CQL for the DataStax Distribution of Apache Cassandra™ 3.11 Latest DDAC patch: 5.1.17
Contents

Chapter 1. About CQL.....................................................................................................................9

Chapter 2. CQL quickstart..............................................................................................................10
  Introduction.................................................................................................................................10
  Running CQL commands...........................................................................................................10
  Data modeling............................................................................................................................10
  Creating a keyspace....................................................................................................................14
  Creating a table with a simple primary key...............................................................................14
  Creating a table with a compound primary key........................................................................16
  Creating a table with a composite partition key.........................................................................17
  Using a secondary index............................................................................................................20
  Updating data.............................................................................................................................20
  Deleting and truncating data.......................................................................................................21
  Dropping tables and keyspaces..................................................................................................22
  Increase your knowledge............................................................................................................23

Chapter 3. CQL quick reference....................................................................................................24

Chapter 4. CQL data modeling.....................................................................................................30
  Data modeling concepts.............................................................................................................30
  Data modeling analysis.............................................................................................................33
  Using materialized views..........................................................................................................34
    Understanding materialized views.........................................................................................34
    Known limitations..................................................................................................................36
    Creating a materialized view.................................................................................................37
    Altering a materialized view.................................................................................................39
    Dropping a materialized view..............................................................................................39
    Best practices.........................................................................................................................40
    FAQ........................................................................................................................................40

Chapter 5. Introduction to CQL commands..................................................................................41
  Accessing data using CQL.........................................................................................................41
  Starting standalone cqlsh..........................................................................................................41
  Starting cqlsh............................................................................................................................42
  Understanding the CQL command syntax................................................................................43
  Uppercase and lowercase.........................................................................................................43
Valid characters in names........................................................................................................44
Keyspace qualifier..................................................................................................................44
Escaping characters..............................................................................................................45
Valid literals..........................................................................................................................45
Exponential notation...........................................................................................................46
Code comments....................................................................................................................46
Keywords..............................................................................................................................46
Limitations............................................................................................................................51

**Chapter 6. Managing keyspaces**..................................................................................52

About keyspaces................................................................................................................52
Creating a keyspace............................................................................................................52
Modifying the replication factor.......................................................................................53
Dropping a keyspace...........................................................................................................54
About system keyspace tables............................................................................................54
  Keyspace, table, and column information......................................................................58
  Cluster information...........................................................................................................59
  Functions, aggregates, and user types.............................................................................60

**Chapter 7. Managing types, functions, and aggregates**...........................................61

Creating user-defined function (UDF)..............................................................................61
Creating User-Defined Aggregate Function (UDA).........................................................62
Modifying a user-defined type...........................................................................................62
Dropping a user-defined function (UDF)..........................................................................63

**Chapter 8. Managing tables**......................................................................................64

Creating a table..................................................................................................................64
  Defining a basic primary key............................................................................................65
  Defining a multi-column partition key............................................................................66
  Defining a partition key with clustering columns..........................................................67
  Using advanced data types for columns......................................................................68
Creating a counter table...................................................................................................74
  Using a counter...............................................................................................................75
  Counter type....................................................................................................................75
Creating a table with COMPACT STORAGE..................................................................76
Fixing a table schema collision........................................................................................76
Altering a table...................................................................................................................76
  Altering columns in a table............................................................................................77
CQL for the DataStax Distribution of Apache Cassandra™ 3.11 Latest DDAC patch:

Chapter 9. Querying data ................................................................. 84

- About SELECT statements ......................................................... 84
- Setting consistency levels ........................................................... 84
  - Data consistency and performance ........................................ 84
  - Testing performance impact .................................................. 85
- Restricting queries using WHERE clauses ..................................... 90
  - Partition keys ................................................................. 91
  - Clustering columns .......................................................... 92
- Retrieval using the IN keyword .................................................. 103
- Sorting and limiting results ..................................................... 104
- Returning collections ............................................................. 108
- Aggregating results ............................................................... 108
  - Applying user-defined aggregate (UDA) functions ..................... 109
- Using user-defined functions (UDF) ............................................ 110
- Returning the write timestamp .................................................. 110
- Formatting query results as JSON .............................................. 111
- Legacy tables ........................................................................... 111
  - Working with legacy applications ....................................... 111
  - Querying a legacy table ...................................................... 112
  - Using a CQL legacy table query .......................................... 112

Chapter 10. Inserting and updating data ............................................ 114

- Using INSERT and UPDATE to write values .................................. 114
  - UUID and timeuuid column ................................................... 114
  - Set column ......................................................................... 115
  - List fields ........................................................................... 116
  - Map fields ........................................................................... 117
  - Tuple column ....................................................................... 118
  - User-defined type (UDT) column ......................................... 118
- Date column ............................................................................ 120
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time column</td>
<td>121</td>
</tr>
<tr>
<td>Timestamp column</td>
<td>122</td>
</tr>
<tr>
<td>Duration column</td>
<td>123</td>
</tr>
<tr>
<td>Using lightweight transactions</td>
<td>125</td>
</tr>
<tr>
<td>Inserting JSON formatted values</td>
<td>126</td>
</tr>
<tr>
<td>Importing data from a CSV file</td>
<td>127</td>
</tr>
<tr>
<td>Deleting values</td>
<td>127</td>
</tr>
<tr>
<td>Expiring data with time-to-live</td>
<td>127</td>
</tr>
<tr>
<td>Expiring data with time-to-live (TTL) example</td>
<td>128</td>
</tr>
<tr>
<td>Setting time-to-live</td>
<td>128</td>
</tr>
<tr>
<td><strong>Chapter 11. Batching inserts and updates</strong></td>
<td>130</td>
</tr>
<tr>
<td>Batching inserts, updates, and deletes</td>
<td>130</td>
</tr>
<tr>
<td>Good use of BATCH statement</td>
<td>131</td>
</tr>
<tr>
<td>Misuse of BATCH statement</td>
<td>133</td>
</tr>
<tr>
<td><strong>Chapter 12. CQL reference</strong></td>
<td>135</td>
</tr>
<tr>
<td>Introduction</td>
<td>135</td>
</tr>
<tr>
<td>CQL data types</td>
<td>135</td>
</tr>
<tr>
<td>Date, time, and timestamp format</td>
<td>138</td>
</tr>
<tr>
<td>CQL native functions</td>
<td>141</td>
</tr>
<tr>
<td>CQL native aggregates</td>
<td>145</td>
</tr>
<tr>
<td><strong>cqish reference</strong></td>
<td>147</td>
</tr>
<tr>
<td>cqish (startup options)</td>
<td>147</td>
</tr>
<tr>
<td>Configuring a cqishrc file</td>
<td>151</td>
</tr>
<tr>
<td>CAPTURE</td>
<td>157</td>
</tr>
<tr>
<td>CLEAR</td>
<td>158</td>
</tr>
<tr>
<td>CONSISTENCY</td>
<td>159</td>
</tr>
<tr>
<td>COPY TO</td>
<td>162</td>
</tr>
<tr>
<td>COPY FROM</td>
<td>168</td>
</tr>
<tr>
<td>DESCRIBE AGGREGATE</td>
<td>174</td>
</tr>
<tr>
<td>DESCRIBE CLUSTER</td>
<td>175</td>
</tr>
<tr>
<td>DESCRIBE SCHEMA</td>
<td>176</td>
</tr>
<tr>
<td>DESCRIBE FUNCTION</td>
<td>177</td>
</tr>
<tr>
<td>DESCRIBE INDEX</td>
<td>179</td>
</tr>
<tr>
<td>DESCRIBE KEYSPACE</td>
<td>180</td>
</tr>
<tr>
<td>DESCRIBE MATERIALIZED VIEW</td>
<td>181</td>
</tr>
</tbody>
</table>
CQL for the DataStax Distribution of Apache Cassandra™ 3.11 Latest DDAC patch:
5.1.17

- DESCRIBE TABLE.................................................................................................................. 182
- DESCRIBE TYPE.................................................................................................................... 184
- EXECUTE AS.......................................................................................................................... 186
- EXPAND.................................................................................................................................. 187
- EXIT......................................................................................................................................... 189
- LOGIN..................................................................................................................................... 190
- PAGING..................................................................................................................................... 191
- SERIAL CONSISTENCY............................................................................................................... 192
- SHOW....................................................................................................................................... 194
- SOURCE.................................................................................................................................... 196
- TRACING..................................................................................................................................... 197

CQL commands.......................................................................................................................... 200
- ALTER KEYSSPACE.................................................................................................................. 200
- ALTER MATERIALIZED VIEW..................................................................................................... 202
- ALTER ROLE........................................................................................................................... 204
- ALTER TABLE............................................................................................................................ 205
- ALTER TYPE............................................................................................................................. 209
- ALTER USER............................................................................................................................. 211
- BATCH....................................................................................................................................... 212
- CREATE AGGREGATE................................................................................................................. 217
- CREATE INDEX........................................................................................................................... 221
- CREATE FUNCTION.................................................................................................................... 223
- CREATE KEYSSPACE.................................................................................................................. 225
- CREATE MATERIALIZED VIEW..................................................................................................... 228
- CREATE ROLE............................................................................................................................ 231
- CREATE TABLE............................................................................................................................ 234
- CREATE TRIGGER...................................................................................................................... 245
- CREATE TYPE............................................................................................................................ 246
- CREATE USER (Deprecated)........................................................................................................ 248
- DELETE....................................................................................................................................... 249
- DROP AGGREGATE..................................................................................................................... 253
- DROP FUNCTION........................................................................................................................ 253
- DROP INDEX............................................................................................................................... 254
- DROP KEYSSPACE..................................................................................................................... 255
- DROP MATERIALIZED VIEW....................................................................................................... 256
CQL for the DataStax Distribution of Apache Cassandra™ 3.11 Latest DDAC patch:

Chapter 13. Cycling keyspace

Cycling tables

birthday_list

comments

cyclist_alt_stats

cyclist_name

cyclist_races

events

race_times

rank_by_year_and_name
Chapter 1. About CQL

CQL for DataStax Distribution of Apache Cassandra™ (DDAC) provides information about the Cassandra Query Language (CQL). CQL is the primary language for communicating with the database.

DataStax offers subscription-based support (Luna) for open-source Cassandra. Learn more.

To ensure that you get the best experience in using this document, take a moment to look at the Tips for using DataStax documentation. This page provides information on using search, navigational aids, and providing feedback.

DDAC documentation and other information sources

<table>
<thead>
<tr>
<th>DataStax Distribution of Apache Cassandra documentation and other information sources</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DDAC</strong></td>
</tr>
<tr>
<td><strong>Other information sources</strong></td>
</tr>
<tr>
<td><strong>Landing pages</strong></td>
</tr>
</tbody>
</table>
| **DSE drivers** | C/C++ driver, C# driver, Java driver, Node.js driver, PHP driver, Python driver, and Ruby driver.  
The PHP and Ruby drivers are maintenance only. |
| **Planning and testing DSE deployments** | Includes hardware selection, estimating disk capacity, anti-patterns, and cluster testing. |
| **DSE Troubleshooting Guide** | Various troubleshooting topics including Linux settings, search, analytics, security, starting DDAC, and installing. |
| **Upgrade Guide** | Information on upgrading various versions of DataStax Distribution of Apache Cassandra™ (DDAC). |
| **Sources of support** | DataStax Support and DataStax Academy forums. |
| **DataStax Academy** | DataStax Academy for courses on CQL, database administration, data modeling, and others. |
Chapter 2. CQL quickstart

Introduction
Cassandra Query Language (CQL) enables you to communicate with DataStax Distribution of Apache Cassandra™ (DDAC) and create database tables, manipulate data, and perform other activities. CQL has a similar syntax to SQL.

In this document, you will perform these tasks:

• Learn to retrieve data using the CQL `SELECT` statement.
• Begin learning about Cassandra data modeling and how it differs from relational database modeling. A data model is used to define the database tables, and the tables are created using CQL.
• Use CQL statements to create and drop database keyspaces and tables.
• Insert, retrieve, update, and delete data.

You can use the DataStax Bulk Loader application to load data from CSV or JSON files into the database.

Running CQL commands

Prerequisites: Ensure DataStax Distribution of Apache Cassandra™ (DDAC) is installed and running. See Installing DDAC.

Use CQL shell to run CQL commands.

1. Start CQL shell:

   ![CQL shell output]

   You will see the version numbers for your installation.

Data modeling
This section describes some basic data modeling concepts.

• Data modeling identifies the tables that store the data. You create your data model and tables using CQL.
• Data modeling in DataStax Distribution of Apache Cassandra™ (DDAC) is based on queries. Define your application queries first and then use the queries to design the database tables.
• Cassandra and relational databases model data relationships differently:

  # In a relational database, you define the relationships between tables using foreign keys and then write queries to refer from one table to another.
In Cassandra, there are no foreign keys. Instead, relationships between tables are typically managed by
the application, not the database server.

- Cassandra and relational databases use different data normalization:
  - Relational databases use a normalized data model, which removes as much data duplication as possible.
  - Data in Cassandra is often retrieved using one query for each table. Data can be repeated among tables,
    a process known as denormalization. This enables high performance.
    For example, in a relational database, a user entity and an address entity each have a table, and a
    foreign key defines the relationship between a user and their address. In Cassandra, the user and their
    address are stored in one table.

- Cassandra uses a partitioned row store with tunable data consistency:
  - Consistency refers to how up-to-date and synchronized all replicas of a row are.
  - Tunable consistency means that you specify the required consistency for the read and write operations.
  - Some data requires high consistency. For example, an update to a user password requires high
    consistency, but an update to a user profile picture might require low consistency.
  - High consistency operations have a greater performance penalty.

- Rows are organized into tables:
  - A row is uniquely identified by a primary key. In Cassandra, a primary key has a partition key and one or
    more optional clustering columns.
  - A partition key can contain one or more columns. The partition key determines which node stores the
    data.
  - Clustering columns set the row sorting order in the partition. The order can be ascending or descending.

This quickstart and the CQL documentation use a data model that stores details about cyclists and cycling races:
As mentioned earlier, queries are used to design the database tables. For example, you can run a query that returns a list of cyclists, including each cyclist's unique id identifier, firstname, and lastname. The id column is the table's partition key (K). This diagram shows the logical model for the cyclist details:
Figure 2: Query 1: Find a cyclist's name with a specified id

```
cyclist_name
```

- id
- lastName
- firstName

partition key

Figure 3: Query 2: Find cyclists given a specified category

```
cyclist_category
```

- category
- id
- points
- lastName

partition key
clustering column
Creating a keyspace

A keyspace is similar to a relational database schema. A keyspace is the top-level container for database objects. In Cassandra, data is replicated across nodes in a cluster to ensure reliability and fault tolerance. Each keyspace has a replication strategy and a replication factor:

- **Simple strategy replication** (SimpleStrategy): Applies the same replication setting across the cluster. Only use in test environments.
- **Network topography replication** (NetworkTopologyStrategy): Applies the replication setting for each datacenter. Use in production environments.
- Replication factor: The number of nodes that contain copies of the data. In general, set to this to at least 3. Lower numbers are acceptable for simple test environments.

For details, see Data replication.

The examples in this document feature a keyspace that stores cycling race information. For simplicity, the example keyspace uses SimpleStrategy replication with a replication factor of 1. This means that the replicated data is identical across all datacenters in the cluster and there are no replica nodes.

1. Create a keyspace to store the cycling information:

   ```
   CREATE KEYSPACE IF NOT EXISTS cycling
   WITH replication = {
       'class' : 'SimpleStrategy',
       'replication_factor' : 1
   };
   ```

2. Use the keyspace as the default in future CQL commands:

   ```
   USE cycling;
   ```

Creating a table with a simple primary key

Data is stored in tables composed of rows and columns. A simple primary key consists of one column, which acts as the partition key. The partition key determines which node stores the data. In this section, you will learn how to use these CQL statements:

- **CREATE TABLE** to create a table.
- **INSERT** to add rows to a table.
- **SELECT** to retrieve rows from a table.

The examples in this section use a table named cyclist_name with these columns:

- An ID number for each cyclist, which are stored in a universally unique identifier UUID column named id. The id column is the primary key for the table.
- The cyclist's first and last names, which are stored in text columns named first_name and last_name.

The following examples show the CQL statements:

1. This CREATE TABLE statement creates the cyclist_name table in the cycling keyspace:

   ```
   CREATE TABLE cycling.cyclist_name (id UUID PRIMARY KEY,
   last_name text,
   first_name text
   );
   ```
2. These INSERT statements add rows to the table. The column names and corresponding values are specified in the statements.

```cql
INSERT INTO cycling.cyclist_name (id, last_name, first_name) VALUES (e7cd5752-bc0d-4157-a80f-7523add8dbcd, 'VAN DER BREGGEN', 'Anna');
```

```cql
INSERT INTO cycling.cyclist_name (id, last_name, first_name) VALUES (e7ae5cf3-d358-4d99-b900-85902fda9bb0, 'FRAME', 'Alex');
```

```cql
INSERT INTO cycling.cyclist_name (id, last_name, first_name) VALUES (220844bf-4860-49d6-9a4b-6b5d3a79cbfb, 'TIRALONGO', 'Paolo');
```

```cql
INSERT INTO cycling.cyclist_name (id, last_name, first_name) VALUES (6ab09bec-e68e-48d9-a5f8-9e6fb4c9b47, 'KRUIKSWIJK', 'Steven');
```

```cql
INSERT INTO cycling.cyclist_name (id, last_name, first_name) VALUES (fb372533-eb95-4bb4-8685-6ef61e994caa, 'MATTHEWS', 'Michael');
```

3. This SELECT statement retrieves the rows from the table. The column names and values are returned.

```cql
SELECT * FROM cycling.cyclist_name;
```

<table>
<thead>
<tr>
<th>id</th>
<th>first_name</th>
<th>last_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>e7ae5cf3-d358-4d99-b900-85902fda9bb0</td>
<td>Alex</td>
<td>FRAME</td>
</tr>
<tr>
<td>fb372533-eb95-4bb4-8685-6ef61e994caa</td>
<td>Michael</td>
<td>MATTHEWS</td>
</tr>
<tr>
<td>220844bf-4860-49d6-9a4b-6b5d3a79cbfb</td>
<td>Paolo</td>
<td>TIRALONGO</td>
</tr>
<tr>
<td>6ab09bec-e68e-48d9-a5f8-9e6fb4c9b47</td>
<td>Steven</td>
<td>KRUIKSWIJK</td>
</tr>
<tr>
<td>e7cd5752-bc0d-4157-a80f-7523add8dbcd</td>
<td>Anna</td>
<td>VAN DER BREGGEN</td>
</tr>
</tbody>
</table>

(5 rows)

4. This SELECT statement uses a WHERE clause to limit the rows retrieved. The WHERE clause returns the row with the specified id value.

```cql
SELECT * FROM cycling.cyclist_name WHERE id = fb372533-eb95-4bb4-8685-6ef61e994caa;
```

<table>
<thead>
<tr>
<th>id</th>
<th>first_name</th>
<th>last_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>fb372533-eb95-4bb4-8685-6ef61e994caa</td>
<td>Michael</td>
<td>MATTHEWS</td>
</tr>
</tbody>
</table>
Creating a table with a compound primary key

This section shows how to create a table with a compound primary key and populate it with data.

- A compound primary key has a partition key and one or more clustering columns.
- The partition key determines the node that stores the data.
- The clustering columns specify the order of the rows in the partition.
- When rows for a partition key are stored based on the clustering column values, row retrieval is fast. If the data is distributed throughout the nodes, data retrieval from a large partition can be slow when the entire partition must be read.

The examples in this section use a table named `cyclist_category`. The table has these columns:

- Race category, which is the partition key.
- Race points, which is the clustering column.
- Cyclist id.
- Cyclist lastname.

You can query the `cyclist_category` table to retrieve a list of cyclists and their race points in a category.

1. This `CREATE TABLE` statement uses the `WITH CLUSTERING ORDER BY` clause to sort the race points in descending order for each category:

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

The combination of the category and points uniquely identifies a row in the `cyclist_category` table. You can add more than one row with the same category to the table as long as the rows contain different points values.

2. These `INSERT` statements add rows to the table:

```plaintext
INSERT INTO cycling.cyclist_category (
    category, points, id, lastname
) VALUES (
    'GC', 1269, 220844bf-4860-49d6-9a4b-6b5d3a79cbfb, 'TIRALONGO'
);
INSERT INTO cycling.cyclist_category (
    category, points, id, lastname
) VALUES (
    'One-day-races', 367, 220844bf-4860-49d6-9a4b-6b5d3a79cbfb, 'TIRALONGO'
);
INSERT INTO cycling.cyclist_category (
    category, points, id, lastname
) VALUES (
    'GC', 1269, 220844bf-4860-49d6-9a4b-6b5d3a79cbfb, 'TIRALONGO'
);
```


```cql
) VALUES ('Time-trial', 182, 220844bf-4860-49d6-9a4b-6b5d3a79cbfb, 'TIRALONGO')
;

INSERT INTO cycling.cyclist_category (category, points, id, lastname)
) VALUES ('Sprint', 0, 220844bf-4860-49d6-9a4b-6b5d3a79cbfb, 'TIRALONGO')
;

INSERT INTO cycling.cyclist_category (category, points, id, lastname)
) VALUES ('GC', 1324, 6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47, 'KRUIJSWIJK')
;

INSERT INTO cycling.cyclist_category (category, points, id, lastname)
) VALUES ('One-day-races', 198, 6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47, 'KRUIJSWIJK')
;

INSERT INTO cycling.cyclist_category (category, points, id, lastname)
) VALUES ('Sprint', 39, 6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47, 'KRUIJSWIJK')
;

3. This `SELECT` statement retrieves rows from the table. Within each category the rows are sorted in descending points order, which means that the highest points are listed first.

```cql
SELECT *
FROM cycling.cyclist_category;
```

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

(8 rows)

Creating a table with a composite partition key

This section shows how to create a table with a composite primary key and populate it with data.

- A composite partition key contains multiple columns.
- Using multiple columns in a partition key breaks the data into smaller groups than if using a single partition key column. Smaller row groups reduce cluster hotspots, where one partition is repeatedly written to.
The examples in this section use a table named `rank_by_year_and_name`. The table stores the ranking and name of cyclists who competed in races.

1. This CREATE TABLE statement creates the `rank_by_year_and_name` table. The `race_year` and `race_name` columns are the composite partition key. The primary key also has a column named `rank`, which is a clustering column. Notice the double parentheses around the first two columns in the PRIMARY KEY clause, which identifies the composite partition key.

   ```
   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)
   )
   ```

2. These INSERT statements add rows to the table:

   ```
   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 ('2014', 'Tour of Japan - Stage 4 - Minami > Shinshu', 'Benjamin PRADES', 3);
   INSERT INTO cycling.rank_by_year_and_name (race_year, race_name, cyclist_name, rank) VALUES ('2014', 'Tour of Japan - Stage 4 - Minami > Shinshu', 'Daniel MARTIN', 1);
   INSERT INTO cycling.rank_by_year_and_name (race_year, race_name, cyclist_name, rank) VALUES ('2014', 'Tour of Japan - Stage 4 - Minami > Shinshu', 'Johan Esteban CHAVES', 2);
   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);
   ```
3. This `SELECT` statement returns the sorted rank of the cyclists for each race year and race name:

```sql
SELECT *
FROM cycling.rank_by_year_and_name;
```

<table>
<thead>
<tr>
<th>race_year</th>
<th>race_name</th>
<th>rank</th>
<th>cyclist_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2014</td>
<td>4th Tour of Beijing</td>
<td>1</td>
<td>Phillippe GILBERT</td>
</tr>
<tr>
<td>2014</td>
<td>4th Tour of Beijing</td>
<td>2</td>
<td>Daniel MARTIN</td>
</tr>
<tr>
<td>2014</td>
<td>4th Tour of Beijing</td>
<td>3</td>
<td>Johan Esteban CHAVES</td>
</tr>
<tr>
<td>2014</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>1</td>
<td>Daniel MARTIN</td>
</tr>
<tr>
<td>2014</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>2</td>
<td>Johan Esteban CHAVES</td>
</tr>
<tr>
<td>2014</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>3</td>
<td>Benjamin PRADES</td>
</tr>
<tr>
<td>2015</td>
<td>Giro d'Italia - Stage 11 - Forli &gt; Imola</td>
<td>1</td>
<td>Ilnur ZAKARIN</td>
</tr>
<tr>
<td>2015</td>
<td>Giro d'Italia - Stage 11 - Forli &gt; Imola</td>
<td>2</td>
<td>Carlos BETANCUR</td>
</tr>
<tr>
<td>2015</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>1</td>
<td>Benjamin PRADES</td>
</tr>
<tr>
<td>2015</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>2</td>
<td>Adam PHELAN</td>
</tr>
<tr>
<td>2015</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>3</td>
<td>Thomas LEBAS</td>
</tr>
</tbody>
</table>
Using a secondary index

Certain queries return an error. For example, you might want to see the cyclist rankings for a particular year. The `rank_by_year_and_name` table has a composite partition key with two columns, `race_year` and `race_name`; this query fails because only the `race_year` column is specified in the `WHERE` clause:

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

InvalidRequest: Error from server: code=2200 [Invalid query] message=
"Cannot execute this query as it might involve data filtering and thus may have unpredictable performance. If you want to execute this query despite the performance unpredictability, use ALLOW FILTERING"

To run the previous query successfully, you can create a secondary index on the `race_year` column.

1. This `CREATE INDEX` statement adds the index and then the query works:

```sql
CREATE INDEX ryear ON
cycling.rank_by_year_and_name (race_year);
```

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

<table>
<thead>
<tr>
<th>race_year</th>
<th>race_name</th>
<th>rank</th>
<th>cyclist_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015</td>
<td>Giro d'Italia - Stage 11 - Forli &gt; Imola</td>
<td>1</td>
<td>Ilnur ZAKARIN</td>
</tr>
<tr>
<td>2015</td>
<td>Giro d'Italia - Stage 11 - Forli &gt; Imola</td>
<td>2</td>
<td>Carlos BETANCUR</td>
</tr>
<tr>
<td>2015</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>1</td>
<td>Benjamin PRADES</td>
</tr>
<tr>
<td>2015</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>2</td>
<td>Adam PHELAN</td>
</tr>
<tr>
<td>2015</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>3</td>
<td>Thomas LEBAS</td>
</tr>
</tbody>
</table>

Points to consider before using secondary indexes

Secondary indexes can impact performance when multiple nodes are accessed because the index table is stored on each node in a cluster. Before creating a secondary index, see when not to create an index. Guidelines:

- In general, only add a secondary index a column with low cardinality, which means that the column contains a few unique values.
- If you have a high cardinality column, consider creating a separate table to store the data in the column on which you want to create a secondary index.

Updating data

Use the `UPDATE` statement to write one or more column values to a row.
Use the `SET` clause of the `UPDATE` statement to assign new column values. `UPDATE` cannot update the values of primary key columns; instead, delete the row and insert a row with the new primary key column values.

The `WHERE` clause identifies the row or rows to be updated. To specify a row, the `WHERE` clause must specify a value for each column of the row’s primary key.

To specify more than one row to update, use the `IN` clause with a list of possible values. You can only do that for the last column of the primary key.

The examples in this section use the `rank_by_year_and_name` table:

```cql
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)
);
```

1. This example updates the cyclist name. Notice that a value for each column in the primary key is provided in the `WHERE` clause.

   ```cql
   UPDATE cycling.rank_by_year_and_name
   SET cyclist_name = 'John SMITH'
   WHERE race_year = 2015
       AND race_name = 'Tour of Japan - Stage 4 - Minami > Shinshu'
       AND rank = 2;
   ```

2. To specify more than one row, use `IN` with a list of possible values. You can only use `IN` with a list of values for the last column of the primary key.

   ```cql
   UPDATE cycling.rank_by_year_and_name
   SET cyclist_name = 'Jane DOE'
   WHERE race_year = 2015
       AND race_name = 'Tour of Japan - Stage 4 - Minami > Shinshu'
       AND rank IN (2, 3, 4);
   ```

3. To perform the update only if the specified row exists, use `IF EXISTS`:

   ```cql
   UPDATE cycling.rank_by_year_and_name
   SET cyclist_name = 'John SMITH'
   WHERE race_year = 2015
       AND race_name = 'Tour of Japan - Stage 4 - Minami > Shinshu'
       AND rank = 3
   IF EXISTS;
   ```

### Deleting and truncating data

Use the `DELETE` statement to remove data from one or more selected columns (column data is replaced with a null value), or to remove the entire row when no column is specified.

The data is not removed from disk immediately. The data is marked with a tombstone and removed during a `nodetool repair` operation.

Use the `TRUNCATE` statement to remove all data from the specified table immediately.

The examples in this section use the `rank_by_year_and_name` table:

```cql
CREATE TABLE cycling.rank_by_year_and_name (
    race_year int,
    race_name text,
    cyclist_name text,
);```
CQL quickstart

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

1. This DELETE statement deletes the cyclist name:

```
DELETE cyclist_name
FROM cycling.rank_by_year_and_name
WHERE race_year = 2015
  AND race_name = 'Tour of Japan - Stage 4 - Minami > Shinshu'
  AND rank = 3;
```

2. To delete a row, omit the column names after the DELETE statement. This example deletes one row:

```
DELETE
FROM cycling.rank_by_year_and_name
WHERE race_year = 2015
  AND race_name = 'Tour of Japan - Stage 4 - Minami > Shinshu'
  AND rank = 3;
```

3. To delete multiple rows, use a WHERE clause that specifies multiple rows:

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

4. To perform the delete only if the specified row exists, use IF EXISTS:

```
DELETE
FROM cycling.rank_by_year_and_name
WHERE race_year = 2014
  AND race_name = '4th Tour of Beijing'
  AND rank = 3
IF EXISTS;
```

5. To remove all data from the specified table, use TRUNCATE:

```
TRUNCATE cycling.rank_by_year_and_name;
```

Dropping tables and keyspaces

To remove a table and the table data, use the DROP TABLE statement.
To remove a keyspace and all of the objects in it, use the DROP KEYSPACE statement.
To perform the removal only if the object exists, add the IF EXISTS clause.
1. **This DROP TABLE example drops the** `rank_by_year_and_name` **table:**

   ```sql
   DROP TABLE IF EXISTS cycling.rank_by_year_and_name;
   ```

2. **This DROP KEYSPACE example drops the** `cycling` **keyspace:**

   ```sql
   DROP KEYSPACE cycling IF EXISTS;
   ```

**Increase your knowledge**

See these documents for more information:

- [Introduction to CQL commands](#)
- [CQL reference](#)
- [CQL data modeling](#)
- [CQL quick reference](#)
- [DataStax Bulk Loader](#) (Loads data from CSV or JSON files into the database.)

DataStax Academy also has online classes about CQL and data modeling. See [academy.datastax.com](http://academy.datastax.com).
Chapter 3. CQL quick reference

Provides a consolidated syntax list of Cassandra Query Language (CQL) commands for quick reference.

See this quick reference guide for other CQL versions: 5.1 | 6.0 | 6.7.

Download a printable CQL reference with the ten most frequently use CQL commands and a list of the CQL data types.

**ALTER KEYSPACE**

```cql
ALTER KEYSPACE keyspace_name
    WITH REPLICATION = {replication_map}
    [AND DURABLE_WRITES = true|false];
```

Learn more.

**ALTER MATERIALIZED VIEW**

```cql
ALTER MATTERNALIZED VIEW [keyspace_name.]view_name
    WITH table_options [ AND table_options ... ];
```

Learn more.

**ALTER ROLE**

```cql
ALTER ROLE role_name
    [ WITH [ PASSWORD = 'password'
        | LOGIN = (true | false)
        | SUPERUSER = (true | false)
        | OPTIONS = option_map ]
```

Learn more.

**ALTER TABLE**

```cql
ALTER TABLE [keyspace_name.]table_name
    [ ADD (column_definition | column_definition_list) ]
    [ DROP (column | column_list | COMPACT STORAGE) ]
    [ RENAME column_name TO column_name ]
    [ WITH table_properties [ , ... ] ];
```

Learn more.

**ALTER TYPE**

```cql
ALTER TYPE field_name
    [ ADD field_name cql_datatype [ , ... ]
    | RENAME field_name TO new_field_name [ AND field_name TO new_field_name ... ] ];
```

Learn more.

**ALTER USER**

```cql
ALTER USER user_name
    WITH PASSWORD 'password'
```

Learn more.
Learn more.

BATCH

```
BEGIN [ ( UNLOGGED | COUNTER ) ] BATCH
  [ USING TIMESTAMP [ epoch_microseconds ] ]
  dml_statement [ USING TIMESTAMP [ epoch_microseconds ] ] ;
  [ dml_statement [ USING TIMESTAMP [ epoch_microseconds ] ] [ ; ... ] ] ;
APPLY BATCH ;
```

Learn more.

CREATE AGGREGATE

```
CREATE [ OR REPLACE ] AGGREGATE [ IF NOT EXISTS ] [keyspace_name.]aggregate_name (cql_type)
  SFUNC udf_name
  STYPE cql_type
  FINALFUNC udf_name
  INITCOND init_value
  [ DETERMINISTIC ] ;
```

Learn more.

CREATE FUNCTION

```
CREATE [ OR REPLACE ] FUNCTION [ IF NOT EXISTS ] [keyspace_name.]function_name (argument_list [ , ... ])
  ( CALLED | RETURNS NULL ) ON NULL INPUT RETURNS cql_data_type
  [ DETERMINISTIC ]
  [ MONOTONIC [ ON argument_name ] ]
  LANGUAGE language_name AS 'code_block' ;
```

Learn more.

CREATE INDEX

```
CREATE INDEX [ IF NOT EXISTS ] index_name
  ON [keyspace_name.]table_name
  ( [ ( KEYS | FULL ) ] column_name)
  (ENTRIES column_name);
```

Learn more.

CREATE KEYSpace

```
CREATE KEYSpace [ IF NOT EXISTS ] keyspace_name
  WITH REPLICATION = { replication_map }
  [AND DURABLE_WRITES = true | false ] ;
```

Learn more.

CREATE MATERIALIZED VIEW

```
CREATE MATERIALIZED VIEW [ IF NOT EXISTS ] [keyspace_name.]view_name
  AS SELECT [ (column_list) ]
  FROM [keyspace_name.]table_name
  WHERE column_name IS NOT NULL [ AND column_name IS NOT NULL ... ]
  [ AND relation [ AND ... ] ]
```
PRIMARY KEY (column_list)
[ WITH [ table_properties ]
[ [ AND ] CLUSTERING ORDER BY (cluster_column_name order_option) ] ] } ;

CREATE ROLE
CREATE ROLE [ IF NOT EXISTS ] role_name
[ WITH [ SUPERUSER = ( true | false ) ] ]
[ [ AND ] LOGIN = ( true | false ) ]
[ [ AND ] PASSWORD = 'role_password' ]
[ [ AND ] OPTIONS = option_map ] ) ;

CREATE TABLE
CREATE TABLE [ IF NOT EXISTS ] [keyspace_name.]table_name
( column_definition [ , . . . ] | PRIMARY KEY (column_list) )
[ WITH [ table_options ]
[ [ AND ] CLUSTERING ORDER BY [ clustering_column_name order ] ]
[ [ AND ] ID = 'table_hash_tag' ]
[ [ AND ] COMPACT STORAGE ] ;

CREATE TRIGGER
CREATE TRIGGER trigger_name
ON [keyspace_name.]table_name
USING 'org.apache.cassandra.triggers.AuditTrigger' ;

CREATE TYPE
CREATE TYPE [ IF NOT EXISTS ] [keyspace_name.]type_name
( field_name cql_datatype [ , field_name cql_datatype . . . ] ) ;

DELETE
DELETE [ column_name [ term ] [ , . . . ] ]
FROM [keyspace_name.]table_name
[ USING TIMESTAMP timestamp_value ]
WHERE PK_column_conditions
[ ( IF EXISTS | IF static_column_conditions ) ] ;

DROP AGGREGATE
DROP AGGREGATE [ IF EXISTS ] [keyspace_name.]aggregate_name [ (argument_name [ , . . . ] ) ] ;
### DROP FUNCTION

```cql
DROP FUNCTION [ IF EXISTS ] [keyspace_name.]function_name [ (argument_name [ , ... ]) ] ;
```

Learn more.

### DROP INDEX

```cql
DROP INDEX [ IF EXISTS ] [keyspace.]index_name ;
```

Learn more.

### DROP KEYPSPACE

```cql
DROP KEYSPACE [ IF EXISTS ] keyspace_name ;
```

Learn more.

### DROP MATERIALIZED VIEW

```cql
DROP MATERIALIZED VIEW [ IF EXISTS ] [keyspace_name.]view_name ;
```

Learn more.

### DROP ROLE

```cql
DROP ROLE [ IF EXISTS ] role_name ;
```

Learn more.

### DROP TABLE

```cql
DROP TABLE [ IF EXISTS ] [keyspace_name.]table_name ;
```

Learn more.

### DROP TRIGGER

```cql
DROP TRIGGER [ IF EXISTS ] trigger_name
ON [keyspace_name.]table_name ;
```

Learn more.

### DROP TYPE

```cql
DROP TYPE [ IF EXISTS ] [keyspace_name]type_name ;
```

Learn more.

### GRANT

```cql
GRANT permission
[ ON object ]
TO role_name ;
```

Learn more.
**CQL quick reference**

**INSERT**

```
INSERT INTO [keyspace_name.]table_name
  [ column_list VALUES column_values ]
  [ IF NOT EXISTS ]
  [ USING [ TTL seconds ] [ [ AND ] TIMESTAMP epoch_in_microseconds ] ]
;
```

Learn more.

**LIST PERMISSIONS**

```
LIST permission
  [ ON resource_name ]
  [ OF role_name ]
  [ NORECURSIVE ]
;
```

Learn more.

**LIST ROLES**

```
LIST ROLES
  [ OF role_name ]
  [ NORECURSIVE ]
;
```

Learn more.

**RESTRICT ROWS**

```
RESTRICT ROWS
  ON [keyspace_name.]table_name
  USING pk_column_name
;
```

Learn more.

**REVOKE**

```
REVOKE privilege
  ON resource_name
  FROM role_name
;
```

Learn more.

**SELECT**

```
SELECT selectors
  FROM [keyspace_name.]table_name
  [ WHERE [ primary_key_conditions ] [ AND ] [ index_conditions ] ]
  [ GROUP BY column_name [ , ... ] ]
  [ ORDER BY pk_column_name [ , ... ] ( ASC | DESC ) ]
  [ ( LIMIT N | PER PARTITION LIMIT N ) ]
  [ ALLOW FILTERING ]
;
```

Learn more.
TRUNCATE

TRUNCATE [ TABLE ] [keyspace_name.]table_name;

Learn more.

UNRESTRICT

UNRESTRICT ROWS ON [keyspace_name.]table_name;

Learn more.

UPDATE

UPDATE [keyspace_name.]table_name
[ USING TTL time_value ]
[ AND ] USING TIMESTAMP timestamp_value ]
SET assignment [ , assignment , ... ]
WHERE row_specification
[ ( IF EXISTS | IF NOT EXISTS | IF condition [ AND condition ... ] ) ] ;

Learn more.

USE

USE keyspace_name;

Learn more.
Chapter 4. CQL data modeling

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

Data modeling concepts

Data modeling is a process that involves identifying the entities (items to be stored) and the relationships between entities. For an introduction, take the DataStax Academy course on data modeling. See academy.datastax.com.

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 DataStax Distribution of Apache Cassandra™ (DDAC) 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 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 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.
A related query (Q2) searches for all cyclists by a particular race category. 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.
CQL data modeling

Figure 6: Query 2: Find cyclists given a specified category

<table>
<thead>
<tr>
<th>cyclist_category</th>
</tr>
</thead>
<tbody>
<tr>
<td>category</td>
</tr>
<tr>
<td>id</td>
</tr>
<tr>
<td>points</td>
</tr>
<tr>
<td>lastName</td>
</tr>
</tbody>
</table>

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. Relational databases normalize data, removing as much duplication as possible. Data in Cassandra is often arranged as one query per table, and data is repeated amongst many tables, a process known as denormalization. The relationship of the entities is important, because the order in which data is stored 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 modeling 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, 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 DataStax Distribution of Apache Cassandra™ and later, 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 DataStax Distribution of Apache Cassandra™ (DDAC) 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 Blog.

• Understanding the guarantees, limitations, and tradeoffs of materialized views
• Materialized view performance in Cassandra 3.x

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 DDAC 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.

Learn more about materialized views [Materialized views handle automated server-side denormalization, ensuring eventual consistency between the base and view data.]

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

- 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 the database to access all nodes in a cluster, causing high read latency.

Restrictions for materialized views:

- Include all of the source table primary keys in the materialized view’s primary key.
- Only one 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.

Example: Materialized view example

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

```cql
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 cyclists.

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

```cql
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:
CQL data modeling

• 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 data is copied.
• 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.

In this materialized view, `age` is used as the primary key and `cid` is a clustering column. In DataStax Distribution of Apache Cassandra™ (DDAC), clustering columns have a maximum size of 64 KB.

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;
```

<table>
<thead>
<tr>
<th>age</th>
<th>name</th>
<th>birthday</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>Adrian COSTA</td>
<td>1997-08-19</td>
</tr>
<tr>
<td>18</td>
<td>Bram WELTEN</td>
<td>1997-03-29</td>
</tr>
<tr>
<td>18</td>
<td>Pascal EENKHOORN</td>
<td>1997-02-08</td>
</tr>
</tbody>
</table>

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';
```

<table>
<thead>
<tr>
<th>age</th>
<th>name</th>
<th>birthday</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>Bram WELTEN</td>
<td>1997-03-29</td>
</tr>
<tr>
<td>28</td>
<td>Steven KRIUKSWIJK</td>
<td>1987-06-07</td>
</tr>
<tr>
<td>18</td>
<td>Pascal EENKHOORN</td>
<td>1997-02-08</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>age</th>
<th>name</th>
<th>birthday</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>Cristian EGIDIO</td>
<td>1987-08-04</td>
</tr>
<tr>
<td>27</td>
<td>Johannes HEIDER</td>
<td>1987-09-04</td>
</tr>
</tbody>
</table>
In DDAC, a materialized view can be created using a filtering statement that includes a restriction on a non-primary key column.

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

This materialized view will only store information for cyclists from the Netherlands because of the following addition to the `WHERE` clause.

```
AND country = 'Netherlands'
```

Now, a query can be submitted to find those cyclists from the Netherlands with a particular birthday:

```
SELECT age, name, birthday FROM cyclist_by_birthday WHERE birthday = '1997-02-08';
```

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

Cassandra can only write data directly to source tables, not to materialized views. The database updates a materialized view asynchronously after inserting data into the source table, so the update of materialized view is delayed. A read-repair operation to a materialized view is performed only after updating the source table.

```
CREATE MATERIALIZED VIEW [Optimizes read requests and eliminates the need for multiple write requests by duplicating data from a base table.]
ALTER MATERIALIZED VIEW [Changes the table properties of a materialized view.]
DROP MATERIALIZED VIEW [Removes the named materialized view.]
```

### Altering a materialized view

Materialized views are created with the default table properties. Use the `ALTER MATERIALIZED VIEW` command to 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.

- Alter a materialized view to change the caching properties.

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

```
CREATE MATERIALIZED VIEW [Optimizes read requests and eliminates the need for multiple write requests by duplicating data from a base table.]
ALTER MATERIALIZED VIEW [Changes the table properties of a materialized view.]
DROP MATERIALIZED VIEW [Removes the named materialized view.]
```

### Dropping a materialized view

Use the `DROP` command to drop a materialized view.
• Drop the `cycling.cyclist_by_age` materialized view in DataStax Distribution of Apache Cassandra™.

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

**CREATE MATERIALIZED VIEW** [Optimizes read requests and eliminates the need for multiple write requests by duplicating data from a base table.]

**ALTER MATERIALIZED VIEW** [Changes the table properties of a materialized view.]

**DROP MATERIALIZED VIEW** [Removes the named 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.
  
  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.

**What is the most recommended DDAC version for using materialized views?**

DataStax recommends using the most recent version of DataStax Distribution of Apache Cassandra™ (DDAC) to get the most stable version of materialized views. Several bugs fixes and performance improvements were added for materialized views.
Chapter 5. Introduction to CQL commands

CQL is the primary language for managing database resources, such as keyspaces, tables, functions, aggregates, user defined types, roles, access permissions, insert, update, and query tables commands.

For production, DataStax supplies a number of drivers so that CQL statements and search commands can be passed from client to cluster and back.

Schema changes, which involve CREATE, DROP, and ALTER statements, are not safe for concurrent use. Performing more than one change in flight has a high probability of causing irreconcilable conflicts, with nodes in the cluster using different schema “versions” indefinitely.

Accessing data using CQL

Common ways to access data using CQL are:

- **CQL shell (cqlsh)**: a Python-based command line shell to access the database and issue CQL (Cassandra Query Language) commands.

- **DataStax drivers**: the primary resource for application developers to connect to database clusters.

- **DataStax Studio**: an interactive developer’s tool with self-documenting notebooks.

Starting the standalone cqlsh tool

Launch the standalone cqlsh tool to remotely access a DataStax database.

Execute the cqlsh python script to start the CQL shell. The CQL shell is a python-based command line client for executing CQL commands interactively. The CQL shell supports tab completion.

**Prerequisites:** Install the standalone CQLSH tool that is compatible with your DataStax database.

1. Change to the directory where the tarball was extracted.
2. On the command line, run the cqlsh script.
Introduction to CQL commands

The cqlsh shell connects to 127.0.0.1 9042 by default. To change the defaults, set the $CQLSH_HOST and $CQLSH_PORT environment variables. When a host and optional port number are given on the command line, they take precedence over any defaults.

$ ./bin/cqlsh

You can optionally specify the IP address and port to interface with a remote DataStax database.

$ ./bin/cqlsh host_name port

For example:

$ ./bin/cqlsh 10.100.176.166 9042

If you use security features, provide the username and password to authenticate with the database:

$ ./bin/cqlsh -u username -p password host_name port

For example:

$ ./bin/cqlsh -u janeappdev -p j4nesp&swd 10.100.176.166 9042

3. Print the help menu for cqlsh.

$ ./bin/cqlsh --help

For a complete list, see cqlsh (startup options).

Starting cqlsh

Execute the cqlsh python script to start the CQL shell. The CQL shell is a python-based command line client for executing CQL commands interactively. The CQL shell supports tab completion.

To connect to a security-enabled cluster, see Using cqlsh with authentication.

For a complete list, see cqlsh (startup options).

1. Navigate to the DataStax Distribution of Apache 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 100.0.0.75 9042

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. Use easy_install to install tab completion capabilities on Mac OSX:

   easy_install readline

Understanding the CQL command syntax

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

For example, these are valid CQL statements:

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

The example contains select and update statements. The select is on one line. A statement can be split across multiple lines, as shown in the update statement.

In cqlsh, type help to list all available topics. Type help name to find out more about the name command. For example, help CAPTURE or help ALTER_KEYSPACE.

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, the database stores the names in lowercase. You can force the case 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 1: What Works and What Doesn’t

<table>
<thead>
<tr>
<th>Queries that Work</th>
<th>Queries that Don’t Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT foo FROM . . .</td>
<td>SELECT &quot;Foo&quot; FROM . . .</td>
</tr>
<tr>
<td>SELECT Foo FROM . . .</td>
<td>SELECT &quot;BAR&quot; FROM . . .</td>
</tr>
<tr>
<td>SELECT FOO FROM . . .</td>
<td>SELECT bar FROM . . .</td>
</tr>
<tr>
<td>SELECT &quot;Bar&quot; FROM . . .</td>
<td>SELECT Bar FROM . . .</td>
</tr>
</tbody>
</table>
Introduction to CQL commands

<table>
<thead>
<tr>
<th>Queries that Work</th>
<th>Queries that Don’t Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT &quot;foo&quot; FROM . . .</td>
<td>SELECT &quot;bar&quot; FROM . . .</td>
</tr>
</tbody>
</table>

SELECT "foo" FROM . . . works because the database stores foo in lowercase.

