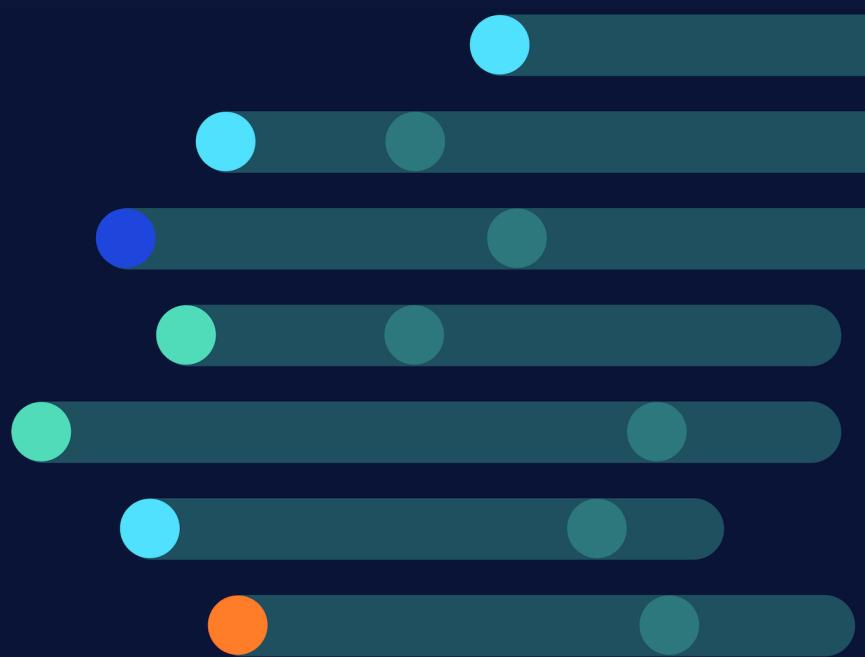


CQL for Apache Cassandra 3.0



© 2020 DataStax, Inc. All rights reserved.

DataStax, Titan, and TitanDB are registered trademarks of DataStax, Inc. and its subsidiaries in the United States and/or other countries.

Apache, Apache Cassandra, Cassandra, Apache Tomcat, Tomcat, Apache Lucene, Apache Solr, Apache Hadoop, Hadoop, Apache Spark, Spark, Apache TinkerPop, TinkerPop, Apache Kafka and Kafka are either

registered trademarks or trademarks of the Apache Software Foundation or its subsidiaries in Canada, the United States and/or other countries.

Kubernetes is the registered trademark of the Linux Foundation.

Contents

Introduction to Cassandra Query Language.....	1
CQL data modeling.....	3
Data modeling concepts.....	3
Data modeling analysis.....	5
Using materialized views.....	6
Understanding materialized views.....	6
Known limitations.....	8
Creating a materialized view.....	10
Altering a materialized view.....	13
Dropping a materialized view.....	13
Best practices.....	13
FAQ.....	14
Using CQL.....	15
Starting cqlsh on Linux and Mac OS X.....	15
Creating and updating a keyspace.....	16
Example of creating a keyspace.....	17
Updating the replication factor.....	18
Creating a table.....	19
Creating a table.....	19
Using the keyspace qualifier.....	21
Simple Primary Key.....	22
Composite Partition Key.....	23
Compound Primary Key.....	25
Creating a counter table.....	26
Create table with COMPACT STORAGE.....	28
Migrating from compact storage.....	28
Table schema collision fix.....	30
Creating advanced data types in tables.....	31
Creating collections.....	31
Creating a table with a tuple.....	33

Creating a user-defined type (UDT).....	34
Creating functions.....	35
Creating user-defined function (UDF).....	35
Creating User-Defined Aggregate Function (UDA).....	36
Inserting and updating data.....	37
Inserting simple data into a table.....	37
Inserting and updating data into a set.....	38
Inserting and updating data into a list.....	39
Inserting and updating data into a map.....	40
Inserting tuple data into a table.....	41
Inserting or updating data into a user-defined type (UDT).....	42
Inserting JSON data into a table.....	43
Using lightweight transactions.....	44
Sharing a static column.....	44
Expiring data with time-to-live.....	45
Batching data insertion and updates.....	47
Batching inserts, updates and deletes.....	47
Good use of BATCH statement.....	48
Misuse of BATCH statement.....	52
Querying tables.....	52
Retrieval and sorting results.....	52
Retrieval using collections.....	55
Retrieval using JSON.....	57
Retrieval using the IN keyword.....	57
Retrieval by scanning a partition.....	59
Retrieval using standard aggregate functions.....	59
Retrieval using a user-defined function (UDF).....	61
Retrieval using user-defined aggregate (UDA) functions.....	61
Querying a system table.....	62
Indexing.....	68
When to use an index.....	68
Using a secondary index.....	69
Using multiple indexes.....	71

Indexing a collection.....	72
Altering a table.....	74
Altering columns in a table.....	75
Altering a table to add a collection.....	75
Altering the data type of a column.....	76
Altering the table properties.....	76
Altering a user-defined type.....	77
Removing a keyspace, schema, or data.....	78
Dropping a keyspace or table.....	78
Deleting columns and rows.....	78
Dropping a user-defined function (UDF).....	78
Securing a table.....	79
Database roles.....	79
Database Permissions.....	80
Database users.....	82
Tracing consistency changes.....	83
Setup to trace consistency changes.....	84
Trace reads at different consistency levels.....	85
How consistency affects performance.....	88
Displaying rows from an unordered partitioner with the TOKEN function.....	89
Determining time-to-live (TTL) for a column.....	91
Determining the date/time of a write.....	92
Legacy tables.....	93
Working with legacy applications.....	93
Querying a legacy table.....	93
Using a CQL legacy table query.....	94
CQL reference.....	95
Introduction.....	95
CQL lexical structure.....	95
Uppercase and lowercase.....	95
Escaping characters.....	96
Valid literals.....	97
Exponential notation.....	98

CQL code comments.....	98
CQL Keywords.....	99
CQL data types.....	103
Blob type.....	107
Collection type.....	108
Counter type.....	108
UUID and timeuuid types.....	109
UUID and timeuuid functions.....	109
Timestamp type.....	111
Tuple type.....	112
User-defined type.....	113
Functions.....	113
CQL limits.....	114
CQL shell commands.....	114
Starting cqlsh.....	115
Configuring cqlsh from a file.....	119
CAPTURE.....	127
CLEAR.....	129
CONSISTENCY.....	130
COPY.....	135
DESCRIBE.....	144
cqlshExpand.....	147
EXIT.....	150
LOGIN.....	151
PAGING.....	152
SERIAL CONSISTENCY.....	154
SHOW.....	156
SOURCE.....	159
TRACING.....	160
CQL commands.....	165
ALTER KEYSPACE.....	165
ALTER MATERIALIZED VIEW.....	167
ALTER ROLE.....	170

ALTER TABLE.....	172
ALTER TYPE.....	181
ALTER USER.....	183
BATCH.....	185
CREATE AGGREGATE.....	191
CREATE INDEX.....	195
CREATE FUNCTION.....	199
CREATE KEYSPACE.....	202
CREATE MATERIALIZED VIEW.....	206
CREATE TABLE.....	209
CREATE TRIGGER.....	226
CREATE TYPE.....	228
CREATE ROLE.....	230
CREATE USER (Deprecated).....	233
DELETE.....	236
DROP AGGREGATE.....	241
DROP FUNCTION.....	242
DROP INDEX.....	243
DROP KEYSPACE.....	245
DROP MATERIALIZED VIEW.....	246
DROP ROLE.....	248
DROP TABLE.....	250
DROP TRIGGER.....	251
DROP TYPE.....	253
DROP USER (Deprecated).....	254
GRANT.....	256
INSERT.....	263
LIST PERMISSIONS.....	267
LIST ROLES.....	271
LIST USERS (Deprecated).....	273
REVOKE.....	275
SELECT.....	278
TRUNCATE.....	292

UPDATE.....	294
USE.....	304

1. Introduction to Cassandra Query Language

About this document

Welcome to the CQL documentation provided by DataStax. To ensure that you get the best experience in using this document, take a moment to look at the [Tips for using DataStax documentation](#).

Overview of the Cassandra Query Language

Cassandra Query Language (CQL) is a query language for the Cassandra database. This release of CQL works with Cassandra 3.0.

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

Important: This document assumes you are familiar with the [Cassandra 3.0 documentation](#).

Table 1. CQL for Cassandra 3.0 features

New CQL features	<ul style="list-style-type: none">• JSON support for CQL3• User Defined Functions (UDFs)• User Defined Aggregates (UDAs)• Role Based Access Control (RBAC)• Native Protocol v.4• Materialized Views• Addition of CLEAR command for cqlsh
Improved CQL features	<ul style="list-style-type: none">• Additional COPY command options• New WITH ID option with <code>CREATE TABLE</code> command• Support IN clause on any partition key column or clustering column• Accept Dollar Quoted Strings• Allow Mixing Token and Partition Key Restrictions• Support Indexing Key/Value Entries on Map Collections

Table 1. CQL for Cassandra 3.0 features (continued)

	<ul style="list-style-type: none">• Date data type added and improved time/date conversion functions• Tinyint and smallint data types added• Change to CREATE TABLE syntax for compression options
Removed CQL features	<ul style="list-style-type: none">• Removal of CQL2• Removal of cassandra-cli
Native protocol	<ul style="list-style-type: none">• The Native Protocol has been updated to version 4, with implications for CQL use in the DataStax drivers.

2. CQL data modeling

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

Data modeling concepts

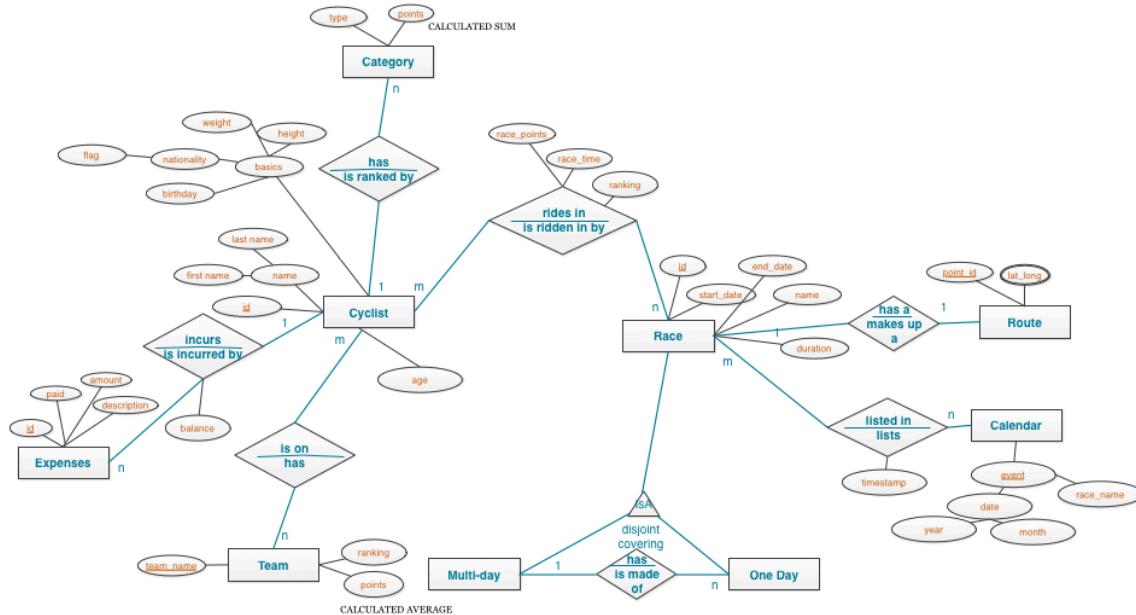
Data modeling is a process that involves identifying the entities (items to be stored) and the relationships between entities. To create your data model, identify the patterns used to access data and the types of queries to be performed. These two ideas inform the organization and structure of the data, and the design and creation of the database's tables. [Indexing the data](#) can lead to either performance or degradation of queries, so understanding indexing is an important step in the data modeling process.

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

In contrast, relational databases normalize data based on the tables and relationships designed, and then writes the queries that will be made. Data modeling in relational databases is table-driven, and any relationships between tables are expressed as table joins in queries.

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

Cassandra data modeling focuses on the queries. Throughout this topic, the example of Pro Cycling statistics demonstrates how to model the Cassandra table schema for specific queries. The conceptual model for this data model shows the entities and relationships.



The entities and their relationships are considered during table design. Queries are best designed to access a single table, so all entities involved in a relationship that a query encompasses must be in the table. Some tables will involve a single entity and its attributes, like the first example shown below. Others will involve more than one entity and its attributes, such as the second example. Including all data in a single Cassandra table contrasts with a relational database approach, where the data would be stored in two or more tables and foreign keys would be used to relate the data between the tables. Because Cassandra uses this single table-single query approach, queries can perform faster.

One basic query (Q1) for Pro Cycling statistics is a list of cyclists, including each cyclist's **id**, **firstname**, and **lastname**. To uniquely identify a cyclist in the table, an **id** using UUID is used. For a simple query to list all cyclists a table that includes all the columns identified and a partition key (K) of **id** is created. The diagram below shows a portion of the logical model for the Pro Cycling data model.

Figure 1. Query 1: Find a cyclist's name with a specified id

cyclist_name	
id	K
lastname	
firstname	

partition key

A related query (Q2) searches for all cyclists by a particular race category. For Cassandra, this query is more efficient if a table is created that groups all cyclists by category. Some of the same columns are required (**id**, **lastname**), but now the primary key of the table includes **category** as the partition key (K), and groups within the partition by the **id** (C). This choice ensures that unique records for each cyclist are created.

Figure 2. Query 2: Find cyclists given a specified category

cyclist_category	
category	K
id	C!
points	
lastname	

partition key
clustering column

These are two simple queries; more examples will be shown to illustrate data modeling using CQL.

Notice that the main principle in designing the table is not the relationship of the table to other tables, as it is in relational database modeling. Data in Cassandra is often arranged as one query per table, and data is repeated amongst many tables, a process known as [denormalization](#). Relational databases instead [normalize](#) data, removing as much duplication as possible. The relationship of the entities is important, because the order in which data is stored in Cassandra can greatly affect the ease and speed of data retrieval. The schema design captures much of the relationship between entities by including related attributes in the same table. Client-side joins in application code is used only when table schema cannot capture the complexity of the relationships.

Data modeling analysis

You've created a conceptual model of the entities and their relationships. From the conceptual model, you've used the expected queries to create table schema. The last step in data model involves completing an analysis of the logical design to discover modifications that might be needed. These modifications can arise from understanding partition size limitations, cost of data consistency, and performance costs due to a number of design choices still to be made.

For efficient operation, partitions must be sized within certain limits. Two measures of partition size are the number of values in a partition and the partition size on disk. The maximum number of rows per partition is not theoretically limited, although practical limits can be found with experimentation. Sizing the disk space is more complex, and involves the number of rows and the number of columns, primary key columns and static columns in each table. Each application will have different efficiency parameters, but a good rule of thumb is to keep the maximum number of values below 100,000 items and the disk size under 100MB.

Data redundancy must be considered as well. Two redundancies that are a consequence of Cassandra's distributed design are duplicate data in tables and multiple partition replicates.

Data is generally duplicated in multiple tables, resulting in performance latency during writes and requires more disk space. Consider storing a cyclist's name and id in more than one data, along with other items like race categories, finished races, and cyclist statistics. Storing the name and id in multiple tables results in linear duplication, with two values stored in each table. Table design must take into account the possibility of higher order duplication, such as unlimited keywords stored in a large number of rows. A case of n

keywords stored in m rows is not a good table design. You should rethink the table schema for better design, still keeping the query foremost.

Cassandra replicates partition data based on the replication factor, using more disk space. Replication is a necessary aspect of distributed databases and sizing disk storage correctly is important.

Application-side joins can be a performance killer. In general, you should analyze your queries that require joins and consider pre-computing and storing the join results in an additional table. In Cassandra, the goal is to use one table per query for performant behavior. Lightweight transactions (LWT) can also affect performance. Consider whether or not the queries using LWT are necessary and remove the requirement if it is not strictly needed.

Using materialized views

In Cassandra, a [materialized view](#) is a table built from data in another table with a new primary key and new properties. Queries are optimized by the primary key definition. Standard practice is to create a table for the query, and create a new table with the same data if a different query is needed. Client applications then manually update the additional tables as well as the original. In the materialized view, data is updated automatically by changes to the source table.

Understanding materialized views

Learn how Cassandra propagates updates from a base table to its materialized views, and consider the performance impacts and consistency requirements.

How materialized views work

The following steps illustrate how Cassandra propagates updates from a base table to its materialized views.

1. The coordinator node receives an update from a client for the base table and forwards it to the configured replica nodes.
 - a. When the `cassandra.mv_enable_coordinator_batchlog` property is enabled, the coordinator will write a batchlog to QUORUM nodes containing the base table write before forwarding them to the replicas. This configuration provides better protection against a coordinator failing in the middle of a request, but slows the view write operation considerably. See [CASSANDRA-10230](#) for more information about the batchlog coordinator.
2. Upon receiving an update from the coordinator for the base table, each replica node completes the following tasks:
 - a. Generate view updates for each materialized view of the base table.

- A local read is completed in the base table row to determine if a previous view row must be removed or modified.
 - A local lock is acquired on the base table partition when generating the view update to ensure that the view updates are serialized. This lock is released after updates to the view are propagated to the replicas and base updates are applied locally.
- b. After generating view updates, deterministically compute its paired view replica for each view update, so that the view replication work is distributed among base replicas.
- If the base replica is also a view replica, the base replica chooses itself as the paired view replica, and applies the view update synchronously.
 - Otherwise, the update is written synchronously to the local **batchlog** for durability, and sent asynchronously to the remote paired view replica.
- c. Acknowledge the write to the coordinator node.
- d. After receiving an acknowledgement of all asynchronous paired view writes, remove the local batchlog. Otherwise, replay the batchlog at a later time to propagate the view update to the replica. If a replica is down during batchlog replay, one hint is written for each mutation.
3. After receiving an acknowledgement from all nodes (based on consistency level), the coordinator node returns a successfully write response to the client.

For additional information on how materialized views work, see the following posts on the DataStax Developer [Understanding the guarantees, limitations, and tradeoffs of materialized views](#) blog.

Performance considerations

Materialized views allow fast lookup of data using the normal read path. However, materialized views do not have the same write performance as normal table writes because the database performs an additional read-before-write operation to update each materialized view. To complete an update, the database performs a data consistency check on each replica. A write to the source table incurs latency (~10% for each materialized view), and the performance of deletes on the source table also suffers.

If a delete on the source table affects two or more contiguous rows, this delete is tagged with one tombstone. However, these same rows may not be contiguous in materialized views derived from the source table. If they are not, the database creates multiple tombstones in the materialized views.

Additional work is required to ensure that all correct state changes to a given row are applied to materialized views, especially regarding concurrent updates. By using materialized views, performance is traded for data correctness.

Consistency considerations

Each base table replica writes the view updates locally (when it is also a view replica), or writes a local batchlog before returning the base table write (as described in [2.b](#)). If the base table replica cannot update a remote view during the write operation, the replica retries the update during batchlog replay. This mechanism ensures that all changes to each base table replica are reflected in the views, unless data loss occurs in the base table replica.

The write operation for the view replica is asynchronous to ensure availability is not compromised. A consequence is that a read operation for a view might not immediately see a successful write to the base table until the write operation is propagated by the base replicas. Under normal conditions, data is quickly made available in the views. Use the `ViewWriteMetrics` metric to track the view propagation time.

Scenario that can result in base-view inconsistency

In an ordinary Cassandra table, when a row is successfully written to consistency level replicas, data loss can occur if those replicas become permanently unavailable before the update is propagated to the remaining replicas. The following example illustrates this scenario.

1. Write to a table with a replication factor of three (RF=3) and a consistency level of ONE.
2. The base replica is also the coordinator node.
3. The coordinator responds to the client that the write was successful.
4. The machine hosting the coordinator node dies.

In the case of materialized views, the previous example carries additional implications. If the base table (coordinator node) successfully wrote the view update to another node, the row will exist only in the view but not in the base table, creating an orphaned view row.

Another scenario that can create an orphaned view row is when a base table row loses all replicas without repair between failures. If a view row loses its replicas, the base table row will not have its corresponding view row.

To avoid inconsistency between base tables and materialized views, review the [Best practices for materialized views](#).

Related information

[Learn more about materialized views](#)

Known limitations of materialized views

As of writing, the following limitations are known for materialized views.

- Currently, there is not way to automatically detect and fix permanent inconsistency between the base and the view ([CASSANDRA-10346](#))
- Incremental repair is not supported on base tables with materialized views ([CASSANDRA-12888](#))
- Cannot filter materialized views by non-primary key columns ([CASSANDRA-13798](#))
- Deleting individual columns from a base table not selected in a materialized view can potentially prevent updates with lower timestamps (from repair or hints) from being applied ([CASSANDRA-13826](#)).

To illustrate this limitation, consider the following statement, which creates a base table with primary key columns `base_pk1` and `base_pk2`.

```
CREATE TABLE base_table (
    base_pk1 int,
    base_pk2 int,
    view_pk int,
    unselected int)
PRIMARY KEY (base_pk1, base_pk2);
```

A materialized view is created, including the base table primary keys `base_pk1` and `base_pk2`, plus an additional column `view_pk` on its primary key. The `unselected` column from the base table is not included in the view definition.

```
CREATE MATERIALIZED VIEW mv AS SELECT
    base_pk1,
    base_pk2,
    view_pk
FROM base_table
PRIMARY KEY (base_pk2, base_pk1, view_pk);
```

A `DELETE` statement is issued to the `unselected` column of the base table.

```
DELETE unselected from base_table
WHERE base_pk1=1 AND base_pk2=1
USING TIMESTAMP 500;
```

Later, an update with a lower timestamp arrives from a repair or hint.

```
INSERT INTO base_table (
    base_pk1,
    base_pk2,
    view_pk)
```

```
VALUES (1, 1, 1) USING TIMESTAMP 100;
```

After this update, the following SELECT statement will not return the row previously inserted.

```
SELECT * FROM mv WHERE base_pk1 = 1 AND base_pk2 = 1 and view_pk =1;
```

To overcome this limitation, always include columns that might be individually deleted in the materialized view definition when the primary key contains a column not present in the primary key of the base table. Alternatively, avoid performing individual column deletions on materialized views with these attributes.

- Full repairs on base tables must go through the write path to generate view updates, which can cause higher load and increased repair time (compared to a table without materialized views) when many changes are present.

Creating a materialized view

Materialized views are suited for high cardinality data. The data in a materialized view is arranged serially based on the view's primary key. Materialized views cause hotspots when low cardinality data is inserted.

[Secondary indexes](#) are suited for low cardinality data. Queries of high cardinality columns on secondary indexes require Cassandra to access all nodes in a cluster, [causing high read latency](#).

Restrictions for materialized views:

- Include all of the source table's primary keys in the materialized view's primary key.
- Only one new column can be added to the materialized view's primary key. [Static columns](#) are not allowed.
- Exclude rows with null values in the materialized view primary key column.

You can create a materialized view with its own WHERE conditions and its own properties.

Materialized view example

The following table is the original, or source, table for the materialized view examples in this section.

```
CREATE TABLE cyclist_mv (cid UUID PRIMARY KEY, name text, age int,
    birthday date, country text);
```

This table holds values for the name, age, birthday, and country affiliation of several

cid	age	birthday	country	name
ffdःa2a7-5fc6-49a7-bfdc-3fcdfdd7156	18	1997-02-08	Netherlands	Pascal EENHOORN
15a116fc-b833-4da6-ab9a-4a7775752836	18	1997-08-19	United States	Adrien COSTA
e7ae5cf3-d358-4d99-b900-85902fda9bb0	22	1993-06-18	New Zealand	Alex FRAME
c9c9c484-5e4a-4542-8203-8d047a01b8a8	27	1987-09-04	Brazil	Cristian EGIDIO
d1aad83b-be60-47a4-bd6e-069b8da0d97b	27	1987-09-04	Germany	Johannes HEIDER
862cc51f-00a1-4d5a-a359cab7300e	20	1994-09-04	Denmark	Joakim BUKDAL
18f471bf-f631-4bc4-a9a2-d6f6cf5ea503	18	1997-03-29	Netherlands	Bram WELTEN
220844bf-4860-49d6-9a4b-6b5d3a79cbfb	38	1977-07-08	Italy	Paolo TIRALONGO
6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47	28	1987-06-07	Netherlands	Steven KRIKSWIJK

cyclists.

The `cyclist_mv` table can be the basis of a materialized view that uses `age` in the primary key.

```
CREATE MATERIALIZED VIEW cyclist_by_age
AS SELECT age, birthday, name, country
FROM cyclist_mv
WHERE age IS NOT NULL AND cid IS NOT NULL
PRIMARY KEY (age, cid);
```

This `CREATE MATERIALIZED VIEW` statement has several features:

- The `AS SELECT` phrase identifies the columns copied from the base table to the materialized view.
- The `FROM` phrase identifies the source table from which Cassandra will copy the data.
- The `WHERE` clause must include all primary key columns with the `IS NOT NULL` phrase so that only rows with data for all the primary key columns are copied to the materialized view.
- As with any table, the materialized view must specify the primary key columns. Because `cyclist_mv`, the source table, uses `cid` as its primary key, `cid` must be present in the materialized view's primary key.

Note: In this materialized view, `age` is used as the primary key and `cid` is a clustering column. In Cassandra3.0 and earlier, clustering columns have a maximum size of 64 KB.

CQL data modeling

Because the new materialized view is partitioned by `age`, it supports queries based on the cyclists' ages.

```
SELECT age, name, birthday FROM cyclist_by_age WHERE age = 18;
```

age	name	birthday
18	Adrien COSTA	1997-08-19
18	Bram WELTEN	1997-03-29
18	Pascal EENKHOORN	1997-02-08

Other materialized views, based on the same source table, can organize information by cyclists' birthdays or countries of origin.

```
CREATE MATERIALIZED VIEW cyclist_by_birthday
AS SELECT age, birthday, name, country
FROM cyclist_mv
WHERE birthday IS NOT NULL AND cid IS NOT NULL
PRIMARY KEY (birthday, cid);
```

```
CREATE MATERIALIZED VIEW cyclist_by_country
AS SELECT age, birthday, name, country
FROM cyclist_mv
WHERE country IS NOT NULL AND cid IS NOT NULL
PRIMARY KEY (country, cid);
```

The following queries use the new materialized views.

```
SELECT age, name, birthday FROM cyclist_by_country WHERE country =
'Netherlands';
```

age	name	birthday
18	Bram WELTEN	1997-03-29
28	Steven KRIKSWIJK	1987-06-07
18	Pascal EENKHOORN	1997-02-08

```
SELECT age, name, birthday FROM cyclist_by_birthday WHERE birthday =
'1987-09-04';
```

age	name	birthday
27	Cristian EGIDIO	1987-09-04
27	Johannes HEIDER	1987-09-04

When another `INSERT` is executed on `cyclist_mv`, Cassandra updates the source table and both of these materialized views. When data is deleted from `cyclist_mv`, Cassandra deletes the same data from any related materialized views.

Cassandra can only write data directly to source tables, not to materialized views. Cassandra updates a materialized view asynchronously after inserting data into the source

table, so the update of materialized view is delayed. Cassandra performs a read repair to a materialized view only after updating the source table.

[Altering a materialized view](#)
[CREATE MATERIALIZED VIEW](#)
[ALTER MATERIALIZED VIEW](#)
[DROP MATERIALIZED VIEW](#)

Altering a materialized view

A materialized view has [table properties](#) like its source tables. Use the `ALTER MATERIALIZED VIEW` command alter the view's properties. Specify updated properties and values in a `WITH` clause. Materialized views do not perform repair, so properties regarding repair are invalid.

Procedure

Alter a materialized view to change the caching properties.

```
cqlsh> ALTER MATERIALIZED VIEW cycling.cyclist_by_birthday
    WITH caching = { 'keys' : 'NONE', 'rows_per_partition' : '15' };
```

[Creating a materialized view](#)
[CREATE MATERIALIZED VIEW](#)
[ALTER MATERIALIZED VIEW](#)
[DROP MATERIALIZED VIEW](#)

Dropping a materialized view

Use the `DROP` command to drop a materialized view.

Procedure

Drop the `cycling.cyclist_by_age` materialized view in Cassandra 3.0 and later.

```
DROP MATERIALIZED VIEW cycling.cyclist_by_age;
```

[CREATE MATERIALIZED VIEW](#)
[ALTER MATERIALIZED VIEW](#)
[DROP MATERIALIZED VIEW](#)

Best practices for materialized views

Because there is currently no automatic mechanism to detect and fix inconsistencies between the base and view (other than dropping and recreating the view), adhere to the following best practices to ensure consistency between the base and table views.

- Write to base tables with materialized views using consistency levels greater than `ONE` (such as `LOCAL_QUORUM`) to avoid [base-view inconsistency](#). Alternatively, use the `-Dmv_enable_coordinator_batchlog=true` option to provide better protection against a coordinator failing in the middle of a request.

Note: Using the `-Dmv_enable_coordinator_batchlog=true` option will slow the view write operation considerably.

- Run repair on both the base table and the view whenever a node is removed, replaced, down for longer than the value specified by `max_hint_window_in_ms`, or a new datacenter is added. This recommendation is valid to prevent data loss for any tables, not just tables with materialized views.
- Run repair periodically on views (at least one time every period specified by `gc_grace_seconds`) to ensure that tombstones for views are successfully propagated to all replicas, and to prevent data resurrection. This recommendation is valid for any tables where delete operations occurred, such as manually denormalized tables.

Frequently asked questions about materialized views

Can materialized views be used in production environments?

Before using materialized views, be aware of the [known limitations](#) and test them against your application requirements to determine if materialized views are suitable for your environment.

After materialized views are deployed, regular maintenance repairs are required to ensure that base tables and views are consistent. Provided that limitations are validated against the application and [best practices](#) are observed, materialized views can be deployed in production environments.

Is manual denormalization better than using materialized views?

This choice depends on the use case and requirements. Ensuring consistency between views and tables in the face of complex failures and concurrent updates requires additional mechanisms (such as row locking, view repair, and paired view replication), which requires extra work. In practice, no guarantees are lost when using built-in materialized views versus manually denormalized tables.

One differentiator of doing manual denormalization versus using materialized views is when consistency is less important, or data is never updated or deleted. In these instances, write to multiple tables from the client rather than using materialized views.

3. Using CQL

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

Accessing CQL

Common ways to access CQL are:

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

This document presents examples using `cqlsh`.

Starting cqlsh on Linux and Mac OS X

This procedure briefly describes how to start `cqlsh` on Linux and Mac OS X. The `cqlsh` command is covered in detail later.

Procedure

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

```
$ bin/cqlsh
```

If you use security features, provide a user name and password.

3. Print the help menu for `cqlsh`.

```
$ bin/cqlsh --help
```

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

```
$ bin/cqlsh 1.2.3.4 9042
```

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

```
$ easy_install readline
```

Creating and updating a keyspace

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

When you create a keyspace, specify a [Table 39: Replication strategy class and factor settings](#) for replicating keyspaces. Using the `SimpleStrategy` class is fine for evaluating Cassandra. For production use or for use with mixed workloads, use the `NetworkTopologyStrategy` class.

To use `NetworkTopologyStrategy` for evaluation purposes using, for example, a single node cluster, the default data center name is used. 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; see [Snitch](#). For example, if the cluster uses the `PropertyFileSnitch`, create the keyspace using the user-defined data center and rack names in the `cassandra-topologies.properties` file. If the cluster uses the `Ec2Snitch`, create the keyspace using EC2 data center and rack names. If the cluster uses the `GoogleCloudSnitch`, create the keyspace using GoogleCloud data center and rack names.

If you fail to change the default snitch and use `NetworkTopologyStrategy`, 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.
```

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

Related reference

[CREATE KEYSPACE](#)

Example of creating a keyspace

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

Procedure

1. Determine the default data center name, if using `NetworkTopologyStrategy`, using `nodetool status`.

```
$ bin/nodetool status
```

The output is:

```
Datacenter: datacenter1
=====
Status=Up/Down
| / State=Normal/Leaving/Joining/Moving
```

Using CQL

```
-- Address      Load        Tokens  Owns (effective)  Host ID      Rack
UN  127.0.0.1   41.62 KB    256     100.0%           75dcca8f...  rack1
```

2. Create a keyspace.

```
cqlsh> CREATE KEYSPACE IF NOT EXISTS cycling WITH REPLICATION =
{ 'class' : 'NetworkTopologyStrategy', 'datacenter1' : 3 };
```

3. Use the keyspace.

```
cqlsh> USE cycling;
```

Updating the replication factor

Increasing the replication factor increases the total number of copies of keyspace data stored in a Cassandra cluster. For more information about replication, see [Data replication](#).

When you change the replication factor of a keyspace, you affect each node that the keyspaces replicates to (or no longer replicates to). Follow this procedure to prepare all affected nodes for this change.

Procedure

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

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

Or if using **SimpleStrategy**:

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

Note: Datacenter names are case sensitive. Verify the case of the using utility, such as `nodetool status`.

2. On each affected node, run `nodetool repair` with the `-full` option.

3. Wait until repair completes on a node, then move to the next node.

For more about replication strategy options, see [Changing keyspace replication strategy](#)

What's next:

What's next

Changing the replication factor of the `system_auth` keyspace

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

Restricting replication for a keyspace

The example above shows how to configure a keyspace to create different numbers of replicas on different data centers. In some cases, you may want to prevent the keyspace from sending replicas to particular data centers — or restrict a keyspace to just one data center.

To do this, use `ALTER KEYSPACE` to configure the keyspace to use `NetworkTopologyStrategy`, as shown above. You can prevent the keyspace from sending replicas to a specific datacenter by setting its replication factor to 0 (zero). For example:

```
cqlsh> ALTER KEYSPACE keyspace1 WITH REPLICATION =
    {'class' : 'NetworkTopologyStrategy', 'dc1' : 0, 'dc2' : 3, 'dc3' : 0 };
```

This command configures `keyspace1` to create replicas only on `dc2`. The data centers `dc1` and `dc3` receive no replicas from tables in `keyspace1`.

Related reference

[ALTER KEYSPACE](#)

Creating a table

In CQL, data is stored in tables containing rows of columns.

Creating a table

In CQL, data is stored in tables containing rows of columns, similar to SQL definitions.

Note: The concept of rows and columns in the internal implementation of Cassandra are **not** the same. For more information, see [A thrift to CQL3 upgrade guide](#) or [CQL3 for Cassandra experts](#).

Tables can be created, dropped, and altered at runtime without blocking updates and queries. To create a table, you must define a primary key and other data columns. Add the optional `WITH` clause and keyword arguments to configure table properties (caching, compaction, etc.). See the [table_options](#) page for details.

Create schema using cqlsh

Create table schema using `cqlsh`. Cassandra does not support dynamic schema generation — collision can occur if multiple clients attempt to generate tables simultaneously. To recover from collisions, follow the instructions in [schema collision fix](#).

Primary Key

A [primary key](#) identifies the location and order of stored data. The primary key is defined when the table is created and cannot be altered. If you must change the primary key, create a new table schema and write the existing data to the new table. See [ALTER TABLE](#) for details on altering a table after creation.

Cassandra is a partition row store. The first element of the primary key, the partition key, specifies which node will hold a particular table row. At the minimum, the primary key must consist of a [partition key](#). You can define a compound partition key to split a data set so that related data is stored on separate partitions. A compound primary key includes [clustering columns](#) which order the data on a partition.

Note: In Cassandra 3.0 and earlier, you cannot insert any value larger than 64K bytes into a clustering column.

The definition of a table's primary key is critical in Cassandra. Carefully model how data in a table will be inserted and retrieved before choosing which columns to define in the primary key. The size of the partitions, the order of the data within partitions, the distribution of the partitions among the nodes of the cluster — you must consider all of these when selecting the table's primary key.

Table characteristics

The name of a table can be a string of alphanumeric characters and underscores, but it must begin with a letter. Tips for the table name:

- To specify the keyspace that contains the table, put the keyspace name followed by a period before the table name: `keyspace_name.table_name`. This allows you to create a new table in a keyspace that is different from the one set for the current session (by the `USE` command, for example).
- To create a table in the current keyspace, just use the new table name.

Column characteristics

CQL supports several column types. You assign a [data type](#) to each column when you create a table. The table definition defines (non-collection) columns in a comma-delimited list of name and type pairs. The following example illustrates three data types, **UUID**, **text** and **timestamp**:

```
CREATE TABLE cycling.cyclist_alt_stats ( id UUID PRIMARY KEY, lastname text, birthday timestamp, nationality text, weight text, height text );
```

CQL supports the following collection column types: **map**, **set**, and **list**. A collection column is defined using the collection type, followed by another type, such as int or text, in angle brackets. The collection column definition is included in the column list as described

above. The following example illustrates each collection type, but is not designed for an actual query:

```
CREATE TABLE cycling.whimsey ( id UUID PRIMARY KEY, lastname text,
cyclist_teams set<text>, events list<text>, teams map<int,text> );
```

Collection types cannot be nested. Collections can include **frozen** data types. For examples and usage, see [Collection type](#).

A column of type **tuple** holds a fixed-length set of typed positional fields. Use a **tuple** as an alternative to a user-defined type. A **tuple** can accommodate many fields (32768) — although it would not be a good idea to use this many. A typical **tuple** holds 2 to 5 fields. Specify a **tuple** in a table definition, using angle brackets; within these, use a comma-delimited list to define each component type. **Tuples** can be nested. The following example illustrates a **tuple** type composed of a **text** field and a nested **tuple** of two **float** fields:

```
CREATE TABLE cycling.route (race_id int, race_name text, point_id
int, lat_long tuple<text, tuple<float,float>>, PRIMARY KEY (race_id,
point_id));
```

Note: Cassandra no longer requires the use of **frozen** for **tuples**:

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

For more information, see ["Tuple type"](#).

Create a User-defined type (UDTs) as a data type of several fields, using [CREATE TYPE](#). It is best to create a UDT for use with multiple table definitions. The user-defined column type (UDT) requires the **frozen** keyword. A frozen value serializes multiple components into a single value. Non-frozen types allow updates to individual fields. Cassandra treats the value of a frozen type as a blob. The entire value must be overwritten. The scope of a user-defined type is the keyspace in which you define it. Use dot notation to access a type from a keyspace outside its scope: keyspace name followed by a period followed the name of the type, for example: `test.myType` where `test` is the keyspace name and `myType` is the type name. Cassandra accesses the type in the specified keyspace, but does not change the current keyspace; otherwise, if you do not specify a keyspace, Cassandra accesses the type within the current keyspace. For examples and usage information, see ["Using a user-defined type"](#).

A counter is a special column used to store a number that is changed in increments. A counter can only be used in a dedicated table that includes a column of [counter data type](#). For more examples and usage information, see ["Using a counter"](#).

Using the keyspace qualifier

Sometimes issuing a USE statement to select a keyspace is inconvenient. Connection pooling requires managing multiple keyspaces. To simplify tracking multiple keyspaces,

Using CQL

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 that contains the table, use the name of the keyspace followed by a period, then the table name. For example, **cycling.race_winners**.

```
cqlsh> INSERT INTO cycling.race_winners ( race_name, race_position,
cyclist_name ) VALUES (
    'National Championships South Africa WJ-ITT (CN)',
    1,
    {firstname:'Frances',lastname:'DU TOUT'}
);
```

Simple Primary Key

For a table with a simple primary key, Cassandra uses one column name as the partition key. The primary key consists of only the partition key in this case. Data stored with a simple primary key will be fast to insert and retrieve if many values for the column can distribute the partitions across many nodes.

Often, your first venture into using Cassandra involves tables with simple primary keys. Keep in mind that only the primary key can be specified when retrieving data from the table. If an application needs a simple lookup table using a single unique identifier, then a simple primary key is the right choice. The table shown uses **id** as the primary key.

id	firstname	lastname
e7ae5cf3-d358-4d99-b900-85902fda9bb0	Alex	FRAME
5b6962dd-3f90-4c93-8f61-eabfa4a803e2	Marianne	VOS
220844bf-4860-49d6-9a4b-6b5d3a79cbfb	Paolo	TIRALONGO
6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47	Steven	KRUIKSWIJK
e7cd5752-bc0d-4157-a80f-7523add8dbcd	Anna	VAN DER BREGGEN

If you have simple retrieval needs, [use a simple primary key](#).

Using a simple primary key

Use a simple primary key to create a single column that you can use to query and return results. This example creates a **cyclist_name** table storing an ID number and a cyclist's first and last names in columns. The table uses a UUID as a **primary key**. This table can be queried to discover the name of a cyclist given their ID number.

A simple primary key table can be created in three different ways, as shown.

Procedure

- Create the table **cyclist_name** in the **cycling** keyspace, making **id** the primary key.

Insert the **PRIMARY KEY** keywords after the column name in the **CREATE TABLE** definition. Before creating the table, set the keyspace with a **USE** statement.

```
cqlsh> USE cycling;
CREATE TABLE cyclist_name ( id UUID PRIMARY KEY, lastname text,
    firstname text );
```

- This same example can be written with the primary key identified at the end of the table definition. Insert the **PRIMARY KEY** keywords after the last column definition in the **CREATE TABLE** definition, followed by the column name of the key. The column name is enclosed in parentheses.

```
cqlsh> USE cycling;
CREATE TABLE cyclist_name ( id UUID, lastname text, firstname text,
    PRIMARY KEY (id) );
```

- The keyspace name can be used to identify the keyspace in the **CREATE TABLE** statement instead of the **USE** statement.

```
cqlsh> CREATE TABLE cycling.cyclist_name ( id UUID, lastname text,
    firstname text, PRIMARY KEY (id) );
```

Composite Partition Key

For a table with a composite partition key, Cassandra uses multiple columns as the partition key. These columns form logical sets inside a partition to facilitate retrieval. In contrast to a simple partition key, a composite partition key uses two or more columns to identify where data will reside. Composite partition keys are used when the data stored is too large to reside in a single partition. Using more than one column for the partition key breaks the data into chunks, or buckets. The data is still grouped, but in smaller chunks. This method can be effective if a Cassandra cluster experiences hotspotting, or congestion in writing data to one node repeatedly, because a partition is heavily writing. Cassandra is often used for time series data, and hotspotting can be a real issue. Breaking incoming data into buckets by year:month:day:hour, using four columns to route to a partition can decrease hotspots.

Data is retrieved using the partition key. Keep in mind that to retrieve data from the table, values for all columns defined in the partition key have to be supplied. The table shown

Using CQL

uses **race_year** and **race_name** in the primary key, as a composition partition key. To retrieve data, both parameters must be identified.

race_year	race_name	rank	cyclist_name
2014	4th Tour of Beijing	1	Phillippe GILBERT
2014	4th Tour of Beijing	2	Daniel MARTIN
2014	4th Tour of Beijing	3	Johan Esteban CHAVES
2015	Giro d'Italia - Stage 11 - Forli > Imola	1	Ilnur ZAKARIN
2015	Giro d'Italia - Stage 11 - Forli > Imola	2	Carlos BETANCUR
2015	Tour of Japan - Stage 4 - Minami > Shinshu	1	Benjamin PRADES
2015	Tour of Japan - Stage 4 - Minami > Shinshu	2	Adam PHELAN
2015	Tour of Japan - Stage 4 - Minami > Shinshu	3	Thomas LEBAS

Cassandra stores an entire row of data on a node by partition key.

Using a composite partition key

Use a composite partition key in your primary key to create a set of columns that you can use to distribute data across multiple partitions and to query and return sorted results. This example creates a **rank_by_year_and_name** table storing the ranking and name of cyclists who competed in races. The table uses **race_year** and **race_name** as the columns defining the composition partition key of the [primary key](#). The query discovers the ranking of cyclists who competed in races by supplying year and race name values.

A composite partition key table can be created in two different ways, as shown.

Procedure

- Create the table **rank_by_year_and_name** in the **cycling** keyspace. Use **race_year** and **race_name** for the composite partition key. The table definition shown has an additional column **rank** used in the primary key. Before creating the table, set the keyspace with a **USE** statement. This example identifies the primary key at the end of the table definition. Note the double parentheses around the first two columns defined in the **PRIMARY KEY**.

```
cqlsh> USE cycling;
CREATE TABLE rank_by_year_and_name (
    race_year int,
    race_name text,
    cyclist_name text,
    rank int,
    PRIMARY KEY ((race_year, race_name), rank)
);
```

- The keyspace name can be used to identify the keyspace in the **CREATE TABLE** statement instead of the **USE** statement.

