Contents

Chapter 1. About CQL..................................................................................................................10
Chapter 2. CQL quick reference.................................................................................................11
Chapter 3. CQL data modeling..................................................................................................18
  Data modeling concepts...........................................................................................................18
  Data modeling analysis...........................................................................................................21
  Using materialized views.........................................................................................................22
    Understanding materialized views.......................................................................................22
    Known limitations.................................................................................................................24
    Creating a materialized view...............................................................................................24
    Altering a materialized view...............................................................................................27
    Dropping a materialized view.............................................................................................27
    Best practices.......................................................................................................................28
    FAQ......................................................................................................................................28
Chapter 4. Introduction to CQL.................................................................................................29
  Accessing data using CQL.......................................................................................................29
    Starting standalone cqlsh.....................................................................................................29
    Starting cqlsh.....................................................................................................................30
  Understanding the CQL command syntax...........................................................................31
    Uppercase and lowercase.....................................................................................................31
    Valid characters in names....................................................................................................32
    Keyspace qualifier...............................................................................................................32
    Escaping characters............................................................................................................33
    Valid literals.........................................................................................................................33
    Exponential notation..........................................................................................................34
    Code comments...................................................................................................................34
    Keywords..............................................................................................................................35
    Limitations............................................................................................................................39
  About system keyspace tables...............................................................................................40
    Keyspace, table, and column information.........................................................................42
    Querying system tables for cluster topology information................................................44
    Functions, aggregates, and user types...............................................................................44
  About virtual keyspaces and tables.......................................................................................46
Chapter 5. Managing database resources

Keyspaces

About keyspaces
Creating a keyspace
Modifying the replication factor
Dropping a keyspace

Types, functions and aggregates
Creating a user-defined function (UDF)
Creating a user-defined aggregate function (UDA)
Modifying a user-defined type (UDT)
Dropping a user-defined function (UDF)

Tables
Creating a table
Creating a counter table
Fixing a table schema collision
Altering a table
Dropping a keyspace or table

Indexing tables

Search Indexes
About search commands
Adjusting timeout for index management
Creating an index
Managing search index fields
Configuring search index joins
Reloading the search index
Removing a search index
Updating the index after data expires (TTL)
Inserting/updating data

Chapter 6. Inserting and updating data
Using INSERT and UPDATE to write values
UUID and timeuuid column
Set column
List fields
Map fields
Tuple column
Joining cores .............................................................................................................................. 188
Spatial queries with polygons require JTS .............................................................................. 191
UDT query examples ............................................................................................................... 192
Querying CQL collections ........................................................................................................ 194
Using date ranges in solr_query ............................................................................................. 195
Restricted query routing ........................................................................................................... 199
Returning collections ............................................................................................................. 200
Aggregating results .................................................................................................................. 201
Applying user-defined aggregate (UDA) functions ............................................................... 202
Using user-defined functions (UDF) ...................................................................................... 202
Returning the write timestamp ............................................................................................... 203
Formatting query results as JSON ............................................................................................ 203

Chapter 8. cqlsh reference .................................................................................................... 205
cqlsh (startup options) .......................................................................................................... 205
Configuring a cqlshrc file ....................................................................................................... 208
CAPTURE .................................................................................................................................. 215
CLEAR ....................................................................................................................................... 216
CONSISTENCY .......................................................................................................................... 217
COPY TO .................................................................................................................................... 221
COPY FROM ................................................................................................................................ 226
DESCRIBE AGGREGATE ........................................................................................................... 232
DESCRIBE CLUSTER ................................................................................................................. 233
DESCRIBE SCHEMA ................................................................................................................ 234
DESCRIBE FUNCTION ............................................................................................................. 235
DESCRIBE INDEX ...................................................................................................................... 237
DESCRIBE KEYSpace .............................................................................................................. 238
DESCRIBE MATERIALIZED VIEW ............................................................................................. 239
DESCRIBE SEARCH INDEX .................................................................................................... 241
DESCRIBE TABLE .................................................................................................................... 244
DESCRIBE TYPE ..................................................................................................................... 245
EXECUTE AS ............................................................................................................................ 247
EXPAND ..................................................................................................................................... 248
EXIT .......................................................................................................................................... 250
LOGIN ....................................................................................................................................... 251
PAGING ..................................................................................................................................... 252
Chapter 9. CQL reference