When using legacy tables, case-sensitivity rules in earlier versions of CQL apply.

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

To escape characters, see Escaping characters.

Valid characters in names

Only alpha-numeric characters and underscores are allowed in keyspace and table names; all other names support any characters, such as COLUMN, FUNCTION, AGGREGATE, TYPE, etc.

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

Table 2: What Works and What Doesn’t

<table>
<thead>
<tr>
<th>Creations that Work</th>
<th>Creations that Don’t Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE TABLE foo ...</td>
<td>CREATE TABLE foo$% ...</td>
</tr>
<tr>
<td>CREATE TABLE foo_bar ...</td>
<td>CREATE TABLE foo][90 ...</td>
</tr>
<tr>
<td>CREATE TABLE foo (&quot;what&quot;&amp;* text, ...)</td>
<td>CREATE TABLE foo (what&amp;* text, ...)</td>
</tr>
<tr>
<td>ALTER TABLE foo5 ...</td>
<td>ALTER TABLE &quot;foo5$$&quot;...</td>
</tr>
<tr>
<td>CREATE FUNCTION &quot;foo5$$%&quot; ...</td>
<td>CREATE FUNCTION foo5$$%...</td>
</tr>
<tr>
<td>CREATE AGGREGATE &quot;foo5$$@#&quot; ...</td>
<td>CREATE AGGREGATE foo5$$</td>
</tr>
<tr>
<td>CREATE TYPE foo5 (&quot;bar&quot;#9 text, ...)</td>
<td>CREATE TYPE foo5 (bar#9 text ...</td>
</tr>
</tbody>
</table>

Keyspace qualifier

Sometimes issuing a USE statement to select a keyspace is inconvenient. Connection pooling requires managing multiple keyspaces. To simplify tracking multiple keyspaces, use the keyspace qualifier instead of the USE statement. You can specify the keyspace using the keyspace qualifier in these statements:

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

1. 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.

```cpp
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'})
```
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

A 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
Introduction to CQL commands

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

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

Exponential notation

DataStax Distribution of Apache Cassandra™ (DDAC) supports exponential notation. This example shows exponential notation in the output from a cqlsh command.

```sql
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;
```

<table>
<thead>
<tr>
<th>id</th>
<th>value_double</th>
<th>value_float</th>
</tr>
</thead>
<tbody>
<tr>
<td>test1</td>
<td>-2.6034e+38</td>
<td>-2.6034e+38</td>
</tr>
</tbody>
</table>

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

Keywords

This table lists keywords and whether 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.
Reserved keywords

To use reserved keywords as an identifier in CQL, enclose it in double quotes. For example:

```
CREATE TABLE test."add" (a int PRIMARY KEY);
```

Reserved keyword list

- ADD
- ALLOW
- ALTER
- AND
- APPLY
- ASC
- AUTHORIZE
- BATCH
- BEGIN
- BY
- COLUMNFAMILY
- CREATE
- DEFAULT
- DELETE
- DESC
- DESCRIBE
- DROP
- ENTRIES
- EXECUTE
- FROM
- FULL
- GRANT
- IF
- IN
- INDEX
- INFINITY
- INSERT
- INTO
- IS
- KEY
Introduction to CQL commands

- LANGUAGE
- MATERIALIZED
- MBEAN
- MBEANS
- MODIFY
- NAN
- NORECURSIVE
- NOT
- NULL
- OF
- ON
- OR
- ORDER
- PRIMARY
- RENAME
- REPLACE
- REVOKE
- SCHEMA
- SELECT
- SET
- TABLE
- TO
- TOKEN
- TRUNCATE
- UNLOGGED
- UNSET
- UPDATE
- USE
- USING
- VIEW
- WHERE
- WITH

Non-reserved keywords

Non-reserved keywords have a specific meaning in certain context but can be used as an identifier outside this context.

Non-reserved keyword list
• AGGREGATE
• ALL
• ANY
• AS
• ASCII
• BIGINT
• BLOB
• BOOLEAN
• CALLED
• CAST
• CLUSTERING
• COMPACT
• CONTAINS
• COUNT
• COUNTER
• CUSTOM
• DATE
• DECIMAL
• DETERMINISTIC
• DISTINCT
• DOUBLE
• DURATION
• EXISTS
• FILTERING
• FINALFUNC
• FLOAT
• FROZEN
• FUNCTION
• FUNCTIONS
• GROUP
• INET
• INITCOND
• INPUT
• INT
• JSON
Introduction to CQL commands

• KEYS
• KEYSPACE
• KEYSITIES
• LIKE
• LIMIT
• LIST
• LOGIN
• MAP
• MONOTONIC
• NOLOGIN
• NOSUPERUSER
• OPTIONS
• PARTITION
• PASSWORD
• PER
• PERMISSION
• PERMISSIONS
• RESOURCE
• RETURNS
• ROLE
• ROLES
• SFUNC
• SMALLINT
• STATIC
• STORAGE
• STYPE
• SUBMISSION
• SUPERUSER
• TEXT
• TIME
• TIMESTAMP
• TIMEUUID
• TINYINT
• TRIGGER
• TTL
Introduction to CQL commands

• TUPLE
• TYPE
• USER
• USERS
• UUID
• VALUES
• VARCHAR
• VARINT
• WORKPOOL
• WRITETIME

Limitations
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 \(2^{16} \cdot 1\)
• Key length: 65535 \(2^{16} \cdot 1\)
• Table / CF name length: 48 characters
• Keyspace name length: 48 characters
• Query parameters in a query: 65535 \(2^{16} \cdot 1\)
• Statements in a batch: 65535 \(2^{16} \cdot 1\)
• 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 \(2^{16} \cdot 1\)
• Collection (Set): collection limit: ~2 billion \(2^{31}\); values size: 65535 \(2^{16} \cdot 1\)
• Collection (Map): collection limit: number of keys: 65535 \(2^{16} \cdot 1\); values size: 65535 \(2^{16} \cdot 1\)
• Blob size: 2 GB (less than 1 MB is recommended)

The limits specified for collections are for non-frozen collections.
Chapter 6. Managing keyspaces

Create and drop keyspaces. Manage the replication factor (data availability).

About keyspaces

A keyspace is the top-level database object that controls the replication for the object it contains at each datacenter in the cluster. Keyspaces contain tables, materialized views and user-defined types, functions and aggregates. Typically, a cluster has one keyspace per application. Since replication is controlled on a per-keyspace basis, store data with different replication requirements (at the same datacenter) in different keyspaces. Keyspaces are not a significant map layer within the data model.

Replication strategy and factor

When you create or modify a keyspace, specify a Table 2 for replicating keyspaces.

- **SimpleStrategy** class is for evaluating DataStax Distribution of Apache Cassandra™ (DDAC). For production use or for use with mixed workloads, use the NetworkTopologyStrategy class.

- **NetworkTopologyStrategy** class is for production use.

  You can also use NetworkTopologyStrategy for evaluation purposes by changing the default snitch, SimpleSnitch, to a network-aware snitch. Choose a snitch and define one or more datacenter names in the snitch properties file, and then use the datacenter name(s) to set the keyspace replication factor. For example, if the cluster uses the GossipingPropertyFileSnitch, create the keyspace using the user-defined datacenter and rack names in the cassandra-rackdc.properties file. If the cluster uses the Ec2Snitch, create the keyspace using EC2 datacenter and rack names. If the cluster uses the GoogleCloudSnitch, create the keyspace using GoogleCloud datacenter and rack names.

Showing the keyspace details

In cqlsh, the DESCRIBE KEYSPACE command outputs the full details of the keyspace, including all database objects the keyspace contains. Use the output from the DESCRIBE command to recreate the keyspace and all its components.

The output includes all settings for tables and other objects including the default settings. When recreating a keyspace in another environment, verify that the settings are compatible, such as the keyspace replication factor and datacenter names.

Errors related to the SimpleSnitch

When the default snitch, SimpleSnitch, is used with NetworkTopologyStrategy, the write request, such as inserting data into a table, may fail with the following error message:

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

Creating a keyspace

A keyspace is the equivalent to a SQL database. It is the top level container for database objects. The keyspace replication strategy and factor control data availability for a set of tables in each datacenter of the cluster.

- **SimpleStrategy**: Applies the same replication setting across the cluster. Use this setting only on single node test or development environments.

- **NetworkTopography**: Applies the replication setting per datacenter. Use in production environments.
1. **Verify the datacenter name using** `nodetool status`.

   ```bash
   $ installation_location/bin/nodetool status
   ```

   The output is:

   ```
   Datacenter: datacenter1
   ===========
   Status=Up/Down
   |/ State=Normal/Leaving/Joining/Moving
   --  Address         Load       Tokens       Owns    Host ID
   Rack
   UN  10.200.181.134  4.97 MiB   1            ?       c9027497-011c-4390-ae59-
   e4ed1ac794cc  rack1
   UN  10.200.181.135  5.16 MiB   1            ?       af2af8ec-5fa2-4a04-ac9c-
   c75669d8d3a0  rack1
   ```

2. **Verify the datacenter names, which are case sensitive and ensure that you exactly match the case in the next step.**

3. **Start cqlsh and create a keyspace.**

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

4. **Switch to the keyspace.**

   ```
   USE cycling;
   ```

   **CREATE KEYSPACE** *(Defines a new keyspace.)*

### Modifying the replication factor

Increasing the replication factor increases the total number of copies of keyspace data stored in a cluster. 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.

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

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

   ```
   ALTER KEYSPACE keyspace_name
   WITH REPLICATION = {'class' : 'NetworkTopologyStrategy', 'dc_name_1' : 3,
                      'dc_name_2' : 2};
   ```

   Or if using **SimpleStrategy**:

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

   Datacenter names are case sensitive. Verify the case of the using utility, such as `dsetool status`. 
Managing keyspaces

See Changing keyspace replication strategy.

2. Run a full repair of the keyspace when adding datacenters.

```bash
nodetool repair --full keyspace_name
```

**ALTER KEYSSPACE** *Changes keyspace replication strategy and enables or disables commit log.*

Dropping a keyspace

Use the DROP KEYSACE command to remove a keyspace along with all its data and the database objects; such as tables, and any user-defined functions, aggregates, and types contained in the keyspace.

When **auto_snapshot** is enabled in *cassandra.yaml*, the keyspace is automatically backed up on the node where the command is executed. To manually backup the keyspace, see Taking a snapshot.

- Drop the keyspace.

```sql
DROP KEYSACE keyspace_name;
```

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

### Table 3: Columns in system tables

<table>
<thead>
<tr>
<th>Table name</th>
<th>Column name</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>available_ranges</td>
<td>keyspace_name, ranges</td>
<td></td>
</tr>
<tr>
<td>batches</td>
<td>id, mutations, version</td>
<td></td>
</tr>
<tr>
<td>batchlog</td>
<td>id, data, version, written_at</td>
<td></td>
</tr>
<tr>
<td>built_views</td>
<td>keyspace_name, view_name</td>
<td>Information on materialized views</td>
</tr>
<tr>
<td>compaction_history</td>
<td>id, bytes_in, bytes_out, columnfamily_name, compacted_at, keyspace_name,</td>
<td>Information on compaction history</td>
</tr>
<tr>
<td></td>
<td>rows_merged</td>
<td></td>
</tr>
<tr>
<td>&quot;IndexInfo&quot;</td>
<td>table_name, index_name</td>
<td>Information on indexes</td>
</tr>
<tr>
<td>local</td>
<td>key, bootstrapped, broadcast_address, cluster_name, cql_version,</td>
<td>Information on a node has about itself and a superset of gossip.</td>
</tr>
<tr>
<td></td>
<td>data_center, dse_version, gossip_generation, graph, host_id, listen_address,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>native_protocol_version, partitioner, rack, release_version, rpc_address,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>schema_version, server_id, thrift_version, tokens, truncated_at, workload,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>workloads</td>
<td></td>
</tr>
<tr>
<td>paxos</td>
<td>row_key, cf_id, in_progress_ballot, most_recent_commit,</td>
<td>Information on lightweight Paxos transactions</td>
</tr>
<tr>
<td></td>
<td>most_recent_commit_at, most_recent_commit_version, proposal,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>proposal_ballot, proposal_version</td>
<td></td>
</tr>
<tr>
<td>peers</td>
<td>peer, data_center, dse_version, graph, host_id, preferred_ip, rack,</td>
<td>Each node records what other nodes tell it about themselves over the gossip.</td>
</tr>
<tr>
<td></td>
<td>release_version, rpc_address, schema_version, server_id, tokens, workload,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>workloads</td>
<td></td>
</tr>
<tr>
<td>peer_events</td>
<td>peer, hints_dropped</td>
<td></td>
</tr>
<tr>
<td>prepared_statements</td>
<td>prepared_id, logged_keyspace, query_string</td>
<td>Prepared statements used by drivers for queries.</td>
</tr>
<tr>
<td>range_xfers</td>
<td>token_bytes, requested_at</td>
<td></td>
</tr>
<tr>
<td>size_estimates</td>
<td>keyspace_name, table_name, range_start, range_end, mean_partition_size,</td>
<td>Information on partitions</td>
</tr>
<tr>
<td></td>
<td>partitions_count</td>
<td></td>
</tr>
<tr>
<td>sstable_activity</td>
<td>keyspace_name, columnfamily_name, generation, rate_120m, rate_15m</td>
<td></td>
</tr>
<tr>
<td>views_builds_in_progress</td>
<td>keyspace_name, view_name, generation_number, last_token</td>
<td></td>
</tr>
</tbody>
</table>
### Table 4: Columns in system_schema tables

<table>
<thead>
<tr>
<th>Table name</th>
<th>Column name</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>aggregates</td>
<td>keyspace_name, aggregate_name, argument_types, final_func, initcond, return_type, state_func, state_type</td>
<td>Information about user-defined aggregates</td>
</tr>
<tr>
<td>columns</td>
<td>keyspace_name, table_name, column_name, clustering_order, column_name_bytes, kind, position, type</td>
<td>Information about table columns</td>
</tr>
<tr>
<td>dropped_columns</td>
<td>keyspace_name, table_name, column_name, dropped_time, type</td>
<td>Information about dropped columns</td>
</tr>
<tr>
<td>functions</td>
<td>keyspace_name, function_name, argument_types, argument_names, body, called_on_null_input, language, return_type</td>
<td>Information on user-defined functions</td>
</tr>
<tr>
<td>indexes</td>
<td>keyspace_name, table_name, index_name, kind, options</td>
<td>Information about indexes</td>
</tr>
<tr>
<td>keyspaces</td>
<td>keyspace_name, durable_writes, replication</td>
<td>Information on keyspace durable writes and replication</td>
</tr>
<tr>
<td>tables</td>
<td>keyspace_name, table_name, bloom_filter_fp_chance, caching, comment, compaction, compression, crc_check_chance, dlocal_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</td>
<td>Information on columns and column indexes. Used internally for compound primary keys.</td>
</tr>
<tr>
<td>triggers</td>
<td>keyspace_name, table_name, trigger_name, options</td>
<td>Information on triggers</td>
</tr>
<tr>
<td>types</td>
<td>keyspace_name, type_name, field_names, field_types</td>
<td>Information about user-defined types</td>
</tr>
<tr>
<td>views</td>
<td>keyspace_name, view_name, base_table_id, base_table_name, bloom_filter_fp_chance, caching, comment, compaction, compression, crc_check_chance, dlocal_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</td>
<td>Information about materialized views</td>
</tr>
</tbody>
</table>

### Table 5: Columns in system_distributed tables

<table>
<thead>
<tr>
<th>Table name</th>
<th>Column names</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>repair_history</td>
<td>keyspace_name, columnfamily_name, id, coordinator, exception_message, exception_stacktrace, finished_at, parent_id, participants, range_begin, range_end, started_at, status</td>
<td>Contains detailed information on keyspace repair history.</td>
</tr>
</tbody>
</table>

### Security keyspaces and tables

Read access to these system tables is implicitly given to every authenticated user because the tables are used by most DSE tools:

- system_auth keyspace
  - Contains authorization and internal authentication data.
### Table 6: system_auth tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Columns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>role_permissions</td>
<td>role (PK), resource, permissions</td>
<td>Stores the role, resource (for example keyspace_name/table_name), and the permission that the role has to access the resource.</td>
</tr>
<tr>
<td>role_members</td>
<td>role (PK), member</td>
<td>Stores the roles and role members.</td>
</tr>
<tr>
<td>roles</td>
<td>role (PK), can_login, is_superuser, member_of, salted_hash</td>
<td>Stores the role name, whether the role can be used for login, whether the role is a superuser, what other roles the role may be a member of, and a bcrypt salted hash password for the role.</td>
</tr>
</tbody>
</table>

**dse_security keyspace**

Contains DSE Spark, Kerberos digest data, and role options.

### Table 7: dse_security tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Columns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>role_options</td>
<td>role, options</td>
<td>Role options.</td>
</tr>
<tr>
<td>digest_tokens</td>
<td>id, password</td>
<td>Kerberos digest tokens when enabled.</td>
</tr>
<tr>
<td>spark_security</td>
<td>dc, shared_secret</td>
<td>Share secret for Spark.</td>
</tr>
</tbody>
</table>
Managing keyspaces

**Keyspace, table, and column information**

An alternative to the `cqlsh DESCRIBE` command or using DataStax Studio 2.0 to discover keyspace, table, and column information is to query the `system_schema` tables directly.

The examples in this section show how to query the `system_schema.keyspaces` table and additional tables in the `system_schema` keyspace.

The `system.schema_keyspaces` table no longer exists in DataStax Enterprise 5.0 and higher, and is replaced by the `system_schema.keyspaces` table.

- Query the `system_schema.keyspaces` table using this `SELECT` statement.

```sql
SELECT *
FROM system_schema.keyspaces;
```

<table>
<thead>
<tr>
<th>keyspace_name</th>
<th>durable_writes</th>
<th>replication</th>
</tr>
</thead>
<tbody>
<tr>
<td>cycling</td>
<td>True</td>
<td>{'class': 'org.apache.cassandra.locator.SimpleStrategy', 'replication_factor': '1'}</td>
</tr>
<tr>
<td>test</td>
<td>True</td>
<td>{'Cassandra': '1', 'class': 'org.apache.cassandra.locator.NetworkTopologyStrategy'}</td>
</tr>
<tr>
<td>system_auth</td>
<td>True</td>
<td>{'class': 'org.apache.cassandra.locator.SimpleStrategy', 'replication_factor': '1'}</td>
</tr>
<tr>
<td>system_schema</td>
<td>True</td>
<td>{'class': 'org.apache.cassandra.locator.LocalStrategy'}</td>
</tr>
<tr>
<td>system1</td>
<td>True</td>
<td>{'class': 'org.apache.cassandra.locator.SimpleStrategy', 'replication_factor': '2'}</td>
</tr>
<tr>
<td>system_distributed</td>
<td>True</td>
<td>{'class': 'org.apache.cassandra.locator.SimpleStrategy', 'replication_factor': '3'}</td>
</tr>
<tr>
<td>system_traces</td>
<td>True</td>
<td>{'class': 'org.apache.cassandra.locator.SimpleStrategy', 'replication_factor': '2'}</td>
</tr>
</tbody>
</table>

(15 rows)

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

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

The following results shows the first record formatted with the cqlsh `EXPAND` on option.

```
@ Row 1
-------------------------------
<table>
<thead>
<tr>
<th>keyspace_name</th>
<th>table_name</th>
<th>cycling</th>
<th>birthday_list</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bloom_filter_fp_chance</td>
<td>0.01</td>
<td></td>
</tr>
<tr>
<td></td>
<td>caching</td>
<td>{'keys': 'ALL', 'rows_per_partition': 'NONE'}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cdc</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td></td>
<td>comment</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

CQL for the DataStax Distribution of Apache Cassandra ™ 3.11 Latest DDAC

patch: 5.1.17
Managing keyspaces

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

```
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 |
|---------------+--------------+-------------+------------------|-------------------|
| cycling       | cyclist_name | firstname   | none             | text              |
|               |              | id          | none             | uuid              |
|               |              | lastname    | none             | text              |

(3 rows)

The `system_schema` tables do 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, datacenter and rack names, token values, and other information. "The Data Dictionary" article describes querying system tables in detail.

1. After setting up a cluster, query the peers table.

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

The following example output is truncated because of the large number of columns.

```
peer      | data_center
----------|-------------
127.0.0.3 | datacenter1
127.0.0.2 | datacenter1
```
Managing keyspaces

Functions, aggregates, and user types

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

- Show all user-defined functions in the system_schema.functions table.

```cql
SELECT *
FROM system_schema.functions;
```

<table>
<thead>
<tr>
<th>keyspace_name</th>
<th>function_name</th>
<th>signature</th>
<th>argument_names</th>
<th>argument_types</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Show all user-defined aggregates in the system_schema.aggregates table.

```cql
SELECT *
FROM system_schema.aggregates;
```

<table>
<thead>
<tr>
<th>keyspace_name</th>
<th>aggregate_name</th>
<th>signature</th>
<th>argument_types</th>
<th>state_func</th>
<th>state_type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Show all user-defined types in the system_schema.types table.

```cql
SELECT *
FROM system_schema.types;
```

<table>
<thead>
<tr>
<th>keyspace_name</th>
<th>type_name</th>
<th>field_names</th>
<th>field_types</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 7. Managing types, functions, and aggregates

In DataStax Distribution of Apache Cassandra™, users can create user-defined functions (UDFs), user-defined aggregate functions (UDAs), and user-defined types. 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)

Write custom functions using Java and other programming languages for use in SELECT, INSERT, and UPDATE statements. Function are only available within the keyspace where it is defined.

You can define functions that are executed against 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 is performed on each row of the table.

Adding additional languages

DataStax Distribution of Apache Cassandra™ supports functions written in Java and JavaScript by default. Add support for other scripting languages, Python, Ruby, and Scala by adding a JAR to the classpath. Install the JAR file into: `installation_location/lib/jsr223/[language]/[jar-name].jar` where language is 'jruby', 'jython', or 'scala'.

Prerequisites:

By default the ability to add user-defined functions is disabled. To enable, change the following settings in the cassandra.yaml file:

- Java: Set `enable_user_defined_functions` to `true`
- Javascript and other custom languages: Set `enable_scripted_user_defined_functions` to `true`.
- 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.

```
CREATE FUNCTION cycling.flog(target_name double)
    CALLED ON NULL INPUT
    RETURNS double
    LANGUAGE java
    AS $$return Double.valueOf(Math.log(input.doubleValue()));$$;
```

- Actions when the input from the target column is null:
  - `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 CQL 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
Creating User-Defined Aggregate Function (UDA)

DataStax Distribution of Apache Cassandra™ 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.

• 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.

```java
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.

```java
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.

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

What's next: For more information on user-defined aggregates, see Cassandra Aggregates - min, max, avg, group by and A few more Cassandra aggregates.

Modifying a user-defined type

Use the ALTER TYPE command to add new fields to a user-defined type or to rename an existing field.
• Add a **middlename** field of type **text** to the user-defined type **cycling.fullname**.

```sql
ALTER TYPE cycling.fullname ADD middlename text;
```

This creates the field metadata and adds the field to the type schema.

To verify the changes, use **DESC TYPE**.

```sql
DESC TYPE cycling.fullname;
```

The middle name columns shows in the type definition.

```sql
CREATE TYPE cycling.fullname(
    firstname text,
    lastname text,
    middlename text
);
```

• To change the name of an existing field use **RENAME**.

```sql
ALTER TYPE cycling.fullname
RENAME middlename TO middleinitial;
```

```sql
DESC TYPE cycling.fullname;
```

Shows the new name in the type definition.

```sql
CREATE TYPE cycling.fullname(
    firstname text,
    lastname text,
    middleinitial text
);
```

**Dropping a user-defined function (UDF)**

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

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

```sql
DROP FUNCTION [IF EXISTS] fLog;
```
Chapter 8. Managing tables

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.

The concept of rows and columns in the internal implementation of databases is 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 table_options.

Create schema using cqlsh

Create table schema using cqlsh. DataStax Distribution of Apache Cassandra™ (DDAC) does not support dynamic schema generation — collisions 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’s database 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.

The definition of a table’s primary key is critical. 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:

```sql
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
**Freezing collection types**

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:

```sql
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));
```

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 UDT column requires the **frozen** keyword. 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. The database accesses the type in the specified
keyspace, but does not change the current keyspace; otherwise, if you do not specify a keyspace, the database
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**.

**Defining a basic primary key**

For a table with a simple primary key, DataStax Distribution of Apache Cassandra™ (DDAC) 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 (unless you use secondary indexes). 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.

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.
Managing tables

- 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.

  ```
  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.

  ```
  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.

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

### Defining a multi-column partition key

For a table with a composite partition key, DataStax Distribution of Apache Cassandra™ (DDAC) 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 cluster experiences hotpotting, or congestion in writing data to one node repeatedly, because a partition is heavily writing. Cassandra is often used for time series data, and hotpotting 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, if secondary indexes are not used. The table shown uses **race_year** and **race_name** in the primary key, as a composite partition key. To retrieve data, both parameters must be identified.

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

### 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 composite 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.

- 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.

```cql
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.

```cql
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)
};
```

Defining a partition key with clustering columns

For a table with a compound primary key, DataStax Distribution of Apache 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 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.

<table>
<thead>
<tr>
<th>category</th>
<th>points</th>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td>One-day-races</td>
<td>367</td>
<td>112006464-f48-496-949-8d06-8d06-965d3a795c9f</td>
</tr>
<tr>
<td>One-day-races</td>
<td>198</td>
<td>112006464-f48-496-949-8d06-8d06-965d3a795c9f</td>
</tr>
<tr>
<td>Team-trial</td>
<td>1022</td>
<td>112006464-f48-496-949-8d06-8d06-965d3a795c9f</td>
</tr>
<tr>
<td>Sprint</td>
<td>39</td>
<td>112006464-f48-496-949-8d06-8d06-965d3a795c9f</td>
</tr>
<tr>
<td>Sprint</td>
<td>1260</td>
<td>112006464-f48-496-949-8d06-8d06-965d3a795c9f</td>
</tr>
</tbody>
</table>

The database 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 DataStax Distribution of Apache Cassandra™, you denormalize the data because joins are not performant in a distributed system. Later, other schema are shown that improve database 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.

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

• 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.

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

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.

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

In both of these examples, points is defined as a clustering column.

Using advanced data types for columns

Creating collections

DataStax Distribution of Apache Cassandra™ (DDAC) 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.

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

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

Also, see Using frozen in a collection.

**Using 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.

In the following example, a set called teams stores all the teams that a cyclist has been a member of during their career.

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

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

**Using 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.

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

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

**Using 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.
Managing tables

Using the map type, you can store timestamp-related information in user profiles. Each element of the map is internally stored as a single column that you can modify, replace, delete, and query. Each element can have an individual time-to-live and expire when the TTL ends.

1. 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.

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

Freezing collection types

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

```cql
frozen<set<data_type>>
```

To nest a collection type, freeze the nested collection. For example, nest a set within a map:

```cql
map<frozen <set<int>>>  
```

Indexes may be created on a collection column of any type.

Using frozen in a collection

Use frozen on a set, map, or list to serialize multiple components into a single value, `frozen<collection_definition>`. Non-frozen types allow updates to individual fields, but values in a frozen collection are treated like blobs, any upsert overwrites the entire value.

```cql
column_name collection_type<data_type, frozen<column_name>>
```

For example:

```cql
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   );
```

In a non-frozen collection, a tombstone is created for an insert and a non-incremental update in the collection. An incremental update adds a value to an existing value in the collection. The inserts and non-incremental updates for a non-frozen collection can cause large numbers of tombstones.
Managing tables

Creating a tuple column
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.

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

```cql
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.

```cql
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.

```cql
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.

- Use the `cycling` keyspace.

```cql
USE cycling;
```

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

```cql
CREATE TYPE cycling.basic_info (  
    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. In DataStax Distribution of Apache Cassandra™, the `frozen` keyword is not required for UDTs that contain only non-collection fields.

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

When using the `frozen` keyword, you cannot update parts of a user-defined type value. The entire value must be overwritten. The database treats the value of a frozen, user-defined type like a blob.
A user-defined type can be nested in another column type. This example nests a UDT in a list.

```cql
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 Blob column**

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`:

```cql
CREATE TABLE bios {
    user_name varchar PRIMARY KEY,
    bio blob
};
INSERT INTO bios (user_name, bio) VALUES ('fred', bigintAsBlob(3));
SELECT * FROM bios;
```

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

This example shows how to use `blobAsBigInt`.

```cql
ALTER TABLE bios ADD id bigint;
INSERT INTO bios (user_name, id) VALUES ('fred', blobAsBigInt(0x0000000000000003));
SELECT * FROM bios;
```

```cql
user_name | bio | id
-----------|------|----
fred | 0x0000000000000003 |    
```
Using UUID and timeuuid types

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

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:

```cql
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) |
```

You can filter a selection using a tuple.

```cql
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:

```cql
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:
Managing tables

- `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. The database accesses the type in the specified keyspace, but does not change the current keyspace; otherwise, if you do not specify a keyspace, the database accesses the type within the current keyspace.

Creating columns with a single value

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.

```cql
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:

<table>
<thead>
<tr>
<th>k</th>
<th>s</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>&quot;I'm still shared&quot;</td>
<td>0</td>
</tr>
<tr>
<td>k</td>
<td>&quot;I'm still shared&quot;</td>
<td>1</td>
</tr>
</tbody>
</table>

- 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, the database retrieves only the beginning (static column) of the partition.

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.

<table>
<thead>
<tr>
<th>id</th>
<th>popularity</th>
</tr>
</thead>
<tbody>
<tr>
<td>6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47</td>
<td>62</td>
</tr>
</tbody>
</table>

Tracking count in a distributed database presents an interesting challenge. In DataStax Distribution of Apache 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.

Because counters are implemented differently from other columns, counter columns can only be created in dedicated tables. A counter column must have the 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. The database 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. DataStax Distribution of Apache Cassandra™ rejects USING TIMESTAMP or USING TTL in the command to update a counter column.

• Create a table for the counter column.

```sql
USE cycling;
CREATE TABLE popular_count (  id UUID PRIMARY KEY,  popularity counter);
```

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

```sql
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.

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

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

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.
Managing tables

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.

Creating a 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.

Fixing a table schema collision

Dynamic schema creation or updates can cause a schema collision that results in errors.

1. Run a rolling restart on all nodes to ensure the schema matches. Run `nodetool describecluster` on all nodes. Ensure that there is only one schema version.

2. On each node, examine the data directory for the table to fix. The default location for the data directory is /var/lib/cassandra/data. If there is only one directory for the table, go to the next node and repeat this step. If there are two or more directories for the table, continue to the next step.

   If there are two directories, the older directory contains the old table data before the update. The newer directory contains the new data after the update.

3. Identify which id (column family ID) is the newest table ID in `system_schema.tables`.

   ```
   $ cqlsh -e "SELECT * FROM system_schema.tables" | grep <tablename>
   ```

4. Move the data from the older table to the newer table's directory, and then remove the older directory. Repeat this step as needed.

5. Run `nodetool refresh`.

Altering a table

Change tables using 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.

- **Add an age column of type int to the table **cycling.cyclist_alt_stats**.**

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

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

  Verify that the column was added with null values.

  ```
  SELECT id, age FROM cycling.cyclist_alt_stats LIMIT 3;
  ```

  The result set shows the first three rows.

  ```
  id                                   | age
  --------------------------------------+------
  e0953617-07eb-4c82-8f91-3b2757981625 | null
  a9e96714-2dd0-41f9-8bd0-557196a44ecf | null
  ed584e99-80f7-4b13-9a90-9dc5571e6821 | null
  (3 rows)
  ```

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

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

  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

Use the **ALTER TABLE** command to add new collection column to a table.

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

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

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

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

  Verify the results.

  ```
  SELECT description FROM cycling.upcoming_calendar WHERE year = 2015 AND month = 6;
  ```

  Map type columns display in a JSON format.

  ```
  description
  ```
Managing tables

Altering the table properties

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

1. Alter a table to change the caching properties.

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

Dropping a table

You drop a table or keyspace using the DROP command.

- Drop the test keyspace.
  ```sql
  DROP KEYSPACE test;
  ```
- Drop the cycling.last_3_days table.
  ```sql
  DROP TABLE cycling.last_3_days;
  ```

Indexing tables

Data can be queried from tables using indexes, once created.

Indexing

An index provides a means to access data in DataStax Distribution of Apache 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. A number of techniques exist 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, static columns, and any other columns except counter columns.

When to use an index

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.

• Do not add a secondary index and a search index to the same table.

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 built-in index. For columns containing unique data, it is sometimes better for performance 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

The database 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

Create indexes on a column after defining a table. Secondary indexes are used to query a table using a column that is not normally query-able.

Secondary indexes 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 problem if multiple nodes are accessed. A general rule is to index a column with low cardinality of few values. Before creating an index, be aware of when not to create an index.

• The table rank_by_year_and_name stores the rank of cyclists for races.

```
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.

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

<table>
<thead>
<tr>
<th>race_year</th>
<th>race_name</th>
<th>rank</th>
<th>cyclist_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>1</td>
<td>Benjamin PRADIES</td>
</tr>
<tr>
<td>2015</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>2</td>
<td>Adam PHELAN</td>
</tr>
</tbody>
</table>
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.

```cql
SELECT * FROM cycling.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, DataStax Distribution of Apache Cassandra™ assigns a name like `race_year_idx`.

```cql
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.

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

<table>
<thead>
<tr>
<th>race_year</th>
<th>race_name</th>
<th>rank</th>
<th>cyclist_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2014</td>
<td>4th Tour of Beijing</td>
<td>1</td>
<td>Phillippe GILBERT</td>
</tr>
<tr>
<td>2015</td>
<td>Giro d'Italia - Stage 11 - Forli &gt; Imola</td>
<td>1</td>
<td>Ilnur ZAKARIN</td>
</tr>
<tr>
<td>2015</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>1</td>
<td>Benjamin PRADES</td>
</tr>
</tbody>
</table>

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.
• The table `cycling.alt_stats` can yield the statistics about cyclists.

```cql
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`.

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

• Query for all the cyclists with a particular `birthday` from a certain `country`.

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

• 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, the database requires the ALLOW FILTERING directive. The error is not due to multiple indexes, but the lack of a partition key definition in the query.

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

### Indexing a collection
Collections can be indexed and queried to find a collection containing a particular value. **Sets** and **lists** are indexed a bit 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.

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

• 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.

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

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

• 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.

```cql
CREATE INDEX team_year_idx
ON cycling.cyclist_teams (KEYS (teams));

SELECT *
FROM cycling.cyclist_teams
WHERE teams CONTAINS KEY 2015;
```

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

```cql
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';
```

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

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

• Create an index on the map values and find cyclists who have a particular value found in the specified map.

```cql
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
WHERE blist['nation'] = 'NETHERLANDS';
```
Managing tables

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';

<table>
<thead>
<tr>
<th>cyclist_name</th>
<th>blist</th>
</tr>
</thead>
<tbody>
<tr>
<td>Luc HAGENAARS</td>
<td>['age': '28', 'bday': '27/07/1987', 'nation': 'NETHERLANDS']</td>
</tr>
<tr>
<td>Toine POELS</td>
<td>['age': '52', 'bday': '27/07/1963', 'nation': 'NETHERLANDS']</td>
</tr>
</tbody>
</table>

- 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];

<table>
<thead>
<tr>
<th>cyclist_name</th>
<th>rnumbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>John DEGENKOLB</td>
<td>[39, 7, 14]</td>
</tr>
</tbody>
</table>

Building and maintaining indexes

An advantage of indexes is the operational ease of populating and maintaining the index. Indexes are built in the background automatically, without blocking reads or writes. Client-maintained tables as indexes must be created manually; for example, if the rank column had been indexed by creating a table such as rank_idx_table, your client application would have to populate the table with data from the rank_by_year_and_name table.

To perform a hot rebuild of an index, use the node tool rebuild_index command.
Chapter 9. Querying data

Retrieve data sets from a table using SELECT statements.

About SELECT statements

Use a SELECT statement to retrieve a results set from a table as standard output or in JSON format. Functions allow you to manipulate the column values or aggregate the entire results set of specific columns.

```
SELECT column_list FROM [keyspace_name.|table_name
[WHERE primary_key_conditions[ AND clustering_columns_conditions]]] | PRIMARY KEY LIMIT
```

Setting consistency levels

In a distributed system such as DataStax Distribution of Apache 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.

Data consistency and 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;
```

Testing performance impact

Tracing records all activity related to a request. These steps use tracing to show queries on a keyspace with a replication factor of three using different consistency levels (CL):

- **ONE** processes responses from one of three replicas
- **QUORUM** from two of three replicas
- **ALL** from three of three replicas

For more information on tracing data, see this post on the DataStax Support Blog, which explains in detail how to locate data on disk.

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

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

2. Create a table, and insert some values:

   ```
   USE cycling_alt;
   CREATE TABLE cycling_alt.tester
   ( id int PRIMARY KEY, col1 int, col2 int );
   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.

   ```
   TRACING on;
   CONSISTENCY;
   ```

   The output should be:

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

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

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

   The output includes tracing information:

   ```
   id | col1 | col2
   ----+------|------
   0 | 0 | 0
   (1 rows)
   ```
Tracing session: 983f3ae0-fdfe-11e6-8b40-23a5e4e49022

<table>
<thead>
<tr>
<th>activity</th>
<th>timestamp</th>
<th>source_elapsed</th>
<th>client</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute CQL3 query</td>
<td>2017-02-28 21:40:59.918000</td>
<td>0</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Parsing SELECT * FROM cycling_alt.tester</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WHERE id = 0;</td>
<td>2017-02-28 21:40:59.918000</td>
<td>248</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Preparing statement</td>
<td>2017-02-28 21:40:59.918000</td>
<td>473</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>speculating read retry</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sending READ message to /10.200.176.229 message size 134 bytes</td>
<td>2017-02-28 21:40:59.921000</td>
<td>2660</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Executing single-partition query on tester [ReadStage-3]</td>
<td>2017-02-28 21:40:59.923000</td>
<td>4706</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Acquiring sstable references [ReadStage-3]</td>
<td>2017-02-28 21:40:59.923000</td>
<td>4841</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Merging memtable contents [ReadStage-3]</td>
<td>2017-02-28 21:40:59.923000</td>
<td>4910</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Read 1 live and 0 tombstone cells [ReadStage-3]</td>
<td>2017-02-28 21:40:59.923000</td>
<td>5192</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>REQUEST_RESPONSE message received from /10.200.176.229</td>
<td>2017-02-28 21:40:59.924000</td>
<td>5644</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Processing response from /10.200.176.229 [RequestResponseStage-3]</td>
<td>2017-02-28 21:40:59.924000</td>
<td>6116</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Initiating read-repair [RequestResponseStage-3]</td>
<td>2017-02-28 21:40:59.924000</td>
<td>6256</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Executing single-partition query on tester [ReadStage-1]</td>
<td>2017-02-28 21:40:59.969000</td>
<td>711</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Acquiring sstable references [ReadStage-1]</td>
<td>2017-02-28 21:40:59.969000</td>
<td>781</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Merging memtable contents [ReadStage-1]</td>
<td>2017-02-28 21:40:59.969000</td>
<td>824</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Read 1 live and 0 tombstone cells [ReadStage-1]</td>
<td>2017-02-28 21:40:59.969000</td>
<td>1056</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Enqueuing response to /10.200.176.228 [ReadStage-1]</td>
<td>2017-02-28 21:40:59.969000</td>
<td>1118</td>
<td>127.0.0.1</td>
</tr>
</tbody>
</table>
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.

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

<table>
<thead>
<tr>
<th>id</th>
<th>col1</th>
<th>col2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(1 rows)

Tracing session: c96811a0-fdfe-11e6-8b40-23a5e4e49022

<table>
<thead>
<tr>
<th>activity</th>
<th>timestamp</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>source_elapsed</td>
<td>client</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-------------------------------</td>
<td></td>
</tr>
<tr>
<td>Execute CQL3 query</td>
<td>2017-02-28 21:42:22.394000</td>
<td>0</td>
</tr>
<tr>
<td>READ message received from /10.200.176.228</td>
<td>2017-02-28 21:42:22.369000</td>
<td>10.200.176.23</td>
</tr>
<tr>
<td>Executing single-partition query on tester</td>
<td>2017-02-28 21:42:22.370000</td>
<td>10.200.176.23</td>
</tr>
<tr>
<td>Acquiring sstable references</td>
<td>2017-02-28 21:42:22.370000</td>
<td>10.200.176.23</td>
</tr>
<tr>
<td>Merging memtable contents</td>
<td>2017-02-28 21:42:22.370000</td>
<td>10.200.176.23</td>
</tr>
<tr>
<td>Read 1 live and 0 tombstone cells</td>
<td>2017-02-28 21:42:22.371000</td>
<td>10.200.176.23</td>
</tr>
<tr>
<td>Read 1 live and 0 tombstone cells</td>
<td>2017-02-28 21:42:22.371000</td>
<td>10.200.176.23</td>
</tr>
<tr>
<td>Parsing SELECT * FROM cycling_alt.tester WHERE id = 0;</td>
<td>2017-02-28 21:42:22.395000</td>
<td>10.200.176.23</td>
</tr>
<tr>
<td>Preparing statement</td>
<td>2017-02-28 21:42:22.395000</td>
<td>10.200.176.23</td>
</tr>
<tr>
<td>Executing single-partition query on tester</td>
<td>2017-02-28 21:42:22.395000</td>
<td>10.200.176.23</td>
</tr>
<tr>
<td>Querying data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10.200.176.228</td>
<td>1288</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>10.200.176.228</td>
<td>1344</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>reading digest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10.200.176.228</td>
<td>1514</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Read 1 live and 0 tombstone cells [ReadStage-2]</td>
<td>2017-02-28 21:42:22.396000</td>
<td></td>
</tr>
<tr>
<td>10.200.176.228</td>
<td>1584</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>10.200.176.228</td>
<td>1781</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Sending READ message to /10.200.176.229 message size 135 bytes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10.200.176.228</td>
<td>1981</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Sending READ message to /10.200.176.23 message size 135 bytes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10.200.176.228</td>
<td>2243</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>10.200.176.228</td>
<td>4734</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>10.200.176.228</td>
<td>4869</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>10.200.176.228</td>
<td>38</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>10.200.176.228</td>
<td>4559</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>10.200.176.228</td>
<td>4711</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>10.200.176.228</td>
<td>19</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Executing single-partition query on tester [ReadStage-1]</td>
<td>2017-02-28 21:42:22.444000</td>
<td></td>
</tr>
<tr>
<td>10.200.176.229</td>
<td>346</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Acquiring sstable references [ReadStage-1]</td>
<td>2017-02-28 21:42:22.444000</td>
<td></td>
</tr>
<tr>
<td>10.200.176.229</td>
<td>418</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Merging memtable contents [ReadStage-1]</td>
<td>2017-02-28 21:42:22.444000</td>
<td></td>
</tr>
<tr>
<td>10.200.176.229</td>
<td>477</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Read 1 live and 0 tombstone cells [ReadStage-1]</td>
<td>2017-02-28 21:42:22.444000</td>
<td></td>
</tr>
<tr>
<td>10.200.176.229</td>
<td>854</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Read 1 live and 0 tombstone cells [ReadStage-1]</td>
<td>2017-02-28 21:42:22.444000</td>
<td></td>
</tr>
<tr>
<td>10.200.176.229</td>
<td>975</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>10.200.176.229</td>
<td>1039</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Sending REQUEST_RESPONSE message to /10.200.176.228 message size 96 bytes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10.200.176.229</td>
<td>1617</td>
<td>127.0.0.1</td>
</tr>
</tbody>
</table>
6. Change the consistency level to ALL and run the SELECT statement again.

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

<table>
<thead>
<tr>
<th>id</th>
<th>col1</th>
<th>col2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(1 row)

Tracing session: ea9a9aa0-fdfe-11e6-8b40-23a5e4e49022

activity

<table>
<thead>
<tr>
<th>source_elapsed</th>
<th>client</th>
<th>timestamp</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Read 1 live and 0 tombstone cells</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Parsing SELECT * FROM cycling_alt.tester WHERE id = 0;</td>
</tr>
</tbody>
</table>
### Querying data

Sending READ message to /10.200.176.23 message size 135 bytes


Executing


Read 1 live and 0 tombstone cells [ReadStage-3] | 2017-02-28 21:43:18.095000 |

REQUEST_RESPONSE message received from /10.200.176.23

| 10.200.176.228 | 5123 | 127.0.0.1 |


Sending READ message to /10.200.176.229 message size 135 bytes


REQUEST_RESPONSE message received from /10.200.176.229

| 10.200.176.228 | 8493 | 127.0.0.1 |


READ message received from /10.200.176.228

| 10.200.176.228 | 19 | 127.0.0.1 |

Executing


Read 1 live and 0 tombstone cells [ReadStage-2] | 2017-02-28 21:43:18.144000 |

| 10.200.176.228 | 1182 | 127.0.0.1 |

Read 1 live and 0 tombstone cells [ReadStage-2] | 2017-02-28 21:43:18.144000 |

| 10.200.176.228 | 1280 | 127.0.0.1 |


Sending REQUEST_RESPONSE message to /10.200.176.228 message size 96 bytes

| 10.200.176.228 | 1521 | 127.0.0.1 |

| 10.200.176.228 | 10053 | 127.0.0.1 |

### Restricting queries using WHERE clauses

Tables have two types of columns with special roles, the partition key and the clustering columns. Together, they define the primary key.
• **Partition key (PK) columns**: The first part of primary key, define the hash that is used to spread data evenly across the data center.

• **Clustering columns**: The last part of the primary key, order the data within a partition.

Partition keys, clustering, and normal columns have different sets of restrictions within the WHERE clause. Those restrictions differ depending of the type of command: **SELECT**, **UPDATE** or **DELETE**.

**Partition keys**

Identifying the partition in a WHERE clause, allows the database to quickly retrieve the data from the appropriate node. Avoid running queries across multiple partitions whenever possible. The database requires that all partition keys are restricted or none. All the partition key columns are required to compute the hash and locate the node containing the partition.

If no restrictions are specified on the partition keys but some are specified on the clustering keys **ALLOW FILTERING** is required to execute the query.

DataStax recommends limiting queries to a single partition to avoid performance issues that occur with multi-partition operations. Performance issue can occur when using the IN operator, omitting a WHERE clause with logical statements that identifies the partition, or other operators that require the **ALLOW FILTERING** option; for more details see **ALLOW FILTERING explained**.

Use the following operators for partition key logical statements:

• **Equals (=)**

• **IN**

• **Ranges (>, >=, <, <=) on tokenized partition keys**

**Prerequisites:**

This section uses the **rank_by_year_and_name** example.

• **Exact values using equals (=) operator**

To filter on a **regular** or clustering columns, restrict all the partition key columns.