```
cqlsh> CREATE TABLE cycling.rank_by_year_and_name (
    race_year int,
    race_name text,
    cyclist_name text,
```

```
rank int,
PRIMARY KEY ((race_year, race_name), rank)
);
```

Compound Primary Key

For a table with a compound primary key, Cassandra uses a partition key that is either simple or composite. In addition, clustering column(s) are defined. [Clustering](#) is a storage engine process that sorts data within each partition based on the definition of the clustering columns. Normally, columns are sorted in ascending alphabetical order. Generally, a different grouping of data will benefit reads and writes better than this simplistic choice.

Remember that data is distributed throughout the Cassandra cluster. An application can experience high latency while retrieving data from a large partition if the entire partition must be read to gather a small amount of data. 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. Grouping data in tables using clustering columns is the equivalent of [JOINS](#) in a relational database, but are much more performant because only one table is accessed. This table uses **category** as the partition key and **points** as the clustering column. Notice that for each **category**, the **points** are ordered in descending order.

category	points	id	lastname
One-day-races	367	220844bf-4860-49d6-9a4b-6b5d3a79cbfb	TIRALONGO
One-day-races	198	6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47	KRUIJSWIJK
Time-trial	182	220844bf-4860-49d6-9a4b-6b5d3a79cbfb	TIRALONGO
Time-trial	3	6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47	KRUIJSWIJK
Sprint	39	6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47	KRUIJSWIJK
Sprint	0	220844bf-4860-49d6-9a4b-6b5d3a79cbfb	TIRALONGO
GC	1324	6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47	KRUIJSWIJK
GC	1269	220844bf-4860-49d6-9a4b-6b5d3a79cbfb	TIRALONGO

Cassandra stores an entire row of data on a node by partition key and can order the data for retrieval with clustering columns. Retrieving data from a partition is more versatile with clustering columns. For the example shown, a [query](#) could retrieve all point values greater than 200 for the **One-day-races**. If you have more complex needs for querying, [use a compound primary key](#).

Using a compound primary key

Use a compound primary key to create multiple columns that you can use to query and return sorted results. If our pro cycling example was designed in a relational database, you would create a cyclists table with a foreign key to the races. In Cassandra, you denormalize the data because joins are not performant in a distributed system. Later, other schema are shown that improve Cassandra performance. Collections and indexes are two data modeling methods. This example creates a **cyclist_category** table storing a cyclist's **last name**, **ID**, and **points** for each type of race **category**. The table uses **category** for the partition key and **points** for a single clustering column. This table can be queried to retrieve a list of cyclists and their **points** in a category, sorted by **points**.

Using CQL

A compound primary key table can be created in two different ways, as shown.

Procedure

- To create a table having a compound primary key, use two or more columns as the primary key. This example uses an additional clause `WITH CLUSTERING ORDER BY` to order the points in descending order. Ascending order is more efficient to store, but descending queries are faster due to the nature of the storage engine.

```
cqlsh> USE cycling;
CREATE TABLE cyclist_category (
    category text,
    points int,
    id UUID,
    lastname text,
    PRIMARY KEY (category, points)
) WITH CLUSTERING ORDER BY (points DESC);
```

Note: The combination of the **category** and **points** uniquely identifies a row in the **cyclist_category** table. More than one row with the same **category** can exist as long as the rows contain different **points** values.

- The keyspace name can be used to identify the keyspace in the `CREATE TABLE` statement instead of the `USE` statement.

```
cqlsh> CREATE TABLE cycling.cyclist_category (
    category text,
    points int,
    id UUID,
    lastname text,
    PRIMARY KEY (category, points)
) WITH CLUSTERING ORDER BY (points DESC);
```

Note: In both of these examples, **points** is defined as a clustering column. In Cassandra3.0 and earlier, you cannot insert any value larger than 64K bytes into a clustering column.

Creating a counter table

A counter is a special column used to store an integer that is changed in increments.

Counters are useful for many data models. Some examples:

- To keep track of the number of web page views received on a company website
- To keep track of the number of games played online or the number of players who have joined an online game

The table shown below uses **id** as the primary key and keeps track of the popularity of a cyclist based on thumbs up/thumbs down clicks in the **popularity** field of a counter table.

<code>id</code>	<code>popularity</code>
<code>6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47</code>	62

Tracking count in a distributed database presents an interesting challenge. In Cassandra, at any given moment, the counter value may be stored in the Memtable, commit log, and/or one or more SSTables. Replication between nodes can cause consistency issues in certain edge cases. Cassandra counters were redesigned in Cassandra 2.1 to alleviate some of the difficulties. Read "["What's New in Cassandra 2.1: Better Implementation of Counters"](#)" to discover the improvements made in the counters.

Because counters are implemented differently from other columns, counter columns can only be created in dedicated tables. A counter column must have the datatype [counter data type](#). This data type cannot be assigned to a column that serves as the primary key or partition key. To implement a counter column, create a table that only includes:

- The primary key (can be one or more columns)
- The counter column

Many [counter-related settings](#) can be set in the `cassandra.yaml` file.

A counter column cannot be indexed or deleted.. To load data into a counter column, or to increase or decrease the value of the counter, use the `UPDATE` command. Cassandra rejects `USING TIMESTAMP` or `USING TTL` when updating a counter column.

To create a table having one or more counter columns:

- Use `CREATE TABLE` to define the counter and non-counter columns. Use all non-counter columns as part of the `PRIMARY KEY` definition.

Using a counter

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

Procedure

- Create a table for the counter column.

```
cqlsh> USE cycling;
CREATE TABLE popular_count (
    id UUID PRIMARY KEY,
    popularity counter
```

Using CQL

```
) ;
```

- Loading data into a counter column is different than other tables. The data is updated rather than inserted.

```
UPDATE cycling.popular_count  
    SET popularity = popularity + 1  
    WHERE id = 6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47;
```

- Take a look at the counter value and note that **popularity** has a value of 1.

```
SELECT * FROM cycling.popular_count;
```

- Additional increments or decrements will change the value of the counter column.

Create table with COMPACT STORAGE

Use WITH COMPACT STORAGE to create a table that is compatible with clients written to work with 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 WITH COMPACT STORAGE directive prevents you from defining more than one column that is not part of a compound primary key. A compact table with 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.

Collections and static columns cannot be used with COMPACT STORAGE tables.

Migrating from compact storage

Before upgrading to a version of DataStax Enterprise that does not support COMPACT STORAGE, remove Thrift compatibility mode from the table. Migrate to the CQL-compatible format using the ALTER TABLE DROP COMPACT STORAGE option.

Migration changes the CQL table schema to expose any Thrift written data that was previously hidden from CQL according to the following rules:

- COMPACT STORAGE table that has no clustering columns:

- Two new columns `column1 text` and `value blob` are added. These columns contain any data written outside the CQL table schema to the Thrift table.
 - `column1` becomes a clustering column.
 - All regular columns become static columns.
- COMPACT STORAGE table with one or more clustering columns that has no regular columns:
 - Column named `value` with type `empty` is added.
- Thrift-created SuperColumn table exposes a compact value map with an empty name.
- Thrift-created Compact Tables column data types correspond to the Thrift definition.

Restriction: Removing Thrift compatibility from a table that also has a search index disables HTTP writes and deletes-by-ID on the search index.

Use the following syntax to change the table storage type:

```
ALTER TABLE keyspace_name.table_name
DROP COMPACT STORAGE;
```

Example

Simple partition key table

The following table has only a single primary key column:

```
CREATE TABLE cycling.cyclist_alt_stats (
    id UUID PRIMARY KEY,
    lastname text,
    birthday date,
    nationality text,
    weight float,
    w_units text,
    height float,
    first_race date,
    last_race date)
WITH COMPACT STORAGE;
```

Migrate to a standard CQL table:

```
ALTER TABLE cycling.cyclist_alt_stats
```

Using CQL

```
DROP COMPACT STORAGE;
```

Show the updated table schema:

```
DESC TABLE cycling.cyclist_alt_stats ;
```

Two columns were added, `column1` and `value`. `column1` was added to the PRIMARY KEY as a clustering column. And all the *regular* columns are changed to static columns.

```
CREATE TABLE cycling.cyclist_alt_stats (
    id uuid,
    column1 text,
    birthday date static,
    first_race date static,
    height float static,
    last_race date static,
    lastname text static,
    nationality text static,
    value blob,
    w_units text static,
    weight float static,
    PRIMARY KEY (id, column1)
) WITH CLUSTERING ORDER BY (column1 ASC)
```

Table schema collision fix

Dynamic schema creation or updates can cause schema collision resulting in errors.

Procedure

1. Run a rolling restart on all nodes to ensure schema matches. Run `nodetool describecluster` on all nodes. Check that there is only one schema version.
2. On each node, check the `data` directory, looking for two directories for the table in question. If there is only one directory, go on to the next node. If there are two or more directories, the old table directory before update and a new table directory for after the update, continue.
3. Identify which `cf_id` (column family ID) is the newest table ID in `system.schema_columnfamilies`. The column family ID is fifth column in the results.

```
$ cqlsh -e "SELECT * FROM system.schema_column_families" |grep <tablename>
```

4. Move the data from the older table to the newer table's directory and remove the old directory. Repeat this step as necessary.

5. Run `nodetool refresh`.

Creating advanced data types in tables

Data can be inserted into tables using more advanced types of data.

Creating collections

Cassandra provides collection types as a way to group and store data together in a column. For example, in a relational database a grouping such as a user's multiple email addresses is related with a many-to-one joined relationship between a user table and an email table. Cassandra avoids joins between two tables by storing the user's email addresses in a collection column in the user table. Each collection specifies the data type of the data held.

A collection is appropriate if the data for collection storage is limited. If the data has unbounded growth potential, like messages sent or sensor events registered every second, do not use collections. Instead, use a table with a [compound primary key](#) where data is stored in the clustering columns.

CQL contains these collection types:

- [set](#)
- [list](#)
- [map](#)

Observe the following limitations of collections:

- Never insert more than 2 billion items in a collection, as only that number can be queried.
- The maximum number of keys for a `map` collection is 65,535.
- The maximum size of an item in a `list` or a `map` collection is 2GB.
- The maximum size of an item in a `set` collection is 65,535 bytes.
- Keep collections small to prevent delays during querying.

Collections cannot be "sliced"; Cassandra reads a collection in its entirety, impacting performance. Thus, collections should be much smaller than the maximum limits listed. The collection is not paged internally.

- Lists can incur a read-before-write operation for some insertions. Sets are preferred over lists whenever possible.

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

Using CQL

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

Also see [Using frozen in a collection](#).

Creating the set type

A **set** consists of a group of elements with unique values. Duplicate values will not be stored distinctly. The values of a **set** are stored unordered, but will return the elements in sorted order when queried. Use the **set** data type to store data that has a many-to-one relationship with another column. For example, in the example below, a **set** called **teams** stores all the teams that a cyclist has been a member of during their career.

Procedure

Define **teams** in a table **cyclist_career_teams**. Each team listed in the **set** will have a **text** data type.

```
cqlsh> CREATE TABLE cycling.cyclist_career_teams ( id UUID PRIMARY KEY,
    lastname text, teams set<text> );
```

lastname	teams
ARMITSTEAD	{'AA Drink - Leontien.nl', 'Boels-Dolmans Cycling Team', 'Team Garmin - Cervelo'}
VOS	{'Nederland bloeit', 'Rabobank Women Team', 'Rabobank-Liv Giant', 'Rabobank-Liv Woman Cycling Team'}
BRAND	{'AA Drink - Leontien.nl', 'Leontien.nl', 'Rabobank-Liv Giant', 'Rabobank-Liv Woman Cycling Team'}
VAN DER BREGGEN	{'Rabobank-Liv Woman Cycling Team', 'Sengers Ladies Cycling Team', 'Team Flexpoint'}

Creating the list type

A **list** has a form much like a **set**, in that a **list** groups and stores values. Unlike a **set**, the values stored in a **list** do not need to be unique and can be duplicated. In addition, a **list** stores the elements in a particular order and may be inserted or retrieved according to an index value.

Use the **list** data type to store data that has a possible many-to-many relationship with another column. For example, in the example below, a **list** called **events** stores all the race events on an upcoming calendar. Each month/year pairing might have several events occurring, and the races are stored in a **list**. The **list** can be ordered so that the races appear in the order that they will take place, rather than alphabetical order.

Procedure

Define **events** in a table **upcoming_calendar**. Each event listed in the **list** will have a **text** data type.

```
cqlsh> CREATE TABLE cycling.upcoming_calendar ( year int, month int,
    events list<text>, PRIMARY KEY ( year, month ) );
```

year	month	events
2015	6	['Criterium du Dauphine', 'Tour de Suisse']
2015	7	['Tour de France']

Creating the map type

A map relates one item to another with a key-value pair. For each key, only one value may exist, and duplicates cannot be stored. Both the key and the value are designated with a data type.

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

Define **teams** in a table **cyclist_teams**. Each team listed in the **map** will have an **integer** data type for the **year** a cyclist belonged to the team and a **text** data type for the **team name**. The map collection is specified with a map column name and the pair of data types enclosed in angle brackets.

```
cqlsh> CREATE TABLE cycling.cyclist_teams ( id UUID PRIMARY KEY, lastname text, firstname text, teams map<int,text> );
```

lastname	firstname	teams
ARMITSTEAD	Elizabeth	{2011: 'Team Garmin - Cervelo', 2012: 'AA Drink - Leontien.nl', 2013: 'Boels-Dolmans Cycling Team', 2014: 'Boels-Dolmans Cycling Team', 2015: 'Boels-Dolmans Cycling Team'}
VOS	Marianne	{2011: 'Nederland bloeit', 2012: 'Rabobank Women Team', 2013: 'Rabobank-Liv Giant', 2014: 'Rabobank-Liv Woman Cycling Team', 2015: 'Rabobank-Liv Woman Cycling Team'}
VAN DER BREGGEN	Anna	{2009: 'Team Flexpoint', 2012: 'Sengers Ladies Cycling Team', 2013: 'Sengers Ladies Cycling Team', 2014: 'Rabobank-Liv Woman Cycling Team', 2015: 'Rabobank-Liv Woman Cycling Team'}

Creating a table with a tuple

Tuples are a data type that allow two or more values to be stored together in a column. A user-defined type can be used, but for simple groupings, a **tuple** is a good choice.

Using CQL

Procedure

- Create a table **cycling.route** using a **tuple** to store each waypoint location name, latitude, and longitude.

```
cqlsh> CREATE TABLE cycling.route (race_id int, race_name text,  
    point_id int, lat_long tuple<text, tuple<float,float>>, PRIMARY KEY  
(race_id, point_id));
```

- Create a table **cycling.nation_rank** using a **tuple** to store the rank, cyclist name, and points total for a cyclist and the country name as the primary key.

```
CREATE TABLE cycling.nation_rank ( nation text PRIMARY KEY, info  
tuple<int,text,int> );
```

- The table **cycling.nation_rank** is keyed to the country as the primary key. It is possible to store the same data keyed to the rank. Create a table **cycling.popular** using a **tuple** to store the country name, cyclist name and points total for a cyclist and the rank as the primary key.

```
CREATE TABLE cycling.popular (rank int PRIMARY KEY, cinfo  
tuple<text,text,int> );
```

Creating a user-defined type (UDT)

User-defined types (UDTs) can attach multiple data fields, each named and typed, to a single column. The fields used to create a UDT may be any valid data type, including collections and other existing UDTs. Once created, UDTs may be used to define a column in a table.

Procedure

- Use the **cycling** keyspace.

```
cqlsh> USE cycling;
```

- Create a user-defined type named **basic_info**.

```
cqlsh> CREATE TYPE cycling.basic_info (br  
    birthday timestamp,  
    nationality text,  
    weight text,  
    height text  
);
```

- Create a table for storing cyclist data in columns of type **basic_info**. Use the **frozen** keyword in the definition of the user-defined type column.

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

```
cqlsh> CREATE TABLE cycling.cyclist_stats ( id uuid PRIMARY KEY,
    lastname text, basics FROZEN<basic_info>);
```

- A user-defined type can be nested in another column type. This example nests a **UDT** in a **list**.

```
CREATE TYPE cycling.race (race_title text, race_date timestamp,
    race_time text);
CREATE TABLE cycling.cyclist_races ( id UUID PRIMARY KEY, lastname
    text, firstname text, races list<FROZEN <race>> );
```

Creating functions

Users can create user-defined functions (UDFs) and user-defined aggregate functions (UDAs). Functions are used to manipulate stored data in queries. Retrieving [results using standard aggregate functions](#) are also available for queries.

Creating user-defined function (UDF)

Allows users to define functions that can be applied to data stored in a table as part of a query result. The function must be created prior to its use in a SELECT statement. The function will be performed on each row of the table. To use user-defined functions with Java or Javascript, `enable_user_defined_functions` must be set `true` in the **cassandra.yaml** file setting to enable the functions. User-defined functions are defined within a keyspace; if no keyspace is defined, the current keyspace is used. User-defined functions are executed in a sandbox

By default, Cassandra supports defining functions in `java` and `javascript`. Other scripting languages, such as `Python`, `Ruby`, and `Scala` can be added by adding a JAR to the classpath. Install the JAR file into `$CASSANDRA_HOME/lib/jsr223/[language]/[jar-name].jar` where language is '`jruby`', '`jpython`', or '`scala`'

Procedure

- Create a function, specifying the data type of the returned value, the language, and the actual code of the function to be performed. The following function, `fLog()`, computes the logarithmic value of each input. It is a built-in `java` function and used

to generate linear plots of non-linear data. For this example, it presents a simple math function to show the capabilities of user-defined functions.

```
cqlsh> CREATE OR REPLACE FUNCTION fLog (input double) CALLED  
ON NULL INPUT RETURNS double LANGUAGE java AS 'return  
Double.valueOf(Math.log(input.doubleValue()));';
```

Note:

- CALLED ON NULL INPUT ensures the function will always be executed.
- RETURNS NULL ON NULL INPUT ensures the function will always return NULL if any of the input arguments is NULL.
- RETURNS defines the data type of the value returned by the function.
- A function can be replaced with a different function if OR REPLACE is used as shown in the example above. Optionally, the IF NOT EXISTS keywords can be used to create the function only if another function with the same signature does not exist in the keyspace. OR REPLACE and IF NOT EXISTS cannot be used in the same command.

```
CREATE FUNCTION IF NOT EXISTS fLog (input double)  
CALLED ON NULL INPUT  
RETURNS double  
LANGUAGE java AS '  
    return Double.valueOf(Math.log(input.doubleValue()));  
';
```

Creating User-Defined Aggregate Function (UDA)

Allows users to define aggregate functions that can be applied to data stored in a table as part of a query result. The aggregate function must be created prior to its use in a SELECT statement and the query must only include the aggregate function itself, but no columns. The state function is called once for each row, and the value returned by the state function becomes the new state. After all rows are processed, the optional final function is executed with the last state value as its argument. Aggregation is performed by the coordinator.

The example shown computes the team average for race time for all the cyclists stored in the table. The race time is computed in seconds.

Procedure

- Create a state function, as a [user-defined function \(UDF\)](#), if needed. This function adds all the race times together and counts the number of entries.

```
cqlsh> CREATE OR REPLACE FUNCTION avgState ( state  
tuple<int,bigint>, val int ) CALLED ON NULL INPUT RETURNS  
tuple<int,bigint> LANGUAGE java AS
```

```
'if (val !=null) { state.setInt(0, state.getInt(0)+1);
state.setLong(1, state.getLong(1)+val.intValue()); } return
state;';
```

- Create a final function, as a [user-defined function \(UDF\)](#), if needed. This function computes the average of the values passed to it from the state function.

```
cqlsh> CREATE OR REPLACE FUNCTION avgFinal ( state
tuple<int,bigint> ) CALLED ON NULL INPUT RETURNS double LANGUAGE
java AS
'double r = 0; if (state.getInt(0) == 0) return null; r =
state.getLong(1); r/= state.getInt(0); return Double.valueOf(r);';
```

- Create the aggregate function using these two functions, and add an STYPE to define the data type for the function. Different STYPES will distinguish one function from another with the same name. An aggregate can be replaced with a different aggregate if OR REPLACE is used as shown in the examples above. Optionally, the IF NOT EXISTS keywords can be used to create the aggregate only if another aggregate with the same signature does not exist in the keyspace. OR REPLACE and IF NOT EXISTS cannot be used in the same command.

```
cqlsh> CREATE AGGREGATE IF NOT EXISTS average ( int )
SFUNC avgState STYPE tuple<int,bigint> FINALFUNC avgFinal INITCOND
(0,0);
```

What's next:

What's next

For more information on user-defined aggregates, see [Cassandra Aggregates - min, max, avg, group by](#) and [A few more Cassandra aggregates](#).

Inserting and updating data

Data can be inserted into tables using the INSERT command using the regular syntax or in JSON format.

Inserting simple data into a table

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

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

Using CQL

Procedure

- To insert simple data into the table **cycling.cyclist_name**, use the `INSERT` command. This example inserts a single record into the table.

```
cqlsh> INSERT INTO cycling.cyclist_name (id, lastname, firstname)
VALUES (5b6962dd-3f90-4c93-8f61-eabfa4a803e2, 'VOS', 'Marianne');
```

- You can insert complex string constants using double dollar signs to enclose a string with quotes, backslashes, or other characters that would normally need to be escaped.

```
cqlsh> INSERT INTO cycling.calendar (race_id, race_start_date,
race_end_date, race_name) VALUES
(201, '2015-02-18', '2015-02-22', $$Women's Tour of New
Zealand$$);
```

Inserting and updating data into a set

If a table specifies a **set** to hold data, then either `INSERT` or `UPDATE` is used to enter data.

Procedure

- Insert data into the **set**, enclosing values in curly brackets.
Set values must be unique, because no order is defined in a **set** internally.

```
cqlsh> INSERT INTO cycling.cyclist_career_teams (id, lastname, teams)
VALUES (5b6962dd-3f90-4c93-8f61-eabfa4a803e2, 'VOS',
{ 'Rabobank-Liv Woman Cycling Team', 'Rabobank-Liv Giant', 'Rabobank
Women Team', 'Nederland bloeit' } );
```

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

```
cqlsh> UPDATE cycling.cyclist_career_teams
SET teams = teams + {'Team DSB - Ballast Nedam'} WHERE id =
5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
```

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

```
cqlsh> UPDATE cycling.cyclist_career_teams
SET teams = teams - {'WOMBATS - Womens Mountain Bike & Tea
Society'} WHERE id = 5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
```

- 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 because an empty set, list, or map is stored as a null set.

```
cqlsh> UPDATE cyclist.cyclist_career_teams SET teams = {} WHERE id =
5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
```

```
DELETE teams FROM cycling.cyclist_career_teams WHERE id =
5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
```

A query for the **teams** returns null.

```
cqlsh> SELECT id, teams FROM users WHERE id =
5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
```

lastname	teams
Vos	null

Inserting and updating data into a list

If a table specifies a **list** to hold data, then either `INSERT` or `UPDATE` is used to enter data.

Procedure

- Insert data into the **list**, enclosing values in square brackets.

```
INSERT INTO cycling.upcoming_calendar (year, month, events) VALUES
(2015, 06, ['Criterium du Dauphine', 'Tour de Suisse']);
```

- Use the `UPDATE` command to insert values into the **list**. Prepend an element to the **list** by enclosing it in square brackets and using the addition (+) operator.

```
cqlsh> UPDATE cycling.upcoming_calendar SET events = ['The Parx
Casino Philly Cycling Classic'] + events WHERE year = 2015 AND
month = 06;
```

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

```
cqlsh> UPDATE cycling.upcoming_calendar SET events = events + ['Tour
de France Stage 10'] WHERE year = 2015 AND month = 06;
```

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.

Using CQL

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

```
cqlsh> UPDATE cycling.upcoming_calendar SET events[2] = 'Vuelta  
Ciclista a Venezuela' WHERE year = 2015 AND month = 06;
```

To add an element at a particular position, Cassandra reads the entire list, and then rewrites the part of the list that needs to be shifted to the new index positions. Consequently, adding an element at a particular position results in greater latency than appending or prefixing an element to a list.

- Remove an element from a **list**, use the `DELETE` command and the list index position in square brackets. For example, remove the **event** just placed in the **list** in the last step.

```
cqlsh> DELETE events[2] FROM cycling.upcoming_calendar WHERE year =  
2015 AND month = 06;
```

The method of removing elements using an indexed position from a **list** requires an internal read. In addition, the client-side application could only discover the indexed position by reading the whole list and finding the values to remove, adding additional latency to the operation. If another thread or client prepends elements to the list before the operation is done, incorrect data will be removed.

- Remove all elements having a particular value using the `UPDATE` command, the subtraction operator (-), and the list value in square brackets.

```
cqlsh> UPDATE cycling.upcoming_calendar SET events = events - ['Tour  
de France Stage 10'] WHERE year = 2015 AND month = 06;
```

Using the `UPDATE` command as shown in this example is recommended over the last example because it is safer and faster.

Inserting and updating data into a map

If a table specifies a **map** to hold data, then either `INSERT` or `UPDATE` is used to enter data.

Procedure

- Set or replace **map** data, using the `INSERT` or `UPDATE` command, and enclosing the integer and text values in a map collection with curly brackets, separated by a colon.

```
cqlsh> INSERT INTO cycling.cyclist_teams (id, lastname, firstname,  
teams)  
VALUES (  
5b6962dd-3f90-4c93-8f61-eabfa4a803e2,  
'VOS',
```

```
'Marianne',
{2015 : 'Rabobank-Liv Woman Cycling Team', 2014 : 'Rabobank-Liv
Woman Cycling Team', 2013 : 'Rabobank-Liv Giant',
2012 : 'Rabobank Women Team', 2011 : 'Nederland bloeit' } );
```

Note: Using `INSERT` in this manner will replace the entire map.

- Use the `UPDATE` command to insert values into the **map**. Append an element to the **map** by enclosing the key-value pair in curly brackets and using the addition (+) operator.

```
cqlsh> UPDATE cycling.cyclist_teams SET teams = teams
+ {2009 : 'DSB Bank - Nederland bloeit'} WHERE id =
5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
```

- Set a specific element using the `UPDATE` command, enclosing the specific key of the element, an **integer**, in square brackets, and using the equals operator to map the value assigned to the key.

```
cqlsh> UPDATE cycling.cyclist_teams SET teams[2006] = 'Team DSB -
Ballast Nedam' WHERE id = 5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
```

- Delete an element from the map using the `DELETE` command and enclosing the specific key of the element in square brackets:

```
cqlsh> DELETE teams[2009] FROM cycling.cyclist_teams WHERE
id=e7cd5752-bc0d-4157-a80f-7523add8dbcd;
```

- Alternatively, remove all elements having a particular value using the `UPDATE` command, the subtraction operator (-), and the map key values in curly brackets.

```
cqlsh> UPDATE cycling.cyclist_teams SET teams = teams -
{'2013','2014'} WHERE id=e7cd5752-bc0d-4157-a80f-7523add8dbcd;
```

Inserting tuple data into a table

Tuples are used to group small amounts of data together that are then stored in a column.

Procedure

- Insert data into the table **cycling.route** which has **tuple** data. The **tuple** is enclosed in parentheses. This **tuple** has a **tuple** nested inside; nested parentheses are required for the inner **tuple**, then the outer **tuple**.

```
cqlsh> INSERT INTO cycling.route (race_id, race_name, point_id, lat_long) VALUES (500, '47th Tour du Pays de Vaud', 2, ('Champagne', (46.833, 6.65)));
```

- Insert data into the table **cycling.nation_rank** which has **tuple** data. The **tuple** is enclosed in parentheses. The **tuple** called **info** stores the rank, name, and point total of each cyclist.

```
cqlsh> INSERT INTO cycling.nation_rank (nation, info) VALUES ('Spain', (1,'Alejandro VALVERDE' , 9054));
```

- Insert data into the table **popular** which has **tuple** data. The **tuple** called **cinfo** stores the country name, cyclist name, and points total.

```
cqlsh> INSERT INTO cycling.popular (rank, cinfo) VALUES (4, ('Italy', 'Fabio ARU', 163));
```

Inserting or updating data into a user-defined type (UDT)

If a table specifies a **user-defined type (UDT)** to hold data, then either **INSERT** or **UPDATE** is used to enter data.

Procedure

Inserting data into a UDT

- Set or replace **user-defined type** data, using the **INSERT** or **UPDATE** command, and enclosing the user-defined type with curly brackets, separating each key-value pair in the **user-defined type** by a colon.

```
cqlsh> INSERT INTO cycling.cyclist_stats (id, lastname, basics)
VALUES (
    e7ae5cf3-d358-4d99-b900-85902fda9bb0,
    'FRAME',
    { birthday : '1993-06-18', nationality : 'New Zealand', weight :
        null, height : null }
);
```

Note: Note the inclusion of **null** values for **UDT** elements that have no value. A value, whether null or otherwise, must be included for each element of the **UDT**.

- Data can be inserted into a **UDT** that is nested in another column type. For example, a **list of races**, where the race name, date, and time are defined in a **UDT** has elements enclosed in curly brackets that are in turn enclosed in square brackets.

```
cqlsh> INSERT INTO cycling.cyclist_races (id, lastname, firstname, races) VALUES (
  5b6962dd-3f90-4c93-8f61-eabfa4a803e2,
  'VOS',
  'Marianne',
  [ { race_title : 'Rabobank 7-Dorpenomloop Aalburg', race_date : '2015-05-09', race_time : '02:58:33' },
    { race_title : 'Ronde van Gelderland', race_date : '2015-04-19', race_time : '03:22:23' } ]
);
```

Note: The **UDT** nested in the **list** is **frozen**, so the entire **list** will be read when querying the table.

Inserting JSON data into a table

In a production database, inserting columns and column values programmatically is more practical than using cqlsh, but often, testing queries using this SQL-like shell is very convenient. For JSON data all values will be inserted as a string if they are not a number, but will be stored using the column data type. For example, the **id** below is inserted as a string, but is stored as a **UUID**. For more information, see [What's New in Cassandra 2.2: JSON Support](#).

Procedure

- To insert JSON data, add **JSON** to the **INSERT** command.. Note the absence of the keyword **VALUES** and the list of columns that is present in other **INSERT** commands.

```
cqlsh> INSERT INTO cycling.cyclist_category JSON '{
  "category" : "GC",
  "points" : 780,
  "id" : "829aa84a-4bba-411f-a4fb-38167a987cda",
  "lastname" : "SUTHERLAND" }';
```

- A null value will be entered if a defined column like **lastname**, is not inserted into a table using JSON format.

```
cqlsh> INSERT INTO cycling.cyclist_category JSON '{
  "category" : "Sprint",
  "points" : 700,
  "id" : "829aa84a-4bba-411f-a4fb-38167a987cda"
```

Using CQL

```
};';
```

category	points	id	lastname
Sprint	700	829aa84a-4bba-411f-a4fb-38167a987cda	null
GC	1269	220844bf-4860-49d6-9a4b-6b5d3a79cbfb	TIRALONGO
GC	780	829aa84a-4bba-411f-a4fb-38167a987cda	SUTHERLAND

Using lightweight transactions

[INSERT](#) and [UPDATE](#) statements using the `IF` clause support lightweight transactions, also known as Compare and Set (CAS). A common use for lightweight transactions is an insertion operation that must be unique, such as a cyclist's identification. [Lightweight transactions](#) should not be used casually, as the latency of operations increases fourfold due to the due to the round-trips necessary between the CAS coordinators.

Cassandra supports non-equal conditions for lightweight transactions. You can use `<`, `<=`, `>`, `>=`, `!=` and `IN` operators in `WHERE` clauses to query lightweight tables.

It is important to note that using `IF NOT EXISTS` on an [INSERT](#), the timestamp will be designated by the lightweight transaction, and [USING TIMESTAMP](#) is prohibited.

Procedure

- Insert a new cyclist with their `id`.

```
cqlsh> INSERT INTO cycling.cyclist_name (id, lastname, firstname)
VALUES (4647f6d3-7bd2-4085-8d6c-1229351b5498, 'KNETEMANN',
'Roxxane')
IF NOT EXISTS;
```

- 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 Roxane Knetemann's `firstname` because of a spelling error.

```
cqlsh> UPDATE cycling.cyclist_name
SET firstname = 'Roxane'
WHERE id = 4647f6d3-7bd2-4085-8d6c-1229351b5498
IF firstname = 'Roxxane';
```

Sharing a static column

In a table that uses clustering columns, non-clustering columns can be declared static in the table definition. Static columns are only static within a given partition.

```
CREATE TABLE t (
  k text,
  s text STATIC,
  i int,
  PRIMARY KEY (k, i)
```

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

Output is:

k	s	i
k	"I'm still shared"	0
k	"I'm still shared"	1

Restriction:

- 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](#).

You can use the DISTINCT keyword to select static columns. In this case, Cassandra retrieves only the beginning (static column) of the partition.

Epiring data with time-to-live

Columns and tables support an optional expiration period called TTL (time-to-live); TTL is not supported on counter columns. Define the TTL value in seconds. Data expires once it exceeds the TTL period and is then marked with a [tombstone](#). Expired data continues to be available for read requests during the grace period, see [gc_grace_seconds](#). Normal compaction and repair processes automatically remove the tombstone data.

Note:

- TTL precision is one second, which is calculated by the coordinator node. When using TTL, ensure that all nodes in the cluster have synchronized clocks.
- A very short TTL is not very useful.
- Expiring data uses additional 8 bytes of memory and disk space to record the TTL and grace period.

Setting a TTL for a specific column

Use CQL to [set the TTL](#).

Using CQL

To change the TTL of a specific column, you must re-insert the data with a new TTL. Cassandra upserts the column with the new TTL.

To remove TTL from a column, set TTL to zero. For details, see the [UPDATE](#) documentation.

Setting a TTL for a table

Use [CREATE TABLE](#) or [ALTER TABLE](#) to define the `default_time_to_live` property for all columns in a table. If any column exceeds TTL, the entire table is tombstoned.

For details and examples, see [Expiring data with TTL example](#).

Expiring data with TTL example

Both the `INSERT` and `UPDATE` commands support setting a time for data in a column to expire. Use CQL to set the expiration time (**TTL**).

Procedure

- Use the `INSERT` command to set a calendar listing in the **calendar** table to expire in 86400 seconds (one day).

```
INSERT INTO cycling.calendar (race_id, race_name, race_start_date, race_end_date) V...
```

- Extend the expiration period to three days (259200 seconds) by using the `UPDATE` command with the `USING TTL` keyword. Also set the race name.

```
UPDATE cycling.calendar USING TTL 259200  
SET race_name = 'Tour de France - Stage 12'  
WHERE race_id = 200 AND race_start_date = '2015-05-27' AND race_end_date = '2015-...
```

- Delete a column's existing TTL by setting its value to zero.

```
UPDATE cycling.calendar USING TTL 0  
SET race_name = 'Tour de France - Stage 12'  
WHERE race_id = 200 AND race_start_date = '2015-05-27' AND race_end_date = '2015-...
```

You can set a default TTL for an entire table by setting the table's `default_time_to_live` property. Setting TTL on a column using the `INSERT` or `UPDATE` command overrides the table TTL.

Inserting data using COPY and a CSV file

In a production database, inserting columns and column values programmatically is more practical than using `cqlsh`, but often, testing queries using this SQL-like shell is very convenient. A comma-delimited file, or CSV file, is useful if several records need inserting. While not strictly an `INSERT` command, it is a common method for inserting data.

Procedure

1. Locate your CSV file and [check options to use](#).

```
category|point|id|lastname
GC|1269|2003|TIRALONGO
One-day-races|367|2003|TIRALONGO
GC|1324|2004|KRUIJSWIJK
```

2. To insert the data, using the `COPY` command with CSV data.

```
$ COPY cycling.cyclist_catgory FROM 'cyclist_category.csv' WITH
  DELIMITER='|' AND HEADER=TRUE
```

Batching data insertion and updates

Batching is used to insert or update data in tables. Understanding the use of batching, if used, is crucial to performance.

Batching inserts, updates and deletes

Batch operations for both single partition and multiple partitions ensure atomicity. An atomic transaction is an indivisible and irreducible series of operations such that either all occur, or nothing occurs. Single partition batch operations are atomic automatically, while multiple partition batch operations require the use of a batchlog to ensure atomicity.

Use batching if atomicity is a primary concern for a group of operations. Single partition batch operations are processed server-side as a single mutation for improved performance, provided the number of operations do not exceed the [maximum size of a single operation](#) or cause the query to time out. Multiple partition batch operations often suffer from performance issues, and should only be used if atomicity must be ensured.

Batching can be effective for single partition write operations. But batches are often mistakenly used in an attempt to optimize performance. Depending on the batch operation, the performance may actually worsen. Some batch operations place a greater burden on the coordinator node and lessen the efficiency of the data insertion. The number of partitions involved in a batch operation, and thus the potential for multi-node accessing, can increase the latency dramatically. In all batching, the coordinator node manages all write operations, so that the coordinator node can pose a bottleneck to completion.

Good reasons for batching operations in Apache Cassandra are:

- Inserts, updates or deletes to a single partition when atomicity and isolation is a requirement. Atomicity ensures that either all or nothing is written. Isolation ensures that partial insertion or updates are not accessed until all operations are complete.

Single partition batching will send one message to the coordinator for all operations. All replicas for the single partition receive the data, and the coordinator waits for

acknowledgement. No batchlog mechanism is necessary. The number of nodes involved in the batch is bounded by the number of replicas.

- Ensuring atomicity for small inserts or updates to multiple partitions when inconsistency cannot occur.

Multiple partition batching will send one message to the coordinator for all operations. The coordinator writes a batchlog that is replicated to other nodes to ensure that inconsistency will not occur if the coordinator fails. Then the coordinator must wait for all nodes with an affected partition to acknowledge the operations before removing the logged batch. The number of nodes involved in the batch is bounded by number of distinct partition keys in the logged batch plus (possibly) the batchlog replica nodes. While a batch operation for a small number of partitions may be critical for consistency, this use case is more the exception than the rule.

Poor reasons for batching operations in Apache Cassandra are:

- Inserting or updating data to multiple partitions, especially when a large number of partitions are involved.

As stated above, [batching to multiple partitions](#) has performance costs. Unlogged batch operations are possible, to avoid the additional time cost of the batchlog, but the coordinator node will still be a bottleneck the performance due to the synchronous nature. A better alternative uses asynchronous writes using driver code; the token aware loading balancing will distribute the writes to several coordinator nodes, decreasing the time to complete the insert and update operations.

Batched statements can save network round-trips between the client and the server, and possibly between the coordinator and the replicas. However, consider carefully before implementing batch operations, and decide if they are truly necessary. For information about the fastest way to load data, see "[Cassandra: Batch loading without the Batch keyword](#)".

Good use of BATCH statement

Batch operations can be beneficial. See these examples below to see a good use of BATCH. The examples below use the following table **cyclist_expenses**:

```
cqlsh> CREATE TABLE cycling.cyclist_expenses (
    cyclist_name text,
    balance float STATIC,
    expense_id int,
    amount float,
    description text,
    paid boolean,
    PRIMARY KEY (cyclist_name, expense_id)
```

```
) ;
```

Note that `balance` is `STATIC`.

Note: If there are two different tables in the same keyspace and the two tables have the same partition key, this scenario is considered a single partition batch. There will be a single mutation for each table. This happens because the two tables could have different columns, even though the keyspace and partition are the same. Batches allow a caller to bundle multiple operations into a single batch request. All the operations are performed by the same coordinator. The best use of a batch request is for a single partition in multiple tables in the same keyspace. Also, batches provide a guarantee that mutations will be applied in a particular order.

Procedure

Single partition batch

- The first `INSERT` in the `BATCH` statement sets the `balance` to zero. The next two statements insert an `expense_id` and change the `balance` value. All the `INSERT` and `UPDATE` statements in this batch write to the same partition, keeping the latency of the write operation low.

```
cqlsh> BEGIN BATCH
    INSERT INTO cycling.cyclist_expenses (cyclist_name, balance)
    VALUES ('Vera ADRIAN', 0) IF NOT EXISTS;
    INSERT INTO cycling.cyclist_expenses (cyclist_name, expense_id,
    amount, description, paid) VALUES ('Vera ADRIAN', 1, 7.95,
    'Breakfast', false);
    APPLY BATCH;
```

This batching example includes conditional updates combined with using `static columns`. Recall that single partition batches are not logged.

Note: It would be reasonable to expect that an `UPDATE` to the `balance` could be included in this `BATCH` statement:

```
cqlsh> UPDATE cycling.cyclist_expenses SET balance = -7.95 WHERE
    cyclist_name = 'Vera ADRIAN' IF balance = 0;
```

However, it is important to understand that all the statements processed in a `BATCH` statement timestamp the records with the same value. The operations may not perform in the order listed in the `BATCH` statement. The `UPDATE` might be processed BEFORE the first `INSERT` that sets the `balance` value to zero, allowing the conditional to be met.

Using CQL

An acknowledgement of a batch statement is returned if the batch operation is successful.

[**applied**]

True

The resulting table will only have one record so far.

cyclist_name	expense_id	balance	amount	description	paid
Vera ADRIAN	1	0	7.95	Breakfast	False

- The balance can be adjusted separately with an UPDATE statement. Now the `balance` will reflect that breakfast was unpaid.

```
cqlsh> UPDATE cycling.cyclist_expenses SET balance = -7.95 WHERE
cyclist_name = 'Vera ADRIAN' IF balance = 0;
```

cyclist_name	expense_id	balance	amount	description	paid
Vera ADRIAN	1	-7.95	7.95	Breakfast	False

- The table **cyclist_expenses** stores records about each purchase by a cyclist and includes the running balance of all the cyclist's purchases. Because the balance is static, all purchase records for a cyclist have the same running balance. This BATCH statement inserts expenses for two more meals changes the balance to reflect that breakfast and dinner were unpaid.

```
cqlsh> BEGIN BATCH
INSERT INTO cycling.cyclist_expenses (cyclist_name, expense_id,
amount, description, paid) VALUES ('Vera ADRIAN', 2, 13.44,
'Lunch', true);
INSERT INTO cycling.cyclist_expenses (cyclist_name, expense_id,
amount, description, paid) VALUES ('Vera ADRIAN', 3, 25.00,
'Dinner', false);
UPDATE cycling.cyclist_expenses SET balance = -32.95 WHERE
cyclist_name = 'Vera ADRIAN' IF balance = -7.95;
APPLY BATCH;
```

cyclist_name	expense_id	balance	amount	description	paid
Vera ADRIAN	1	-32.95	7.95	Breakfast	False
Vera ADRIAN	2	-32.95	13.44	Lunch	True
Vera ADRIAN	3	-32.95	25	Dinner	False

- Finally, the cyclist pays off all outstanding bills and the **balance** of the account goes to zero.

```
cqlsh> BEGIN BATCH
UPDATE cycling.cyclist_expenses SET balance = 0 WHERE cyclist_name =
  'Vera ADRIAN' IF balance = -32.95;
UPDATE cycling.cyclist_expenses SET paid = true WHERE cyclist_name =
  'Vera ADRIAN' AND expense_id = 1 IF paid = false;
UPDATE cycling.cyclist_expenses SET paid = true WHERE cyclist_name =
  'Vera ADRIAN' AND expense_id = 3 IF paid = false;
APPLY BATCH;
```

cyclist_name	expense_id	balance	amount	description	paid
Vera ADRIAN	1	0	7.95	Breakfast	True
Vera ADRIAN	2	0	13.44	Lunch	True
Vera ADRIAN	3	0	25	Dinner	True

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 possible because a batch having conditional updates cannot span multiple partitions.

Multiple partition logged batch

- A classic example for using `BATCH` for a multiple partition insert involves writing the same data to two related tables:

```
cqlsh> BEGIN LOGGED BATCH
INSERT INTO cycling.cyclist_names (cyclist_name, race_id) VALUES
  ('Vera ADRIAN', 100);
INSERT INTO cycling.cyclist_by_id (race_id, cyclist_name) VALUES
  (100, 'Vera ADRIAN');
APPLY BATCH;
```

Here, it is important that the same data is written to both tables to keep them in synchronization. Another common use for this batch operation is updating usernames and passwords.

Misuse of BATCH statement

Misused, `BATCH` statements can cause many problems in a distributed database like Cassandra. Batch operations that involve multiple nodes are a definite anti-pattern. Keep in mind which partitions data will be written to when grouping `INSERT` and `UPDATE` statements in a `BATCH` statement. Writing to several partitions might require interaction with several nodes in the cluster, causing a great deal of latency for the write operation.

Procedure

This example shows an anti-pattern since the `BATCH` statement will write to several different partitions, given the partition key `id`.

