using the WITH clause.

Sharing a column in a partition

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 ('k0', 'I''m shared', 0);
INSERT INTO t (k, s, i) VALUES ('k', 'I''m still shared', 1);
SELECT * FROM t2;
```

Output is:

```
 k |                  s | i
----------------------------
k0 | "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.

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);
```

Using compact storage

When you create a table, for every piece of data stored, the column name needs to be stored along with it. 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.

CREATE TRIGGER

Registers a trigger on a table.

Synopsis

```
CREATE TRIGGER 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 initial implementation of triggers includes the capability to register a trigger on a table using the familiar CREATE TRIGGER syntax. This implementation is experimental.

```
CREATE TRIGGER myTrigger
  ON myTable
  USING 'org.apache.cassandra.triggers.InvertedIndex'
```

The actual logic that makes up 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. You place the trigger code in a lib/triggers subdirectory of the Cassandra installation directory, it loads during cluster startup, and exists on every node that participates in a cluster. The trigger defined on a table fires before a requested DML statement occurs, which ensures the atomicity of the transaction.

CREATE USER

Create a new user.

Synopsis

```
CREATE 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

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;
```

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
```

term is:

```
[ list_position ] | key_value
```

row_specification is one of:

```
primary_key_name = key_value
primary_key_name IN ( key_value, 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.

Specifying Columns

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.

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, 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 timestamp 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';
```

DROP INDEX

Drop the named index.

Synopsis

```
DROP INDEX IF EXISTS 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.

Example

```
DROP INDEX user_state;
```

```
DROP INDEX users_zip_idx;
```

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.

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.

Dropping a table triggers an automatic snapshot, which backs up the data only, not the schema.

Example

```
DROP TABLE worldSeriesAttendees;
```

DROP TRIGGER

Removes registration of a trigger.

Synopsis

```
DROP TRIGGER 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 experimental DROP TRIGGER statement removes the registration of a trigger created using CREATE TRIGGER.

DROP USER

Remove a user.

Synopsis

```
DROP USER 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. You have to be logged in as a superuser to issue a DROP USER statement. A user 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 10: 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;
```

To grant access to a keyspace to just one user, assuming nobody else has ALL KEYSPACES access, you use this statement.

```
GRANT ALL ON KEYSPACE keyspace_name TO user_name
```

INSERT

Add or update columns.

Synopsis

```
INSERT INTO keyspace_name.table_name
  ( column_name, column_name...)
  VALUES ( value, value ... ) IF NOT EXISTS
  USING option AND option
```

identifier is a column or a collection name.

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.

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.

The TIMESTAMP input is in microseconds. If not specified, the time (in microseconds) that the write occurred to the column is used.

Using a collection set or map

To insert data into the set, enclose values in curly brackets. Set values must be unique. For example:

```
INSERT INTO users (user_id, 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 (user_id, todo )
  VALUES('frodo', {'2012-10-2 12:10' : 'die' } );
```

Example of inserting data into playlists

About this task

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
```

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 revens.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
| ( TTL (column_name) )
| (function (selector , selector, ...) )
```

function is a timeuuid function, a token function, or a blob conversion function.

relation is:

```
  column_name ( = | < | > | <= | >= ) key_value
| column_name IN ( ( key_value,... ) )
| TOKEN (column_name, ...) ( = |  < | > | <= | >= )
   ( term | TOKEN ( term, ... ) )
```

term is:

- a constant, such as true or false
- a bind marker (?)
- 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 11: 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;
```

In Cassandra 2.0 and later, you can define an alias on columns.

```
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

If you do not specify a limit, a maximum of 10,000 rows are returned by default. Using the LIMIT option, you can specify that the query return a greater or fewer number of rows.

```
SELECT COUNT(*) FROM big_table;
SELECT COUNT(*) FROM big_table LIMIT 50000;
SELECT COUNT(*) FROM big_table LIMIT 200000;
```

The output of these statements is: 10000, 50000, and 105291

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 Migrations table in the system keyspace:

```
SELECT COUNT(*) FROM system.Migrations;
```

Filtering data using WHERE

The WHERE clause specifies which rows to query. The WHERE clause is composed of conditions on the columns that are part of the primary key or are indexed. Use of the primary key in the WHERE clause tells Cassandra to race to the specific node that has the data. Using the equals conditional operators (= or IN) is unrestricted. The term on the left of the operator must be the name of the column, and the term on the right must be the column value to filter on. There are restrictions on other conditional operators.

Cassandra supports these conditional operators: =, >, >=, <, <=, and IN, described later, but not all in certain situations.

In the WHERE clause, refer to a column using the actual name, not the aliases.

- A filter based on a non-equals condition on a partition key is supported only if the partitioner is an ordered one.
- WHERE clauses can include a 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, described in the next section, is also required.

In Cassandra 2.0.6, you can use a new syntax to compare grouped clustering keys to values for slicing over rows in a partition. For example:

```
SELECT * FROM ruling_stewards WHERE (steward_name, king) = ('Boromir',
 'Brego');
```

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, Name and Occupation, from three rows having employee ids (primary key) 199, 200, or 207:

```
SELECT Name, Occupation FROM People WHERE empID IN (199, 200, 207);
```

Format values for the IN conditional test as a comma-separated list. The list can consist of a range of column values.

In Cassandra 2.0, 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 having 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.

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.

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. The RandomPartitioner and Murmur3Partitioner do not yield a meaningful order.

For example, assuming you have this table defined, the following query shows how to use the TOKEN function:

```
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)
);
SELECT * FROM periods
  WHERE TOKEN(period_name) > TOKEN('Third Age')
  AND TOKEN(period_name) < TOKEN('Fourth Age');
```

Querying 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      | title
-------------+------------+-----------------------+----------------+--------------+------
 62c36092... |          4 | No One Rides for Free |      Fu Manchu | 7db1a490... |
 62c36092... |          3 |             Roll Away | Back Door Slam | 2b09185b... | Outsi
 62c36092... |          2 |          We Must Obey |      Fu Manchu | 8a172618... |    Mo
 62c36092... |          1 |          Tres Hombres |         ZZ Top | a3e64f8f... |
```

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 title FROM playlists WHERE artist = 'Fu Manchu';
```

Output

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

is:

Querying a collection set, list, or map

When you query a table containing a collection, Cassandra retrieves the collection in its entirety.

To return the set of email belonging to frodo, for example:

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

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

(1 rows)
```

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. Truncating a table triggers an automatic snapshot, which backs up the data only, not the schema.

Examples

```
TRUNCATE user_activity;
```

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 . . .
```

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 ( + | - ) ...
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 to existing columns in 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. 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 multiple columns, separate the name-value pairs using commas.

The SET clause specifies the column name-value pairs to update. Separate multiple name-value pairs using commas.

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:

- 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;
```

In Cassandra 2.0, 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

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;
```

Update the value of a counter column:

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

Using 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';
```

Using 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['2012-10-2 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';
```

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';
```

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";
```