```
SELECT rank, cyclist_name as name FROM cycling.rank_by_year_and_name
WHERE race_name = 'Tour of Japan - Stage 4 - Minami > Shinshu' AND race_year = 2015
AND rank <= 2;
```

• **Values in a list for the last partition key column using the IN operator**

Use the IN operator on the last **partition key column**. For example, to return multiple years of the same race and stage:

```
SELECT rank, cyclist_name as name FROM cycling.rank_by_year_and_name
WHERE race_name = 'Tour of Japan - Stage 4 - Minami > Shinshu' AND race_year IN
(2014, 2015);
```

Using IN query across partitions may cause performance issues, see **Not using the “in” query for multiple partitions** for a detailed explanation.

Results are returned in the natural order of the column type.

• **Partition range using the TOKEN function with >, >=, <, <= operators**

The database distributes the partition across the nodes using the selected **partitioner**. Only the ByteOrderedPartitioner keeps an ordered distribution of data. Select partitions in a range by **tokenizing** the partition keys and using greater than and less than operators.

To filter the results using a token range:

```
SELECT TOKEN(race_year, race_name), race_name AS name, race_year AS year
```
Clustering columns

Clustering columns order data within a partition. When a table has multiple clustering columns the data is stored in nested sort order. The database uses the clustering information to identify where the data is within the partition. Use logical statements for clustering columns to identify the clustering segment and return slices of the data.

A well-designed table uses clustering columns to allow a query to return ranges of data. See CQL data modeling.

When a query contains no restrictions on clustering or index columns, all the data from the partition is returned.

How order impacts clustering restrictions

Because the database uses the clustering columns to determine the location of the data on the partition, you must identify the higher level clustering columns definitively using the equals (=) or IN operators. In a query, you can only restrict the lowest level using the range operators (>, >=, <, or <=).

How data is stored

The following table is used to illustrate how clustering works:

```sql
CREATE TABLE numbers (
    key int,
    col_1 int,
    col_2 int,
    col_3 int,
    col_4 int,
    PRIMARY KEY ((key), col_1, col_2, col_3, col_4));
```

The example table contains the following data:

<table>
<thead>
<tr>
<th>key</th>
<th>col_1</th>
<th>col_2</th>
<th>col_3</th>
<th>col_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

(18 rows)

The database stores and locates the data using a nested sort order. The data is stored in a hierarchy that the query must traverse:

```
{ "key" : "100" 
  "col_1" : "1" 
}
To avoid full scans of the partition and to make queries more efficient, the database requires that the higher level columns in the sort order (col_1, col_2, and col_3) are identified using the equals or IN operators. Ranges are allowed on the last column (col_4).

Selecting data from a clustering segment

For example, to find only values in column 4 that are less than or equal to 2:

```sql
SELECT * FROM numbers
WHERE key = 100
AND col_1 = 1 AND col_2 = 1 AND col_3 = 1
AND col_4 <= 2;
```

The results contain the first two rows:

<table>
<thead>
<tr>
<th>key</th>
<th>col_1</th>
<th>col_2</th>
<th>col_3</th>
<th>col_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

The IN operator can impact performance on medium-large datasets. When selecting multiple segments, the database loads and filters all the specified segments.

For example, to find all values less than or equal to 2 in both col_1 segments 1 and 2:

```sql
SELECT * FROM numbers
WHERE key = 100
AND col_1 IN (1, 2)
AND col_2 = 1 AND col_3 = 1
AND col_4 <= 2;
```

The following visualization shows all the segments the database must load to filter multiple segments:
Querying data

Figure 9:
The results return the range from both segments.

<table>
<thead>
<tr>
<th>key</th>
<th>col_1</th>
<th>col_2</th>
<th>col_3</th>
<th>col_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

(4 rows)

Use **TRACING** to analyze the impact of various queries in your environment.

Invalid restrictions

Queries that attempt to return ranges without identifying any of the higher level segments are rejected:

```
SELECT * FROM numbers
WHERE key = 100
AND col_4 <= 2;
```

The request is invalid:

```
InvalidRequest: Error from server: code=2200 [Invalid query] message="PRIMARY KEY column "col_4" cannot be restricted as preceding column "col_1" is not restricted"
```

You can force the query using the ALLOW FILTERING option; however, this loads the entire partition and negatively impacts performance by causing long READ latencies.

Only restricting top level clustering columns

Unlike partition columns, a query can omit lower level clustering column in logical statements.

For example, to filter one of the mid-level columns, restrict the first level column using equals or IN, then specify a range on the second level:

```
SELECT * FROM numbers
WHERE key = 100 AND col_1 = 1
AND col_2 > 1;
```

The query returns the following data:

<table>
<thead>
<tr>
<th>key</th>
<th>col_1</th>
<th>col_2</th>
<th>col_3</th>
<th>col_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

(3 rows)

**Returning ranges that span clustering segments**

Slicing provides a way to look at an entire clustering segment and find a row that matches values in multiple columns. The slice logical statement finds a single row location and allows a query to return all the rows before, including, between, or after the row.

Slice syntax:

```
(clustering1, clustering2[, ...]) range_operator (value1, value2[, ...])
```
Slices across full partition

The slice determines the exact location within the sorted columns; therefore, the highest level is evaluated first, then the second, and so forth in order to drill down to the precise row location. The following statement identifies the row where column 1, 2, and 3 are equal to 2 and column 4 is less than or equal to 1.

```
SELECT * FROM numbers
WHERE key = 100
AND (col_1, col_2, col_3, col_4) <= (2, 2, 2, 1);
```

The database locates the matching row and then returns every record before the identified row in the results set.
### Querying data

Figure 10:

<table>
<thead>
<tr>
<th>Key</th>
<th>Col_1</th>
<th>Col_2</th>
<th>Col_3</th>
<th>Col_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Querying data

Where col_1 = 1, col_4 contains values 2 and 3 in the results (which are greater than 1). The database is
NOT filtering on all values in column 4, it is finding the exact location shown in dark green. Once it locates the
row, the evaluation ends.

The location might be hypothetical, that is the dataset does not contain a row that exactly matches the values.
For example, the query specifies slice values of (2, 1, 1, 4).

```
SELECT * FROM numbers
WHERE key = 100
AND (col_1, col_2, col_3, col_4) <= (2, 1, 1, 4);
```

The query finds where the row would be in the order if a row with those values existed and returns all rows
before it:
<table>
<thead>
<tr>
<th>Key</th>
<th>Col_1</th>
<th>Col_2</th>
<th>Col_3</th>
<th>Col_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Querying data

The value of column 4 is only evaluated to locate the row placement within the clustering segment. The database locates the segment and then finds col_4 = 4. After finding the location, it returns the row and all the rows before it in the sort order (which in this case spans all clustering columns).

Slices of clustering segments

The same rules apply to slice restrictions when finding a slice on a lower level segment; identify the higher level clustering segments using equals or IN and specify a range on the lower segments.

For example, to return rows where the value is greater than (1, 3) and less than or equal to (2, 5):

```sql
SELECT * FROM numbers
WHERE key = 100 AND col_1 = 1 AND col_2 = 1
AND (col_3, col_4) >= (1, 2)
AND (col_3, col_4) < (2, 3);
```

When finding a between range, the two slice statements must be on the same columns for lowest columns in the hierarchy.
Querying data

Figure 11:

<table>
<thead>
<tr>
<th>Key</th>
<th>Col_1</th>
<th>Col_2</th>
<th>Col_3</th>
<th>Col_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CQL for the DataStax Distribution of Apache Cassandra™ 3.11 Latest DDAC patch: 5.1.17
Querying data

Invalid queries

When returning a slice between two rows, the slice statements must define the same clustering columns. The query is rejected if the columns are different:

```
SELECT * FROM numbers
WHERE key = 100 AND col_1 = 1
AND (col_2, col_3, col_4) >= (1, 1, 2)
AND (col_3, col_4) < (2, 3);
```

InvalidRequest: Error from server: code=2200 [Invalid query]
message="Column "col_3" cannot be restricted by two inequalities not starting with the same column"

• Find the road cycling races that start in 2017 between January 15th and February 14th.

Use the `events` example table and data.

```
CREATE TABLE cycling.events (  
    Year int,  
    Start_Month int,  
    Start_Day int,  
    End_Month int,  
    End_Day int,  
    Race TEXT,  
    Discipline TEXT,  
    Location TEXT,  
    UCI_code TEXT,  
    PRIMARY KEY ((YEAR, Discipline), Start_Month, Start_Day, Race));
```

Limit the `start_month` and `start_day` for the range using a slice:

```
SELECT start_month as month, start_day as day, race FROM cycling.events  
WHERE year = 2017 AND discipline = 'Road'  
AND (start_month, start_day) < (2, 14) AND (start_month, start_day) > (1, 15);
```

The results contain events in that time period:

<table>
<thead>
<tr>
<th>month</th>
<th>day</th>
<th>race</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23</td>
<td>Vuelta Ciclista a la Provincia de San Juan</td>
</tr>
<tr>
<td>1</td>
<td>26</td>
<td>Cadel Evans Great Ocean Road Race – Towards Zero Race Melbourne</td>
</tr>
<tr>
<td>1</td>
<td>26</td>
<td>Challenge Mallorca: Trofeo Porreres-Felanitx-Ses Salines-Campos</td>
</tr>
<tr>
<td>1</td>
<td>28</td>
<td>Cadel Evans Great Ocean Road Race</td>
</tr>
<tr>
<td>1</td>
<td>28</td>
<td>Challenge Mallorca: Trofeo Andratx-Mirador des Colomer</td>
</tr>
<tr>
<td>1</td>
<td>28</td>
<td>Challenge Mallorca: Trofeo Serra de Tramuntana –2017</td>
</tr>
<tr>
<td>1</td>
<td>29</td>
<td>Cadel Evans Great Ocean Road Race</td>
</tr>
<tr>
<td>1</td>
<td>29</td>
<td>Grand Prix Cycliste la Marseillaise</td>
</tr>
<tr>
<td>1</td>
<td>29</td>
<td>Mallorca Challenge: Trofeo Palma</td>
</tr>
<tr>
<td>1</td>
<td>31</td>
<td>Ladies Tour of Qatar</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Etoile de Bessages</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Jayco Herald Sun Tour</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Volta a la Comunitat Valenciana</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>G.P. Costa degli Etruschi</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>Tour of Qatar</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>South African Road Championships</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>Trofeo Laiguegia</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>Clasica de Almeria</td>
</tr>
</tbody>
</table>
Retrieving data

Querying data

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.

- Turn off paging.

```
PAGING OFF
```

- Retrieve and sort results in descending order.

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

<table>
<thead>
<tr>
<th>category</th>
<th>id</th>
<th>points</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sprint</td>
<td>6ab09bec-e68e-48d9-a5f8-9766f6b49cb7</td>
<td>39</td>
<td>KRUIJSWIJK</td>
</tr>
<tr>
<td>Time-trial</td>
<td>6ab09bec-e68e-48d9-a5f8-9766f6b49cb7</td>
<td>3</td>
<td>KRUIJSWIJK</td>
</tr>
<tr>
<td>Sprint</td>
<td>220844bf-48d9-4966-9a4b-6b5d3a79c7bf</td>
<td>0</td>
<td>TIRALONGO</td>
</tr>
<tr>
<td>Time-trial</td>
<td>220844bf-48d9-4966-9a4b-6b5d3a79c7bf</td>
<td>182</td>
<td>TIRALONGO</td>
</tr>
</tbody>
</table>

- Alternatively, retrieve and sort results in ascending order.

To retrieve results, use the SELECT command.

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

<table>
<thead>
<tr>
<th>category</th>
<th>id</th>
<th>points</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time-trial</td>
<td>220844bf-48d9-4966-9a4b-6b5d3a79c7bf</td>
<td>182</td>
<td>TIRALONGO</td>
</tr>
<tr>
<td>Sprint</td>
<td>6ab09bec-e68e-48d9-a5f8-9766f6b49cb7</td>
<td>39</td>
<td>KRUIJSWIJK</td>
</tr>
<tr>
<td>Time-trial</td>
<td>6ab09bec-e68e-48d9-a5f8-9766f6b49cb7</td>
<td>3</td>
<td>KRUIJSWIJK</td>
</tr>
</tbody>
</table>

- 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.

```
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'));
```

<table>
<thead>
<tr>
<th>race_id</th>
<th>race_start_date</th>
<th>race_end_date</th>
<th>race_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>2015-05-09 00:00:00-0700</td>
<td>2015-05-31 00:00:00-0700</td>
<td>Giro d'Italia</td>
</tr>
</tbody>
</table>

- Retrieve several rows using multiple clustering columns and inequality.

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

<table>
<thead>
<tr>
<th>race_id</th>
<th>race_start_date</th>
<th>race_end_date</th>
<th>race_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>2015-05-09 00:00:00-0700</td>
<td>2015-05-31 00:00:00-0700</td>
<td>Giro d'Italia</td>
</tr>
<tr>
<td>101</td>
<td>2015-06-07 00:00:00-0700</td>
<td>2015-06-14 00:00:00-0700</td>
<td>Critérium du Dauphine</td>
</tr>
<tr>
<td>102</td>
<td>2015-06-13 00:00:00-0700</td>
<td>2015-06-21 00:00:00-0700</td>
<td>Tour de Suisse</td>
</tr>
</tbody>
</table>
Querying data

Sorting and limiting 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. This section uses several example tables in the cycling keyspace.

Controlling the number of rows returned using PER PARTITION LIMIT

In DataStax Distribution of Apache Cassandra™ (DDAC), the PER PARTITION LIMIT option sets the maximum number of rows that the query returns from each partition. Create a table that will sort data into more than one partition.

```cql
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)
);
```

After inserting data, the table holds:

<table>
<thead>
<tr>
<th>race_year</th>
<th>race_name</th>
<th>rank</th>
<th>cyclist_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2014</td>
<td>4th Tour of Beijing</td>
<td>1</td>
<td>Phillippe GILBERT</td>
</tr>
<tr>
<td>2014</td>
<td>4th Tour of Beijing</td>
<td>2</td>
<td>Daniel MARTIN</td>
</tr>
<tr>
<td>2014</td>
<td>4th Tour of Beijing</td>
<td>3</td>
<td>Johan Esteban CHAVES</td>
</tr>
<tr>
<td>2014</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>1</td>
<td>Daniel MARTIN</td>
</tr>
<tr>
<td>2014</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>2</td>
<td>Johan Esteban CHAVES</td>
</tr>
<tr>
<td>2014</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>3</td>
<td>Benjamin PRADES</td>
</tr>
<tr>
<td>2015</td>
<td>Giro d'Italia - Stage 11 - Forli &gt; Imola</td>
<td>1</td>
<td>Ilnur ZAKARIN</td>
</tr>
<tr>
<td>2015</td>
<td>Giro d'Italia - Stage 11 - Forli &gt; Imola</td>
<td>2</td>
<td>Carlos BETANCUR</td>
</tr>
<tr>
<td>2015</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>1</td>
<td>Benjamin PRADES</td>
</tr>
<tr>
<td>2015</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>2</td>
<td>Adam PHELAN</td>
</tr>
</tbody>
</table>

Now, to get the top two racers in every race year and race name, use the following command with **PER PARTITION LIMIT 2**.

```
SELECT * FROM cycling.rank_by_year_and_name PER PARTITION LIMIT 2;
```

Output:

<table>
<thead>
<tr>
<th>race_year</th>
<th>race_name</th>
<th>rank</th>
<th>cyclist_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2014</td>
<td>4th Tour of Beijing</td>
<td>1</td>
<td>Phillippe GILBERT</td>
</tr>
<tr>
<td>2014</td>
<td>4th Tour of Beijing</td>
<td>2</td>
<td>Daniel MARTIN</td>
</tr>
<tr>
<td>2015</td>
<td>Giro d'Italia - Stage 11 - Forli &gt; Imola</td>
<td>1</td>
<td>Ilnur ZAKARIN</td>
</tr>
<tr>
<td>2015</td>
<td>Giro d'Italia - Stage 11 - Forli &gt; Imola</td>
<td>2</td>
<td>Carlos BETANCUR</td>
</tr>
<tr>
<td>2015</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>1</td>
<td>Benjamin PRADES</td>
</tr>
<tr>
<td>2015</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>2</td>
<td>Adam PHELAN</td>
</tr>
</tbody>
</table>

- Use a simple **SELECT** query to display all data from the cyclist category table.

```cql
CREATE TABLE cycling.cyclist_category (
    category text,
    points int,
    PRIMARY KEY (points);
```

Page 104 CQL for the DataStax Distribution of Apache Cassandra™ 3.11 Latest DDC patch: 5.1.17
id UUID,
    lastname text,
    PRIMARY KEY (category, points)
    WITH CLUSTERING ORDER BY (points DESC);

SELECT *
FROM cycling.cyclist_category;

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

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

```sql
SELECT *
FROM cycling.cyclist_category
WHERE category = 'Sprint';
```

<table>
<thead>
<tr>
<th>category</th>
<th>points</th>
<th>id</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sprint</td>
<td>39</td>
<td>6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47</td>
<td>KRUIJSWIJK</td>
</tr>
<tr>
<td>Sprint</td>
<td>0</td>
<td>220844bf-4860-49d6-9a4b-6b5d3a79cbfb</td>
<td>TIRALONGO</td>
</tr>
</tbody>
</table>

Note that the database rejects this query if category is not a partition key or clustering column. Queries require a sequential retrieval across the entire cyclist_category table. In 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.

You can filter queries using secondary indexes, see Building and maintaining indexes. A query based on lastname can result in satisfactory results if the lastname column is indexed.

• In Cassandra, clustering columns can be defined in WHERE clauses if ALLOW FILTERING is also used even if a secondary index is not created. The table definition is given and then the SELECT command. Note that race_start_date is a clustering column that has no secondary index.

```sql
CREATE TABLE cycling.calendar (
    race_id int,
    race_name text,
    race_start_date timestamp,
    race_end_date timestamp,
    PRIMARY KEY (race_id, race_start_date, race_end_date)
);

SELECT *
FROM cycling.calendar
WHERE race_start_date = '2015-06-13'
```
Querying data

ALLOW FILTERING;

<table>
<thead>
<tr>
<th>race_id</th>
<th>race_start_date</th>
<th>race_end_date</th>
<th>race_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>102</td>
<td>2015-06-13 07:00:00.000000+0000</td>
<td>2015-06-21 07:00:00.000000+0000</td>
<td>Tour de Suisse</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>category</th>
<th>points</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPRINT</td>
<td>34</td>
<td>THOMAS</td>
</tr>
<tr>
<td>SPRINT</td>
<td>120</td>
<td>JIM</td>
</tr>
<tr>
<td>GC</td>
<td>1234</td>
<td>KENNY</td>
</tr>
<tr>
<td>GC</td>
<td>2234</td>
<td>JIM</td>
</tr>
</tbody>
</table>

- 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.

```
CREATE TABLE cycling.cyclist_name (
    id UUID PRIMARY KEY,
    lastname text,
    firstname text
);
SELECT * FROM cycling.cyclist_name
LIMIT 3;
```

<table>
<thead>
<tr>
<th>id</th>
<th>firstname</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>e7ae5c43-d358-4d99-b900-8592f3d39b00</td>
<td>Alex</td>
<td>FRAME</td>
</tr>
<tr>
<td>5b69d26a-3f90-4c93-8f1d-e9bfca9bf3c28</td>
<td>Paolo</td>
<td>TIRALONGO</td>
</tr>
<tr>
<td>228044b5-4860-9a4b-6b5d3a79c5bf</td>
<td>Paolo</td>
<td>TIRALONGO</td>
</tr>
</tbody>
</table>

- 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.

```
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;
```

<table>
<thead>
<tr>
<th>category</th>
<th>points</th>
<th>id</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>GC</td>
<td>788</td>
<td>829ba3a4a4ba-d1f-a4fb-38167a9c7c0</td>
<td>SUTHERLAND</td>
</tr>
<tr>
<td>GC</td>
<td>1269</td>
<td>228044b5-4860-9a4b-6b5d3a79c5bf</td>
<td>TIRALONGO</td>
</tr>
</tbody>
</table>

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

```
SELECT race_name, point_id,
lat_long AS CITY_LATITUDE_LONGITUDE
```
Querying data

FROM cycling.route;

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

- In Cassandra, the **PER PARTITION LIMIT** option sets the maximum number of rows that the query returns from each partition. This is interesting because it allows a query to select a "Top 3" selection if the partitions are separated correctly. Create a table that will sort data into more than one partition and insert some data:

```sql
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>
<thead>
<tr>
<th>race_year</th>
<th>race_name</th>
<th>rank</th>
<th>cyclist_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2014</td>
<td>4th Tour of Beijing</td>
<td>1</td>
<td>Phillippe GILBERT</td>
</tr>
<tr>
<td>2014</td>
<td>4th Tour of Beijing</td>
<td>2</td>
<td>Daniel MARTIN</td>
</tr>
<tr>
<td>2014</td>
<td>4th Tour of Beijing</td>
<td>3</td>
<td>Johan Esteban CHAVES</td>
</tr>
<tr>
<td>2015</td>
<td>Giro d'Italia - Stage 11 - Forli &gt; Imola</td>
<td>1</td>
<td>Ilnur ZAKARIN</td>
</tr>
<tr>
<td>2015</td>
<td>Giro d'Italia - Stage 11 - Forli &gt; Imola</td>
<td>2</td>
<td>Carlos BETANCUR</td>
</tr>
<tr>
<td>2015</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>1</td>
<td>Benjamin PRADES</td>
</tr>
<tr>
<td>2015</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>2</td>
<td>Adam PHELAN</td>
</tr>
<tr>
<td>2015</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>3</td>
<td>Thomas LEBAS</td>
</tr>
</tbody>
</table>

- Now use a **PER PARTITION LIMIT** to get the top two races for each race year and race name pair:

```sql
SELECT *
FROM cycling.rank_by_year_and_name
PER PARTITION LIMIT 2;
```

<table>
<thead>
<tr>
<th>race_year</th>
<th>race_name</th>
<th>rank</th>
<th>cyclist_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2014</td>
<td>4th Tour of Beijing</td>
<td>1</td>
<td>Phillippe GILBERT</td>
</tr>
<tr>
<td>2014</td>
<td>4th Tour of Beijing</td>
<td>2</td>
<td>Daniel MARTIN</td>
</tr>
<tr>
<td>2015</td>
<td>Giro d'Italia - Stage 11 - Forli &gt; Imola</td>
<td>1</td>
<td>Ilnur ZAKARIN</td>
</tr>
<tr>
<td>2015</td>
<td>Giro d'Italia - Stage 11 - Forli &gt; Imola</td>
<td>2</td>
<td>Carlos BETANCUR</td>
</tr>
<tr>
<td>2015</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>1</td>
<td>Benjamin PRADES</td>
</tr>
</tbody>
</table>
Querying data

Returning data from collection columns

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.

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

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

To query a table containing a collection, the database 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.

The database returns results in an order based on elements types 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.

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

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

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

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

```cql
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.

Aggregating results

In DataStax Distribution of Apache Cassandra™, the standard aggregate functions of min, max, avg, sum, and count are built-in functions.
• A table `cyclist_points` records the race points for cyclists.

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

<table>
<thead>
<tr>
<th>id</th>
<th>race_points</th>
<th>firstname</th>
<th>lastname</th>
<th>race_title</th>
</tr>
</thead>
<tbody>
<tr>
<td>e3b19ec4-774a-4d1c-9e5a-dece1e30aac</td>
<td>6</td>
<td>Giorgia</td>
<td>BRONZINI</td>
<td>Trofeo Alfredo Binda - Comune di Cittiglio</td>
</tr>
<tr>
<td>e3b19ec4-774a-4d1c-9e5a-dece1e30aac</td>
<td>75</td>
<td>Giorgia</td>
<td>BRONZINI</td>
<td>Acht van Westerveld</td>
</tr>
<tr>
<td>e3b19ec4-774a-4d1c-9e5a-dece1e30aac</td>
<td>120</td>
<td>Giorgia</td>
<td>BRONZINI</td>
<td>Tour of Championship Island World Cup</td>
</tr>
</tbody>
</table>

• 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.

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

```cql
system.sum(race_points)
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>201</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>country</th>
<th>cyclist_name</th>
<th>flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Belgium</td>
<td>Andre</td>
<td>1</td>
</tr>
<tr>
<td>Belgium</td>
<td>Jacques</td>
<td>1</td>
</tr>
<tr>
<td>France</td>
<td>Andre</td>
<td>3</td>
</tr>
<tr>
<td>France</td>
<td>George</td>
<td>3</td>
</tr>
</tbody>
</table>

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

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

```cql
system.count(cyclist_name)
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>2</td>
</tr>
</tbody>
</table>

**Applying 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.
Querying data

1. List all the data in the table.

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

<table>
<thead>
<tr>
<th>team_name</th>
<th>cyclist_name</th>
<th>race_title</th>
</tr>
</thead>
<tbody>
<tr>
<td>TWENTY16</td>
<td>Lauren KOMANSKI</td>
<td>Amgen Tour of California Women's Race presented by SRAM - Stage 1 - Lake Tahoe</td>
</tr>
<tr>
<td>Lake Tahoe</td>
<td>Hannah BARNES</td>
<td>Amgen Tour of California Women's Race presented by SRAM - Stage 1 - Lake Tahoe</td>
</tr>
<tr>
<td>Lake Tahoe</td>
<td>Katie WALL</td>
<td>Amgen Tour of California Women's Race presented by SRAM - Stage 1 - Lake Tahoe</td>
</tr>
<tr>
<td>Lake Tahoe</td>
<td>Linda VILLUMSEN</td>
<td>Amgen Tour of California Women's Race presented by SRAM - Stage 1 - Lake Tahoe</td>
</tr>
<tr>
<td>Lake Tahoe</td>
<td>Alena AMALIUSIK</td>
<td>Amgen Tour of California Women's Race presented by SRAM - Stage 1 - Lake Tahoe</td>
</tr>
<tr>
<td>Lake Tahoe</td>
<td>Tritek WORRACK</td>
<td>Amgen Tour of California Women's Race presented by SRAM - Stage 1 - Lake Tahoe</td>
</tr>
</tbody>
</table>

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

   ```
   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
   ```

Using user-defined functions (UDF)

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

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

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

<table>
<thead>
<tr>
<th>id</th>
<th>lastname</th>
<th>cycling.log(race_points)</th>
</tr>
</thead>
<tbody>
<tr>
<td>228044b-4860-496e-94a4-6b5d3a79c5fb</td>
<td>TIRALONGO</td>
<td>0.693147</td>
</tr>
<tr>
<td>e319c4-774a-4d1c-9e5a-dece1e30a0ac</td>
<td>BRONZINI</td>
<td>1.79176</td>
</tr>
<tr>
<td>e319c4-774a-4d1c-9e5a-dece1e30a0ac</td>
<td>BRONZINI</td>
<td>4.31749</td>
</tr>
<tr>
<td>e319c4-774a-4d1c-9e5a-dece1e30a0ac</td>
<td>BRONZINI</td>
<td>4.78749</td>
</tr>
</tbody>
</table>

Returning the write timestamp

A table contains a timestamp representing the date and time that a write occurred to a column. Using the `WRTETIME` function in a SELECT statement returns the timestamp that the column was written to the database. The output of the function is microseconds. Counter column writetime is milliseconds.
• 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:

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

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

### Formatting query results as JSON

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:

```cql
cqlsh:cycling> select json name, checkin_id, timestamp from checkin;
```

```
[json]
------------------------------------------------------------------------------------------------------------------
<table>
<thead>
<tr>
<th>name</th>
<th>checkin_id</th>
<th>system.tojson(timestamp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRAND</td>
<td>50554d6e-29bb-11e5-b345-fee8194dc9f</td>
<td>&quot;2016-08-28 21:45:10.406Z&quot;</td>
</tr>
<tr>
<td>VOSS</td>
<td>50554d6e-29bb-11e5-b345-fee819cde9f</td>
<td>&quot;2016-08-28 21:44:04.113Z&quot;</td>
</tr>
</tbody>
</table>
(2 rows)
```

#### Retrieving selected columns in JSON format

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

```cql
cqlsh:cycling> select name, checkin_id, toJson(timestamp) from checkin;
```

```
name | checkin_id                           | system.toJson(timestamp)
-------+--------------------------------------+----------------------------
BRAND | 50554d6e-29bb-11e5-b345-fee8194dc9f | "2016-08-28 21:45:10.406Z" |
VOSS | 50554d6e-29bb-11e5-b345-fee819cde9f | "2016-08-28 21:44:04.113Z" |
```

DataStax Enterprise supports returning a JSON-formatted `timestamp` with complete time zone information.

### 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...
Querying data

for new applications. For an in-depth comparison of Thrift and CQL, see "A Thrift to CQL Upgrade Guide" and CQL for Cassandra experts.

Creating a legacy table

You can create legacy (Thrift/CLI-compatible) tables in CQL using the COMPACT STORAGE directive. The COMPACT STORAGE directive used with the CREATE TABLE command provides backward compatibility with older 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-bcfd-0800200c9a66:,value =, timestamp =1357602286168000 )
    => ( column =7db1a490-5878-11e2-bcfd-0800200c9a66:album, value =4e6f204f6e65205269646573206f722046726565, 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 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, the database 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';
Chapter 10. Inserting and updating data

Data can be inserted into tables using the INSERT command. DataStax Distribution of Apache Cassandra™ can insert JSON data.

Using INSERT and UPDATE to write values

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.

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

```
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', $$Women's Tour of New Zealand$$);
```

UUID and timeuuid column

The `uuid()` function takes no parameters and generates a random Type 4 UUID suitable for use in `INSERT` or `SET` statements.

Several `timeuuid()` functions are designed for use with the `timeuuid()` type:

- `dateOf()`
  Used in a `SELECT` clause, this function extracts the timestamp of a `timeuuid` column in a 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.

The DataStax Distribution of Apache Cassandra™ supports 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:

```cql
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()));
```

Select data and convert it to a new format:

```cql
SELECT a, b, toDate(c), toDate(d) FROM sample_times;
```

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2015-07-13 17:13:37-0700</td>
<td>50554d6e-29bb-11e5-b345-feff819cdc9f</td>
<td>1436832817476</td>
</tr>
</tbody>
</table>

**Set column**

Write multiple values into a set column. If a table specifies a `set` to hold data, then use `INSERT` or `UPDATE` to enter data.
Inserting and updating data

- 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 bloeiit' } );
  ```

- Add an element to a set using 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 using the subtraction (-) operator.

  ```
  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.

  ```
  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.

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

<table>
<thead>
<tr>
<th>lastname</th>
<th>teams</th>
</tr>
</thead>
<tbody>
<tr>
<td>VOS</td>
<td>null</td>
</tr>
</tbody>
</table>

**List fields**

If a table specifies a **list** to hold data, then use **INSERT** or **UPDATE** to enter data.
• Insert data into the list, enclosing values in square brackets.

```cql
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.

```cql
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.

```cql
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.

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

```cql
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, the database 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.

• To 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.

```cql
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.

```cql
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.

**Map fields**

If a table specifies a map to hold data, then use INSERT or UPDATE to enter data.

• 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.

```cql
INSERT INTO cycling.cyclist_teams (id, lastname, firstname, teams) VALUES ('5b6962dd-3f90-4c93-8f61-eabfa4a803e2', 'VOS',
```
Inserting and updating data

'Marianne',
  2012 : 'Rabobank Women Team', 2011 : 'Nederland bloeit ' });

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.

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.

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:

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.

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

Tuple column

Tuples group small amounts of data together and stores the data in a single column.

• 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.

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.

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.

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

User-defined type (UDT) column

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

Inserting data into a UDT
Set or replace user-defined type data, using the \texttt{INSERT} or \texttt{UPDATE} command, and enclosing the user-defined type with curly brackets, separating each key-value pair in the user-defined type by a colon.

\begin{verbatim}
INSERT INTO cycling.cyclist_stats (id, lastname, basics) VALUES ('e7ae5cf3-d358-4d99-b900-85902fda9bb0', 'FRAME', 
\end{verbatim}

Note the inclusion of \texttt{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.

\begin{verbatim}
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' },
]);
\end{verbatim}

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

Updating individual field data in a UDT

In DataStax Distribution of Apache Cassandra™, user-defined types that include only non-collection fields can update individual field values. Update an individual field in user-defined type data using the \texttt{UPDATE} command. The desired key-value pair are defined in the command. In order to update, the UDT must be defined in the \texttt{CREATE TABLE} command as an unfrozen data type.

\begin{verbatim}
CREATE TABLE cycling.cyclist_stats ( id UUID, lastname text, basics basic_info, PRIMARY KEY (id) );
INSERT INTO cycling.cyclist_stats (id, lastname, basics) VALUES (220844bf-4860-49d6-9a4b-6b5d3a79cbfb, 'TIRALONGO', 
{ birthday:'1977-07-08',nationality:'Italy',weight:'63 kg',height:'1.78 m' });
UPDATE cyclist_stats SET basics.birthday = '2000-12-12' WHERE id = 220844bf-4860-49d6-9a4b-6b5d3a79cbfb;
\end{verbatim}

The UDT is defined in the table with \texttt{basics basic_info}. This example shows an inserted row, followed by an update that only updates the value of \texttt{birthday} inside the UDT \texttt{basics}.

\begin{verbatim}
SELECT * FROM cycling.cyclist_stats WHERE id = 220844bf-4860-49d6-9a4b-6b5d3a79cbfb;
\end{verbatim}

\begin{verbatim}
+-----------------+------------------------+-------------------+
|     id          |       basics           |     lastname      |
+-----------------+------------------------+-------------------+
| 220844bf-4860-49d6-9a4b-6b5d3a79cbfb | { birthday:'1977-07-08',nationality:'Italy',weight:'63 kg',height:'1.78 m' } | 'TIRALONGO' |
+-----------------+------------------------+-------------------+
\end{verbatim}
Inserting and updating data

The resulting change is evident, as is the unchanged values for nationality, weight, and height.

UDTs with collection fields must be frozen in table creation, and individual field values cannot be updated.

Date column

Write values using the current day/time or a string format into a date column.

Commands in this section, use the Cycling keyspace and cyclist_alt_stats table and data.

String format

date

date string format:

```
yyyy-mm-dd
```

Where the elements of a date are separated with a hyphen:

- **yyyy**: four digit year
- **mm**: two digit month
- **dd**: two digit day

For example, May 5, 2017:

```
2017-05-05
```

Getting the current day

Use `toDate(now())` to insert the current day into a date field.

- To **INSERT** dates:

  ```
  # Current date (today) for last_race values.
  INSERT INTO cycling.cyclist_alt_stats (id, last_race) VALUES (ed584e99-80f7-4b13-9a90-9dc5571e6821, toDate(now()));
  ```

  ```
  # String format (yyyy-mm-dd) for first_race value.
  INSERT INTO cycling.cyclist_alt_stats (id, first_race) VALUES (ed584e99-80f7-4b13-9a90-9dc5571e6821, '2006-03-15');
  ```

- To **UPDATE** a date field:
# Current date (today) for last_race.

```sql
UPDATE cycling.cyclist_alt_stats
SET last_race = toDate(now())
WHERE id = ed584e99-80f7-4b13-9a90-9dc5571e6821;
```

# Use formatted string (yyyy-mm-dd) for birthday.

```sql
UPDATE cycling.cyclist_alt_stats
SET birthday = '1987-03-07'
WHERE id = ed584e99-80f7-4b13-9a90-9dc5571e6821;
```

Select the date columns from the table.

```sql
SELECT first_race, last_race, birthday FROM cycling.cyclist_alt_stats ;
```

<table>
<thead>
<tr>
<th>first_race</th>
<th>last_race</th>
<th>birthday</th>
</tr>
</thead>
</table>

**Time column**

Write values into a time column.

Commands in this section, use the Cycling keyspace and cyclist_races table.

**String format**

**time**

```sql
time string format:

\[HH:MM:SS[.fff]\]
```

where

- **HH**: two digit hour using a 24-hour clock.
- **MM**: two digit minute.
- **SS**: two digit seconds.
- (Optional) **.fff**: up to three digit sub-seconds; when excluded it's set to zero (0).

For example, one o'clock in the afternoon

```
13:00:00.000
13:00:00
```

- To **INSERT** time:

```sql
INSERT INTO cycling.cyclist_races (id, races)
VALUES (5b6962dd-3f90-4c93-8f61-eabfa4a803e2, [
    { race_time:'07:00:00'},
...]
```
Inserting and updating data

```
{ race_time:'08:00:00' } ]]);
```

- To **UPDATE** a time field:

```
UPDATE cycling.cyclist_races
SET races[1] = { race_time:'06:00:00'}
WHERE id = 5b6962dd-3f90-4c93-8f61-eabfa4a803e2 ;
```

**Timestamp column**

Upset the current or a specific date/time into a timestamp field using string format.

**String format**

A **timestamp** combines both date and time with the addition of time zone in ISO 8601 format:

```
yyyy-mm-dd[ (T | ) hh:mm:ss[.fff] ][(+|-) NNNN]
```

Where only the date portion of the timestamp is required:

- **(Required) date** (**yyyy-mm-dd**), where
  - # **yyyy**: four digit year
  - # **mm**: two digit month
  - # **dd**: two digit day

- **time** (**hh:mm:ss[.fff]**), where
  - # **HH**: two digit hour using a 24-hour clock.
  - # **MM**: two digit minute.
  - # **SS**: two digit seconds.
  - # **fff**: up to three digit sub-seconds; when excluded it's set to zero (0).

  When time is excluded, it's set to zero.

- **timezone** (**(+|-) NNNN**): is the offset from GMT.
  - # **(+|-) indicates whether to add or subtract the NNNN from GMT
  - # **NNNN** is the RFC-822 4-digit time zone, for example **+0000** is GMT and **-0800** is PST.

  When timezone is excluded, it's set to the client or coordinator timezone.

Commands in this section, use the **Cycling keyspace** and **comments** table and data.

**Inserting the current timestamp**

Use functions to insert the current date into date or timestamp fields as follows:

- Current date and time into timestamp field: `toTimestamp(now())` sets the timestamp to the current time of the coordinator.

- Current date (midnight) into timestamp field: `toTimestamp(toDate(now()))` sets the timestamp to the current date beginning of day (midnight).

- Using the current date/time:
# Date with time set to midnight using UTC.

```
INSERT INTO cycling.comments  
    (id, 
     created_at) 
values (  
    e7ae5cf3-d358-4d99-b900-85902fda9bb0,  
    toTimeStamp(toDate(now())));
```

# Full timestamp

```
INSERT INTO cycling.comments  
    (id, 
     created_at) 
values (  
    e7ae5cf3-d358-4d99-b900-85902fda9bb0,  
    toTimeStamp(now()));
```

- Using string format:
  
  # Date with time and no timezone sets the timezone to UTC.

  ```
  INSERT INTO cycling.comments  
    (id, 
     created_at) 
values (  
    e7ae5cf3-d358-4d99-b900-85902fda9bb0,  
    '2017-04-01');
  ```

  # Full timestamp using UTC.

  ```
  INSERT INTO cycling.comments  
    (id, 
     created_at) 
values (  
    e7ae5cf3-d358-4d99-b900-85902fda9bb0,  
    '2017-04-01T11:21:59.001+0000');
  ```

### Duration column

Use a string formatted value to upsert a duration value into a column. A duration can be expressed in the formats described in this section.

Provide values for a duration without quotes.

### Duration format

A duration can be expressed in this format:

```
NyNmoNmWdNhNsMmsMuNsUsNs
```

Where:

- N: Number
- y: Years
- mo: Months
- w: Weeks
- d: Days
Inserting and updating data

- **h**: Hours
- **m**: Minutes
- **s**: Seconds
- **ms**: Milliseconds
- **µs** or **us**: Microseconds
- **ns**: Nanoseconds

For example, 2mo10d12h for 2 months 10 days and 12 hours. Missing duration designators are set to 0.

**ISO 8601 format**

A duration can be expressed in this ISO 8601 format:

\[ PNYMNDTNSNS \]

Where:

- **P**: Period designator placed at the start of the duration
- **N**: Number
- **Y**: Years
- **M**: Months
- **D**: Days
- **T**: Time
  - **H**: Hours
  - **M**: Minutes
  - **S**: Seconds

For example, P4Y6M3DT12H30M5S for four years, six months, three days, twelve hours, thirty minutes, and five seconds.

A duration can also be expressed in this week ISO 8601 format:

\[ PNW \]

Where:

- **P**: Period designator placed at the start of the duration
- **N**: Number
- **W**: Weeks

For example, P6W for six weeks.

**Alternative ISO 8601 format**

A duration can be expressed in this alternative ISO 8601 format:

\[ P[YYYY]-[MM]-[DD]T[hh]:[mm]:[ss] \]

Where:

- **P**: Period designator placed at the start of the duration
• **YYYY**: Years
• **MM**: Months
• **DD**: Days
• **T**: Time
  - **# hh**: Hours
  - **# mm**: Minutes
  - **# ss**: Seconds

For example, `P0004-06-03T12:30:05` for four years, six months, three days, twelve hours, thirty minutes, and five seconds.

• **INSERT** a duration.

```sql
INSERT INTO cycling.race_times (
    finish_time,
    race_name, cyclist_name, race_date
) VALUES (
    '1h4m48s20ms',
    '17th Santos Tour Down Under', 'Rohan DENNIS', '2017-04-14'
);
```

• **UPDATE** a duration.

```sql
UPDATE cycling.race_times
SET finish_time = '4h7m41s19ms'
WHERE race_name = '17th Santos Tour Down Under'
  AND cyclist_name = 'Rohan DENNIS'
  AND race_date = '2017-04-14';
```

**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 round-trips necessary between the CAS coordinators.

Non-equal conditions for lightweight transactions are supported; 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.

• Insert a new cyclist with their **id**.

```sql
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.

```sql
UPDATE cycling.cyclist_name
SET firstname = 'Roxane'
WHERE id = '4647f6d3-7bd2-4085-8d6c-1229351b5498'
```
Inserting and updating data

```
IF firstname = 'Roxxane';
```

# Inserting JSON formatted values

In a production database, inserting columns and column values programmatically is more practical than using \texttt{cqlsh}. The CQL INSERT commands supports JSON to provide a manual testing from the \texttt{cqlsh} command line utility.

Use the following syntax:

```
INSERT INTO [keyspace_name.\]table_name JSON '{"column_name": value [,\ldots]}' [DEFAULT UNSET];
```

Enclose all values other than numbers in double quotes. Booleans, UUID, and other data types typically recognized in \texttt{cqlsh} must be in double quotes.

- To insert JSON data, add JSON to the INSERT command.

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

- When upserting data if any columns are missing from the JSON, the value in the missing column is overwritten with null (by default). The following removes the \texttt{lastname} value "SUTHERLAND" from the previous example:

```
INSERT INTO cycling.cyclist_category JSON '
  "category": "Sprint",
  "points": 780,
  "id": "829aa84a-4bba-411f-a4fb-38167a987cda"
}';
```

- Use the DEFAULT UNSET option to only overwrite values found in the JSON string:

```
INSERT INTO cycling.cyclist_category JSON '
  "category": "Sprint",
  "points": 780,
  "id": "829aa84a-4bba-411f-a4fb-38167a987cda"
}'
DEFAULT UNSET;
```

- Only the PRIMARY KEY fields are required when inserting a new row, any other column not define in the JSON is set to null:

```
INSERT INTO cycling.cyclist_category JSON '{
  "category": "Sprint",
  "points": 700,
  "id": "829aa84a-4bba-411f-a4fb-38167a987cda"
}';
```
Importing data from 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.

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

Deleting values from a column or entire row

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

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

   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.

   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 the database to delete the data automatically after a certain amount of time has elapsed. For details, see Expanding data with Time-To-Live.

Expanding 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 after the data 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.

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

The database storage engine can only encode TTL timestamps through January 19 2038 03:14:07 UTC due to the Year 2038 problem. The TTL date overflow policy determines whether...
Inserting and updating data

requests with expiration timestamps later than the maximum date are rejected or inserted. See -
Dcassandra.expiration_date_overflow_policy=POLICY.

Setting a TTL for a specific column

Use CQL to set the TTL.

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

To remove the TTL from a column, set the 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 the TTL, the entire row is deleted.

Examples

For details and examples, see Expiring data with TTL example.

Expiring data with time-to-live (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).

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

```cql
INSERT INTO cycling.calendar (race_id, race_name, race_start_date, race_end_date)
USING TTL 86400;
```

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

```cql
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 = '2015-05-27';
```

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

```cql
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 = '2015-05-27';
```

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.

Setting the time-to-live (TTL) for value

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

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

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

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

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

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

```plaintext
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';
```
Chapter 11. Batching inserts 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 that are either all executed, or none are executed. 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 on the server 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 have 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 therefore the potential for multi-node access, can significantly increase latency. In all batching, the coordinator node manages all write operations, so that the coordinator node can be a bottleneck.

Good reasons for batching operations:

• 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 sends 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 data inconsistency must not occur.

  Multiple partition batching sends 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:

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

  As previously stated, 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 be a performance bottleneck because of synchronization. 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 if batch operations are truly necessary. For information about the fastest way to load data, see "Batch loading without the Batch keyword."
Good use of BATCH statement

Batch operations can be beneficial. See the following examples for a good use of BATCH. The examples use the table cyclist_expenses:

```sql
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.

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.

```sql
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.

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

```sql
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.

An acknowledgement of a batch statement is returned if the batch operation is successful. The resulting table will only have one record so far.
Batching inserts and updates

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

```
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.

```
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  | 13.44  | Lunch       | True
Vera ADRIAN | 2         | -32.95  | 13.44  | Breakfast   | False
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.

```
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
```

---

Page 132  CQL for the DataStax Distribution of Apache Cassandra™ 3.11 Latest DDAC
patch: 5.1.17
Batching inserts and updates

WHERE cyclist_name = 'Vera ADRIAN'
AND expense_id = 3
IF paid = false;
APPLY BATCH;

Because the column is static, you can provide only the partition key when updating the data. To update
a non-static column, you would also have to provide a clustering key. Using batched conditional updates,
you can maintain a running balance. If the balance were stored in a separate table, maintaining a
running balance would not be possible because a batch having conditional updates cannot span multiple
partitions.

Multiple partition logged batch

- Another example is using `BATCH` for a multiple partition insert involves writing the same data to two related
tables that must be synchronized. The following example modifies multiple partitions, which in general is to be
avoided, but the batch only contains two statements:

```
BEGIN 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;
```

Another common use for this type of batch operation is updating usernames and passwords.

Misuse of `BATCH` statement

Misused `BATCH` statements can cause many problems in DataStax Distribution of Apache 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.

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

```
BEGIN BATCH
  INSERT INTO cycling.cyclist_name (id, lastname, firstname) VALUES (6d5f1663-89c0-45fc-8cfd-60a373b01622,'HOSKINS', 'Melissa');
  INSERT INTO cycling.cyclist_name (id, lastname, firstname) VALUES (6d5f1663-89c0-45fc-8cfd-60a373b01622,'HOSKINS', 'Melissa');
APPLY BATCH;
```
Batching inserts and updates

```cql
) 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.
Chapter 12. CQL reference

Introduction

All of the commands included in the CQL language and cqlsh only commands are available from the cqlsh command line. You can run CQL in a number of ways.

This reference covers CQL and cqlsh based on the CQL specification 3.3.

CQL data types

Data type is declared and enforced for each column in a table.

String types

INSERT or UPDATE string values in single quotes; additional escaping is required for field values that contain single quotes, see Escaping characters. For example, lastname = 'Smith'.

ascii
US-ASCII characters.

text
UTF-8 encoded string.

varchar
UTF-8 encoded string.

Numeric types

INSERT or UPDATE numeric values without quotes. For example, age = 31.

Integers

tinyint
8-bit signed integer.

smallint
16-bit signed integer.

int
32-bit signed integer.

bigint
64-bit signed integer.

varint
Arbitrary-precision integer.

Decimals

The default decimal separator is a period (.), change the decimal separator in the driver settings or using cqlshrc DECIMALSEP option. Internally the decimal separator is stored as a period.

decimal
Variable-precision decimal, supports integers and floats.

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.

float
CQL reference

32-bit IEEE-754 floating point.

**double**

64-bit IEEE-754 floating point.

**Date and time types**

**INSERT** or **UPDATE** date/time values using single quotes around string format or no quotes for integers. For example, setting a date in string format `purchase_date = '2017-05-12'` vs specifying it as an integer in days since epoch `purchase_date = 17298`.

**date**

32-bit unsigned integer representing the number of days since epoch (January 1, 1970) with no corresponding time value.

**INSERT** or **UPDATE** values as an integer (days since epoch) or in string format `

'yyyy-mm-dd'

`, for example `2017-05-13`.