```
cqlsh> BEGIN BATCH
  INSERT INTO cycling.cyclist_name (id, lastname, firstname) VALUES
    (6d5f1663-89c0-45fc-8cf8-60a373b01622, 'HOSKINS', 'Melissa');
  INSERT INTO cycling.cyclist_name (id, lastname, firstname) VALUES
    (38ab64b6-26cc-4de9-ab28-c257cf011659, 'FERNANDES', 'Marcia');
  INSERT INTO cycling.cyclist_name (id, lastname, firstname) VALUES
    (9011d3be-d35c-4a8d-83f7-a3c543789ee7, 'NIEWIADOMA', 'Katarzyna');
  INSERT INTO cycling.cyclist_name (id, lastname, firstname) VALUES
    (95addc4c-459e-4ed7-b4b5-472f19a67995, 'ADRIAN', 'Vera');
APPLY BATCH;
```

In this example, four partitions are accessed, but consider the effect of including 100 partitions in a batch - the performance would degrade considerably.

Querying tables

Data can be queried from tables using the `SELECT` command. With Cassandra 3.0, many new options are available, such as retrieving JSON data, using standard aggregate functions, and manipulating retrieved data with user-defined functions (UDFs) and user-defined aggregate functions (UDAs).

Retrieval and sorting results

Querying tables to select data is the reason data is stored in databases. Similar to SQL, CQL can `SELECT` data using simple or complex qualifiers. At its simplest, a query selects all data in a table. At its most complex, a query delineates which data to retrieve and display and even calculate new values based on user-defined functions.

Setting up the example table

Create a table that will sort data into more than one partition.

```
CREATE TABLE cycling.rank_by_year_and_name (
  race_year int,
  race_name text,
  cyclist_name text,
  rank int,
```

```
PRIMARY KEY ((race_year, race_name), rank) );
```

Insert the data:

```
INSERT INTO cycling.rank_by_year_and_name (race_year, race_name,
cyclist_name, rank)
VALUES (2015, 'Tour of Japan - Stage 4 - Minami > Shinshu', 'Benjamin
PRADES', 1);
INSERT INTO cycling.rank_by_year_and_name (race_year, race_name,
cyclist_name, rank)
VALUES (2015, 'Tour of Japan - Stage 4 - Minami > Shinshu', 'Adam
PHELAN', 2);
INSERT INTO cycling.rank_by_year_and_name (race_year, race_name,
cyclist_name, rank)
VALUES (2015, 'Tour of Japan - Stage 4 - Minami > Shinshu', 'Thomas
LEBAS', 3);
INSERT INTO cycling.rank_by_year_and_name (race_year, race_name,
cyclist_name, rank)
VALUES (2015, 'Giro d''Italia - Stage 11 - Forli > Imola', 'Ilnur
ZAKARIN', 1);
INSERT INTO cycling.rank_by_year_and_name (race_year, race_name,
cyclist_name, rank)
VALUES (2015, 'Giro d''Italia - Stage 11 - Forli > Imola', 'Carlos
BETANCUR', 2);
INSERT INTO cycling.rank_by_year_and_name (race_year, race_name,
cyclist_name, rank)
VALUES (2014, '4th Tour of Beijing', 'Phillippe GILBERT', 1);
INSERT INTO cycling.rank_by_year_and_name (race_year, race_name,
cyclist_name, rank)
VALUES (2014, '4th Tour of Beijing', 'Daniel MARTIN', 2);
INSERT INTO cycling.rank_by_year_and_name (race_year, race_name,
cyclist_name, rank)
VALUES (2014, '4th Tour of Beijing', 'Johan Esteban CHAVES', 3);
```

Procedure

- Use a simple `SELECT` query to display all data from a table.

```
cqlsh> SELECT * FROM cycling.cyclist_category;
```

race_year	race_name	rank
cyclist_name		
2014	4th Tour of Beijing	1
Phillippe GILBERT		
2014	4th Tour of Beijing	2
Daniel MARTIN		
2014	4th Tour of Beijing	3
Johan Esteban CHAVES		

Using CQL

```
2015 | Giro d'Italia - Stage 11 - Forli > Imola | 1 |
Ilnur ZAKARIN
2015 | Giro d'Italia - Stage 11 - Forli > Imola | 2 |
Carlos BETANCUR
2015 | Tour of Japan - Stage 4 - Minami > Shinshu | 1 |
Benjamin PRADES
2015 | Tour of Japan - Stage 4 - Minami > Shinshu | 2 |
Adam PHELAN
2015 | Tour of Japan - Stage 4 - Minami > Shinshu | 3 |
Thomas LEBAS
```

- The example below illustrates how to create a query that uses **category** as a filter.

```
cqlsh> SELECT * FROM cycling.cyclist_category WHERE category =
'SPRINT';
```

category	id	lastname	points
SPRINT	1	TERRI	34
SPRINT	2	JIM	120

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

A query based on **lastname** can result in satisfactory results if the **lastname** column is [indexed](#).

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

```
cqlsh> SELECT category, points, lastname FROM
cycling.cyclist_category;
```

category	points	lastname
SPRINT	34	TERRI
SPRINT	120	JIM
GC	1234	TERRI
GC	2234	JIM

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

```
cqlsh> SELECT * From cycling.cyclist_name LIMIT 3;
```

<code>id</code>	<code>firstname</code>	<code>lastname</code>
<code>e7ae5cf3-d358-4d99-b900-85902fda9bb0</code>	Alex	FRAME
<code>5b6962dd-3f90-4c93-8f61-eabfa4a803e2</code>	Marianne	VOS
<code>220844bf-4860-49d6-9a4b-6b5d3a79cbfb</code>	Paolo	TIRALONGO

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

```
cqlsh> CREATE TABLE cycling.cyclist_cat_pts ( category text, points int, id UUID, lastname text, PRIMARY KEY (category, points) );
SELECT * FROM cycling.cyclist_cat_pts WHERE category = 'GC' ORDER BY points ASC;
```

<code>category</code>	<code>points</code>	<code>id</code>	<code>lastname</code>
GC	780	<code>829aa84a-4bba-411f-a4fb-38167a987cda</code>	SUTHERLAND
GC	1269	<code>220844bf-4860-49d6-9a4b-6b5d3a79cbfb</code>	TIRALONGO

- Tuples** are retrieved in their entirety. This example uses `AS` to change the header of the tuple name.

```
cqlsh> SELECT race_name, point_id, lat_long AS
CITY_LATITUDE_LONGITUDE FROM cycling.route;
```

<code>race_name</code>	<code>point_id</code>	<code>city_latitude_longitude</code>
47th Tour du Pays de Vaud	1	('Onnens', (46.8444, 6.6667))
47th Tour du Pays de Vaud	2	('Champagne', (46.833, 6.65))
47th Tour du Pays de Vaud	3	('Novalle', (46.833, 6.6))
47th Tour du Pays de Vaud	4	('Vuiteboeuf', (46.8, 6.55))
47th Tour du Pays de Vaud	5	('Baulmes', (46.7833, 6.5333))
47th Tour du Pays de Vaud	6	('Les Clées', (46.7222, 6.5222))

Retrieval using collections

Collections do not differ from other columns in retrieval. To query for a subset of the collection, a [secondary index for the collection](#) must be created.

Procedure

- Retrieve **teams** for a particular cyclist **id** from the **set**.

```
cqlsh> SELECT lastname, teams FROM cycling.cyclist_career_teams  
WHERE id = 5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
```

To query a table containing a collection, Cassandra retrieves the collection in its entirety. Keep collections small enough to be manageable because the collection store in memory. Alternatively, construct a data model to replace a collection if it must 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**.

```
lastname | teams  
-----+-----  
VOS | {'Nederland bloeft', 'Rabobank Women Team', 'Rabobank-Liv Giant', 'Rabobank-Liv Woman Cycling Team'}
```

- Retrieve **events** stored in a **list** from the upcoming calendar for a particular **year** and **month**.

```
cqlsh> SELECT * FROM cycling.upcoming_calendar WHERE year=2015 AND  
month=06;
```

```
year | month | events  
-----+-----+-----  
2015 | 6 | ['The Parx Casino Philly Cycling Classic', 'Critérium du Dauphiné', 'Vuelta Ciclista a Venezuela']
```

Note: The order is not alphabetical, but rather in the order of insertion.

- Retrieve **teams** for a particular cyclist **id** from the **map**.

```
cqlsh> SELECT lastname, firstname, teams FROM cycling.cyclist_teams  
WHERE id=5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
```

The order of the map output depends on the key type of the map. In this case, the key is an **integer** type.

```
lastname | firstname | teams  
-----+-----+-----  
VOS | Marianne | {2011: 'Nederland bloeft', 2012: 'Rabobank Women Team', 2013: 'Rabobank-Liv Giant', 2014: 'Rabobank-Liv Woman Cycling Team', 2015: 'Rabobank-Liv Woman Cycling Team'}
```

Retrieval using JSON

You can use CQL SELECT keywords to retrieve data from a table in the JSON format. For more information, see [What's New in Cassandra 2.2: JSON Support](#).

Retrieving all results in the JSON format

To get this result, insert the JSON keyword between the SELECT command and the data specifications. For example, :

```
cqlsh:cycling> select json name, checkin_id, timestamp from checkin;
[json]
-----
-----
{ "name": "BRAND", "checkin_id": "50554d6e-29bb-11e5-b345-feff8194dc9f",
  "timestamp": "2016-08-28 21:45:10.406Z" }
{ "name": "VOSS", "checkin_id": "50554d6e-29bb-11e5-b345-feff819cdc9f",
  "timestamp": "2016-08-28 21:44:04.113Z" }
(2 rows)
```

Retrieving selected columns in JSON format

To specify the JSON format for a selected column, enclose its name in `toJson()`. For example:

```
cqlsh:cycling> select name, checkin_id, toJson(timestamp) from checkin;
      name | checkin_id          | system.toJson(timestamp)
-----+-----+-----+
      -
 BRAND | 50554d6e-29bb-11e5-b345-feff8194dc9f | "2016-08-28 21:45:10.406Z"
    VOSS | 50554d6e-29bb-11e5-b345-feff819cdc9f | "2016-08-28 21:44:04.113Z"
```

Note: Cassandra 2.2 and later return a JSON-formatted `timestamp` with complete time zone information.

Retrieval using the IN keyword

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

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

Using CQL

Procedure

- Turn off paging.

```
cqlsh> PAGING OFF
```

- Retrieve and sort results in descending order.

```
cqlsh> SELECT * FROM cycling.cyclist_cat_pts WHERE category IN
('Time-trial', 'Sprint') ORDER BY id DESC;
```

category	id	points	lastname
Sprint	6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47	39	KRUIJSWIJK
Time-trial	6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47	3	KRUIJSWIJK
Sprint	220844bf-4860-49d6-9a4b-6b5d3a79cbfb	0	TIRALONGO
Time-trial	220844bf-4860-49d6-9a4b-6b5d3a79cbfb	182	TIRALONGO

- Alternatively, retrieve and sort results in ascending order.

To retrieve results, use the SELECT command.

```
cqlsh> SELECT * FROM cycling.cyclist_cat_pts WHERE category IN
('Time-trial', 'Sprint') ORDER BY id ASC;
```

category	id	points	lastname
Time-trial	220844bf-4860-49d6-9a4b-6b5d3a79cbfb	182	TIRALONGO
Sprint	220844bf-4860-49d6-9a4b-6b5d3a79cbfb	0	TIRALONGO
Time-trial	6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47	3	KRUIJSWIJK
Sprint	6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47	39	KRUIJSWIJK

- Retrieve rows using multiple clustering columns. This example searches the partition key **race_ids** for several races, but the partition key can also be composed as an equality for one value.

```
cqlsh> SELECT * FROM cycling.calendar WHERE race_id IN
(100, 101, 102) AND (race_start_date, race_end_date) IN
('2015-05-09', '2015-05-31'), ('2015-05-06', '2015-05-31');
```

race_id	race_start_date	race_end_date	race_name
100	2015-05-09 00:00:00-0700	2015-05-31 00:00:00-0700	Giro d'Italia

- Retrieve several rows using multiple clustering columns and inequality.

```
cqlsh> SELECT * FROM cycling.calendar WHERE race_id IN
  (100, 101, 102) AND (race_start_date, race_end_date) >=
  ('2015-05-09','2015-05-24');
```

race_id	race_start_date	race_end_date	race_name
100	2015-05-09 00:00:00-0700	2015-05-31 00:00:00-0700	Giro d'Italia
101	2015-06-07 00:00:00-0700	2015-06-14 00:00:00-0700	Criterium du Dauphine
102	2015-06-13 00:00:00-0700	2015-06-21 00:00:00-0700	Tour de Suisse

Retrieval by scanning a partition

Queries can scan a partition to retrieve a segment of stored data. The segment must be sequentially stored, so clustering columns can be used to define the [slice](#) of data selected.

Procedure

- Create a table **race_times** to hold the race times of various cyclists for various races.

```
CREATE TABLE cycling.race_times (race_name text, cyclist_name text,
  race_time text, PRIMARY KEY (race_name, race_time));
```

race_name	race_time	cyclist_name
17th Santos Tour Down Under	19:15:18	Rohan DENNIS
17th Santos Tour Down Under	19:15:20	Richie PORTE
17th Santos Tour Down Under	19:15:38	Cadel EVANS
17th Santos Tour Down Under	19:15:40	Tom DUMOULIN

- Scan the race times in the table to find a particular segment of data using a conditional operator.

```
SELECT * FROM cycling.race_times WHERE race_name = '17th Santos
Tour Down Under' AND race_time >= '19:15:19' AND race_time <=
'19:15:39' );
```

race_name	race_time	cyclist_name
17th Santos Tour Down Under	19:15:20	Richie PORTE
17th Santos Tour Down Under	19:15:38	Cadel EVANS

Retrieval using standard aggregate functions

The standard aggregate functions of `min`, `max`, `avg`, `sum`, and `count` are built-in functions.

Using CQL

Procedure

- A table **cyclist_points** records the race points for cyclists.

```
cqlsh> CREATE TABLE cycling.cyclist_points (id UUID, firstname text, lastname text, race_title text, race_points int, PRIMARY KEY (id, race_points));
```

id	race_points	firstname	lastname	race_title
e3b19ec4-774a-4d1c-9e5a-dece1e30aac	6	Giorgia	BRONZINI	Trofeo Alfredo Binda - Comune di Cittiglio
e3b19ec4-774a-4d1c-9e5a-dece1e30aac	75	Giorgia	BRONZINI	Acht van Westerveld
e3b19ec4-774a-4d1c-9e5a-dece1e30aac	120	Giorgia	BRONZINI	Tour of Chongming Island World Cup

- Calculate the standard aggregation function `sum` to find the sum of race points for a particular cyclist. The value of the aggregate will be returned.

```
cqlsh> SELECT sum(race_points) FROM cycling.cyclist_points WHERE id=e3b19ec4-774a-4d1c-9e5a-dece1e30aac;
```

```
system.sum(race_points)
-----
201
```

- Another standard aggregate function is `count`. A table **country_flag** records the country of each cyclist.

```
CREATE TABLE cycling.country_flag (country text, cyclist_name text, flag int STATIC, PRIMARY KEY (country, cyclist_name));
```

country	cyclist_name	flag
Belgium	Andre	1
Belgium	Jacques	1
France	Andre	3
France	George	3

- Calculate the standard aggregation function `count` to find the number of cyclists from Belgium. The value of the aggregate will be returned.

```
cqlsh> SELECT count(cyclist_name) FROM cycling.country_flag WHERE country='Belgium';
```

```
system.count(cyclist_name)
-----
2
```

Retrieval using a user-defined function (UDF)

The SELECT command can be used to retrieve data from a table while applying a user-defined function (UDF) to it.

Procedure

Use the user-defined function (UDF) `fLog()` created previously to retrieve data from a table `cycling.cyclist_points`.

```
cqlsh> SELECT id, lastname, fLog(race_points) FROM
    cycling.cyclist_points;
```

<code>id</code>	<code>lastname</code>	<code>cycling.flog(race_points)</code>
220844bf-4860-49d6-9a4b-6b5d3a79cbfb	TIRALONGO	0.693147
e3b19ec4-774a-4d1c-9e5a-decec1e30aac	BRONZINI	1.79176
e3b19ec4-774a-4d1c-9e5a-decec1e30aac	BRONZINI	4.31749
e3b19ec4-774a-4d1c-9e5a-decec1e30aac	BRONZINI	4.78749

Retrieval using user-defined aggregate (UDA) functions

Referring back to the user-defined aggregate `average()`, retrieve the average of the column `cyclist_time_sec` from a table.

Procedure

1. List all the data in the table.

```
cqlsh> SELECT * FROM cycling.team_average;
```

<code>team_name</code>	<code>cyclist_name</code>	<code>race_title</code>	<code>cyclist_time_sec</code>
TWENTY16 presented by Sho-Air	Lauren KOMANSKI	Amgen Tour of California Women's Race presented by SRAM - Stage 1 - Lake Tahoe	11451
> Lake Tahoe Lauren KOMANSKI 11451			
UnitedHealthCare Pro Cycling Womens Team	Hannah BARNES	Amgen Tour of California Women's Race presented by SRAM - Stage 1 - Lake Tahoe	11490
> Lake Tahoe Hannah BARNES 11490			
UnitedHealthCare Pro Cycling Womens Team	Katie HALL	Amgen Tour of California Women's Race presented by SRAM - Stage 1 - Lake Tahoe	11449
> Lake Tahoe Katie HALL 11449			
UnitedHealthCare Pro Cycling Womens Team	Linda VILLUMSEN	Amgen Tour of California Women's Race presented by SRAM - Stage 1 - Lake Tahoe	11485
> Lake Tahoe Linda VILLUMSEN 11485			
	Velocio-SRAM	Amgen Tour of California Women's Race presented by SRAM - Stage 1 - Lake Tahoe	
> Lake Tahoe Alena AMIALIUSIK	Velocio-SRAM	Amgen Tour of California Women's Race presented by SRAM - Stage 1 - Lake Tahoe	11451
> Lake Tahoe Trixi WORRACK	Velocio-SRAM	Amgen Tour of California Women's Race presented by SRAM - Stage 1 - Lake Tahoe	11453
> Lake Tahoe Trixi WORRACK 11453			

2. Apply the user-defined aggregate function `average()` to the `cyclist_time_sec` column.

```
cqlsh> SELECT average(cyclist_time_sec) FROM cycling.team_average
    WHERE team_name='UnitedHealthCare Pro Cycling Womens Team' AND
        race_title='Amgen Tour of California Women's Race presented by
        SRAM - Stage 1 - Lake Tahoe > Lake Tahoe';
```

```
cycling.average(cyclist_time_sec)
-----
11474.66667
```

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 2. Columns in System Tables

Table name	Column name	Comment
available_-ranges	keyspace_name, ranges	
batches	id, mutations, version	
batchlog	id, data, version, written_at	
built_views	keyspace_name, view_name	Information on materialized views
compaction_-history	id, bytes_in, bytes_out, columnfamily_name, compacted_at, keyspace_name, rows_merged	Information on compaction history
hints	target_id, hint_id, message_version, mutation	
"IndexInfo"	table_name, index_name	Information on indexes
local	key, bootstrapped, broadcast_address, cluster_name, cql_version, data_center, gossip_generation, host_id, listen_address, native_protocol_version, partitioner, rack, release_version, rpc_address, schema_version, thrift_version, tokens, truncated_at map	Information on a node has about itself and a superset of gossip .
paxos	row_key, cf_id, in_progress_bal lot, most_recent_commit, most_recent_commit_at, most_recent_commit_version, proposal,	Information on lightweight Paxos transactions

Table 2. Columns in System Tables (continued)

Table name	Column name	Comment
	proposal_ballot, proposal_version	
peers	peer, data_center, host_id, preferred_ip, rack, release_version, rpc_address, schema_version, tokens	Each node records what other nodes tell it about themselves over the gossip.
peer_events	peer, hints_dropped	
range_xfers	token_bytes, requested_at	
size_estimates	keyspace_name, table_name, range_start, range_end, mean_partition_size, partitions_count	Information on partitions
sstable_activity	keyspace_name, columnfamily_name, generation, rate_120m, rate_15m	
views_builds_in_progress	keyspace_name, view_name, generation_number, last_token	

Cassandra populates these tables in the system_schema keyspace.

Table 3. Columns in System_Schema Tables

Table name	Column name	Comment
aggregates	keyspace_name, aggregate_name, argument_types, final_func, initcond, return_type, state_func, state_type	Information about user-defined aggregates
columns	keyspace_name, table_name, column_name, clustering_order, column_name_bytes, kind, position, type	Information about table columns
dropped_columns	keyspace_name, table_name, column_name, dropped_time, type	Information about dropped columns
functions	keyspace_name, function_name, argument_types, argument_names, body, called_on_null_input, language, return_type	Information on user-defined functions

Table 3. Columns in System_Schema Tables (continued)

Table name	Column name	Comment
indexes	keyspace_name, table_name, index_name, kind,options	Information about indexes
keyspaces	keyspace_name, durable_writes, replication	Information on keyspace durable writes and replication
tables	keyspace_name, table_name, bloom_filter_fp_chance, caching, comment, compaction, compression, crc_check_chance, dclocal_read_repair_chance, default_time_to_live, extensions, flags, gc_grace_seconds, id, max_index_interval, memtable_flush_period_in_ms, min_index_interval, read_repair_chance, speculative_retry	Information on columns and column indexes. Used internally for compound primary keys.
triggers	keyspace_name, table_name, trigger_name, options	Information on triggers
types	keyspace_name, type_name, field_names, field_types	Information about user-defined types
views	keyspace_name, view_name, base_table_id, base_table_name, bloom_filter_fp_chance, caching, comment, compaction, compression, crc_check_chance, dclocal_read_repair_chance, default_time_to_live, extensions, flags, gc_grace_seconds, include_all_columns, max_index_interval, memtable_flush_period_in_ms, min_index_interval, read_repair_chance, speculative_retry, where_clause	Information about materialized views

Keyspace, table, and column information

An alternative to the `cqlsh describe_*` functions or using [DataStax DevCenter](#) to discover keyspace, table, and column information is querying `system.schema_*` table directly.

Procedure

- Query the defined keyspaces using the SELECT statement.

```
SELECT * FROM system.schema_keyspaces;

keyspace_name      | durable_writes | replication
-----+-----+-----
cycling |           True | {'class':
'org.apache.cassandra.locator.SimpleStrategy', 'replication_factor':
'1'}
test |           True | {'Cassandra': '1', 'class':
'org.apache.cassandra.locator.NetworkTopologyStrategy'}
system_auth |           True | {'class':
'org.apache.cassandra.locator.SimpleStrategy', 'replication_factor':
'1'}
system_schema |           True | {'class':
'org.apache.cassandra.locator.LocalStrategy'}
keyspace1 |           True | {'class':
'org.apache.cassandra.locator.SimpleStrategy', 'replication_factor':
'2'}
system_distributed |           True | {'class':
'org.apache.cassandra.locator.SimpleStrategy', 'replication_factor':
'3'}
system |           True |
{'class': 'org.apache.cassandra.locator.LocalStrategy'}
system_traces |           True | {'class':
'org.apache.cassandra.locator.SimpleStrategy', 'replication_factor':
'2'}

(15 rows)
```

- Get the schema information for tables in the `cycling` keyspace.

```
SELECT * FROM system_schema.tables
WHERE keyspace_name = 'cycling';
```

The following results shows the first record formatted with the cqlsh [cqlshExpand](#) ON option.

```
@ Row 1
-----
keyspace_name      | cycling
table_name         | birthday_list
bloom_filter_fp_chance | 0.01
caching | {'keys': 'ALL', 'rows_per_partition':
'NONE'}
cdc | null
```

Using CQL

```
comment | {'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy', 'max_threshold': '32', 'min_threshold': '4'}
compaction | {'chunk_length_in_kb': '64', 'class': 'org.apache.cassandra.io.compress.LZ4Compressor'}
compression | 1
crc_check_chance | 0.1
dclocal_read_repair_chance | 0
default_time_to_live | 0
extensions | {'RLACA': 0x6379636c6973745f6e616d65}
flags | {'compound'}
gc_grace_seconds | 864000
id | e439b922-2bc5-11e8-891b-23da85222d3d
max_index_interval | 2048
memtable_flush_period_in_ms | 0
min_index_interval | 128
nodesync | null
read_repair_chance | 0
speculative_retry | 99PERCENTILE
...
...
```

- Get details about a table's columns from **system_schema.columns**.

```
SELECT * FROM system_schema.columns
WHERE keyspace_name = 'cycling' AND table_name = 'cyclist_name';
```

keyspace_name	table_name	column_name	clustering_order	column_name_bytes	kind	position	type
cycling	cyclist_name	firstname	none	0x66697273746e616d65	regular	-1	text
cycling	cyclist_name	id	none	0x6964	partition_key	0	uuid
cycling	cyclist_name	lastname	none	0x6c6173746e616d65	regular	-1	text

(3 rows)

Note: The system_schema table does NOT show search index or row-level access control settings.

Cluster information

You can query system tables to get cluster topology information. Display 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 cluster, query the peers and local tables.

```
SELECT * FROM system.peers;
```

Output from querying the peers table looks something like this:

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

(2 rows)

Functions, aggregates, and user types

Currently, the **system** tables are the only method of displaying information about user-defined functions, aggregates, and user types.

Procedure

- Show all user-defined functions in the **system.schema_functions** table.

```
cqlsh> SELECT * FROM system.schema_functions;
```

keyspace_name	function_name	signature	argument_names	argument_types
		body		
		called_on_null_input	language	return_type
	cycling	avgfinal [tuple<int, bigint>] ['state']		
		[org.apache.cassandra.db.marshal.TupleType(org.apache.cassandra.db.marshal.Int32Type, org.apache.cassandra.db.marshal.LongType)] double r = 0; if (state.getInt(0) == 0) return null; r = state.getLong(1); r /= state.getInt(0); return Double.valueOf(r); True java org.apache.cassandra.db.marshal.DoubleType		
		cycling avgstate [tuple<int, bigint>, 'int'] ['state', 'val'] [org.apache.cassandra.db.marshal.TupleType(org.apache.cassandra.db.marshal.Int32Type, org.apache.cassandra.db.marshal.LongType), 'org.apache.cassandra.db.marshal.Int32Type'] if (val !=null) { state.setInt(0, state.getInt(0)+1); state.setLong(1, state.getLong(1)+val.intValue()); } return state; True java org.apache.cassandra.db.marshal.TupleType(org.apache.cassandra.db.marshal.Int32Type, org.apache.cassandra.db.marshal.LongType)		
		cycling flog ['double'] ['input'] [org.apache.cassandra.db.marshal.DoubleType] Double.valueOf(Math.log(input.doubleValue())); True java org.apache.cassandra.db.marshal.DoubleType		
		cycling fsin ['double'] ['input'] [org.apache.cassandra.db.marshal.DoubleType] Double.valueOf(Math.sin(input.doubleValue())); True java org.apache.cassandra.db.marshal.DoubleType		

Using CQL

- Show all user-defined aggregates in the **system.schema_aggregates** table.

```
cqlsh> SELECT * FROM system.schema_aggregates;
```

keyspace_name	aggregate_name	signature	argument_types	return_type	final_func	initcond	state_func	state_type
cycling	average	['int']	['org.apache.cassandra.db.marshal.Int32Type']	avgfinal	0x00000004000000000000000000000000	00000000000000000000000000000000	org.apache.cassandra.db.marshal.DoubleType	avgstate org.apache.cassandra.db.marshal.TupleType(org.apache.cassandra.db.marshal.Int32Type,org.apache.cassandra.db.marshal.LongType)

- Show all user-defined types in the **system.schema_usertypes** table.

```
cqlsh> SELECT * FROM system.schema_usertypes;
```

keyspace_name	type_name	field_names	field_types
cycling	basic_info	['birthday', 'lastname', 'nationality', 'weight', 'height']	['org.apache.cassandra.db.marshal.TimestampType', 'org.apache.cassandra.db.marshal.UTF8Type', 'org.apache.cassandra.db.marshal.UTF8Type', 'org.apache.cassandra.db.marshal.UTF8Type', 'org.apache.cassandra.db.marshal.UTF8Type']

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 data might be incorrectly retrieved during a query involving indexes on the basis of stale values in the index.

Indexes can be used for collections, collection columns, and any other columns except counter columns and static columns.

When to use an index

Cassandra's built-in indexes are best on a table having many rows that contain the indexed value. The more unique values that exist in a particular column, the more overhead you will have, on average, to query and maintain the index. For example, suppose you had a **races** table with a billion entries for cyclists in hundreds of races and wanted to look up rank by the cyclist. Many cyclists' ranks will share the same column value for race year. The **race_year** 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 for a query of a huge volume of records for a small number of results. See [Problems using a high-cardinality column index](#) below.
- In tables that use a counter column.
- On a frequently updated or deleted column. See [Problems using an index on a frequently updated or deleted column](#) below.
- To look for a row in a large partition unless narrowly queried. See [Problems using an index to look for a row in a large partition unless narrowly queried](#) below.

Problems using a high-cardinality column index

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

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

Problems using an index on a frequently updated or deleted column

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

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

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

Using a secondary index

Using CQL, you can create an index on a column after defining a table. You can also [index a collection column](#). Secondary indexes are used to query a table using a column that is not normally queryable.

Secondary indexes are tricky to use and can impact performance greatly. The index table is stored on each node in a cluster, so a query involving a secondary index can rapidly become a performance nightmare if multiple nodes are accessed. A general rule of thumb is to index a column with low cardinality of few values. Before creating an index, be aware of when and [when not to create an index](#).

Procedure

- The table **rank_by_year_and_name** can yield the rank of cyclists for races.

```
cqlsh> CREATE TABLE cycling.rank_by_year_and_name (
    race_year int,
    race_name text,
    cyclist_name text,
    rank int,
    PRIMARY KEY ((race_year, race_name), rank)
);
```

- Both **race_year** and **race_name** must be specified as these columns comprise the partition key.

```
cqlsh> SELECT * FROM cycling.rank_by_year_and_name WHERE
    race_year=2015 AND race_name='Tour of Japan - Stage 4 - Minami >
Shinshu';
```

race_year	race_name	rank	cyclist_name
2015	Tour of Japan - Stage 4 - Minami > Shinshu	1	Benjamin PRADES
2015	Tour of Japan - Stage 4 - Minami > Shinshu	2	Adam PHELAN
2015	Tour of Japan - Stage 4 - Minami > Shinshu	3	Thomas LEVAS

- A logical query to try is a listing of the rankings for a particular year. Because the table has a composite partition key, this query will fail if only the first column is used in the conditional operator.

```
cqlsh> SELECT * FROM cycling.rank_by_year_and_name WHERE
    race_year=2015;
```

```
cqlsh:cycling> SELECT * from rank_by_year_and_name where race_year=2015;
InvalidRequest: code=2200 [Invalid query] message="Partition key parts: race_name must be restricted as other parts are"
```

- An index is created for the race year, and the query will succeed. An index name is optional and must be unique within a keyspace. If you do not provide a name, Cassandra will assign a name like **race_year_idx**.

```
cqlsh> CREATE INDEX ryear ON cycling.rank_by_year_and_name
    (race_year);
SELECT * FROM cycling.rank_by_year_and_name WHERE race_year=2015;
```

race_year	race_name	rank	cyclist_name
2015	Giro d'Italia - Stage 11 - Forli > Imola	1	Ilnur ZAKARIN
2015	Giro d'Italia - Stage 11 - Forli > Imola	2	Carlos BETANCUR
2015	Tour of Japan - Stage 4 - Minami > Shinshu	1	Benjamin PRADES
2015	Tour of Japan - Stage 4 - Minami > Shinshu	2	Adam PHELAN
2015	Tour of Japan - Stage 4 - Minami > Shinshu	3	Thomas LEBAS

- A clustering column can also be used to create an index. An index is created on **rank**, and used in a query.

```
cqlsh> CREATE INDEX rrank ON cycling.rank_by_year_and_name (rank);
SELECT * FROM cycling.rank_by_year_and_name WHERE rank = 1;
```

race_year	race_name	rank	cyclist_name
2014	4th Tour of Beijing	1	Phillippe GILBERT
2015	Giro d'Italia - Stage 11 - Forli > Imola	1	Ilnur ZAKARIN
2015	Tour of Japan - Stage 4 - Minami > Shinshu	1	Benjamin PRADES

Using multiple indexes

Indexes can be created on multiple columns and used in queries. The general rule about cardinality applies to all columns indexed. In a real-world situation, certain columns might not be good choices, depending on their [cardinality](#).

Procedure

- The table **cycling.alt_stats** can yield the statistics about cyclists.

```
cqlsh> CREATE TABLE cycling.cyclist_alt_stats ( id UUID PRIMARY KEY,
    lastname text, birthday timestamp, nationality text, weight text,
    height text );
```

- Create indexes on the columns **birthday** and **nationality**.

```
cqlsh> CREATE INDEX birthday_idx ON cycling.cyclist_alt_stats
( birthday );
```

Using CQL

```
CREATE INDEX nationality_idx ON cycling.cyclist_alt_stats  
  ( nationality );
```

- Query for all the cyclists with a particular **birthday** from a certain **country**.

```
cqlsh> SELECT * FROM cycling.cyclist_alt_stats WHERE birthday =  
  '1982-01-29' AND nationality = 'Russia';
```

```
cqlsh:cycling> SELECT * FROM cycling.cyclist_alt_stats WHERE birthday = '1982-01-29' AND nationality = 'Russia';  
InvalidRequest: code=2200 [Invalid query] message="Cannot execute this query as it might involve data filtering and thus ma  
y have unpredictable performance. If you want to execute this query despite the performance unpredictability, use ALLOW FIL  
TERING"
```

- The indexes have been created on appropriate low cardinality columns, but the query still fails. Why? The answer lies with the partition key, which has not been defined. When you attempt a potentially expensive query, such as searching a range of rows, Cassandra requires the ALLOW FILTERING directive. The error is not due to multiple indexes, but the lack of a partition key definition in the query.

```
cqlsh> SELECT * FROM cycling.cyclist_alt_stats WHERE birthday =  
  '1990-05-27' AND nationality = 'Portugal' ALLOW FILTERING
```

id	birthday	height	lastname	nationality	weight
1ba0417d-62da-4103-b710-de6fb227db6f	1990-05-27 00:00:00-0700	null	PAULINHO	Portugal	null

Indexing a collection

Collections can be indexed and queried to find a collection containing a particular value. **Sets** and **lists** are indexed slightly differently from **maps**, given the key-value nature of **maps**.

Sets and **lists** can index all values found by indexing the collection column. **Maps** can index a map key, map value, or map entry using the methods shown below. Multiple indexes can be created on the same map column in a table, so that map keys, values, or entries can be queried. In addition, frozen collections can be indexed using **FULL** to index the full content of a frozen collection.

Note: All the cautions about using secondary indexes apply to indexing collections.

Procedure

- For **set** and **list** collections, create an index on the column name. Create an index on a set to find all the cyclists that have been on a particular team.

```
CREATE INDEX team_idx ON cycling.cyclist_career_teams ( teams );
```

```
SELECT * FROM cycling.cyclist_career_teams WHERE teams CONTAINS
  'Nederland bloeit';
```

<code>id</code>	<code>lastname</code>	<code>teams</code>
<code>Sb6962dd-3f90-4c93-8f61-eabfa4a803e2</code>	<code>VOS</code>	{'Nederland bloeit', 'Rabobank Women Team', 'Rabobank-Liv Giant', 'Rabobank-Liv Woman Cycling Team'}

- For **map** collections, create an index on the map key, map value, or map entry. Create an index on a **map key** to find all cyclist/team combinations for a particular year.

```
CREATE INDEX team_year_idx ON cycling.cyclist_teams ( KEYS
  (teams) );
SELECT * From cycling.cyclist_teams WHERE teams CONTAINS KEY 2015;
```

<code>id</code>	<code>firstname</code>	<code>lastname</code>	<code>teams</code>
<code>cb07baad-eac8-4f65-b28a-bddc06a0de23</code>	<code>Elizabeth</code>	<code>ARMITSTEAD</code>	{2011: 'Team Garmin - Cervelo', 2012: 'AA Drink - Leontien.nl', 2013: 'Boels-Dolmans Cycling Team', 2014: 'Boels-Dolmans Cycling Team', 2015: 'Boels-Dolmans Cycling Team'}
<code>Sb6962dd-3f90-4c93-8f61-eabfa4a803e2</code>	<code>Marianne</code>	<code>VOS</code>	{2011: 'Nederland bloeit', 2012: 'Rabobank Women Team', 2013: 'Rabobank-Liv Giant', 2014: 'Rabobank-Liv Woman Cycling Team', 2015: 'Rabobank-Liv Woman Cycling Team'}
<code>e7cd5752-bc0d-4157-a80f-7523add8bcd</code>	<code>Anna</code>	<code>VAN DER BREGGEN</code>	{2009: 'Team Flexpoint', 2012: 'Sengers Ladies Cycling Team', 2013: 'Sengers Ladies Cycling Team', 2014: 'Rabobank-Liv Woman Cycling Team', 2015: 'Rabobank-Liv Woman Cycling Team'}

- Create an index on the map entries and find cyclists who are the same age. An index using **ENTRIES** is only valid for **maps**.

```
CREATE TABLE cycling.birthday_list (cyclist_name text PRIMARY KEY,
  blist map<text,text>);
CREATE INDEX blist_idx ON cycling.birthday_list (ENTRIES(blist));
SELECT * FROM cycling.birthday_list WHERE blist['age'] = '23';
```

<code>cyclist_name</code>	<code>blist</code>
<code>Claudio HEINEN</code>	{'age': '23', 'bday': '27/07/1992', 'nation': 'GERMANY'}
<code>Laurence BOURQUE</code>	{'age': '23', 'bday': '27/07/1992', 'nation': 'CANADA'}

Using CQL

- Using the same index, find cyclists from the same country.

```
SELECT * FROM cyclist.birthday_list WHERE blist['nation'] =  
    'NETHERLANDS';
```

cyclist_name		blist
Luc HAGENAARS		{'age': '28', 'bday': '27/07/1987', 'nation': 'NETHERLANDS'}
Toine POELS		{'age': '52', 'bday': '27/07/1963', 'nation': 'NETHERLANDS'}

- Create an index on the map values and find cyclists who have a particular value found in the specified map. An index using `VALUES` is only valid for **maps**.

```
CREATE TABLE cycling.birthday_list (cyclist_name text PRIMARY KEY,  
    blist  
        map<text,text>;  
    ) ; CREATE INDEX blist_idx ON cycling.birthday_list (VALUES(blist));  
SELECT * FROM cycling.birthday_list CONTAINS 'NETHERLANDS';
```

lorina@cqlsh:cycling> SELECT * FROM cycling.birthday_list WHERE blist CONTAINS 'NETHERLANDS';
cyclist_name blist
Luc HAGENAARS {'age': '28', 'bday': '27/07/1987', 'nation': 'NETHERLANDS'}
Toine POELS {'age': '52', 'bday': '27/07/1963', 'nation': 'NETHERLANDS'}

- Create an index on the full content of a **FROZEN map**. The table in this example stores the number of Pro wins, Grand Tour races, and Classic races that a cyclist has competed in. The `SELECT` statement finds any cyclist who has 39 Pro race wins, 7 Grand Tour starts, and 14 Classic starts.

```
CREATE TABLE cycling.race_starts (cyclist_name text PRIMARY KEY,  
    rnumbers FROZEN<LIST<int>>);  
CREATE INDEX rnumbers_idx ON cycling.race_starts (FULL(rnumbers));  
SELECT * FROM cycling.race_starts WHERE rnumbers = [39,7,14];
```

cyclist_name		rnumbers
John DEGENKOLB		[39, 7, 14]

Altering a table

Tables can be changed with the `ALTER` command.

Altering columns in a table

The `ALTER TABLE` command can be used to add new columns to a table and to alter the column type of an existing column.

Procedure

- Add a `age` column of type `int` to the table `cycling.cyclist_alt_stats`.

```
cqlsh> ALTER TABLE cycling.cyclist_alt_stats ADD age int;
```

This creates the column metadata and adds the column to the table schema, and sets the value to `NULL` for all rows.

<code>id</code>	<code>age</code>	<code>birthday</code>	<code>height</code>	<code>lastname</code>	<code>nationality</code>	<code>weight</code>
e0953617-07eb-4c82-8f91-3b2757981625	null	1982-01-29 00:00:00-0800	1.78 m	BRUTT	Russia	68 kg
a9e96714-2dd0-41f9-8bd0-557196a44ecf	null	1986-04-21 00:00:00-0800	1.88 m	ISAYCHEV	Russia	80 kg
ed584e99-80f7-4b13-9a90-9dc5571e6821	null	1989-07-05 00:00:00-0700	1.69 m	TSATEVICH	Russia	64 kg
078654a6-42fa-4142-ae43-cebdcc67bd902	null	1981-01-14 00:00:00-0800	1.82 m	LAGUTIN	Russia	63 kg
1ba0417d-62da-4103-b710-de6fb227db6f	null	1990-05-27 00:00:00-0700	null	PAULINHO	Portugal	null
d74d6e70-7484-4df5-8551-f5090c37f617	null	1991-08-25 00:00:00-0700	1.75 m	GRMAY	Ethiopia	63 kg
c09e9451-50da-483d-8108-e6bea2e827b3	null	1981-03-29 00:00:00-0800	1.78 m	VEIKKANEN	Finland	66 kg
823ec386-2a46-45c9-be41-2425a4b7658e	null	1985-01-09 00:00:00-0800	1.84 m	BELKOV	Russia	71 kg
f1def54-7d96-4981-b14a-b70be4da82d2	null	1987-03-07 00:00:00-0800	null	TLEUBAYEV	Kazakhstan	null
4ceb495c-55ab-4f71-83b9-81117252bb13	null	1990-05-27 00:00:00-0700	null	DUVAL	France	null

- Add a column `favorite_color` of `varchar`, and then change the data type of the same column to `text`.

```
cqlsh> ALTER TABLE cycling.cyclist_alt_stats ADD favorite_color
      varchar;
ALTER TABLE cycling.cyclist_alt_stats ALTER favorite_color TYPE
      text;
```

Note: There are limitations on altering the data type of a column. The two data types, the original and the one changing to, must be compatible.

Altering a table to add a collection

The `ALTER TABLE` command can be used to add new collection columns to a table and to alter the column type of an existing column.

Procedure

- Alter the table **cycling.upcoming_calendar** to add a collection **map** description that can store a description for each race listed.

```
cqlsh> ALTER TABLE cycling.upcoming_calendar ADD description  
map<text,text>;
```

- After updating **cycling.upcoming_calendar** table to insert some data, **description** can be displayed.

```
cqlsh> UPDATE cycling.upcoming_calendar  
SET description = description + {'Criterium du Dauphine' : "Easy  
race", 'Tour du Suisse' : 'Hard uphill race'} WHERE year = 2015 AND  
month = 6;
```

year	month	description	events
2015	6	{'Criterium du Dauphine': 'Easy race', 'Tour de Suisse': 'Hard uphill race'} ['Criterium du Dauphine', 'Tour de Suisse']	null
2015	7	['Tour de France']	null
2015	8	tian', 'Tour de Pologne', 'Eneco Tour', 'Vuelta a Espana']	null ['Clasica Ciclista San Sebas

Altering the data type of a column

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

Procedure

Change the `favorite_color` column to store as `text` instead of `varchar` by changing the data type of the column.

```
ALTER TABLE cycling.cyclist_alt_stats ADD favorite_color varchar;  
ALTER TABLE cycling.cyclist_alt_stats ALTER favorite_color TYPE text;
```

Only newly inserted values will be created with the new type. However, the data type before must be [compatible with the new data type specified](#).

Altering the table properties

Using ALTER TABLE, you can change the table properties of a table.

Procedure

Alter a table to change the caching properties.

```
cqlsh> ALTER TABLE cycling.race_winners WITH caching = {'keys' : 'NONE',
   'rows_per_partition' : '15' };
```

Altering a user-defined type

The `ALTER TYPE` command can be used to add new columns to a user-defined type and to alter the data type of an existing column in a user-defined type.

Procedure

- Add a `middlename` column of type `text` to the user-defined type `cycling.fullname`.

```
cqlsh> ALTER TYPE cycling.fullname ADD middlename text;
```

This creates the column metadata and adds the column to the type schema. To verify, display the table `ssystem.schema_usertypes`.

keyspace_name	type_name	field_names	field_types
cycling basic_info	['birthday', 'lastname', 'nationality', 'weight', 'height']	['org.apache.cassandra.db.marshal.TimestampType', 'org.apache.cassandra.db.marshal.UTF8Type', 'org.apache.cassandra.db.marshal.UTF8Type', 'org.apache.cassandra.db.marshal.UTF8Type']	
cycling fullname	['firstname', 'lastname', 'middlename']	['org.apache.cassandra.db.marshal.UTF8Type', 'org.apache.cassandra.db.marshal.UTF8Type', 'org.apache.cassandra.db.marshal.UTF8Type']	

- A column can be renamed in either `ALTER TABLE` or `ALTER TYPE`. In `ALTER TABLE`, only primary key columns may be renamed.