When loading data from CSV use `datetimeformat` option in a `cqlshrc` file to change the `cqlsh` `COPY TO` date parsing format.

**DateRangeType**

Stores a date range. Truncated timestamps represent the entire date span or use the range syntax to create a custom range.


```
'beginning_date TO end_date'
```

Specify a custom range between single quotes and, where necessary, square brackets. For example:

- `2018-01` – Beginning of the first day to the end of the last day in January 2018.
- `2018-01T15` – All minutes in a specific hour. For example, 15 is 3:00 PM to 3:59 PM.
- `[2017-01-15 TO 2017-11-01]` – The start of the fifteenth of January through the end of the first day of November.
- `[2017 TO 2017-11-01]` – Start of 2017 till the end of the first day of November.
- `[* TO 2018-01-31]` – From the earliest representable time thru till the end of the day on 2018-01-31.

The data type name is case sensitive. Use single quotes to specify `DateRangeType` in a CQL `CREATE TABLE` statement. For example:

**CREATE TABLE test.example (  
id int PRIMARY KEY,  
daterange 'DateRangeType'  
);
**

This example shows an **INSERT** statement with a date range value:

**INSERT INTO test.example (  
id,  
daterange  
) VALUES (  
1,  
'[2017-02-02T14:57:00 TO 2017-02-02T15:10:17]'  
);
**

duration

Time duration encoded as these signed integers of variable lengths:

1. Months
2. Days

3. Nanoseconds

The number of days in a month is variable. A day can have 23 or 25 hours, depending on the daylight saving time. Internally, the number of months and days are decoded as 32-bit integers; the nanoseconds are decoded as a 64-bit integer.

time

Encoded 64-bit signed integers representing the number of nanoseconds since midnight with no corresponding date value.

INSERT or UPDATE string format is ‘hh:mm:ss[.fff]’, where milliseconds (f) are optional.

timestamp

64-bit signed integer representing the date and time since epoch (January 1 1970 at 00:00:00 GMT) in milliseconds.

INSERT or UPDATE string format is ISO-8601; the string must contain the date and optionally can include the time and time zone, ‘yyyy-mm-dd [hh:mm:ss[.fff]] [+/−NNNN]’ where NNNN is the RFC 822 4-digit time zone specification (+0000 refers to GMT and US PST is -0800). If no time zone is specified, the client timezone is assumed. For example ‘2015-05-03 13:30:54.234-0800’, ‘2015-05-03 13:30:54+0400’, or ‘2015-05-03’.

Unique identifiers

uuid

128 bit universally unique identifier (UUID). Generate with the UUID function.

timeuuid

Version 1 UUID; unique identifier that includes a "conflict-free" timestamp. Generate with the NOW function.

Specialized types

blob

Arbitrary bytes (no validation), expressed as hexadecimal. See Blob conversion functions.

boolean

True or false. Stored internally as true or false; when using the COPY TO in cqlsh to import or export data, change the format using the BOOLSTYLE option, for example when importing survey results that have yes/no style answer column.

counter

64-bit signed integer. Only one counter column is allowed per table. All other columns in a counter table must be PRIMARY KEYS. Increment and decrement the counter with an UPDATE statement using the + and - operators. Null values are not supported in the counter column, the initial count equals 0.

inet

IP address string in IPv4 or IPv6 format.

Collection types

CQL supports storing multiple values in a single column. Use collections to store or denormalize small amounts of data, such as phone numbers, tags, or addresses. Collections are not appropriate for data that is expected to grow unbounded, such as all events for a particular user; instead use a table with clustering columns.

Non-frozen collections have the following characteristics and limitations:

- Because collections are not indexed or paged internally, the entire collection is read in order to access a single element.
- Some operations on lists incur a read-before-write. Also list operations are not idempotent by nature and can cause timeout issues in the case of retries. INSERT on sets and maps never incur a read-before-write internally, therefore DataStax recommends sets over lists whenever possible.

Storing a large amount of data in a single collection is an anti-pattern and therefore not supported.

frozen
Use frozen on a set, map, or list to serialize multiple components into a single value, `frozen<collection_definition>`. Non-frozen types allow updates to individual fields, but values in a frozen collection are treated like blobs, any upsert overwrites the entire value.

**list**

Comma separated list of non-unique values of the same data type, `list<data_type>`. Elements are ordered by their position in the list; the first position is zero. Supports appending and prepending elements in INSERT and UPDATE statements using the + and - operators.

Lists have limitations and performance impact, whenever possible use `set` or a `frozen` list, for example `frozen<list<int>>`. The append and prepend operations are not idempotent. If either of these operations timeout, the retry operation may (or may not) result in appending/prepending the value twice.

**map**

Set of key-value pairs, where keys are unique and the map is sorted by its keys, `map<data_type[, data_type,...]>`.

For **INSERT** and **UPDATE**, setting TTL is only apply to the newly inserted/updated elements.

**set**

Comma separated list of unique values sorted by position starting at zero.

**tuple**

Fixed length set of elements of different types. Unlike other collection types, a tuple is always frozen (without the need of the frozen keyword). The entire field is overwritten when using **INSERT** and **UPDATE**, therefore the expressions must provide a value for each element; explicitly declare null for elements that have no value. Tuples can contain tuples, for example `tuple<int,tuple<string,string>,boolean>` and also be specified as a data type of another collection type, for example `set<tuple<string,inet>,int>`.

**user defined type (UDT)**

Customize collection type that belongs to a specific keyspace. The UDT is only available in the keyspace where it is created. The `system_schema.types` contains a list of all UDT, the `keyspace_name`, `type_name`, `field_names`, and `field_types`.

**Deprecated types**

The following types are supported for backward compatibility only.

**custom type**

Deprecated supported for backward compatibility. Customized type added as a sub-class to `AbstractType`, where the class name is fully qualified or relative to the `org.apache.cassandra.db.marshal` package.

Replaced by `user defined type (UDT)`.

**Date, time, and timestamp format**

To upsert a value into `date`, `time`, or `timestamp` columns, use the string format or the date or conversion functions (TODATE and TOTIMESTAMP) with the NOW function.

Timestamp also supports upserting the value as an integer. The integer is the number of milliseconds after the Unix epoch (January 1, 1970).

Use the formats in **INSERT** and **UPDATE** statements.

**date**

date string format:

```
yyyy-mm-dd
```

Where the elements of a date are separated with a hyphen:

- **yyyy**: four digit year
- **mm**: two digit month
• **dd**: two digit day

For example, May 5, 2017:

```
2017-05-05
```

### time

time string format:

```
HH:MM:SS[.fff]
```

where

• **HH**: two digit hour using a 24-hour clock.
• **MM**: two digit minute.
• **SS**: two digit seconds.
• (Optional) **.fff**: up to three digit sub-seconds; when excluded it’s set to zero (0).

For example, one o’clock in the afternoon

```
13:00:00.000
13:00:00
```

### timestamp

timestamp combines both date and time with the addition of time zone in ISO 8601 format:

```
yyyy-mm-dd[ (T | ) HH:MM:SS[.fff]] [ (+ | - ) NNNN]
```

Where only the date portion of the timestamp is required:

• **date** (*yyyy-mm-dd*), where
  
  # **yyyy**: four digit year
  # **mm**: two digit month
  # **dd**: two digit day

• **time** (*HH:MM:SS[.fff]*) , where
  
  # **HH**: two digit hour using a 24-hour clock.
  # **MM**: two digit minute.
  # **SS**: two digit seconds.
  # (Optional) **.fff**: up to three digit sub-seconds; when excluded it’s set to zero (0).
  
  When time is excluded, it’s set to zero.

• **timezone** (*(+ | - ) NNNN*): is the offset from GMT.
  
  # +|- indicates whether to add or subtract the NNNN from GMT
  # NNNN is the RFC-822 4-digit time zone, for example +0000 is GMT and -0800 is PST.
  
  When timezone is excluded, it’s set to the client or coordinator timezone.
For example, May 5, 2017 midnight GMT:

- 2017-05-05 00:00:00.000+0000
- 2017-05-05 00:00:00.000
- 2017-05-05 00:00:00
- 2017-05-05

This table shows additional examples. The z, X, and Z elements represent timezones.

<table>
<thead>
<tr>
<th>Cassandra 2.2.0 Pattern</th>
<th>Example for UTC Timezone</th>
<th>Example for Europe/Paris Timezone</th>
<th>Example for GMT-07:00 Timezone</th>
</tr>
</thead>
<tbody>
<tr>
<td>yyyy-mm-dd HH:SSS.XX</td>
<td>2018-04-26 12:59:38Z</td>
<td>2018-04-26 14:59:38:02:00:00</td>
<td>2018-04-26 05:59:38:07:00:00</td>
</tr>
<tr>
<td>Format</td>
<td>UTC</td>
<td>CEST</td>
<td>GMT-07:00</td>
</tr>
<tr>
<td>--------</td>
<td>-----</td>
<td>------</td>
<td>-----------</td>
</tr>
</tbody>
</table>

### CQL native functions

CQL supports several functions that transform a column value into a new value.

In addition, users can define functions and aggregates.

The native database functions are:
CAST

Converts the data returned by the selector to a native CQL data type.

```
CAST( selector AS to_type )
```

Cast strictly relies on the Java semantics, for more details on the underlying type see Table 1. Only use in SELECT statements.

Silently ignores casting a column into its own data type.

<table>
<thead>
<tr>
<th>Selector column type</th>
<th>Output data type (to_type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ascii</td>
<td>text, varchar</td>
</tr>
<tr>
<td>bigint</td>
<td>tinyint, smallint, int, varint, float, double, decimal, text, varchar</td>
</tr>
<tr>
<td>boolean</td>
<td>text, varchar</td>
</tr>
<tr>
<td>counter</td>
<td>tinyint, smallint, int, bigint, float, double, decimal, varint, text, varchar</td>
</tr>
<tr>
<td>date</td>
<td>timestamp</td>
</tr>
<tr>
<td>decimal</td>
<td>tinyint, smallint, int, bigint, float, double, decimal, varint, text, varchar</td>
</tr>
<tr>
<td>double</td>
<td>tinyint, smallint, int, bigint, float, decimal, varint, text, varchar</td>
</tr>
<tr>
<td>float</td>
<td>tinyint, smallint, int, bigint, double, decimal, varint, text, varchar</td>
</tr>
<tr>
<td>inet</td>
<td>text, varchar</td>
</tr>
<tr>
<td>int</td>
<td>tinyint, smallint, bigint, float, double, decimal, varint, text, varchar</td>
</tr>
<tr>
<td>smallint</td>
<td>tinyint, int, bigint, float, double, decimal, varint, text, varchar</td>
</tr>
<tr>
<td>time</td>
<td>text, varchar</td>
</tr>
<tr>
<td>timestamp</td>
<td>date, text, varchar</td>
</tr>
<tr>
<td>timeuuid</td>
<td>timestamp, date, text, varchar</td>
</tr>
<tr>
<td>tinyint</td>
<td>tinyint, smallint, int, bigint, float, double, decimal, varint, text, varchar</td>
</tr>
<tr>
<td>uuid</td>
<td>text, varchar</td>
</tr>
<tr>
<td>varint</td>
<td>tinyint, smallint, int, bigint, float, double, decimal, text, varchar</td>
</tr>
</tbody>
</table>

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: CQL types Java derivation**

<table>
<thead>
<tr>
<th>CQL type</th>
<th>Java type</th>
</tr>
</thead>
<tbody>
<tr>
<td>decimal</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>float</td>
<td>java.lang.Float</td>
</tr>
<tr>
<td>double</td>
<td>java.lang.Double</td>
</tr>
<tr>
<td>varint</td>
<td>java.math.BigInteger</td>
</tr>
</tbody>
</table>

**BLOBAsto_type**

Converts the target column or literal (enclose strings in single quotes) from a blob to the specified type.

```
blobAsto_type(column_name | literal)
```

- Display blob columns as another data type in results of SELECT statements.
• Convert raw blob data into another type for storage **INSERT** and **UPDATE**.

**typeASBlob**

Converts the target column or literal (enclose strings in single quotes) to a blob from the specified type, where the from type corresponds to a valid **CQL data type**.

```
typeAsBlob(column_name | literal)
```

Use in the following types of CQL statements:

• **SELECT** selectors to return a value stored in another CQL type as a blob.
• **INSERT** and **UPDATE** convert another CQL data type into a blob for storage.

**TOKEN**

Computes a token for values in the target column.

```
TOKEN(column_name)
```

The exact signature of the token function depends on the table concerned and of the partitioner used by the cluster.

• **Murmur3Partitioner** type is `bigint`
• **RandomPartitioner** type is `varint`
• **ByteOrderedPartitioner** type is `blob`

Use in the following CQL requests:

• **SELECT** selector clause to return values in the target column as tokens; useful for manually paging through the data or to determine the physical location of a row or set of rows.
• **SELECT** where clause to return a set of records in a given token range. Allows comparison (greater than and less than) in the where clause without the use of ALLOW FILTERING. Useful when dividing query workloads between clients, such as multiple Spark clients for analytics.

**UPDATE** does not support the token function in the where clause.

**minTimeuuid**

Computes the smallest fake timeuuid from the specified date (`'yyyy-mm-dd'`) or timestamp (`'yyyy-mm-dd [hh:mm:ss[.fff][+-NNNN]]'`) formatted string.

```
minTimeuuid(date_string | timestamp_string)
```

Use in a **SELECT** statement **WHERE** clause to return rows in a date range from a `timeuuid` column, for example `WHERE id >= minTimeuuid('2017-05-01')`.

**maxTimeuuid**

Computes the largest fake timeuuid from the specified date (`'yyyy-mm-dd'`) or timestamp (`'yyyy-mm-dd [hh:mm:ss[.fff][+-NNNN]]'`) formatted string.

```
MAXTIMEUUID(date_string | timestamp_string)
```

Use in a **SELECT** statement **where** clause to return rows in a date range from a `timeuuid` column, for example `WHERE id <= maxTimeuuid('2017-05-01')`.

**NOW**
CQL reference

Takes no arguments and generates, on the coordinator node, a new unique `timeuuid` at the time when the statement executed.

```
NOW()
```

Use in `INSERT` or `UPDATE` statements to generate a new `timeuuid` value.

Although allowed in `WHERE` clauses, no values would match a given `SELECT` or `UPDATE` statement because `now()` is guaranteed to be unique.

`UPDATE` statements SET clause cannot be used to change PRIMARY KEY fields; therefore a new `timeuuid` can only be set if the target field is not part of the PRIMARY KEY field.

**TODATE**

Converts a timestamp value from milliseconds to days (since epoch) and for timeuuid extracts the timestamp and converts it to days since epoch.

```
TODATE(column_name)
```

Dates and times display in string format, but are stored and compared as integers.

**TOTIMESTAMP**

Converts a date value from days into milliseconds since epoch and for `timeuuid` extracts the timestamp.

```
TOTIMESTAMP(column_name)
```

Time (hh:mm:ss.ffffff+NNNN) of a date are all set to zero in the resulting timestamp.

**TOUNIXTIMESTAMP**

Converts the `timeuuid`, `timestamp`, or `date` column into the number of seconds since epoch.

```
TOUNIXTIMESTAMP(column_name)
```

**UUID**

Takes no parameters and generates a random type 4 uuid.

```
UUID()
```

Use to generate a value in `INSERT` and `UPDATE`.

`UPDATE` statements SET clause cannot be used to change PRIMARY KEY fields; therefore a new `UUID` can only be set if the target field is not part of the PRIMARY KEY field.

**WRITETIME**

Shows the unix timestamp (number of seconds since epoch) that the data was written.

```
WRITETIME(column_name)
```

Use only in `selectors` clause of `SELECT` statements on non-partition key columns.

**TTL**

Count down in seconds until the value in the field expires and the data is automatically removed; null indicates that TTL for the column is not set (or the count down has ended).

```
TTL(column_name)
```

Use only in `selectors` clause of `SELECT` statements on non-partition key columns.
**Deprecated functions**
The following functions are supported for backward compatibility only.

**DATEOF**
Similar to \texttt{TOTIMESTAMP(timeuuid)}.

**UNIXTIMESTAMPOF**
Similar to \texttt{TOUNIXTIMESTAMP(timeuuid)}.

---

**CQL native aggregates**
Aggregate functions work on a set of rows matching a \texttt{SELECT} statement to return a single value. Null values in the data set are ignored. When other columns, columns with user-defined types, or functions, are also specified in the selector clause of a \texttt{SELECT} statement with an aggregate function, the values in the first row matching the query are returned.

Aggregating data across partitions may cause performance issues. DataStax recommends using aggregates on one partition, to get a list of partition keys use \texttt{SELECT DISTINCT partition\_key} to get a list of unique keys (for compound keys use a comma separated list of column names).

Define custom aggregates using \texttt{CREATE AGGREGATE}.

**Synopsis**

<table>
<thead>
<tr>
<th>aggregate_name(column_name)</th>
</tr>
</thead>
</table>

Where the system aggregate names are listed below:

**AVG**
\texttt{AVG(column\_name)}: Provides the average value of the target column using the data type of target column. Null values are ignored.

Only works on numeric columns, that is \texttt{tinyint}, \texttt{smallint}, \texttt{int}, \texttt{bigint}, \texttt{decimal}, \texttt{float}, and \texttt{double}.

**COUNT**
\texttt{COUNT(column\_name)}: Provides the number of rows in the result set that do not have null values. For a complete row count use partition key column, since partition keys cannot contain nulls.

Works on all \texttt{CQL data types} columns.

**MIN**
\texttt{MIN(column\_name)}: Provides the smallest value. Null values are ignored.

Only works numeric columns, that is \texttt{tinyint}, \texttt{smallint}, \texttt{int}, \texttt{bigint}, \texttt{decimal}, \texttt{float}, and \texttt{double}.

**MAX**
\texttt{MAX(column\_name)}: Provides the largest value. Null values are ignored.

Only works numeric columns, that is \texttt{tinyint}, \texttt{smallint}, \texttt{int}, \texttt{bigint}, \texttt{decimal}, \texttt{float}, and \texttt{double}.

**SUM**
\texttt{SUM(column\_name)}: Provides the total of the target column; nulls are ignored.

Only works on numeric columns, that is \texttt{tinyint}, \texttt{smallint}, \texttt{int}, \texttt{bigint}, \texttt{decimal}, \texttt{float}, and \texttt{double}.

**Examples**
The following examples show how to use aggregates using the \texttt{cyclist examples}.

**Find Average (AVG)**

Get average time in seconds for the team:

\begin{verbatim}
SELECT AVG(cyclist\_time\_sec) \ AS \ Average \ FROM \ cycling\_team\_average
\end{verbatim}
WHERE team_name = 'UnitedHealthCare Pro Cycling Womens Team';

Results:

<table>
<thead>
<tr>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>11474</td>
</tr>
</tbody>
</table>

Find Count (COUNT)

Find the number rows for the United Health Care Pro Cycling Women's Team:

```
SELECT count(cyclist_name) AS Row_Count FROM cycling.team_average
WHERE team_name = 'UnitedHealthCare Pro Cycling Womens Team';
```

Results:

<table>
<thead>
<tr>
<th>row_count</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

Find lowest value (MIN)

Find the slowest time recorded for the United Health Care Pro Cycling Women's Team:

```
SELECT MIN(cyclist_time_sec) AS Fastest FROM cycling.team_average
WHERE team_name = 'UnitedHealthCare Pro Cycling Womens Team';
```

Results:

<table>
<thead>
<tr>
<th>fastest</th>
</tr>
</thead>
<tbody>
<tr>
<td>11449</td>
</tr>
</tbody>
</table>

Find highest value (MAX)

Find the fastest time recorded for the United Health Care Pro Cycling Women's Team:

```
SELECT MAX(cyclist_time_sec) AS Slowest FROM cycling.team_average
WHERE team_name = 'UnitedHealthCare Pro Cycling Womens Team';
```

Results:

<table>
<thead>
<tr>
<th>slowest</th>
</tr>
</thead>
<tbody>
<tr>
<td>11490</td>
</tr>
</tbody>
</table>
Find total (SUM)

Find the total of all times recorded for the United Health Care Pro Cycling Women's Team:

```
SELECT SUM(cyclist_time_sec) AS Total_Time FROM cycling.team_average
WHERE team_name = 'UnitedHealthCare Pro Cycling Womens Team';
```

Results:

<table>
<thead>
<tr>
<th>total_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>34424</td>
</tr>
</tbody>
</table>

CQL shell (cqlsh) reference

Commands specific to the Cassandra Query Language shell (cqlsh) utility.

In cqlsh, type `help` to list all available topics. Type `help name` to find out more about the `name` command. For example, `help CAPTURE` or `help ALTER_KEYSPACE`.

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 host. For configuration information, see the `cassandra.yaml` file.

See Accessing data using CQL.

cqlsh (startup options)

Execute the `cqlsh` python script to start the CQL shell. The CQL shell is a python-based command line client for executing CQL commands interactively. The CQL shell supports tab completion.

Synopsis

```
$ cqlsh [ options ] [ host_name[:port_number] ]
```

Table 9: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italic</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[]</td>
<td>Optional. Square brackets ([] ) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>()</td>
<td>Group. Parentheses (( )) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis (…) indicates that you can repeat the syntax element as often as required.</td>
</tr>
</tbody>
</table>
Syntax conventions | Description
--- | ---
'literal string' | Single quotation (‘’) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
{} | 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.
--browser="launched_browser_cmd %s" | 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.
-C, --color | Always use color output.
--connect-timeout="timeout" | Connection timeout in seconds. Default: 5.
--cqlshrc="/folder_name" | Folder that contains the cqlshrc file. Use tilde (~) for paths relative to the user's home directory.
--cqlversion="version_number" | CQL version to use. The CQL version displays after starting cqlsh.
--debug | Show additional debugging information.
--dse-protocol-version=PROTOCOL_VERSION | Specify a specific DSE protocol version; otherwise the client will default and downgrade as necessary. Mutually exclusive with --protocol-version.
--encoding="output_encoding" | Output encoding. Default encoding: utf8.
--execute="cql_statement" | Execute the CQL statement, then exit. To direct the command output to a file, see saving CQL output.
-f file_name, --file=file_name | Execute commands from a CQL file, then exit. After starting cqlsh, use the SOURCE command and the path to the file using the cqlsh command line.
-h, --help | Show help.
host_name:port | To connect the CQL session to a specified node, specify a hostname or IP address and optional port after the cqlsh command, along with any additional CQL shell options. By default, the CQL shell launches a session with the local host on address 127.0.0.1. You can connect the CQL shell to remote hosts that have a higher or equal CQL shell version than the local CQL shell version. When no port is specified, the connection uses the default port of 9042.
-k keyspace_name, --keyspace=keyspace_name | Automatically use the specified keyspace after starting the CQL shell.
--no-color | Do not display color output.
-p password, --password="password" | Connect with the specified user's password.
--protocol-version=PROTOCOL_VERSION |
Specify a specific protocol version. If omitted, the client will use a default, and possibly lower version protocol, as needed. Mutually exclusive with --dse-protocol-version.

--request-timeout="timeout"
CQL request timeout in seconds. Default: 10.

--ssl
Use SSL.

-t, --tty
Force TTY command prompt mode.

-U user_name, --username="user_name"
Connect with the specified user account.

--version
Show the cqlsh version number.

Environment variables
You can use environment variables to overwrite default values for cqlsh commands. For example, increase the timeout values of a user running cqlsh on a particular computer.

CQLSH_SEARCH_MANAGEMENT_TIMEOUT_SECONDS
Overwrite the default 600 seconds (10 minutes) request timeout for search-specific CQL statements. To prevent timeouts, increase the timeout value. Typical use case is to ensure that no timeouts occur when large indexes are reloaded.

The timeout applies only to these search CQL index management commands:

- ALTER SEARCH INDEX
- COMMIT SEARCH INDEX
- CREATE SEARCH INDEX
- DESCRIBE SEARCH INDEX
- DROP SEARCH INDEX
- RELOAD SEARCH INDEX
- REBUILD SEARCH INDEX

The timeout is used only if the cqlsh request timeout is equal to the default value of 10 seconds:

$ cqlsh --request-timeout 10

To increase the timeout request timeout for search-specific CQL statements to 15 minutes (900 seconds):

$ export CQLSH_SEARCH_MANAGEMENT_TIMEOUT_SECONDS=900;

Examples
Starting the CQL shell
On start up, cqlsh shows the name of the cluster, IP address, and connection port. The cqlsh prompt initially is cqlsh>. If you specify a keyspace, it is added after the prompt.

1. Start the CQL shell:

   $ cqlsh

   The cluster and host information appears. For example:

   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]

CQL for the DataStax Distribution of Apache Cassandra™ 3.11 Latest DDAC patch: 5.1.17
Use HELP for help.

2. Use the cycling keyspace:

```
USE cycling;
```

The prompt now includes the keyspace name:

```
cqlsh:cycling>
```

Querying using CQL commands

At the cqlsh prompt, you can enter CQL commands. Use a semicolon to terminate a command. A new line does not terminate a command, and commands can be spread over several lines. For example:

```
SELECT *
FROM calendar
WHERE race_id = 201;
```

The returned results are shown in the standard output:

```
race_id | race_start_date                  | race_end_date                  | race_name
---------+---------------------------------+---------------------------------+-------------------------------
+---------------------------------+---------------------------------+-------------------------------
 201    | 2015-02-18 08:00:00.000000+0000 | 2015-02-22 08:00:00.000000+0000 | Women's Tour of New Zealand
```

Understanding the CQL command syntax describes:

- How upper- and lower-case literals are treated in commands.
- When to use quotation marks in strings.
- How to enter exponential notation.

Saving CQL output in a file

To save output from a CQL statement to a file, use the cqlsh -e option, followed by the CQL statement placed inside quotation marks, and redirect the output to a file. For example, to save the output of a SELECT statement to myoutput.txt:

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

Connecting to a remote node

Specify a remote node IP address:

```
$ 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]
Configuring a cqlshrc file

A cqlshrc file contains CQL shell session settings that are used when the CQL shell starts. If you place a cqlshrc file in the ~/.cassandra directory on a local computer, that file is used by default. You can also specify the directory that the cqlshrc file is in using the --cqlshrc option. Place only one cqlshrc file in a directory.

Do not confuse the cqlshrc file with the --cqlshrc option. The first is a file containing CQL session settings, the second is an option that specifies the directory where the cqlshrc file is located.

You can 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"
```

Tilde (~) expands to the user's home directory; you can also specify the absolute path, for example /Users/jdoe/cqlshprofiles/west.

Table 10: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td><em>italics</em></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[]</td>
<td>Optional. Square brackets {[]} surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>( )</td>
<td>Group. Parentheses {( )} identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.</td>
</tr>
</tbody>
</table>
Using a sample cqlshrc file

A sample file is installed with DataStax Distribution of Apache Cassandra™ (DDAC) cqlshrc sample files. These files contain all the available settings. Some settings are commented out using a single semi-colon.

DDAC provides sample cqlshrc files:

• cqlshrc.sample: Contains all available settings.

This file is located in the installation_location/conf directory.

Make changes as appropriate for your environment.

To use the sample file:

1. Copy the cqlshrc.sample file to the ~/.cassandra/ directory.

2. Rename the file to cqlshrc. The file should be located in ~/.cassandra/cqlshrc.
   
   If cqlsh finds the .cqlshrc file located in the home directory, cqlsh moves the file to ~/.cassandra/cqlshrc upon its next invocation and outputs a message that the file moved.

3. Remove the semi-colon to uncomment an option (options must be in brackets) and its corresponding settings. The example below uncomments [copy] and header = false to import all CSV without a header row:

   ;; 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 optionally choose a keyspace.

Only set a user name and password for hosts that use internal authentication, see Encrypting 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 authentication.)

Changing the CQL shell display
The cqlsh console display and COPY TO date parsing settings.

[ui]
color
    Shows query results with color.
        on - 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.
        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.

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]
**CQL reference**

**max_trace_wait**
- The maximum number of seconds to wait for a trace to complete.

**Configuring SSL**
Specify connection SSL settings.

For more information, see Encrypting with SSL.

**[ssl]**

certfile
- The path to the Cassandra certificate.

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 the `[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.

```
[copy]
BOOLSTYLE       = True,False
CONFIGFILE      = /home/
DATETIMEFORMAT  = %m/%d/%Y
DECIMALSEP      = ,
DELIMITER       = |,
ESCAPE          = `\
HEADER          = true
MAXATTEMPTS     = 10
NULL            = NULL
NUMPROCESSES    = -1
QUOTE           = "
REPORTFREQUENCY = 1
RATEFILE        = /home/copyrate.log
SKIPCOLUMNS     = firstname
SKIPIROWS       = 6
THOUSANDSSEP    = ,
```

**BOOLSTYLE**
- Boolean indicators for true and false. The values are case-insensitive. For example, yes, no and YES, NO are the same.
- Default: True, False

**CONFIGFILE**
- Specify a directory that contains the `cqlshrc` configuration file.
  - Command line options always override the `cqlshrc` file.

**DATETIMEFORMAT**
- Time format for reading or writing CSV time data. The timestamp uses the `strftime` format. If not set, the default value is set to the `datetimeformat` value in the `cqlshrc` file.
- Default: %Y-%m-%d %H:%M:%S

**DECIMALSEP**
- Decimal value separator.
- Default: . (period)
**DELIMITER**
- Field separator.
- Default: `,` (comma)

**ESCAPE**
- Single character that escapes literal uses of the QUOTE character.
- Default: `\` (backslash)

**HEADER**
- `true` - first row contains headers (column names)
- `false` - first row does not have headers
- Default: `false`

**MAXATTEMPTS**
- Maximum number of attempts for errors.
- Default: `5`

**NULL**
- Value used when no value is in the field.
- Default: `<empty>`

**NUMPROCESSES**
- Number of worker processes. Maximum value is 16.
- Default: `-1`

**QUOTE**
- Encloses field values.
- Default: `"` (double quotation mark)

**REPORTFREQUENCY**
- Frequency with which status is displayed in seconds.
- Default: `0.25`

**RATEFILE**
- Print output statistics to this file.

**SKIPCOLS**
- Name of column to skip.

**SKIPROWS**
- Number of rows starting from the first row of data to skip.

**THOUSANDSSEP**
- Separator for thousands digit groups.

**Setting COPY TO specific options**
Under the `copy-to` section set any of the export options.

**BEGIN_TOKEN**
- Minimum token string for exporting data.

**DOUBLEPRECISION**
- Number of digits to display after the decimal point for CQL double precision values.
- Default: `12`

**ENCODING**
- Output string type.
- Default: `UTF8`

**END_TOKEN**
- Maximum token string for exporting data.

**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.

**FLOATPRECISION**
- Number of digits to display after the decimal point for CQL float (single precision) values.
- Default: `5`

**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: `-1`

**MAXREQUESTS**
CQL reference

Maximum number of requests each worker can process in parallel.
Default: 6

**PAGESIZE**
Page size for fetching results.
Default: 1000

**PAGETIMEOUT**
Page timeout for fetching results.
Default: 10

**TTL**
Time to live in seconds. By default, data will not expire.
Default: 3600

Setting COPY FROM specific options

Under the `[copy-from]` section set any of the import options.

**CHUNKSIZE**
Chunk size passed to worker processes.
Default: 1000

**INGESTRATE**
Approximate ingest rate in rows per second. Must be greater than the chunk size.
Default: 100000

**MAXBATCHSIZE**
Maximum size of an import batch.
Default: 20

**MAXINSERTERRORS**
Maximum global number of insert errors. Use -1 for no maximum.
Default: -1

**MAXPARSEERRORS**
Maximum global number of parsing errors. Use -1 for no maximum.
Default: -1

**MAXROWS**
Maximum number of rows. Use -1 for no maximum.
Default: -1

**MINBATCHSIZE**
Minimum size of an import batch.
Default: 2

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:

```
[copystart]cycling.cyclist_names
chunksize = 1000
[copystart]cycling.rank_by_year_and_name
chunksize = 10000
```

```
[copystart:keystorage_name.table_name]
chunksize
Chunk size passed to worker processes.
[copyfromstart:keystorage_name.table_name]
ingestrate
Approximate ingest rate in rows per second. Must be greater than the chunk size.
[copytostart:keystorage_name.table_name]
pagetimeout
Page timeout for fetching results.
```
CAPTURE
Captures CQL query output and appends the output to a specified file. Error messages and cqlsh commands are displayed in the standard terminal output. All other cqlsh output is appended to the file. To display the current capture status, enter CAPTURE with no options.

Synopsis

CAPTURE [ 'file_name' | OFF ]

Table 11: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italic</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>{}</td>
<td>Optional. Square brackets ( {}) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>()</td>
<td>Group. Parentheses ( () ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation ( ' ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>[--]</td>
<td>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.</td>
</tr>
<tr>
<td>' '&lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ' ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

file_name
The file to write the query output to. When you run the first query after starting 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 your home directory. Absolute paths are not supported.

OFF
Stops capture.
CQL reference

Examples
Capture results to the winners text file:

\[ \text{CAPTURE '~/results/winners.txt'} \]

The results directory must exist in your home directory. The winners text file is created if it does not exist.

\[ \text{Now capturing query output to '/Users/local_system_user/results/winners.txt'.} \]

Execute a query that selects all cycling race winners:

\[ \text{SELECT *} \]
\[ \text{FROM cycling.race_winners;} \]

Results are appended to the end of the capture file. The results are not displayed in the terminal.

CLEAR
Clears the CQL shell terminal window. The keyboard shortcut is CTRL+I.

Synopsis
CLEAR

or

CLS

Table 12: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italic</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets ([ ]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>{ }</td>
<td>Group. Parentheses ( { } ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'Literal string'</td>
<td>Single quotation ( ' ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon (;) terminates all CQL statements.</td>
</tr>
<tr>
<td>Syntax conventions</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>[--]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks (') surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

There are no parameters for this command.

**Example**

Clear the terminal window:

```
CLEAR
```

**CONSISTENCY**

Sets and displays the consistency level. The consistency level determines the number of replica nodes that must respond for the coordinator node to successfully process a non-lightweight transaction (non-LWT).

The CQL shell supports only read requests (SELECT statements) when the consistency level is set to SERIAL or LOCAL_SERIAL. See Data consistency in the documentation.

To set the consistency level of a lightweight transaction (LWT), use the SERIAL CONSISTENCY command. When using a LWT, you must have both a CONSISTENCY and a SERIAL CONSISTENCY level set. CONSISTENCY cannot be set to SERIAL or LOCAL_SERIAL, only SERIAL CONSISTENCY can be set to SERIAL or LOCAL_SERIAL.

**Synopsis**

```
CONSISTENCY [ consistency_level ]
```

**Table 13: Legend**

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td><em>italics</em></td>
<td>Optional. Square brackets ([ ]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Group. Parentheses (( )) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td>( )</td>
<td>Or. A vertical bar (</td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation (') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets (&lt; &gt;) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
</tbody>
</table>
Syntax conventions

| 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. |

**consistency_level**

The consistency level. See the documentation for the list of valid levels and their description, which is extensive.

**Displaying the current consistency level**

To show the current consistency level, use the CONSISTENCY command with no options.

```plaintext
CONSISTENCY
Current consistency level is ONE.
```

The default consistency level is ONE.

**Setting a consistency level**

The consistency 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 overrides the consistency-level global setting.

Before changing this setting it is important to understand these topics in the documentation: How the database reads and writes data, Data replication, How QUORUM is calculated, and partition keys. See the documentation for the links.

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 14: Read Consistency Levels**

<table>
<thead>
<tr>
<th>Level</th>
<th>Replicas</th>
<th>Consistency</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>All</td>
<td>Highest</td>
<td>Lowest</td>
</tr>
<tr>
<td>EACH_QUORUM</td>
<td>Quorum in each datacenter.</td>
<td>Same across datacenters</td>
<td></td>
</tr>
<tr>
<td>QUORUM</td>
<td>Quorum of all nodes across all datacenters. Some level of failure is possible.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOCAL_QUORUM</td>
<td>Quorum of replicas in the same datacenter as the coordinator node. Avoids communication latency between datacenters.</td>
<td>Low in multi-datacenter</td>
<td></td>
</tr>
<tr>
<td>ONE</td>
<td>Closest replica as determined by the snitch. Satisfies the needs of most users because the consistency requirements are not stringent.</td>
<td>Lowest (READ)</td>
<td>Highest (READ)</td>
</tr>
<tr>
<td>TWO</td>
<td>Closest two replicas as determined by the snitch.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>THREE</td>
<td>Closest three replicas as determined by the snitch.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOCAL_ONE</td>
<td>Returns a response from the closest replica in the local datacenter. For security and quality, use in an offline datacenter to prevent automatic connection to online nodes in other datacenters.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The SERIAL and LOCAL_SERIAL consistency levels support read transactions. Those levels do not support writes and generate an error; see the INSERT and UPDATE examples at the end of this section. Use the ANY consistency level if possible.

Examples

Set the `CONSISTENCY QUORUM` level to force the majority of the nodes to respond:

```sql
CONSISTENCY QUORUM
```

Set the level to serial for LWT read requests:

```sql
CONSISTENCY SERIAL
```

Consistency level set to SERIAL.

Query that returns all cycling race winners:

```sql
SELECT * FROM cycling.race_winners;
```

The query results are as follows:

<table>
<thead>
<tr>
<th>Row</th>
<th>Race Name</th>
<th>Position</th>
<th>Cyclist Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>National Championships South Africa WJ-ITT (CN)</td>
<td>1</td>
<td>{firstname: 'Frances', lastname: 'DU TOUT'}</td>
</tr>
<tr>
<td>2</td>
<td>National Championships South Africa WJ-ITT (CN)</td>
<td>2</td>
<td>{firstname: 'Lynette', lastname: 'BENSON'}</td>
</tr>
<tr>
<td>3</td>
<td>National Championships South Africa WJ-ITT (CN)</td>
<td>3</td>
<td>{firstname: 'Anja', lastname: 'GERBER'}</td>
</tr>
<tr>
<td>4</td>
<td>National Championships South Africa WJ-ITT (CN)</td>
<td>4</td>
<td>{firstname: 'Ame', lastname: 'VENTER'}</td>
</tr>
<tr>
<td>race_name</td>
<td>National Championships South Africa WJ-ITT (CN)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>race_position</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cyclist_name</td>
<td>{firstname: 'Danielle', lastname: 'VAN NIEKERK'}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(5 rows)

The previous query format uses `expand ON` for legibility.

A LWT is a write request that contains `IF EXISTS` or `IF NOT EXISTS` statements. To set the consistency level of LWTs, see `SERIAL CONSISTENCY`. When using a LWT, you must have both a `CONSISTENCY` and a `SERIAL CONSISTENCY` level set. `CONSISTENCY` cannot be set to `SERIAL` or `LOCAL_SERIAL`, only `SERIAL CONSISTENCY` can be set to `SERIAL` or `LOCAL_SERIAL`.

The following examples show LWT failure scenarios. Inserts for a LWT with `CONSISTENCY SERIAL` fail:

```
CONSISTENCY SERIAL

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="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 for a LWT with `CONSISTENCY SERIAL` also fail:

```
CONSISTENCY SERIAL

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="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"

Omitting the IF clause generates errors. Also, using `CONSISTENCY LOCAL_SERIAL` generates the errors as that consistency level results in an invalid request.

**COPY TO**

Exports data from a table to a comma-separated values (CSV) file or delimited text file. Each row is written to a line in the target file with the fields separated by the delimiter.
Synopsis

COPY `table_name` [ ( `column_list` ) ]
TO `file_name` [ , `file2_name` , ... ] | STDOUT
[ WITH option = 'value' [ AND ... ] ]

COPY supports a list of one or more comma-separated file names or python glob expressions.

Table 15: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td><code>**italics**</code></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets { [ ] } surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>( )</td>
<td>Group. Parentheses ( { ( ) } ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td><code>**literal string**</code></td>
<td>Single quotation ( ' ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td><code>&lt;datatype1,datatype2&gt;</code></td>
<td>Set, list, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td><code>cql_statement;</code></td>
<td>End CQL statement. A semicolon (;) terminates all CQL statements.</td>
</tr>
<tr>
<td>[--]</td>
<td>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.</td>
</tr>
<tr>
<td><code>'&lt;schema&gt; ... &lt;/schema&gt;'</code></td>
<td>Search CQL only: Single quotation marks ( ' ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td><code>@xml_entity='xml_entity_type'</code></td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

Setting copy options

Copy options set in the COPY statement take precedence over the `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.

**BOOLSTYLE**

Boolean indicators for true and false. The values are case-insensitive. For example, yes,no and YES,NO are the same.

Default: True,False

**CONFIGFILE**

Specify a directory that contains the `cqlshrc` configuration file.

Command line options always override the `cqlshrc` file.

**DATETIMEFORMAT**

Time format for reading or writing CSV time data. The timestamp uses the `strftime` format. If not set, the default value is set to the `datetimeformat` value in the `cqlshrc` file.

Default: %Y-%m-%d %H:%M:%S%z
DECIMALSEP
Decimal value separator.
Default: . (period)

DELIMITER
Field separator.
Default: , (comma)

ESCAPE
Single character that escapes literal uses of the QUOTE character.
Default: \ (backslash)

HEADER
- true - first row contains headers (column names)
- false - first row does not have headers
Default: false

MAXATTEMPTS
Maximum number of attempts for errors.
Default: 5

NULL
Value used when no value is in the field.
Default: <empty>

NUMPROCESSES
Number of worker processes. Maximum value is 16.
Default: -1

QUOTE
Encloses field values.
Default: " (double quotation mark)

REPORTFREQUENCY
Frequency with which status is displayed in seconds.
Default: 0.25

RATEFILE
Print output statistics to this file.

SKIPCOLS
Name of column to skip.

SKIPROWS
Number of rows starting from the first row of data to skip.

THOUSANDSSEP
Separator for thousands digit groups.

BEGIN_TOKEN
Minimum token string for exporting data.

DOUBLEPRECISION
Number of digits to display after the decimal point for CQL double precision values.
Default: 12

ENCODING
Output string type.
Default: UTF8

END_TOKEN
Maximum token string for exporting data.

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.

FLOATPRECISION
Number of digits to display after the decimal point for CQL float (single precision) values.
Default: 5

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.
Default: -1

MAXREQUESTS
Maximum number of requests each worker can process in parallel.
Default: 6

PAGESIZE
Page size for fetching results.
Default: 1000

PAGETIMEOUT
Page timeout for fetching results.
Default: 10

TTL
Time to live in seconds. By default, data will not expire.
Default: 3600

Examples
Create the sample dataset
Set up the environment used for the COPY command examples:

1. Using CQL, create a cycling keyspace:

```sql
CREATE KEYSPACE cycling
WITH REPLICAION = {
  'class': 'NetworkTopologyStrategy',
  'datacenter1' : 1
};
```

2. Create the cycling.cyclist_name table:

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

3. Insert data into cycling.cyclist_name:

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

```sql
INSERT INTO cycling.cyclist_name (id, lastname, firstname)
VALUES (e7cd5752-bc0d-4157-a80f-7523add8dbcd, 'VAN DER BREGGEN','Anna');</p>

```sql
INSERT INTO cycling.cyclist_name (id, lastname, firstname)
VALUES (e7ae5cf3-d358-4d99-b900-85902fda9bb0, 'FRAME','Alex');</p>

```sql
INSERT INTO cycling.cyclist_name (id, lastname, firstname)
VALUES (220844bf-4860-49d6-9a4b-6b5d3a79c6bf, 'TIRALONGO','Paolo');</p>

```sql
INSERT INTO cycling.cyclist_name (id, lastname, firstname)
VALUES (6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47, 'KRUIKSWIJK','Steven');</p>

```sql
INSERT INTO cycling.cyclist_name (id, lastname, firstname)
VALUES (fb372533-eb95-4bb4-8685-6ef61e994caa, 'MATTHEWS', 'Michael');</p>

Export data from the cyclist_name table

1. Export only the id and lastname columns from the cyclist_name table to a CSV file:

```sql
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 (indicated by ../). If the CSV file already exists, it is overwritten. If you do not have permission to create the file in the directory, you can use a different directory; for example, to use the current working directory, omit the directory path before the file name.

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 firstname to a different CSV file named cyclist_firstname.csv:

COPY cycling.cyclist_name (id,firstname)
TO '../cyclist_firstname.csv' WITH HEADER = TRUE;

The CSV 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:

<table>
<thead>
<tr>
<th>id</th>
<th>firstname</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(0 rows)

5. Import the cyclist first names:

COPY cycling.cyclist_name (id,firstname)
FROM '../cyclist_firstname.csv' WITH HEADER = TRUE;

The rows are imported:

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. Verify the new rows:

```sql
SELECT *
FROM cycling.cyclist_name;
```

The rows were created with null last names because the `lastname` field was not in the imported data set:

<table>
<thead>
<tr>
<th>id</th>
<th>firstname</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>e7ae5cf3-d358-4d99-b900-85902fda9bb0</td>
<td>Alex</td>
<td>null</td>
</tr>
<tr>
<td>fb372533-eb95-4bb4-8685-6ef61e994caa</td>
<td>Michael</td>
<td>null</td>
</tr>
<tr>
<td>5b6962dd-3f90-4c93-8f61-eabfa4a803e2</td>
<td>Marianne</td>
<td>null</td>
</tr>
<tr>
<td>22084bf-4860-49d6-9a4b-6b5d3a79c9fbf</td>
<td>Paolo</td>
<td>null</td>
</tr>
<tr>
<td>6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47</td>
<td>Steven</td>
<td>null</td>
</tr>
<tr>
<td>e7cd5752-bc0d-4157-a80f-7523add8dbc</td>
<td>Anna</td>
<td>null</td>
</tr>
</tbody>
</table>

(6 rows)

7. Import the last names:

```sql
COPY cycling.cyclist_name (id, lastname)
FROM '../cyclist_lastname.csv' WITH HEADER = TRUE;
```

The records are imported but no new records are 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 that the records were updated:

```sql
SELECT *
FROM cycling.cyclist_name;
```

The PRIMARY KEY `id` matched for all records and the `lastname` is populated:

<table>
<thead>
<tr>
<th>id</th>
<th>firstname</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>e7ae5cf3-d358-4d99-b900-85902fda9bb0</td>
<td>Alex</td>
<td>FRAME</td>
</tr>
<tr>
<td>fb372533-eb95-4bb4-8685-6ef61e994caa</td>
<td>Michael</td>
<td>MATTHEWS</td>
</tr>
<tr>
<td>5b6962dd-3f90-4c93-8f61-eabfa4a803e2</td>
<td>Marianne</td>
<td>VOS</td>
</tr>
<tr>
<td>22084bf-4860-49d6-9a4b-6b5d3a79c9fbf</td>
<td>Paolo</td>
<td>TIRALONGO</td>
</tr>
<tr>
<td>6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47</td>
<td>Steven</td>
<td>KRIUKSNIJK</td>
</tr>
<tr>
<td>e7cd5752-bc0d-4157-a80f-7523add8dbc</td>
<td>Anna</td>
<td>VAN DER BREGGEN</td>
</tr>
</tbody>
</table>

To copy data from standard input to a table:
1. Clear the data from the `cyclist_name` table:

   ```sql
   TRUNCATE cycling.cyclist_name;
   ```

2. Start the copy input operation using the `FROM STDIN` option:

   ```sql
   COPY cycling.cyclist_name FROM STDIN;
   ```

   The line prompt changes to `[copy]`:

   ```plaintext
   Using 7 child processes
   Starting copy of cycling.cyclist_name with columns [id, firstname, lastname].
   [Use . on a line by itself to end input]
   ```

3. Next to the `[copy]` prompt, enter the field values in a common-separated list; on the last line of data, enter a period:

   ```plaintext
   [copy] e7cd5752-bc0d-4157-a80f-7523add8dbcd,Anna,VAN DER BREGGEN
   [copy] .
   ```

4. Press Enter after the period:

   ```plaintext
   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. Run this query to view the contents of the `cyclist_name` table:

   ```sql
   SELECT *
   FROM cycling.cyclist_name;
   ```

<table>
<thead>
<tr>
<th>id</th>
<th>firstname</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>e7cd5752-bc0d-4157-a80f-7523add8dbcd</td>
<td>Anna</td>
<td>VAN DER BREGGEN</td>
</tr>
</tbody>
</table>

**COPY FROM**

Imports data from a comma-separated values (CSV) file or a delimited text 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 `HEADER = true`, the first row of a file is a header row.

Only use COPY FROM to import datasets that have less than two million rows. To import large datasets, use `sstableloader`.

**Synopsis**

```sql
COPY table_name [ ( column_list ) ]
FROM 'file_name'[ , 'file2_name', ... ] | STDIN
```
COPY supports a list of one or more comma-separated file names or python glob expressions.

Table 16: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italic</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>()</td>
<td>Optional. Square brackets ( { } ) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>( )</td>
<td>Group. Parentheses ( { } ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation ( ’ ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>( key : value )</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>[--]</td>
<td>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.</td>
</tr>
<tr>
<td>‘&lt;schema&gt; ... &lt;/schema&gt; ’</td>
<td>Search CQL only: Single quotation marks ( ’ ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

Setting copy options

Copy options set in the COPY statement take precedence over the 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.

**BOOLSTYLE**
- Boolean indicators for true and false. The values are case-insensitive. For example, yes, no and YES, NO are the same.
- Default: True, False

**CONFIGFILE**
- Specify a directory that contains the cqlshrc configuration file.
  - Command line options always override the cqlshrc file.

**DATETIMEFORMAT**
- Time format for reading or writing CSV time data. The timestamp uses the strftime format. If not set, the default value is set to the datetimeformat value in the cqlshrc file.
- Default: %Y-%m-%d %H:%M:%S%z

**DECIMALSEP**
- Decimal value separator.
- Default: . (period)

**DELEMITER**
Field separator.
Default: , (comma)

**ESCAPE**
Single character that escapes literal uses of the QUOTE character.
Default: \ (backslash)

**HEADER**
- **true** - first row contains headers (column names)
- **false** - first row does not have headers

Default: false

**MAXATTEMPTS**
Maximum number of attempts for errors.
Default: 5

**NULL**
Value used when no value is in the field.
Default: <empty>

**NUMPROCESSES**
Number of worker processes. Maximum value is 16.
Default: -1

**QUOTE**
Encloses field values.
Default: " (double quotation mark)

**REPORTFREQUENCY**
Frequency with which status is displayed in seconds.
Default: 0.25

**RATEFILE**
Print output statistics to this file.

**SKIPCOLS**
Name of column to skip.

**SKIPROWS**
Number of rows starting from the first row of data to skip.

**THOUSANDSSEP**
Separator for thousands digit groups.

**CHUNKSIZE**
Chunk size passed to worker processes.
Default: 1000

**INGESTRATE**
Approximate ingest rate in rows per second. Must be greater than the chunk size.
Default: 100000

**MAXBATCHSIZE**
Maximum size of an import batch.
Default: 20

**MAXINSERTERRORS**
Maximum global number of insert errors. Use -1 for no maximum.
Default: -1

**MAXPARSEERRORS**
Maximum global number of parsing errors. Use -1 for no maximum.
Default: -1

**MAXROWS**
Maximum number of rows. Use -1 for no maximum.
Default: -1

**MINBATCHSIZE**
Minimum size of an import batch.
Default: 2

**Examples**
Create the sample dataset
Set up the environment used for the COPY command examples:

1. **Using CQL, create a cycling keyspace:**

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

2. **Create the cycling.cyclist_name table:**

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

3. **Insert data into cycling.cyclist_name:**

   ```cql
   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, 'KRUIKSWIJK', '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

1. **Export only the id and lastname columns from the cyclist_name table to a CSV file:**

   ```cql
   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 (indicated by ../). If the CSV file already exists, it is overwritten. If you do not have permission to create the file in the directory, you can use a different directory; for example, to use the current working directory, omit the directory path before the file name.

   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 firstname to a different CSV file named cyclist_firstname.csv:**

   ```cql
   COPY cycling.cyclist_name (id, firstname)
   ```
3. Remove all records from the cyclist name table:

```sql
TRUNCATE cycling.cyclist_name;
```

4. Verify that there are no rows:

```sql
SELECT *
FROM cycling.cyclist_name;
```

Query results are empty:

```
id | firstname | lastname
----+-----------+----------
(0 rows)
```

5. Import the cyclist first names:

```sql
COPY cycling.cyclist_name (id,firstname)
FROM '../cyclist_firstname.csv' WITH HEADER = TRUE;
```

The rows are imported:

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:

```sql
SELECT *
FROM cycling.cyclist_name;
```

The rows were created with null last names because the `lastname` 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-48d8-a5f8-97e6fb4c9b47 |    Steven |     null
e7cd5752-bc0d-4157-a80f-7523add8dbcd |      Anna |     null
```
7. Import the last names:

```sql
COPY cycling.cyclist_name (id, lastname)
FROM '../cyclist_lastname.csv' WITH HEADER = TRUE;
```

The records are imported but no new records are 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:

```sql
SELECT *
FROM cycling.cyclist_name;
```

The PRIMARY KEY `id` matched for all records and the `lastname` is populated.

<table>
<thead>
<tr>
<th>id</th>
<th>firstname</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>e7ae5cf3-d358-4d99-b900-85902fda9bb0</td>
<td>Alex</td>
<td>FRAME</td>
</tr>
<tr>
<td>fb372533-eb95-4bb4-8685-6ef61e994ca</td>
<td>Michael</td>
<td>MATTHEWS</td>
</tr>
<tr>
<td>5b6962dd-3f90-4c93-8f61-eabfa4a803e2</td>
<td>Marianne</td>
<td>VOS</td>
</tr>
<tr>
<td>220844bf-4860-49d6-9a4b-6b5d3a79cbfb</td>
<td>Paolo</td>
<td>TIRALONGO</td>
</tr>
<tr>
<td>6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47</td>
<td>Steven</td>
<td>KRIUKSWIJK</td>
</tr>
<tr>
<td>e7cd5752-bc0d-4157-a80f-7523add8dbcd</td>
<td>Anna</td>
<td>VAN DER BREGGEN</td>
</tr>
</tbody>
</table>

Copy data from standard input to a table.

1. Clear the data from the `cyclist_name` table:

```sql
TRUNCATE cycling.cyclist_name;
```

2. Start the copy input operation using the `FROM STDIN` option:

```sql
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 the `[copy]` prompt, enter the field values in a common-separated list; on the last line of data, enter a period:

```
[copys] e7cd5752-bc0d-4157-a80f-7523add8dbcd,Anna,VAN DER BREGGEN
```
4. Press Enter after the period:

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. Run this query to view the contents of the cyclist_name table:

```
SELECT *
FROM cycling.cyclist_name;
```

<table>
<thead>
<tr>
<th>id</th>
<th>firstname</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>e7cd5752-bc0d-4157-a80f-7523add8dbcd</td>
<td>Anna</td>
<td>VAN DER BREGGEN</td>
</tr>
</tbody>
</table>

(1 rows)

**DESCRIBE AGGREGATE**

Shows this output depending on the options selected:

- All user-defined aggregates (UDAs) in the current keyspace.
- All UDAs in the cluster when no keyspace is selected.
- Definition as an executable CQL statement for a specific UDA.

**Synopsis**

```
DESCRIBE AGGREGATES | AGGREGATE [keyspace_name.]uda_name
```

**Table 17: Legend**

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UPPERCASE</strong></td>
<td>Literal keyword.</td>
</tr>
<tr>
<td><strong>Lowercase</strong></td>
<td>Not literal.</td>
</tr>
<tr>
<td><strong>Italics</strong></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets ( [ ] ) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>( )</td>
<td>Group. Parentheses ( ( ) ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td><code>'literal string'</code></td>
<td>Single quotation ( ’ ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
</tbody>
</table>
### Syntax conventions

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[---]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt;'</td>
<td>Search CQL only: Single quotation marks (‘’ ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

### AGGREGATES

**All aggregates.**

**AGGREGATE**

**One aggregate.**

*keyspace_name*

Name of the keyspace.

*uda_name*

Name of the user-defined aggregate.

### Examples

List all cycling aggregates:

```cql
USE cycling; DESC AGGREGATES;
```

*average(int)*

Show the definition of the cycling average aggregate:

```cql
DESC AGGREGATE cycling.average
```

The result is an executable CQL statement:

```cql
CREATE AGGREGATE cycling.average(int)
  SFUNC avgstate
  STYPE frozen<tuple<int, bigint>>
  FINALFUNC avgfinal
  INITCOND (0, 0);
```

### DESCRIBE CLUSTER

Shows cluster information, including the cluster name and partitioner.

**Synopsis**

DESCRIBE CLUSTER

### Table 18: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italic</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets ([ ]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
</tbody>
</table>
CQL reference

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>Group. Parentheses ( ( ) ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation ( ' ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1, datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets (&lt; &gt;) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>[---]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ' ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>$xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

CLUSTER
Shows general information, including the cluster and partitioner name.

Examples
Show high-level details for a single node test cluster:

DESC CLUSTER
Cluster: Test Cluster
Partitioner: Murmur3Partitioner

DESCRIBE SCHEMA
Shows definitions for all objects in the cluster.

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 quotation marks.

Synopsis

DESCRIBE [ FULL ] SCHEMA

Table 19: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italic</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets ( [ ] ) surround optional command arguments. Do not type the square brackets.</td>
</tr>
</tbody>
</table>
### Syntax conventions

<table>
<thead>
<tr>
<th>Group. Parentheses (() identify a group to choose from. Do not type the parentheses.</th>
<th>Group. Parentheses (() identify a group to choose from. Do not type the parentheses.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Or. A vertical bar (</td>
<td>) separates alternative elements. Do not type the vertical bar.</td>
</tr>
<tr>
<td>Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.</td>
<td>Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation (‘) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ({{}} enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1, datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets (&lt;&gt;) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon (;) terminates all CQL statements.</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt;'</td>
<td>Search CQL only: Single quotation marks (‘) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>$xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

### Full

Shows definitions for all objects.

### Schema

Shows definitions for all non-system objects.

### Examples

Shows definitions for all non-system objects:

```
DESC SCHEMA
```

This output shows the beginning of the returned results:

```
CREATE KEYSPACE dse_system_local WITH replication =
  {'class': 'LocalStrategy'} AND durable_writes = true;

CREATE TABLE dse_system_local.solr_resources (
  core_name text,
  resource_name text,
  resource_value blob,
  PRIMARY KEY (core_name, resource_name)
) WITH CLUSTERING ORDER BY (resource_name ASC)
```

### Describe function

Shows this output depending on the options selected:

- All user-defined functions (UDFs) in the current keyspace.
- All UDFs in the cluster when no keyspace is selected.
- Definition as an executable CQL statement for a specific UDF.

To select a keyspace, use the `USE` command.
### Synopsis

**DESCRIBE FUNCTIONS** | **FUNCTION** `{keyspace_name}.udf_name`

**Table 20: Legend**

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UPPERCASE</strong></td>
<td>Literal keyword.</td>
</tr>
<tr>
<td><strong>Lowercase</strong></td>
<td>Not literal.</td>
</tr>
<tr>
<td><strong>Italics</strong></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>{}</td>
<td>Optional. Square brackets <code>{[]}</code> surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>()</td>
<td>Group. Parentheses <code>{()}</code> identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td>Or. A vertical bar `{</td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis <code>{...}</code> indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>&quot;literal string&quot;</td>
<td>Single quotation <code>{’}</code> marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces <code>{{}}</code> enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1, datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets <code>{&lt; &gt;}</code> enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon <code>{;}</code> terminates all CQL statements.</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks <code>{’}</code> surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity=&quot;xml_entity_type&quot;</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

**FUNCTIONS**

All functions.

**FUNCTION**

One function.

*keyspace_name*

Name of the keyspace.

*udf_name*

Name of the function.

**Examples**

List the functions in the cycling keyspace:

```
USE cycling; DESC FUNCTIONS
```

```cql
flog(double) avgfinal(frozen<tuple<int, bigint>>)>
```
avgstate(frozen<tuple<int, bigint>>,int)  left(text,int)

Show the definition of the cycling flog function:

DESC FUNCTION cycling.flog

CREATE FUNCTION cycling.flog(input double)
     CALLED ON NULL INPUT
     RETURNS double
     LANGUAGE java
     AS $$return Double.valueOf(Math.log(input.doubleValue()));$$;

DESCRIBE INDEX

Shows the definition as an executable CQL statement of a specified index.

Synopsis

DESCRIBE INDEX [keyspace_name.]index_name

Table 21: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td><em>Italics</em></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets ([ ]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>( )</td>
<td>Group. Parentheses ( () ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation ( ’ ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>( key : value )</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon (;) terminates all CQL statements.</td>
</tr>
<tr>
<td>[--]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ’ ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity=&quot;xml_entity_type&quot;</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

keyspace_name
Name of the keyspace.

index_name

CQL reference

Name of the index. If an index name was not specified when the index was created, the default name is the column name, underscore, followed by idx.

Examples

Create an index:

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

Show the definition of the index:

```
DESC INDEX cycling.nationality_idx
```

Output:

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

DESCRIBE KEYSERACE

Shows this output depending on the options selected:

- List of all keyspaces in the cluster.
- Definition of each object in a specified keyspace as an executable CQL statement.

Verify all settings before executing the full output, some options may be cluster specific in the WITH statement.

Synopsis

```
DESCRIBE KEYSERACES | KEYSERACE keyspace_name
```

Table 22: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td><code>itotals</code></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[]</td>
<td>Optional. Square brackets ([ ]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>{}</td>
<td>Group. Parentheses ( { } ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td><code>'literal string'</code></td>
<td>Single quotation ( ‘ ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
</tbody>
</table>
### Syntax conventions

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;datatype1, datatype2&gt;</code></td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td><code>cql_statement;</code></td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td><code>[--]</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>' &lt;schema&gt; ... &lt;/schema&gt; '</code></td>
<td>Search CQL only: Single quotation marks ( ’ ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td><code>@xml_entity='xml_entity_type'</code></td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

### KEYSIZES

**All keyspaces.**

**KEYSPACE**

One keyspace.

**keyspace_name**

Name of the keyspace.

### Examples

**Show all keyspaces:**

```cql
DESC KEYSACES
```

All the keyspaces on the cluster are listed:

<table>
<thead>
<tr>
<th>dse_system_local</th>
<th>cycling</th>
<th>system</th>
<th>keyspace1</th>
<th>system_traces</th>
</tr>
</thead>
<tbody>
<tr>
<td>dse_security</td>
<td>system_schema</td>
<td>dse_leases</td>
<td>system_distributed</td>
<td>dse_perf</td>
</tr>
<tr>
<td>solr_admin</td>
<td>system_auth</td>
<td>dse_audit</td>
<td>test</td>
<td>dse_system</td>
</tr>
</tbody>
</table>

### DESCRIBE MATERIALIZED VIEW

Shows the definition as an executable CQL statement for the specified materialized view.

Verify all settings before executing the full output, some options may be cluster specific in the WITH statement.

### Synopsis

```
DESCRIBE MATERIALIZED VIEW [keyspace_name.]view_name
```

### Table 23: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italic</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets ( [ ] ) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>{ }</td>
<td>Group. Parentheses ( { } ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td>{ }</td>
<td>Or. A vertical bar (</td>
</tr>
</tbody>
</table>
## Syntax conventions

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>...</code></td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td><code>'literal string'</code></td>
<td>Single quotation ( ' ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td><code>{ key : value }</code></td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td><code>&lt;datatype1,datatype2&gt;</code></td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td><code>cql_statement;</code></td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td><code>[--]</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>' &lt;schema&gt; ... &lt;/schema&gt; '</code></td>
<td>Search CQL only: Single quotation marks ( ' ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td><code>@xml_entity=xml_entity_type</code></td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

### keyspace_name
Name of the keyspace.

### view_name
Name of the materialized view.

## Examples

Show details for cyclist by birthday materialized view:

```
DESC MATERIALIZED VIEW cycling.cyclist_by_birthday
```

```
CREATE MATERIALIZED VIEW cycling.cyclist_by_birthday AS
  SELECT birthday, cid, age, country, name
  FROM cycling.cyclist_mv
  WHERE birthday IS NOT NULL AND cid IS NOT NULL
  PRIMARY KEY (birthday, cid)
  WITH CLUSTERING ORDER BY (cid 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 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 speculative_retry = '99PERCENTILE';
```

The table that the materialized view is based on is shown in the FROM clause of the returned CQL statement.

## DESCRIBE TABLE

Shows this output depending on the options selected:
• List of all tables in a keyspace.
• Executable CQL information for a single table, including materialized views that are based on the table.

Verify all settings before executing the full CQL output for recreating a table returned by this command. Some table options might be cluster-specific.

Synopsis

DESCRIBE TABLES | TABLE [keyspace_name.]table_name

Table 24: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td><em>italics</em></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets ([ ]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>( )</td>
<td>Group. Parentheses ( { } ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation ( ’ ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon (; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>[--]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ’ ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

TABLES
All tables.

TABLE
One table.

dataset
Name of the keyspace.

dataset
Name of the table.
Examples

Show a list of tables in the system keyspace:

```
USE system; DESC TABLES
```

A list of all the tables in the keyspace is returned:

```
repairs              view_builds_in_progress  paxos           transferred_ranges
available_ranges     peers                     size_estimates
batches              compaction_history       built_views
prepared_statements  sstable_activity        range_xfers
"IndexInfo"           peer_events              local
```

Show the CQL for the cycling calendar table:

```
DESC cycling.calendar
```

A complete CQL table description is returned, which can be used to recreate the table:

```
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 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 speculative_retry = '99PERCENTILE';
```

DESCRIBE TYPE

Shows this output depending on the options selected:

- Lists all user-defined types (UDTs) in the current keyspace.
- Lists all UDTs in the cluster when no keyspace is selected.
- CQL definition for a specified UDT.

To select a keyspace, use the `USE` command.
Synopsis

DESCRIBE TYPES [ keyspace_name ] | TYPE [ keyspace_name.]udt_name

Table 25: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italic</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>()</td>
<td>Optional. Square brackets ( [ ] ) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>( )</td>
<td>Group. Parentheses ( ( ) ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation ( ‘ ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>[--]</td>
<td>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.</td>
</tr>
<tr>
<td>'&lt; &lt;schema&gt; ... &lt;/schema&gt;'</td>
<td>Search CQL only: Single quotation marks ( ‘ ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

TYPES
All types.

TYPE
One type.

keyspace_name
Name of the keyspace.

udt_name
Name of the type.

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 quotation marks.
Examples

List all the types in the cycling keyspace:

```
USE cycling; DESC TYPES
```

```
fullname  race  basic_info
```

Show the definition of the cycling fullname type:

```
DESC TYPE cycling.fullname
```

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

EXECUTE AS

Provides a testing mechanism when setting up proxy roles by running CQL commands as another role during a CQL shell session.

To start, use EXECUTE AS `role_name`. The commands run with the permissions of the specified role. To stop, use EXECUTE AS without a role name.

Requires PROXY.EXECUTE on the specified role, see GRANT.

Synopsis

```
EXECUTE AS [ role_name ]
```

Table 26: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italic</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>()</td>
<td>Optional. Square brackets ( [ ] ) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>{}</td>
<td>Group. Parentheses ( { } ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'Literal string'</td>
<td>Single quotation ( ' ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
</tbody>
</table>
### Syntax conventions

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>[---]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ' ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

### role_name

The name of the role to run the commands as. The commands run with the permissions of the specified role.

**Example**

Use the read_race role to execute commands:

```
EXECUTE AS read_race;
```

Executing queries as read_race.

Perform a query:

```
SELECT *
FROM cycling.race_times;
```

<table>
<thead>
<tr>
<th>race_name</th>
<th>race_time</th>
<th>cyclist_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>17th Santos Tour Down Under</td>
<td>19:15:18</td>
<td>Rohan DENNIS</td>
</tr>
<tr>
<td>17th Santos Tour Down Under</td>
<td>19:15:20</td>
<td>Richie PORTE</td>
</tr>
<tr>
<td>17th Santos Tour Down Under</td>
<td>19:15:38</td>
<td>Cadel EVANS</td>
</tr>
<tr>
<td>17th Santos Tour Down Under</td>
<td>19:15:40</td>
<td>Tom DUMOULIN</td>
</tr>
</tbody>
</table>

(4 rows)

Stop using the read_race role:

```
EXECUTE AS
```

### EXPAND

Formats query output vertically. For each row, the column values are listed vertically. Use this command to read wide data. To show the current expand setting, run EXPAND without specifying ON or OFF.
**CQL reference**

**Synopsis**

```
EXPAND [ ON | OFF ]
```

**Table 27: Legend**

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UPPERCASE</strong></td>
<td>Literal keyword.</td>
</tr>
<tr>
<td><strong>Lowercase</strong></td>
<td>Not literal.</td>
</tr>
<tr>
<td><strong>Italics</strong></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>{}</td>
<td>Optional. Square brackets ( {} ) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>()</td>
<td>Group. Parentheses ( () ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>&quot;literal string&quot;</td>
<td>Single quotation ( ' ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement: A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>[...]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ' ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>$xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

**Examples**

**View rows vertically:**

```
EXPAND ON
```

Select all rows from the cycling race winners table:

```
SELECT *
FROM cycling.race_winners;
```

Each field is shown in a vertical row table:

```
0 Row 1
-----------------------------
```
### Table 28: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td><code>@tiles=</code></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets ([ ]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>( )</td>
<td>Group. Parentheses (( )) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
</tbody>
</table>
### Syntax conventions

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'literal string'</td>
<td>Single quotation (‘’) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ({} ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets (&lt;&gt;) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon (; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>[--]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks (‘’) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

There are no parameters for this command.

**Example**

End the CQL shell and return to the system command prompt:

```
EXIT
```

**LOGIN**

Switches the user account without ending the CQL shell session. To log in as a different user, specify the credentials for that user.

**Synopsis**

```
LOGIN user_name [ 'password' ]
```

**Table 29: Legend**

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italic</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets ([ ]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>( )</td>
<td>Group. Parentheses ( ( ) ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( . . . ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation (’’) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ({} ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets (&lt;&gt;) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
</tbody>
</table>
Syntax conventions

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cql_statement;</code></td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td><code>[-]</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>' &lt;schema&gt; ... &lt;/schema&gt; '</code></td>
<td>Search CQL only: Single quotation marks ( ’ ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td><code>@xml_entity=&quot;xml_entity_type&quot;</code></td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

**Examples**

Log in as the cycling administrator:

```
LOGIN cycling_admin
```

When no password is specified on the command line, a password prompt appears:

```
password: ********
```

**PAGING**

Performs one of these actions, depending on the options selected:

- Shows the current query paging status.
- Enables or disables paging.
- Sets the number of lines in each page.

Query results are displayed in 100-line groups, known as pages, followed by the `more` prompt. Press the space bar to move to the next group. **PAGING** without an option shows the current paging status, which is either enabled or disabled.

**Synopsis**

```
PAGING [ ON | OFF ] ( page_size )
```

**Table 30: Legend**

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italic</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets ( [ ] ) surround optional command arguments. Do not type the square brackets.</td>
</tr>
</tbody>
</table>
### Syntax conventions

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>Group. Parentheses ( ( ) ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td>Or. A vertical bar (</td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation ( ' ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1, datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets (&lt; &gt;) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>[--]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ' ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>$xml_entity=&quot;xml_entity_type&quot;</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

### Examples

#### Show the current paging status:

```
PAGING
```

#### Reports the current status and page size, which are the number of lines in each group:

```
Query paging is currently enabled. Use PAGING OFF to disable
Page size: 100
```

#### Sets the page size to 500:

```
PAGING 500
```

### SERIAL CONSISTENCY

Sets the serial consistency for lightweight transactions (LWTs). A LWT contains an IF EXISTS or IF NOT EXISTS clause. See Data consistency in the documentation.

When using a LWT:

- Levels must be set for SERIAL CONSISTENCY and CONSISTENCY.
- SERIAL CONSISTENCY can be set to SERIAL or LOCAL_SERIAL.
• CONSISTENCY cannot be set to SERIAL or LOCAL_SERIAL.

To set the consistency level for non-LWTs, use the CONSISTENCY command.

**Synopsis**

```
SERIAL CONSISTENCY [ consistency_level ]
```

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td><em>italics</em></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets ([ ]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>( )</td>
<td>Group. Parentheses ( ( ) ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'Literal string'</td>
<td>Single quotation ( ' ') marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1, datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon (;) terminates all CQL statements.</td>
</tr>
<tr>
<td>[---]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ' ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>$xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

**consistency_level**

Consistency level. Can be set to SERIAL or LOCAL_SERIAL.
Examples

Show the 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.
```

You can use the TRACE command to examine the difference between INSERT statements with and without IF EXISTS.

When using a LWT, you must have levels set for CONSISTENCY and SERIAL CONSISTENCY.

For example, write data using a LWT with IF NOT EXISTS to examine if the row already exists before performing the write:

```
CONSISTENCY ONE

SERIAL CONSISTENCY SERIAL

INSERT INTO cycling.cyclist_name (id, firstname, lastname)
    VALUES (e7ae5cf3-d358-4d99-b900-85902fda9bb0, 'Alex', 'FRAME')
IF NOT EXISTS;
```

If the record already exists then the write is not applied, as shown in the following result:

```
[applied] | id                        | firstname | lastname
----------|----------------------------|-----------|----------
False     | e7ae5cf3-d358-4d99-b900-85902fda9bb0 | Alex      | FRAME
```

SHOW

Shows this information depending on the options selected:

- Software version.
- Current session node.
- Tracing session details captured in the past 24 hours.
## Synopsis

SHOW VERSION | HOST | SESSION tracing_session_id

### Table 32: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td><em>italics</em></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets ([ ]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>( )</td>
<td>Group. Parentheses ( ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td>Or. A vertical bar (</td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation ( ’ ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets (&lt; &gt;) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon (;) terminates all CQL statements.</td>
</tr>
<tr>
<td>[---]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ’ ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>$xml_entity=xml_entity_type</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

### VERSION
Show the version for cqlsh, the database, the CQL specification, and native protocol.

### HOST
Show the node details for the cqlsh session host.

### SESSION tracing_session_id
Show the activity details for a specific query provided in tracing_session_id. Session IDs are shown in the query results and are recorded to the system_traces.sessions table.

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.
Examples

Show the version:

```
SHOW VERSION
```

Example output, which varies depending on your implementation:

```
[cqlsh 5.0.1 | DSE 6.7.3 | CQL spec 3.4.5 | DSE protocol v2]
```

Show the host information for the cqlsh session host:

```
SHOW HOST
```

Returns the host name, IP address, and port of the CQL shell session. For example:

```
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
```

Use a session ID from the query results or from the system_traces.sessions table.

Sample output from SHOW SESSION:

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

<table>
<thead>
<tr>
<th>activity</th>
<th>timestamp</th>
<th>source</th>
<th>source_elapsed</th>
</tr>
</thead>
<tbody>
<tr>
<td>execute_cql3_query</td>
<td>12:19:52,372</td>
<td>127.0.0.1</td>
<td>0</td>
</tr>
<tr>
<td>Parsing CREATE TABLE emp</td>
<td>12:19:52,372</td>
<td>127.0.0.1</td>
<td>153</td>
</tr>
<tr>
<td>PRIMARY KEY (empID, deptID)</td>
<td>12:19:52,372</td>
<td>127.0.0.1</td>
<td></td>
</tr>
<tr>
<td>Request complete</td>
<td>12:19:52,372</td>
<td>127.0.0.1</td>
<td>650</td>
</tr>
</tbody>
</table>
```

SOURCE

Executes a file containing CQL statements.

The output of each statement is shown in the standard output (STDOUT), including error messages. You can use `IF NOT EXISTS` to suppress errors for some statements, such as `CREATE KEYSPACE`. All statements in the file are executed, even if a no-operation error occurs.
Synopsis

Table 33: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td><em>italics</em></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>()</td>
<td>Optional. Square brackets ( [ ] ) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>()</td>
<td>Group. Parentheses ( { } ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td>Or. A vertical bar (</td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation ( ' ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1, datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets (&lt; &gt;) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ' ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

file_name

Name of the file to execute. Specify the path of the file relative to the current directory, which is the directory where cqlsh was started on your local computer. Enclose the file name in single quotation marks. Use tilde (~) for your home directory.

Examples

Execute CQL statements from a file:

SOURCE '~/cycling_setup/create_ks_and_tables.cql'

To execute a CQL file without starting a shell session, use bin/cqlsh --file 'file_name'.

TRACING

Enables or disables tracing for transactions on all nodes in the cluster. Use tracing to troubleshoot performance problems. Detailed transaction information related to internal operations is captured in the system_traces keyspace.
When a query runs in a tracing session, a session ID is shown in the query results. In addition, an entry with the high-level details such as session ID, client, and session length, is written to the `system_traces.session` table. Details for each operation are written to the `system_traces.events` table.

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 for longer than 24 hours, copy the data to another location. For information about probabilistic tracing, see `nodetool settraceprobability`.

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>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italics</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[]</td>
<td>Optional. Square brackets <code>{[]}</code> surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>()</td>
<td>Group. Parentheses <code>( )</code> identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis <code>{...}</code> indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>&quot;literal string&quot;</td>
<td>Single quotation (<code>'</code>) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>( key : value )</td>
<td>Map collection. Braces <code>{ }</code> enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets <code>&lt; &gt;</code> enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon <code>;</code> terminates all CQL statements.</td>
</tr>
<tr>
<td>[--]</td>
<td>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.</td>
</tr>
<tr>
<td><code> '&lt;schema&gt; ... &lt;/schema&gt;'</code></td>
<td>Search CQL only: Single quotation marks (<code>'</code>) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td><code>@xml_entity=&quot;xml_entity_type&quot;</code></td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

**ON**

Enables tracing.

**OFF**

Disables tracing.
**Examples**

**Tracing a write request**

The following example shows tracing activity on a three node cluster. The cluster has a keyspace that has a replication factor of three, and an employee table similar to the table in Using a compound primary key.

The tracing shows that the coordinator performs these actions:

- Identifies the target nodes for replication of the row.
- Writes the row to the commit log 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');
```

The request and each step are captured and displayed:

```
<table>
<thead>
<tr>
<th>activity</th>
<th>timestamp</th>
<th>source</th>
<th>source_elapsed</th>
<th>client</th>
</tr>
</thead>
<tbody>
<tr>
<td>CQL3 query</td>
<td>2016-11-22 16:34:34.300000</td>
<td>127.0.0.1</td>
<td>0</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Parsing INSERT INTO cycling.cyclist_name (id, lastname, firstname) VALUES (e7ae5cf3-d358-4d99-b900-85902fda9bb0, 'FRAME', 'Alex'); [Native-Transport-Requests-1]</td>
<td>2016-11-22 16:34:34.305000</td>
<td>127.0.0.1</td>
<td>5935</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Preparing statement [Native-Transport-Requests-1]</td>
<td>2016-11-22 16:34:34.308000</td>
<td>127.0.0.1</td>
<td>9199</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Determining replicas for mutation [Native-Transport-Requests-1]</td>
<td>2016-11-22 16:34:34.330000</td>
<td>127.0.0.1</td>
<td>30530</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Appending to commitlog [MutationStage-3]</td>
<td>2016-11-22 16:34:34.330000</td>
<td>127.0.0.1</td>
<td>30979</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Adding to cyclist_name memtable [MutationStage-3]</td>
<td>2016-11-22 16:34:34.330000</td>
<td>127.0.0.1</td>
<td>31510</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Request complete</td>
<td>2016-11-22 16:34:34.333633</td>
<td>127.0.0.1</td>
<td>33633</td>
<td>127.0.0.1</td>
</tr>
</tbody>
</table>
```

The `source_elapsed` column value is the elapsed time of the event on the source node in microseconds.

**Tracing a sequential scan**

A single row is spread across multiple SSTables. Reading one row involves reading data from multiple SSTables, as shown by the trace for this request to read rows from the `cyclist_name` table:

```
SELECT *
```
FROM cycling.cyclist_name;

The query results display first, followed by the session ID and session details:

<table>
<thead>
<tr>
<th>id</th>
<th>firstname</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>e7ae5cf3-d358-4d99-b900-85902fda9bb0</td>
<td>Alex</td>
<td>FRAME</td>
</tr>
<tr>
<td>fb372533-eb95-4bb4-8685-6ef61e994c6a</td>
<td>Michael</td>
<td>MATTHEWS</td>
</tr>
<tr>
<td>5b6962dd-3f90-4c93-8f61-eabfa4a803e2</td>
<td>Marianne</td>
<td>VOS</td>
</tr>
<tr>
<td>220844bf-4860-49d6-9a4b-6b5d3a897c6b</td>
<td>Paolo</td>
<td>TIRALONGO</td>
</tr>
<tr>
<td>6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47</td>
<td>Steven</td>
<td>KRUIKSWIJK</td>
</tr>
<tr>
<td>e7cd5752-bc0d-4157-a80f-7523add8dbcd</td>
<td>Anna</td>
<td>VAN DER BREGGEN</td>
</tr>
</tbody>
</table>

(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 16:45:02.212000 | 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 16:45:02.212000 | 127.0.0.1 | 1632| 127.0.0.1    | Submitted 1 concurrent range requests |
| [Native-Transport-Requests-1] | 2016-11-22 16:45:02.212000 | 127.0.0.1 | 3002| 127.0.0.1    | Executing seq scan across 1 sstables for (min(-9223372036854775808), min(-9223372036854775808)), [ReadStage-2] |
| [Native-Transport-Requests-1] | 2016-11-22 16:45:02.212000 | 127.0.0.1 | 3130| 127.0.0.1    | Read 6 live and 0 tombstone cells [ReadStage-2] |
| [Native-Transport-Requests-1] | 2016-11-22 16:45:02.212000 | 127.0.0.1 | 3932| 127.0.0.1    | Request complete |
| [Native-Transport-Requests-1] | 2016-11-22 16:45:02.212000 | 127.0.0.1 | 4252| 127.0.0.1    |  

CQL commands

This section describes the commands that are specific to CQL.

ALTER KEYSPACE

Modifies the keyspace replication strategy, which is the number of copies of the data created in each datacenter, Table 2, and/or disable the commit log for writes, Durable Writes.

Datacenter names are case sensitive. Use nodetool status to verify the datacenter name.

Changing the keyspace name is not supported.
Synopsis

```
ALTER KEYSPACE keyspace_name
    WITH REPLICATION = {replication_map}
    [AND DURABLE_WRITES = true|false] ;
```

Table 35: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italic</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[]</td>
<td>Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>()</td>
<td>Group. Parentheses () identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation (’) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ({}) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1, datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets (&lt;&gt;) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon (;) terminates all CQL statements.</td>
</tr>
<tr>
<td>[--]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt;'</td>
<td>Search CQL only: Single quotation marks (’) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

replication_map

`'class' : 'SimpleStrategy', 'replication_factor' : N`

Assigns the same replication factor to the entire cluster. Use for evaluation and single datacenter test and development environments only.

```
REPLICATION = {
    'class' : 'SimpleStrategy',
    'replication_factor' : N
}
```

`'class' : 'NetworkTopologyStrategy', 'datacenter_name' : N, ...`
After the class declaration, assign replication factors to each datacenter by name in a comma-separated list. Use in production environments and multi-DC test and development environments. Datacenter names must match the snitch DC name; see Snitches.

```
REPLICATION = {
    'class' : 'NetworkTopologyStrategy',
    'datacenter_name' : N [, 'datacenter_name' : N]
}
```

Use only replication strategy implementations bundled with DDAC.

**DURABLE_WRITES = true | false**

Optional. (Not recommended), false bypasses the commit log when writing to the keyspace. Default value is true.

Never disable durable writes when using SimpleStrategy replication.

**Example**

Change the cycling keyspace to NetworkTopologyStrategy in a single datacenter and turn off durable writes (not recommended). This example uses the default datacenter name with a replication factor of 3.

```
ALTER KEYSPACE cycling
WITH REPLICAATION = {
    'class' : 'NetworkTopologyStrategy',
    'datacenter1' : 3 }
AND DURABLE_WRITES = false;
```

**ALTER MATERIALIZED VIEW**

Changes materialized view table properties. The statement returns no results.

- Changing columns is not supported.
- Change log, CDC, is not available for materialized views.

**Synopsis**

```
ALTER MATERIALIZED VIEW [keyspace_name.]view_name
WITH table_options [ AND table_options ... ] ;
```

**Table 36: Legend**

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td><em>italics</em></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets ([ ]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>{ }</td>
<td>Group. Parentheses ({ }) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Syntax conventions</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>&quot;literal string&quot;</td>
<td>Single quotation ( ' ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>( key : value )</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>[---]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ' ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity=&quot;xml_entity_type&quot;</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

**keyspace_name**
Selects a keyspace.

**view_name**
Selects the materialized view.

**table_options**
Table options are defined when the materialized view is created. Modify the table_options in the WITH clause using the following syntax:

- Single value using the option_name = 'value'. Enclose string values in single quotes, and no quotes for numbers, boolean, etc.
- Specify options with multiple subproperties in simple JSON format, option_name = { option_map }.
- Set multiple table options using AND.

**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 }
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 dcllocal_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 sets superuser or login options.

**Synopsis**

```
ALTER ROLE role_name
    [ WITH [ PASSWORD = 'password'
    | LOGIN = (true | false)
    | SUPERUSER = (true | false)
    | OPTIONS = option_map ]
```

**Table 37: Legend**

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italic</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
</tbody>
</table>
### Syntax conventions

| [ ]  | 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. |
| 'literal string' | Single quotation ( ' ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case. |
| ( 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. |

### Description

**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

Add new columns, drop existing columns, renames columns, and change table properties. The command returns no results.

- Can only rename clustering columns in the primary key.
- Cannot change the data type of a column.
- For a table that has a materialized view, cannot drop a column from the table even if the column is not used in the materialized view.
- Cannot rename or drop columns that have dependent secondary indexes or Datastax Enterprise Search indexes.
Do not add a column with the same name as an existing column but with a different data type. It will prevent commit log replays and corrupt existing SSTables with old data.

ALTER COLUMNFAMILY is deprecated.

Synopsis

```
ALTER TABLE [keyspace_name.]table_name
[ ADD ( column_definition | column_definition_list ) ]
[ DROP ( column | column_list | COMPACT STORAGE ) ]
[ RENAME column_name TO column_name ]
[ WITH table_properties [ , ... ] ] ;
```

Table 38: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>italics</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>()</td>
<td>Optional. Square brackets ( ) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>{()}</td>
<td>Group. Parentheses ( ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation ( ' ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>( key : value )</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>[---]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ' ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

**ADD column_definition | (column_definition_list)**

Add one or more columns and set the data type. Set the column name followed by the data type. The value is automatically set to null. To add multiple columns, use a comma separated list of columns placed inside parentheses.

```
column_name cql_type [ , ]
[ column_name cql_type [ , ... ]
```

Adding columns to a primary key is not supported after a table has been created.
CQL reference

**DROP column \((column_list)\)**

Drop one or more columns. The values contained in the row are also dropped and not recoverable. To drop multiple columns, use a comma separated list of columns placed inside parentheses.

**RENAME column_name TO column_name**

Changes the name of a primary key column and preserves the existing values.

Not supported on materialized view base-tables, or tables with secondary indexes or Datastax Enterprise Search indexes.

**table_properties**

You can modify an existing table’s properties. Some properties are single options that are set to a value:

```plaintext
option_name = value [ AND ... ]
```

For example, `speculative_retry = '10ms'`. Enclose the value for a string property in single quotation marks.

Other table properties are set using a JSON map:

```plaintext
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:

```plaintext
ALTER TABLE cycling.teams
ADD manager uuid;
```

**Adding a column**

To add a column to a table, use the **ADD** instruction; for example:

```plaintext
ALTER TABLE cycling.cyclist_races
ADD firstname text;
```

To add a column of a collection type:

```plaintext
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:

```plaintext
ALTER TABLE cycling.basic_info
DROP birth_year;
```

**DROP** removes the column from the table definition. The column becomes unavailable for queries immediately after it is dropped. The database 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.

- If you drop a column then re-add it, DataStax Distribution of Apache 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 the **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 because the partition key determines the data storage location on a node. If a different partition name is required, the table must be recreated and the data migrated.

  There are many restrictions when using **RENAME** because SSTables are immutable. To change the state of things on disk, everything must be rewritten.

- 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.

Modifying table properties

To change an existing table’s properties, use **ALTER TABLE** and **WITH**. You can specify a:

- Single property name and value.
- Property map to set the names and values, 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.

If you change the compaction strategy of a table with existing data, the database 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 the compaction strategy on a production cluster and Impact of Changing Compaction Strategy.

Changing caching

Set the number of rows per partition to store in the row cache for a table to 10 rows:

```cql
ALTER TABLE cycling.events
WITH caching = {
'keys': 'NONE',
'rows_per_partition': 10
};
```

Reviewing the table definition

Use `DESCRIBE` or `DESC` to view the table definition.

```cql
DESC TABLE cycling.events;
```

The table details including the column names are returned.

```cql
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 memtableFlush_period_in_ms = 0  
AND min_index_interval = 128  
AND read_repair_chance = 0.0  
AND speculative_retry = '99PERCENTILE';
```

**ALTER TYPE**

Modifies an existing user-defined type (UDT).

Modifying UDTs used in primary keys or index columns is not supported. Changing the field type is not supported.

**Synopsis**

```cql
ALTER TYPE field_name
    ( ADD field_name 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.

### AND
Use between clauses to make multiple changes.

### Examples
**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.

```cql
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 in the cycling.fullname UDT.

```sql
ALTER TYPE cycling.fullname
RENAME middlename TO middle
AND lastname to last
AND firstname to first;
```

Verify the changes using describe:

```sql
DESC TYPE cycling.fullname
```

The new field names appear in the description.

```sql
CREATE TYPE cycling.fullname (  
    first text,
    last text,
    middle text
);
```

**ALTER USER**

Alter existing user options.

*Deprecated.* `ALTER USER` is supported for backwards compatibility only. Use `ROLE` for authentication and authorization.

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**

```sql
ALTER USER user_name
WITH PASSWORD 'password'  
[ ( SUPERUSER | NOSUPERUSER ) ]
```

**Table 40: Legend**

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UPPERCASE</strong></td>
<td>Literal keyword.</td>
</tr>
<tr>
<td><strong>Lowercase</strong></td>
<td>Not literal.</td>
</tr>
<tr>
<td><strong>Italics</strong></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td><strong>[]</strong></td>
<td>Optional. Square brackets ( [ ] ) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td><strong>{}</strong></td>
<td>Group. Parentheses ( { } ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td>**</td>
<td>**</td>
</tr>
<tr>
<td><strong>...</strong></td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td><strong>'literal string'</strong></td>
<td>Single quotation ( ‘ ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
</tbody>
</table>
### Syntax conventions

<table>
<thead>
<tr>
<th><strong>Key</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{ key : value }</code></td>
<td>Map collection. Braces (<code>{ }</code>) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td><code>&lt;datatype1,datatype2&gt;</code></td>
<td>Set, list, map, or tuple. Angle brackets (<code>&lt; &gt;</code>) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td><code>cql_statement;</code></td>
<td>End CQL statement. A semicolon (<code>;</code>) terminates all CQL statements.</td>
</tr>
<tr>
<td><code>[--]</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>' &lt;schema&gt; ... &lt;/schema&gt; '</code></td>
<td>Search CQL only: Single quotation marks (<code>'</code>) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td><code>@xml_entity='xml_entity_type'</code></td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

### 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.

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.

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**.

Batches are not isolated among client programs. Other client programs can read the first modified rows from the batch while the other remaining statements in the batch are in progress. There is no batch rollback functionality, which means that a batch cannot be undone.

### Synopsis

```
BEGIN [ { UNLOGGED | COUNTER } ] BATCH
[ USING TIMESTAMP [ epoch_microseconds ] ]
dml_statement [ USING TIMESTAMP [ epoch_microseconds ] ] ;
[ dml_statement [ USING TIMESTAMP [ epoch_microseconds ] ] [ ; ... ] ] ;
```
Table 41: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td><em>italics</em></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>{}</td>
<td>Optional. Square brackets ({} ) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>( )</td>
<td>Group. Parentheses (( )) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td>Or. A vertical bar (</td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation (’ ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ({} ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets (&lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon (; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>[...]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt;'</td>
<td>Search CQL only: Single quotation marks (’ ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

A batch can contain these *dml_statements*:  
- **INSERT**  
- **UPDATE**  
- **DELETE**

**UNLOGGED | COUNTER**

If **UNLOGGED** is not specified, the batch is logged. If multiple partitions are involved, batches are logged by default. A logged batch ensures that all or none of the batch operations succeed (atomicity). First the serialized batch is written to the **batchlog system table** which consumes the serialized batch as blob data. After a successful write, the rows are persisted (or hinted) and the batchlog data is removed. Logging incurs a performance penalty, the batchlog is written to two other nodes. Options for thresholds, **warning about or failure due to batch size**, are available.

**UNLOGGED** runs the batch without logging penalties. Unlogged batching **issues a warning** when 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), there is 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.

Use the **COUNTER** option for batched counter updates. Unlike other updates, counter updates are not idempotent.

**USING TIMESTAMPS**

Sets the write time for transactions executed in a **BATCH**.

**USING TIMESTAMP** does not support LWT (lightweight transactions), such as DML statements that have an **IF NOT EXISTS** clause.

By default, the database 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 [epoch_microseconds]**. When the time is not specified, the database inserts the current time.

To use the 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 ;
```

For 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);
```

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.

```
0 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
0 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 ( 
```
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 DataStax Distribution of Apache 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**

Defines a user-defined aggregate. An 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 single value, for example average or standard deviation.

**Synopsis**

```sql
CREATE [ OR REPLACE ] AGGREGATE [ IF NOT EXISTS ] [ keyspace_name. ] aggregate_name (cql_type)
SPFUNC udf_name
STYPE cql_type
FINALFUNC udf_name
INITCOND init_value
[ DETERMINISTIC ] ;
```

**Table 42: Legend**

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>@table=</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets ( [ ] ) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>( )</td>
<td>Group. Parentheses ( ( ) ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation ( ‘ ’ ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1, datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>-- --</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ‘ ’ ) surround an entire XML schema declaration.</td>
</tr>
</tbody>
</table>
CQL reference

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>@xml_entity='xml_entity_type'</code></td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

**OR REPLACE**

Overwrites existing aggregate (with the same name). When OR REPLACE is not specified the operations fails if an aggregate with the same name already exists.

**IF NOT EXISTS**

Creates an aggregate if it does not already exist, and displays no error if it does exist.

IF NOT EXISTS and OR REPLACE are not supported in the same statement.

**cql_type**

Specify the CQL type input.

Frozen collections are not supported.

**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** `init_value`

Define the initial condition, values, of the first parameter in the SFUNC. Set to null when no value defined.

**DETERMINISTIC**

Always returns the same output for a certain input. Requires an initial condition and returns a single value.

Default: false (non-deterministic).

**Examples**

Create an aggregate that calculates average in the cycling keyspace.