```
cqlsh> ALTER TYPE cycling.fullname RENAME middlename TO
   middleinitial;
```

keyspace_name	type_name	field_names	field_types
cycling basic_info	['birthday', 'lastname', 'nationality', 'weight', 'height']	['org.apache.cassandra.db.marshal.TimestampType', 'org.apache.cassandra.db.marshal.UTF8Type', 'org.apache.cassandra.db.marshal.UTF8Type', 'org.apache.cassandra.db.marshal.UTF8Type']	
cycling fullname	['firstname', 'lastname', 'middleinitial']	['org.apache.cassandra.db.marshal.UTF8Type', 'org.apache.cassandra.db.marshal.UTF8Type']	

Removing a keyspace, schema, or data

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

Dropping a keyspace or table

Drop a table or keyspace using the `DROP` command.

Procedure

- Drop the **test** keyspace.

```
cqlsh> DROP KEYSPACE test;
```

- Drop the **cycling.last_3_days** table.

```
cqlsh> DROP TABLE cycling.last_3_days;
```

Deleting columns and rows

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

Procedure

1. Delete the values of the column **lastname** from the table **cyclist_name**.

```
cqlsh> DELETE lastname FROM cycling.cyclist_name WHERE id =  
c7fceba0-c141-4207-9494-a29f9809de6f;
```

2. Delete entire row for a particular race from the table **calendar**.

```
cqlsh> DELETE FROM cycling.calendar WHERE race_id = 200;
```

You can also define a [Time To Live](#) value for an individual column or an entire table. This property causes Cassandra to delete the data automatically after a certain amount of time has elapsed. For details, see [Expiring data with Time-To-Live](#).

Dropping a user-defined function (UDF)

You drop a user-defined function (UDF) using the `DROP` command.

Procedure

Drop the `fLog()` function. The conditional option `IF EXISTS` can be included.

```
cqlsh> DROP FUNCTION [ IF EXISTS] fLog;
```

Securing a table

Use role-based security commands to control access to keyspaces and objects.

Change the following settings in the `cassandra.yaml` to enable authentication and authorization:

```
authenticator: PasswordAuthenticator
authorizer: CassandraAuthorizer
```

And then restart Cassandra to apply the changes.

Database roles

Roles enable authorization management on a larger scale than security per user can provide. A role is created and may be granted to other roles. Hierarchical sets of permissions can be created. For more information, see [Role Based Access Control in Cassandra](#).

Procedure

- Create a role with a password. `IF NOT EXISTS` is included to ensure a previous role definition is not overwritten.

```
cqlsh> CREATE ROLE IF NOT EXISTS team_manager WITH PASSWORD =
'RockIt4Us!';
```

- Create a role with `LOGIN` and `SUPERUSER` privileges. `LOGIN` allows a client to identify as this role when connecting. `SUPERUSER` grants the ability to create roles unconditionally if the role has `CREATE` permissions.

```
cqlsh> CREATE ROLE sys_admin WITH PASSWORD = 'IcanDoIt4ll' AND LOGIN
= true AND SUPERUSER = true;
```

- Alter a role to change options. A role with `SUPERUSER` status can alter the `SUPERUSER` status of another role, but not the role currently held. `PASSWORD`,

Using CQL

`LOGIN`, and `SUPERUSER` can be modified with `ALTER ROLE`. To modify properties of a role, the user must have `ALTER` permission.

```
cqlsh> ALTER ROLE sys_admin WITH PASSWORD = 'All4one1forAll' AND  
SUPERUSER = false;
```

- Grant a role to a user or a role. To execute `GRANT` and `REVOKE` statements requires `AUTHORIZE` permission on the role being granted/revoked.

```
cqlsh> GRANT sys_admin TO team_manager;  
GRANT team_manager TO sandy;
```

- List roles of a user.

```
cqlsh> LIST ROLES;  
LIST ROLES OF sandy;
```

Note: `NORECURSIVE` is an option to discover all roles directly granted to a user. Without `NORECURSIVE`, transitively acquired roles are also listed.

role	super	login	options
<code>cassandra</code>	<code>True</code>	<code>True</code>	<code>{}</code>
<code>sysadmin</code>	<code>True</code>	<code>True</code>	<code>{}</code>
<code>team_manager</code>	<code>True</code>	<code>False</code>	<code>{}</code>

- Revoke role that was previously granted to a user or a role. Any permission that derives from the role is revoked.

```
cqlsh> REVOKE sys_admin FROM team_manager;  
REVOKE team_manager FROM sandy;
```

- Drop role that is not a current role. User must be a `SUPERUSER`.

```
DROP ROLE IF EXISTS sys_admin;
```

Database Permissions

Authentication and authorization should be set based on roles, rather than users. Authentication and authorization are based on roles, and user commands are included only for legacy backwards compatibility.

Roles may be granted to other roles to create hierarchical permissions structures; in these hierarchies, permissions and SUPERUSER status are inherited, but the LOGIN privilege is not.

Permissions can be granted at any level of the database hierarchy and flow downwards.

Keyspaces and tables are hierarchical as follows: ALL KEYSPACES > KEYSPACE > TABLE. Functions are hierarchical in the following manner: ALL FUNCTIONS > KEYSPACE > FUNCTION. ROLES can also be hierarchical and encompass other ROLES. Permissions can be granted on:

- CREATE - keyspace, table, function, role, index
- ALTER - keyspace, table, function, role
- DROP - keyspace, table, function, role, index
- SELECT - keyspace, table
- MODIFY - INSERT, UPDATE, DELETE, TRUNCATE - keyspace, table
- AUTHORIZE - GRANT PERMISSION, REVOKE PERMISSION - keyspace, table, function, and role
- DESCRIBE - LIST ROLES
- EXECUTE - SELECT, INSERT, UPDATE - functions

Note: Index must additionally have ALTER permission on the base table in order to CREATE or DROP.

The permissions are extensive with many variations. A few examples are described below.

Procedure

- The first line grants anyone with the **team_manager** role the ability to INSERT, UPDATE, DELETE, and TRUNCATE any table in the keyspace **cycling**. The second line grants anyone with the **sys_admin** role the ability to view all roles in the database.

```
GRANT MODIFY ON KEYSPACE cycling TO team_manager;
GRANT DESCRIBE ON ALL ROLES TO sys_admin;
```

- The first line revokes SELECT in all keyspaces for anyone with the **team_manager** role. The second line prevents the **team_manager** role from executing the named function **fLog()**.

```
REVOKE SELECT ON ALL KEYSPACES FROM team_manager;
```

Using CQL

```
REVOKE EXECUTE ON FUNCTION cycling.fLog(double) FROM team_manager;
```

- All permissions can be listed, for either all keyspaces or a single keyspace.

```
LIST ALL PERMISSIONS OF sandy;
LIST ALL PERMISSIONS ON cycling.cyclist_name OF chuck;
```

role	username	resource	permission
sysadmin	sysadmin	<keyspace cycling>	MODIFY
sysadmin	sysadmin	<keyspace cycling>	AUTHORIZE

- Grant permission to drop all functions, including aggregate in the current keyspace.

```
GRANT DROP ON ALL FUNCTIONS IN KEYSPACE TO coach;
```

Database users

User-based access control enables authorization management on a per-user basis.

Note: Creating users is supported for backwards compatibility. Authentication and authorization are based on roles, and role-based commands should be used.

Procedure

- Create a user with a password. `IF NOT EXISTS` is included to ensure a previous user definition is not overwritten.

```
cqlsh> CREATE USER IF NOT EXISTS sandy WITH PASSWORD 'Ride2Win@'
NOSUPERUSER;
```

- Create a user with SUPERUSER privileges. SUPERUSER grants the ability to create users and roles unconditionally.

```
cqlsh> CREATE USER chuck WITH PASSWORD 'Always1st$' SUPERUSER;
```

Note: `WITH PASSWORD` implicitly specifies `LOGIN`.

- Alter a user to change options. A role with SUPERUSER status can alter the SUPERUSER status of another user, but not the user currently held. To modify properties of a user, the user must have permission.

```
cqlsh> ALTER USER sandy SUPERUSER;
```

- List the users.

```
cqlsh> LIST USERS;
```

name	super
cassandra	True
chuck	True
sandy	False
sysadmin	True

- Drop user that is not a current user. User must be a SUPERUSER.

```
DROP USER IF EXISTS chuck;
```

Tracing consistency changes

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

- ONE
Returns data from the nearest replica.
- QUORUM
Returns the most recent data from the majority of replicas.
- ALL
Returns the most recent data from all replicas.

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

Setup to trace consistency changes

To setup five nodes on your local computer, trace reads at different consistency levels, and then compare the results. This example uses [ccm](#), a tool for running multiple nodes of Cassandra on a local computer.

Procedure

1. Get the [ccm library of scripts](#) from github.

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

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

Refer to the ccm README for prerequisites.

2. Optional: For Mac computers, set up loopback aliases. All other platforms, skip this step.

```
$ sudo ifconfig lo0 alias 127.0.0.2 up $ sudo ifconfig lo0 alias  
127.0.0.3 up $ sudo ifconfig lo0 alias 127.0.0.4 up $ sudo ifconfig  
lo0 alias 127.0.0.5 up
```

3. Download Apache [Cassandra source code](#) into the ./ccm/repository.

4. Start the ccm cluster named `trace_consistency` using Cassandra version 3.0.5. The source code to run the cluster will automatically download and compile.

```
$ ccm create trace_consistency -v 3.0.5
```

```
Current cluster is now: trace_consistency
```

5. Use the following commands to populate and check the cluster:

```
$ ccm populate -n 5 $ ccm start
```

6. Check that the cluster is up:

```
$ ccm node1 ring
```

The output shows the status of all five nodes.

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

```
$ ccm node1 cqlsh
```

Related information

[Cassandra cassandra.yaml](#)

Trace reads at different consistency levels

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

Tip: For more information on tracing data, see [this post](#) on the DataStax Support Blog, which explains in detail how to locate data on disk.

Procedure

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

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

2. Create a table, and insert some values:

```
cqlsh> USE cycling_alt;
cqlsh> CREATE TABLE cycling_alt.tester ( id int PRIMARY KEY, col1
  int, col2 int );
cqlsh> INSERT INTO cycling_alt.tester (id, col1, col2) VALUES (0, 0,
  0);
```

3. Turn on tracing and use the [CONSISTENCY command](#) to check that the consistency level is ONE, the default.

```
cqlsh> TRACING on;
cqlsh> CONSISTENCY;
```

The output should be:

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

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

```
cqlsh> SELECT * FROM cycling_alt.tester WHERE id = 0;
```

The output includes tracing information:

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

(1 rows)

Tracing session: 65bd3150-0109-11e6-8b46-15359862861c

activity
| timestamp | source | source_elapsed
-----+-----+-----
                               Execute
CQL3 query | 2016-04-12 16:50:55.461000 | 127.0.0.1 | 0
Parsing SELECT * FROM cycling_alt.tester WHERE id = 0;
[SharedPool-Worker-1] | 2016-04-12 16:50:55.462000 | 127.0.0.1 | 276
                               Preparing statement
[SharedPool-Worker-1] | 2016-04-12 16:50:55.462000 | 127.0.0.1 | 509
                               Executing single-partition query on tester
[SharedPool-Worker-2] | 2016-04-12 16:50:55.463000 | 127.0.0.1 | 1019
                               Acquiring sstable references
[SharedPool-Worker-2] | 2016-04-12 16:50:55.463000 | 127.0.0.1 | 1106
                               Merging memtable contents
[SharedPool-Worker-2] | 2016-04-12 16:50:55.463000 | 127.0.0.1 | 1159
                               Read 1 live and 0 tombstone cells
[SharedPool-Worker-2] | 2016-04-12 16:50:55.463000 | 127.0.0.1 | 1372
                               Request
complete | 2016-04-12 16:50:55.462714 | 127.0.0.1 | 1714
```

The tracing results list all the actions taken to complete the SELECT statement.

5. Change the consistency level to QUORUM to trace what happens during a read with a QUORUM consistency level.

```
cqlsh> CONSISTENCY quorum;
cqlsh> SELECT * FROM cycling_alt.tester WHERE id = 0;

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

(1 rows)

Tracing session: 5e3601f0-0109-11e6-8b46-15359862861c

activity
      | timestamp          | source    | source_elapsed
-----+-----+-----+
                               Execute
CQL3 query | 2016-04-12 16:50:42.831000 | 127.0.0.1 | 0
Parsing SELECT * FROM cycling_alt.tester WHERE id = 0;
[SharedPool-Worker-1] | 2016-04-12 16:50:42.831000 | 127.0.0.1 |
259
                               Preparing statement
[SharedPool-Worker-1] | 2016-04-12 16:50:42.831000 | 127.0.0.1 |
557
                               Executing single-partition query on tester
[SharedPool-Worker-3] | 2016-04-12 16:50:42.832000 | 127.0.0.1 |
1076
                               Acquiring sstable references
[SharedPool-Worker-3] | 2016-04-12 16:50:42.832000 | 127.0.0.1 |
1182
                               Merging memtable contents
[SharedPool-Worker-3] | 2016-04-12 16:50:42.832000 | 127.0.0.1 |
1268
                               Read 1 live and 0 tombstone cells
[SharedPool-Worker-3] | 2016-04-12 16:50:42.832000 | 127.0.0.1 |
1632
                               Request
complete | 2016-04-12 16:50:42.832887 | 127.0.0.1 | 1887
```

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

```
cqlsh> CONSISTENCY ALL;
cqlsh> SELECT * FROM cycling_alt.tester WHERE id = 0;

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

Using CQL

```
(1 rows)

Tracing session: 6c4678b0-0109-11e6-8b46-15359862861c

activity
| timestamp | source | source_elapsed
-----+-----+-----+
[SharedPool-Worker-1] | 2016-04-12 16:51:06.427000 | 127.0.0.1 | 324
[SharedPool-Worker-1] | 2016-04-12 16:51:06.427000 | 127.0.0.1 | 524
[SharedPool-Worker-1] | 2016-04-12 16:51:06.427000 | 127.0.0.1 | 1016
[SharedPool-Worker-3] | 2016-04-12 16:51:06.428000 | 127.0.0.1 | 1793
[SharedPool-Worker-3] | 2016-04-12 16:51:06.428000 | 127.0.0.1 | 1886
[SharedPool-Worker-3] | 2016-04-12 16:51:06.429000 | 127.0.0.1 | 1951
[SharedPool-Worker-3] | 2016-04-12 16:51:06.429000 | 127.0.0.1 | 2176
complete | 2016-04-12 16:51:06.429391 | 127.0.0.1 | 2391
```

How consistency affects performance

Changing the consistency level can affect read performance. The tracing output shows that as you change the consistency level from ONE to QUORUM to ALL, performance degrades in from 1714 to 1887 to 2391 microseconds, respectively. If you follow the steps in this tutorial, it is not guaranteed that you will see the same trend because querying a one-row table is a degenerate case, used for example purposes. The difference between QUORUM and ALL is slight in this case, so depending on conditions in the cluster, performance using ALL might be faster than QUORUM.

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

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

Tracing queries on large datasets

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

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

Displaying rows from an unordered partitioner with the TOKEN function

The ByteOrdered partitioner arranges tokens the same way as key values, but the RandomPartitioner and Murmur3Partitioner distribute tokens in a completely unordered manner. When using the RandomPartitioner or Murmur3Partitioner, Cassandra rows are ordered by the hash of their partition key, or for one partition queries, rows are ordered by their clustering key. Hence, the order of rows is not meaningful, because of the hashes generated.

To order the rows for display when using RandomPartitioner or Murmur3Partitioner, the token function may be used. However, ordering with the TOKEN function does not always provide the expected results. Use the TOKEN function to express a conditional relation on a partition key column. In this case, the query returns rows based on the token of the partition key rather than on the value.

The TOKEN function can also be used to select a range of partitions for a ByteOrderedPartitioner. Using the TOKEN function with ByteOrderedPartitioner will generally yield expected results.

The type of the arguments to the TOKEN function depends on the type of the columns used as the argument of the function. The return type depends on the partitioner in use:

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

Procedure

- Select data based on a range of tokens of a particular column value.

```
SELECT * FROM cycling.last_3_days WHERE TOKEN(year) >
TOKEN('2015-05-24');
```

Using CQL

year	rank	cyclist_name	race_name
2015-05-26 00:00:00-0700	1	Mikel Landa	Giro d'Italia Stage 16
2015-05-26 00:00:00-0700	2	Steven Kruijswijk	Giro d'Italia Stage 16
2015-05-26 00:00:00-0700	3	Alberto Contador	Giro d'Italia Stage 16
2015-05-25 00:00:00-0700	1	Matthew Busche	National Championships United States - Road Race (NC)
2015-05-25 00:00:00-0700	2	Joe Dombrowski	National Championships United States - Road Race (NC)
2015-05-25 00:00:00-0700	3	Kiel Reijnen	National Championships United States - Road Race (NC)

- The results will not always be consistent with expectations, because the token function actually queries directly using tokens. Underneath, the token function uses token-based comparisons and does not convert year to token (not year > '2015-05-26').

```
SELECT * FROM cycling.last_3_days WHERE TOKEN(year) >
TOKEN('2015-05-26');
```

year	rank	cyclist_name	race_name
2015-05-25 00:00:00-0700	1	Matthew Busche	National Championships United States - Road Race (NC)
2015-05-25 00:00:00-0700	2	Joe Dombrowski	National Championships United States - Road Race (NC)
2015-05-25 00:00:00-0700	3	Kiel Reijnen	National Championships United States - Road Race (NC)

- Display the tokens for all values of the column year.

```
SELECT TOKEN(year) FROM cycling.last_3_days;
```

```
system.token(year)
-----
7269113853363653308
7269113853363653308
7269113853363653308
8466757759759754561
8466757759759754561
8466757759759754561
```

- Tokens and partition keys can be mixed in conditional statements. The results will not always be straightforward, but they are not unexpected if you understand what the TOKEN function does.

```
SELECT * FROM cycling.last_3_days WHERE TOKEN(year) <
TOKEN('2015-05-26') AND year IN ('2015-05-24','2015-05-25');
```

year	rank	cyclist_name	race_name
------	------	--------------	-----------

--	--	--	--

Determining time-to-live (TTL) for a column

To set the TTL for data, use the `USING TTL` keywords. The `TTL` function may be used to retrieve the TTL information.

The `USING TTL` keywords can be used to insert data into a table for a specific duration of time. To determine the current time-to-live for a record, use the `TTL` function.

Procedure

- Insert data into the table **cycling.calendar** and use the `USING TTL` clause to set the expiration period to 86400 seconds.

```
INSERT INTO cycling.calendar (race_id, race_name, race_start_date,
race_end_date) VALUES (200, 'placeholder','2015-05-27',
'2015-05-27') USING TTL 86400;
```

- Issue a `SELECT` statement to determine how much longer the data has to live.

```
SELECT TTL (race_name) from cycling.calendar WHERE race_id = 200;
```

ttl(race_name)

86371

If you repeat this step after some time, the time-to-live value will decrease.

Using CQL

- The time-to-live value can also be updated with the **USING TTL** keywords in an **UPDATE** command.

```
UPDATE cycling.calendar USING TTL 300 SET race_name = 'dummy' WHERE  
race_id = 200 AND race_start_date = '2015-05-27' AND race_end_date  
= '2015-05-27';
```

ttl(race_name)

295

Determining the date/time of a write

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

Procedure

Retrieve the date/time that the value **Paolo** was written to the **firstname** column in the table **cyclist_points**. Use the WRITETIME function in a SELECT statement, followed by the name of a column in parentheses:

```
SELECT WRITETIME (firstname) FROM cycling.cyclist_points WHERE  
id=220844bf-4860-49d6-9a4b-6b5d3a79cbfb;
```

writetime(firstname)

1435108325093225

Note: The writetime output in microseconds converts to Wed, 24 Jun 2015 01:12:05 GMT.

Legacy tables

Legacy tables must be handled differently from currently built CQL tables.

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 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.

Querying a legacy table

Using CQL, you can query a legacy table. A legacy table managed in CQL includes an implicit WITH COMPACT STORAGE directive.

Using a music service example, select all the columns in the **playlists** table that was created in CQL. This output appears:

```
[default@music] GET playlists [62c36092-82a1-3a00-93d1-46196ee77204];
=> (column =7db1a490-5878-11e2-bcf0-0800200c9a66:, value
=, timestamp =1357602286168000 )
=> (column =7db1a490-5878-11e2-bcf0-0800200c9a66:album,
value =4e6f204f6e6520526964657320666f722046726565,
timestamp =1357602286168000 )
.
.
.
=> (column =a3e64f8f-bd44-4f28-b8d9-6938726e34d4:title,
value =4c61204772616e6765, timestamp
=1357599350478000 )
Returned 16 results.
```

The output of cell values is unreadable because GET returns the values in byte format.

Using CQL

Using a CQL legacy table query

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

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

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

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

INSERT INTO clicks (userid, url,timestamp) VALUES
(148e9150-1dd2-11b2-0000-242d50cf1fff,'http://google.com', '2016-02-03');

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';

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

4. 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 supported 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 3.3.

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

Identifiers 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 4. 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 "Bar" 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.

Valid characters in names

Keyspace and table names must begin with an alpha-numeric character and can only contain alpha-numeric characters and underscores. All other names, such as COLUMN, FUNCTION, AGGREGATE, TYPE, etc., can begin with and contain any character.

To specify a name that contains a special character, like period (.) or hyphen (-), enclose the name in double quotes.

Table 5. What Works and What Doesn't

Creations that Work	Creations that Don't Work
CREATE TABLE foo ...	CREATE TABLE foo!\$% ...
CREATE TABLE foo_bar ...	CREATE TABLE foo[]!"90 ...
CREATE TABLE foo ("what#*&" text, ...)	CREATE TABLE foo (what#*& text, ...)
ALTER TABLE foo5 ...	ALTER TABLE "foo5\$\$%"...
CREATE FUNCTION "foo5\$\$\$\$^%" ...	CREATE FUNCTION foo5\$\$...%
CREATE AGGREGATE "foo5!@#" ...	CREATE AGGREGATE foo5\$\$
CREATE TYPE foo5 ("bar#"9 text, ...)	CREATE TYPE foo5 (bar#9 text ...)

Escaping characters

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

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

```
INSERT INTO cycling.calendar (race_id, race_start_date, race_end_date,
race_name) VALUES
(201, '2015-02-18', '2015-02-22', 'Women''s Tour of New Zealand');
```

An alternative is to use dollar-quoted strings. Dollar-quoted string constants can be used to create functions, insert data, and select data when complex quoting is needed. Use double dollar signs to enclose the desired string.

```
INSERT INTO cycling.calendar (race_id, race_start_date, race_end_date,
race_name) VALUES
(201, '2015-02-18', '2015-02-22', $$Women's Tour of New Zealand$$);
```

Valid literals

Valid literal consist of these kinds of values:

- blob
hexadecimal defined as 0[xX](hex)+
- boolean
true or false, case-insensitive, not enclosed in quotation marks
- numeric constant

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

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

NaN and Infinity are floats.

- identifier

Names of tables, columns, types, and other objects are identifiers. Since keyspace and table names are used in system file names, they must start with a letter or number and can only contain alphanumeric characters and underscores. All other identifiers, such as column and user-defined function names can contain any character. To specify an identifier that contains a special character enclose the name in quotes.

- 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 possessive: 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 802 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	-2.6034345E+38	-2.6034345E+38

CQL code comments

Use the following notation to include comments in CQL code:

- For a single line or end of line put a double hyphen before the text, this comments out the rest of the line:

```
select * from cycling.route; -- End of line comment
```

- For a single line or end of line put a double forward slash before the text, this comments out the rest of the line:

```
select * from cycling.route; // End of line comment
```

- For a block of comments put a forward slash asterisk at the beginning of the comment and then asterisk forward slash at the end.

```
/* This is the first line of
   of a comment that spans multiple
   lines */
select * from cycling.route;
```

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 6. Keywords

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

Table 6. Keywords (continued)

Keyword	Reserved
BATCH	yes
BEGIN	yes
BIGINT	no
BLOB	no
BOOLEAN	no
BY	yes
CLUSTERING	no
COLUMNFAMILY	yes
COMPACT	no
CONSISTENCY	no
COUNT	no
COUNTER	no
CREATE	yes
CUSTOM	no
DECIMAL	no
DELETE	yes
DESC	yes
DISTINCT	no
DOUBLE	no
DROP	yes
EACH_QUORUM	yes
ENTRIES	yes
EXISTS	no
FILTERING	no
FLOAT	no
FROM	yes
FROZEN	no

Table 6. Keywords (continued)

Keyword	Reserved
FULL	yes
GRANT	yes
IF	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
MATERIALIZED	yes
MODIFY	yes
NAN	yes
NORECURSIVE	yes
NOSUPERUSER	no
NOT	yes
OF	yes
ON	yes

Table 6. Keywords (continued)

Keyword	Reserved
ONE	yes
ORDER	yes
PARTITION	yes
PASSWORD	yes
PER	yes
PERMISSION	no
PERMISSIONS	no
PRIMARY	yes
QUORUM	yes
RENAME	yes
REVOKE	yes
SCHEMA	yes
SELECT	yes
SET	yes
STATIC	no
STORAGE	no
SUPERUSER	no
TABLE	yes
TEXT	no
TIME	yes
TIMESTAMP	no
TIMEUUID	no
THREE	yes
TO	yes
TOKEN	yes
TRUNCATE	yes
TTL	no

Table 6. Keywords (continued)

Keyword	Reserved
TUPLE	no
TWO	yes
TYPE	no
UNLOGGED	yes
UPDATE	yes
USE	yes
USER	no
USERS	no
USING	yes
UUID	no
VALUES	no
VARCHAR	no
VARINT	no
VIEW	yes
WHERE	yes
WITH	yes
WRITETIME	no

CQL data types

CQL defines built-in data types for columns. The [counter type](#) is unique.

Table 7. CQL Data Types

CQL Type	Constants supported	Description
ascii	strings	US-ASCII character string
bigint	integers	64-bit signed long
blob	blobs	Arbitrary bytes (no validation), expressed as hexadecimal

Table 7. CQL Data Types (continued)

CQL Type	Constants supported	Description
boolean	booleans	true or false
counter	integers	Distributed counter value (64-bit long)
date	strings	Value is a date with no corresponding time value; Cassandra encodes date as a 32-bit integer representing days since epoch (January 1, 1970). Dates can be represented in queries and inserts as a string, such as 2015-05-03 (yyyy-mm-dd)
decimal	integers, floats	Variable-precision decimal Java type Note: When dealing with currency, it is a best practice to have a currency class that serializes to and from an int or use the Decimal form.
double	integers, floats	64-bit IEEE-754 floating point Java type
float	integers, floats	32-bit IEEE-754 floating point Java type
frozen	user-defined types, collections, tuples	A frozen value serializes multiple components into a single value. Non-frozen types allow updates to individual fields. Cassandra treats the value of a frozen type as a blob. The entire value must be overwritten. Note: Cassandra no longer requires the use of frozen for tuples : <code>frozen <tuple <int, tuple<text, double>>></code>
inet	strings	IP address string in IPv4 or IPv6 format, used by the python-cql driver and CQL native protocols
int	integers	32-bit signed integer

Table 7. CQL Data Types (continued)

CQL Type	Constants supported	Description
list	n/a	<p>A collection of one or more ordered elements: [literal, literal, literal].</p> <p>CAUTION: Lists have limitations and specific performance considerations. Use a frozen list to decrease impact. In general, use a set instead of list.</p>
map	n/a	A JSON-style array of literals: { literal : literal, literal : literal ... }
set	n/a	A collection of one or more elements: { literal, literal, literal }
smallint	integers	2 byte integer
text	strings	UTF-8 encoded string
time	strings	Value is encoded as a 64-bit signed integer representing the number of nanoseconds since midnight. Values can be represented as strings, such as 13:30:54.234.
timestamp	integers, strings	Date and time with millisecond precision, encoded as 8 bytes since epoch. Can be represented as a string, such as 2015-05-03 13:30:54.234.
timeuuid	uuids	Version 1 UUID only
tinyint	integers	1 byte integer
tuple	n/a	A group of 2-3 fields.
uuid	uuids	A UUID in standard UUID format
varchar	strings	UTF-8 encoded string
varint	integers	Arbitrary-precision integer Java type

In addition to the CQL types listed in this table, you can use a string containing the name of a JAVA class (a sub-class of `AbstractType` loadable by Cassandra)

as a CQL type. The class name should either be fully qualified or relative to the org.apache.cassandra.db.marshal package.

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

Java types

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

Table 8. Derivation of selective CQL types

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

CQL type compatibility

CQL data types have strict requirements for conversion compatibility. The following table shows the allowed alterations for data types:

Data type may be altered to:	Data type
ascii, bigint, boolean, decimal, double, float, inet, int, timestamp, timeuuid, uuid, varchar, varint	blob
int	varint
text	varchar
timeuuid	uuid
varchar	text

Clustering columns have even stricter requirements, because clustering columns mandate the order in which data is written to disk. The following table shows the allow alterations for data types used in clustering columns:

Data type may be altered to:	Data type
int	varint
text	varchar
varchar	text

Blob type

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

Blob conversion functions

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

- `typeAsBlob(value)`
- `blobAsType(value)`

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

This example shows how to use `bigintAsBlob`:

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

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

SELECT * FROM bios;

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

This example shows how to use `blobAsBigInt`.

```
ALTER TABLE bios ADD id bigint;

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

SELECT * FROM bios;

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

```
fred | 0x0000000000000003 | 3
```

Collection type

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

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

Collection types cannot be nested, but frozen collection types can be nested inside frozen or non-frozen collections. For example, you may define a list within a list, provided the inner list is frozen:

```
list<frozen <list<int>>>
```

Indexes may be created on a collection column of any type.

Using frozen in a collection

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

```
column_name collection_type<data_type, frozen<column_name>>
```

For example:

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

Counter type

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

Use counter types as described in the ["Using a counter"](#) section. Do not assign this type to a column that serves as the primary key or partition key. Also, do not use the counter type in a table that contains anything other than counter types and the primary key. To generate sequential numbers for surrogate keys, use the `timeuuid` type instead of the counter type.

You cannot create an index on a counter column or set data in a counter column to expire using the Time-To-Live (TTL) property.

UUID and timeuuid types

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

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

A valid timeuuid conforms to the timeuuid format shown in [valid literals](#).

UUID and timeuuid functions

The `uuid()` function takes no parameters and generates a random Type 4 UUID suitable for use in `INSERT` or `UPDATE` statements.

Several `timeuuid()` functions are designed for use with the `timeuuid()` type:

- `dateOf()`

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

- `now()`

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

- `minTimeuuid()` and `maxTimeuuid()`

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

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

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.

- `unixTimestampOf()`

Used in a `SELECT` clause, this function extracts the timestamp in milliseconds of a `timeuuid` column in a result set. Returns the value as a raw, 64-bit integer timestamp.

Cassandra 2.2 and later support some additional `timeuuid` and `timestamp` functions to manipulate dates. The functions can be used in `INSERT`, `UPDATE`, and `SELECT` statements.

- `toDate(timeuuid)`

Converts `timeuuid` to date in `YYYY-MM-DD` format.

- `toTimestamp(timeuuid)`

Converts `timeuuid` to `timestamp` format.

- `toUnixTimestamp(timeuuid)`

Converts `timeuuid` to `UNIX timestamp` format.

- `toDate(timestamp)`

Converts `timestamp` to date in `YYYY-MM-DD` format.

- `toUnixTimestamp(timestamp)`

Converts `timestamp` to **UNIX timestamp** format.

- `toTimestamp(date)`

Converts `date` to `timestamp` format.

- `toUnixTimestamp(date)`

Converts `date` to `UNIX timestamp` format.

An example of the new functions creates a table and inserts various time-related values:

```
CREATE TABLE sample_times (a int, b timestamp, c timeuuid, d bigint,
PRIMARY KEY (a,b,c,d));
INSERT INTO sample_times (a,b,c,d) VALUES (1, toUnixTimestamp(now()),
50554d6e-29bb-11e5-b345-feff819cdc9f, toTimestamp(now()));
```

a b	c	d
1 2015-07-13 17:13:37-0700	50554d6e-29bb-11e5-b345-feff819cdc9f	1436832817476

Select data and convert it to a new format:

```
SELECT toDate(c) FROM sample_times;
```

a	b	system.todate(c)	system.todate(d)
1	2015-07-13 17:13:37-0700	2015-07-14	2015-07-14

Timestamp type

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

```
YYYY-mm-dd HH:mm
YYYY-mm-dd HH:mm:ss
YYYY-mm-dd HH:mmZ
YYYY-mm-dd HH:mm:ssZ
YYYY-mm-dd 'T'HH:mm
YYYY-mm-dd 'T'HH:mmZ
YYYY-mm-dd 'T'HH:mm:ss
YYYY-mm-dd 'T'HH:mm:ssZ
YYYY-mm-dd 'T'HH:mm:ss.fffffffZ
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, the date and time of Feb 3, 2011, 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, you can also omit the time of day. For example:

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

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

Timestamp output appears in the following format by default in `cqlsh`:

```
YYYY-mm-dd HH:mm:ssZ
```

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

```
[ui]
  datetimeformat = %Y-%m-%d %H:%M
```

Tuple type

The `tuple` data type holds fixed-length sets of typed positional fields. Use a `tuple` as an alternative to a user-defined type. A `tuple` can accommodate many fields (32768), more than can be prudently used. Typically, create a `tuple` with a few fields.

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

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

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

SELECT * FROM collect_things;

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

Note: Cassandra no longer requires the use of `frozen` for tuples:

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

You can filter a selection using a tuple.

```
CREATE INDEX on collect_things (v);

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

k | v
---+-----
```

```
0 | (3, 'bar', 2.1)
```

You can nest **tuples** as shown in the following example:

```
CREATE TABLE nested (k int PRIMARY KEY, t tuple <int, tuple<text,
double>>);

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

User-defined type

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

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

- [CREATE TYPE](#)
- [ALTER TYPE](#)
- [DROP TYPE](#)

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

- [DESCRIBE TYPE](#)
- [DESCRIBE TYPES](#)

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

Functions

CQL supports several functions that transform one or more column values into a new value. In addition, users can [define functions](#) and [aggregates](#). The native Cassandra functions are:

- [Blob conversion functions](#)
- [UUID and Timeuuid functions](#)

- [Token function](#)
- [WRITETIME function](#)
- [TTL function](#)
- Standard aggregate functions such as [MIN\(\)](#), [MAX\(\)](#), [SUM\(\)](#), and [AVG\(\)](#).
- [TOKEN function](#)

CQL limits

Observe the following upper limits:

- Cells in a partition: ~2 billion (2^{31}); single column value size: 2 GB (1 MB is recommended)
- Clustering column value, length of: 65535 ($2^{16}-1$)
- Key length: 65535 ($2^{16}-1$)
- Table / CF name length: 48 characters
- Keyspace name length: 48 characters
- Query parameters in a query: 65535 ($2^{16}-1$)
- Statements in a batch: 65535 ($2^{16}-1$)
- Fields in a tuple: 32768 (2^{15}) (just a few fields, such as 2-10, are recommended)
- Collection (List): collection limit: ~2 billion (2^{31}); values size: 65535 ($2^{16}-1$)
- Collection (Set): collection limit: ~2 billion (2^{31}); values size: 65535 ($2^{16}-1$)
- Collection (Map): collection limit: number of keys: 65535 ($2^{16}-1$); values size: 65535 ($2^{16}-1$)
- Blob size: 2 GB (less than 1 MB is recommended)

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

CQL shell commands

Important: The CQL shell commands described in this section work only within the [cqlsh shell](#) and are not accessible from drivers. CQL shell uses native protocol and the Datastax python driver to execute CQL commands on the connected Cassandra host. For configuration information, see the [cassandra.yaml](#) file.

Starting cqlsh

Execute the `cqlsh` Cassandra python script to start the CQL shell; CQL shell is a python-based command line client for executing [CQL commands](#) interactively. CQL shell supports tab completion.

Synopsis

```
bin/cqlsh [options] [host [port]]
```

Table 9. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key</i> : <i>value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syn-

Table 9. Legend (continued)

Syntax conventions	Description
	tax is useful when arguments might be mistaken for command line options.
' <schema> . . . </schema> '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@xml_entity='xml_entity_type'	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Table 10. Options

Short	Long	Description
	--version	cqlsh version number.
-h	--help	Help message.
-C	--color	Always use color output.
	--no-color	Never use color output.
	--browser=" <i>launch_browser_cmd %s</i> "	Browser to display the CQL command help. See Web Browser Control for a list of supported browsers. Replace the URL in the command with %s.
	--ssl	Use SSL.
-u <i>user_name</i>	--username=" <i>user_name</i> "	Connect with the user account.
-p <i>password</i>	--password=" <i>password</i> "	User's password.
-k <i>keyspace_name</i>	--keyspace= <i>keyspace_name</i>	Automatically switch to the keyspace.
-f <i>file_name</i>	--file= <i>file_name</i>	Execute commands from a CQL file, then exit. Note: After starting cqlsh, use the SOURCE command and the path to the file on the cqlsh command line.
	--debug	Show additional debugging information.
	--encoding=" <i>output_encoding</i> "	Output encoding. Default encoding: utf8.

Table 10. Options (continued)

Short	Long	Description
	--cqlshrc="/ <i>folder_name</i> "	Folder that contains the cqlshrc file. Use tilde (~) for paths relative to the user's home directory.
	--cqlversion=" <i>version_number</i> "	CQL version to use. Version displays after starting cqlsh.
-e " <i>cql_statement</i> "	--execute=" <i>cql_statement</i> "	Execute the CQL statement and exit. To direct the command output to a file see saving CQL output .
	--connect-timeout=" <i>timeout</i> "	Connection timeout in seconds; default: 5.
	--request-timeout=" <i>timeout</i> "	CQL request timeout in seconds; default: 10.
-t	--tty	Force time-to-live (tty) mode.

Connecting to a specific host or IP address

Specifying a hostname or IP address after the cqlsh command (and options) connects the CQL session to a specified Cassandra node. By default, CQL shell launches a session with the local host on 127.0.0.1. You can only connect CQL shell to remote hosts that have a higher or equal version than the local copy. When no port is specified, the connection uses the default port: 9042.

Examples

Starting the CQL shell

On startup, cqlsh shows the name of the cluster, IP address, and connection port. The cqlsh prompt initially is cqlsh>. After you specify a keyspace, it's added to the prompt.

1. Start the CQL shell:

```
bin/cqlsh
```

The host information appears.

```
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.3.0 | CQL spec 3.4.0 | Native protocol v4]
```

```
Use HELP for help.
```

2. Switch to the `cycling` keyspace:

```
USE cycling;
```

The prompt now includes the keyspace name.

```
cqlsh:cycling>
```

Querying using CQL commands

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.

```
SELECT * FROM calendar  
WHERE race_id = 201 ;
```

The results display in standard output.

race_id	race_start_date	race_end_date
201	2015-02-18 08:00:00.000000+0000	2015-02-22 08:00:00.000000+0000

Women's Tour of New Zealand

The [lexical structure of commands](#) includes how upper- and lower-case literals are treated in commands, when to use quotation marks in strings, and how to enter exponential notation.

Saving CQL output in a file

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

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

Setting the CQL help browser

Set the browser to display the CQL help to Chrome on Mac OS X:

```
bin/cqlsh --browser="/Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome %s"
```

```
[cqlsh 5.0.1 | Cassandra 3.9 | CQL spec 3.4.2 | Native protocol v4]
```

```
Use HELP for help.
```

```
HELP UPDATE
```

```
2016-11-11 13:59:22.068 Google Chrome[89120:2909863] NSWindow warning:  
adding an unknown subview: <FullSizeContentView: 0x7fa35ae9af40>. Break  
on NSLog to debug.  
2016-11-11 13:59:22.069 Google Chrome[89120:2909863] Call stack:  
(  
    "+callStackSymbols disabled for performance reasons"  
)
```

Note: cqlsh help displays in the terminal. CQL help is only available online in HTML.

Connecting to a remote node

Specify a remote node IP address:

```
bin/cqlsh 10.0.0.30
```

```
Connected to West CS Cluster at 10.0.0.30:9042.  
[cqlsh 5.0.1 | Cassandra 3.3.0 | CQL spec 3.4.0 | Native protocol v4]  
Use HELP for help.
```

Configuring cqlsh from a file

The `cqlshrc` file configures the `cqlsh` session when starting the utility. Use the file by default by saving it in the `~/.cassandra` directory on the local computer or specify the directory that the file is in with the `--cqlshrc` option. Only one `cqlshrc` per directory.

Pre-configure the following options:

- [Automatically logging in and selecting a keyspace](#)
- [Changing the CQL shell display](#)
- [Forcing the CQL version](#)
- [Connecting to a CQL host](#)
- [Limiting the field size](#)
- [Setting tracing timeout](#)
- [Configuring SSL](#)
- [Overriding SSL local settings](#)
- [Setting common COPY TO and COPY FROM options](#)

- Setting COPY TO specific options
- Setting COPY FROM specific options
- Setting table specific COPY TO/FROM options

Synopsis

```
./cqlsh CQLSHRC="~/directory_name"
```

Note: Tilde (~) expands to the user's home directory; you can also specify the absolute path, for example /Users/jdoe/cqlshprofiles/west.

Table 11. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (. . .) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key : value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.

Table 11. Legend (continued)

Syntax conventions	Description
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' <schema> . . . </schema> '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@xml_entity='xml_entity_type'	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Using the cqlshrc.sample

A sample file is installed with Cassandra, [cqlshrc.sample](#). The file contains all the available settings. Some settings are commented out using a single semi-colon.

To use this file:

1. Copy the file to the [home](#) directory.
2. Rename it `cqlshrc`.
3. Remove the semi-colon to uncomment an option (options must be in brackets) and corresponding settings, for example to import all CSV without a header row, uncomment `[copy]` and `header = false`:

```
;; Options that are common to both COPY TO and COPY FROM
[copy]

;; The string placeholder for null values
; nullval = null

;; For COPY TO, controls whether the first line in the CSV output
;; file will
;; contain the column names. For COPY FROM, specifies whether the
;; first
;; line in the CSV file contains column names.
header = false
```

4. Restart the CQL shell.

Automatically logging in and selecting a keyspace

Set up credentials to automatically log in when CQL shell starts and/or choose a keyspace.

Note: Only set a user name and password for hosts that use Cassandra internal authentication, see [Encrypting Cassandra with SSL](#).

[authentication]

username

Log in account name.

password

Log in password.

keyspace

Optional. Opens the specified keyspace. Equivalent to issuing a [USE keyspace](#) command immediately after starting cqlsh. (Does not require internal Cassandra authentication).

Changing the CQL shell display

The cqlsh console display and COPY TO date parsing settings.

[ui]

color

Shows query results with color.

`on` In color.

`off` No color.

datetimeformat

Configure the format of timestamps using [Python strftime](#) syntax.

timezone

Display timestamps in Etc/UTC.

float_precision, double_precision

Sets the number of digits displayed after the decimal point for single and double precision numbers.

Note: Increasing this to large numbers can result in unusual values.

completekey

Set the key for automatic completion of a cqlsh shell entry. Default is the tab key.

encoding

The encoding used for characters. The default is UTF8.

browser

Sets the browser for cqlsh help. If the value is not specified, cqlsh uses the default browser. Available browsers are those supported by the Python [webbrowser module](#). For example, to use Google Chrome:

- Mac OSX: `browser = open -a /Applications/Google\ Chrome.app %s`
- Linux: `browser = /usr/bin/google-chrome-stable %s`

This setting can be overridden with the `--browser` command line option.

Forcing the CQL version

Use the specified version of CQL only.

[cql]

version

Only use the specified version of CQL.

Connecting to a CQL host

Specify the host and connection details for the CQL shell session.

[connection]

hostname

The host for the cqlsh connection.

port

The connection port. Default: 9042 (native protocol).

ssl

Always connect using SSL. Default: `false`.

timeout

Configures timeout in seconds when opening new connections.

request_timeout

Configures the request timeout in seconds for executing queries. Set the number of seconds of inactivity.

Limiting the field size

[csv]

field_size_limit

Set to a particular field size, such as `field_size_limit = 1000000000`.

Setting tracing timeout

Specify the wait time for tracing.

[tracing]**max_trace_wait**

The maximum number of seconds to wait for a trace to complete.

Configuring SSL

Specify connection SSL settings.

[ssl]**certfile**

The path to the cassandra certificate. See *Using cqlsh with SSL encryption* in the Cassandra documentation ([links above](#)).

validate

Optional. Default: `true`.

userkey

Must be provided when `require_client_auth=true` in [cassandra.yaml](#).

usercert

Must be provided when `require_client_auth=true` in [cassandra.yaml](#).

Overriding SSL local settings

Overrides default `certfiles` in [ssl] section. Create an entry for each remote host.

[certfiles]**remote_host=path_to_cert**

Specify the IP address or remote host name and path to the certificate file on your local system.

Setting common COPY TO and COPY FROM options

Settings common to both `COPY TO` and `COPY FROM`.

Also see the [COPY](#) table.

[copy]**nullval**

The string placeholder for null values.

header

For `COPY TO`, controls whether the first line in the CSV output file contains the column names.

For `COPY FROM`, specifies whether the first line in the CSV file contains column names.

decimalsep

Separator for decimal values. Default value: period (.).

thousandssep

Separator for thousands digit groups. Default value: `None`. Default: empty string.

boolstyle

Boolean indicators for True and False. The values are case insensitive, for example: yes,no and YES,NO are the same. Default values: `True, False`.

numprocesses

Sets the number of child worker processes.

maxattempts

Maximum number of attempts for errors. Default value: 5.

reportfrequency

Frequency with which status is displayed in seconds. Default value: 0.25.

ratefile

Print output statistics to this file.

Setting COPY TO specific options**[copy-to]****maxrequests**

Maximum number of requests each worker can process in parallel.

pagesize

Page size for fetching results.

pagetimeout

Page timeout for fetching results.

begintoken

Minimum token string for exporting data.

endtoken

Maximum token string for exporting data.

maxoutputsize

Maximum size of the output file, measured in number of lines. When set, the output file is split into segment when the value is exceeded. Use "-1" for no maximum.

encoding

The encoding used for characters. The default is UTF8.

Setting COPY FROM specific options

[copy-from]

ingestrate

Approximate ingest rate in rows per second. Must be greater than the chunk size.

maxrows

Maximum number of rows. Use "-1" for no maximum.

skiprows

Number of rows to skip.

skipcols

Comma-separated list of column names to skip.

maxPARSEerrors

Maximum global number of parsing errors. Use "-1" for no maximum.

maxINSErTerrors

Maximum global number of insert errors. Use "-1" for no maximum.

errfile

File to store all rows that are not imported. If no value is set, the information is stored in `import_ks_table.err` where `ks` is the keyspace and `table` is the table name.

maxbatchsize

Maximum size of an import batch.

minbatchsize

Minimum size of an import batch.

chunksize

Chunk size passed to worker processes. Default value: 1000

Setting table specific COPY TO/FROM options

Use these options to configure table specific settings; create a new entry for each table, for example to set the chunk size for cyclist names and rank:

```
[copy:cycling.cyclist_names]  
chunksize = 1000
```

```
[copy:cycling.rank_by_year_and_name]  
chunksize = 10000
```

[copy:keyspace_name.table_name]

chunksize

Chunk size passed to worker processes. Default value: 1000

[copy-from:*keyspace_name.table_name*]

ingestrate

Approximate ingest rate in rows per second. Must be greater than the chunk size.

[copy-to:*keyspace_name.table_name*]

pagetimeout

Page timeout for fetching results.

CAPTURE

Appends the query results to a file in exponential notation format. Results do not appear in standard output; however error messages and cqlsh commands output displays in STDOUT.

Synopsis

```
CAPTURE ['file_name' | OFF]
```

Table 12. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.

Table 12. Legend (continued)

Syntax conventions	Description
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key : value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > . . . </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@ <i>xml_entity='xml_entity_type'</i>	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Options

Option	Description
	Shows status.
OFF	Stops capture
' <i>file_name</i> '	Starts capture for CQL queries. Results of queries that run after capture begins are appended to the specified file; when you run the first query after beginning the capture, the file is created if it does not already exist. Use a relative path from the current working directory or specify tilde (~) for the user's HOME directory. Absolute paths are not supported.

Examples

Begin capturing results to the winners text file:

```
CAPTURE '~/results/winners.txt'
```

Note: The folder must exist in the user's home directory, but the file is created if it does not exist.

```
Now capturing query output to
'/Users/local_system_user/results/winners.txt'.
```

Execute a query that selects all winners:

```
select * from cycling.race_winners;
```

Results are written to end of the capture file and do not display in the terminal window.

CLEAR

CLEAR (shorthand: CLS) clears the CQL shell terminal window. To clear with keystrokes use CTRL+L.

Synopsis

```
CLEAR
```

Table 13. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.

Table 13. Legend (continued)

Syntax conventions	Description
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key : value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2</i> >	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > . . . </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@ <i>xml_entity</i> =' <i>xml_entity_type</i> '	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Example

Clear the previous requests in the terminal window:

```
CLEAR
```

CONSISTENCY

Consistency level determines how many nodes in the replica must respond for the coordinator node to successfully process a non-lightweight transaction.

Restriction: CQL shell only supports **read** requests (SELECT statements) when the consistency level is set to SERIAL or LOCAL_SERIAL. For more information, see [Data consistency](#). Set level for LWT, write requests that contain `IF EXISTS` or `IF NOT EXISTS`, using [SERIAL CONSISTENCY](#).

Synopsis

```
CONSISTENCY [ level ]
```

Table 14. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key : value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > ... </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.

Table 14. Legend (continued)

Syntax conventions	Description
<code>@xml_entity='xml_entity_type'</code>	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Displaying the current level

Use consistency with no options, to show the current consistency level:

```
CONSISTENCY
```

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

The default CQL shell level is ONE.

Setting a level

Level determines data availability versus data accuracy for transactions during the CQL shell session, some settings also may have high impact other transactions occurring in the cluster, such as ALL and SERIAL. The CQL shell setting supersedes Cassandra consistency level global setting.

Important: Before changing this setting it is important to understand [How Cassandra reads and writes data](#), [Data replication strategy](#), [How quorum is calculated](#), and partition keys.

When you initiate a transaction from the CQL shell, the coordinator node is typically the node where you started cqlsh; if you connect to a remote host, then the remote node is the coordinator.

Table 15. Read Consistency Levels

Level	Replicas	Consistency	Availability
ALL	All	Highest	Lowest
EACH_QUORUM	Quorum in each data centers . <i>Reads NOT supported.</i>	Same across data centers	
QUORUM	Quorum of all nodes across all data centers . Some level of failure.		
LOCAL_QUORUM	Quorum of replicas in the same data center as the coordinator . Avoids latency of inter-datacenter communication.	Low in multi-data center	

Table 15. Read Consistency Levels (continued)

Level	Replicas	Consistency	Availability
ONE	Closest replica as determined by the snitch . Satisfies the needs of most users because consistency requirements are not stringent.	Lowest (READ)	Highest (READ)
TWO	Closest two replicas as determined by the snitch .		
THREE	Closest three replicas as determined by the snitch .		
LOCAL_ONE	Returns a response from the closest replica in the local data center. For security and quality, use in an off line datacenter to prevent automatic connection to online nodes in other datacenters.		
ANY	Closest replica, as determined by the snitch . If all replica nodes are down, write succeeds after a hinted handoff . Provides low latency, guarantees writes never fail.	Lowest (WRITE)	Highest (WRITE)
SERIAL	Returns results with the most recent data including inflight LWT (uncommitted). Commits inflight LWT as part of the read. <i>Writes NOT supported.</i>		
LOCAL_SERIAL	Same as SERIAL, but confined to the data center. <i>Writes NOT supported.</i>		

Restriction: SERIAL and LOCAL_SERIAL settings support read transactions.

Examples

Set `CONSISTENCY` to force the majority of the nodes to respond:

```
CONSISTENCY QUORUM
```

Set level to serial for LWT read requests:

```
CONSISTENCY SERIAL
```

```
Consistency level set to SERIAL.
```

List all winners.

```
SELECT * FROM cycling.race_winners ;
```

The results are shown; this example uses expand ON.

```
@ Row 1
-----
race_name      | National Championships South Africa WJ-ITT (CN)
race_position  | 1
cyclist_name   | {firstname: 'Frances', lastname: 'DU TOUT'}
```



```
@ Row 2
-----
race_name      | National Championships South Africa WJ-ITT (CN)
race_position  | 2
cyclist_name   | {firstname: 'Lynette', lastname: 'BENSON'}
```



```
@ Row 3
-----
race_name      | National Championships South Africa WJ-ITT (CN)
race_position  | 3
cyclist_name   | {firstname: 'Anja', lastname: 'GERBER'}
```



```
@ Row 4
-----
race_name      | National Championships South Africa WJ-ITT (CN)
race_position  | 4
cyclist_name   | {firstname: 'Ame', lastname: 'VENTER'}
```



```
@ Row 5
-----
race_name      | National Championships South Africa WJ-ITT (CN)
race_position  | 5
cyclist_name   | {firstname: 'Danielle', lastname: 'VAN NIEKERK'}
```

```
(5 rows)
```

Note: The query format above uses `expand ON` for legibility.

Inserts with CONSISTENCY SERIAL fail:

```
INSERT INTO cycling.race_winners (
    race_name ,
    race_position ,
    cyclist_name
)
VALUES (
    'National Championships South Africa WJ-ITT (CN)' ,
    7 ,
    { firstname: 'Joe' , lastname: 'Anderson' }
)
IF NOT EXISTS ;
```

```
InvalidRequest: Error from server: code=2200 [Invalid query]
message="LOCAL_SERIAL is not supported as conditional update commit
consistency. Use ANY if you mean "make sure it is accepted but I don't
care how many replicas commit it for non-SERIAL reads""
```

Updates with CONSISTENCY SERIAL also fail:

```
UPDATE cycling.race_winners SET
    cyclist_name = { firstname: 'JOHN' , lastname: 'DOE' }
WHERE
    race_name='National Championships South Africa WJ-ITT (CN)'
    AND race_position = 6
IF EXISTS ;
```

```
InvalidRequest: Error from server: code=2200 [Invalid query]
message="LOCAL_SERIAL is not supported as conditional update commit
consistency. Use ANY if you mean "make sure it is accepted but I don't
care how many replicas commit it for non-SERIAL reads""
```

COPY

CQL shell commands that import and export CSV (comma-separated values or delimited text files).

- COPY TO exports data from a table into a CSV file. Each row is written to a line in the target file with fields separated by the delimiter. All fields are exported when no column names are specified. To drop columns, specify a column list.
- COPY FROM imports data from a CSV file into an existing table. Each line in the source file is imported as a row. All rows in the dataset must contain the same

number of fields and have values in the PRIMARY KEY fields. The process verifies the PRIMARY KEY and updates existing records. If `HEADER = false` and no column names are specified, the fields are imported in deterministic order. When column names are specified, fields are imported in that order. Missing and empty fields are set to null. The source cannot have more fields than the target table, however it can have fewer fields.

Note: Only use COPY FROM to import datasets that have less than 2 million rows. To import large datasets, use the [Cassandra bulk loader](#).

Synopsis

```
COPY table_name [( column_list )]
  FROM 'file_name'[, 'file2_name', ...] | STDIN
  [WITH option = 'value' [AND ...]]
```

```
COPY table_name [( column_list )]
  TO 'file_name'[, 'file2_name', ...] | STDOUT
  [WITH option = 'value' [AND ...]]
```

Note: COPY supports a list of one or more comma-separated file names or python glob expressions.

Table 16. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.

Table 16. Legend (continued)

Syntax conventions	Description
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key : value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > . . . </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@ <i>xml_entity</i> =' <i>xml_entity_type</i> '	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Setting copy options

Copy options set in the statement take precedence over `cqlshrc` file and the default settings. If an option is not set on the command line, the `cqlshrc` file takes precedence over the default settings.

Table 17. Options

Command	Options	Description
TO/FROM	DELIMITER	Single character used to separate fields. Default value: ,.
TO/FROM	QUOTE	Single character that encloses field values. Default value: ".
TO/FROM	ESCAPE	Single character that escapes literal uses of the QUOTE character. Default value: \.
TO/FROM	HEADER	Boolean (true false) that indicates column names on the first line. True indicates that first line has

Table 17. Options (continued)

Command	Options	Description
		column names, False indicates that the first line does not have column names. Default value: <code>false</code> .
TO/FROM	NULL	No value in field. Default value is an empty string <code>()</code> .
TO/FROM	DATETIMEFORMAT	Time format for reading or writing CSV time data. The timestamp uses the <code>strftime</code> format. If not set, the default value is set to the <code>datetimeformat</code> value in the <code>cqlshrc</code> file. Default format: <code>%Y-%m-%d %H:%M:%S%z</code> .
TO/FROM	MAXATTEMPTS	Maximum number of attempts for errors. Default value: 5.
TO/FROM	REPORTFREQUENCY	Frequency with which status is displayed in seconds. Default value: <code>0 . 25</code> .
TO/FROM	DECIMALSEP	Separator for decimal values. Default value: period <code>(.)</code> .
TO/FROM	THOUSANDSSEP	Separator for thousands digit groups. Default value: <code>None</code> .
TO/FROM	BOOLSTYLE	Boolean indicators for True and False. The values are case insensitive, for example: yes,no and YES,NO are the same. Default values: <code>True , False</code> .
TO/FROM	NUMPROCESSES	Number of worker processes. Maximum value is 16. Default value: <code>-1</code> .
TO/FROM	CONFIGFILE	Specify a <code>cqlshrc</code> configuration file to set WITH options. Note: Command line options always override the <code>cqlshrc</code> file.
TO/FROM	RATEFILE	Print output statistics to this file.
FROM	CHUNKSIZE	Chunk size passed to worker processes. Default value: 1000
FROM	INGESTRATE	Approximate ingest rate in rows per second. Must be greater than the chunk size. Default value: <code>100000</code>

Table 17. Options (continued)

Command	Options	Description
FROM	MAXBATCHSIZE	Maximum size of an import batch. Default value:20
FROM	MINBATCHSIZE	Minimum size of an import batch. Default value:2
FROM	MAXROWS	Maximum number of rows. Use "-1" for no maximum. Default value:-1
FROM	SKIPROWS	Number of rows to skip. Default value:0
FROM	SKIPCOLS	Comma-separated list of column names to skip.
FROM	MAXPARSEERRORS	Maximum global number of parsing errors. Use "-1" for no maximum. Default value:-1
FROM	MAXINSERTERRORS	Maximum global number of insert errors. Use "-1" for no maximum. Default value:-1
FROM	ERRFILE	File to store all rows that are not imported. If no value is set, the information is stored in <code>import_ks_table.err</code> where <code>ks</code> is the keyspace and <code>table</code> is the table name.
FROM	TTL	Time to live in seconds. By default, data will not expire. Default value:3600
TO	ENCODING	Output string type. Default value: UTF8.
TO	PAGESIZE	Page size for fetching results. Default value: 1000.
TO	PAGETIMEOUT	Page timeout for fetching results. Default value: 10.
TO	BEGINTOKEN	Minimum token string for exporting data. Default value: .
TO	ENDTOKEN	Maximum token string for exporting data.
TO	MAXREQUESTS	Maximum number of requests each worker can process in parallel. Default value: 6.
TO	MAXOUTPUTSIZE	Maximum size of the output file, measured in number of lines. When set, the output file is split into segments when the value is exceeded. Use "-1" for no maximum. Default value: -1.

Examples

Create the sample dataset

Set up the environment used for the COPY command examples.

1. Using CQL, create a cycling keyspace:

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

2. Create the cycling.cyclist_name table:

```
CREATE TABLE cycling.cyclist_name (
    id UUID PRIMARY KEY,
    lastname text,
    firstname text
) ;
```

3. Insert data into cycling.cyclist_name:

```
INSERT INTO cycling.cyclist_name (id, lastname, firstname)
    VALUES (5b6962dd-3f90-4c93-8f61-eabfa4a803e2, 'VOS', 'Marianne');
INSERT INTO cycling.cyclist_name (id, lastname, firstname)
    VALUES (e7cd5752-bc0d-4157-a80f-7523add8dbcd, 'VAN DER BREGGEN', 'Anna');
INSERT INTO cycling.cyclist_name (id, lastname, firstname)
    VALUES (e7ae5cf3-d358-4d99-b900-85902fda9bb0, 'FRAME', 'Alex');
INSERT INTO cycling.cyclist_name (id, lastname, firstname)
    VALUES (220844bf-4860-49d6-9a4b-6b5d3a79cbfb, 'TIRALONGO', 'Paolo');
INSERT INTO cycling.cyclist_name (id, lastname, firstname)
    VALUES (6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47, 'KRIJKSWIJK', 'Steven');
INSERT INTO cycling.cyclist_name (id, lastname, firstname)
    VALUES (fb372533-eb95-4bb4-8685-6ef61e994caa, 'MATTHEWS', 'Michael');
```

Export and import data from the cyclist_name table

Round trip the cycling names data.

1. Export only the id and lastname columns from the cyclist_name table to a CSV file.

```
COPY cycling.cyclist_name (id,lastname)
TO '../cyclist_lastname.csv' WITH HEADER = TRUE ;
```

The cyclist_lastname.csv file is created in the directory above the current working directory. If the file already exists it is overwritten.

```
Using 7 child processes
```

```
Starting copy of cycling.cyclist_name with columns [id, lastname].
Processed: 6 rows; Rate:      29 rows/s; Avg. rate:      29 rows/s
6 rows exported to 1 files in 0.223 seconds.
```

2. Copy the id and first name to a different CSV file.

```
COPY cycling.cyclist_name (id,firstname)
TO '../cyclist_firstname.csv' WITH HEADER = TRUE ;
```

The first name file is created.

```
Using 7 child processes
```

```
Starting copy of cycling.cyclist_name with columns [id, firstname].
Processed: 6 rows; Rate:      30 rows/s; Avg. rate:      30 rows/s
6 rows exported to 1 files in 0.213 seconds.
```

3. Remove all records from the cyclist name table.

```
TRUNCATE cycling.cyclist_name ;
```

4. Verify that there are no rows.

```
SELECT * FROM cycling.cyclist_name ;
```

Query results are empty.

id	firstname	lastname
(0 rows)		

5. Import the cyclist firstnames.

```
COPY cycling.cyclist_name (id,firstname) FROM
'../cyclist_firstname.csv' WITH HEADER = TRUE ;
```

The rows are imported. Since the lastname was not in the dataset it is set to null for all rows.

```
Using 7 child processes
```

```
Starting copy of cycling.cyclist_name with columns [id, firstname].
Processed: 6 rows; Rate:      10 rows/s; Avg. rate:      14 rows/s
```

```
6 rows imported from 1 files in 0.423 seconds (0 skipped).
```

6. Verify the new rows.

```
SELECT * FROM cycling.cyclist_name ;
```

The rows were created with null last names because the field was not in the imported data set.

id	firstname	lastname
e7ae5cf3-d358-4d99-b900-85902fda9bb0	Alex	null
fb372533-eb95-4bb4-8685-6ef61e994caa	Michael	null
5b6962dd-3f90-4c93-8f61-eabfa4a803e2	Marianne	null
220844bf-4860-49d6-9a4b-6b5d3a79cbfb	Paolo	null
6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47	Steven	null
e7cd5752-bc0d-4157-a80f-7523add8dbcd	Anna	null

(6 rows)

7. Import the last names.

```
COPY cycling.cyclist_name (id,lastname) FROM
'../cyclist_lastname.csv' WITH HEADER = TRUE ;
```

The records are imported but no new records get created.

```
Using 7 child processes

Starting copy of cycling.cyclist_name with columns [id, lastname].
Processed: 6 rows; Rate:      10 rows/s; Avg. rate:      14 rows/s
6 rows imported from 1 files in 0.422 seconds (0 skipped).
```

8. Verify the that the records were updated.

```
SELECT * FROM cycling.cyclist_name ;
```

PRIMARY KEY, id, matched for all records and the last name is updated.

id	firstname	lastname
e7ae5cf3-d358-4d99-b900-85902fda9bb0	Alex	FRAME
fb372533-eb95-4bb4-8685-6ef61e994caa	Michael	MATTHEWS
5b6962dd-3f90-4c93-8f61-eabfa4a803e2	Marianne	VOS
220844bf-4860-49d6-9a4b-6b5d3a79cbfb	Paolo	TIRALONGO
6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47	Steven	KRUIKSWIJK

e7cd5752-bc0d-4157-a80f-7523add8dbcd	Anna	VAN DER BREGGEN
--------------------------------------	------	-----------------

Copy data from standard input to a table.

1. Clear the data from the cyclist_name table.

```
TRUNCATE cycling.cyclist_name ;
```

2. Start the copy input operation using STDIN option.

```
COPY cycling.cyclist_name FROM STDIN ;
```

The line prompt changes to [COPY].

```
Using 7 child processes

Starting copy of cycling.cyclist_name with columns [id, firstname,
lastname].
[Use . on a line by itself to end input]
[copy]
```

3. Next to prompt enter the field values in a common separated list; on the last line of data enter a period.

```
[copy] e7cd5752-bc0d-4157-a80f-7523add8dbcd,Anna,VAN DER BREGGEN
[copy] .
```

4. Press Return (or Enter) after inserting a period on the last line to begin processing the records.

```
Processed: 1 rows; Rate: 0 rows/s; Avg. rate: 0 rows/s
1 rows imported from 1 files in 36.991 seconds (0 skipped).
```

5. Verify that the records were imported.

```
select * FROM cycling.cyclist_name ;
```

id	firstname	lastname
e7cd5752-bc0d-4157-a80f-7523add8dbcd	Anna	VAN DER BREGGEN

(1 rows)

DESCRIBE

DESCRIBE (shorthand: DESC) outputs detailed information in CQL format that you can run.

CAUTION: Verify all settings before executing the full output, some options may be cluster specific in the WITH statement.

Synopsis

```
DESCRIBE [ FULL ] SCHEMA | CLUSTER
| KEYSPACES | KEYSPACE keyspace_name
| TABLES | TABLE [keyspace_name.]table_name
| TYPES | TYPE [keyspace_name.]udt_name
| FUNCTIONS | FUNCTION [keyspace_name.]udf_name
| AGGREGATES | AGGREGATE [keyspace_name.]uda_name
| INDEX [keyspace_name.]index_name
| MATERIALIZED VIEW [keyspace_name.]view_name
```

Table 18. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.

Table 18. Legend (continued)

Syntax conventions	Description
{ key : value }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
<datatype1,datatype2>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
cql_statement ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' <schema> . . . </schema> '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@xml_entity='xml_entity_type'	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Note: On Linux systems object names, such as keyspace names, table names, and so forth are case sensitive. By default, CQL converts names to lowercase unless enclosed in double quotes. Names on Windows are case insensitive.

Options	Description
CLUSTER	Cluster information including cluster name, partitioner, and snitch and for non-system keyspaces, the endpoint-range ownership information is also shown.
FULL SCHEMA	Details for all objects in the cluster.
SCHEMA	Details for all non-system objects in the cluster.
KEYSPACES	List of all keyspace names on the cluster.
KEYSPACE <i>keyspace-name</i>	Details for the specified keyspace and objects it contains.
TABLES	List of tables in the current keyspace or all tables in the cluster when no keyspace is selected.
TABLE <i>keyspace-name.table_name</i>	Details on the specified table.

Options	Description
	Note: To query the system tables , use SELECT.
INDEX <i>keyspace.table</i>	Details on the specified index.
TYPES	List of user-defined types in the current keyspace or all user-defined types in the cluster when no keyspace is selected.
TYPE <i>keyspace-name.type_name</i>	Details on the specified user-defined type.
FUNCTIONS	List of user-defined functions in the current keyspace or all user-defined functions in the cluster when no keyspace is selected.
FUNCTION <i>keyspace-name.function_name</i>	Details on the specified user-defined function.
AGGREGATES	List of user-defined aggregates in the current keyspace or all user-defined aggregates in the cluster when no keyspace is selected.
AGGREGATE <i>keyspace-name.aggregate_name</i>	Details on the specified user-defined aggregate.
MATERIALIZED VIEW <i>keyspace_name.view_name</i>	Details on the specified materialized view.

Examples

Show all keyspaces:

```
DESC keyspaces
```

All the keyspaces on the cluster are listed.

```
test_cycling    system_auth      test
cycling        system_schema   system      system_distributed  system_traces
```

Show details for the Cycling Calendar table:

```
DESC cycling.calendar
```

A complete table description in CQL that can be used to recreate the table is returned.

```
CREATE TABLE cycling.calendar (
    race_id int,
    race_start_date timestamp,
```

```

race_end_date timestamp,
race_name text,
PRIMARY KEY (race_id, race_start_date, race_end_date)
) WITH CLUSTERING ORDER BY (race_start_date ASC, race_end_date ASC)
AND bloom_filter_fp_chance = 0.01
AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
AND comment = ''
AND compaction = {'class':
'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy',
'max_threshold': '32', 'min_threshold': '4'}
    AND compression = {'chunk_length_in_kb': '64', 'class':
'org.apache.cassandra.io.compress.LZ4Compressor'}
        AND crc_check_chance = 1.0
        AND dclocal_read_repair_chance = 0.1
        AND default_time_to_live = 0
        AND gc_grace_seconds = 864000
        AND max_index_interval = 2048
        AND memtable_flush_period_in_ms = 0
        AND min_index_interval = 128
        AND read_repair_chance = 0.0
        AND speculative_retry = '99PERCENTILE';

```

cqlshExpand

For each row, lists column values vertically; use to read wide data.

Synopsis

```
cqlshExpand [ ON | OFF ]
```

Table 19. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.

Table 19. Legend (continued)

Syntax conventions	Description
<code>...</code>	Repeatable. An ellipsis (<code>...</code>) indicates that you can repeat the syntax element as often as required.
<code>'Literal string'</code>	Single quotation (<code>'</code>) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
<code>{ key : value }</code>	Map collection. Braces (<code>{ }</code>) enclose map collections or key value pairs. A colon separates the key and the value.
<code><datatype1,datatype2></code>	Set, list, map, or tuple. Angle brackets (<code>< ></code>) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<code>cql_statement ;</code>	End CQL statement. A semicolon (<code>;</code>) terminates all CQL statements.
<code>[--]</code>	Separate the command line options from the command arguments with two hyphens (<code>--</code>). This syntax is useful when arguments might be mistaken for command line options.
<code>' <schema> ... </schema> '</code>	Search CQL only: Single quotation marks (<code>'</code>) surround an entire XML schema declaration.
<code>@xml_entity='xml_entity_type'</code>	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Examples

View rows vertically.

```
cqlshExpand ON
```

The view is enabled.

```
Now printing cqlshExpanded output
```

Select all from race winners table.

```
select * from cycling.race_winners ;
```

Each field is shown in a vertical row table.

```
@ Row 1
-----
race_name      | National Championships South Africa WJ-ITT (CN)
race_position  | 1
cyclist_name   | {firstname: 'Frances', lastname: 'DU TOUT' }

@ Row 2
-----
race_name      | National Championships South Africa WJ-ITT (CN)
race_position  | 2
cyclist_name   | {firstname: 'Lynette', lastname: 'BENSON' }

@ Row 3
-----
race_name      | National Championships South Africa WJ-ITT (CN)
race_position  | 3
cyclist_name   | {firstname: 'Anja', lastname: 'GERBER' }

@ Row 4
-----
race_name      | National Championships South Africa WJ-ITT (CN)
race_position  | 4
cyclist_name   | {firstname: 'Ame', lastname: 'VENTER' }

@ Row 5
-----
race_name      | National Championships South Africa WJ-ITT (CN)
race_position  | 5
cyclist_name   | {firstname: 'Danielle', lastname: 'VAN NIEKERK' }
```

(5 rows)

EXIT

Terminates the CQL shell.

Synopsis

EXIT

Table 20. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key</i> : <i>value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syn-

Table 20. Legend (continued)

Syntax conventions	Description
	tax is useful when arguments might be mistaken for command line options.
' <schema> . . . </schema> '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@xml_entity='xml_entity_type'	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Example

End the CQL shell and return to the system command prompt.

```
EXIT
```

LOGIN

Switches user accounts without ending the CQL shell session.

Log in as different user by specifying credentials. A password entered directly into the login command appears in plain text. To securely enter a password, use the user name only.

Synopsis

```
LOGIN user_name [password]
```

Table 21. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.

Table 21. Legend (continued)

Syntax conventions	Description
...	Repeatable. An ellipsis (. . .) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key</i> : <i>value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > . . . </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@ <i>xml_entity</i> =' <i>xml_entity_type</i> '	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Examples

Log in as the cycling admin.

```
LOGIN cycling_admin
```

When no password is specified a password prompt appears.

```
password: *****
```

PAGING

PAGING ON displays query results in 100-line chunks followed by the *more* prompt. Press the space bar to move to the next chunk. Disabled (OFF) displays the entire results. PAGING without an option shows the current status (ON or OFF).

Synopsis

PAGING [ON | OFF]

Table 22. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key : value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > ... </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.

Table 22. Legend (continued)

Syntax conventions	Description
<code>@xml_entity='xml_entity_type'</code>	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Examples

Show PAGING status:

```
PAGING
```

Reports the current status and size.

```
Query paging is currently enabled. Use PAGING OFF to disable  
Page size: 100
```

SERIAL CONSISTENCY

Sets consistency for lightweight transaction. LWT use IF EXISTS and IF NOT EXISTS. Valid values are: `SERIAL` and `LOCAL_SERIAL`.

Tip: To set the consistency level of non-lightweight transaction, see [CONSISTENCY](#).

Synopsis

```
SERIAL CONSISTENCY [level]
```

Table 23. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
<code>[]</code>	Optional. Square brackets (<code>[]</code>) surround optional command arguments. Do not type the square brackets.
<code>()</code>	Group. Parentheses (<code>()</code>) identify a group to choose from. Do not type the parentheses.
<code> </code>	Or. A vertical bar (<code> </code>) separates alternative elements. Type any one of the elements. Do not type the vertical bar.

Table 23. Legend (continued)

Syntax conventions	Description
<code>...</code>	Repeatable. An ellipsis (<code>...</code>) indicates that you can repeat the syntax element as often as required.
<code>'Literal string'</code>	Single quotation (<code>'</code>) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
<code>{ key : value }</code>	Map collection. Braces (<code>{ }</code>) enclose map collections or key value pairs. A colon separates the key and the value.
<code><datatype1,datatype2></code>	Set, list, map, or tuple. Angle brackets (<code>< ></code>) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<code>cql_statement ;</code>	End CQL statement. A semicolon (<code>;</code>) terminates all CQL statements.
<code>[--]</code>	Separate the command line options from the command arguments with two hyphens (<code>--</code>). This syntax is useful when arguments might be mistaken for command line options.
<code>' <schema> ... </schema> '</code>	Search CQL only: Single quotation marks (<code>'</code>) surround an entire XML schema declaration.
<code>@xml_entity='xml_entity_type'</code>	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Examples

Display current SERIAL CONSISTENCY status.

```
SERIAL CONSISTENCY
```

Reports the current setting.

```
Current serial consistency level set to SERIAL.
```

Set the serial consistency level with a value.

```
SERIAL CONSISTENCY LOCAL_SERIAL
```

Confirms the level is set.

```
Serial consistency level set to LOCAL_SERIAL.
```

Note: Trace transactions to compare the difference between INSERT statements with and without IF EXISTS.

Write data using IF NOT EXISTS.

```
INSERT INTO cycling.cyclist_name (id, firstname , lastname )
    VALUES (e7ae5cf3-d358-4d99-b900-85902fda9bb0,'Alex','FRAME' )
    IF NOT EXISTS ;
```

Since the record already exists, the insert is not applied.

[applied]	id	firstname	lastname
False	e7ae5cf3-d358-4d99-b900-85902fda9bb0	Alex	FRAME

SHOW

Display the Cassandra instance version, the session CQL and cqlsh versions, current session node information, and tracing session details captured in the past 24 hours.

Synopsis

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

Table 24. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.

Table 24. Legend (continued)

Syntax conventions	Description
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key : value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > ... </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@ <i>xml_entity</i> =' <i>xml_entity_type</i> '	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Option	Description
VERSION	The version, build number, and native protocol of the connected Cassandra instance, as well as the CQL specification version for cqlsh.
HOST	The node details for the cqlsh session host.
SESSION <i>tracing_session_id</i>	<p>Request activity details for a specific query. Session ids are shown in the query results and are recorded to the system_traces.sessions table.</p> <p>Note: All queries run from a TRACING enabled cqlsh session are captured in the session and events table and saved for 24 hours. After that time, the tracing information time-to-live expires.</p>

Examples

Show the version information.

```
SHOW VERSION
```

Reports the current versions for cqlsh, Cassandra, CQL and the native protocol.

```
[cqlsh 5.0.1 | Cassandra 2.1.0 | CQL spec 3.3 | Native protocol v3]
```

Show the host information for the cqlsh session host.

```
SHOW HOST
```

Displays the host name, IP address, and port of the CQL shell session.

```
Connected to Test Cluster at 127.0.0.1:9042.
```

Show the request activity details for a specific session.

```
SHOW SESSION d0321c90-508e-11e3-8c7b-73ded3cb6170
```

Note: Use a session id from the query results or from the system_traces.sessions table.

Sample output of SHOW SESSION is:

```
Tracing session: d0321c90-508e-11e3-8c7b-73ded3cb6170
activity
| source      | source_elapsed | timestamp
```

```
-----
-----+-----+
-----+-----+
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.

Specify the path of the file relative to the current directory (that is the directory where cqlsh was started on the local host). Enclose the file name in single quotation marks. Use tilde (~) for the user's home directory.

The output of each statement displays, including error messages, in STDOUT. You can use `IF NOT EXISTS` to suppress errors for some statements, such as `CREATE KEYSPACE`. All statements in the file execute, even if a no-operation error occurs.

Tip: The `DESC` command outputs an executable CQL statement.

Synopsis

```
SOURCE 'file_name'
```

Table 25. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.

Table 25. Legend (continued)

Syntax conventions	Description
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key : value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2</i> >	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > ... </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@ <i>xml_entity</i> =' <i>xml_entity_type</i> '	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Examples

Execute CQL statements from a file:

```
SOURCE '~/cycling_setup/create_ks_and_tables.cql'
```

Note: To execute a CQL file without starting a shell session use `bin/cqlsh --file 'file_name'`.

TRACING

Enables and disables tracing for transactions on all nodes in the cluster. Use tracing to troubleshoot performance problems. Detailed transaction information related Cassandra

internal operations is captured in the `system_traces` keyspace. When a query runs a session id displays in the query results and an entry with the high-level details such as session id and client, and session length, is written to the `system_traces.session` table. More detailed data for each operation Cassandra performed is written to the `system_traces.events` table.

Note:

The session id is used by the `SHOW SESSION tracing_session_id` command to display detailed event information.

Tracing information is saved for 24 hours. To save tracing data longer than 24 hours, copy it to another location. For information about probabilistic tracing, see [Cassandra 3.0 documentation](#).

Tip: For more information on tracing data, see [this post](#) on the DataStax Support Blog, which explains in detail how to locate data on disk.

Synopsis

TRACING [ON | OFF]

Table 26. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.

Table 26. Legend (continued)

Syntax conventions	Description
{ key : value }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
<datatype1,datatype2>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
cql_statement;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' <schema> . . . </schema> '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@xml_entity='xml_entity_type'	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Examples

Tracing a write request

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

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

Turn on tracing:

```
TRACING ON
```

Insert a record into the cyclist_name table:

```
INSERT INTO cycling.cyclist_name (
    id,
    lastname,
```

```
firstname
)
VALUES (
    e7ae5cf3-d358-4d99-b900-85902fda9bb0 ,
    'FRAME' ,
    'Alex'
)i
```

The request and each step are captured and displayed.

Tracing session: 9b378c70-b114-11e6-89b5-b7fad52e1885

activity	timestamp	source	source_elapsed
client			
-----	-----	-----	-----
-----	-----	-----	-----
Execute CQL3 query	2016-11-22 16:34:34.300000	127.0.0.1	0 127.0.0.1
Parsing INSERT INTO cycling.cyclist_name (id, lastname, firstname)			
VALUES (e7ae5cf3-d358-4d99-b900-85902fda9bb0, 'FRAME', 'Alex');			
[Native-Transport-Requests-1]	2016-11-22 16:34:34.305000	127.0.0.1	5935 127.0.0.1
		Preparing statement	
[Native-Transport-Requests-1]	2016-11-22 16:34:34.308000	127.0.0.1	9199 127.0.0.1
		Determining replicas for mutation	
[Native-Transport-Requests-1]	2016-11-22 16:34:34.330000	127.0.0.1	30530 127.0.0.1
		Appending to commitlog	
[MutationStage-3]	2016-11-22 16:34:34.330000	127.0.0.1	30979 127.0.0.1
		Adding to cyclist_name memtable	
[MutationStage-3]	2016-11-22 16:34:34.330000	127.0.0.1	31510 127.0.0.1
		Request complete	
		2016-11-22 16:34:34.333633	127.0.0.1
		33633 127.0.0.1	

Tracing a sequential scan

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 cyclist_name table.

```
SELECT * FROM cycling.cyclist_name ;
```

The query results display first, followed by the session ID and session details.

id	firstname	lastname
e7ae5cf3-d358-4d99-b900-85902fda9bb0	Alex	FRAME
fb372533-eb95-4bb4-8685-6ef61e994caa	Michael	MATTHEWS
5b6962dd-3f90-4c93-8f61-eabfa4a803e2	Marianne	VOS
220844bf-4860-49d6-9a4b-6b5d3a79cbfb	Paolo	TIRALONGO
6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47	Steven	KRUIKSWIJK
e7cd5752-bc0d-4157-a80f-7523add8dbcd	Anna	VAN DER BREGGEN

(6 rows)

Tracing session: 117c1440-b116-11e6-89b5-b7fad52e1885

activity	timestamp		
source	source_elapsed	client	
Execute CQL3 query	2016-11-22		
16:45:02.212000 127.0.0.1 0 127.0.0.1			
			Parsing SELECT *
FROM cycling.cyclist_name ; [Native-Transport-Requests-1]			2016-11-22
16:45:02.212000 127.0.0.1 372 127.0.0.1			
Preparing statement [Native-Transport-Requests-1]			2016-11-22
16:45:02.212000 127.0.0.1 541 127.0.0.1			
Computing ranges to query [Native-Transport-Requests-1]			2016-11-22
16:45:02.213000 127.0.0.1 807 127.0.0.1			
Submitting range requests on 257 ranges with a concurrency of 257 (0.3			
rows per range expected) [Native-Transport-Requests-1]			2016-11-22
16:45:02.213000 127.0.0.1 1632 127.0.0.1			
			Submitted 1
concurrent range requests [Native-Transport-Requests-1]			2016-11-22
16:45:02.215000 127.0.0.1 3002 127.0.0.1			
Executing seq scan across 1 sstables for			
(min(-9223372036854775808), min(-9223372036854775808)] [ReadStage-2]			2016-11-22
2016-11-22 16:45:02.215000 127.0.0.1 3130 127.0.0.1			
Read 6 live and 0 tombstone cells [ReadStage-2]			2016-11-22
16:45:02.216000 127.0.0.1 3928 127.0.0.1			

16:45:02.216252 | 127.0.0.1 |

Request complete | 2016-11-22
4252 | 127.0.0.1

CQL commands

This section describes the commands that are specific to CQL.

ALTER KEYSPACE

Modifies the keyspace replication strategy, the number of copies of the data Cassandra creates in each data center, [REPLICATION](#), and/or disable the commit log for writes, [DURABLE_WRITES](#).

Restriction: Changing the keyspace name is not supported.

Synopsis

```
ALTER KEYSPACE keyspace_name
    WITH REPLICATION = {
        'class' : 'SimpleStrategy', 'replication_factor' : N
        | 'class' : 'NetworkTopologyStrategy', 'dc1_name' : N [, ...]
    }
    [AND DURABLE_WRITES = true|false] ;
```

Table 27. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.

Table 27. Legend (continued)

Syntax conventions	Description
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key : value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2</i> >	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > . . . </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@ <i>xml_entity='xml_entity_type'</i>	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

REPLICATION

Simple strategy

Assign the same replication factor to the entire cluster. Use for evaluation and single data center test and development environments only.

```
REPLICATION = {
    'class' : 'SimpleStrategy',
    'replication_factor' : N
}
```

Network topology strategy

Assign replication factors to each data center in a comma separated list. Use in production environments and multi-DC test and development

environments. Data center names must match the snitch DC name; refer to [Snitches](#) for more details.

```
REPLICATION = {
    'class' : 'NetworkTopologyStrategy',
    'datacenter_name' : N [, 'datacenter_name' : N]
}
```

Note: Datacenter names are case sensitive. Verify the case of the using utility, such as `nodetool status`.

DURABLE_WRITES

Optionally (not recommended), bypass the commit log when writing to the keyspace by disabling durable writes (`DURABLE_WRITES = false`). Default value is `true`.

CAUTION: Never disable durable writes when using SimpleStrategy replication.

Example

Change the `cycling` keyspace to `NetworkTopologyStrategy` in a single data center and turn off durable writes (not recommended). This example uses the default data center name in Cassandra with a replication factor of 3.

```
ALTER KEYSPACE cycling
WITH REPLICATION = {
    'class' : 'NetworkTopologyStrategy',
    'datacenter1' : 3 }
AND DURABLE_WRITES = false ;
```

ALTER MATERIALIZED VIEW

Changes materialized view table properties. The statement returns no results.

Restriction: Cassandra does not support changing the name or columns of the materialized view.

Synopsis

```
ALTER MATERIALIZED VIEW [keyspace_name.] view_name
[WITH table_options]
```

Table 28. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.

Table 28. Legend (continued)

Syntax conventions	Description
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key : value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > ... </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@ <i>xml_entity</i> =' <i>xml_entity_type</i> '	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

keyspace_name

To alter a materialized view in a keyspace other than the current keyspace, put the keyspace name in front of the name of the materialized view, followed by a period.

view_name

The name of the new materialized view.

table_options

Table options are defined when the materialized view is created. Modify the [table property](#) in the WITH clause using the following syntax:

- Single value:

```
option_name = 'string_value'
```

Enclose string values in single quotes.

- Specify options with multiple subproperties in simple JSON format:

```
option_name = {
    'subproperty_name' : 'value',
    'subproperty_name' : 'value',
    'subproperty_name' : 'value' [, ...] }
```

- Separate multiple options with AND:

```
ALTER MATERIALIZED VIEW keyspace_name.table_name
WITH option_name = 'string_value'
AND option_name = {
    'subproperty_name' : 'string_value',
    'subproperty_name' : numeric_value[, ...] };
```

Examples

Modifying table properties

For an overview of properties that apply to materialized views, see [table_options](#).

```
ALTER MATERIALIZED VIEW cycling.cyclist_by_age
WITH comment = 'A most excellent and useful view'
AND bloom_filter_fp_chance = 0.02;
```

Modifying compression and compaction

Use a property map to specify new properties for compression or compaction.

```
ALTER MATERIALIZED VIEW cycling.cyclist_by_age
WITH compression = {
    'sstable_compression' : 'DeflateCompressor',
    'chunk_length_kb' : 64 }
```

CQL reference

```
AND compaction = {  
    'class': 'SizeTieredCompactionStrategy',  
    'max_threshold': 64};
```

Changing caching

You can create and change caching properties using a property map.

This example changes the keys property to `NONE` (the default is `ALL`) and changes the `rows_per_partition` property to 15.

```
ALTER MATERIALIZED VIEW cycling.cyclist_by_age  
WITH caching = {  
    'keys' : 'NONE',  
    'rows_per_partition' : '15' };
```

Viewing current materialized view properties

Use `DESCRIBE MATERIALIZED VIEW` to see all current properties.