1. Set up a test table with data:

```sql
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_title  
) VALUES (  
    'UnitedHealthCare Pro Cycling Womens Team','Katie HALL',11449,  
    'Amgen Tour of California Women''s Race presented by SRAM - Stage 1 - Lake Tahoe > Lake Tahoe'  
);  

INSERT INTO cycling.team_average (  
    team_name, cyclist_name, cyclist_time_sec, race_title  
) VALUES (  
    'UnitedHealthCare Pro Cycling Womens Team','Linda VILLUMSEN',11485,  
    'Amgen Tour of California Women''s Race presented by SRAM - Stage 1 - Lake Tahoe > Lake Tahoe'  
);```

Page 218  
CQL for the DataStax Distribution of Apache Cassandra ™ 3.11 Latest DDAC  
patch: 5.1.17
INSERT INTO cycling.team_average (team_name, cyclist_name, cyclist_time_sec, race_title) VALUES ('UnitedHealthCare Pro Cycling Womens Team', 'Hannah BARNES', 11490, 'Amgen Tour of California Women''s Race presented by SRAM - Stage 1 - Lake Tahoe > Lake Tahoe');

INSERT INTO cycling.team_average (team_name, cyclist_name, cyclist_time_sec, race_title) VALUES ('Velocio-SRAM', 'Alena AMIALIUSIK', 11451, 'Amgen Tour of California Women''s Race presented by SRAM - Stage 1 - Lake Tahoe > Lake Tahoe');

INSERT INTO cycling.team_average (team_name, cyclist_name, cyclist_time_sec, race_title) VALUES ('Velocio-SRAM', 'Trixi WORRACK', 11453, 'Amgen Tour of California Women''s Race presented by SRAM - Stage 1 - Lake Tahoe > Lake Tahoe');

INSERT INTO cycling.team_average (team_name, cyclist_name, cyclist_time_sec, race_title) VALUES ('TWENTY16 presented by Sho-Air', 'Lauren KOMANSKI', 11451, 'Amgen Tour of California Women''s Race presented by SRAM - Stage 1 - Lake Tahoe > Lake Tahoe');

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

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);

INSERT INTO cycling.test_avg (id, state, val) VALUES (1, (6, 9949), 51, 5);
3. Create a function that divides the total value for the selected column by the number of records.

```java
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:

```sql
CREATE OR REPLACE AGGREGATE cycling.average (int
) SFUNC avgState
STYPE tuple<int,bigint>
FINALFUNC avgFinal
INITCOND (0, 0)
;
```

5. Test the function using a select statement:

```sql
SELECT cycling.average(cyclist_time_sec)
FROM cycling.team_average
WHERE team_name = 'UnitedHealthCare Pro Cycling Womens Team'
```

The first value was incremented by one and the second value is the result of the initial state value and `val`.

<table>
<thead>
<tr>
<th>state</th>
<th>cycling.avgstate(state, val)</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>(6, 9949)</td>
<td>(7, 10000)</td>
<td>51</td>
</tr>
<tr>
<td>(79, 10000)</td>
<td>(80, 19999)</td>
<td>9999</td>
</tr>
</tbody>
</table>
CREATE INDEX

Define a new index on a single column of a table. If the column already contains data, it is indexed during the execution of this statement. Once an index has been created, it is automatically updated when data in the column changes. DataStax Distribution of Apache Cassandra™ (DDAC) supports creating an index on most columns, including the partition and cluster columns of a PRIMARY KEY, collections, and static columns. Indexing can impact performance. Before creating an index, be aware of when and when not to create an index.

Indexing counter columns is not supported. For maps, index either the key or the value.

Synopsis

```
CREATE INDEX [ IF NOT EXISTS ] index_name
ON [keyspace_name.]table_name
(( ( KEYS | FULL ) column_name)
{ENTRIES column_name});
```

Table 43: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italics</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>(</td>
<td>Optional. Square brackets ( () ) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>)</td>
<td>Group. Parentheses ( () ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td>Or. A vertical bar (</td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation ( ' ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>( key : value )</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>[--]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ' ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

**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, you 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

You can create an index on map collection keys. If an index of the map values of the collection exists, drop that index before creating an index on the map collection keys.

To index map keys, you use the KEYS keyword and map name in nested parentheses. For example, index the collection keys, the timestamps, in the todo map in the users table:

```
CREATE INDEX todo_dates ON users (KEYS(todo));
```

To query the table, you can use CONTAINS KEY in WHERE clauses.

Creating an index on the map entries

You can 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
You can 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`.

```cql
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 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.

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 file.

**Synopsis**

```cql
CREATE [ OR REPLACE ] FUNCTION [ IF NOT EXISTS ] [ keyspace_name.]function_name (argument_list
[ , ... ]) ( CALLED | RETURNS NULL ) ON NULL INPUT RETURNS cql_data_type
[ DETERMINISTIC ]
[ MONOTONIC [ ON argument_name ] ]
LANGUAGE language_name AS 'code_block' ;
```

**Table 44: Legend**

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td><em>Italics</em></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[]</td>
<td>Optional. Square brackets ([ ]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>()</td>
<td>Group. Parentheses ( ( ) ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Or. A vertical bar (</td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>&quot;literal string&quot;</td>
<td>Single quotation ( ' ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>( key : value )</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>[--]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ' ) surround an entire XML schema declaration.</td>
</tr>
</tbody>
</table>
**Syntax conventions**

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>@xml_entity='xml_entity_type'</code></td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

**CREATE function_name**

Creates a new function and errors if it already exists, use with IF NOT EXITS to suppress error.

**OR REPLACE**

Creates a new function or overwrites it if one with the same name already exists.

**IF NOT EXITS**

Suppresses the error message if a function with the same name already exists, and performs no operation.

**var_name var_type**

The variable name followed by the CQL data type; these arguments are passed from request to the code block for execution. Use of literals or terms and prepared statement placeholders is also supported. Specify multiple arguments in a comma separated list. 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.

**DETERMINISTIC**

Specify for functions that always returns the same output for a certain input. For example, toJson() is a deterministic function; now() and currentDate() are not. Default: false (non-deterministic).

**MONOTONIC [ ON argument_name ]**

All arguments or the specified argument are monotonic if they are either entirely non-increasing or non-decreasing.

**LANGUAGE language_name**

Supported types are Java and Javascript. See [User Defined Functions](User Defined Functions) to add support for additional JSR-223 compliant scripting languages, such as Python, Ruby, and Scala.

```sql
'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.

```sql
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.

```sql
CREATE OR REPLACE FUNCTION 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:

```cql
SELECT left(firstname, 1), lastname
FROM cycling.cyclist_name;
```

<table>
<thead>
<tr>
<th>cycling.left(firstname, 1)</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>FRAME</td>
</tr>
<tr>
<td>M</td>
<td>MATTHEWS</td>
</tr>
<tr>
<td>P</td>
<td>TIRALONGO</td>
</tr>
<tr>
<td>S</td>
<td>KRUIKSWIJK</td>
</tr>
<tr>
<td>A</td>
<td>VAN DER BREGGEN</td>
</tr>
</tbody>
</table>

**CREATE KEYSPACE**

Creates a top-level keyspace. Configure the replica placement strategy, replication factor, and durable writes setting.

```cql
CREATE KEYSPACE [IF NOT EXISTS] keyspace_name
WITH REPLICATION = {replication_map}
[AND DURABLE_WRITES = true|false] ;
```

Use only replication strategy implementations bundled with DataStax Distribution of Apache Cassandra™ (DDAC).

**Table 45: Legend**

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italics</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets ([ ]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>( )</td>
<td>Group. Parentheses ( ( ) ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'Literal string'</td>
<td>Single quotation ( ' ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
</tbody>
</table>
### Syntax conventions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>( key : value )</code></td>
<td>Map collection. Braces ( {} ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td><code>&lt;datatype1,datatype2&gt;</code></td>
<td>Set, list, map, or tuple. Angle brackets (&lt; &gt;) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td><code>cql_statement;</code></td>
<td>End CQL statement. A semicolon (;) terminates all CQL statements.</td>
</tr>
<tr>
<td><code>[--]</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>' &lt;schema&gt; ... &lt;/schema&gt; '</code></td>
<td>Search CQL only: Single quotation marks ( ’ ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td><code>@xml_entity=&quot;xml_entity_type&quot;</code></td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

### keyspace_name

Maximum of 48 characters. Can contain alpha-numeric characters and underscores; only letters and numbers are supported as the first character. Unquoted names are forced to lowercase. 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_map

The replication map determines how many copies of the data are kept in a given datacenter. This setting impacts consistency, availability and request speed, for more details see replica placement strategy.

#### Table 46: Replication strategy class and factor settings

<table>
<thead>
<tr>
<th>Class</th>
<th>Replication factor</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'SimpleStrategy'</td>
<td>'replication_factor' : N</td>
<td></td>
<td>Assign the same replication factor to the entire cluster. Use for evaluation and single datacenter test and development environments only.</td>
</tr>
<tr>
<td>'NetworkTopologyStrategy'</td>
<td>'datacenter_name' : N</td>
<td></td>
<td>Assign replication factors to each datacenter in a comma-separated list. Use in production environments and multi-DC test and development environments. Datacenter names must match the snitch DC name; refer to Snitches for more details.</td>
</tr>
</tbody>
</table>

**Simple Topology syntax:**

```
'class' : 'SimpleStrategy', 'replication_factor' : N
```

**Network Topology syntax:**

```
'class' : 'NetworkTopologyStrategy',
    'dc1_name' : N [, ...]
```

### DURABLE_WRITES = true|false

Optional. (Not recommended), false bypasses the commit log when writing to the keyspace. Default value is true.

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 REPLICAATION = {
    '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 REPLICAATION = {
    'class' : 'NetworkTopologyStrategy',
    'datacenter1' : 1
};
```

datacenter1 is the default datacenter name. To display the datacenter 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 datacenters. The data center name must match the name configured in the snitch.

```
CREATE KEYSPACE "Cycling"
WITH REPLICAATION = {
    'class' : 'NetworkTopologyStrategy',
    'boston' : 3 , // Datacenter 1
    'seattle' : 2 , // Datacenter 2
    'tokyo'   : 2   // Datacenter 3
};
```

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 REPLICAATION = {
    'class' : 'NetworkTopologyStrategy',
    'datacenter1' : 3
};
```
AND DURABLE_WRITES = false;

CREATE MATERIALIZED VIEW
Optimizes read requests by allowing different partitioning and clustering columns than the base table and eliminates the need for individual write requests to multiple tables. When data is written to the base table, it is also automatically written to all associated materialized views.

- Use all base table primary keys in the materialized view.
- 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.]table_name WHERE column_name IS NOT NULL [ AND column_name IS NOT NULL ... ] [ AND relation [ AND ... ] ] PRIMARY KEY (column_list) [ WITH [ table_properties ] [ [ AND ] CLUSTERING ORDER BY (cluster_column_name order_option) ] ] ;

Table 47: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italic</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets ([ ]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>( )</td>
<td>Group. Parentheses (( )) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation ( ’ ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets (&lt; &gt;) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>[---]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ’ ) surround an entire XML schema declaration.</td>
</tr>
</tbody>
</table>
**Syntax conventions**

| `@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**

Optional. Suppresses the error message when attempting to create a materialized view that already exists. Use to continue executing commands, such as 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.

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.

**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.

**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.

**PRIMARY KEY (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 ... ] )
```

**table_properties**

Optional. Specify table properties if different than default. Separate table property definitions with an `AND`. See `table properties`.

The base table properties are not copied.

Change log, CDC, is not available for materialized views. Not all table properties are available when creating a materialized view; for example, `default_time_to_live` is not available.

**Examples**

This section shows example scenarios that illustrate the use of materialized views.

Basic example of a materialized view
This cyclist_base table is used in the first example scenario:

```
CREATE TABLE IF NOT EXISTS cycling.cyclist_base (
    cid UUID PRIMARY KEY,
    name text,
    age int,
    birthday date,
    country text
);
```

The following materialized view cyclist_by_age uses the base table cyclist_base. The WHERE clause ensures that only rows whose age and cid columns are non-NULL are added to the materialized view. In the materialized view, age is the partition key, and cid is the clustering column. In the base table, cid is the partition key.

```
CREATE MATERIALIZED VIEW cycling.cyclist_by_age AS
    SELECT age, cid, birthday, country, name
    FROM cycling.cyclist_base
    WHERE age IS NOT NULL
    AND cid IS NOT NULL
    PRIMARY KEY (age, cid)
    WITH CLUSTERING ORDER BY (cid ASC)
    AND caching = { 'keys' : 'ALL', 'rows_per_partition' : '100' }
    AND comment = 'Based on table cyclist';
```

The results of this query:

```
SELECT *
FROM cycling.cyclist_by_age;
```

are:

<table>
<thead>
<tr>
<th>age</th>
<th>cid</th>
<th>birthday</th>
<th>country</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47</td>
<td>1987-06-07</td>
<td>Netherlands</td>
<td>Steven KRUIKSWIJK</td>
</tr>
<tr>
<td>18</td>
<td>15a116fc-b833-4da6-ab9a-4a7775752836</td>
<td>1997-08-19</td>
<td>United States</td>
<td>Adrien COSTA</td>
</tr>
<tr>
<td>18</td>
<td>18f471bf-f631-4bc4-a9a2-d6f6cf5ea503</td>
<td>1997-03-29</td>
<td>Netherlands</td>
<td>Bram WELTEN</td>
</tr>
<tr>
<td>18</td>
<td>ffdfa2a7-5fc6-49a7-bf7e-3fcdcf7156</td>
<td>1997-02-08</td>
<td>Netherlands</td>
<td>Pascal EENKHOORN</td>
</tr>
<tr>
<td>22</td>
<td>e7ae5cf3-d358-4d99-b900-85902fda9bb0</td>
<td>1993-06-18</td>
<td>New Zealand</td>
<td>Alex FRAME</td>
</tr>
<tr>
<td>27</td>
<td>c99c484e-5e4a-4542-8203-8d047a01b8a8</td>
<td>1987-09-04</td>
<td>Brazil</td>
<td>Cristian EGIDIO</td>
</tr>
<tr>
<td>27</td>
<td>d1aad83b-be60-47a4-b6d6-e09b8da0d97b</td>
<td>1987-09-04</td>
<td>Germany</td>
<td>Johannes HEIDER</td>
</tr>
<tr>
<td>20</td>
<td>6d2e511f-00a1-4d5a-a976-ba359cab7300e</td>
<td>1994-09-04</td>
<td>Denmark</td>
<td>Joakim BUKDAL</td>
</tr>
<tr>
<td>38</td>
<td>220844bf-4860-49d6-9a4b-6b553a79cbcfb</td>
<td>1977-07-08</td>
<td>Italy</td>
<td>Paolo TIRALONGO</td>
</tr>
</tbody>
</table>

(9 rows)

Using a materialized view to perform queries that are not possible on a base table
The following scenario shows how to use a materialized view to perform queries that are not possible on a base table unless ALLOW FILTERING is used. ALLOW FILTERING is not recommended because of the performance degradation. This table stores the cycling team mechanic information:

```sql
CREATE TABLE IF NOT EXISTS cycling.mechanic {
    emp_id int,
    dept_id int,
    name text,
    age int,
    birthdate date,
    PRIMARY KEY (emp_id, dept_id)
};
```

The table contains these rows:

<table>
<thead>
<tr>
<th>emp_id</th>
<th>dept_id</th>
<th>age</th>
<th>birthdate</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
<td>25</td>
<td>1996-10-04</td>
<td>Lisa SMITH</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>21</td>
<td>1992-06-18</td>
<td>Fred GREEN</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>22</td>
<td>1993-01-15</td>
<td>John SMITH</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>24</td>
<td>1995-08-19</td>
<td>Jack JONES</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>23</td>
<td>1994-02-07</td>
<td>Jane DOE</td>
</tr>
</tbody>
</table>

This materialized view selects the columns from the previous table and contains a different primary key from the table:

```sql
CREATE MATERIALIZED VIEW cycling.mechanic_view AS
    SELECT emp_id, dept_id, name, age, birthdate
    FROM cycling.mechanic
    WHERE emp_id IS NOT NULL
    AND dept_id IS NOT NULL
    AND name IS NOT NULL
    AND age IS NOT NULL
    AND birthdate IS NOT NULL
    PRIMARY KEY (age, emp_id, dept_id);
```

This query retrieves the rows where the age is 21:

```sql
SELECT *
FROM cycling.mechanic_view
WHERE age = 21;
```

The previous query cannot be run on the base table without ALLOW FILTERING. The output from the previous query is as follows:

<table>
<thead>
<tr>
<th>age</th>
<th>emp_id</th>
<th>dept_id</th>
<th>birthdate</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>1</td>
<td>1</td>
<td>1992-06-18</td>
<td>Fred GREEN</td>
</tr>
</tbody>
</table>

(1 row)

**CREATE ROLE**

Creates a cluster wide database object used for 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.
CQL reference

- Create login accounts for internal authentication. (Not recommended for production environments.)

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 ) ]
[ [ AND ] LOGIN = ( true | false ) ]
[ [ AND ] PASSWORD = 'role_password' ]
```

Table 48: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italics</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets [ ] surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>( )</td>
<td>Group. Parentheses ( ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'Literal string'</td>
<td>Single quotation ( ' ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>( key : value )</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>[--]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt;'</td>
<td>Search CQL only: Single quotation marks ( ' ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

role_name

Use a unique name for the role. DataStax Distribution of Apache 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. 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 = 'All4One2day!'
   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.
   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;
```

<table>
<thead>
<tr>
<th>role</th>
<th>username</th>
<th>resource</th>
<th>permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>cycling_admin</td>
<td>cycling_admin</td>
<td>&lt;keyspace cycling&gt;</td>
<td>CREATE</td>
</tr>
<tr>
<td>cycling_admin</td>
<td>cycling_admin</td>
<td>&lt;keyspace cycling&gt;</td>
<td>ALTER</td>
</tr>
<tr>
<td>cycling_admin</td>
<td>cycling_admin</td>
<td>&lt;keyspace cycling&gt;</td>
<td>DROP</td>
</tr>
<tr>
<td>cycling_admin</td>
<td>cycling_admin</td>
<td>&lt;keyspace cycling&gt;</td>
<td>SELECT</td>
</tr>
<tr>
<td>cycling_admin</td>
<td>cycling_admin</td>
<td>&lt;keyspace cycling&gt;</td>
<td>MODIFY</td>
</tr>
<tr>
<td>cycling_admin</td>
<td>cycling_admin</td>
<td>&lt;keyspace cycling&gt;</td>
<td>AUTHORIZE</td>
</tr>
<tr>
<td>cycling_admin</td>
<td>cycling_admin</td>
<td>&lt;all roles&gt;</td>
<td>AUTHORIZE</td>
</tr>
</tbody>
</table>

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 TABLE**

Creates a new table in the selected keyspace. Use IF NOT EXISTS suppress the error message if the table already exists, no table is created. 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.

Tables support a single counter column.

**Synopsis**

```
CREATE TABLE [ IF NOT EXISTS ] [keyspace_name.]table_name
  ( column_definition [, ... ] | PRIMARY KEY (column_list) )
  [ WITH [table_options] ]
  [ [ AND ] CLUSTERING ORDER BY [clustering_column_name order] ]
  [ [ AND ] ID = 'table_hash_tag' ]
  [ [ AND ] COMPACT STORAGE ] ;
```

**Table 49: Legend**

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italic</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>Syntax conventions</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets ([ ]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>( )</td>
<td>Group. Parentheses (( )) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>' literal string '</td>
<td>Single quotation ( ' ) marks surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>( key : value )</td>
<td>Map collection. Braces ({ }) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets (&lt; &gt;) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon (;) terminates all CQL statements.</td>
</tr>
<tr>
<td>[--]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt;'</td>
<td>Search CQL only: Single quotation marks ( ' ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>&amp;xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

column_definition

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] [ , ... ]

- 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.

column_name

Use a unique name for each column in table. To preserve case or use special characters, enclose the name in double-quotes.

cql_type_definition

Defines the type of data allowed in the column, see CQL data type or a user-defined type.

STATIC

Optional, the column has a single value.

PRIMARY 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.

PRIMARY KEY (column_list)

Uniquely identifies rows, determines storage partitions, and data ordering (clustering) within the partition.
Primary keys cannot have the data type: counter, non-frozen collection, or static.

column_list
Defines a partition and clustering columns, which affects how the data is stored.

• Compound primary key: the first column is the partition key, and the additional columns are clustering keys. Syntax: `PRIMARY KEY (partition_column_name, clustering_column_name [, ...])`

• Composite partition key: Multiple columns in the partition key. Enclose the partition key columns in parenthesis. Syntax: `PRIMARY KEY ((partition_column_name[, ...]), clustering_column_name [, ...])`

table_options
Tunes data handling, including I/O operations, compression, and compaction. Table property options use the following syntax:

• Single values: `option_name = 'value'`

• Multiple values: `option_name = { 'subproperty' : 'value' [, ...] } [AND ...]`

Simple JSON format, key-value pairs in a comma-separated list enclosed by curly brackets.

When no value is specified, the default is used.

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 = N**
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. Values range from 0 to 1.0, where: 0 is the minimum value use to enable the largest possible bloom filter (uses the most memory) and 1.0 is the maximum value disabling the bloom filter.

Recommended setting: 0.1. A higher value yields diminishing returns.

Default: `bloom_filter_fp_chance = '0.01'`

**caching = { 'keys' : 'value', 'rows_per_partition' : 'value' }**
Optimizes the use of cache memory without manual tuning. Weighs the cached data by size and access frequency. Coordinate this setting with the global caching properties in the cassandra.yaml file.

Valid values:

• `ALL`— all primary keys or rows

• `NONE`— no primary keys or rows

• `N` (rows per partition only) — specify a whole number

Default: `{ 'keys': 'ALL', 'rows_per_partition': 'NONE' }

cdc = TRUE | FALSE
Creates a Change Data Capture (CDC) log on the table.

Valid values:

• `TRUE`— create CDC log
FALSE: do not create CDC log

comments = 'some text that describes the table etc'

Provide documentation on the table.

Enter a description of the types of queries the table was designed to satisfy.

dclocal_read_repair_chance

Probability that a successful read operation triggers a read repair, between 0 and 1. Unlike the repair controlled by read_repair_chance, this repair is limited to replicas in the same DC as the coordinator.

Default value: 0.01

In DataStax Distribution of Apache Cassandra™ (DDAC), dclocal_read_repair_chance is deprecated and should be set to 0.0.

default_time_to_live

TTL (Time To Live) in seconds, where zero is disabled. The maximum configurable value is 630720000 (20 years). Beginning in 2018, the expiration timestamp can exceed the maximum value supported by the storage engine, see the warning below. 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).

The database storage engine can only encode TTL timestamps through January 19 2038 03:14:07 UTC due to the Year 2038 problem. The TTL date overflow policy determines whether requests with expiration timestamps later than the maximum date are rejected or inserted. See Dcassandra.expiration_date_overflow_policy=POLICY.

gc_grace_seconds

Seconds after data is marked with a tombstone (deletion marker) before it is eligible for garbage-collection. Default value: 864000. The default value allows time for the database to maximize consistency prior to deletion.

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. The database 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, the database 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 property 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.

When memtable_flush_period_in_ms=0, the memtable will flush when:

• the flush threshold is met
• on shutdown
• on nodetool flush
• when commitlogs get full
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 the database 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
In Cassandra, read_repair_chance is deprecated and should be set to 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'.

Configures rapid read protection. Normal read requests are sent to just enough replica nodes to satisfy the consistency level. In rapid read protection, extra read requests are sent 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.
- X percentile: Track each table's typical read latency (in milliseconds). Coordinator node retrieves the typical latency time of the table being read and calculates 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.
- N ms: 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.

compression = { compression_map }

Configure the compression_map by specifying the compression algorithm class followed by the subproperties in simple JSON format.

Implement custom compression classes using the org.apache.cassandra.io.compress.ICompressor interface.

```json
compression = {
    'class': 'compression_algorithm_name',
    'chunk_length_kb': 'value',
    'crc_check_chance': 'value',
    'sstable_compression': ''
}
```

class
Sets the compressor name, DataStax Distribution of Apache Cassandra™ (DDAC) provides the following built-in classes:

- LZ4Compressor
- SnappyCompressor
- DeflateCompressor

Use only compression implementations bundled with DDAC.

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.

Default: LZ4Compressor.

chunk_length_in_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: 64.

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.

**compaction = {compaction_map}**

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',...] }
```

For more guidance, see When to Use Leveled Compaction, the Leveled Compaction blog, and How data is maintained?

DataStax Distribution of Apache Cassandra™ (DDAC) provides the following compaction classes, each class has different subproperties:

- SizeTieredCompactionStrategy (STCS)
- TimeWindowCompactionStrategy (TWCS)
- LeveledCompactionStrategy (LCS)
- DateTieredCompactionStrategy (deprecated) (deprecated)

compaction_strategy_name

SizeTieredCompactionStrategy (STCS)

Triggers a minor compaction when table meets the min_threshold. Minor compactions do not involve all the tables in a keyspace. See SizeTieredCompactionStrategySizeTieredCompactionStrategy.
Use only compaction implementations bundled with DDAC.

Default compaction strategy.

**Table 50: subproperties**

<table>
<thead>
<tr>
<th>Subproperty</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bucket_high</td>
<td>1.5</td>
<td>Size-tiered compaction merges sets of SSTables that are approximately the same size. The database compares each SSTable size to the average of all SSTable sizes for this table on the node. It merges SSTables whose size in KB are within (\text{average-size} \times \text{bucket_low}) and (\text{average-size} \times \text{bucket_high}).</td>
</tr>
<tr>
<td>bucket_low</td>
<td>0.5</td>
<td>See bucket_high.</td>
</tr>
<tr>
<td>enabled</td>
<td>true</td>
<td>Enables background compaction.</td>
</tr>
<tr>
<td>log_all</td>
<td>false</td>
<td>Activates advanced logging for the entire cluster.</td>
</tr>
<tr>
<td>max_threshold</td>
<td>32</td>
<td>The maximum number of SSTables to allow in a minor compaction.</td>
</tr>
<tr>
<td>min_threshold</td>
<td>4</td>
<td>The minimum number of SSTables to trigger a minor compaction.</td>
</tr>
<tr>
<td>min_sstable_size</td>
<td>50 MB</td>
<td>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.</td>
</tr>
<tr>
<td>only_purge_repaired_tombstones</td>
<td>false</td>
<td>Setting to true allows purging tombstones only from repaired SSTables. This prevents data from resurrecting when repair is not run within gc_grace_seconds. If you do not run repair for a long time, the database keeps all tombstones, which can cause problems.</td>
</tr>
<tr>
<td>tombstone_compaction_interval</td>
<td>86400</td>
<td>Number of seconds before compaction can run on an SSTable after it is created. An SSTable is eligible for compaction when it exceeds the tombstone_threshold. Because it might not be possible to drop tombstones when doing a single SSTable compaction, and since the compaction is triggered base on an estimated tombstone ratio, this setting makes the minimum interval between two single SSTable compactions tunable to prevent an SSTable from being constantly re-compacted.</td>
</tr>
<tr>
<td>tombstone_threshold</td>
<td>0.2</td>
<td>The ratio of garbage-collectable tombstones to all contained columns. If the ratio exceeds this limit, compactions starts only on that table to purge the tombstones.</td>
</tr>
<tr>
<td>unchecked_tombstone_compaction</td>
<td>false</td>
<td>Setting to true allows tombstone compaction to run 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.</td>
</tr>
</tbody>
</table>

The cold_reads_to_omit property for SizeTieredCompactionStrategy is no longer supported.

**DateTieredCompactionStrategy (deprecated)**

Use TimeWindowCompactionStrategy (TWCS) instead.

Stores data written within a certain period of time in the same SSTable.

**Table 51: Subproperties**

<table>
<thead>
<tr>
<th>Subproperty</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_time_seconds</td>
<td>3600</td>
<td>The size of the first time window.</td>
</tr>
<tr>
<td>enabled</td>
<td>true</td>
<td>Enables background compaction.</td>
</tr>
<tr>
<td>log_all</td>
<td>false</td>
<td>Setting to true activates advanced logging for the entire cluster.</td>
</tr>
</tbody>
</table>
CQL reference

<table>
<thead>
<tr>
<th>Subproperty</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>max_sstable_age_days</td>
<td>1000</td>
<td>Cassandra does not compact SSTables if its most recent data is older than this property. Fractional days can be set.</td>
</tr>
<tr>
<td>max_window_size_seconds</td>
<td>86400</td>
<td>The maximum window size in seconds.</td>
</tr>
<tr>
<td>max_threshold</td>
<td>32</td>
<td>The maximum number of SSTables allowed in a minor compaction.</td>
</tr>
<tr>
<td>min_threshold</td>
<td>1</td>
<td>The minimum number of SSTables that trigger a minor compaction.</td>
</tr>
<tr>
<td>timestamp_resolution</td>
<td>microseconds</td>
<td>Units, microseconds, or milliseconds, to match the timestamp of inserted data.</td>
</tr>
<tr>
<td>tombstone_compaction_interval</td>
<td>86400</td>
<td>The minimum number of seconds after which an SSTable is created before the database considers the SSTable for tombstone compaction. An SSTable is eligible for tombstone compaction if the table exceeds the tombstone_threshold ratio.</td>
</tr>
<tr>
<td>tombstone_threshold</td>
<td>0.2</td>
<td>The ratio of garbage-collectable tombstones to all contained columns. If the ratio exceeds this limit, compactions starts only on that table to purge the tombstones.</td>
</tr>
<tr>
<td>unchecked_tombstone_compaction</td>
<td>false</td>
<td>Setting to true allows tombstone compaction to run 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.</td>
</tr>
</tbody>
</table>

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.

All of the subproperties for STCS are also valid for TWCS.

### Table 52: Subproperties

<table>
<thead>
<tr>
<th>Compaction Subproperties</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>compaction_window_unit</td>
<td>milliseconds</td>
<td>Time unit used to define the bucket size, milliseconds, seconds, hours, and so on.</td>
</tr>
<tr>
<td>compaction_window_size</td>
<td></td>
<td>Units per bucket.</td>
</tr>
<tr>
<td>log_all</td>
<td>false</td>
<td>Setting this to true activates advanced logging for the entire cluster.</td>
</tr>
</tbody>
</table>

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).

### Table 53: Subproperties

<table>
<thead>
<tr>
<th>Subproperties</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enabled</td>
<td>true</td>
<td>Enables background compaction.</td>
</tr>
<tr>
<td>log_all</td>
<td>false</td>
<td>Setting to true activates advanced logging for the entire cluster.</td>
</tr>
</tbody>
</table>
### CQL reference

#### Subproperties

<table>
<thead>
<tr>
<th>Subproperties</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sstable_size_in_mb</td>
<td>160MB</td>
<td>The target size for SSTables that use the Leveled Compaction Strategy. Although SSTable sizes should be less or equal to <code>sstable_size_in_mb</code>, it is possible that compaction may produce a larger SSTable during compaction. This occurs when data for a given partition key is exceptionally large. The Cassandra database does not split the data into two SSTables.</td>
</tr>
<tr>
<td>tombstone_compaction_interval</td>
<td>864000 (one day)</td>
<td>The minimum number of seconds after an SSTable is created before SSTable tombstone compaction. Tombstone compaction begins when the SSTable's <code>tombstone_threshold</code> exceeds the set value.</td>
</tr>
<tr>
<td>tombstone_threshold</td>
<td>0.2</td>
<td>The ratio of garbage-collectable tombstones to all contained columns. If the ratio exceeds this limit, compaction starts only on that table to purge the tombstones.</td>
</tr>
<tr>
<td>unchecked_tombstone_compaction</td>
<td>false</td>
<td>Setting to <code>true</code> allows tombstone compaction to run 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.</td>
</tr>
</tbody>
</table>

#### 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

**COMPACT STORAGE**

Use `COMPACT STORAGE` to store data in the legacy (Thrift) storage engine format to conserve disk space.

For DataStax Distribution of Apache Cassandra™ (DDAC), 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.

#### Examples

Create a table that has a frozen user-defined type.

```cql
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. In DataStax Distribution of Apache Cassandra™ (DDAC), UDTs can be created unfrozen if only non-collection fields are used in the user-defined type creation. If the table is created with an unfrozen UDT, then individual field values can be updated and deleted.

Create the cyclist_name table with UUID as the primary key:

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

Creating a compound primary key
Create the cyclist category table and store the data in reverse order:

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

Creating a composite partition key

Create a table that is optimized for query by cyclist rank by year:

```cql
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) );
```

Setting caching

The database 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:

```cql
ALTER MATERIALIZED VIEW cycling.cyclist_by_age  
WITH caching = {  
'keys' : 'ALL',  
'rows_per_partition' : 'ALL' }; 
```

Creating a table with a CDC log:

Create a change log for the cyclist name table:

```cql
CREATE TABLE cycling.cyclist_name  
WITH cdc = TRUE;
```

Adding a comment

For example, note the base table for the materialized view:

```cql
ALTER MATERIALIZED VIEW cycling.cyclist_by_age  
WITH comment = "Basetable: cyclist_mv";
```

Change the speculative retries

Modify the user table to use 10 milliseconds:

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

Modify the user table to 99 percent:

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

Enabling and disabling background compaction
The following example sets the `enable` property to disable background compaction:

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

Disabling background compaction can be harmful: without it, the database 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.

Reading extended compaction logs

Set the `log_all` subproperty to `true` to collect in-depth information about compaction activity on a node in a dedicated log file.

If you enable extended logging for any table on any node, it is enabled for all tables on all nodes in the cluster.

When extended compaction is enabled, the database creates a file named `compaction-%d.log` (where `%d` is a sequential number) in `home/logs`.

The compaction logging service logs detailed information about four types of compaction events:

- **type: `enable`**
  Lists SSTables that have been flushed previously

```json
{"type":"enable","keyspace":"test","table":"t","time":1470071098866,"strategies": [{ "strategyId":"0","type":"LeveledCompactionStrategy","tables": [],"repaired":true,"folders": ["/home/carl/oss/cassandra/bin/../../../data/data"]},
 { "strategyId":"1","type":"LeveledCompactionStrategy","tables": [],"repaired":false,"folders": ["/home/carl/oss/cassandra/bin/../../../data/data"] }
 ]
 }
```

- **type: `flush`**
  Logs a flush event from a memtable to an SSTable on disk, including the CompactionStrategy for each table.

```json
{"type":"flush","keyspace":"test","table":"t","time":1470083335639,"tables": [{ "strategyId":"1","table": { "generation":1,"version":"mb","size":106846362,"details": {"level":0,"min_token":"-9221834874718566760","max_token":"9221396997139245178"} } }
 ]
 }
```

- **type: `compaction`**
  Logs a compaction event.

```json
{"type":"compaction","keyspace":"test","table":"t","time":1470083660267,"start":"1470083660188","end":"1470083660267","input": [{ "strategyId":"1","table": { "generation":1372,"version":"mb","size":1064979,"details": {"level":1,"min_token":"7199305267944662291","max_token":"732343447996777057"} } }
 ],"output":
 ```
Lists the number of pending tasks for a compaction strategy

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

Using compaction storage

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;

Creating the user table from the commit log

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.

Synopsis

CREATE TRIGGER trigger_name  
ON [keyspace_name.]table_name
You must 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. DSE supports lightweight transactions for creating a trigger. Attempting to create an existing trigger returns an error unless the IF NOT EXISTS option is used. If the option is used, the statement is a no-op if the table already exists.

**CREATE TYPE**

Creates a custom data type in the keyspace that contains one or more fields of related information, such as address (street, city, state, and postal code).

User-defined types (UDT) cannot contain counter fields.

**Synopsis**

```
CREATE TYPE [ IF NOT EXISTS ] [keyspace_name.]type_name
```
Table 55: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italic</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets ([ ]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>( )</td>
<td>Group. Parentheses ( ( ) ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>’literal string’</td>
<td>Single quotation ( ’ ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>[--]</td>
<td>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.</td>
</tr>
<tr>
<td>’ &lt;schema&gt; ... &lt;/schema&gt; ’</td>
<td>Search CQL only: Single quotation marks ( ’ ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity=’xml_entity_type’</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

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.

Example

This example creates a user-defined type cycling.basic_info that consists of personal data about an individual cyclist.

```cql
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 USER (Deprecated)**

`CREATE USER` is supported only for backwards compatibility. Authentication and authorization for DataStax Distribution of Apache Cassandra™ (DDAC) are based on roles. 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 56: Legend**

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UPPERCASE</strong></td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td><em>italics</em></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets ( [ ] ) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>{ }</td>
<td>Group. Parentheses ( { } ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'Literal string'</td>
<td>Single quotation ( ' ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>[--]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ' ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity=&quot;xml_entity_type&quot;</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>
Examples

Creating internal user accounts

Use \texttt{WITH PASSWORD} to create a user account for internal authentication. Enclose the password in single quotation marks.

\begin{verbatim}
CREATE USER spillman WITH PASSWORD 'Niner27';
CREATE USER akers WITH PASSWORD 'Niner2' SUPERUSER;
CREATE USER boone WITH PASSWORD 'Niner75' NOSUPERUSER;
\end{verbatim}

If internal authentication has not been set up, \texttt{WITH PASSWORD} is not required.

\begin{verbatim}
CREATE USER test NOSUPERUSER;
\end{verbatim}

Creating a user account conditionally

You can test that the user does not have an account before attempting to create one. Attempting to create an existing user results in an invalid query condition unless the \texttt{IF NOT EXISTS} option is used. If the option is used, the statement will be a no-op if the user exists.

1. Login with the default account (this account uses QUOROM, only use to create a new account once).

\begin{verbatim}
$ cqlsh -u cassandra -p cassandra
\end{verbatim}

The Cassandra and component versions display, and then the command prompt.

\begin{verbatim}
Connected to Testcluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.11.0.1805 | DSE 5.1.3 | CQL spec 3.4.4 | Native protocol v4]
Use HELP for help.
cqlsh>
\end{verbatim}

2. Create the new user:

\begin{verbatim}
CREATE USER newuser WITH PASSWORD 'password';
\end{verbatim}

\textbf{DELETE}

Removes data from one or more selected columns (data is replaced with null) or removes the entire row when no column is specified. Deletes data in each selected partition atomically and in isolation. Data is not removed from disk immediately; it is marked with a tombstone and then removed after the grace period.

Using delete can impact performance.

\textbf{Synopsis}

\begin{verbatim}
DELETE [ column_name [ term ] [ , ... ] ]
FROM [keyspace_name.]table_name
[ USING TIMESTAMP timestamp_value ]
WHERE PK_column_conditions
[ ( IF EXISTS | IF static_column_conditions ) ] ;
\end{verbatim}

\textbf{Table 57: Legend}

\begin{tabular}{|c|c|}
\hline
\textbf{Syntax conventions} & \textbf{Description} \\
\hline
UPPERCASE & Literal keyword. \\
\hline
\end{tabular}
## CQL reference

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italic</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets { [ ] } surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>( )</td>
<td>Group. Parentheses { ( ) } identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis { ... } indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation { ' } marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces { { } } enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets { &lt; &gt; } enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon { ; } terminates all CQL statements.</td>
</tr>
<tr>
<td>[ -- ]</td>
<td>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.</td>
</tr>
<tr>
<td>'{ &lt;schema&gt; ... &lt;/schema&gt;}'</td>
<td>Search CQL only: Single quotation marks { ' } surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

### 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, for:
- Lists specify the index number of the item, where 0 is the first.
- Maps specify the element key of the item.

### timestamp_value
Deletes values older than the timestamp_value.

### PK_column_conditions
Syntax to match PRIMARY KEY values. Separate multiple conditions with AND.
- 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. 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. The deleted values are set 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

<table>
<thead>
<tr>
<th>[applied]</th>
<th>firstname</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>Alex</td>
<td>FRAME</td>
</tr>
</tbody>
</table>

Conditionally deleting columns
You can 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:

<table>
<thead>
<tr>
<th>[applied]</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
</tr>
</tbody>
</table>

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:

```cql
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     | id |

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**.

```cql
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.

```cql
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:

```cql
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:

```cql
DELETE sponsorship ['sponsor_name'] FROM cycling.races
WHERE race_name = 'Criterium du Dauphine';
```

To delete an element from a list, specify the `column_name` followed by the list index position in square brackets:

```cql
DELETE categories[3] FROM cycling.cyclist_history
WHERE lastname = 'TIRALONGO';
```

To delete all elements from a set, specify the `column_name` by itself:

```cql
DELETE sponsorship FROM cycling.races
```
WHERE race_name = 'Criterium du Dauphine';

**DROP AGGREGATE**

Deletes a user-defined aggregate (UDA) from a keyspace.

**Synopsis**

```
DROP AGGREGATE [ IF EXISTS ] [keyspace_name.]aggregate_name [ (argument_name [ , ... ])];
```

**Examples**

Drop the avgState aggregate from the cycling keyspace.

```
DROP AGGREGATE IF EXISTS cycling.avgState;
```

**DROP FUNCTION**

Deletes a user-defined function from a keyspace.

Remove the function from any aggregates before dropping.
**Synopsis**

```
DROP FUNCTION [ IF EXISTS ] [keyspace_name.]
function_name [ (argument_name [ , ... ]) ] ;
```

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td><em>italics</em></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>()</td>
<td>Optional. Square brackets ( []) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>()</td>
<td>Group. Parentheses ( () ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>&quot;literal string&quot;</td>
<td>Single quotation ( ' ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1, datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>[---]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt;'</td>
<td>Search CQL only: Single quotation marks ( ' ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

**Examples**

Drops the UDF from the cycling keyspace.

```
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>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Syntax conventions</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>Lowercase</strong></td>
<td>Not literal.</td>
</tr>
<tr>
<td><em>italics</em></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[]</td>
<td>Optional. Square brackets ([ ]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>()</td>
<td>Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td>[]</td>
<td>Or. A vertical bar (</td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td><code>'literal string'</code></td>
<td>Single quotation (‘’) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>(key: value)</td>
<td>Map collection. Braces ({}) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets (&lt;&gt;) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon (;) terminates all CQL statements.</td>
</tr>
<tr>
<td>[---]</td>
<td>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.</td>
</tr>
<tr>
<td><code>'&lt;schema&gt; ... &lt;/schema&gt;'</code></td>
<td>Search CQL only: Single quotation marks (‘’) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

**Example**

Drop the index `ryear` from the `cycling.rank_by_year_and_name` table.

```sql
DROP INDEX cycling.ryear;
```

**DROP KEYSPACE**

Immediate, irreversible removal of the keyspace, including objects such as tables, functions, and data in the keyspace.

A snapshot is automatically created, before the keyspace is dropped; see **Restoring from a snapshot** for recovery details.

**Synopsis**

```sql
DROP KEYSPACE [ IF EXISTS ] keyspace_name ;
```

**Table 61: Legend**

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UPPERCASE</strong></td>
<td>Literal keyword.</td>
</tr>
<tr>
<td><strong>Lowercase</strong></td>
<td>Not literal.</td>
</tr>
<tr>
<td><em>italics</em></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
</tbody>
</table>
CQL for the DataStax Distribution of Apache Cassandra™ 3.11 Latest DDAC

Example

Drop the cycling keyspace:

```cql
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.

Drop all materialized views associated with a base table before dropping the table.

Synopsis

```cql
DROP MATERIALIZED VIEW [ IF EXISTS ] [keyspace_name.]view_name ;
```
Syntax conventions

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Or. A vertical bar (</td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>&quot;literal string&quot;</td>
<td>Single quotation ( ’ ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>( key : value )</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>[--]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ’ ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity=&quot;xml_entity_type&quot;</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

**IF EXISTS**

If the materialized view does not exist, the operation fails without an error. 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;
```

**DROP ROLE**

Removes an existing role. Enclose role names with special characters and capitalization in single quotation marks.

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 63: Legend**

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italic</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
</tbody>
</table>
### Syntax conventions

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>()</td>
<td>Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>&quot;literal string&quot;</td>
<td>Single quotation (‘’) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>( key : value )</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets (&lt;&gt;) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon (;) terminates all CQL statements.</td>
</tr>
<tr>
<td>--</td>
<td>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.</td>
</tr>
<tr>
<td>&quot; &lt;schema&gt; ... &lt;/schema&gt; &quot;</td>
<td>Search CQL only: Single quotation marks (‘’) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity=&quot;xml_entity_type&quot;</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

### 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.

Drop all materialized view associated with the table before dropping the table.

**Synopsis**

```
DROP TABLE [ IF EXISTS ] [keyspace_name.]table_name ;
```

### Table 64: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italics</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[]</td>
<td>Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>()</td>
<td>Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.</td>
</tr>
</tbody>
</table>
### Syntax conventions

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Or. A vertical bar (</td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ...) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation ( ' ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>[--]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ' ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

### 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_name.]table_name ;
```

### Table 65: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italic</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets ( [ ] ) surround optional command arguments. Do not type the square brackets.</td>
</tr>
</tbody>
</table>
**Syntax conventions**

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>Group. Parentheses ( ( ) ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation ( ' ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>( key : value )</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>[--]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ' ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>$xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

**DROP TYPE**

Immediately and irreversibly removes a UDT (user-defined type).

Dropping a user-defined type that is in use by a table or another type is not supported.

**Synopsis**

```sql
DROP TYPE [ IF EXISTS ] [keyspace_name]type_name;
```

**Table 66: Legend**

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italics</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>{}</td>
<td>Optional. Square brackets ( {} ) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>()</td>
<td>Group. Parentheses ( ( ) ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation ( ' ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>( key : value )</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
</tbody>
</table>
### Syntax conventions

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;datatype1, datatype2&gt;</code></td>
<td>Set, list, map, or tuple. Angle brackets (&lt; &gt;) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td><code>cql_statement;</code></td>
<td>End CQL statement. A semicolon (;) terminates all CQL statements.</td>
</tr>
<tr>
<td><code>[--]</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>'&lt;schema&gt; ... &lt;/schema&gt;'</code></td>
<td>Search CQL only: Single quotation marks (‘’) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td><code>@xml_entity='xml_entity_type'</code></td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

### 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)**

Removes a user.

```
DROP USER  
```

is supported for backwards compatibility. Authentication and authorization for DataStax Distribution of Apache Cassandra™ are based on ROLES, and use `DROP ROLE`.

### Synopsis

```
DROP USER [ IF EXISTS ] user_name ;
```

### Table 67: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italic</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[]</td>
<td>Optional. Square brackets ([]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>()</td>
<td>Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.</td>
</tr>
</tbody>
</table>
Syntax conventions

| | 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. |
| 'literal string' | Single quotation ( ‘ ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case. |
| ( 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. |

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.

Permissions apply immediately, even to active client sessions.

**Synopsis**

```
GRANT permission
[ ON object ]
TO role_name ;
```

Enclose the role name in single quotation marks if it contains special characters or capital letters.

To modify a base table that has a materialized view (MV) using an **INSERT** or **UPDATE** command if **access permissions are enabled**, a user must be granted **modify** or **all permissions** on the base table.
Table 68: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italic</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>()</td>
<td>Optional. Square brackets ( [ ] ) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>()</td>
<td>Group. Parentheses ( ( ) ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'Literal string'</td>
<td>Single quotation ( ‘ ’ ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>( key : value )</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1, datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>[---]</td>
<td>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.</td>
</tr>
<tr>
<td>'&lt; &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ‘ ’ ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

**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
- PROXY.EXECUTE
- PROXY.LOGIN
- SEARCH.ALTER
- SEARCH.COMMIT
CQL reference

- SEARCH.CREATE
- SEARCH.DROP
- SEARCH.REBUILD
- SEARCH.RELOAD
- SELECT

resource_name

The DataStax Distribution of Apache Cassandra™ database objects to which permissions are applied. Database resources have modeled hierarchy. Grant permissions on a resource higher in the chain to automatically grant that same permission on all resources lower down.

Not all privileges apply to every type of resource. For instance, `EXECUTE` is only relevant in the context of functions, MBeans, RPC, and authentication schemes. Attempting to grant privileges on a resource that the permission is not applicable results in an error.

Functions

- ALL FUNCTIONS
- ALL FUNCTIONS IN KEYSpace `keyspace_name`
- FUNCTION `keyspace_name.function_name`

Data

- ALL KEYSPACES
- TABLE `table_name`
- Rows `('filtering_data' ROWS IN `table_name')`
- Indexes (other than search indexes) belong to a table but permission cannot be directly assigned. ALTER permission on a table allows users to CREATE and DROP indexes.