```
DESCRIBE MATERIALIZED VIEW cycling.cyclist_by_age
```

```
CREATE MATERIALIZED VIEW cycling.cyclist_by_age AS  
    SELECT age, cid, birthday, country, name  
    FROM cycling.cyclist_mv  
    WHERE age IS NOT NULL AND cid IS NOT NULL  
    PRIMARY KEY (age, cid)  
    WITH CLUSTERING ORDER BY (cid ASC)  
    AND bloom_filter_fp_chance = 0.02  
    AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}  
    AND comment = 'A most excellent and useful view'  
    AND compaction = {'class':  
        'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy',  
        'max_threshold': '32', 'min_threshold': '4'}  
        AND compression = {'chunk_length_in_kb': '64', 'class':  
            'org.apache.cassandra.io.compress.DeflateCompressor'}  
            AND crc_check_chance = 1.0  
            AND dclocal_read_repair_chance = 0.1  
            AND default_time_to_live = 0  
            AND gc_grace_seconds = 864000  
            AND max_index_interval = 2048  
            AND memtable_flush_period_in_ms = 0  
            AND min_index_interval = 128  
            AND read_repair_chance = 0.0  
            AND speculative_retry = '99PERCENTILE';
```

ALTER ROLE

Changes password, and set superuser or login options.

Synopsis

```
ALTER ROLE role_name
  [WITH [PASSWORD = 'password']
   [LOGIN = true | false]
   [SUPERUSER = true | false]
   [OPTIONS = map_literal]]
```

Table 29. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (. . .) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key</i> : <i>value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.

Table 29. Legend (continued)

Syntax conventions	Description
' <schema> . . . </schema> '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@xml_entity='xml_entity_type'	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

PASSWORD

Change the password of the logged in role. Superusers (and roles with ALTER PERMISSION to a role) can also change the password of other roles.

SUPERUSER

Enable or disable superuser status for another role, that is any role other than the one that is currently logged in. Setting superuser to false, revokes permission to create new roles; disabling does not automatically revoke the AUTHORIZE, ALTER, and DROP permissions that may already exist.

LOGIN

Enable or disable log in for roles other than currently logged in role.

OPTIONS

Reserved for external authenticator plug-ins.

Example

Change the password for coach:

```
ALTER ROLE coach WITH PASSWORD='bestTeam' ;
```

ALTER TABLE

Changes the datatype of a columns, add new columns, drop existing columns, renames columns, and change table properties. The command returns no results.

Restriction: Altering PRIMARY KEY columns is not supported. Altering columns in a table that has a materialized view is not supported.

Note: `ALTER COLUMNFAMILY` is deprecated.

Synopsis

```
ALTER TABLE [keyspace_name.] table_name
[ALTER column_name TYPE cql_type]
[ADD (column_definition_list)]
[DROP column_list | COMPACT STORAGE ]
```

```
[RENAME column_name TO column_name]
[WITH table_properties];
```

Table 30. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key</i> : <i>value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > ... </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.

Table 30. Legend (continued)

Syntax conventions	Description
<code>@xml_entity='xml_entity_type'</code>	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

ALTER *column_name* TYPE *cql_type*

column_name

Name of column to alter.

cql_type

Change column data type to a compatible type.

Note: Cassandra does not support changing the type of collections (lists, sets and maps) and counters.

Table 31. CQL Data Types

CQL Type	Constants supported	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)
date	strings	Value is a date with no corresponding time value; Cassandra encodes date as a 32-bit integer representing days since epoch (January 1, 1970). Dates can be represented in queries and inserts as a string, such as 2015-05-03 (yyyy-mm-dd)
decimal	integers, floats	Variable-precision decimal Java type

Table 31. CQL Data Types (continued)

CQL Type	Constants supported	Description
		Note: When dealing with currency, it is a best practice to have a currency class that serializes to and from an int or use the Decimal form.
double	integers, floats	64-bit IEEE-754 floating point Java type
float	integers, floats	32-bit IEEE-754 floating point Java type
frozen	user-defined types, collections, tuples	A frozen value serializes multiple components into a single value. Non-frozen types allow updates to individual fields. Cassandra treats the value of a frozen type as a blob. The entire value must be overwritten. Note: Cassandra no longer requires the use of frozen for tuples: <code>frozen <tuple <int, tuple<text, double>>></code>
inet	strings	IP address string in IPv4 or IPv6 format, used by the python-cql driver and CQL native protocols
int	integers	32-bit signed integer
list	n/a	A collection of one or more ordered elements: [literal, literal, literal]. CAUTION: Lists have limitations and specific performance considerations. Use a frozen list to decrease impact. In general, use a set instead of list.

Table 31. CQL Data Types (continued)

CQL Type	Constants supported	Description
map	n/a	A JSON-style array of literals: { literal : literal, literal : literal ... }
set	n/a	A collection of one or more elements: { literal, literal, literal }
smallint	integers	2 byte integer
text	strings	UTF-8 encoded string
time	strings	Value is encoded as a 64-bit signed integer representing the number of nanoseconds since midnight. Values can be represented as strings, such as 13:30:54.234.
timestamp	integers, strings	Date and time with millisecond precision, encoded as 8 bytes since epoch. Can be represented as a string, such as 2015-05-03 13:30:54.234.
timeuuid	uuids	Version 1 UUID only
tinyint	integers	1 byte integer
tuple	n/a	A group of 2-3 fields.
uuid	uuids	A UUID in standard UUID format
varchar	strings	UTF-8 encoded string
varint	integers	Arbitrary-precision integer Java type

ADD (*column_definition_list*)

Add one or more columns and set the data type, enter the name followed by the data types. The value is automatically set to null. To add multiple columns, use a comma separated list.

```
column_name cql_type [ , ]
```

```
[column_name cql_type [, ...]]
```

Restriction: Adding PRIMARY KEYs is not supported once a table has been created.

DROP (*column_list*)

Comma separated list of columns to drop. The values contained in the row are also dropped and not recoverable.

DROP COMPACT STORAGE

Use this option only to migrate tables to a DataStax Enterprise version that does not support COMPACT STORAGE. Removes Thrift compatibility mode from the table, which exposes the underlying structure of the Thrift table.

See .

Note: Removing Thrift compatibility from a table that also has a search index disables HTTP writes and deletes-by-ID on the search index.

RENAME *column_name* TO *column_name*

Changes the name of a column and preserves the existing values.

table_properties

After a table has been created, you can modify the properties. There are two types of properties, a single option that is set equal to a value:

```
option_name = value [AND ...]
```

For example, `speculative_retry = '10ms'`. Enclose the value for a string property in single quotation marks.

Some table properties are defined as a map in simple JSON format:

```
option_name = { subproperty_name : value [, ...] }
```

See [table_options](#) for more details.

Examples

Specifying the table and keyspace

You can qualify the table name by prepending the name of its keyspace. For example, to specify the `teams` table in the `cycling` keyspace:

```
ALTER TABLE cycling.teams ALTER ID TYPE uuid;
```

Changing the type of a column

Change the column data type to a [compatible](#) type.

Change the birthday timestamp column to type blob.

```
ALTER TABLE cycling.cyclist_alt_stats
ALTER birthday TYPE blob;
```

You can only change the datatype of a column when the column already exists. When a column's datatype changes, the bytes stored in values for that column remain unchanged. If existing data cannot be serialized to conform to the new datatype, the CQL driver or interface returns errors.

Warning:

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

Adding a column

To add a column (other than a column of a collection type) to a table, use the ADD instruction:

```
ALTER TABLE cycling.cyclist_races
ADD firstname text;
```

To add a column of a collection type:

```
ALTER TABLE cycling.upcoming_calendar
ADD events list<text>;
```

This operation does not validate the existing data.

You cannot use the ADD instruction to add:

- A column with the same name as an existing column
- A static column if the table has no clustering columns and uses COMPACT STORAGE.

Dropping a column

To remove a column from the table, use the DROP instruction:

```
ALTER TABLE cycling.basic_info
DROP birth_year;
```

DROP removes the column from the table definition and marks the column values with [tombstones](#). The column becomes unavailable for queries immediately after it is dropped. Cassandra drops the column data during the next compaction. To force the removal of dropped columns before compaction occurs, use ALTER TABLE to update the metadata, and then run [nodetool upgradesstables](#) to put the drop into effect.

Restriction:

- If you drop a column then re-add it, Cassandra does not restore the values written before the column was dropped.
- Do not re-add a dropped column that contained timestamps generated by a client; you can re-add columns with timestamps generated by Cassandra's [write time](#) facility.
- You cannot drop columns from tables defined with the [COMPACT STORAGE](#) option.

Renaming a column

The main purpose of RENAME is to change the names of CQL-generated primary key and column names that are missing from a [legacy table](#). The following restrictions apply to the RENAME operation:

- You can only rename clustering columns, which are part of the primary key.
- You cannot rename the partition key.
- You can index a renamed column.
- You cannot rename a column if an index has been created on it.
- You cannot rename a static column, since you cannot use a static column in the table's primary key.

Modifying table properties

To change the table storage properties set when the table was created, use one of the following formats:

- Use ALTER TABLE and a WITH instruction that introduces the property name and value.
- Use ALTER TABLE WITH and a property map, as shown in the [next section on compression and compaction](#).

This example uses the WITH instruction to modify the [read_repair_chance property](#), which configures [read repair](#) for tables that use for a non-quorum consistency and how to change multiple properties using AND:

```
ALTER TABLE cyclist_mv
    WITH comment = 'ID, name, birthdate and country'
        AND read_repair_chance = 0.2;
```

Enclose a text property value in single quotation marks. You cannot modify properties of a table that uses [COMPACT STORAGE](#).

Modifying compression and compaction

Use a property map to alter a table's compression or compaction setting:

```
ALTER TABLE cycling_comments
WITH compression = {
  'sstable_compression' : 'DeflateCompressor',
  'chunk_length_kb' : 64 };
```

Enclose the name of each key in single quotes. If the value is a string, enclose this in quotes as well.

CAUTION: If you change the compaction strategy of a table with existing data, Cassandra rewrites all existing SSTables using the new strategy. This can take hours, which can be a major problem for a production system. For strategies to minimize this disruption, see [How to change Cassandra compaction strategy on a production cluster](#) and [Impact of Changing Compaction Strategy](#).

Changing caching

Create and change the caching options using a property map to optimize queries that return 10.

```
ALTER TABLE cycling.events
WITH caching = {
  'keys': 'NONE',
  'rows_per_partition': 10 };
```

Reviewing the table definition

Use DESCRIBE to view the table definition.

```
cqlsh:cycling> desc table cycling.events ;
```

The details including the column names is returned.

```
CREATE TABLE cycling.events (
  month int,
  end timestamp,
  class text,
  title text,
  location text,
  start timestamp,
  type text,
  PRIMARY KEY (month, end, class, title)
) WITH CLUSTERING ORDER BY (end ASC, class ASC, title ASC)
AND bloom_filter_fp_chance = 0.01
AND caching = {'keys': 'NONE', 'rows_per_partition': '10'}
AND comment = ''
AND compaction = {'class':
'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy',
'max_threshold': '32', 'min_threshold': '4'}
```

```

AND compression = { 'chunk_length_in_kb': '64', 'class':
'org.apache.cassandra.io.compress.LZ4Compressor' }
AND crc_check_chance = 1.0
AND dclocal_read_repair_chance = 0.1
AND default_time_to_live = 0
AND gc_grace_seconds = 864000
AND max_index_interval = 2048
AND memtable_flush_period_in_ms = 0
AND min_index_interval = 128
AND read_repair_chance = 0.0
AND speculative_retry = '99PERCENTILE';

```

ALTER TYPE

Modify an existing user-defined type (UDT).

Restriction: Modifying UDTs used in primary keys or index columns is not supported.

Synopsis

```

ALTER TYPE field_name
[ALTER field_name TYPE new_cql_datatype
| ADD (field_name cql_datatype[,...])
| RENAME field_name TO new_field_name[AND ...]]

```

Table 32. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.

Table 32. Legend (continued)

Syntax conventions	Description
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key : value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > . . . </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@ <i>xml_entity</i> =' <i>xml_entity_type</i> '	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

To make multiple changes to the UDT, use AND between the clauses.

ALTER *field_name* TYPE *new_cql_datatype*

Change the data type of a field. Specify the field name and the new cql datatype.

ADD (*field_name* *cql_datatype*[,...])

Add fields by entering a field name followed by the data type in a comma separated list; the values for existing rows is set to null.

RENAME *field_name* TO *new_field_name*

Enter the old name and new name of the field.

Examples

Changing the data type

To change the type of a field, the field must already exist and be compatible with the new type [CQL type compatibility](#).

Tip: Carefully choose the data type for each column at the time of table creation.

Change the birthday timestamp to a blob.

```
ALTER TABLE cycling.cyclist_alt_stats
ALTER birthday TYPE blob;
```

Adding a field

To add a new field to a user defined type, use ALTER TYPE and the ADD keyword. For existing UDTs, the field value is null.

```
ALTER TYPE fullname ADD middlename text ;
```

Changing a field name

To change the name of a field in a user-defined type, use the RENAME old_name TO new_name syntax. Rename multiple fields by separating the directives with AND.

Remove name from all the field names the cycling.fullname UDT.

```
ALTER TYPE cycling.fullname
RENAME middlename TO middle
AND lastname to last
AND firstname to first;
```

Verify the changes using describe:

```
desc type cycling.fullname
```

The new field names appear in the description.

```
CREATE TYPE cycling.fullname (
    first text,
    last text,
    middle text
) ;
```

ALTER USER

Alter existing user options.

Note: `ALTER USER` is supported for backwards compatibility. Authentication and authorization are based on ROLES.

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. See [CREATE ROLE](#) for more information about SUPERUSER and NOSUPERUSER.

Synopsis

```
ALTER USER user_name
WITH PASSWORD 'password'
[SUPERUSER | NOSUPERUSER]
```

Table 33. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key</i> : <i>value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syn-

Table 33. Legend (continued)

Syntax conventions	Description
	tax is useful when arguments might be mistaken for command line options.
' <schema> . . . </schema> '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@xml_entity='xml_entity_type'	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Examples

Alter a user's password:

```
ALTER USER moss WITH PASSWORD 'bestReceiver';
```

Alter a user to make that a superuser:

```
ALTER USER moss SUPERUSER;
```

BATCH

Combines multiple data modification language (DML) statements (such as INSERT, UPDATE, and DELETE) to achieve atomicity and isolation when targeting a single partition, or only atomicity when targeting multiple partitions.

A batch applies all DML statements within a single partition before the data is available, ensuring atomicity and isolation. A well-constructed batch targeting a single partition can reduce client-server traffic and more efficiently update a table with a single row mutation.

Note: If there are two different tables in the same keyspace and the two tables have the same partition key, this scenario is considered a single partition batch. There will be a single mutation for each table. This happens because the two tables could have different columns, even though the keyspace and partition are the same. Batches allow a caller to bundle multiple operations into a single batch request. All the operations are performed by the same coordinator. The best use of a batch request is for a single partition in multiple tables in the same keyspace. Also, batches provide a guarantee that mutations will be applied in a particular order.

For multiple partition batches, [logging](#) ensures that all DML statements are applied. Either all or none of the batch operations will succeed, ensuring atomicity. Batch isolation occurs only if the batch operation is writing to a single partition.

Important: Only use a multiple partition batch when there is no other viable option, such as [asynchronous statements](#). Multiple partition batches may decrease throughput and increase latency.

Optionally, a batch can apply a client-supplied timestamp. Before implementing or executing a batch see [Batching inserts and updates](#).

Synopsis

```
BEGIN [UNLOGGED | LOGGED] BATCH
[USING TIMESTAMP [epoch_microseconds]]
    dml_statement [USING TIMESTAMP [epoch_microseconds]];
    [dml_statement; ...]
APPLY BATCH;
```

Table 34. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key</i> : <i>value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.

Table 34. Legend (continued)

Syntax conventions	Description
<code>cql_statement ;</code>	End CQL statement. A semicolon (<code>;</code>) terminates all CQL statements.
<code>[--]</code>	Separate the command line options from the command arguments with two hyphens (<code>--</code>). This syntax is useful when arguments might be mistaken for command line options.
<code>' <schema> . . . </schema> '</code>	Search CQL only: Single quotation marks (<code>'</code>) surround an entire XML schema declaration.
<code>@xml_entity='xml_entity_type'</code>	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

A batch can contain the following types of *dml_statements*:

- [INSERT](#)
- [UPDATE](#)
- [DELETE](#)

LOGGED | UNLOGGED

If multiple partitions are involved, batches are logged by default. Running a batch with logging enabled ensures that either all or none of the batch operations will succeed, ensuring atomicity. Cassandra first writes the serialized batch to the [batchlog system table](#) that consumes the serialized batch as blob data. After Cassandra has successfully written and persisted (or hinted) the rows in the batch, it removes the batchlog data. There is a performance penalty associated with the batchlog, as it is written to two other nodes. Thresholds for [warning about or failure due to batch size](#) can be set.

If you do not want to incur a penalty for logging, run the batch operation without using the batchlog table by using the `UNLOGGED` keyword. Unlogged batching will [issue a warning](#) if too many operations or too many partitions are involved. Single partition batch operations are unlogged by default, and are the only unlogged batch operations recommended.

Although a logged batch enforces atomicity (that is, it guarantees if all DML statements in the batch succeed or none do), Cassandra does no other transactional enforcement at the batch level. For example, there is no batch isolation unless the batch operation is writing to a single partition. In multiple partition batch operations, clients are able to read the first updated rows from the batch, while other rows are still being updated on the server. In single

partition batch operations, clients cannot read a partial update from any row until the batch is completed.

USING TIMESTAMPS

Sets the write time for transactions executed in a BATCH.

Restriction: `USING TIMESTAMP` does not support LWT (lightweight transactions), such as DML statements that have an `IF NOT EXISTS` clause.

By default, Cassandra applies the same timestamp to all data modified by the batch; therefore statement order does not matter within a batch, thus a batch statement is not very useful for writing data that must be timestamped in a particular order. Use client-supplied timestamps to achieve a particular order.

User-defined timestamp

Specify the epoch time in microseconds after `USING TIMESTAMP`:

```
USING TIMESTAMP [epoch_microseconds]
```

When the time is not specified, Cassandra inserts the current time.

Same timestamp for all DMLs

Insert on first line of batch.

```
BEGIN BATCH USING TIMESTAMP [epoch_microseconds]
  DML_statement1;
  DML_statement2;
  DML_statement3;
APPLY BATCH;
```

Individual transactions

Insert at the end of a DML:

```
BEGIN BATCH
  DML_statement1;
  DML_statement2 USING TIMESTAMP [epoch_microseconds];
  DML_statement3;
APPLY BATCH;
```

Examples

Applying a client supplied timestamp to all DMLs

Insert meals paid for Vera Adrian using the user-defined date when inserting the records:

```
BEGIN BATCH USING TIMESTAMP 1481124356754405
INSERT INTO cycling.cyclist_expenses
(cyclist_name, expense_id, amount, description, paid)
VALUES ('Vera ADRIAN', 2, 13.44, 'Lunch', true);
INSERT INTO cycling.cyclist_expenses
(cyclist_name, expense_id, amount, description, paid)
VALUES ('Vera ADRIAN', 3, 25.00, 'Dinner', true);
APPLY BATCH;
```

Note: Combining two statements for the same partition results in a single table mutation.

View the records vertically:

```
expand ON
```

Verify that the timestamps are all the same:

```
SELECT cyclist_name, expense_id,
       amount, WRITETIME(amount),
       description, WRITETIME(description),
       paid,WRITETIME(paid)
  FROM cycling.cyclist_expenses
 WHERE cyclist_name = 'Vera ADRIAN';
```

Both records were entered with the same timestamp.

@ Row 1	
cyclist_name	Vera ADRIAN
expense_id	2
amount	13.44
writetime(amount)	1481124356754405
description	Lunch
writetime(description)	1481124356754405
paid	True
writetime(paid)	1481124356754405

@ Row 2	
cyclist_name	Vera ADRIAN
expense_id	3
amount	25
writetime(amount)	1481124356754405
description	Dinner
writetime(description)	1481124356754405
paid	False
writetime(paid)	1481124356754405

(2 rows)

If any DML statement in the batch uses compare-and-set (CAS) logic, for example the following batch with `IF NOT EXISTS`, an error is returned:

```
BEGIN BATCH USING TIMESTAMP 1481124356754405
    INSERT INTO cycling.cyclist_expenses
        (cyclist_name, expense_id, amount, description, paid)
        VALUES ('Vera ADRIAN', 2, 13.44, 'Lunch', true);
    INSERT INTO cycling.cyclist_expenses
        (cyclist_name, expense_id, amount, description, paid)
        VALUES ('Vera ADRIAN', 3, 25.00, 'Dinner', false) IF NOT EXISTS;
APPLY BATCH;
```

```
InvalidRequest: Error from server: code=2200 [Invalid query]
message="Cannot provide custom timestamp for conditional BATCH"
```

Batching conditional updates

Batch conditional updates introduced as lightweight transactions. However, a batch containing conditional updates can only operate within a single partition, because the underlying Paxos implementation only works at partition-level granularity. If one statement in a batch is a conditional update, the conditional logic must return true, or the entire batch fails. If the batch contains two or more conditional updates, all the conditions must return true, or the entire batch fails. 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;
```

Conditional batches cannot provide custom timestamps. `UPDATE` and `DELETE` statements within a conditional batch cannot use `IN` conditions to filter rows.

A [continuation of this example](#) shows how to use a static column with conditional updates in batch.

Batching counter updates

A batch of counters should use the `COUNTER` option because, unlike other writes in Cassandra, a counter update is not an `idempotent` operation.

```
BEGIN COUNTER BATCH
    UPDATE UserActionCounts SET total = total + 2 WHERE keyalias = 523;
    UPDATE AdminActionCounts SET total = total + 2 WHERE keyalias = 701;
APPLY BATCH;
```

Counter batches cannot include non-counter columns in the DML statements, just as a non-counter batch cannot include counter columns. Counter batch statements cannot provide custom timestamps.

CREATE AGGREGATE

Executes a user-defined function (UDF) on each row in a selected data set, optionally runs a final UDF on the result set and returns a value, for example average or standard deviation.

Synopsis

```
CREATE [OR REPLACE] AGGREGATE [IF NOT EXISTS]
keyspace_name.aggregate_name ( cql_type )
SFUNC udf_name
STYPE cql_type
FINALFUNC udf_name
INITCOND [value]
```

Table 35. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.

Table 35. Legend (continued)

Syntax conventions	Description
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key : value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2</i> >	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > . . . </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@ <i>xml_entity</i> =' <i>xml_entity_type</i> '	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

CREATE AGGREGATE [OR REPLACE] *aggregate_name(cql_type)* [IF EXISTS]

Creates an aggregate with specified name or replaces an existing one.

- CREATE AGGREGATE without *OR REPLACE* fails if an aggregate with the same signature already exists.
- CREATE AGGREGATE with the optional IF NOT EXISTS keywords creates an aggregate if it does not already exist and displays no error if it does.
- Only use *OR REPLACE* or *IF NOT EXISTS*.

Specify the CQL type returned by the aggregate function.

Restriction: Frozen collections are not supported as cql types for aggregates.

SFUNC *udf_name*

Specify a user-defined function. Calls the state function (SFUNC) for each row. The first parameter declared in the user-defined function is the *state* parameter; the function's return value is assigned to the state parameter, which is passed to the next call. Pass multiple values using collection types, such as tuples.

STYPE *cql_type*

CQL type of the parameter returned by the state function.

FINALFUNC *udf_name*

User-defined function executed on the final values in the state parameter.

INITCOND [*value*]

Define the initial condition, values, of the first parameter in the SFUNC. Set to null when no value defined.

Examples

Create an aggregate that calculates average in the cycling keyspace.

1. Set up a test table with data:

```
CREATE TABLE cycling.team_average (
    team_name text,
    cyclist_name text,
    cyclist_time_sec int,
    race_title text,
    PRIMARY KEY (team_name, race_title,cyclist_name));
INSERT INTO cycling.team_average (team_name, cyclist_name, cyclist_time_sec, race_t
```

2. Create a function with a state parameter as a tuple that counts the rows (by incrementing 1 for each record) in the first position and finds the total by adding the current row value to the existing subtotal the second position, and returns the updated state.

```
CREATE OR REPLACE FUNCTION cycling.avgState ( state tuple<int,bigint>, val int )
CALLED ON NULL INPUT
RETURNS tuple<int,bigint>
LANGUAGE java AS
$$ if (val !=null) {
    state.setInt(0, state.getInt(0)+1);
    state.setLong(1, state.getLong(1)+val.intValue());
}
```

```
    return state; $$  
;
```

Note: Use a simple test to verify that your function works properly.

```
CREATE TABLE cycling.test_avg (  
    id int PRIMARY KEY,  
    state frozen<tuple<int, bigint>>,  
    val int PRIMARY KEY);  
INSERT INTO test_avg (id,state,val) values (1,(6,9949),51);  
INSERT INTO test_avg (id,state,val) values (2,(79,10000),9999);
```

```
select state, avgstate(state,val) , val from test_avg;
```

The first value was incremented by one and the second value is the results of the initial state value and val.

state	cycling.avgstate(state, val)	val
(0, 9949)	(1, 10000)	51
(1, 10000)	(2, 19999)	9999

3. Create a function that divides the total value for the selected column by the number of records.

```
CREATE OR REPLACE FUNCTION cycling.avgFinal ( state tuple<int,bigint> )  
CALLED ON NULL INPUT  
RETURNS double  
LANGUAGE java AS  
$$ double r = 0;  
    if (state.getInt(0) == 0) return null;  
    r = state.getLong(1);  
    r/= state.getInt(0);  
    return Double.valueOf(r); $$  
;
```

4. Create the user-defined aggregate to calculate the average value in the column:

```
CREATE AGGREGATE cycling.average(int)  
SFUNC avgState  
STYPE tuple<int,bigint>  
FINALFUNC avgFinal
```

```
INITCOND (0,0);
```

5. Test the function using a select statement.

```
SELECT cycling.average(cyclist_time_sec) FROM cycling.team_average
WHERE team_name='UnitedHealthCare Pro Cycling Womens Team'
AND race_title='Amgen Tour of California Women's Race presented by SRAM - Stage 1'
```

CREATE INDEX

Define a new index on a single column of a 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.

Cassandra supports creating an index on most columns, excluding counter columns but including a clustering column of a [compound primary key](#) or on the partition (primary) key itself. Cassandra supports creating an index on a collection or the key of a collection map. Cassandra rejects an attempt to create an index on the collection key and value, as well as static columns.

Indexing can impact performance greatly. Before creating an index, be aware of when and [when not to create an index](#).

Synopsis

```
CREATE INDEX IF NOT EXISTS index_name
ON keyspace_name.table_name ( KEYS ( column_name ) )
```

Table 36. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.

Table 36. Legend (continued)

Syntax conventions	Description
...	Repeatable. An ellipsis (. . .) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key</i> : <i>value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > . . . </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@ <i>xml_entity</i> =' <i>xml_entity_type</i> '	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

index_name is an identifier, enclosed or not enclosed in double quotation marks, excluding reserved words.

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);
```

index_name

Optional identifier for index. If no name is specified, Cassandra names the index: *table_name_column_name_idx*. Enclose in quotes to use special characters or preserve capitalization.

Examples

Creating an index on a clustering column

Define a table having a [composite partition key](#), and then create an index on a clustering column.

```
CREATE TABLE mykeyspace.users (
    userID uuid,
    fname text,
    lname text,
    email text,
    address text,
    zip int,
    state text,
    PRIMARY KEY ((userID, fname), state)
) ;

CREATE INDEX ON mykeyspace.users (state);
```

Creating an index on a set or list collection

Create an index on a set or list collection column as you would any other column. Enclose the name of the collection column in parentheses at the end of the CREATE INDEX statement. For example, add a collection of phone numbers to the users table to index the data in the phones set.

```
ALTER TABLE users ADD phones set<text>;
```

```
CREATE INDEX ON users (phones);
```

If the collection is a map, Cassandra can create an [index on map values](#). Assume the users table contains this map data from the [example of a todo map](#):

```
{'2014-10-2 12:10' : 'die' }
```

The map key, the timestamp, is located to the left of the colon, and the map value is located to the right of the colon, 'die'. Indexes can be created on both map keys and map entries.

Creating an index on map keys

Create an index on [map collection keys](#). If an index of the map values of the collection exists, drop that index before creating an index on the map collection keys.

To index map keys, you use the `KEYS` keyword and map name in nested parentheses. For example, index the collection keys, the timestamps, in the todo map in the users table:

```
CREATE INDEX todo_dates ON users (KEYS(todo));
```

To query the table, you can use [CONTAINS KEY](#) in [WHERE](#) clauses.

Creating an index on the map entries

Create an index on map entries. An `ENTRIES` index can be created only on a map column of a table that doesn't have an existing index.

To index collection entries, you use the `ENTRIES` keyword and map name in nested parentheses. For example, index the collection entries in a list in a race table:

```
CREATE INDEX entries_idx ON race (ENTRIES(race_wins));
```

To query the table, you can use a [WHERE](#) clause.

Creating an index on a full collection

Create an index on a full `FROZEN` collection. An `FULL` index can be created on a set, list, or map column of a table that doesn't have an existing index.

To index collection entries, you use the `FULL` keyword and collection name in nested parentheses. For example, index the list **rnumbers**.

```
CREATE INDEX rnumbers_idx
ON cycling.race_starts (FULL(rnumbers));
```

To query the table, you can use a [WHERE](#) clause.

CREATE FUNCTION

Executes user-provided code in Cassandra in SELECT, INSERT and UPDATE statements. The UDF scope is keyspace-wide. By default, UDF includes support for Java generic methods and Javascript.

See [User Defined Functions](#) to add support for additional JSR-223 compliant scripting languages, such as Python, Ruby, and Scala.

Note: Before creating user-defined functions, set `enable_user_defined_functions=true` and if implementing Javascript also set `enable_scripted_user_defined_functions=true` in the [cassandra.yaml](#).

Synopsis

```
CREATE [OR REPLACE] FUNCTION [IF NOT EXISTS]
[keyspace_name.]function_name (
    var_name var_type [, ...] )
[CALLED | RETURNS NULL] ON NULL INPUT
RETURNS cql_data_type
LANGUAGE language_name AS
'code_block';
```

Table 37. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.

Table 37. Legend (continued)

Syntax conventions	Description
{ key : value }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
<datatype1,datatype2>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
cql_statement;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' <schema> . . . </schema> '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@xml_entity='xml_entity_type'	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

CREATE [OR REPLACE] FUNCTION *function_name* [IF NOT EXISTS]

Use the following options:

- CREATE *function_name*: Creates a new function and errors if it already exists, use with IF NOT EXISTS to suppress error.
- CREATE OR REPLACE: Creates a new function and overwrites it if it already exists.
- Only use either OR REPLACE or IF NOT EXISTS.

var_name var_type

The variable name and data type passed from request to the code block for execution. Use a comma separated list to declare multiple variables.

You can also use complex types, such as collections, tuple and user-defined types as argument. Tuple types and user-defined types are handled by the DataStax Java Driver.

Arguments for functions can be literals or terms. Prepared statement placeholders can be used, too.

For example:

```
column text, num int
```

CALLED ON NULL INPUT

Executes the user-provided code block even if the input value is null or missing.

RETURNS NULL ON NULL INPUT

Does not execute the user-provided code block on null values; returns null.

RETURNS *cql_data_type*

Map the expected output from the code block to a compatible CQL data type.

LANGUAGE *language_name*

Supported types are Java and Javascript. See [User Defined Functions](#) to add support for additional JSR-223 compliant scripting languages, such as Python, Ruby, and Scala.

'*code_block*' | \$\$ *code_block* \$\$

Enclose the code block in single quotes or if the code block contains any special characters enclose it in double dollar signs (\$\$). The code is wrapped as a function and applied to the target variables.

UDFs are susceptible to all of the normal issues that may occur with the chosen programming language. Safe guard against exceptions, such as null pointer exceptions, illegal arguments, or any other potential sources. An exception during function execution results in the entire statement failing.

Examples

Overwrite or create the fLog function that computes the logarithm of an input value. CALLED ON NULL INPUT ensures that the function will always be executed.

```
CREATE OR REPLACE FUNCTION cycling.fLog (input double)
CALLED ON NULL INPUT
RETURNS double LANGUAGE java AS
'return Double.valueOf(Math.log(input.doubleValue()));';
```

Create a function that returns the first *N* characters from a text field in Javascript. RETURNS NULL ON NULL INPUT ensures that if the input value is null then the function is not executed.

```
CREATE FUNCTION IF NOT EXISTS cycling.left (column TEXT,num int)
RETURNS NULL ON NULL INPUT
RETURNS text
LANGUAGE javascript AS
```

```
$$ column.substring(0,num) $$;
```

Use the function in requests:

```
SELECT left(firstname,1), lastname from cycling.cyclist_name;
```

```
cycling.left(firstname, 1) | lastname
-----
A | FRAME
A | PIETERS
M | MATTHEWS
M | VOS
P | TIRALONGO
S | KRUIKSWIJK
A | VAN DER BREGGEN
```

CREATE KEYSPACE

Creates a top-level namespace. Configure the replica placement strategy, replication factor, and durable writes setting.

Synopsis

```
CREATE KEYSPACE [IF NOT EXISTS] keyspace_name
    WITH REPLICATION = {
        'class' : 'SimpleStrategy', 'replication_factor' : N }
        | 'class' : 'NetworkTopologyStrategy',
        'dc1_name' : N [, ...]
    }
    [AND DURABLE_WRITES = true|false] ;
```

Table 38. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.

Table 38. Legend (continued)

Syntax conventions	Description
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key : value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > ... </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@ <i>xml_entity</i> =' <i>xml_entity_type</i> '	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

CREATE KEYSPACE [IF NOT EXISTS] *keyspace_name*

Keyspace names can have up to 48 alpha-numeric characters and contain underscores; only letters and numbers are supported as the first character. Cassandra forces keyspace names to lowercase when entered without quotes.

If a keyspace with the same name already exists, an error occurs and the operation fails; use IF NOT EXISTS to suppress the error message.

REPLICATION = { *replication_map* }

The replication map determines how many copies of the data are kept in a given data center. This setting impacts consistency, availability and request speed, for more details see [replica placement strategy](#).

Table 39. Replication strategy class and factor settings

Class	Replica-factor	Value Description
'SimpleStrategy'	'replication_factor' : N	Assign the same replication factor to the entire cluster. Use for evaluation and single data center test and development environments only.
'NetworkTopologyStrategy'	'datacenter-name' : N	Assign replication factors to each data center in a comma separated list. Use in production environments and multi-DC test and development environments. Data center names must match the snitch DC name; refer to Snitches for more details.

Simple Topolgy syntax:

```
'class' : 'SimpleStrategy', 'replication_factor' : N
```

Network Topology syntax:

```
'class' : 'NetworkTopologyStrategy',
'dc1_name' : N [, ...]
```

DURABLE_WRITES = true|false

Optionally (not recommended), bypass the commit log when writing to the keyspace by disabling durable writes (DURABLE_WRITES = false). Default value is `true`.

CAUTION: Never disable durable writes when using SimpleStrategy replication.

Examples

Create a keyspace for a single node evaluation cluster

Create cycling keyspace on a single node evaluation cluster:

```
CREATE KEYSPACE cycling
```

```
WITH REPLICATION = {
    'class' : 'SimpleStrategy',
    'replication_factor' : 1
};
```

Create a keyspace NetworkTopologyStrategy on an evaluation cluster

This example shows how to create a keyspace with network topology in a single node evaluation cluster.

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

Note: `datacenter1` is the default data center name. To display the data center name, use `nodetool status`.

```
nodetool status
```

The node tool returns the data center name, rack name, host name and IP address.

```
Datacenter: datacenter1
=====
Status=Up/Down
| / State=Normal/Leaving/Joining/Moving
-- Address      Load      Tokens   Owns      Host ID
   Rack
UN  127.0.0.1  46.59 KB  256      100.0%
  dd867d15-6536-4922-b574-e22e75e46432  rack1
```

Create the cycling keyspace in an environment with multiple data centers

Set the replication factor for the Boston, Seattle, and Tokyo data centers. The data center name must match the name configured in the snitch.

```
CREATE KEYSPACE "Cycling"
  WITH REPLICATION = {
    'class' : 'NetworkTopologyStrategy',
    'boston' : 3 , // Datacenter 1
    'seattle' : 2 , // Datacenter 2
    'tokyo' : 2 // Datacenter 3
};
```

Note: For more about replication strategy options, see [Changing keyspace replication strategy](#)

Disabling durable writes

Disable write commit log for the cycling keyspace. Disabling the commit log increases the risk of data loss. Do not disable in SimpleStrategy environments.

```
CREATE KEYSPACE cycling
  WITH REPLICATION = {
    'class' : 'NetworkTopologyStrategy',
    'datacenter1' : 3
  }
  AND DURABLE_WRITES = false ;
```

CREATE MATERIALIZED VIEW

Create a materialized view creates a query only table from a base table; when changes are made to the base table the materialized view is automatically updated. Use materialized views to more efficiently query the same data in different ways, see [Creating a materialized view](#).

Restriction:

- Use all base table primary keys in the materialized view as primary keys.
- Optionally, add one non-PRIMARY KEY column from the base table to the materialized view's PRIMARY KEY.
- [Static columns](#) are not supported as a PRIMARY KEY.

Synopsis

```
CREATE MATERIALIZED VIEW [IF NOT EXISTS] [keyspace_name.] view_name
AS SELECT column_list
  FROM [keyspace_name.] base_table_name
  WHERE column_name IS NOT NULL [AND column_name IS NOT NULL ...]
        [AND relation...]
  PRIMARY KEY ( column_list )
  [WITH [table_properties]
    [AND CLUSTERING ORDER BY (cluster_column_name order_option )]]
```

Table 40. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.

Table 40. Legend (continued)

Syntax conventions	Description
()	Group. Parentheses () identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key</i> : <i>value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > ... </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@ <i>xml_entity</i> =' <i>xml_entity_type</i> '	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Create MATERIALIZED VIEW clause

IF NOT EXISTS

Optional. Suppresses the error message when attempting to create a materialized view that already exists. Use to continuing executing commands, such as in a `SOURCE` command. The option only validates that a materialized view with the same name exists; columns, primary keys, properties and other settings can differ.

***keyspace_name*.**

Optional. When no keyspace is selected or to create the view in another keyspace, enter keyspace name before the materialized view name.

Note: Base tables and materialized views are always in the same keyspace.

view_name

Materialized view names can only contain alpha-numeric characters and underscores. The view name must begin with a number or letter and can be up to 49 characters long.

AS SELECT *column_list*

column_list

Comma separated list of non-PRIMARY KEY columns from the base table to include in the materialized view. All primary key columns are automatically included.

Static columns, even when specified, are not included in the materialized view.

FROM [*keyspace_name*] . *base_table_name*

***keyspace_name*.**

Keyspace where the base table is located. Only required when creating a materialized view in a different keyspace than the current keyspace.

base_table_name

Name of the table that the materialized view is based on.

WHERE *PRIMARY_KEY_column_name* IS NOT NULL [AND *PK_column_name* IS NOT NULL ...]

***PK_column_name* IS NOT NULL**

Test all primary key columns for null values in the where clause. Separate each condition with AND. Rows with null values in the primary key are not inserted into the materialized view table.

AND relation

Other relations that target the specific data needed. See the [relation](#) section of the CQL SELECT documentation.

PRIMARY KEY (*column_list*)

column_list

Comma separated list of columns used to partition and cluster the data.

You can add a single non-primary key column from the base table. Reorder the primary keys as needed to query the table more efficiently, including changing the partitioning and clustering keys.

List the partition key first, followed by the clustering keys. Create a compound partition key by enclosing column names in parenthesis, for example:

```
PRIMARY KEY (
    (PK_column1[, PK_column2...]),
    clustering_column1[, clustering_column2...])
```

Note: [Static columns](#) are not supported in materialized views.

WITH *table_properties*

table_properties

Optional. Specify table properties if different than default. Separate table property definitions with an AND.

Note: The base table properties are not copied.

Example

Creates the materialized view `cyclist_by_age` based on the source table `cyclist_mv`. The WHERE clause ensures that only rows whose `age` and `cid` columns are non-NULL are added to the materialized view.

```
CREATE MATERIALIZED VIEW cycling.cyclist_by_age
AS SELECT age, name, country
FROM cycling.cyclist_mv
WHERE age IS NOT NULL AND cid IS NOT NULL
PRIMARY KEY (age, cid)
WITH caching = { 'keys' : 'ALL', 'rows_per_partition' : '100' }
AND comment = 'Based on table cyclist' ;
```

CREATE TABLE

Define a new table.

Synopsis

```
CREATE TABLE [IF NOT EXISTS] [keyspace_name.]table_name (
    column_definition [, ...]
    PRIMARY KEY (column_name [, column_name ...])
    [WITH table_options
        | CLUSTERING ORDER BY (clustering_column_name order)]
        | ID = 'table_hash_tag'
        | COMPACT STORAGE]
```

Table 41. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.

Table 41. Legend (continued)

Syntax conventions	Description
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key : value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > ... </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@ <i>xml_entity</i> =' <i>xml_entity_type</i> '	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Description

The `CREATE TABLE` command creates a new [table](#) under the current keyspace.

The `IF NOT EXISTS` keywords may be used in creating a table. Attempting to create an existing table returns an error unless the `IF NOT EXISTS` option is used. If the option is used, the statement is a no-op if the table already exists.

A [static column](#) can store the same data in multiple clustered rows of a partition, and then retrieve that data with a single `SELECT` statement.

You can add a [counter column](#) to a counter table.

column_definition

Columns are defined enclosed in parenthesis after the table name, use a comma-separated list to define multiple columns. All tables must have at least one primary key column. Each column is defined using the following syntax:

```
column_name cql_type_definition [STATIC | PRIMARY KEY] [, ...]
```

	Description
<code>column_name</code>	Use a unique name for each column in table. To preserve case or use special characters, enclose the name in double-quotes.
<code>cql_type_definition</code>	Defines the type of data allowed in the column, see CQL data type or a user-defined type .
<code>STATIC</code>	Optional, the column has a single value.
<code>PRIMARY KEY</code>	Optional, use to indicate that the column is the only primary key for the table; to define a key that uses multiple columns see PRIMARY KEY .

Restriction:

- When primary key is at the end of a column definition that column is the only primary key for the table.
- A table must have at least one `PRIMARY KEY`.
- A static column cannot be a primary key.
- Primary keys can include frozen collections.

Create a table that has a frozen user defined type.

```
CREATE TABLE cycling.race_winners (
    race_name text,
    race_position int,
    cyclist_name FROZEN<fullname>,
```

```
PRIMARY KEY (race_name, race_position));
```

See "[Creating a user-defined type](#)" for information on creating UDTs. .

PRIMARY KEY

Single key

When the PRIMARY KEY is one column, append PRIMARY KEY to the end of the column definition. This is only schema information required to create a table. When there is one primary key, it is the partition key; the data is divided and stored by the unique values in this column.

```
column_name cql_type_definition PRIMARY KEY
```

Alternatively, you can declare the primary key consisting of only one column in the same way as you declare a compound primary key.

Create the cyclist_name table with UUID as the primary key:

```
CREATE TABLE cycling.cyclist_name (
    id UUID PRIMARY KEY,
    lastname text,
    firstname text );
```

Restriction:

Primary keys cannot have the data type: counter, non-frozen collection, or static.

Compound key

A compound primary key consists of more than one column; the first column is the partition key, and the additional columns are clustering keys. To define compound primary key as follows:

```
PRIMARY KEY (partition_column_name, clustering_column_name [, ...])
```

Create the cyclist category table and store the data in reverse order:

```
CREATE TABLE cycling.cyclist_category (
    category text,
    points int,
    id UUID,
    lastname text,
```

```
PRIMARY KEY (category, points))
WITH CLUSTERING ORDER BY (points DESC);
```

Composite partition key

A composite partition key is a partition key consisting of multiple columns. Enclose the partition key columns in parenthesis.

```
PRIMARY KEY (
  (partition_column_name[ , ...]),
  clustering_column_name [ , ...])
```

Create a table that is optimized for query by cyclist rank by year:

```
CREATE TABLE cycling.rank_by_year_and_name (
  race_year int,
  race_name text,
  cyclist_name text,
  rank int,
  PRIMARY KEY ((race_year, race_name), rank) );
```

table_options

Table properties tune data handling, including I/O operations, compression, and compaction. Set table properties in the following CQL requests:

- [CREATE TABLE](#)
- [ALTER TABLE](#)
- [CREATE MATERIALIZED VIEW](#)
- [ALTER MATERIALIZED VIEW](#)

Note: Cassandra applies the default value if an option is not defined.

Table property options that have one value use the following syntax:

```
option_name = 'value' [AND ...]
```

Table property that have multiple subproperties are specified in a map. Maps are in simple JSON format, key value pairs in a comma separated list enclosed by braces:

```
option_name = { 'subproperty' : 'value' [ , ... ] } [AND ...]
```

Note: Enclose strings in quotes; not required for numeric values.

In a CQL statement use a `WITH` clause to define table property options, separate multiple values with `AND`, for example:

```
ALTER TABLE [keyspace_name.]table_name
  WITH option_name = 'value'
    AND option_name = option_map;
```

bloom_filter_fp_chance

False-positive probability for SSTable [bloom filter](#). When a client requests data, the bloom filter checks if the row exists before executing disk I/O.

```
bloom_filter_fp_chance = N
```

Value ranges from 0 to 1.0, where:

- 0: (Min value) enables the largest possible bloom filter and uses the most memory.
- 1.0: (Max value) disables the bloom filter.

Tip: Recommended setting: 0.1. A higher value yields diminishing returns.

Default: `bloom_filter_fp_chance = '0.01'`

caching

Caching optimizes the use of cache memory of a table without manual tuning. Cassandra weighs the cached data by size and access frequency.

Coordinate this setting with the global caching properties in the `cassandra.yaml` file. See [Cassandra 3.0 documentation](#).

```
caching = {
  'keys' = 'ALL | NONE',
  'rows_per_partition' = 'ALL' | 'NONE' | N}
```

Valid values:

- ALL— all primary keys or rows
- NONE— no primary keys or rows
- N : (rows per partition only) Number of rows; specify a whole number

Cassandra caches only the first N rows in a partition, as determined by the clustering order.

For example, to cache all riders in each age partition:

```
ALTER MATERIALIZED VIEW cycling.cyclist_by_age
WITH caching = {
  'keys' : 'ALL',
  'rows_per_partition' : 'ALL' } ;
```

Default: { 'keys' : 'ALL', 'rows_per_partition' : 'NONE' }

cdc

Create a Change Data Capture (CDC) log on the table.

```
cdc = TRUE | FALSE
```

Valid values:

- TRUE- create CDC log
- FALSE- do not create CDC log

`CREATE TABLE` and `ALTER TABLE` can be modified with this table option, but not `CREATE MATERIALIZED VIEW` or `ALTER MATERIALIZED VIEW`.

For example, create a table with a CDC log:

```
CREATE TABLE cycling.cyclist_name
WITH cdc = TRUE;
```

comments

Provide documentation on the table.

```
comments = 'text'
```

Tip: Enter a description of the types of queries the table was designed to satisfy.

For example, note the base table for the materialized view:

```
ALTER MATERIALIZED VIEW cycling.cyclist_by_age
WITH comment = "Basetable: cyclist_mv";
```

dclocal_read_repair_chance

Probability that a successful read operation triggers a read repair, between 0 and 1; default value: 0.01. Unlike the repair controlled by [read_repair_chance](#), this repair is limited to replicas in the same DC as the coordinator.

default_time_to_live

TTL (Time To Live) in seconds, where zero is disabled. The maximum configurable value is 630720000 (20 years). If the value is greater than zero, TTL is enabled for the entire table and an expiration timestamp is added to each column. A new TTL timestamp is calculated each time the data is updated and the row is removed after all the data expires.

Default value: 0 (disabled).

gc_grace_seconds

Seconds after data is marked with a tombstone (deletion marker) before it is eligible for garbage-collection. Default value: 864000 (10 days). The default value allows time for Cassandra to maximize consistency prior to deletion.

Note: Tombstoned records within the grace period are excluded from [hints](#) or [batched mutations](#).

In a single-node cluster, this property can safely be set to zero. You can also reduce this value for tables whose data is not explicitly deleted — for example, tables containing only data with [TTL](#) set, or tables with [default_time_to_live](#) set. However, if you lower the **gc_grace_seconds** value, consider its interaction with these operations:

- **hint replays** — When a node goes down and then comes back up, other nodes replay the write operations (called [hints](#)) that are queued for that node while it was unresponsive. Cassandra does not replay hints older than **gc_grace_seconds** after creation. The [max_hint_window_in_ms](#) setting in the [cassandra.yaml](#) file sets the time limit (3 hours by default) for collecting hints for the unresponsive node.
- **batch replays** — Like hint queues, [batch operations](#) store database mutations that are replayed in sequence. As with hints, Cassandra does not replay a batched mutation older than **gc_grace_seconds** after creation. If your application uses batch operations, consider the possibility that decreasing **gc_grace_seconds** increases the chance that a batched write operation may restore deleted data. The [batchlog_replay_throttle_in_kb](#) and [concurrent_batchlog_writes](#) properties in the [cassandra.yaml](#) file give some control of the batch replay process. The most important factors, however, are the size and scope of the batches you use.

memtable_flush_period_in_ms

Milliseconds before memtables associated with the table are flushed.

Default: 0

min_index_interval

Minimum gap between index entries in the index summary. A lower `min_index_interval` means the index summary contains more entries from the index, which allows Cassandra to search fewer index entries to execute a read. A larger index summary may also use more memory. The value for `min_index_interval` is the densest possible sampling of the index.

max_index_interval

If the total memory usage of all index summaries reaches this value, Cassandra decreases the index summaries for the coldest SSTables to the maximum set by `max_index_interval`. The `max_index_interval` is the sparsest possible sampling in relation to memory pressure.

read_repair_chance

The probability that a successful read operation triggers a read repair. Unlike the repair controlled by `dclocal_read_repair_chance`, this repair is not limited to replicas in the same DC as the coordinator. The value must be between `0` and `1`; default value: `0 . 0`.

speculative_retry

Overrides normal read timeout when `read_repair_chance` is not 1.0, sending another request to read. Specify the value as a number followed by a type, `ms` (milliseconds) or `percentile`. For example, `speculative_retry = '3ms'`.

Use the speculative retry property to configure [rapid read protection](#). In a normal read, Cassandra sends data requests to just enough replica nodes to satisfy the [consistency level](#). In rapid read protection, Cassandra sends out extra read requests to other replicas, even after the consistency level has been met. The speculative retry property specifies the trigger for these extra read requests.

- **ALWAYS:** The coordinator node sends extra read requests to all other replicas after every read of that table.
- **Xpercentile:** Cassandra constantly tracks each table's typical read latency (in milliseconds). Set speculative retry to `xpercentile` to tell the coordinator node to retrieve the typical latency time of the table being read and calculate X percent of that figure. The coordinator sends redundant read requests if the number of milliseconds it waits without responses exceeds that calculated figure. (For example, if the `speculative_retry` property for `Table_A` is set to `80percentile`, and that table's typical latency is 60 milliseconds, the coordinator node handling a read of `Table_A` would send a normal read request first, and send out redundant read requests if it received no responses within 48ms, which is 80 % of 60ms.)

- **Nms**: The coordinator node sends extra read requests to all other replicas if the coordinator node has not received any responses within N milliseconds.
- **NONE**: The coordinator node does not send extra read requests after any read of that table.

For example:

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

Or:

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

compression

Configure compression by specifying the compression algorithm class followed by the subproperties in simple JSON format. 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.

```
compression = {
    'class' : 'compression_algorithm_name',
    'chunk_length_kb' : 'value',
    'crc_check_chance' : 'value',
    | 'sstable_compression' : ''
}
```

Table 42. compression subproperties

Subproperty	Description
class	<p>Sets the compressor name,</p> <p>The class name of the compression algorithm. Cassandra provides the following built-in classes:</p> <ul style="list-style-type: none"> • <code>LZ4Compressor</code>, see • <code>SnappyCompressor</code> • <code>DeflateCompressor</code>

Table 42. compression subproperties (continued)

Subproperty	Description
	Default: <code>LZ4Compressor</code> .
chunk_length_kb	Size (in KB) of the block. On disk, SSTables are compressed by block to allow random reads. 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. Default value: 64KB. Default value:.
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 checked during read. By default they are always checked. Set to 0 to disable checksum checking and to 0.5, for instance, to check them on every other read. Default: 1.0.
sstable_compression	Disables compression. Specify a null value.

To disable compression, specify the `sstable_compression` option with value of empty string (""):

```
ALTER TABLE cycling.cyclist_name
WITH COMPRESSION = {'sstable_compression': ''};
```

compaction

The compaction option in [CREATE TABLE](#) or [ALTER TABLE](#) WITH clause defines the strategy for cleaning up data after writes. Define a compaction class and properties in simple JSON format:

```
compaction = {
    'class' : 'compaction_strategy_name'
    [, 'subproperty_name' : 'value', ...]
}
```

Note:

For more guidance, see the [When to Use Leveled Compaction](#), [Leveled Compaction in Apache Cassandra](#) blog, and [How data is maintained](#).

compaction properties

Cassandra provides the following compaction classes, each class has different subproperties:

- [SizeTieredCompactionStrategy \(STCS\)](#)
- [DateTieredCompactionStrategy](#)
- [TimeWindowCompactionStrategy \(TWCS\)](#)
- [LeveledCompactionStrategy \(LCS\)](#)

SizeTieredCompactionStrategy (STCS)

Triggers a minor compaction when table meets the `min_threshold`. Minor compactations do not involve all the tables in a keyspace. See [SizeTieredCompactionStrategy](#) in the Cassandra documentation for more details.

Default compaction strategy.

Table 43. subproperties

Subproperty	Description
bucket_high	Size-tiered compaction merges sets of SSTables that are approximately the same size. Cassandra compares each SSTable size to the average of all SSTable sizes on the node. It merges SSTables whose size in KB are within [average-size × <code>bucket_low</code>] and [average-size × <code>bucket_high</code>]. Default value: <code>1 . 5</code>
bucket_low	See bucket_high . Default value: <code>0 . 5</code> .
enabled	Enables background compaction. Default value: <code>true</code> . See Enabling and disabling background compaction .
max_threshold	The maximum number of SSTables to allow in a minor compaction. Default value: <code>32</code> .

Table 43. subproperties (continued)

Subproperty	Description
min_threshold	The minimum number of SSTables to trigger a minor compaction. Default value: 4.
min_sstable_size	STCS groups SSTables into buckets. The bucketing process groups SSTables that differ in size by less than 50%. This bucketing process 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. Default value: 50MB.
only_purge_repaired_tombstones	<i>true</i> allows purging tombstones only from repaired SSTables. The purpose is to prevent data from resurrecting if repair is not run within <code>gc_grace_seconds</code> . If you do not run repair for a long time, Cassandra keeps all tombstones — this may cause problems. Default value: <code>false</code> .
tombstone_compaction_interval	The minimum number of seconds after which an SSTable is created before Cassandra considers the SSTable for tombstone compaction. An SSTable is eligible for tombstone compaction if the table exceeds the tombstone_threshold ratio. Default value: 86400.
tombstone_threshold	The ratio of garbage-collectable tombstones to all contained columns. If the ratio exceeds this limit, Cassandra starts compaction on that table alone, to purge the tombstones. Default value: 0.2.
unchecked_tombstone_compaction	<i>True</i> allows Cassandra to run tombstone compaction without pre-checking which tables are eligible for this operation. Even without this pre-check, Cassandra checks an SSTable to make sure it is safe to drop tombstones. Default value: <code>false</code> .

Note: Cassandra 3.0 does not support the `cold_reads_to omit` property for [SizeTieredCompactionStrategy](#).

Note: For more details, see [How data is maintained > Compaction Strategy > STCS](#)

DateTieredCompactionStrategy

Stores data written within a certain period of time in the same SSTable. Also see [DateTieredCompactionStrategy](#) in the Cassandra documentation.

Table 44. Subproperties

Subproperty	Description
base_time_seconds	The size of the first time window. Default value: 3600.
enabled	Enables background compaction. Default value: <code>true</code> . See Enabling and disabling background compaction .
max_sstable_age_days	Cassandra does not compact SSTables if its most recent data is older than this property. Fractional days can be set. Default value: 1000. Attention: This parameter is deprecated.
max_window_size_seconds	The maximum window size in seconds. Default value: 86400.
max_threshold	The maximum number of SSTables allowed in a minor compaction. Default value: 32.
min_threshold	The minimum number of SSTables that trigger a minor compaction. Default value: 4.
timestamp_resolution	Units, <i>MICROSECONDS</i> or <i>MILLISECONDS</i> , to match the timestamp of inserted data. Default value: <i>MICROSECONDS</i> .
tombstone_compaction_interval	The minimum number of seconds after an SSTable is created before Cassandra considers the SSTable for tombstone compaction. Tombstone compaction is triggered if the number of garbage-collectable tombstones in the SSTable is greater than tombstone_threshold . Default value: 86400.
tombstone_threshold	The ratio of garbage-collectable tombstones to all contained columns. If the ratio exceeds this limit, Cassandra starts compaction on that table alone, to purge the tombstones. Default value: 0 . 2.

Table 44. Subproperties (continued)

Subproperty	Description
unchecked_tombstone_compaction	<i>True</i> allows Cassandra to run tombstone compaction without pre-checking which tables are eligible for this operation. Even without this pre-check, Cassandra checks an SSTable to make sure it is safe to drop tombstones. Default value: <code>false</code> .

TimeWindowCompactionStrategy (TWCS)

Compacts SSTables using a series of *time windows* or *buckets*. TWCS creates a new time window within each successive time period. During the active time window, TWCS compacts all SSTables flushed from memory into larger SSTables using STCS. At the end of the time period, all of these SSTables are compacted into a single SSTable. Then the next time window starts and the process repeats. See [TimeWindowCompactionStrategy](#) in the Cassandra documentation for more details.

Table 45. Subproperties

Compaction Subproperties	Description
compaction_window_-unit	Time unit used to define the bucket size, milliseconds, seconds, hours, etc. Default value: milliseconds.
compaction_window_-size	Units per bucket.

LeveledCompactionStrategy (LCS)

Creates SSTables of a fixed, relatively small size (160 MB by default) that are grouped into levels. Within each level, SSTables are guaranteed to be non-overlapping. Each level (L0, L1, L2 and so on) is 10 times as large as the previous. Disk I/O is more uniform and predictable on higher than on lower levels as SSTables are continuously being compacted into progressively larger levels. At each level, row keys are merged into non-overlapping SSTables in the next level. See [LeveledCompactionStrategy \(LCS\)](#) in the Cassandra documentation.

Table 46. Subproperties

Subproperties	Default	Description
enabled	true	Enables background compaction. See Enabling and disabling background compaction below.
sstable_size_in_mb	160MB	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 that compaction may produce a larger SSTable during compaction. This occurs when data for a given partition key is exceptionally large. Cassandra does not split the data into two SSTables.
tombstone_compaction_interval	86400 (one day)	The minimum number of seconds after an SSTable is created before Cassandra considers the SSTable for tombstone compaction. Cassandra begins tombstone compaction SSTable's tombstone_threshold exceeds value of the following property.
tombstone_threshold	0.2	The ratio of garbage-collectable tombstones to all contained columns. If the ratio exceeds this limit, Cassandra starts compaction on that table alone, to purge the tombstones.
unchecked_tombstone_compaction	false	<i>True</i> allows Cassandra to run tombstone compaction without pre-checking which tables are eligible for this operation. Even without this pre-check, Cassandra checks an SSTable to make sure it is safe to drop tombstones.

Enabling and disabling background compaction

The following example sets the *enable* property to disable background compaction:

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

```
'enabled': 'false' }
```

Disabling background compaction can be harmful: without it, Cassandra does not regain disk space, and may allow [zombies](#) to propagate. Although compaction uses I/O, it is better to leave it enabled in most cases.

Table keywords

CLUSTERING ORDER BY (column_name ASC | DESC)

Order rows storage to make use of the on-disk sorting of columns. Specifying order can make query results more efficient. Options are:

ASC: ascending (default order)

DESC: descending, reverse order

The following example shows a table definition that changes the clustering order to descending by insertion time.

The following example shows a table definition stores the categories with the highest points first.

```
CREATE TABLE cycling.cyclist_category (
    category text,
    points int,
    id UUID,
    lastname text,
    PRIMARY KEY (category, points))
WITH CLUSTERING ORDER BY (points DESC);
```

COMPACT STORAGE

Use [COMPACT STORAGE](#) to store data in the legacy (Thrift) storage engine format to conserve disk space.

Use compact storage for the category table.

```
CREATE TABLE cycling.cyclist_category (
    category text,
    points int,
    id UUID,
    lastname text,
    PRIMARY KEY (category, points))
WITH CLUSTERING ORDER BY (points DESC)
AND COMPACT STORAGE;
```

Important: The [storage engine is much more efficient](#) at storing data, and compact storage is not necessary.

ID

If a table is accidentally dropped with `DROP TABLE`, use this option to recreate the table and run a commitlog replayer to retrieve the data.

```
CREATE TABLE users (
    userid text PRIMARY KEY,
    emails set<text>
) WITH ID='5a1c395e-b41f-11e5-9f22-ba0be0483c18';
```

Configuring read repairs

Cassandra performs [read repair](#) whenever a read reveals inconsistencies among replicas. You can also configure Cassandra to perform read repair after a completely consistent read. Cassandra compares and coordinates all replicas, even those that were not accessed in the successful read. The `dclocal_read_repair_chance` and `read_repair_chance` set the probability that a consistent read of a table triggers a read repair. The first of these properties sets the probability for a read repair that is confined to the same datacenter as the coordinator node. The second property sets the probability for a read repair across all datacenters that contain matching replicas. This cross-datacenter operation is much more resource-intensive than the local operation.

Recommendations: if the table is for time series data, both properties can be set to 0 (zero). For other tables, the more performant strategy is to set `dc_local_read_repair_chance` to 0.1 and `read_repair_chance` to 0. If you want to use `read_repair_chance`, set this property to 0.1.

CREATE TRIGGER

The implementation of triggers includes the capability to register a trigger on a table using the familiar `CREATE TRIGGER` syntax. The Trigger API is semi-private and subject to change.

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

Enclose trigger names that use uppercase characters in double quotation marks. The logic comprising the trigger can be written in any Java (JVM) language and exists outside the database. The Java class in this example that implements the trigger is named `org.apache.cassandra.triggers` and defined in an [Apache repository](#). The trigger defined on a table fires before a requested DML statement occurs to ensure the atomicity of the transaction.

Place the custom trigger code (JAR) in the `triggers` directory on every node. The custom JAR loads at startup. The location of triggers directory depends on the installation:

Cassandra supports lightweight transactions for creating a trigger. Attempting to create an existing trigger returns an error unless the `IF NOT EXISTS` option is used. If the option is used, the statement is a no-op if the table already exists.

Synopsis

```
CREATE TRIGGER IF NOT EXISTS trigger_name ON table_name
USING 'java_class'
```

Table 47. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key : value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2</i> >	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > ... </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.

Table 47. Legend (continued)

Syntax conventions	Description
<code>@xml_entity='xml_entity_type'</code>	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

CREATE TYPE

A [user-defined type](#) contains one or more typed fields of related information, such as address information: street, city, and postal code.

Synopsis

```
CREATE TYPE [ IF NOT EXISTS ]
keyspace_name.type_name(
  field_name cql_datatype[,]
  [field_name cql_datatype] [,...]
)
```

Table 48. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (' ') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.

Table 48. Legend (continued)

Syntax conventions	Description
{ key : value }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
<datatype1,datatype2>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
cql_statement ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' <schema> . . . </schema> '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@xml_entity='xml_entity_type'	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

IF NOT EXISTS

Suppresses the error if the type already exists in the keyspace. UDT scope is keyspace-wide.

type_name

Unique name for the type, CQL types are reserved for a list see [type names](#).

field_name cql_datatype

Define fields that are in the UDT in a comma separated list:

```
field_name cql_datatype, field_name cql_datatype
```

Restriction: UDTs cannot contain counter fields.

Example

This example creates a user-defined type `cycling.basic_info` that consists of personal data about an individual cyclist.

```
CREATE TYPE cycling.basic_info (
    birthday timestamp,
    nationality text,
    weight text,
```

```
height text
);
```

After defining the UDT, you can create a table that has columns with the UDT. CQL collection columns and other columns support the use of user-defined types, as shown in [Using CQL examples](#).

CREATE ROLE

Create roles to manage access control to database resources, such as keyspaces, tables, functions. Use roles to:

- Define a set of permissions that can be assigned to other roles and mapped to external users.
- Create login accounts for [internal authentication](#). (Not recommended for production environments.)

Warning: A full access login account *cassandra* (password *cassandra*) is enabled by default; create your own full access role and drop the *cassandra* account.

Synopsis

```
CREATE ROLE [ IF NOT EXISTS] role_name
[WITH SUPERUSER = true | false
 | LOGIN = true | false
 | PASSWORD = 'password'
 | OPTIONS = option_map]
```

Table 49. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.

Table 49. Legend (continued)

Syntax conventions	Description
<code>...</code>	Repeatable. An ellipsis (<code>...</code>) indicates that you can repeat the syntax element as often as required.
<code>'Literal string'</code>	Single quotation (<code>'</code>) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
<code>{ key : value }</code>	Map collection. Braces (<code>{ }</code>) enclose map collections or key value pairs. A colon separates the key and the value.
<code><datatype1,datatype2></code>	Set, list, map, or tuple. Angle brackets (<code>< ></code>) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<code>cql_statement ;</code>	End CQL statement. A semicolon (<code>;</code>) terminates all CQL statements.
<code>[--]</code>	Separate the command line options from the command arguments with two hyphens (<code>--</code>). This syntax is useful when arguments might be mistaken for command line options.
<code>' <schema> ... </schema> '</code>	Search CQL only: Single quotation marks (<code>'</code>) surround an entire XML schema declaration.
<code>@xml_entity='xml_entity_type'</code>	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

role_name

Use a unique name for the role. Cassandra forces all names to lowercase; enclose in quotes to preserve case or use special characters in the name.

SUPERUSER

True automatically grants AUTHORIZE, CREATE and DROP permission on ALL ROLES.

Superusers can only manage roles by default. To manage other resources, you must grant the permission set to that resource. For example, to allow access management for all keyspaces: `GRANT ALL PERMISSIONS ON ALL KEYSPACES TO role_name.`

Default: false.

LOGIN

True allows the role to log in. Use true to create login accounts for [internal authentication](#) PasswordAuthenticator.

Default: false.

PASSWORD

Enclose the password in single quotes. Cassandra [internal authentication](#) requires a password.

OPTIONS = { *option_map* }

Reserved for use with authentication plug-ins. Refer to the authenticator documentation for details.

Examples

Creating a login account

1. Create a login role for coach.

```
CREATE ROLE coach  
WITH PASSWORD = 'All14One2day!'  
AND LOGIN = true;
```

Internal authentication requires the role to have a password.

2. Verify that the account works by logging in:

```
LOGIN coach
```

3. Enter the password at the prompt.

```
Password:
```

4. The cqlsh prompt includes the role name:

```
coach@cqlsh>
```

Creating a role

A best practice when using internal authentication is to create separate roles for permissions and login accounts. Once a role has been created it can be assigned as permission to another role, see GRANT for more details. Roles for externally authenticators users are mapped to the user's group name; LDAP mapping is case sensitive.

Create a role for the cycling keyspace administrator, that is a role that has full permission to only the cycling keyspace.

1. Create the role:

```
CREATE ROLE cycling_admin;
```

At this point the role has no permissions. Manage permissions using GRANT and REVOKE.

Note: A role can only modify permissions of another role and can only modify (GRANT or REVOKE) role permissions that it also has.

2. Assign the role full access to the cycling keyspace:

```
GRANT ALL PERMISSIONS on KEYSPACE cycling to cycling_admin;
```

3. Now assign the role to the coach.

```
GRANT cycling_admin TO coach;
```

This allows you to manage the permissions of all cycling administrators by modifying the cycling_admin role.

4. View the coach's permissions.

```
list all permissions of coach;
```

role	username	resource	permission
cycling_admin	cycling_admin	<keyspace cycling>	CREATE
cycling_admin	cycling_admin	<keyspace cycling>	ALTER
cycling_admin	cycling_admin	<keyspace cycling>	DROP
cycling_admin	cycling_admin	<keyspace cycling>	SELECT
cycling_admin	cycling_admin	<keyspace cycling>	MODIFY
cycling_admin	cycling_admin	<keyspace cycling>	AUTHORIZE
cycling_admin	cycling_admin	<all roles>	AUTHORIZE

Changing a password

A role can change the password to itself, or another role that it has permission to modify. A superuser can change the password of any role. Use ALTER to change a role's password:

```
ALTER ROLE coach WITH PASSWORD = 'NewPassword'
```

CREATE USER (Deprecated)

`CREATE USER` is supported for backwards compatibility only. Authentication and authorization are based on ROLES, and use `CREATE ROLE` instead.

`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. See [CREATE ROLE](#) for more information about `SUPERUSER` and `NOSUPERUSER`.

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](#).

Synopsis

```
CREATE USER [ IF NOT EXISTS] user_name
WITH PASSWORD 'password'
[ SUPERUSER | NOSUPERUSER]
```

Table 50. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (. . .) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key</i> : <i>value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1</i> , <i>datatype2</i> >	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.

Table 50. Legend (continued)

Syntax conventions	Description
<code>cql_statement ;</code>	End CQL statement. A semicolon (<code>;</code>) terminates all CQL statements.
<code>[--]</code>	Separate the command line options from the command arguments with two hyphens (<code>--</code>). This syntax is useful when arguments might be mistaken for command line options.
<code>' <schema> . . . </schema> '</code>	Search CQL only: Single quotation marks (<code>'</code>) surround an entire XML schema declaration.
<code>@xml_entity='xml_entity_type'</code>	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Examples

Creating internal user accounts

Use `WITH PASSWORD` to create 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, `WITH PASSWORD` is not required.

```
CREATE USER test NOSUPERUSER;
```

Creating a user account conditionally

In Test that the user does not have an account before attempting to create one. Attempting to create an existing user results in an invalid query condition unless the `IF NOT EXISTS` option is used. If the option is used, the statement will be a no-op if the user exists.

```
$ bin/cqlsh -u cassandra -p cassandra
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 2.1.0 | CQL spec 3.3.0 | Native protocol v3]
Use HELP for help.

cqlsh> CREATE USER newuser WITH PASSWORD 'password';

cqlsh> CREATE USER newuser WITH PASSWORD 'password';
code=2200 [Invalid query] message="User newuser already exists"

cqlsh> CREATE USER IF NOT EXISTS newuser WITH PASSWORD 'password';
```

```
cqlsh>
```

DELETE

Removes data from one or more selected columns (data is replaced with null) or removes the entire row when no column is specified. Cassandra deletes data in each selected partition atomically and in isolation.

Deleted data is not removed from disk immediately. Cassandra marks the deleted data with a tombstone and then removes it after the grace period.

CAUTION: Using delete may impact performance.

Synopsis

```
DELETE [column_name (term)][, ...]
  FROM [keyspace_name.] table_name
  [USING TIMESTAMP timestamp_value]
  WHERE PK_column_conditions
  [IF EXISTS | IF static_column_conditions]
```

Table 51. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.

Table 51. Legend (continued)

Syntax conventions	Description
{ key : value }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
<datatype1,datatype2>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
cql_statement ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' <schema> . . . </schema> '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@xml_entity='xml_entity_type'	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

column_name

Set column to delete or use a comma separated list of columns. When no column is specified the entire row is deleted.

term

Element identifier for collection types, where:

list	Index number of item, where 0 is the first.
map	Element key of item.

timestamp_value

Deletes values older than the **timestamp_value**.

PK_column_conditions

Syntax to match PRIMARY KEY values. Separate multiple conditions with AND.

Restriction:

- Only equals (=) or IN are supported.
- Ranges (IN) are not supported when specifying a static column condition, see [IF condition](#).
- When removing data from columns in matching rows, you must specify a condition for all primary keys.

IF EXISTS

Error when the statement results in no operation.

IF condition

Specify conditions for static fields to match. Separate multiple conditions with AND.

Restriction: Modifies the primary key statement, all primary keys required.

Examples

Delete data from row

Delete the data in specific columns by listing them after the DELETE command, separated by commas. Change the data in first and last name columns to null.

```
DELETE firstname, lastname FROM cycling.cyclist_name  
WHERE id = e7ae5cf3-d358-4d99-b900-85902fda9bb0;
```

Delete an entire row

Entering no column names after DELETE, removes the entire matching row. Remove a cyclist entry from the cyclist_name table and return an error if no rows match.

```
DELETE FROM cycling.cyclist_name  
WHERE id=e7ae5cf3-d358-4d99-b900-85902fda9bb0 IF EXISTS;
```

Delete row based on static column condition

`IF` limits the where clause, allowing selection based on values in non-PRIMARY KEY columns, such as first and last name; remove the cyclist record if the first and last name do not match.

```
DELETE FROM cycling.cyclist_name
WHERE id = e7ae5cf3-d358-4d99-b900-85902fda9bb0
  IF firstname='Alex' AND lastname='Smith';
```

The results show all the data

[applied]	firstname	lastname
False	Alex	FRAME

Conditionally deleting columns

Conditionally delete columns using `IF` or `IF EXISTS`. Deleting a column is similar to making an insert or update conditionally.

Add `IF EXISTS` to the command to ensure that the operation is not performed if the specified row does not exist:

```
DELETE id FROM cyclist_id
WHERE lastname = 'WELTEN' AND firstname = 'Bram'
  IF EXISTS;
```

Without `IF EXISTS`, the command proceeds with no standard output. If `IF EXISTS` returns true (if a row with this primary key does exist), standard output displays a table like the following:

[applied]
True

If no such row exists, however, the conditions returns FALSE and the command fails. In this case, standard output looks like:

[applied]
False

Use `IF` condition to apply tests to one or more column values in the selected row:

```
DELETE id FROM cyclist_id
WHERE lastname = 'WELTEN' AND firstname = 'Bram'
  IF age = 2000;
```

If all the conditions return TRUE, standard output is the same as if `IF EXISTS` returned true (see above). If any of the conditions fails, standard output displays `False` in the `[applied]` column and also displays information about the condition that failed:

[applied]	age
False	18

Conditional deletions incur a non-negligible performance cost and should be used sparingly.

Deleting old data using TIMESTAMP

The `TIMESTAMP` is an integer representing microseconds. You can identify the column for deletion using `TIMESTAMP`.

```
DELETE firstname, lastname
  FROM cycling.cyclist_name
  USING TIMESTAMP 1318452291034
 WHERE lastname = 'VOS';
```

Deleting more than one row

The `WHERE` clause specifies which row or rows to delete from the table.

```
DELETE FROM cycling.cyclist_name
 WHERE id = 6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47;
```

To delete more than one row, use the keyword `IN` and supply a list of values in parentheses, separated by commas:

```
DELETE FROM cycling.cyclist_name
 WHERE firstname IN ('Alex', 'Marianne');
```

CQL supports an empty list of values in the `IN` clause, useful in Java Driver applications.

Deleting from a collection set, list or map

To delete an element from a map that is stored as one column in a row, specify the `column_name` followed by the key of the element in square brackets:

```
DELETE sponsorship [ 'sponsor_name' ] FROM cycling.races
 WHERE race_name = 'Critérium du Dauphiné';
```

To delete an element from a list, specify the `column_name` followed by the list index position in square brackets:

```
DELETE categories[3] FROM cycling.cyclist_history
 WHERE lastname = 'TIRALONGO';
```

To delete all elements from a set, specify the `column_name` by itself:

```
DELETE sponsorship FROM cycling.races
```

```
WHERE race_name = 'Criterium du Dauphine';
```

DROP AGGREGATE

Drop a user-defined aggregate.

Synopsis

```
DROP AGGREGATE [IF EXISTS] [keyspace_name.]aggregate_name
```

Table 52. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key</i> : <i>value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syn-

Table 52. Legend (continued)

Syntax conventions	Description
	tax is useful when arguments might be mistaken for command line options.
' <schema> . . . </schema> '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@xml_entity='xml_entity_type'	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Examples

Drop the avgState aggregate from the cycling keyspace.

```
DROP AGGREGATE IF EXISTS cycling.avgState;
```

DROP FUNCTION

Drop a user-defined function from a keyspace.

Restriction: You cannot drop functions that are in used in an AGGREGATE, redefine or drop the AGGREGATE first.

Synopsis

```
DROP FUNCTION [IF EXISTS] [keyspace_name.]function_name
```

Table 53. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.

Table 53. Legend (continued)

Syntax conventions	Description
<code>...</code>	Repeatable. An ellipsis (<code>...</code>) indicates that you can repeat the syntax element as often as required.
<code>'Literal string'</code>	Single quotation (<code>'</code>) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
<code>{ key : value }</code>	Map collection. Braces (<code>{ }</code>) enclose map collections or key value pairs. A colon separates the key and the value.
<code><datatype1,datatype2></code>	Set, list, map, or tuple. Angle brackets (<code>< ></code>) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<code>cql_statement ;</code>	End CQL statement. A semicolon (<code>;</code>) terminates all CQL statements.
<code>[--]</code>	Separate the command line options from the command arguments with two hyphens (<code>--</code>). This syntax is useful when arguments might be mistaken for command line options.
<code>' <schema> ... </schema> '</code>	Search CQL only: Single quotation marks (<code>'</code>) surround an entire XML schema declaration.
<code>@xml_entity='xml_entity_type'</code>	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Examples

Drops the UDF from the cycling keyspace.

```
cqlsh> DROP FUNCTION IF EXISTS cycling.fLog;
```

DROP INDEX

Removes an existing index. The default index name is `table_name_column_name_idx`.

Synopsis

```
DROP INDEX [IF EXISTS] [keyspace.]index_name
```

Table 54. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key : value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > ... </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.

Table 54. Legend (continued)

Syntax conventions	Description
<code>@xml_entity='xml_entity_type'</code>	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Example

Drop the index `ryear` from the `cycling.rank_by_year_and_name` table.

```
DROP INDEX cycling.ryear;
```

DROP KEYSPACE

Immediate, irreversible removal of the keyspace, including objects such as tables, functions, and data it contains.

Tip: Cassandra takes a snapshot before dropping the keyspace, see [Restoring from a snapshot](#) for recovery details.

Synopsis

```
DROP KEYSPACE [ IF EXISTS ] keyspace_name
```

Table 55. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
<code>[]</code>	Optional. Square brackets (<code>[]</code>) surround optional command arguments. Do not type the square brackets.
<code>()</code>	Group. Parentheses (<code>()</code>) identify a group to choose from. Do not type the parentheses.
<code> </code>	Or. A vertical bar (<code> </code>) separates alternative elements. Type any one of the elements. Do not type the vertical bar.
<code>...</code>	Repeatable. An ellipsis (<code>...</code>) indicates that you can repeat the syntax element as often as required.

Table 55. Legend (continued)

Syntax conventions	Description
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key : value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > . . . </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@ <i>xml_entity</i> =' <i>xml_entity_type</i> '	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Example

Drop the cycling keyspace:

```
DROP KEYSPACE cycling;
```

DROP MATERIALIZED VIEW

Immediate, irreversible removal of a materialized view, including all data it contains. This operation has no effect on the base table.

Restriction: Drop all materialized views associated with a base table before dropping the table.

Synopsis

```
DROP MATERIALIZED VIEW [ IF EXISTS] [keyspace_name.] view_name
```

Table 56. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key : value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > ... </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.

Table 56. Legend (continued)

Syntax conventions	Description
<code>@xml_entity='xml_entity_type'</code>	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

IF EXISTS

Cassandra checks on whether the specified materialized view exists. If the materialized view does not exist, the operation fails. Optional.

keyspace_name

To drop a materialized view in a keyspace other than the current keyspace, put the keyspace name in front of the materialized view name, followed by a period.

view_name

The name of the materialized view to drop

Example

```
DROP MATERIALIZED VIEW cycling.cyclist_by_age;
```

Related information

[Creating a materialized view](#)
[CREATE MATERIALIZED VIEW](#)
[ALTER MATERIALIZED VIEW](#)

DROP ROLE

Removes an existing role. Enclose role names with special characters and capitalization in single quotation marks.

Restriction: The role used to drop roles must have `DROP` permission, directly or on `ALL ROLES` or the selected role. Only superuser roles can drop another superuser role. A role can never drop their own role.

Synopsis

```
DROP ROLE [IF EXISTS] role_name
```

Table 57. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.

Table 57. Legend (continued)

Syntax conventions	Description
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (. . .) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key</i> : <i>value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > . . . </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@ <i>xml_entity</i> =' <i>xml_entity_type</i> '	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Examples

Drop the team manager role.

```
DROP ROLE IF EXISTS team_manager;
```

DROP TABLE

Immediate, irreversible removal of a table, including all data contained in the table.

Note: The alias DROP COLUMNFAMILY has been deprecated.

Restriction: Drop all [materialized views](#) associated with the table before dropping the table.

Synopsis

```
DROP TABLE [IF EXISTS] keyspace_name.table_name
```

Table 58. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (. . .) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key : value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.

Table 58. Legend (continued)

Syntax conventions	Description
<code><datatype1,datatype2></code>	Set, list, map, or tuple. Angle brackets (<code>< ></code>) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<code>cql_statement ;</code>	End CQL statement. A semicolon (<code>;</code>) terminates all CQL statements.
<code>[--]</code>	Separate the command line options from the command arguments with two hyphens (<code>--</code>). This syntax is useful when arguments might be mistaken for command line options.
<code>' <schema> . . . </schema> '</code>	Search CQL only: Single quotation marks (<code>'</code>) surround an entire XML schema declaration.
<code>@xml_entity='xml_entity_type'</code>	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Example

Attempting to drop a table with materialized views that are based on it:

```
DROP TABLE cycling.cyclist_mv ;
```

Error message lists the materialized views that are based on this table:

```
InvalidRequest: Error from server: code=2200 [Invalid query]
message="Cannot drop table when materialized views still depend on it
(cycling.{cyclist_by_age,cyclist_by_country})"
```

Drop the cyclist_name table:

```
DROP TABLE cycling.cyclist_name;
```

DROP TRIGGER

Removes the trigger registration

The Trigger API is semi-private and subject to change.

Synopsis

```
DROP TRIGGER [IF EXISTS] trigger_name ON [keyspace.]table_name;
```

Table 59. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key : value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > ... </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.

Table 59. Legend (continued)

Syntax conventions	Description
<code>@xml_entity='xml_entity_type'</code>	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

DROP TYPE

Immediately and irreversibly removes a UDT (user-defined type).

Restriction: Dropping a user-defined type that is in use by a table or another type is not supported.

Synopsis

```
DROP TYPE [IF EXISTS] keyspace_name.type_name
```

Table 60. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key : value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.

Table 60. Legend (continued)

Syntax conventions	Description
<code><datatype1,datatype2></code>	Set, list, map, or tuple. Angle brackets (<code>< ></code>) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<code>cql_statement ;</code>	End CQL statement. A semicolon (<code>;</code>) terminates all CQL statements.
<code>[--]</code>	Separate the command line options from the command arguments with two hyphens (<code>--</code>). This syntax is useful when arguments might be mistaken for command line options.
<code>' <schema> . . . </schema> '</code>	Search CQL only: Single quotation marks (<code>'</code>) surround an entire XML schema declaration.
<code>@xml_entity='xml_entity_type'</code>	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Examples

Attempting to drop a type that is in use by a table:

```
DROP TYPE cycling.basic_info ;
```

Error message with the table names that contain the type:

```
InvalidRequest: Error from server: code=2200 [Invalid query]
message="Cannot drop user type cycling.basic_info as it is still used by
table cycling.cyclist_stats"
```

Drop the table:

```
DROP TABLE cycling.cyclist_stats ;
```

Drop the type:

```
DROP TYPE cycling.basic_info ;
```

DROP USER (Deprecated)

Remove a user.

Note: `DROP USER` is supported for backwards compatibility. Authentication and authorization are based on ROLES, and use `DROP ROLE`.

Synopsis

```
DROP USER [IF EXISTS] user_name
```

Table 61. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key : value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > ... </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.

Table 61. Legend (continued)

Syntax conventions	Description
<code>@xml_entity='xml_entity_type'</code>	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Description

`DROP USER` removes an existing user. Attempting to drop a user that does not exist results in an invalid query condition unless the `IF EXISTS` option is used. If the option is used, the statement will be a no-op if the user does not exist. A user must have appropriate permission to issue a `DROP USER` statement. Users cannot drop themselves.

Enclose the user name in single quotation marks only if it contains non-alphanumeric characters.

Examples

Drop a user if the user exists:

```
DROP USER IF EXISTS boone;
```

Drop a user:

```
DROP USER montana;
```

GRANT

Assigns privileges to roles on database resources, such as keyspaces, tables, functions.

Important: Permissions apply immediately, even to active client sessions.

Synopsis

```
GRANT privilege
ON resource_name
TO role_name
```

Note: Enclose the role name in single quotation marks if it contains special characters or capital letters.

Table 62. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.

Table 62. Legend (continued)

Syntax conventions	Description
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (. . .) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key</i> : <i>value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > . . . </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@ <i>xml_entity</i> =' <i>xml_entity_type</i> '	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

privilege

Permissions granted on a resource to a role; grant a privilege at any level of the resource hierarchy.

The full set of available privileges is:

- ALL PERMISSIONS
- ALTER
- AUTHORIZE
- CREATE
- DESCRIBE
- DROP
- EXECUTE
- MODIFY
- SELECT

resource_name

Cassandra database objects to which permissions are applied.

The full list of available objects is:

- ALL FUNCTIONS
- ALL FUNCTIONS IN KEYSPACE *keyspace_name*
- FUNCTION *function_name*
- ALL KEYSPACES
- KEYSPACE *keyspace_name*
- TABLE *table_name*
- ALL ROLES
- ROLE *role_name*

Access control matrix

Cassandra resources have modelled hierarchy. Grant permissions on a resource higher in the chain to automatically grant that same permission on all resources lower down.

- **Data resources:** ALL KEYSPACES > KEYSPACE > TABLE *table_name*.
- **Functions:** Includes user defined functions and aggregates, ALL FUNCTIONS > KEYSPACE > FUNCTION *function_name*.
- **Roles:** ALL ROLES > ROLE *role_name*.

Note: Types also belong to keyspaces but there are no privileges specific to user defined types.

Not all privileges apply to every type of resource. For instance, `EXECUTE` is only relevant in the context of functions and mbeans. Attempting to grant privileges on a resource that the permission is not applicable results in an error.

The following table shows the relationship between privileges and resources, and describes the resulting permissions.

Privilege	Resource	Permissions
ALL	<i>resource_name</i>	All operations that are applicable to the resource and its ancestors.
CREATE	ALL KEYSPACES	CREATE KEYSPACE and CREATE TABLE in any keyspace.
CREATE	KEYSPACE <i>keyspace-name</i>	CREATE TABLE in specified keyspace.
CREATE	ALL FUNCTIONS	CREATE FUNCTION in any keyspace and CREATE AGGREGATE in any keyspace.
CREATE	ALL FUNCTIONS IN KEYSPACE <i>keyspace-name</i>	CREATE FUNCTION and CREATE AGGREGATE in specified keyspace.
CREATE	ALL ROLES	CREATE ROLE
ALTER	ALL KEYSPACES	ALTER KEYSPACE and ALTER TABLE in any keyspace.
ALTER	KEYSPACE <i>keyspace-name</i>	ALTER KEYSPACE and ALTER TABLE in specified keyspace.
ALTER	TABLE <i>table_name</i>	ALTER TABLE specified table.
ALTER	ALL FUNCTIONS	CREATE FUNCTION and CREATE AGGREGATE, also replace existing.

Privilege	Resource	Permissions
ALTER	ALL FUNCTIONS IN KEYSPACE <i>keyspace_name</i>	CREATE FUNCTION and CREATE AGGREGATE: , also replace existing in specified keyspace
ALTER	FUNCTION <i>function_name</i>	CREATE FUNCTION and CREATE AGGREGATE, also replace existing.
ALTER	ALL ROLES	ALTER ROLE on any role
ALTER	ROLE <i>role_name</i>	ALTER ROLE specified role.
DROP	ALL KEYSPACES	DROP KEYSPACE and DROP TABLE in any keyspace
DROP	KEYSPACE <i>keyspace_name</i>	DROP TABLE in specified keyspace
DROP	TABLE <i>table_name</i>	DROP TABLE specified.
DROP	ALL FUNCTIONS	DROP FUNCTION and DROP AGGREGATE in any keyspace.
DROP	ALL FUNCTIONS IN KEYSPACE <i>keyspace_name</i>	DROP FUNCTION and DROP AGGREGATE in specified keyspace.
DROP	FUNCTION <i>function_name</i>	DROP FUNCTION specified function.
DROP	ALL ROLES	DROP ROLE on any role.
DROP	ROLE <i>role_name</i>	DROP ROLE specified role.
SELECT	ALL KEYSPACES	SELECT on any table.
SELECT	KEYSPACE <i>keyspace_name</i>	SELECT on any table in specified keyspace.
SELECT	TABLE <i>table_name</i>	SELECT on specified table.
SELECT	ALL MBEANS	Call getter methods on any mbean.

Privilege	Resource	Permissions
SELECT	MBEANS <i>pattern</i>	Call getter methods on any mbean matching a wild-card pattern.
SELECT	MBEAN <i>mbean_name</i>	Call getter methods on named mbean.
MODIFY	ALL KEYSPACES	INSERT, UPDATE, DELETE and TRUNCATE on any table.
MODIFY	KEYSPACE <i>keyspace-name</i>	INSERT, UPDATE, DELETE and TRUNCATE on any table in specified keyspace.
MODIFY	TABLE <i>table_name</i>	INSERT, UPDATE, DELETE and TRUNCATE on specified table.
MODIFY	ALL MBEANS	Call setter methods on any mbean.
MODIFY	MBEANS <i>pattern</i>	Call setter methods on any mbean matching a wild-card pattern.
MODIFY	MBEAN <i>mbean_name</i>	Call setter methods on named mbean.
AUTHORIZE	ALL KEYSPACES	GRANT PERMISSION and REVOKE PERMISSION on any table.
AUTHORIZE	KEYSPACE <i>keyspace-name</i>	GRANT PERMISSION and REVOKE PERMISSION on any table in specified keyspace.
AUTHORIZE	TABLE <i>table_name</i>	GRANT PERMISSION and REVOKE PERMISSION on specified table.
AUTHORIZE	ALL FUNCTIONS	GRANT PERMISSION and REVOKE PERMISSION on any function.

Privilege	Resource	Permissions
AUTHORIZE	ALL FUNCTIONS IN KEYSpace <i>keyspace_name</i>	GRANT PERMISSION and REVOKE PERMISSION in specified keyspace.
AUTHORIZE	FUNCTION <i>function_name</i>	GRANT PERMISSION and REVOKE PERMISSION on specified function.
AUTHORIZE	ALL ROLES	GRANT ROLE and REVOKE ROLE on any role.
AUTHORIZE	ROLES	GRANT ROLE and REVOKE ROLE on specified roles
DESCRIBE	ALL ROLES	LIST ROLES on all roles or only roles granted to another, specified role.
DESCRIBE	ALL MBEANS	Retrieve metadata about any mbean from the platform's MBeanServer.
EXECUTE	ALL FUNCTIONS	SELECT, INSERT and UPDATE using any function, and use of any function in CREATE AGGREGATE.
EXECUTE	ALL FUNCTIONS IN KEYSpace <i>keyspace_name</i>	SELECT, INSERT and UPDATE using any function in specified keyspace and use of any function in keyspace in CREATE AGGREGATE.
EXECUTE	FUNCTION <i>function_name</i>	SELECT, INSERT and UPDATE using specified function and use of the function in CREATE AGGREGATE.
<i>role_name</i>	<i>resource_name</i>	Roles are a collection of privileges; grant all the privileges in a role on a any resource.

Examples

In most environments, user authentication is handled by a plug-in that verifies users credentials against an external directory service such as LDAP. The CQL role is mapped to the external group by matching the role name to a group name. For simplicity, these examples use internal users.

Give the role **coach** permission to perform `SELECT` queries on all tables in all keyspaces:

```
GRANT SELECT ON ALL KEYSPACES TO coach;
```

Give the role **manager** permission to perform `INSERT`, `UPDATE`, `DELETE` and `TRUNCATE` queries on all tables in the `field` keyspace.