- KEYSpace `keyspace_name`
- Types belong to a keyspace but permissions cannot be directly assigned. To manage types set the keyspace permissions to CREATE, DROP, or ALTER which also gives the user the same permissions for tables.

Search index

- ALL SEARCH INDICES
- SEARCH KEYSpace `keyspace_name`
- SEARCH INDICES `[`keyspace_name.`table_name`]

JMX

- ALL MBEANS
  # MBEAN `mbean_name`
  # MBEANS `pattern`

Roles

- ALL ROLES
- ROLE `role_name`
When using the `internal` Role Management mode nest roles using `GRANT role_name TO role_name` to give all the permissions of the first role in the statement to the second role. Use roles to create your own hierarchical permissions structures.

Remote procedure calls (RPC)
- `ALL OBJECTS`
- `OBJECT object_name`
- `METHOD`

Authentication schemes
- `ALL SCHEMES`
- `(LDAP | KERBEROS | INTERNAL) SCHEME`

Spark applications
Submit applications:
- `ANY WORKPOOL`
- `WORKPOOL datacenter_name`

Manage applications:
- `ANY SUBMISSION`
- `ANY SUBMISSION IN WORKPOOL datacenter_name`
- `SUBMISSION application_ID IN WORKPOOL datacenter_name`

Access control matrix
The following table shows the relationship between privileges and resources, and describes the resulting permissions.

<table>
<thead>
<tr>
<th>Resource type</th>
<th>Privilege</th>
<th>Resource</th>
<th>Permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td><code>ALL PERMISSIONS</code></td>
<td><code>resource_name</code></td>
<td>All operations that are applicable to the resource (listed below) and its ancestors, where resource name is listed below.</td>
</tr>
<tr>
<td>Data</td>
<td><code>ALTER</code></td>
<td><code>ALL KEYSPACES</code></td>
<td><code>ALTER KEYSPACE</code>, <code>ALTER TABLE</code>, <code>ALTER TYPE</code>, <code>RESTRICT ROW</code> in any keyspace.</td>
</tr>
<tr>
<td>Data</td>
<td><code>ALTER</code></td>
<td><code>KEYSPACE keyspace_name</code></td>
<td><code>ALTER KEYSPACE</code>, <code>ALTER TABLE</code>, <code>ALTER TYPE</code>, and <code>RESTRICT ROW</code> in specified keyspace.</td>
</tr>
<tr>
<td>Data</td>
<td><code>ALTER</code></td>
<td><code>TABLE table_name</code></td>
<td><code>ALTER TABLE</code> and <code>RESTRICT ROW</code> of specified table.</td>
</tr>
<tr>
<td>Data</td>
<td><code>CREATE</code></td>
<td><code>ALL KEYSPACES</code></td>
<td><code>CREATE KEYSPACE</code>, <code>CREATE TABLE</code>, <code>CREATE FUNCTIONS</code>, and <code>CREATE TYPE</code> in any keyspace.</td>
</tr>
<tr>
<td>Data</td>
<td><code>CREATE</code></td>
<td><code>KEYSPACE keyspace_name</code></td>
<td><code>CREATE TABLE</code> and <code>CREATE TYPE</code> in specified keyspace.</td>
</tr>
<tr>
<td>Data</td>
<td><code>DROP</code></td>
<td><code>ALL KEYSPACES</code></td>
<td><code>DROP KEYSPACE</code>, <code>DROP TABLE</code>, and <code>DROP TYPE</code> in any keyspace</td>
</tr>
<tr>
<td>Data</td>
<td><code>DROP</code></td>
<td><code>KEYSPACE keyspace_name</code></td>
<td><code>DROP TABLE</code> and <code>DROP TYPE</code> in specified keyspace</td>
</tr>
<tr>
<td>Data</td>
<td><code>DROP</code></td>
<td><code>TABLE table_name</code></td>
<td><code>DROP TABLE</code> specified.</td>
</tr>
<tr>
<td>Data</td>
<td><code>MODIFY</code></td>
<td><code>'filtering_data' ROWS IN table_name</code></td>
<td><code>MODIFY</code> on rows that exactly match the <code>'filtering_data'</code> in specified table.</td>
</tr>
<tr>
<td>Resource type</td>
<td>Privilege</td>
<td>Resource</td>
<td>Permissions</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>-----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Data</td>
<td>MODIFY</td>
<td>ALL KEYSPACES</td>
<td>INSERT, UPDATE, DELETE, and TRUNCATE on any table.</td>
</tr>
<tr>
<td>Data</td>
<td>MODIFY</td>
<td>KEYSPACE  <code>keyspace_name</code></td>
<td>INSERT, UPDATE, DELETE, and TRUNCATE on any table in specified keyspace.</td>
</tr>
<tr>
<td>Data</td>
<td>MODIFY</td>
<td>TABLE <code>table_name</code></td>
<td>INSERT, UPDATE, DELETE, and TRUNCATE on specified table.</td>
</tr>
<tr>
<td>Data</td>
<td>SELECT</td>
<td><code>filtering_data</code> ROWS IN <code>table_name</code></td>
<td>SELECT on rows that exactly match the <code>filtering_data</code> in specified table.</td>
</tr>
<tr>
<td>Data</td>
<td>SELECT</td>
<td>ALL KEYSPACES</td>
<td>SELECT on any table.</td>
</tr>
<tr>
<td>Data</td>
<td>SELECT</td>
<td>KEYSPACE  <code>keyspace_name</code></td>
<td>SELECT on any table in specified keyspace.</td>
</tr>
<tr>
<td>Data</td>
<td>SELECT</td>
<td>TABLE <code>table_name</code></td>
<td>SELECT on specified table.</td>
</tr>
<tr>
<td>Functions</td>
<td>ALTER</td>
<td>ALL FUNCTIONS</td>
<td>CREATE FUNCTION and CREATE AGGREGATE, also replace existing.</td>
</tr>
<tr>
<td>Functions</td>
<td>ALTER</td>
<td>ALL FUNCTIONS IN KEYSPACE <code>keyspace_name</code></td>
<td>CREATE FUNCTION and CREATE AGGREGATE, also replace existing in specified keyspace</td>
</tr>
<tr>
<td>Functions</td>
<td>ALTER</td>
<td>FUNCTION <code>function_name</code></td>
<td>CREATE FUNCTION and CREATE AGGREGATE, also replace existing.</td>
</tr>
<tr>
<td>Functions</td>
<td>CREATE</td>
<td>ALL FUNCTIONS</td>
<td>CREATE FUNCTION in any keyspace and CREATE AGGREGATE in any keyspace.</td>
</tr>
<tr>
<td>Functions</td>
<td>CREATE</td>
<td>ALL FUNCTIONS IN KEYSPACE <code>keyspace_name</code></td>
<td>CREATE FUNCTION and CREATE AGGREGATE in specified keyspace.</td>
</tr>
<tr>
<td>Functions</td>
<td>DROP</td>
<td>ALL FUNCTIONS</td>
<td>DROP FUNCTION and DROP AGGREGATE in any keyspace.</td>
</tr>
<tr>
<td>Functions</td>
<td>DROP</td>
<td>ALL FUNCTIONS IN KEYSPACE <code>keyspace_name</code></td>
<td>DROP FUNCTION and DROP AGGREGATE in specified keyspace.</td>
</tr>
<tr>
<td>Functions</td>
<td>DROP</td>
<td>FUNCTION <code>function_name</code></td>
<td>DROP FUNCTION specified function.</td>
</tr>
<tr>
<td>Functions</td>
<td>EXECUTE</td>
<td>ALL FUNCTIONS</td>
<td>SELECT, INSERT, and UPDATE using any function, and use of any function in CREATE AGGREGATE.</td>
</tr>
<tr>
<td>Functions</td>
<td>EXECUTE</td>
<td>ALL FUNCTIONS IN KEYSPACE <code>keyspace_name</code></td>
<td>SELECT, INSERT, and UPDATE using any function in a keyspace in CREATE AGGREGATE.</td>
</tr>
<tr>
<td>Functions</td>
<td>EXECUTE</td>
<td>FUNCTION <code>function_name</code></td>
<td>SELECT, INSERT, and UPDATE using specified function, and use of the function in CREATE AGGREGATE.</td>
</tr>
<tr>
<td>JMX</td>
<td>DESCRIBE</td>
<td>ALL MBEANS</td>
<td>Retrieve metadata about any mbean from the platform's MBeanServer.</td>
</tr>
<tr>
<td>JMX</td>
<td>DESCRIBE</td>
<td>MBEAN <code>mbean_name</code></td>
<td>Retrieve metadata about a named mbean from the platform's MBeanServer.</td>
</tr>
<tr>
<td>JMX</td>
<td>DESCRIBE</td>
<td>MBEANS <code>pattern</code></td>
<td>Retrieve metadata about any mbean matching a wildcard pattern from the platform's MBeanServer.</td>
</tr>
<tr>
<td>JMX</td>
<td>EXECUTE</td>
<td>ALL MBEANS</td>
<td>Execute operations on any mbean.</td>
</tr>
<tr>
<td>JMX</td>
<td>EXECUTE</td>
<td>MBEAN <code>mbean_name</code></td>
<td>Execute operations on named mbean.</td>
</tr>
<tr>
<td>JMX</td>
<td>EXECUTE</td>
<td>MBEANS <code>pattern</code></td>
<td>Execute operations on any mbean matching a wildcard pattern.</td>
</tr>
<tr>
<td>JMX</td>
<td>MODIFY</td>
<td>ALL MBEANS</td>
<td>Call setter methods on any mbean.</td>
</tr>
<tr>
<td>JMX</td>
<td>MODIFY</td>
<td>MBEAN <code>mbean_name</code></td>
<td>Call setter methods on named mbean.</td>
</tr>
<tr>
<td>Resource type</td>
<td>Privilege</td>
<td>Resource</td>
<td>Permissions</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------</td>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>JMX</td>
<td>MODIFY</td>
<td>MBEANS pattern</td>
<td>Call setter methods on any mbean matching a wildcard pattern.</td>
</tr>
<tr>
<td>JMX</td>
<td>SELECT</td>
<td>ALL MBEANS</td>
<td>Call getter methods on any mbean.</td>
</tr>
<tr>
<td>JMX</td>
<td>SELECT</td>
<td>MBEAN mbean_name</td>
<td>Call getter methods on named mbean.</td>
</tr>
<tr>
<td>JMX</td>
<td>SELECT</td>
<td>MBEANS pattern</td>
<td>Call getter methods on any mbean matching a wildcard pattern.</td>
</tr>
<tr>
<td>Role Management</td>
<td>ALTER</td>
<td>ALL ROLES</td>
<td>ALTER ROLE on any role</td>
</tr>
<tr>
<td>Role Management</td>
<td>ALTER</td>
<td>ROLE role_name</td>
<td>ALTER ROLE for specified role.</td>
</tr>
<tr>
<td>Role Management</td>
<td>AUTHORIZE</td>
<td>resource_name</td>
<td>GRANT privilege and REVOKE privilege on the resource. Roles are resources. Requires that user has AUTHORIZE on the resource.</td>
</tr>
<tr>
<td>Role Management</td>
<td>CREATE</td>
<td>ALL ROLES</td>
<td>CREATE ROLE</td>
</tr>
<tr>
<td>Role Management</td>
<td>DROP</td>
<td>ALL ROLES</td>
<td>Drop all roles.</td>
</tr>
<tr>
<td>Role Management</td>
<td>DROP</td>
<td>ROLE role_name</td>
<td>Drop the specified role.</td>
</tr>
<tr>
<td>Role Management</td>
<td>PROXY.EXECUTE</td>
<td>ROLE role_name</td>
<td>After authenticating issue individual requests as a different user.</td>
</tr>
<tr>
<td>Role Management</td>
<td>PROXY.LOGIN</td>
<td>ROLE role_name</td>
<td>After authenticating issue all requests as a different user.</td>
</tr>
<tr>
<td>Role Management</td>
<td>role_name</td>
<td>resource_name</td>
<td>Grant role (as a set of permissions) to another role. Requires AUTHORIZE permission on the permission role and target role.</td>
</tr>
<tr>
<td>Search index</td>
<td>ALL PERMISSIONS</td>
<td>ALL SEARCH INDICES</td>
<td>All search index privileges for all search indexes in the system.</td>
</tr>
<tr>
<td>Search index</td>
<td>ALL PERMISSIONS</td>
<td>SEARCH KEYSPACE keyspace_name</td>
<td>All search index privileges for all tables in specified keyspace.</td>
</tr>
<tr>
<td>Search index</td>
<td>ALL PERMISSIONS</td>
<td>SEARCH INDEX [keyspace_name]:table_name</td>
<td>All search index privileges for specified table.</td>
</tr>
<tr>
<td>Search index</td>
<td>SEARCH.ALTER</td>
<td>ALL SEARCH INDICES</td>
<td>ALTER SEARCH INDEX on all tables in all keyspaces.</td>
</tr>
<tr>
<td>Search index</td>
<td>SEARCH.ALTER</td>
<td>SEARCH KEYSPACE keyspace_name</td>
<td>ALTER SEARCH INDEX on all tables in specified keyspace.</td>
</tr>
<tr>
<td>Search index</td>
<td>SEARCH.ALTER</td>
<td>SEARCH INDEX [keyspace_name]:table_name</td>
<td>ALTER SEARCH INDEX on specified table.</td>
</tr>
<tr>
<td>Search index</td>
<td>SEARCH.COMMIT</td>
<td>ALL SEARCH INDICES</td>
<td>COMMIT SEARCH INDEX on all tables in all keyspaces.</td>
</tr>
<tr>
<td>Search index</td>
<td>SEARCH.COMMIT</td>
<td>SEARCH KEYSPACE keyspace_name</td>
<td>COMMIT SEARCH INDEX on all tables in specified keyspace.</td>
</tr>
<tr>
<td>Search index</td>
<td>SEARCH.COMMIT</td>
<td>SEARCH INDEX [keyspace_name]:table_name</td>
<td>COMMIT SEARCH INDEX on specified table.</td>
</tr>
<tr>
<td>Search index</td>
<td>SEARCH.CREATES</td>
<td>ALL SEARCH INDICES</td>
<td>CREATE SEARCH INDEX on all tables in all keyspaces.</td>
</tr>
</tbody>
</table>
### CQL reference

<table>
<thead>
<tr>
<th>Resource type</th>
<th>Privilege</th>
<th>Resource</th>
<th>Permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search index</td>
<td>SEARCH.CREATE</td>
<td>SEARCH KEYSPACE keyspace_name</td>
<td>CREATE SEARCH INDEX on all tables in specified keyspace.</td>
</tr>
<tr>
<td>Search index</td>
<td>SEARCH.CREATE</td>
<td>[keyspace_name].table_name</td>
<td>CREATE SEARCH INDEX on specified table.</td>
</tr>
<tr>
<td>Search index</td>
<td>SEARCH.DROP</td>
<td>ALL SEARCH INDICES</td>
<td>DROP SEARCH INDEX on all tables in all keyspaces.</td>
</tr>
<tr>
<td>Search index</td>
<td>SEARCH.DROP</td>
<td>SEARCH KEYSPACE keyspace_name</td>
<td>DROP SEARCH INDEX on all tables in specified keyspace.</td>
</tr>
<tr>
<td>Search index</td>
<td>SEARCH.DROP</td>
<td>SEARCH INDEX keyspace_name</td>
<td>DROP SEARCH INDEX on specified table.</td>
</tr>
<tr>
<td>Search index</td>
<td>SEARCH.REBUILD</td>
<td>ALL SEARCH INDICES</td>
<td>REBUILD SEARCH INDEX on any table in all keyspaces.</td>
</tr>
<tr>
<td>Search index</td>
<td>SEARCH.REBUILD</td>
<td>SEARCH KEYSPACE keyspace_name</td>
<td>REBUILD SEARCH INDEX on all tables in specified keyspace.</td>
</tr>
<tr>
<td>Search index</td>
<td>SEARCH.REBUILD</td>
<td>[keyspace_name].table_name</td>
<td>REBUILD SEARCH INDEX on specified table.</td>
</tr>
<tr>
<td>Search index</td>
<td>SEARCH.RELOAD</td>
<td>ALL SEARCH INDICES</td>
<td>RELOAD SEARCH INDEX on all tables in all keyspaces.</td>
</tr>
<tr>
<td>Search index</td>
<td>SEARCH.RELOAD</td>
<td>SEARCH KEYSPACE keyspace_name</td>
<td>RELOAD SEARCH INDEX on all tables in specified keyspace.</td>
</tr>
<tr>
<td>Search index</td>
<td>SEARCH.RELOAD</td>
<td>[keyspace_name].table_name</td>
<td>RELOAD SEARCH INDEX on specified table.</td>
</tr>
<tr>
<td>Spark applications</td>
<td>CREATE</td>
<td>ANY WORKPOOL</td>
<td>Submit an application to the work pool in any datacenter.</td>
</tr>
<tr>
<td>Spark applications</td>
<td>CREATE</td>
<td>WORKPOOL datacenter_name</td>
<td>Submit an application to the work pool in a specific datacenter.</td>
</tr>
<tr>
<td>Spark applications</td>
<td>MODIFY</td>
<td>ANY SUBMISSION</td>
<td>Manage any applications across all datacenters.</td>
</tr>
<tr>
<td>Spark applications</td>
<td>MODIFY</td>
<td>ANY SUBMISSION IN WORKPOOL</td>
<td>Manage applications in a specified datacenter.</td>
</tr>
<tr>
<td>Spark applications</td>
<td>MODIFY</td>
<td>SUBMISSION application_ID IN WORKPOOL datacenter_name</td>
<td>Manage a single application in a specified datacenter.</td>
</tr>
</tbody>
</table>

### 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, the following examples use internal users.
Give the role coach permission to perform SELECT statements on all tables in all keyspaces:

```cql
GRANT SELECT ON ALL KEYSPACES TO coach;
```

Give the role manager permission to perform INSERT, UPDATE, DELETE and TRUNCATE statements on all tables in the field keyspace:

```cql
GRANT MODIFY ON KEYSspace field TO manager;
```

Give the role coach the permission to run ALTER KEYSspace statements on the cycling keyspace, and also ALTER TABLE, CREATE INDEX, and DROP INDEX statements on all tables in the cycling keyspace:

```cql
GRANT ALTER ON KEYSspace cycling TO coach;
```

Give the role coach permission to run all statements on the cycling.name table:

```cql
GRANT ALL PERMISSIONS ON cycling.name TO coach;
```

Create an administrator role with full access to the cycling keyspace:

```cql
GRANT ALL ON KEYSspace cycling TO cycling_admin;
```

Give the role sponsor permission to perform SELECT statements on rows that contain ‘SPONSORED’ in the cycling.rank table:

```cql
GRANT SELECT ON 'SPONSORED' ROWS IN cycling.rank TO sponsor;
```

The filtering_data string is case-sensitive.

To view permissions:

```cql
LIST ALL PERMISSIONS
```

To grant create permissions on a work pool in a specific datacenter:

```cql
GRANT CREATE ON WORKPOOL datacenter_name TO role_name;
```

**INSERT**

Inserts an entire row or upserts data into an existing row; statement must include 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. Returns no results unless IF NOT EXISTS is used.

- Insert does not support counter columns, use UPDATE instead.
- A PRIMARY KEY consists of a the partition key followed by the clustering columns.

**Synopsis**

```cql
INSERT INTO [keyspace_name.]table_name
[ column_list VALUES column_values ]
[ IF NOT EXISTS ]
```
INSERT also supports JSON syntax to provide manual testing and troubleshooting from the command line, see Inserting JSON formatted values.

To modify a base table that has a materialized view (MV) using an INSERT or UPDATE command if access permissions are enabled, a user must be granted MODIFY or ALL PERMISSIONS on the base table.

Table 69: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italics</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[]</td>
<td>Optional. Square brackets ([ ]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>()</td>
<td>Group. Parentheses ( () ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation ( ' ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>( key : value )</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon (;) terminates all CQL statements.</td>
</tr>
<tr>
<td>[---]</td>
<td>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.</td>
</tr>
<tr>
<td>'&lt;&lt;schema&gt;&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ' ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

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 a literal or the following syntax for collections:
- Set: Enter values between curly braces: { literal [, ... ] }.
- List: Enter values between square brackets: [ literal [, ... ] ].
- Map: Enter values between curly braces: { key : value [, ... ] }.

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.

The database storage engine can only encode TTL timestamps through January 19 2038 03:14:07 UTC due to the Year 2038 problem. The TTL date overflow policy determines whether requests with expiration timestamps later than the maximum date are rejected or inserted. See Dcassandra.expiration_date_overflow_policy=Policy.

**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, the actual time of write is used.

**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.

```cql
INSERT INTO cycling.cyclist_name {
  id, lastname, firstname
} VALUES (
  6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47, 'KRUIKSWIJK', '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.

Insert a list of categories for a cyclist.

```cql
INSERT INTO cycling.cyclist_categories {
  id, lastname, categories
} VALUES (
  '6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47',
  'KRUIKSWIJK',
  { 'GC', 'Time-trial', 'Sprint' }
);
```

Insert a map named `teams` that lists two recent team memberships for the user VOS.

```cql
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** is included and there is no row with the same primary key, the command is performed and returns `true` in the \[applied\] column of the results. For example:

```
\[applied\]
--------------
  True
```

With **IF NOT EXISTS**, if the row already exists, the command returns `false` in the \[applied\] column and returns the values for the existing row. For example:

```
\[applied\] | id                                   | firstname | lastname
-----------+--------------------------------------+-----------+----------
    False | c4b65263-fe58-4846-83e8-f0e1c13d518f |  Rissella |    RATTO
```

Using **IF NOT EXISTS** incurs a performance hit associated with using Paxos internally. For information about Paxos, see **Linearizable consistency**.

**LIST PERMISSIONS**

Lists all permissions on all resources, role permissions on all resources, or for a specified resource.

- Only superusers can list all permissions.
- Requires **DESCRIBE** permission on the target resources and roles.

**Synopsis**

```
LIST permission
    [ ON resource_name ]
    [ OF role_name ]
    [ NORECURSIVE ] ;
```

**Table 70: Legend**

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
</tbody>
</table>
### Syntax conventions

<table>
<thead>
<tr>
<th>Lowercase</th>
<th>Not literal.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>italics</strong></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[]</td>
<td>Optional. Square brackets { [ ] } surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>()</td>
<td>Group. Parentheses { ( ) } identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td>Or. A vertical bar (</td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis (...) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation ( ‘ ’ ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces { ( ) } enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1, datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets (&lt; &gt;) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon (;) terminates all CQL statements.</td>
</tr>
<tr>
<td>[--]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ‘ ’ ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity=&quot;xml_entity_type&quot;</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

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**
- **PROXY.EXECUTE**
- **PROXY.LOGIN**
- **SEARCH.ALTER**
- **SEARCH.COMMIT**
- **SEARCH.CREATE**
CQL reference

- SEARCH.DROP
- SEARCH.REBUILD
- SEARCH.RELOAD
- SELECT

resource_name

The DataStax Distribution of Apache Cassandra™ database objects to which permissions are applied. Database resources have modeled hierarchy. Grant permissions on a resource higher in the chain to automatically grant that same permission on all resources lower down.

Not all privileges apply to every type of resource. For instance, **EXECUTE** is only relevant in the context of functions, MBeans, RPC, and authentication schemes. Attempting to grant privileges on a resource that the permission is not applicable results in an error.

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:

<table>
<thead>
<tr>
<th>rolename</th>
<th>resource</th>
<th>permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>coach</td>
<td>&lt;keyspace field&gt;</td>
<td>MODIFY</td>
</tr>
</tbody>
</table>

List permissions given to all the roles:

```
LIST ALL;
```

Output is:

<table>
<thead>
<tr>
<th>rolename</th>
<th>resource</th>
<th>permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>coach</td>
<td>&lt;keyspace field&gt;</td>
<td>MODIFY</td>
</tr>
<tr>
<td>manager</td>
<td>&lt;keyspace cyclist&gt;</td>
<td>ALTER</td>
</tr>
<tr>
<td>manager</td>
<td>&lt;table cyclist.name&gt;</td>
<td>CREATE</td>
</tr>
<tr>
<td>manager</td>
<td>&lt;table cyclist.name&gt;</td>
<td>ALTER</td>
</tr>
<tr>
<td>manager</td>
<td>&lt;table cyclist.name&gt;</td>
<td>DROP</td>
</tr>
<tr>
<td>manager</td>
<td>&lt;table cyclist.name&gt;</td>
<td>SELECT</td>
</tr>
<tr>
<td>manager</td>
<td>&lt;table cyclist.name&gt;</td>
<td>MODIFY</td>
</tr>
<tr>
<td>manager</td>
<td>&lt;table cyclist.name&gt;</td>
<td>AUTHORIZE</td>
</tr>
<tr>
<td>coach</td>
<td>&lt;all keyspaces&gt;</td>
<td>SELECT</td>
</tr>
</tbody>
</table>

List all permissions on the **cyclist.name** table:

```
LIST ALL
```
ON cyclist.name;

Output is:

<table>
<thead>
<tr>
<th>username</th>
<th>resource</th>
<th>permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>manager</td>
<td>&lt;table cyclist.name&gt;</td>
<td>CREATE</td>
</tr>
<tr>
<td>manager</td>
<td>&lt;table cyclist.name&gt;</td>
<td>ALTER</td>
</tr>
<tr>
<td>manager</td>
<td>&lt;table cyclist.name&gt;</td>
<td>DROP</td>
</tr>
<tr>
<td>manager</td>
<td>&lt;table cyclist.name&gt;</td>
<td>SELECT</td>
</tr>
<tr>
<td>manager</td>
<td>&lt;table cyclist.name&gt;</td>
<td>MODIFY</td>
</tr>
<tr>
<td>manager</td>
<td>&lt;table cyclist.name&gt;</td>
<td>AUTHORIZE</td>
</tr>
<tr>
<td>coach</td>
<td>&lt;all keyspaces&gt;</td>
<td>SELECT</td>
</tr>
</tbody>
</table>

List all permissions on the `cyclist.name` table and its parents:

LIST ALL
ON cyclist.name
NORECURSIVE;

Output is:

<table>
<thead>
<tr>
<th>username</th>
<th>resource</th>
<th>permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>manager</td>
<td>&lt;table cyclist.name&gt;</td>
<td>CREATE</td>
</tr>
<tr>
<td>manager</td>
<td>&lt;table cyclist.name&gt;</td>
<td>ALTER</td>
</tr>
<tr>
<td>manager</td>
<td>&lt;table cyclist.name&gt;</td>
<td>DROP</td>
</tr>
<tr>
<td>manager</td>
<td>&lt;table cyclist.name&gt;</td>
<td>SELECT</td>
</tr>
<tr>
<td>manager</td>
<td>&lt;table cyclist.name&gt;</td>
<td>MODIFY</td>
</tr>
<tr>
<td>manager</td>
<td>&lt;table cyclist.name&gt;</td>
<td>AUTHORIZE</td>
</tr>
</tbody>
</table>

LIST ROLES

Lists roles and shows superuser and login status.

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 71: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italic</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets ([ ]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>( )</td>
<td>Group. Parentheses (()) identify a group to choose from. Do not type the parentheses.</td>
</tr>
</tbody>
</table>
## Usage

### Syntax conventions

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td><code>...</code></td>
<td>Repeatable. An ellipsis (<code>...</code>) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td><code>'literal string'</code></td>
<td>Single quotation (`) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td><code>{ key : value }</code></td>
<td>Map collection. Braces (<code>{ }</code>) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td><code>&lt;datatype1, datatype2&gt;</code></td>
<td>Set, list, map, or tuple. Angle brackets (<code>&lt;&gt;</code>) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td><code>cql_statement;</code></td>
<td>End CQL statement. A semicolon (<code>;</code>) terminates all CQL statements.</td>
</tr>
<tr>
<td><code>[--]</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>' &lt;schema&gt; ... &lt;/schema&gt; '</code></td>
<td>Search CQL only: Single quotation marks (`) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td><code>@xml_entity=&quot;xml_entity_type</code></td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

### Examples

Show all the roles that the current role has permission to describe.

```
LIST ROLES;
```

Returns all roles:

```
role | super | login | options
-----+-------+-------+---------
alice | False | True | {}       
cassandra | True | True | {}       
cycling_admin | False | False | {}       
cycling_analyst | False | False | {}       
cycling_app | False | True | {}       
cycling_expense_management | False | False | {}       
dantest1 | False | True | {}       
hockey_analyst | False | False | {}       
jane | False | True | {}       
jason | False | True | {}       
john | False | True | {}       
keyspace_admin | False | False | {}       
martin | False | True | {}       
read_race | False | False | {}       
role_admin | False | False | {}       
root | True | True | {}       
sandy | False | True | {}       
sys_admin | False | True | {}       
update_accounts | False | False | {}       
update_races | False | False | {}       
update_teams | False | False | {}       
```
Show the roles for a particular role. Sufficient privileges are required to show this information.

LIST ROLES OF martin;

Returns only the roles assigned to martin:

<table>
<thead>
<tr>
<th>role</th>
<th>super</th>
<th>login</th>
<th>options</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyspace_admin</td>
<td>False</td>
<td>False</td>
<td>{}</td>
</tr>
<tr>
<td>martin</td>
<td>False</td>
<td>True</td>
<td>{}</td>
</tr>
</tbody>
</table>

LIST USERS (Deprecated)

Lists internally authenticated users, created users with the command CREATE USER, and have not yet changed the default user.

LIST USERS is supported for backwards compatibility. Authentication and authorization for DataStax Distribution of Apache Cassandra™ are based on ROLES, and LIST ROLES should be used.

Synopsis

LIST USERS ;

Table 72: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td><em>italics</em></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets ([ ]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>( )</td>
<td>Group. Parentheses ( ( ) ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation ( ‘ ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>( key : value )</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1, datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>[---]</td>
<td>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.</td>
</tr>
</tbody>
</table>
CQL reference

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ' ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

Example
List the current users:

LIST USERS;

Output is:

<table>
<thead>
<tr>
<th>name</th>
<th>super</th>
</tr>
</thead>
<tbody>
<tr>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>cassandra</td>
<td>True</td>
</tr>
<tr>
<td>boone</td>
<td>False</td>
</tr>
<tr>
<td>akers</td>
<td>True</td>
</tr>
<tr>
<td>spillman</td>
<td>False</td>
</tr>
</tbody>
</table>

RESTRICT
Configures the column on which permissions can be granted.

Tables have only a single restriction. Running the RESTRICT command replaces the existing restriction. Use DESCRIBE TABLE to view the existing restrictions on the table. The column must be part of the partition key.

Synopsis

```
RESTRICT ROWS
ON [{keyspace_name.]}table_name
USING pk_column_name ;
```

Table 73: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italic</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[]</td>
<td>Optional. Square brackets ( [ ] ) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>{}</td>
<td>Group. Parentheses ( { } ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td>Or. A vertical bar (</td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>&quot;Literal string&quot;</td>
<td>Single quotation ( ' ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>( key : value )</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
</tbody>
</table>
Syntax conventions | Description
---|---
<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
For the cyclist_expenses table, configure the cyclist_name column for filtering so that permissions can be assigned. In this example, we identify the column so that each cyclist can view only their own expenses:

RESTRICT ROWS ON cyclist_expenses USING cyclist_name;

RLAC requires two commands: a single RESTRICT and one or more GRANT commands. For example, cyclist Vera Adrian can view her expenses:

GRANT SELECT ON 'Vera ADRIAN' ROWS IN cyclist_expenses TO cycling_accounts;

The filtering_data string is case-sensitive.

REVOKE
Removes privileges on database objects from a role.

REVOKE does not automatically invalidate cached credentials and permissions. Permissions are invalidated the next time they are refreshed.

Synopsis

REVOKE privilege
  ON resource_name
  FROM role_name;

Table 74: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italic</td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets ([ ]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>( )</td>
<td>Group. Parentheses (( )) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td>Or. A vertical bar (</td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
</tbody>
</table>
**Syntax conventions**

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'literal string'</td>
<td>Single quotation (‘’) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1, datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets (&lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon (; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>[--]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks (‘ ’) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity=&quot;xml_entity_type&quot;</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

**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
- PROXY.EXECUTE
- PROXY.LOGIN
- SEARCH.ALTER
- SEARCH.COMMIT
- SEARCH.CREATE
- SEARCH.DROP
- SEARCH.REBUILD
- SEARCH.RELOAD
- SELECT

**resource_name**

The DataStax Distribution of Apache Cassandra™ database objects to which permissions are applied. Database resources have modelled hierarchy. Grant permissions on a resource higher in the chain to automatically grant that same permission on all resources lower down.
Not all privileges apply to every type of resource. For instance, `EXECUTE` is only relevant in the context of functions, MBeans, RPC, and authentication schemes. Attempting to grant privileges on a resource that the permission is not applicable results in an error.

Example

The role `manager` can no longer perform `SELECT` queries on the `cycling.name` table.

```plaintext
REVOKE SELECT
ON cycling.name
FROM manager;
```

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.

The role `coach` can no longer perform `GRANT`, `ALTER` or `REVOKE` commands on all roles:

```plaintext
REVOKE ALTER
ON ALL ROLES
FROM coach;
```

**SELECT**

Returns data from a single table. A select statement without a where clause returns all rows from all partitions, but is not recommended.

DataStax recommends limiting queries to a single partition using the WHERE clause. Queries across multiple partitions can impact performance.

**Synopsis**

```plaintext
SELECT selectors
FROM [keyspace_name.]{table_name}
[ WHERE [ primary_key_conditions ] [ AND ] [ index_conditions ]
[ GROUP BY column_name [ , ... ] ]
[ ORDER BY PK_column_name [ , ... ] \( ASC \mid DESC \) ]
[ ( LIMIT N | PER PARTITION LIMIT N ) ]
[ ALLOW FILTERING ] ;
```

**Table 75: Legend**

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td><em>Italics</em></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets ([ ]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>( )</td>
<td>Group. Parentheses ( { } ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
</tbody>
</table>
selectors
Determines the columns returned in the results set.

\[ \text{column_list} \mid \text{DISTINCT partition_key [ AS output_name ]} \]

Use either a column list or DISTINCT partition_key.

column_list
Determines the columns and column order returned in the result set. Specify a comma separated list of columns or use an asterisk to return all columns in the stored order.

\[ \text{column_name} \mid \text{function_name( argument_list )} \]

- \text{column_name}: Includes a column in result set.
- \text{function_name( arguments )}: Execute a function on the specified argument for each row in the result set, see CQL native functions and Creating user-defined function (UDF).
- \text{aggregate_name( arguments )}: Executes the aggregate on matching data and returns a single result, see CQL native aggregates and CREATE AGGREGATE.

DISTINCT partition_key
Returns unique values for the full partition key. Use a comma-separated list for compound partition keys.

Run \text{DESC TABLE table_name} to get the PRIMARY KEY definition and then \text{SELECT DISTINCT partition_key FROM table_name} to list of the table partition values.

AS output_name
Renames the column to the new output name in the result set; for example:

```
count(id) AS "Cyclist Count"
```

If the name contains special character, spaces, or to retain capitalization, surround the new name with double quotes.

**keyspace_name.table_name**

The keyspace name is required to identify a table in a different keyspace or if no keyspace is set for the session. If the keyspace or table name contain uppercase letters, enclose the name in double quotes; for example:

```
FROM "TestTable"
```

**primary_key_conditions**

Improves the efficiency of the query using logic statements to identify the data location and allows filtering on the last clustering column.

```
[ partition_conditions
  [ AND clustering_conditions ] | [ AND index_conditions ]
```

To return all the data stored on a partition only define the partition key values.

**Logical statement syntax**

To create logic statements that test the column value use the syntax:

```
column_name operator value
```

Separate multiple statements with AND, rows that meet all the conditions are returned. For example:

```
SELECT rank, cyclist_name AS name
FROM cycling.rank_by_year_and_name
WHERE "race_name" = 'Tour of Japan - Stage 4 - Minami > Shinshu'
  AND race_year = 2014;
```

The database does not support queries with logical disjunctions (OR).

**column_name**

Enclose column names that have uppercase or special characters in double quotes.

Enclose string values in single quotes.

**operators**

DataStax supports the following operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Column value exactly matches the specified value.</td>
</tr>
<tr>
<td>IN</td>
<td>Equal to any value in a comma-separated list of values.</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to the value.</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to the value.</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than the value.</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than the value.</td>
</tr>
</tbody>
</table>
value

Enclose string values in single quotes.

Enclose column names that have uppercase or special characters in double quotes.

Identifying the data location and filtering by clustering columns

Use WHERE clauses to maximize read efficiency by identifying the location of the data. The database evaluates the WHERE logical statements hierarchically:

1. Partition key columns: Use the equal operator to identify all partition key values (or none). Ensure that the data model supports single partition queries to avoid performance issues.

   Partitions are typically large sets of data. The partitioner distributes the data by creating a hash of the partition key columns and stores all the rows with the same hash on the same node. Similar or like data, such as partition key date column values 7/01/2017 and 7/02/2017, may not be located on the same node.

2. Clustering columns determines the sort order within the partition; data is sorted by the first clustering column, the second and so forth.

   ALLOW FILTERING overrides restrictions on filtering partition, clustering, and regular columns, but can negatively impact performance by causing read latencies. In test environments use cqlsh TRACING.

partition_conditions

The database requires that all partitions are restricted except when querying a secondary or search index. Use logic statements that identify the partition key columns with following operators:

- **Equals (=)**: Any partition key column.
- **IN**: Restricted to the last column of the partition key to search multiple partitions.
- **Range (>=, <=, >, and <) on tokens**: Fully tokenized partition key (all PK columns specified in order as arguments of the `TOKEN` function). Use token ranges to scan data stored on a particular node.

   For secondary index queries, equals is the only operator supported for partition key logical statements.

   See Partition keys for usage examples and instructions.

clustering_conditions

Use logic statements that identify the clustering segment. Clustering columns set the sort order of the stored data, which is nested when there are multiple clustering columns. After evaluating the partition key, the database evaluates the clustering statements in the nested order, the first (top level), second, third, and so on.

All operators are supported in logical statements if the table has only one clustering column. To efficiently locate the data within the partition for tables with multiple clustering columns, the following restrictions apply:

- Top level clustering columns:
  
  # Equals (=)
  
  # IN
• Last clustering column statement: All operators and multi-column comparisons

Clustering column logic statements also support returning slices across clustering segments:

```
( column1, column2, ... ) operator ( value1, value2, ... )
[ AND ( column1, column2, ... ) operator ( value1, value2, ... ) ]
```

The slice identifies the row that has the corresponding values and allows you to return all rows before, after, or between (when two slice statements are included).

See Clustering columns for usage examples and instructions.

**index_conditions**

The DataStax Distribution of Apache Cassandra™ (DDAC) database supports one type of index.

**secondary index**

Logical statement syntax for secondary index columns:

```
column_name operator value
```

where an operator is:

• =

• CONTAINS on index collection types

• CONTAINS KEY on index map types

**Additional options**

Change the scope and order of the data returned by the query.

**GROUP BY** *column_name*

Return unique values of the target column name.

**ORDER BY** (ASC | DESC)

Sorts the result set in either ascending (ASC) or descending (DESC) order.

When no order is specified the results are returned in the ordered that they are stored.

**ALLOW FILTERING**

Enables filtering without including logic statements that the identify primary key or allows filtering on primary keys.

For more information, see Allow Filtering explained.

**LIMIT** *N* | **PER PARTITION LIMIT** *N*

Limits the number of records returned in the results set.

**Examples**

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:

```
<table>
<thead>
<tr>
<th>event_id</th>
<th>creation_date</th>
<th>content</th>
</tr>
</thead>
</table>
```

CQL for the DataStax Distribution of Apache Cassandra™ 3.11 Latest DDAC patch: 5.1.17
The number of rows returned by the query is shown at the bottom of the output.

Specifying the source table using `FROM`

The following example `SELECT` statement returns the number of rows in the `IndexInfo` table in the `system` keyspace:

```cql
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:

```cql
SELECT lastname
FROM cycling.cyclist_name
LIMIT 50000;
```

Even if the query matches 105,291 rows, the database 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.

Selecting partitions

Simple partition key, select a single partition:

```cql
partition_column = value
```

Simple partition key, select multiple partitions:

```cql
partition_column IN ( value1, value2 [ , ... ] )
```

For compound partition keys, create a condition for each key separated by `AND`:

```cql
partition_column1 = value1
AND partition_column2 = value2 [ AND ... ]
```

Controlling the number of rows returned using `PER PARTITION LIMIT`

In Cassandra, the `PER PARTITION LIMIT` option sets the maximum number of rows that the query returns from each partition. For example, create a table that will sort data into more than one partition.

```cql
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)  
);
```

After inserting data, the table contains these rows:

```
race_year | race_name                                  | rank | cyclist_name
-----------+--------------------------------------------+------+----------------------
2014 | 4th Tour of Beijing | 1 | Phillippe GILBERT
```
To get the top two racers in every race year and race name, use the `SELECT` statement with `PER PARTITION LIMIT 2`.

```
SELECT *
FROM cycling.rank_by_year_and_name
PER PARTITION LIMIT 2;
```

Output:

<table>
<thead>
<tr>
<th>race_year</th>
<th>race_name</th>
<th>rank</th>
<th>cyclist_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2014</td>
<td>4th Tour of Beijing</td>
<td>1</td>
<td>Phillippe GILBERT</td>
</tr>
<tr>
<td>2014</td>
<td>4th Tour of Beijing</td>
<td>2</td>
<td>Daniel MARTIN</td>
</tr>
<tr>
<td>2015</td>
<td>Giro d’Italia - Stage 11 - Forli &gt; Imola</td>
<td>1</td>
<td>Ilnur ZAKARIN</td>
</tr>
<tr>
<td>2015</td>
<td>Giro d’Italia - Stage 11 - Forli &gt; Imola</td>
<td>2</td>
<td>Carlos BETANCUR</td>
</tr>
<tr>
<td>2015</td>
<td>Tour of Japan – Stage 4 – Minami &gt; Shinshu</td>
<td>1</td>
<td>Benjamin PRADIES</td>
</tr>
<tr>
<td>2015</td>
<td>Tour of Japan – Stage 4 – Minami &gt; Shinshu</td>
<td>2</td>
<td>Adam PHELAN</td>
</tr>
</tbody>
</table>

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`.

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 = '12345678-9012-3456-7890-123456789012';
```
FROM cycling.cyclist_career_teams
WHERE id=5b6962dd-3f90-4c93-8f61-eabfa4a803e2;

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-decec1e30aac
AND race_points > 7;

Output:

    system.sum(race_points)  
--------------------------------
       195

(1 rows)

In DataStax Enterprise 5.1 and later, it is possible to add ALLOW FILTERING to filter on a non-indexed clustering column.

Avoid ALLOW FILTERING because it impacts performance.

The table definition is included in this example to show that race_start_date is a clustering column without a secondary index:

CREATE TABLE cycling.calendar (  
race_id int,  
race_name text,  
race_start_date timestamp,  
race_end_date timestamp,  
PRIMARY KEY (race_id, race_start_date, race_end_date)  
);

SELECT *  
FROM cycling.calendar  
WHERE race_start_date='2015-06-13'  
ALLOW FILTERING;

Output:

<table>
<thead>
<tr>
<th>race_id</th>
<th>race_start_date</th>
<th>race_end_date</th>
<th>race_name</th>
</tr>
</thead>
</table>
It is possible to combine the partition key and a clustering column in a single relation. For details, see Comparing clustering columns.

## 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 it impacts performance. 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.

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

<table>
<thead>
<tr>
<th>album</th>
<th>tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tres Hombres</td>
<td>(&quot;1977&quot;, &quot;blues&quot;)</td>
</tr>
</tbody>
</table>

After indexing the music venue map, filter on map values, such as 'The Fillmore':

```cql
SELECT *
FROM playlists
WHERE venue CONTAINS 'The Fillmore';
```

After indexing the collection keys in the venues map, filter on map keys.

```cql
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.)

```cql
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`.

```cql
SELECT *
FROM cycling.birthday_list
```
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.

```cql
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.

```cql
SELECT *
FROM cycling.race_starts
WHERE rnumbers = [39, 7, 14];
```

Range relations

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 selection of a contiguous set of rows.

For example:

```cql
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:

```cql
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 selection of 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.
ALLOW FILTERING is intended for development environments only and is not recommended for production. 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.

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:

```
<table>
<thead>
<tr>
<th>id</th>
<th>song_order</th>
<th>album</th>
<th>artist</th>
<th>song_id</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>62c36092...</td>
<td>1</td>
<td>4 No One Rides For Free</td>
<td>Fu Manchu</td>
<td>7ba2a8b1aa...</td>
<td>Ojo Rajo</td>
</tr>
<tr>
<td>62c36092...</td>
<td>1</td>
<td>Roll Away Back Door Slim</td>
<td>2093150...</td>
<td>Outside Women Blues</td>
<td></td>
</tr>
<tr>
<td>62c36092...</td>
<td>1</td>
<td>2 We Must Obey</td>
<td>Fu Manchu</td>
<td>6b7f2531ab...</td>
<td>Moving in Stereo</td>
</tr>
<tr>
<td>62c36092...</td>
<td>1</td>
<td>Tres Hombres 22 Top</td>
<td>1c62684f3...</td>
<td>La Orange</td>
<td></td>
</tr>
</tbody>
</table>
```

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:
Grouping results

A `GROUP BY` clause condenses the selected rows that share the same values for a set of columns into a group.

For example, this query groups the rows by the race date in the `race_times` table:

```sql
SELECT race_date, race_name
FROM cycling.race_times
GROUP BY race_date;
```

The output:

```
race_date | race_name
------------+----------------------------------
2017-04-14 | 17th Santos Tour Down Under
```

(1 rows)

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. `COUNT` ignores null values.

For example, count the number of last names in the `cyclist_name` table:

```sql
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 counts the number of rows in the cyclist name table:

```sql
SELECT COUNT(*)
FROM cycling.cyclist_name;
```

This example calculates the maximum value for start day in the cycling events table and counts the number of rows returned:

```sql
SELECT start_month, MAX(start_day), COUNT(*)
FROM cycling.events
WHERE year = 2017
    AND discipline = 'Cyclo-cross';
```

This example provides a year that is not stored in the events table:

```sql
SELECT start_month, MAX(start_day)
FROM cycling.events
WHERE year = 2022
```
In the output, notice that the columns are null, and one row is returned:

<table>
<thead>
<tr>
<th>start_month</th>
<th>system.max(start_day)</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

(1 row)

Getting maximum and minimum values in a column

A `SELECT` expression using `MAX(column_name)` returns the maximum value in a column. When the column's data type is numeric (`bigint`, `decimal`, `double`, `float`, `int`, `smallint`), this is the highest value.

```
SELECT MAX(points)
FROM cycling.cyclist_category;
```

Output:

```
+-----------------+
<table>
<thead>
<tr>
<th>system.max(points)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1324</td>
</tr>
</tbody>
</table>
```

`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:

```
+----------+---------------+
<table>
<thead>
<tr>
<th>category</th>
<th>system.min(points)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GC</td>
<td>1269</td>
</tr>
</tbody>
</table>
```

If the column referenced by `MAX` or `MIN` has an `ascii` or `text` data type, these functions return the last or first item in an alphabetic sort of the column values. If the specified column has data type `date` or `timestamp`, these functions return the most recent or least recent times and dates. If a column has a null value, `MAX` and `MIN` ignores that value; if the column for an entire set of rows contains null, `MAX` and `MIN` return null.