```
GRANT MODIFY ON KEYSPACE field TO manager;
```

Give the role **coach** permission to perform `ALTER KEYSPACE` queries on the **cycling** keyspace, and also `ALTER TABLE`, `CREATE INDEX` and `DROP INDEX` queries on all tables in **cycling** keyspace:

```
GRANT ALTER ON KEYSPACE cycling TO coach;
```

Give the role **coach** permission to run all types of queries on **cycling.name** table.

```
GRANT ALL PERMISSIONS ON cycling.name TO coach;
```

Create an administrator role with full access to `cycling`.

```
GRANT ALL ON KEYSPACE cycling TO cycling_admin;
```

INSERT

Inserts an entire row or upserts data into an existing row, using the full [primary key](#). Requires a value for each component of the primary key, but not for any other columns. Missing values are set to null.

`INSERT` returns no results unless `IF NOT EXISTS` is used.

Restriction:

- Insert does not support counter columns use [UPDATE](#) instead.
- A [PRIMARY KEY](#) consists of a the [partition key](#) followed by the clustering columns. You can only insert values smaller than 64 kB into a clustering column.

Synopsis

```
INSERT INTO [keyspace_name.] table_name (column_list)
```

```
VALUES (column_values)
[IF NOT EXISTS]
[USING TTL seconds | TIMESTAMP epoch_in_microseconds]
```

Table 63. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key</i> : <i>value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > ... </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.

Table 63. Legend (continued)

Syntax conventions	Description
<code>@xml_entity='xml_entity_type'</code>	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

column_list

Comma separated list of columns. All PRIMARY KEY fields are required.
Nulls are inserted into any static columns that are excluded.

column_values

For each column, enter the corresponding list of values. Use the same order as the [column_list](#).

Enter data using the following syntax:

- a [literal](#)
- a collection:

Type	Description
set	Enter values between curly braces: <code>{ literal [, ...] }</code>
list	Enter values between square brackets: <code>[literal [, ...]]</code>
map	Enter values between curly braces: <code>{ key : value [, ...] }</code>

TTL seconds

Set [TTL](#) in seconds. After TTL expires, inserted data is automatically marked as deleted (with a tombstone). The TTL settings applies only to the inserted data, not the entire column. Any subsequent updates to the column resets the TTL. By default, values never expire.

You can set a default TTL for an entire table by setting the table's [default_time_to_live](#) property. Setting TTL on a column using the INSERT or UPDATE command overrides the table TTL.

IF NOT EXISTS

Inserts a new row of data if no rows match the PRIMARY KEY values.

TIMESTAMP epoch_in_microseconds

Marks inserted data (write time) with TIMESTAMP. Enter the time since epoch (January 1, 1970) in microseconds. By default, Cassandra uses the actual time of write.

Restriction: INSERT does not support IF NOT EXISTS and USING TIMESTAMP in the same statement.

Examples

Specifying TTL and TIMESTAMP

Insert a cyclist name using both a TTL and timestamp.

```
INSERT INTO cycling.cyclist_name (id, lastname, firstname)
VALUES (6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47, 'KRIJKSWIJK', 'Steven')
USING TTL 86400 AND TIMESTAMP 123456789;
```

- Time-to-live ([TTL](#)) in seconds
- Timestamp in microseconds since epoch

Inserting values into a collection (set and map)

To insert data into a collection, enclose values in curly brackets. Set values must be unique. Example: insert a list of categories for a cyclist.

```
INSERT INTO cycling.cyclist_categories (id, lastname, categories)
VALUES(
  '6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47',
  'KRIJKSWIJK',
  {'GC', 'Time-trial', 'Sprint'});
```

Insert a map named `teams` that lists two recent team memberships for the user VOS.

```
INSERT INTO cycling.cyclist_teams (id, lastname, teams)
VALUES(
  5b6962dd-3f90-4c93-8f61-eabfa4a803e2,
  'VOS',
  { 2015 : 'Rabobank-Liv Woman Cycling Team',
    2014 : 'Rabobank-Liv Woman Cycling Team' } );
```

The size of one item in a collection is limited to 64K.

To insert data into a collection column of a user-defined type, enclose components of the type in parentheses within the curly brackets, as shown in "[Using a user-defined type](#)."

Inserting a row only if it does not already exist

Add IF NOT EXISTS to the command to ensure that the operation is not performed if a row with the same primary key already exists:

```
INSERT INTO cycling.cyclist_name (id, lastname, firstname)
    VALUES (c4b65263-fe58-4846-83e8-f0e1c13d518f, 'RATTO', 'Rissella')
IF NOT EXISTS;
```

Without IF NOT EXISTS, the command proceeds with no standard output. If IF NOT EXISTS returns true (if there is no row with this primary key), standard output displays a table like the following:

```
[applied]
-----
True
```

If, however, the row does already exist, the command fails, and standard out displays a table with the value `false` in the [applied] column, and the values that were not inserted, as in the following example:

[applied]	id	firstname	lastname
False	c4b65263-fe58-4846-83e8-f0e1c13d518f	Rissella	RATTO

Note: Using IF NOT EXISTS incurs a performance hit associated with using Paxos internally. For information about Paxos, see [Cassandra 3.0 documentation](#).

LIST PERMISSIONS

Lists all permissions on all resources, a roles permissions on all resources or for a specified resource.

Restriction:

- Only superusers can list all permissions.
- Requires DESCRIBE permission on the target resources and roles.

Synopsis

```
LIST privilege
[ON resource_name]
[OF role_name]
```

[NORECURSIVE]

Table 64. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key</i> : <i>value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > ... </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.

Table 64. Legend (continued)

Syntax conventions	Description
<code>@xml_entity='xml_entity_type'</code>	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Tip: Omit `ON resource_name` to display all related resources or `OF role_name` to display all related roles.

privilege

Permissions granted on a resource to a role; grant a privilege at any level of the resource hierarchy.

The full set of available privileges is:

- ALL PERMISSIONS
- ALTER
- AUTHORIZE
- CREATE
- DESCRIBE
- DROP
- EXECUTE
- MODIFY
- SELECT

resource_name

Cassandra database objects to which permissions are applied.

The full list of available objects is:

- ALL FUNCTIONS
- ALL FUNCTIONS IN KEYSPACE `keyspace_name`
- FUNCTION `function_name`
- ALL KEYSPACES
- KEYSPACE `keyspace_name`
- TABLE `table_name`
- ALL ROLES

- ROLE *role_name*

role_name

Selects a role. If the role name has capital letters or special characters enclose it in single quotes.

NORECURSIVE

Only display permissions granted to the role. By default permissions checks are recursive; it shows direct and inherited permissions.

Example

List all permissions given to **coach**:

```
LIST ALL
OF coach;
```

Output is:

rolename	resource	permission
coach	<keyspace field>	MODIFY

List permissions given to all the roles:

```
LIST ALL;
```

Output is:

rolename	resource	permission
coach	<keyspace field>	MODIFY
manager	<keyspace cyclist>	ALTER
manager	<table cyclist.name>	CREATE
manager	<table cyclist.name>	ALTER
manager	<table cyclist.name>	DROP
manager	<table cyclist.name>	SELECT
manager	<table cyclist.name>	MODIFY
manager	<table cyclist.name>	AUTHORIZE
coach	<all keyspaces>	SELECT

List all permissions on the **cyclist.name** table:

```
LIST ALL
ON cyclist.name;
```

Output is:

username	resource	permission
manager	<table cyclist.name>	CREATE
manager	<table cyclist.name>	ALTER
manager	<table cyclist.name>	DROP
manager	<table cyclist.name>	SELECT
manager	<table cyclist.name>	MODIFY
manager	<table cyclist.name>	AUTHORIZE
coach	<all keyspaces>	SELECT

List all permissions on the **cyclist.name** table and its parents:

```
LIST ALL
ON cyclist.name
NORECURSIVE;
```

Output is:

username	resource	permission
manager	<table cyclist.name>	CREATE
manager	<table cyclist.name>	ALTER
manager	<table cyclist.name>	DROP
manager	<table cyclist.name>	SELECT
manager	<table cyclist.name>	MODIFY
manager	<table cyclist.name>	AUTHORIZE

LIST ROLES

Lists roles and shows superuser and login status.

Restriction: Roles have describe permission on their own and any inherited roles. Describe permission is required to list a role other than the current role.

Synopsis

```
LIST ROLES
[OF role_name]
```

[NORECURSIVE]

Table 65. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key</i> : <i>value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > ... </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.

Table 65. Legend (continued)

Syntax conventions	Description
<code>@xml_entity='xml_entity_type'</code>	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Examples

Show all the roles that the current role has permission to describe.

```
LIST ROLES;
```

will return the output:

role	super	login	options
cassandra	True	True	{}
sysadmin	True	True	{}
team_manager	True	False	{}

Show the roles for a particular role. Sufficient privileges are required to show this information.

```
LIST ROLES  
OF manager;
```

LIST USERS (Deprecated)

Lists internally authenticated users, created users with the command [CREATE USER](#), and have not yet changed the default user..

Note: `LIST USERS` is supported for backwards compatibility. Authentication and authorization are based on `ROLES`, and `LIST ROLES` should be used.

Synopsis

LIST USERS

Table 66. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key : value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > ... </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.

Table 66. Legend (continued)

Syntax conventions	Description
<code>@xml_entity='xml_entity_type'</code>	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Example

List the current users:

```
LIST USERS;
```

Output is:

name	super
cassandra	True
boone	False
akers	True
spillman	False

REVOKE

Remove privileges on database objects, resources, from a role.

Synopsis

```
REVOKE privilege
ON resource_name
FROM role_name
```

Table 67. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
<code>[]</code>	Optional. Square brackets (<code>[]</code>) surround optional command arguments. Do not type the square brackets.
<code>()</code>	Group. Parentheses (<code>()</code>) identify a group to choose from. Do not type the parentheses.

Table 67. Legend (continued)

Syntax conventions	Description
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key</i> : <i>value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > ... </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@ <i>xml_entity</i> =' <i>xml_entity_type</i> '	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

privilege

Permissions granted on a resource to a role; grant a privilege at any level of the resource hierarchy.

The full set of available privileges is:

- ALL PERMISSIONS
- ALTER
- AUTHORIZE
- CREATE

- DESCRIBE
- DROP
- EXECUTE
- MODIFY
- SELECT

resource_name

Cassandra database objects to which permissions are applied.

The full list of available objects is:

- ALL FUNCTIONS
- ALL FUNCTIONS IN KEYSPACE *keyspace_name*
- FUNCTION *function_name*
- ALL KEYSPACES
- KEYSPACE *keyspace_name*
- TABLE *table_name*
- ALL ROLES
- ROLE *role_name*

Example

```
REVOKE SELECT
ON cycling.name
FROM manager;
```

The role **manager** can no longer perform SELECT queries on the **cycling.name** table. Exceptions: Because of inheritance, the user can perform SELECT queries on **cycling.name** 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 **cycling** keyspace.

```
REVOKE ALTER
ON ALL ROLES
FROM coach;
```

The role **coach** can no longer perform GRANT, ALTER or REVOKE commands on all roles.

SELECT

Returns one or more rows from a single Cassandra table. Although a select statement without a where clause returns all rows from all partitions, it is not recommended.

Synopsis

```
SELECT * | select_expression | DISTINCT partition
FROM [keyspace_name.] table_name
[WHERE partition_value
    [AND clustering_filters
    [AND static_filters]]]
[ORDER BY PK_column_name ASC|DESC]
[LIMIT N]
[ALLOW FILTERING]
```

Table 68. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key : value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2</i> >	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.

Table 68. Legend (continued)

Syntax conventions	Description
<code>cql_statement ;</code>	End CQL statement. A semicolon (<code>;</code>) terminates all CQL statements.
<code>[--]</code>	Separate the command line options from the command arguments with two hyphens (<code>--</code>). This syntax is useful when arguments might be mistaken for command line options.
<code>' <schema> . . . </schema> '</code>	Search CQL only: Single quotation marks (<code>'</code>) surround an entire XML schema declaration.
<code>@xml_entity='xml_entity_type'</code>	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

*

Retrieves all the columns of data from each matched row.

DISTINCT *partition*

Returns unique values for the [partition key](#). Use a comma separate list

select_expression

Sets the column to retrieve from each row in a comma separated list. At least one expression is required.

Option	Description
<code>*</code>	Returns all column values.
<code>DISTINCT partition</code>	
<code>aggregate(arguments)</code>	Returns the selected column values and executes the aggregate to return a single result row. Use Cassandra aggregates , such as <code>count(column_name)</code> or user-defined aggregates .
<code>function(arguments)</code>	Executes a native Cassandra function or a user-defined function on each target value in the returned set. Cassandra native functions include <code>timeuuid</code> , a token function , or a blob conversion function .

AS *alias_name*

Replaces the column name in the result set to the alias name.

```
column_name AS alias
```

partition_value

Condition that restricts query to one or more partitions. Restriction for all parts of the partition key is required for compound keys. The supported conditional operators are equals (=) and IN(*value1,value2[,...]*)

Simple partition key, select a single partition:

```
partition_column = value
```

Simple partition key, select multiple partitions:

```
partition_column IN(value1,value2[,...])
```

For compound partition keys, create a condition for each key separated by AND:

```
partition_column1 = value1  
AND partition_column2 = value2 [AND ...])
```

clustering_filters**relation**

A logical expression. Cassandra returns only those rows that return true for each relation. A relation can consist of:

```
column_name operator term  
| (column_name [, column_name . . .] operator term-tuple  
| column_name IN ( term , term [, term] . . .)  
| (column_name , column_name [, column_name] . . .) IN  
term-tuple [, term-tuple] . . .)  
| TOKEN ( column_name ) operator TOKEN ( column_name )
```

operator

The logical symbol that specifies the relationship between the two sides of the relation. Casandra supports the following operators:

```
= | < | > | <= | >= | CONTAINS | CONTAINS KEY
```

term

- a constant: string, number, uuid, boolean, hex
- a function
- a collection:

set	<code>{ literal [, ...] }</code> (enclosed in curly brackets)
list	<code>[literal [, ...]]</code> (enclosed in square brackets)
map	<code>{ key : value [, ...] }</code> (enclosed in curly brackets)

- a set:
- a list:(note the use of square brackets)
- a map collection, a JSON-style array of literals:

```
{ literal : literal [, ...] }
```

term-tuple

```
( term, term, ... )
```

TOKEN

A Cassandra [token](#)

Examples

Specifying columns

The columns referenced in the SELECT clause must exist in the target table.

Columns in big data applications duplicate values. Use the DISTINCT keyword to return only distinct (different) values of partition keys.

The SELECT statement supports functions that perform calculations on the columns being returned. For details, see [Retrieving aggregate values](#)

Using a column alias

When your selection list includes functions or other complex expressions, use aliases to make the output more readable. This example applies aliases to the `dateOf(created_at)` and `blobAsText(content)` functions:

```
SELECT event_id,  
       dateOf(created_at) AS creation_date,  
       blobAsText(content) AS content  
    FROM timeline;
```

The output labels these columns with more understandable names:

event_id	creation_date	content
550e8400-e29b-41d4-a716	2013-07-26 10:44:33+0200	Some stuff

Specifying the source table using FROM

The `FROM` clause specifies the table to query. You may want to precede the table name with the name of the keyspace followed by a period (.). If you do not specify a keyspace, Cassandra queries the current keyspace.

The following example `SELECT` statement returns the number of rows in the `IndexInfo` table in the `system` keyspace:

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

Controlling the number of rows returned using LIMIT

The `LIMIT` option sets the maximum number of rows that the query returns:

```
SELECT lastname  
  FROM cycling.cyclist_name  
  LIMIT 50000;
```

Even if the query matches 105,291 rows, Cassandra only returns the first 50,000.

The `cqlsh` shell has a default row limit of 10,000. The Cassandra server and native protocol do not limit the number of returned rows, but they apply a timeout to prevent malformed queries from causing system instability.

Filtering data using WHERE

The `WHERE` clause introduces one or more relations that filter the rows returned by `SELECT`.

The column specifications

The column specification of the relation must be one of the following:

- One or more members of the partition key of the table
- A clustering column, only if the relation is preceded by other relations that specify all columns in the partition key
- A column that is [indexed](#) using CREATE INDEX.

Note:

In the `WHERE` clause, refer to a column using the actual name, not an alias.

Filtering on the partition key

For example, the following table definition defines `id` as the table's partition key:

```
CREATE TABLE cycling.cyclist_career_teams ( id UUID PRIMARY KEY, lastname
text, teams set<text> );
```

In this example, the SELECT statement includes in the partition key, so the WHERE clause can use the `id` column:

```
SELECT id, lastname, teams
FROM cycling.cyclist_career_teams
WHERE id=5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
```

Restriction: a relation that references the partition key can only use an equality operator — `=` or `IN`. For more details about the `IN` operator, see [Examples](#) below.

Filtering on a clustering column

Use a relation on a clustering column only if it is preceded by relations that reference all the elements of the partition key.

Example:

```
CREATE TABLE cycling.cyclist_points (
  id UUID,
  firstname text,
  lastname text,
  race_title text,
  race_points int,
  PRIMARY KEY (id, race_points));
```

```
SELECT sum(race_points)
FROM cycling.cyclist_points
```

```
WHERE id=e3b19ec4-774a-4d1c-9e5a-dececle30aac
      AND race_points > 7;
```

Output:

```
system.sum(race_points)
-----
195

(1 rows)
```

Filtering on indexed columns

A WHERE clause in a SELECT on an indexed table must include at least one equality relation to the indexed column. For details, see [Indexing a column](#).

Using the IN operator

Use `IN`, an equals condition operator, to list multiple possible values for a column. This example selects two columns, `first_name` and `last_name`, from three rows having employee ids (primary key) 105, 107, or 104:

```
SELECT first_name, last_name
  FROM emp
 WHERE empID IN (105, 107, 104);
```

The list can consist of a range of column values separated by commas.

Using IN to filter on a compound or composite primary key

Use an `IN` condition on the last column of the partition key only when it is preceded by equality conditions for all preceding columns of the partition key. For example:

```
CREATE TABLE parts (
    part_type text,
    part_name text,
    part_num int,
    part_year text,
    serial_num text,
    PRIMARY KEY ((part_type, part_name), part_num, part_year));
```

```
SELECT *
  FROM parts
```

```
WHERE part_type='alloy' AND part_name='hubcap'
AND part_num=1249 AND part_year IN ('2010', '2015');
```

When using `IN`, you can omit the equality test for clustering columns other than the last. But this usage may require the use of `ALLOW FILTERING`, so its performance can be unpredictable. For example:

```
SELECT *
FROM parts
WHERE part_num=123456 AND part_year IN ('2010', '2015')
ALLOW FILTERING;
```

CQL supports an empty list of values in the `IN` clause, useful in Java Driver applications when passing empty arrays as arguments for the `IN` clause.

When *not* to use IN

Under most conditions, using `IN` in relations on the partition key is not recommended. To process a list of values, the `SELECT` may have to query many nodes, which degrades performance. For example, consider a single local datacenter cluster with 30 nodes, a replication factor of 3, and a consistency level of `LOCAL_QUORUM`. A query on a single partition key query goes out to two nodes. But if the `SELECT` uses the `IN` condition, the operation can involve more nodes — up to 20, depending on where the keys fall in the token range.

Using `IN` for clustering columns is safer. See [Cassandra Query Patterns: Not using the “in” query for multiple partitions](#) for additional logic about using `IN`.

Filtering on collections

Your query can retrieve a collection in its entirety. It can also index the collection column, and then use the `CONTAINS` condition in the `WHERE` clause to filter the data for a particular value in the collection, or use `CONTAINS KEY` to filter by key. This example features a collection of tags in the **playlists** table. The query can index the tags, then filter on 'blues' in the tag set.

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

album	tags
Tres Hombres	{"1973", "blues"}

After [indexing the music venue map](#), filter on map values, such as 'The Fillmore':

```
SELECT *
FROM playlists
```

```
WHERE venue
CONTAINS 'The Fillmore';
```

After [indexing the collection keys](#) in the `venues` map, filter on map keys.

```
SELECT *
FROM playlists
WHERE venue CONTAINS KEY '2014-09-22 22:00:00-0700';
```

Filtering a map's entries

Follow this example query to retrieve rows based on map entries. (This method only works for maps.)

```
CREATE INDEX blist_idx
ON cycling.birthday_list (ENTRIES(blist));
```

This query finds all cyclists who are 23 years old based on their entry in the **blist** map of the table **birthday_list**.

```
SELECT *
FROM cycling.birthday_list
WHERE blist['age'] = '23';
```

Filtering a full frozen collection

This example presents a query on a table containing a **FROZEN** collection (set, list, or map). The query retrieves rows that fully match the collection's values.

```
CREATE INDEX rnumbers_idx
ON cycling.race_starts (FULL(rnumbers));
```

The following SELECT finds any cyclist who has 39 Pro wins, 7 Grand Tour starts, and 14 Classic starts in a frozen **list**.

```
SELECT *
FROM cycling.race_starts
WHERE rnumbers = [39,7,14];
```

Range relations

The [TOKEN function](#) may be used for range queries on the partition key.

Cassandra supports greater-than and less-than comparisons, but for a given partition key, the conditions on the [clustering column](#) are restricted to the filters that allow Cassandra to select a contiguous set 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:

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 set of rows, the WHERE clause must apply an equality condition to the `king` component of the primary key. The [ALLOW FILTERING](#) clause is also required. `ALLOW FILTERING` provides the capability to query the clustering columns using any condition.

CAUTION:

Only use `ALLOW FILTERING` for development! When you attempt a potentially expensive query, such as searching a range of rows, Cassandra displays this message:

```
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 this type of query, use `ALLOW FILTERING`, and restrict the output to `n` rows using `LIMIT n`. For example:

```
Select *
FROM ruling_stewards
```

```
WHERE king = 'none'
  AND reign_start >= 1500
  AND reign_start < 3000
LIMIT 10
ALLOW FILTERING;
```

Using `LIMIT` does not prevent all problems caused by `ALLOW FILTERING`. In this example, if there are no entries without a value for `king`, the `SELECT` scans the entire table, no matter what the `LIMIT` is.

It is not necessary to use `LIMIT` with `ALLOW FILTERING`, and `LIMIT` can be used by itself. But `LIMIT` can prevent a query from ranging over all partitions in a datacenter, or across multiple datacenters..

Comparing clustering columns

The partition key and clustering columns can be grouped and compared to values for [scanning a partition](#). For example:

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

The syntax used in the `WHERE` clause compares records of `steward_name` and `king` as a tuple against the `Boromir, Brego` tuple.

Using compound primary keys and sorting results

`ORDER BY` clauses can only work on a single column. That column must 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 using the `ASC` or `DESC` keywords (default is ascending).

In the `ORDER BY` clause, refer to a column using the actual name, not an alias.

For example, [set up the playlists table](#) (which uses a compound primary key), 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:

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

Or, create an index on playlist artists, and use this query to get titles of Fu Manchu songs on the playlist:

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

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

Output:

album	title
We Must Obey	Moving in Stereo
No One Rides for Free	Ojo Rojo

Displaying rows from an unordered partitioner with the TOKEN function

Use the TOKEN function to display rows based on the hash value of the partition key, for a single partition table. Selecting a slice using TOKEN values will only work with clusters that use the [ByteOrderedPartitioner](#).

For example, create this table:

```
CREATE TABLE cycling.last_3_days (
    race_name text,
    year timestamp,
    rank int,
    cyclist_name text,
    PRIMARY KEY (year, rank, cyclist_name)
);
```

After inserting data, SELECT using the TOKEN function to find the data using the partition key.

```
SELECT *
FROM cycling.last_3_days
WHERE TOKEN(year) < TOKEN('2015-05-26')
    AND year IN ('2015-05-24', '2015-05-25');
```

Computing aggregates

Cassandra provides standard built-in functions that return aggregate values to SELECT statements.

Using COUNT() to get the non-NULL value count for a column

A SELECT expression using COUNT(column_name) returns the number of non-NULL values in a column.

For example, count the number of last names in the `cyclist_name` table:

```
SELECT COUNT(lastname)
  FROM cycling.cyclist_name;
```

Getting the number of matching rows and aggregate values with COUNT()

A SELECT expression using `COUNT(*)` returns the number of rows that matched the query. Use `COUNT(1)` to get the same result. `COUNT(*)` or `COUNT(1)` can be used in conjunction with other aggregate functions or columns.

This example returns the number of rows in the `users` table:

```
SELECT COUNT(*)
  FROM users;
```

This example counts the number of rows and calculates the maximum value for *points* in the `users` table:

```
SELECT name, max(points), COUNT(*)
  FROM users;
```

Getting maximum and minimum values in a column

A SELECT expression using `MAX(column_name)` returns the maximum value in a column. If the column's datatype is numeric (`bigint`, `decimal`, `double`, `float`, `int`, `smallint`), this is the highest value.

```
SELECT MAX(points)
  FROM cycling.cyclist_category;
```

Output:

system.max(points)
1324

`MIN` returns the minimum value. If the query includes a `WHERE` clause, `MAX` or `MIN` returns the largest or smallest value from the rows that satisfy the `WHERE` condition.

```
SELECT category, MIN(points)
  FROM cycling.cyclist_category
 WHERE category = 'GC';
```

Output:

category system.min(points)
GC 1269

Note: If the column referenced by `MAX` or `MIN` has an `ascii` or `text` datatype, these functions return the last or first item in an alphabetic sort of the column values. If the specified column has datatype `date` or `timestamp`, these functions return the most recent or least recent times/dates. If the specified column has `null` values, the `MIN` function ignores it.

Note: Cassandra does not return a null value as the `MIN`.

Getting the sum or average of a column of numbers

Cassandra computes the sum or average of all values in a column when `SUM` or `AVG` is used in the `SELECT`

```
[cqlsh:cycling> select sum(points) from cyclist_category where category = 'Time-trial';
system.sum(points)
-----
412
(1 rows)

[cqlsh:cycling> select avg(points) from cyclist_category where category = 'Sprint';
system.avg(points)
-----
197
statement: (1 rows)
```

Note: If any of the rows returned has a null value for the column referenced for `AVG` aggregation, Cassandra includes that row in the row count, but uses a zero value to calculate the average.

Note: The `sum` and `avg` functions do not work with `text`, `uuid` or `date` fields.

Retrieving the date/time a write occurred

The `WRITETIME` function applied to a column returns the date/time in microseconds at which the column was written to the database.

For example, to retrieve the date/time that a write occurred to the `first_name` column of the user whose last name is Jones:

```
SELECT WRITETIME (first_name)
  FROM users
 WHERE last_name = 'Jones';
```

```
writetime(first_name)
-----
1353010594789000
```

The `WRITETIME` output in microseconds converts to November 15, 2012 at 12:16:34 GMT-8.

Retrieving the time-to-live of a column

The time-to-live (TTL) value of a cell is the number of seconds before the cell is marked with a tombstone. To set the TTL for a single cell, a column, or a column family, for example:

```
INSERT INTO cycling.calendar (race_id, race_name, race_start_date, race_end_date)
VALUES (200, 'placeholder', '2015-05-27', '2015-05-27')
USING TTL;
```

```
UPDATE cycling.calendar
USING TTL 300
SET race_name = 'dummy'
WHERE race_id = 200
    AND race_start_date = '2015-05-27'
    AND race_end_date = '2015-05-27';
```

After inserting the TTL, use SELECT statement to check its current value:

```
SELECT TTL(race_name)
FROM cycling.calendar
WHERE race_id=200;
```

Output:

```
ttl(race_name)
-----
276

(1 rows)
```

Retrieving values in the JSON format

For details, see [Retrieval using JSON](#)

TRUNCATE

Removes all data from the specified table immediately and irreversibly, and removes all data from any materialized views derived from that table.

Synopsis

```
TRUNCATE [TABLE] [keyspace_name.table_name]
```

Table 69. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.

Table 69. Legend (continued)

Syntax conventions	Description
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key : value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > ... </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@ <i>xml_entity</i> =' <i>xml_entity_type</i> '	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Examples

To remove all data from a table without dropping the table:

1. If necessary, use the cqlsh **CONSISTENCY** command to set the consistency level to **ALL**.
2. Use **nodetool status** or some other tool to make sure all nodes are up and receiving connections.
3. Use **TRUNCATE** or **TRUNCATE TABLE**, followed by the table name. For example:

```
TRUNCATE cycling.user_activity;
```

```
TRUNCATE TABLE cycling.user_activity;
```

Note: **TRUNCATE** sends a JMX command to all nodes, telling them to delete SSTables that hold the data from the specified table. If any of these nodes is down or doesn't respond, the command fails and outputs a message like the following:

```
truncate cycling.user_activity;
Unable to complete request: one or more nodes were unavailable.
```

UPDATE

Update columns in a row.

Synopsis

```
UPDATE [keyspace_name.] table_name
[USING TTL time_value | USING TIMESTAMP timestamp_value]
SET assignment [, assignment] . . .
WHERE row_specification
[IF EXISTS | IF condition [AND condition] . . .] ;
```

Table 70. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.

Table 70. Legend (continued)

Syntax conventions	Description
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key</i> : <i>value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > ... </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@ <i>xml_entity</i> =' <i>xml_entity_type</i> '	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Description

UPDATE writes one or more column values to a row in a Cassandra table. Like [INSERT](#), UPDATE is an [upsert](#) operation: if the specified row does not exist, the command creates it. All UPDATEs within the same partition key are applied atomically and in isolation.

The USING clause can add a *time to live* ([TTL](#)) value to the row. You cannot apply TTLs to counter columns.

Assign new values to the row's columns in the SET clause. UPDATE cannot update the values of a row's primary key fields. To update a counter in a counter table, specify the increment or decrement to the counter column.

Note: Unlike the INSERT command, the UPDATE command supports counters. Otherwise, the UPDATE and INSERT operations are identical.

The WHERE clause specifies the row or rows to be updated. To specify a row, the WHERE clause must provide a value for each column of the row's primary key. To specify more than one row, you can use the IN keyword to introduce a list of possible values. You can only do this for the last column of the primary key.

The IF EXISTS or IF keywords introduce a lightweight transaction:

```
UPDATE cycling.cyclist_name
SET comments =='Rides hard, gets along with others, a real winner'
WHERE id = fb372533-eb95-4bb4-8685-6ef61e994caa IF EXISTS;
```

Use the IF keyword to introduce a condition that must return TRUE for the update to succeed. Using an IF condition incurs a performance hit associated with using Paxos to support [linearizable consistency](#).

The UPDATE command does not return any result unless it includes IF EXISTS.

Parameters

keyspace_name

The name of the keyspace containing the table to be updated. Not needed if the keyspace has been set for the session with the USE command.

table_name

The name of the table to be updated.

time_value

The value for [TTL](#) is a number of seconds. Column values in a command marked with TTL are automatically marked as deleted (with a tombstone) after the specified number of seconds. The TTL applies to the marked column values, not the column itself. Any subsequent update of the column resets the value to the TTL specified in the update. By default, values never expire. You cannot set a time_value for data in a counter column.

You can set a default TTL for an entire table by setting the table's [default_time_to_live](#) property. Setting TTL on a column using the INSERT or UPDATE command overrides the table TTL.

In addition, you can delete a column's TTL by setting its time_value to zero.

timestamp_value

If [TIMESTAMP](#) is used, the inserted column is marked with its value – a timestamp in microseconds. If a [TIMESTAMP](#) value is not set, Cassandra uses the time (in microseconds) that the update occurred to the column.

assignment

Assigns a value to an existing element. Can be one of:

```

column_name = column_value [, column_name
= column_value] . . .
| counter_column_name = counter_column_name +
| - counter_offset
| list_name = ['list_item' [, 'list_item'] . . . ]
| list_name = list_name + | - ['list_item' [
'list_item'] . . . ]
| list_name = ['list_item' [, 'list_item'] . . . ] + list_name

| map_name = map_name + | - { map_key : map_value
[, map_key : map_value . . . ]}
| map_name[ index ] = map_value
| set_name = set_name + | - { ['set_item'] }

```

Variable	Description
<i>column_name</i>	The name of the column to be updated.
<i>column_value</i>	The value to be inserted for the specified column name.
<i>counter_column_name</i>	The name of the counter column to be updated.
<i>counter_offset</i>	The value by which the specified counter is be incremented or decremented (depending on whether the counter_offset is preceded by "=" or "-").
<i>list_name</i>	<p>The name of the list to be updated. Format of a list:</p> <div style="background-color: #f0f0f0; padding: 5px; margin-top: 10px;"> [list_item , list_item , list_item] </div> <p>Note the use of square brackets.</p>
<i>list_item</i>	The value to be added to the list, or removed from it.
<i>map_name</i>	<p>The name of the map to be updated. Format of a map:</p> <div style="background-color: #f0f0f0; padding: 5px; margin-top: 10px;"> { key : value , key: value , key: value . . . } </div> <p>Note the use of curly brackets ({}).</p>
<i>map_key</i>	The first term or key in a map entry.

Variable	Description
<i>map_value</i>	The second term or <i>value</i> in a map entry.
<i>set_name</i>	<p>The name of the set to be updated. Format of a set:</p> <pre>{ set_item , set_item , set_item . . . }</pre> <p>Note the use of curly brackets ({}).</p>
<i>set_item</i>	<p>The literal value included in a set.</p> <p>Note: The difference between a list and a set: each item in a set must be unique.</p>

row_specification

The WHERE clause must identify the row or rows to be updated by primary key:

- To specify one row, use `primary_key_name = primary_key_value`. If the primary key is a combination of elements, follow this with `AND primary_key_name = primary_key_value . . .`. The WHERE clause must specify a value for every component of the primary key.
- To specify more than one row, use `primary_key_name IN (primary_key_value, primary_key_value ...)`. This only works for the last component of the primary key.

Note: To update a static column, you only need to specify the partition key.

IF EXISTS | IF condition

An IF clause can limit the command's action on rows that match the WHERE clause:

- Use IF EXISTS to make the UPDATE fail when rows match the WHERE conditions.
- Use IF to specify one or more conditions that must test true for the values in the specified row or rows.

When an IF is used, the command returns a result to standard output. For examples, see [Conditionally updating columns](#).

Examples: updating a column

Update a column in several rows at once:

```
UPDATE users
  SET state = 'TX'
 WHERE user_uuid
   IN (88b8fd18-b1ed-4e96-bf79-4280797cba80,
       06a8913c-c0d6-477c-937d-6c1b69a95d43,
       bc108776-7cb5-477f-917d-869c12dfffa8);
```

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.

Update several columns in a single row:

```
UPDATE cycling.cyclists
  SET firstname = 'Marianne',
      lastname = 'VOS'
 WHERE id = 88b8fd18-b1ed-4e96-bf79-4280797cba80;
```

Update a row in a table with a complex primary key:

To do this, 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=cf66ccc-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;
```

Note: You cannot insert any value larger than 64K bytes into a clustering column.

Updating a counter column

To update a counter column value in a counter table, specify the increment or decrement to apply to the current value.

```
UPDATE cycling.popular_count SET popularity = popularity + 2 WHERE id =
  6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47;
```

To use a lightweight transaction on a counter column to ensure accuracy, put one or more counter updates in the [batch statement](#). For details, see [Performing conditional updates in a batch](#).

Creating a partition using UPDATE

Since Cassandra processes an UPDATE as an upsert, it is possible to create a new row by *updating* it in a table. Example: to create a new partition in the `cyclists` table, whose primary key is `(id)`, you can UPDATE the partition with id `e7cd5752-bc0d-4157-a80f-7523add8dbcd`, even though it does not exist yet:

```
UPDATE cycling.cyclists
SET firstname = 'Anna', lastname = 'VAN DER BREGGEN' WHERE id =
e7cd5752-bc0d-4157-a80f-7523add8dbcd;
```

Updating a list

To insert values into the list:

```
UPDATE cycling.upcoming_calendar
SET events = ['Criterium du Dauphine', 'Tour de Suisse'];
```

To prepend an element to the list, enclose it in square brackets and use the addition (+) operator:

```
UPDATE cycling.upcoming_calendar
SET events = ['Tour de France'] + events WHERE year=2015 AND month=06;
```

To append an element to the list, switch the order of the new element data and the list name:

```
UPDATE cycling.upcoming_calendar
SET events = events + ['Tour de France'] WHERE year=2017 AND month=05;
```

To add an element at a particular position, use the list index position in square brackets:

```
UPDATE cycling.upcoming_calendar
SET events[4] = 'Tour de France' WHERE year=2016 AND month=07;
```

To remove all elements having a particular value, use the subtraction operator (-) and put the list value in square brackets:

```
UPDATE cycling.upcoming_calendar
SET events = events - ['Criterium du Dauphine'] WHERE year=2016 AND
month=07;
```

To update data in a collection column of a user-defined type, enclose components of the type in parentheses within the curly brackets, as shown in "[Using a user-defined type.](#)"

CAUTION: The Java List Index is not thread safe. The `set` or `map` collection types are safer for updates.

Updating a set

To add an element to a set, use the UPDATE command and the addition (+) operator:

```
UPDATE cycling.cyclist_career_teams
SET teams = teams + {'Team DSB - Ballast Nedam'} WHERE
    id=5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
```

To remove an element from a set, use the subtraction (-) operator:

```
UPDATE cycling.cyclist_career_teams
SET teams = teams - {'DSB Bank Nederland bloeit'} WHERE
    id=5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
```

To remove all elements from a set:

```
UPDATE cycling.cyclist_career_teams
SET teams = {} WHERE id=5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
```

Updating a map

To set or replace map data, enclose the values in map syntax: strings in curly brackets, separated by a colon.

```
UPDATE cycling.upcoming_calendar
SET description = description + {'Criterium du Dauphine' : 'Easy race'}
    WHERE year = 2015;
```

To update or set a specific element, such as adding a new race to the calendar in a map named `events`:

```
UPDATE cycling.upcoming_calendar
SET events[2] = 'Vuelta Ciclista a Venezuela' WHERE year = 2016 AND month
    = 06;
```

To set the a TTL for each map element:

```
UPDATE cycling.upcoming_calendar USING TTL <ttl_value>
SET events[2] = 'Vuelta Ciclista a Venezuela' WHERE year = 2016 AND month
    = 06;
```

You can update the map by adding one or more elements separated by commas:

```
UPDATE cycling.upcoming_calendar
SET description = description + {'Criterium du Dauphine' : 'Easy race',
    'Tour du Suisse' : 'Hard uphill race'}
```

```
WHERE year = 2015 AND month = 6;
```

Remove elements from a map in the same way using - instead of +.

About updating sets and maps caution

CQL supports alternate methods for updating sets and maps. These alternatives may seem to accomplish the same tasks, but Cassandra handles them differently in important ways.

For example: CQL provides a straightforward method for creating a new row containing a collection map:

```
UPDATE cycling.upcoming_calendar
SET description =
{'Criterium du Dauphine' : 'Easy race',
 'Tour du Suisse' : 'Hard uphill race'}
WHERE year = 2015 AND month = 6;
```

The easiest way to add a new entry to the map is to use the + operator as described above:

```
UPDATE cycling.upcoming_calendar
SET description = description + { 'Tour de France' : 'Very competitive'}
WHERE year = 2015 AND month = 6;
```

You may, however, try to add the new entry with a command that overwrites the first two and adds the new one:

```
UPDATE cycling.upcoming_calendar
SET description =
{'Criterium du Dauphine' : 'Easy race',
 'Tour du Suisse' : 'Hard uphill race',
 'Tour de France' : 'Very competitive'}
WHERE year = 2015 AND month = 6;
```

These two statements seem to do the same thing. But behind the scenes, Cassandra processes the second statement by deleting the entire collection and replacing it with a new collection containing three entries. This creates tombstones for the deleted entries, even though these entries are identical to the entries in the new map collection. If your code updates all map collections this way, it generates many tombstones, which may slow the system down.

The examples above use map collections, but the same caution applies to updating sets.

Conditionally updating columns

In Conditionally update columns using IF or IF EXISTS.

Add IF EXISTS to the command to ensure that the operation is not performed if the specified row exists:

```
UPDATE cycling.cyclist_id SET age = 28 WHERE lastname = 'WELTEN' and
firstname = 'Bram' IF EXISTS;
```

Without IF EXISTS, the command proceeds with no standard output. If IF EXISTS returns true (if a row with this primary key does exist), standard output displays a table like the following:

[**applied**]

True

If no such row exists, however, the condition returns FALSE and the command fails. In this case, standard output looks like:

[**applied**]

False

Use IF condition to apply tests to one or more column values in the selected row:

```
UPDATE cyclist_id SET id = 15a116fc-b833-4da6-ab9a-4a3775750239 where
lastname = 'WELTEN' and firstname = 'Bram' IF age = 18;
```

If all the conditions return TRUE, standard output is the same as if IF EXISTS returned true (see above). If any of the conditions fails, standard output displays False in the [**applied**] column and also displays information about the condition that failed:

[**applied**] | age

False | 18

Conditional updates are examples of *lightweight transactions*. They incur a non-negligible performance cost and should be used sparingly.

Performing conditional updates in a BATCH

The UPDATE command can create new rows. The new *updated* rows are immediately available for lightweight transactions (for example, conditional statements added to UPDATE commands) when both the UPDATE and the conditional statements are applied in the same BATCH.

For example, consider a simple table with four defined columns:

```
CREATE TABLE mytable (a int, b int, s int static, d text, PRIMARY KEY (a,
b))

BEGIN BATCH
    INSERT INTO mytable (a, b, d) values (7, 7, 'a')
    UPDATE mytable SET s = 7 WHERE a = 7 IF s = NULL;
```

```
APPLY BATCH
```

In the first batch above, the insert command creates a partition with primary key values (7,7) but does not set a value for the `s` column. Even though the `s` column was not defined for this row, the `IF s = NULL` conditional succeeds, so the batch succeeds. (In previous versions of Cassandra, the conditional would have failed, and that failure would have caused the entire batch to fail.)

The second batch demonstrates more complex handling of a lightweight transaction:

```
BEGIN BATCH
    INSERT INTO mytable (a, b, d) values (7, 7, 'a')
    UPDATE mytable SET s = 1 WHERE a = 1 IF s = NULL;
APPLY BATCH
```

In this case, the `IF` statement tests the value of a column in a partition that does not even exist before it is created by the `UPDATE`. Even so, Cassandra recognizes the implicit presence of the partition and its column `s`, and lets the conditional test succeed. This allows the batch to succeed.

USE

Identifies the keyspace 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 single quotes.

Synopsis

```
USE keyspace_name
```

Table 71. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
<i>Italics</i>	Variable value. Replace with a user-defined value.
[]	Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.
()	Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.

Table 71. Legend (continued)

Syntax conventions	Description
	Or. A vertical bar () separates alternative elements. Type any one of the elements. Do not type the vertical bar.
...	Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.
' <i>Literal string</i> '	Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{ <i>key</i> : <i>value</i> }	Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.
< <i>datatype1,datatype2></i>	Set, list, map, or tuple. Angle brackets (< >) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
<i>cql_statement</i> ;	End CQL statement. A semicolon (;) terminates all CQL statements.
[--]	Separate the command line options from the command arguments with two hyphens (--). This syntax is useful when arguments might be mistaken for command line options.
' < <i>schema</i> > ... </ <i>schema</i> > '	Search CQL only: Single quotation marks (') surround an entire XML schema declaration.
@ <i>xml_entity</i> =' <i>xml_entity_type</i> '	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

Example

```
USE PortfolioDemo;
```

```
USE "Excalibur";
```