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` statement:

```
|qsln:cyclingo select sum(points) from cyclist_category where category = 'Time-trial';
+-----------------+
<table>
<thead>
<tr>
<th>system.sum(points)</th>
</tr>
</thead>
<tbody>
<tr>
<td>412</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>

|qsln:cyclingo select avg(points) from cyclist_category where category = 'Sprint';
+-----------------+
<table>
<thead>
<tr>
<th>system.avg(points)</th>
</tr>
</thead>
<tbody>
<tr>
<td>197</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>
```

If any of the rows returned have a null value for the column referenced for a `SUM` or `AVG` aggregation, Cassandra includes that row in the row count, but uses a zero value to calculate the average.

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:

```cql
SELECT WRITETIME (first_name)
FROM users
WHERE last_name = 'Jones';
```

<table>
<thead>
<tr>
<th>writetime(first_name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1353010594789000</td>
</tr>
</tbody>
</table>

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:

```cql
INSERT INTO cycling.calendar (race_id, race_name, race_start_date, race_end_date)
USING TTL 100;
```

```cql
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 this `SELECT` statement to check its current value:

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

<table>
<thead>
<tr>
<th>Output:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ttl(race_name)</td>
</tr>
<tr>
<td>276</td>
</tr>
<tr>
<td>(1 rows)</td>
</tr>
</tbody>
</table>

Retrieving values in the JSON format

This option is available in DDAC. 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 76: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td><em>italics</em></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets ([ ]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>( )</td>
<td>Group. Parentheses ( ( )) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation ( ’ ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets (&lt; &gt;) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement ;</td>
<td>End CQL statement: A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>--</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ’ ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity='xml_entity_type'</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

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;

   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.

**UNRESTRICT**

Removes the partition key selection used to **GRANT** row-level access to roles.

When no column is selected, roles that have been granted access on rows (but not the table or keyspace) no longer have access. To restore or change the selected column for RLAC, use the **RESTRICT** command.

**Synopsis**

```
UNRESTRICT ROWS ON [keyspace_name.]table_name;
```

**Table 77: Legend**

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td><em>variable</em></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Optional. Square brackets ([ ]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>( )</td>
<td>Group. Parentheses ( ( ) ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Or. A vertical bar (</td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'Literal string'</td>
<td>Single quotation ( ‘ ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>{ key : value }</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>[---]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ‘ ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity=&quot;xml_entity_type&quot;</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

**Examples**

Remove row-level permissions from the currently selected column:

```
UNRESTRICT ROWS ON cyclist_name;
```

**UPDATE**

Modifies one or more column values to a row in a table.
Synopsis

```
UPDATE [keyspace_name.]table_name
[ USING TTL time_value ]
[ [ AND ] USING TIMESTAMP timestamp_value ]
SET assignment [ , assignment , ... ]
WHERE row_specification
[ { IF EXISTS | IF NOT EXISTS | IF condition [ AND condition ... ] } ] ;
```

To modify a base table that has a materialized view (MV) using an INSERT or UPDATE command if access permissions are enabled, a user must be granted MODIFY or ALL PERMISSIONS on the base table.

Table 78: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>Not literal.</td>
</tr>
<tr>
<td><em>Italics</em></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>{}</td>
<td>Optional. Square brackets ({} ) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>{}</td>
<td>Group. Parentheses ({} ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation ( ’ ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>( key : value )</td>
<td>Map collection. Braces ({} ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets (&lt; &gt;) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon (;) terminates all CQL statements.</td>
</tr>
<tr>
<td>[---]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ’ ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity=&quot;xml_entity_type&quot;</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

UPDATE writes one or more column values to a row in a 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.

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:

```cql
UPDATE cycling.cyclist_name
SET comments = 'Rides hard, gets along with others, a real winner'
WHERE id = fb372533-eb95-4bb4-8685-6ef61e994ca
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.

**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.

The database storage engine can only encode TTL timestamps through January 19 2038 03:14:07 UTC due to the Year 2038 problem. The TTL date overflow policy determines whether requests with expiration timestamps later than the maximum date are rejected or inserted. See -Dcassandra.expiration_date_overflow_policy=POLICY.

**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, the database uses the time (in microseconds) that the update occurred to the column.

**assignment**

Assigns a value to an existing element.

Can be one of:

```cql
| 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'] } |
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>column_name</td>
<td>The name of the column to be updated.</td>
</tr>
<tr>
<td>column_value</td>
<td>The value to be inserted for the specified column name.</td>
</tr>
<tr>
<td>counter_column_name</td>
<td>The name of the counter column to be updated.</td>
</tr>
<tr>
<td>Variable</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>counter_offset</td>
<td>The value by which the specified counter is be incremented or decremented (depending on whether the counter_offset is preceded by &quot;=&quot; or &quot;-&quot;).</td>
</tr>
<tr>
<td>list_name</td>
<td>The name of the list to be updated. Format of a list:</td>
</tr>
<tr>
<td></td>
<td>[list_item, list_item, list_item]</td>
</tr>
<tr>
<td>list_item</td>
<td>The value to be added to the list, or removed from it.</td>
</tr>
<tr>
<td>map_name</td>
<td>The name of the map to be updated. Format of a map:</td>
</tr>
<tr>
<td></td>
<td>{key: value, key: value, key: value...}</td>
</tr>
<tr>
<td>map_key</td>
<td>The first term or key in a map entry.</td>
</tr>
<tr>
<td>map_value</td>
<td>The second term or value in a map entry.</td>
</tr>
<tr>
<td>set_name</td>
<td>The name of the set to be updated. Format of a set:</td>
</tr>
<tr>
<td></td>
<td>{set_item, set_item, set_item...}</td>
</tr>
<tr>
<td>set_item</td>
<td>The literal value included in a set. The difference between a list and a set: each item in a set must be unique.</td>
</tr>
</tbody>
</table>

**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.

To update a static column, you only need to specify the partition key.

**IF EXISTS / IF condition**

Performs validation before updating records (lightweight transaction). Use as follows:

- **IF EXISTS** - One or more rows must match the query. If no rows match, the statement fails.

  ```
  UPDATE cycling.cyclist_name
  SET comment = 'Rides hard, gets along with others, a real winner'
  WHERE id = fb372533-eb95-4bb4-8685-6ef61e994caa
  IF EXISTS;
  ```

  When no rows match an UPDATE statement that does not have **IF EXISTS**, a new record is created.

- **IF conditional_statement** - Test non-primary key columns on rows that match the query. Applies the update to rows that return true. If no rows match the query and the conditional statement tests for NULL, a new record is inserted.

  ```
  UPDATE cycling.cyclist_name
  ```
Using IF statements impact performance.

Examples

Updating a column

Update a column in several rows at once:

```sql
UPDATE users
SET state = 'TX'
WHERE user_uuid
IN (88b8fd18-b1ed-4e96-bf79-4280797cba80, 06a8913c-c0d6-477c-937d-6c1b69a95d43, bc108776-7cb5-477f-917d-869c12df0fa8);
```

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:

```sql
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:

```sql
UPDATE excelsior.clicks
USING TTL 432000
SET user_name = 'bob'
WHERE userid = cfd66ccc-d857-4e90-b1e5-df98a3d40cd6
    AND url = 'http://google.com';
```

```sql
UPDATE Movies
SET col1 = val1, col2 = val2
WHERE movieID = key1;
```

```sql
UPDATE Movies
SET col3 = val3
WHERE movieID IN (key1, key2, key3);
```

```sql
UPDATE Movies
SET col4 = 22
WHERE movieID = key4;
```

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.

```sql
UPDATE cycling.popular_count
SET popularity = popularity + 2
```
CQL reference

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 the database 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 \( \text{id} \), you can UPDATE the partition with \( \text{id} \) 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."

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'}
```
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 the database 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, the database 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.

Updating a UDT with non-collection fields

In DataStax Distribution of Apache Cassandra™ (DDAC), to change the value of an individual field value in a user-defined type with non-collection fields, use the `UPDATE` command:

```
UPDATE cyclist_stats
SET basics.birthday = '2000-12-12'
WHERE id = 220844bf-4860-49d6-9a4b-6b5d3a79cbfb;
```

Conditionally updating columns

You can 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:

```
True
```

If no such row exists, however, the condition returns `FALSE` and the command fails. In this case, standard output looks like:

```
False
```

Use `IF condition` to apply tests to one or more column values in the selected row:

```
UPDATE cyclist_id
```
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:

<table>
<thead>
<tr>
<th>applied</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>18</td>
</tr>
</tbody>
</table>

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 creates a new row if no matching row is found. New rows are not immediately available for lightweight transactions 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, 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 79: Legend

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td>Literal keyword.</td>
</tr>
</tbody>
</table>
## Syntax conventions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lowercase</strong></td>
<td>Not literal.</td>
</tr>
<tr>
<td><em>italics</em></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[]</td>
<td>Optional. Square brackets ( [] ) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>()</td>
<td>Group. Parentheses ( ( ) ) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>'literal string'</td>
<td>Single quotation ( ‘ ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>( key : value )</td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
<td>Set, list, map, or tuple. Angle brackets ( &lt; &gt; ) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.</td>
</tr>
<tr>
<td>cql_statement;</td>
<td>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<td>[---]</td>
<td>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.</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
<td>Search CQL only: Single quotation marks ( ‘ ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<td>@xml_entity=&quot;xml_entity_type&quot;</td>
<td>Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.</td>
</tr>
</tbody>
</table>

## Example

```
USE PortfolioDemo;

USE "Excalibur";
```
CQL commands to set up the cycling examples used in this guide.

Use the replication factor that is appropriate for the type of snitch in your environment. The following assumes that the DataStax Distribution of Apache Cassandra™ deployment is a single node cluster in a development environment.

```
CREATE KEYSPACE IF NOT EXISTS cycling
WITH replication = {
  'class' : 'SimpleStrategy',
  'replication_factor' : 1
};
```

### Cycling tables

CQL commands that create cycling example tables and insert the corresponding data.

**birthday_list**

Example for maps and indexing search index map fields.

Create the keyspace, table, and load the data:

```
CREATE KEYSPACE IF NOT EXISTS cycling
WITH REPLICATION = { 'class' : 'SimpleStrategy',
  'replication_factor' : 1 };

// START-drop
DROP TABLE IF EXISTS cycling.birthday_list;
// END-drop

/* Map field and search index map fields example */

// START-table
CREATE TABLE cycling.birthday_list (
  cyclist_name text PRIMARY KEY,
  blist_ map<text,text>);
// END-table

// START-insertall
INSERT INTO cycling.birthday_list (cyclist_name, blist_) VALUES ('Allan DAVIS',
  {'blist_age':'35', 'bday':'27/07/1980', 'blist_nation':'AUSTRALIA'});
INSERT INTO cycling.birthday_list (cyclist_name, blist_) VALUES ('Claudio VANDELLI',
  {'blist_age':'54', 'bday':'27/07/1961', 'blist_nation':'ITALY'});
INSERT INTO cycling.birthday_list (cyclist_name, blist_) VALUES ('Laurence BOURQUE',
  {'blist_age':'23', 'bday':'27/07/1992', 'nation':'CANADA'});
INSERT INTO cycling.birthday_list (cyclist_name, blist_) VALUES ('Claudio HEINEN',
  {'blist_age':'23', 'bday':'27/07/1992', 'blist_nation':'GERMANY'});
INSERT INTO cycling.birthday_list (cyclist_name, blist_) VALUES ('Luc HAGENAARS',
  {'blist_age':'28', 'bday':'27/07/1987', 'blist_nation':'NETHERLANDS'});
INSERT INTO cycling.birthday_list (cyclist_name, blist_) VALUES ('Toine POELS',
  {'blist_age':'52', 'bday':'27/07/1963', 'blist_nation':'NETHERLANDS'});
// END-insertall

Create a default index:

```
CREATE KEYSPACE IF NOT EXISTS cycling
WITH REPLICATION = { 'class' : 'SimpleStrategy',
  'replication_factor' : 1 };
```
Cycling keyspace

```
SOURCE "birthday_list-table.cql";
// For different key-value pairs that you want specify a data types
CREATE SEARCH INDEX ON cycling.birthday_list;
// END-index

// Return all fields that were indexed on a key
SELECT * FROM cycling.birthday_list WHERE solr_query = 'blist_age:*';
SELECT * FROM cycling.birthday_list WHERE solr_query = 'bday:*';

// Regular queries -- how do you query indexed maps?
SELECT * FROM cycling.birthday_list WHERE blist_['blist_age'] = '23';
SELECT * FROM cycling.birthday_list WHERE blist_['blist_nation'] = 'GERMANY';
SELECT * FROM cycling.birthday_list WHERE blist_['bday'] = '27/07/1992' ALLOW FILTERING;

// using search index
SELECT * FROM cycling.birthday_list WHERE solr_query = 'blist_age:23';
SELECT * FROM cycling.birthday_list WHERE solr_query = 'blist_nation:GERMANY';

Create a custom index:

```
CREATE TABLE cycling.comments {
    record_id timeuuid,
    id uuid,
    commenter text,
    ...
```
comment text,  
created_at timestamp,  
PRIMARY KEY (id, created_at))  
WITH CLUSTERING ORDER BY (created_at DESC);

Data

Insert the data:

```
INSERT INTO cycling.comments (record_id, id, created_at, comment, commenter) values
    (now(), e7ae5cf3-d358-4d99-b900-85902fda9bb0, '2017-02-14 12:43:20-0800', 'Raining too hard should have postponed', 'Alex');

INSERT INTO cycling.comments (record_id, id, created_at, comment, commenter) values
    (now(), e7ae5cf3-d358-4d99-b900-85902fda9bb0, '2017-02-14 12:43:20.234-0800', 'Raining too hard should have postponed', 'Alex');

INSERT INTO cycling.comments (record_id, id, created_at, comment, commenter) values
    (now(), e7ae5cf3-d358-4d99-b900-85902fda9bb0, '2017-03-21 13:11:09.999-0800', 'Second rest stop was out of water', 'Alex');

INSERT INTO cycling.comments (record_id, id, created_at, comment, commenter) values
    (now(), e7ae5cf3-d358-4d99-b900-85902fda9bb0, '2017-04-01 06:33:02.16-0800', 'LATE RIDERS SHOULD NOT DELAY THE START', 'Alex');

INSERT INTO cycling.comments (record_id, id, created_at, comment, commenter) values
    (now(), c7fceba0-c141-4207-9494-a29f9809de6f, totimestamp(now()), 'The gift certificate for winning was the best', 'Amy');

INSERT INTO cycling.comments (record_id, id, created_at, comment, commenter) values
    (now(), c7fceba0-c141-4207-9494-a29f9809de6f, '2017-02-17 12:43:20.234+0400', 'Glad you ran the race in the rain', 'Amy');

INSERT INTO cycling.comments (record_id, id, created_at, comment, commenter) values
    (now(), c7fceba0-c141-4207-9494-a29f9809de6f, '2017-03-22 5:16:59.001+0400', 'Great snacks at all reststops', 'Amy');

INSERT INTO cycling.comments (record_id, id, created_at, comment, commenter) values
    (now(), c7fceba0-c141-4207-9494-a29f9809de6f, '2017-04-01 17:43:08.030+0400', 'Last climb was a killer', 'Amy');

INSERT INTO cycling.comments (record_id, id, created_at, comment, commenter) values
    (now(), 8566eb59-07df-43b1-a21b-666a3c08c08a, totimestamp(now()), 'Fastest womens time ever way to go amy!', 'Maryanne');

INSERT INTO cycling.comments (record_id, id, created_at, comment, commenter) values
    (now(), 8566eb59-07df-43b1-a21b-666a3c08c08a, '2017-02-13 11:20:17.020-0600', 'Great race on a crappy day', 'Maryanne');

INSERT INTO cycling.comments (record_id, id, created_at, comment, commenter) values
    (now(), 8566eb59-07df-43b1-a21b-666a3c08c08a, '2017-03-20 15:45:10.101-0600', 'Saggers really rocked it', 'Maryanne');

INSERT INTO cycling.comments (record_id, id, created_at, comment, commenter) values
    (now(), 8566eb59-07df-43b1-a21b-666a3c08c08a, '2017-04-14 05:16:52.009-0600', 'Not bad for a flatlander', 'Maryanne');

INSERT INTO cycling.comments (record_id, id, created_at, comment, commenter) values
    (now(), fb372533-eb95-4bb4-8685-6ef61e994caa, totimestamp(now()), 'Great course', 'Michael');

INSERT INTO cycling.comments (record_id, id, created_at, comment, commenter) values
    (now(), fb372533-eb95-4bb4-8685-6ef61e994caa, '2017-02-15 18:22:11-0800', 'Some entries complain a lot', 'Michael');

INSERT INTO cycling.comments (record_id, id, created_at, comment, commenter) values
    (now(), fb372533-eb95-4bb4-8685-6ef61e994caa, '2017-03-16 19:43:01.030-0800', 'Getting read for the race', 'Michael');

INSERT INTO cycling.comments (record_id, id, created_at, comment, commenter) values
    (now(), fb372533-eb95-4bb4-8685-6ef61e994caa, '2017-03-22 1:19:44.060-0800', 'Awesome race glad you held it anyway', 'Michael');

INSERT INTO cycling.comments (record_id, id, created_at, comment, commenter) values
    (now(), fb372533-eb95-4bb4-8685-6ef61e994caa, '2017-04-07 11:21:14.001-0800', 'Thanks for waiting for me!', 'Michael');

INSERT INTO cycling.comments (record_id, id, created_at, comment, commenter) values
    (now(), 9011d3be-d358-4a8d-83fc7-3c543789e7, totimestamp(now()), 'Can''t wait for the next race', 'Katarzyna');
```
INSERT INTO cycling.comments (record_id, id , created_at , comment, commenter ) values
(now(), 9011d3be-d35c-4a8d-83f7-a3c543789ee7, '2017-01-01 11:20:17.020-0600', 'Gearing up for the season', 'Katarzyna');
INSERT INTO cycling.comments (record_id, id , created_at , comment, commenter ) values
(now(), 5b6962dd-3f90-4c93-bf61-eabfa4a803e2, totimestamp(now())), 'Thanks for all your hard work', 'Marianne');
INSERT INTO cycling.comments (record_id, id , created_at , comment, commenter ) values
(now(), 220844bf-4860-49d6-9a4b-6b5d3a79cbfb, totimestamp(now())), 'A for effort!', 'Paolo');
INSERT INTO cycling.comments (record_id, id , created_at , comment, commenter ) values
(now(), c4b65263-fe58-4b48-8e68-f0e13d518f, totimestamp(now())), 'Thanks for all your hard work', 'Marianne');
INSERT INTO cycling.comments (record_id, id , created_at , comment, commenter ) values
(now(), 38ab64b6-26cc-4de9-ab28-c257cf011659, totimestamp(now())), 'Next time guys!', 'Marcia');
INSERT INTO cycling.comments (record_id, id , created_at , comment, commenter ) values
(now(), 38ab64b6-26cc-4de9-ab28-c257cf011659, '2017-02-11 22:09:56+0800', 'First race was amazing, can''t wait for more', 'Marcia');
INSERT INTO cycling.comments (record_id, id , created_at , comment, commenter ) values
(now(), 6ab099e5-e68-48d9-a5f8-97e6fb4c9b47, totimestamp(now())), 'So many great races thanks y''all', 'Steven');
INSERT INTO cycling.comments (record_id, id , created_at , comment, commenter ) values
(now(), 6ab099e5-e68-48d9-a5f8-97e6fb4c9b47, '2017-02-02 09:49.02+0800', 'Best of luck everybody I can''t make it', 'Steven');
INSERT INTO cycling.comments (record_id, id , created_at , comment, commenter ) values
(now(), 6ab099e5-e68-48d9-a5f8-97e6fb4c9b47, '2017-04-05 12:00:00+03', 'Bike damaged in transit bummer', 'Steven');
INSERT INTO cycling.comments (record_id, id , created_at , comment, commenter ) values
(now(), e7cd575e-bc0d-4157-a80f-7523add8dabcd, totimestamp(now())), 'Go team, you rocked it', 'Anna');
INSERT INTO cycling.comments (record_id, id , created_at , comment, commenter ) values
(now(), 6d5f1663-89c0-45f8-8cfd-6a373b01622, totimestamp(now())), 'Next year the tour of california!', 'Melissa');
INSERT INTO cycling.comments (record_id, id , created_at , comment, commenter ) values
(now(), 95acd4c-495e-4ed7-b4b5-472f1b9a799b, totimestamp(now())), 'Next year for sure!', 'Vera');
INSERT INTO cycling.comments (record_id, id , created_at , comment, commenter ) values
(now(), 95acd4c-495e-4ed7-b4b5-472f1b9a799b, '2017-02-13 11:40:16.123-0600', 'I can do without the rain@@!', 'Vera');

cyclist_alt_stats
Table

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);

Data

INSERT INTO cycling.cyclist_alt_stats (id, last_race)
VALUES (ed58a99-80f7-4b13-9a90-9dc5571e6821, todate(now()));
### cycling keyspace

| INSERT INTO cycling.cyclist_alt_stats (id, first_race) VALUES (ed584e99-80f7-4b13-9a90-9dc5571e6821, '2006-03-15'); |
| INSERT INTO cycling.cyclist_alt_stats (id, lastname, birthday, nationality, weight, w_units, height, first_race, last_race) VALUES (ed584e99-80f7-4b13-9a90-9dc5571e6821, 'TSATEVICH', '1989-07-05', 'Russia', 64, 'kg', 1.69, '2006-03-15', '2017-04-16'); |
| INSERT INTO cycling.cyclist_alt_stats (id, lastname, birthday, nationality, weight, w_units, height, first_race, last_race) VALUES (a9e96714-2dd0-41f9-8bd0-557196a44ecf, 'ISAYCHEV', '1986-04-21', 'Russia', 80, 'kg', 1.88, '2003-04-22', '2017-03-05'); |
| INSERT INTO cycling.cyclist_alt_stats (id, lastname, birthday, nationality, weight, w_units, height, first_race, last_race) VALUES (a9e96714-2dd0-41f9-8bd0-557196a44ecf, 'ISAYCHEV', '1986-04-21', 'Russia', 80, 'kg', 1.88, '2003-04-22', '2017-03-05'); |
| INSERT INTO cycling.cyclist_alt_stats (id, lastname, birthday, nationality, weight, w_units, height, first_race, last_race) VALUES (a9e96714-2dd0-41f9-8bd0-557196a44ecf, 'ISAYCHEV', '1986-04-21', 'Russia', 80, 'kg', 1.88, '2003-04-22', '2017-03-05'); |
| INSERT INTO cycling.cyclist_alt_stats (id, lastname, birthday, nationality, weight, w_units, height, first_race, last_race) VALUES (a9e96714-2dd0-41f9-8bd0-557196a44ecf, 'ISAYCHEV', '1986-04-21', 'Russia', 80, 'kg', 1.88, '2003-04-22', '2017-03-05'); |
| INSERT INTO cycling.cyclist_alt_stats (id, lastname, birthday, nationality, weight, w_units, height, first_race, last_race) VALUES (a9e96714-2dd0-41f9-8bd0-557196a44ecf, 'ISAYCHEV', '1986-04-21', 'Russia', 80, 'kg', 1.88, '2003-04-22', '2017-03-05'); |
| INSERT INTO cycling.cyclist_alt_stats (id, lastname, birthday, nationality, weight, w_units, height, first_race, last_race) VALUES (a9e96714-2dd0-41f9-8bd0-557196a44ecf, 'ISAYCHEV', '1986-04-21', 'Russia', 80, 'kg', 1.88, '2003-04-22', '2017-03-05'); |

### cyclist_name

Table contains columns with cyclist first and last names.

#### Table and data

```cql
SOURCE '0_create_keyspace.cql';
DROP TABLE IF EXISTS cycling.cyclist_name;

// Create a table with a simple partition key
// START-simple
CREATE TABLE cycling.cyclist_name (id UUID PRIMARY KEY, lastname text, firstname text);
// END-simple
```
Cycling keyspace

// Insert a record that only contains the min values UUID
// START-uuid
INSERT INTO cycling.cyclist_name {
  id
} VALUES {
  uuid()
};
// END-uuid

// Remove the record from the table
TRUNCATE cycling.cyclist_name;

// Insert data used for COPY commands
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-4d60-49d6-9a4b-6b5d39cbfb, 'TIRALONGO', 'Paolo'
};

INSERT INTO cycling.cyclist_name {
  id, lastname, firstname
} VALUES {
  6ab09bec-e68e-48d9-a5f8-97e6fb49b47, 'KRUIKSWIJK', 'Steven'
};

INSERT INTO cycling.cyclist_name {
  id, lastname, firstname
} VALUES {
  fb372533-eb95-4bb4-8685-6ef61e994caa, 'MATTHEWS', 'Michael'
};

cyclist_races

SOURCE 'create_keyspace.cql'

// START-drop
DROP TABLE IF EXISTS cycling.cyclist_races;
DROP TYPE IF EXISTS cycling.race;
// END-drop

/* Find all races for a particular cyclist
CREATE TYPE - User-Defined Type, races
CREATE TABLE WITH LIST, SIMPLE PRIMARY KEY */

// START-udt
CREATE TYPE cycling.race {
  race_title text,
  race_date timestamp,
  race_time time
};
// END-udt

// START-cyclist_races
CREATE TABLE cycling.cyclist_races (  
id UUID PRIMARY KEY,  
lastname text,  
firstname text,  
races list<FROZEN <race>> );  
// END-cyclist_races

// START-insert_udt
INSERT INTO cycling.cyclist_races (  
id,  
races)  
VALUES (  
5b6962dd-3f90-4c93-8f61-eabfa4a803e2,  
[ { race_title:'Rabobank 7-Dorpenomloop Aalburg',  
race_date:'2015-05-09',race_time:'07:00:00' },  
{ race_title:'Ronde van Gelderland', race_date:'2015-04-19',  
race_time:'08:00:00' } ] );  
// END-insert_udt

// START-time
INSERT INTO cycling.cyclist_races (  
id,  
races)  
VALUES (  
5b6962dd-3f90-4c93-8f61-eabfa4a803e2,  
[ { race_time:'07:00:00'},  
{ race_time:'08:00:00' } ] );  
// END-time

// START-utime
UPDATE cycling.cyclist_races  
SET races[1] = { race_time:'06:00:00'}  
WHERE id = 5b6962dd-3f90-4c93-8f61-eabfa4a803e2 ;  
//END-utime

// START-other
INSERT INTO cycling.cyclist_races (id, lastname, firstname, races) VALUES (e7cd5752-bc0d-4157-a80f-7523add8dbcd, 'VAN DER BREGGEN', 'Anna', [ {race_title:'Festival Luxembourg du cyclisme feminin Elsy Jacobs - Prologue - Garnich > Garnich',race_date:'2015-05-01',race_time:'08:00:00'}, {race_title:'Festival Luxembourg du cyclisme feminin Elsy Jacobs - Stage 2 - Garnich > Garnich',race_date:'2015-05-02',race_time:'06:00:00'}, {race_title:'Festival Luxembourg du cyclisme feminin Elsy Jacobs - Stage 3 - Mamer > Mamer',race_date:'2015-05-03',race_time:'06:00:00'} ] );  
// END-other

// START-sall
SELECT * FROM cycling.cyclist_races;  
// END-sall

/* START-rall
END-rall */

// START-spart
SELECT lastname, races FROM cycling.cyclist_races  
WHERE id = e7cd5752-bc0d-4157-a80f-7523add8dbcd;  
// END-spart

// START-rpart  
// END-rpart

**events**

Uses month and date of cycling races to show how to get ranges that span clustering columns.
```cql
DROP table if exists cycling.events;

CREATE TABLE cycling.events (
    Year int,
    Start_Month int,
    Start_Day int,
    End_Month int,
    End_Day int,
    Race TEXT,
    Discipline TEXT,
    Location TEXT,
    UCI_code TEXT,
    PRIMARY KEY ((YEAR, Discipline), Start_Month, Start_Day, Race));

INSERT INTO cycling.events
VALUES (2017,01,01,null,null,'DVV verzekeringen trofee - GP Sven Nys','Cyclo-cross','Baal','C1');

INSERT INTO cycling.events
VALUES (2017,01,01,01,03,'Mitchelton Bay Cycling Classic','Road', 'Geelong, Victoria','CRT');

INSERT INTO cycling.events
VALUES (2017,01,04,01,08,'Cycling Australia Road National Championships', 'Road', 'Ballarat', 'CN');

INSERT INTO cycling.events
VALUES (2017,01,08,null,null,'Belgian Cyclo-cross National Championships','Cyclo-cross','Oostende','CN');

INSERT INTO cycling.events
VALUES (2017,01,08,null,null,'British Cyclo-cross National Championships','Cyclo-cross','Bradford','CN');

INSERT INTO cycling.events
VALUES (2017,01,14,01,17,'Santos Women''s Tour','Road','South Australia','WE');

INSERT INTO cycling.events
VALUES (2017,01,15,01,22,'Tour Down Under','Road','South Australia','WT');

INSERT INTO cycling.events
VALUES (2017,01,23,01,29,'Vuelta Ciclista a la Provincia de San Juan','Road','San Juan',null);

INSERT INTO cycling.events
VALUES (2017,01,26,null,null,'Cadel Evans Great Ocean Road Race - Towards Zero Race Melbourne','Road','Melbourne','CRT');
```
INSERT INTO cycling.events (Year, Start_Month, Start_Day, End_Month, End_Day, Race, Discipline, Location, UCI_code) Values (2017, 01, 26, null, null, 'Challenge Mallorca: Trofeo Porreres-Felanitx-Ses Salines-Campos', 'Road', 'Mallorca', null);
INSERT INTO cycling.events (Year, Start_Month, Start_Day, End_Month, End_Day, Race, Discipline, Location, UCI_code) Values (2017, 01, 28, null, null, 'Challenge Mallorca: Trofeo Serra de Tramuntana -2017', 'Road', 'Mallorca', null);
INSERT INTO cycling.events (Year, Start_Month, Start_Day, End_Month, End_Day, Race, Discipline, Location, UCI_code) Values (2017, 01, 28, null, null, 'Cadel Evans Great Ocean Road Race', 'Road', 'Geelong', 'WE');
INSERT INTO cycling.events (Year, Start_Month, Start_Day, End_Month, End_Day, Race, Discipline, Location, UCI_code) Values (2017, 01, 29, null, null, 'Challenge Mallorca: Trofeo Andratx-Mirador des Colomer', 'Road', 'Mallorca', null);
INSERT INTO cycling.events (Year, Start_Month, Start_Day, End_Month, End_Day, Race, Discipline, Location, UCI_code) Values (2017, 01, 29, 01, 29, 'UCI Cyclo-cross World Championships', 'Cyclo-cross', 'Bieles', 'CM');
INSERT INTO cycling.events (Year, Start_Month, Start_Day, End_Month, End_Day, Race, Discipline, Location, UCI_code) Values (2017, 01, 29, null, null, 'Cadel Evans Great Ocean Road Race', 'Road', 'Geelong', 'WT');
INSERT INTO cycling.events (Year, Start_Month, Start_Day, End_Month, End_Day, Race, Discipline, Location, UCI_code) Values (2017, 01, 29, null, null, 'Grand Prix Cycliste la Marseillaise', 'Road', 'France', null);
INSERT INTO cycling.events (Year, Start_Month, Start_Day, End_Month, End_Day, Race, Discipline, Location, UCI_code) Values (2017, 01, 29, 02, 04, 'Dubai Tour', 'Road', 'Dubai', '2.1', 'HC');
INSERT INTO cycling.events (Year, Start_Month, Start_Day, End_Month, End_Day, Race, Discipline, Location, UCI_code) Values (2017, 01, 31, 02, 03, 'Ladies Tour of Qatar', 'Road', 'Qatar', 'WE');
INSERT INTO cycling.events (Year, Start_Month, Start_Day, End_Month, End_Day, Race, Discipline, Location, UCI_code) Values (2017, 02, 01, 02, 05, 'Jayco Herald Sun Tour', 'Road', 'Victoria', null);
INSERT INTO cycling.events (Year, Start_Month, Start_Day, End_Month, End_Day, Race, Discipline, Location, UCI_code) Values (2017, 02, 01, 02, 05, 'Volta a la Comunitat Valenciana', 'Road', 'Valencia', null);
INSERT INTO cycling.events (Year, Start_Month, Start_Day, End_Month, End_Day, Race, Discipline, Location, UCI_code) Values (2017, 02, 04, null, null, 'Superprestige - Hoogstraten - 2017', 'Cyclo-cross', 'Lille', 'C1');
// second
INSERT INTO cycling.events (Year, Start_Month, Start_Day, End_Month, End_Day, Race, Discipline, Location, UCI_code) Values (2017, 02, 04, null, null, 'DVV verzekeringen trofee - Krawatencross', 'Cyclo-cross', 'Lille', 'C1');
INSERT INTO cycling.events (Year, Start_Month, Start_Day, End_Month, End_Day, Race, Discipline, Location, UCI_code) Values (2017, 02, 05, null, null, 'G.P. Costa degli Etruschi', 'Road', 'Italy', null);
INSERT INTO cycling.events (Year, Start_Month, Start_Day, End_Month, End_Day, Race, Discipline, Location, UCI_code) Values (2017, 02, 04, null, null, 'Superprestige - Hoogstraten - 2017', 'Cyclo-cross', 'Belgium', 'C1');
INSERT INTO cycling.events (Year, Start_Month, Start_Day, End_Month, End_Day, Race, Discipline, Location, UCI_code) Values (2017, 02, 06, 02, 10, 'Tour of Qatar', 'Road', 'Qatar', 'WT');
INSERT INTO cycling.events (Year, Start_Month, Start_Day, End_Month, End_Day, Race, Discipline, Location, UCI_code) Values (2017, 02, 09, 02, 12, 'South African Road Championships', 'Road', 'Western Cape', 'CN');
INSERT INTO cycling.events (Year, Start_Month, Start_Day, End_Month, End_Day, Race, Discipline, Location, UCI_code) Values
<table>
<thead>
<tr>
<th>Year</th>
<th>Start_Month</th>
<th>Start_Day</th>
<th>End_Month</th>
<th>End_Day</th>
<th>Race</th>
<th>Discipline</th>
<th>Location</th>
<th>UCI_code</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017</td>
<td>02</td>
<td>11</td>
<td>null</td>
<td>null</td>
<td>'Superprestige - Middelkerke - 2017'</td>
<td>'Cyclo-cross'</td>
<td>'Middelkerke'</td>
<td>'C1'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>INSERT INTO cycling.events (Year,Start_Month,Start_Day,End_Month,End_Day,Race,Discipline,Location,UCI_code) Values (2017,02,11,null,null,'Superprestige - Middelkerke - 2017','Cyclo-cross','Middelkerke','C1');</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>INSERT INTO cycling.events (Year,Start_Month,Start_Day,End_Month,End_Day,Race,Discipline,Location,UCI_code) Values (2017,02,12,null,null,'Clasica de Almeria','Road','Almeria','1.1');</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>INSERT INTO cycling.events (Year,Start_Month,Start_Day,End_Month,End_Day,Race,Discipline,Location,UCI_code) Values (2017,02,15,02,19,'Volta ao Algarve em Bicicleta','Road','Algarve','HC');</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>INSERT INTO cycling.events (Year,Start_Month,Start_Day,End_Month,End_Day,Race,Discipline,Location,UCI_code) Values (2017,02,22,03,01,'Le Tour de Langkawi','Road','Langkawi','HC');</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>INSERT INTO cycling.events (Year,Start_Month,Start_Day,End_Month,End_Day,Race,Discipline,Location,UCI_code) Values (2017,02,25,null,null,'Kuurne-Brussel-Kuurne','Road','Kuurne','HC');</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>INSERT INTO cycling.events (Year,Start_Month,Start_Day,End_Month,End_Day,Race,Discipline,Location,UCI_code) Values (2017,02,26,null,null,'Omloop Het Nieuwsblad Elite','Road','Belgium','WT');</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>INSERT INTO cycling.events (Year,Start_Month,Start_Day,End_Month,End_Day,Race,Discipline,Location,UCI_code) Values (2017,03,01,null,null,'Le Samyn des Dames','Road','Hainaut','WE');</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>INSERT INTO cycling.events (Year,Start_Month,Start_Day,End_Month,End_Day,Race,Discipline,Location,UCI_code) Values (2017,03,04,null,null,'Strade Bianche','Road','Tuscany','WT');</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Year</td>
<td>Start Month</td>
<td>Start Day</td>
<td>End Month</td>
<td>End Day</td>
<td>Race</td>
<td>Discipline</td>
<td>Location</td>
<td>UCI_code</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>-----------</td>
<td>-----------</td>
<td>---------</td>
<td>--------------------</td>
<td>------------</td>
<td>-----------</td>
<td>----------</td>
</tr>
<tr>
<td>2017</td>
<td>03</td>
<td>04</td>
<td>null</td>
<td>null</td>
<td>Strade Bianche Women</td>
<td>Road</td>
<td>Tuscany</td>
<td>WWT</td>
</tr>
<tr>
<td>INSERT INTO cycling.events (2017, 03, 05, null, null, 'Dwars door West-Vlaanderen Johan Museeuw Classic', 'Road', 'Flanders', '1.1');</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSERT INTO cycling.events (2017, 03, 05, null, null, 'GP Industria and Artigianato', 'Road', 'Larciano', 'HC');</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSERT INTO cycling.events (2017, 03, 05, null, null, 'Paris - Nice', 'Road', 'France', 'WT');</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSERT INTO cycling.events (2017, 08, 03, 11, 'Semana Ciclista Valenciana', 'Road', 'Valencia', 'WE');</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSERT INTO cycling.events (2017, 08, 03, 14, 'Tirreno-Adriatico', 'Road', 'Italy', 'WT');</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSERT INTO cycling.events (2017, 03, 08, 03, 14, 'Oceania Championships', 'Road', 'Canberra', 'CC');</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSERT INTO cycling.events (2017, 03, 11, null, null, 'Ronde van Drenthe', 'Road', 'Netherlands', 'WWT');</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSERT INTO cycling.events (2017, 03, 12, null, null, 'Drentse Acht van Westerveld', 'Road', 'Dwingeloo', 'WE');</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSERT INTO cycling.events (2017, 03, 15, null, null, 'Nokere Koerse – Danilith Classic', 'Road', 'Flanders', 'HC');</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSERT INTO cycling.events (2017, 03, 17, null, null, 'Handzame Classic', 'Road', 'Flanders', '1.1');</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSERT INTO cycling.events (2017, 03, 18, null, null, 'Classic Loire Atlantique', 'Road', 'Loire', '1.1');</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSERT INTO cycling.events (2017, 03, 18, null, null, 'Milan-San Remo', 'Road', 'Italy', 'WT');</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSERT INTO cycling.events (2017, 03, 19, 03, 26, 'Cape Epic', 'Mountain Bike', 'Cape Town', 'SHC');</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSERT INTO cycling.events (2017, 03, 19, null, null, 'Trofeo Alfredo Binda Comune di Cittiglio', 'Road', 'Italy', 'WWT');</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSERT INTO cycling.events (2017, 03, 20, 03, 26, 'Volta Ciclista a Catalunya', 'Road', 'Catalunya', 'WT');</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSERT INTO cycling.events (2017, 03, 22, null, null, 'Dwers Door Vlaanderen', 'Road', 'Flanders', 'WT');</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSERT INTO cycling.events (2017, 03, 23, null, null, 'Settimana Internazionale Coppi e Bartali', 'Road', 'Emilia-Romagna', '2.1');</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSERT INTO cycling.events (2017, 03, 24, null, null, 'E3 Harelbeke', 'Road', 'Flanders', 'WT');</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSERT INTO cycling.events (2017, 03, 26, null, null, 'Gent Wevelgem', 'Road', 'Flanders', 'WT');</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cycling keyspace

```cql
INSERT INTO cycling.events (Year, Start_Month, Start_Day, End_Month, End_Day, Race, Discipline, Location, UCI_code) Values (2017, 03, 26, null, null, 'Gent Wevelgem Women', 'Road', 'Belgium', 'WWT');
INSERT INTO cycling.events (Year, Start_Month, Start_Day, End_Month, End_Day, Race, Discipline, Location, UCI_code) Values (2017, 03, 28, 03, 30, 'Driedaagse De Panne-Koksijde', 'Road', 'West Flanders', 'HC');
INSERT INTO cycling.events (Year, Start_Month, Start_Day, End_Month, End_Day, Race, Discipline, Location, UCI_code) Values (2017, 03, 29, null, null, 'Pajot Hills Classic', 'Road', 'Flemish Brabant', 'WE');
```

**race_times**

```
source '0_create_keyspace.cql';
DROP TABLE IF EXISTS cycling.race_times;

// START-TABLE
CREATE TABLE cycling.race_times (  
race_name text,  
race_time time,  
finish_time duration,  
race_date date,  
cyclist_name text,  
PRIMARY KEY (race_date, race_name, cyclist_name));
// END-TABLE

// START-DATA
INSERT INTO cycling.race_times (race_name, cyclist_name, race_time, race_date) VALUES ('17th Santos Tour Down Under', 'Rohan DENNIS', '19:15:18', '2017-04-14');
INSERT INTO cycling.race_times (race_name, cyclist_name, race_time, race_date) VALUES ('17th Santos Tour Down Under', 'Richie PORTE', '19:15:20', '2017-04-14');
INSERT INTO cycling.race_times (race_name, cyclist_name, race_time, race_date) VALUES ('17th Santos Tour Down Under', 'Cadel EVANS', '19:15:38', '2017-04-14');
INSERT INTO cycling.race_times (race_name, cyclist_name, race_time, race_date) VALUES ('17th Santos Tour Down Under', 'Tom DUMOULIN', '19:15:40', '2017-04-14');
// END-DATA

/ START-select_race_date_group_by
SELECT  
race_date, race_name  
FROM  
cycling.race_times  
GROUP BY  
race_date;
// END-select_race_date_group_by
```

**rank_by_year_and_name**

```
SOURCE '0_create_keyspace.cql';

// Store cycling race information by year and race name using a COMPOSITE PARTITION KEY

// START-drop_rank_by_year_and_name
DROP TABLE IF EXISTS cycling.rank_by_year_and_name;
// END-drop_rank_by_year_and_name

// START-compositepk
CREATE TABLE cycling.rank_by_year_and_name (  
race_year int,  
race_name text,  
cyclist_name text,  
race_code text,  
race_date date,  
race_time time,  
finish_time duration,  
rank int,  
PRIMARY KEY ((race_year, race_name), rank)
```
Cycling keyspace

// END-compositepk

// START-insert_rows_into_rank_by_year_and_name
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 (2014, 'Tour of Japan - Stage 4 - Minami > Shinshu', 'Benjamin PRADES', 3);

INSERT INTO cycling.rank_by_year_and_name (race_year, race_name, cyclist_name, rank)
VALUES (2014, 'Tour of Japan - Stage 4 - Minami > Shinshu', 'Daniel MARTIN', 1);

INSERT INTO cycling.rank_by_year_and_name (race_year, race_name, cyclist_name, rank)
VALUES (2014, 'Tour of Japan - Stage 4 - Minami > Shinshu', 'Johan Esteban CHAVES', 2);

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)
Cycling keyspace

VALUES (2014, '4th Tour of Beijing', 'Johan Esteban CHAVES', 3);

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);

// END-insert_rows_into_rank_by_year_and_name

// START-select_all_from_rank_by_year_and_name
SELECT *
FROM cycling.rank_by_year_and_name;
// END-select_all_from_rank_by_year_and_name

// START-select_from_rank_by_year_and_name_with_race_year_and_name
SELECT *
FROM cycling.rank_by_year_and_name
WHERE race_year = 2015
AND race_name = 'Tour of Japan - Stage 4 - Minami > Shinshu';
// END-select_from_rank_by_year_and_name_with_race_year_and_name

// START-create_index_race_year
CREATE INDEX ryear ON cycling.rank_by_year_and_name (race_year);
// END-create_index_race_year

// START-select_from_rank_by_year_and_name_with_year
SELECT *
FROM cycling.rank_by_year_and_name
WHERE race_year = 2015;
// END-select_from_rank_by_year_and_name_with_year

// START-update_rank_by_year_and_name_simple
UPDATE cycling.rank_by_year_and_name
SET cyclist_name = 'John SMITH'
WHERE race_year = 2015
AND race_name = 'Tour of Japan - Stage 4 - Minami > Shinshu'
AND rank = 2;
// END-update_rank_by_year_and_name_simple

// START-update_rank_by_year_and_name_in_clause
UPDATE cycling.rank_by_year_and_name
SET cyclist_name = 'Jane DOE'
WHERE race_year = 2015
AND race_name = 'Tour of Japan - Stage 4 - Minami > Shinshu'
AND rank IN (2, 3, 4);
// END-update_rank_by_year_and_name_in_clause

// START-update_rank_by_year_and_name_if_clause
UPDATE cycling.rank_by_year_and_name
SET cyclist_name = 'John SMITH'
WHERE race_year = 2015
AND race_name = 'Tour of Japan - Stage 4 - Minami > Shinshu'
AND rank = 3
IF EXISTS;

// END-update_rank_by_year_and_name_if_clause

// START-delete_from_rank_by_year_and_name_column
DELETE cyclist_name
FROM cycling.rank_by_year_and_name
WHERE race_year = 2015
  AND race_name = 'Tour of Japan - Stage 4 - Minami > Shinshu'
  AND rank = 3;
// END-delete_from_rank_by_year_and_name_column

// START-delete_from_rank_by_year_and_name_row
DELETE
FROM cycling.rank_by_year_and_name
WHERE race_year = 2015
  AND race_name = 'Tour of Japan - Stage 4 - Minami > Shinshu'
  AND rank = 3;
// END-delete_from_rank_by_year_and_name_row

// START-delete_from_rank_by_year_and_name_multiple_rows
DELETE
FROM cycling.rank_by_year_and_name
WHERE race_year = 2015
  AND race_name = 'Tour of Japan - Stage 4 - Minami > Shinshu';
// END-delete_from_rank_by_year_and_name_multiple_rows

// START-delete_from_rank_by_year_and_name_if_clause
DELETE
FROM cycling.rank_by_year_and_name
WHERE race_year = 2014
  AND race_name = '4th Tour of Beijing'
  AND rank = 3
IF EXISTS;
// END-delete_from_rank_by_year_and_name_if_clause

// START-truncate_rank_by_year_and_name
TRUNCATE cycling.rank_by_year_and_name;
// END-truncate_rank_by_year_and_name

// Repopulate the table
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 (2014, 'Tour of Japan - Stage 4 - Minami > Shinshu', 'Benjamin PRADES', 3);
<table>
<thead>
<tr>
<th>Race Year</th>
<th>Race Name</th>
<th>Cyclist Name</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>2014</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>Daniel MARTIN</td>
<td>1</td>
</tr>
<tr>
<td>2014</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>Johan Esteban CHAVES</td>
<td>2</td>
</tr>
<tr>
<td>2015</td>
<td>Giro d'Italia - Stage 11 - Forli &gt; Imola</td>
<td>Ilnur ZAKARIN</td>
<td>1</td>
</tr>
<tr>
<td>2015</td>
<td>Giro d'Italia - Stage 11 - Forli &gt; Imola</td>
<td>Carlos BETANCUR</td>
<td>2</td>
</tr>
<tr>
<td>2014</td>
<td>4th Tour of Beijing</td>
<td>Phillippe GILBERT</td>
<td>1</td>
</tr>
<tr>
<td>2014</td>
<td>4th Tour of Beijing</td>
<td>Daniel MARTIN</td>
<td>2</td>
</tr>
<tr>
<td>2014</td>
<td>4th Tour of Beijing</td>
<td>Johan Esteban CHAVES</td>
<td>3</td>
</tr>
<tr>
<td>2014</td>
<td>4th Tour of Beijing</td>
<td>Phillippe GILBERT</td>
<td>1</td>
</tr>
<tr>
<td>2014</td>
<td>4th Tour of Beijing</td>
<td>Daniel MARTIN</td>
<td>2</td>
</tr>
<tr>
<td>2014</td>
<td>4th Tour of Beijing</td>
<td>Johan Esteban CHAVES</td>
<td>3</td>
</tr>
</tbody>
</table>