Introduction ........................................................................................................................................ 262
CQL data types ................................................................................................................................ 262
  Date, time, and timestamp format .................................................................................................... 266
Solr field type classes ....................................................................................................................... 269
CQL Operators .................................................................................................................................. 272
CQL native functions ........................................................................................................................ 273
CQL native aggregates ...................................................................................................................... 278
CQL commands .................................................................................................................................. 280
  ALTER KEYSPACE ........................................................................................................................... 280
  ALTER MATERIALIZED VIEW ........................................................................................................... 282
  ALTER ROLE ..................................................................................................................................... 284
  ALTER SEARCH INDEX CONFIG ..................................................................................................... 285
  ALTER SEARCH INDEX SCHEMA ..................................................................................................... 294
  ALTER TABLE .................................................................................................................................... 303
  ALTER TYPE .................................................................................................................................... 309
  ALTER USER ...................................................................................................................................... 310
BATCH ................................................................................................................................................ 311
COMMIT SEARCH INDEX .................................................................................................................. 316
CREATE AGGREGATE ........................................................................................................................ 318
CREATE CUSTOM INDEX (SASI) .......................................................................................................... 321
CREATE FUNCTION ............................................................................................................................. 325
CREATE INDEX ..................................................................................................................................... 327
CREATE KEYSspace ............................................................................................................................ 331
CREATE MATERIALIZED VIEW .......................................................................................................... 334
CREATE ROLE ....................................................................................................................................... 341
CREATE SEARCH INDEX .................................................................................................................... 343
CREATE TABLE ..................................................................................................................................... 355
CREATE TYPE ....................................................................................................................................... 365
CREATE USER (Deprecated) ................................................................................................................ 367
DELETE ................................................................................................................................................ 369
Search index examples.................................................................................................................................................. 509
Creating a demo keyspace for tutorials.................................................................................................................................. 509
Multi-faceted search using healthcare data.............................................................................................................................. 510
Quick Start for CQL index management................................................................................................................................... 515
Term and phrase searches using the wikipedia demo.................................................................................................................. 518
Indexing and querying polygons............................................................................................................................................. 521
Chapter 1. About CQL

CQL for DataStax Enterprise 6.7 provides information about the Cassandra Query Language (CQL). CQL is the primary language for communicating with the database.

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

DataStax Enterprise 6.7 documentation and other information sources

<table>
<thead>
<tr>
<th>DataStax Enterprise 6.7 documentation and other information sources</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Architecture Guide</strong></td>
</tr>
<tr>
<td><strong>Administrators Guide</strong></td>
</tr>
<tr>
<td><strong>Developers Guide</strong></td>
</tr>
<tr>
<td><strong>OpsCenter</strong></td>
</tr>
</tbody>
</table>

Other information sources

| **Landing pages** | Getting started with DSE, supported platforms, product compatibility, third-party software, resources for additional information, and earlier documentation. |
| **DSE drivers** | C/C++ driver, C# driver, Java driver, Node.js driver, PHP driver, Python driver, and Ruby driver. |
| **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 DSE, and installing. |
| **Upgrade Guide** | Information on upgrading various versions of DataStax Enterprise and upgrading from Apache Cassandra to DataStax Enterprise. |
| **Sources of support** | DataStax Support, DataStax Academy forums, Stackoverflow for DataStax Enterprise, Stackoverflow for the DataStax Java client driver and the DataStax PHP driver. |
| **DataStax Academy** | DataStax Academy for courses on CQL, database administration, data modeling, and others. |
Chapter 2. 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 | DDAC.

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

**ALTER KEYSPACE**

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

Learn more.

**ALTER MATERIALIZED VIEW**

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

Learn more.

**ALTER ROLE**

```
ALTER ROLE role_name
    [ WITH [ PASSWORD = role_password ]
    [ [ AND ] LOGIN = ( true | false ) ]
    [ [ AND ] SUPERUSER = ( true | false ) ]
    [ [ AND ] OPTIONS = { option_map } ] ] ;
```

Learn more.

**ALTER SEARCH INDEX CONFIG**

```
ALTER SEARCH INDEX CONFIG ON [keyspace_name.]table_name
    ( ADD element_path [ attribute_list ] WITH $$ json_map $$
    SET element_identifier = 'value'
    SET shortcut = value
    DROP element_identifier
    DROP shortcut ) ;
```

Learn more.

**ALTER SEARCH INDEX SCHEMA**

```
ALTER SEARCH INDEX SCHEMA ON [keyspace_name.]table_name
    ( ADD field column_name
    ADD element_path [ attribute_list ] WITH $$ json_map $$
    SET element_identifier = 'value'
    DROP field field_name
    DROP element_identifier ) ;
```

Learn more.
## CQL quick reference

### ALTER TABLE

```
ALTER TABLE {keyspace_name.}table_name
    [ ADD ( column_definition | column_definition_list ) [ , ... ] ]
    [ DROP column_name [ , ... ] ]
    [ RENAME column_name TO column_name ]
    [ WITH table_properties [ , ... ] ];
```

Learn more.

### ALTER TYPE

```
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.

### BATCH

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

Learn more.

### COMMIT SEARCH INDEX

```
COMMIT SEARCH INDEX ON {keyspace_name.}table_name ;
```

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 CUSTOM INDEX

```
CREATE CUSTOM INDEX [ IF NOT EXISTS ] [ index_name ]
    ON {keyspace_name.}table_name (column_name)
    USING 'org.apache.cassandra.index.sasi.SASIIndex'
    [ WITH OPTIONS = { option_map } ] ;
```

Learn more.

### CREATE FUNCTION

```
CREATE [ OR REPLACE ] FUNCTION [ IF NOT EXISTS ] [keyspace_name.]function_name (argument_list
    [ , ... ])
```

Learn more.
CREATE INDEX

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

CREATE KEYSPACE

CREATE KEYSPACE [ IF NOT EXISTS ] keyspace_name
WITH REPLICATION = { replication_map }
[ AND DURABLE_WRITES = ( true | false ) ] ;

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 SEARCH INDEX

CREATE SEARCH INDEX [ IF NOT EXISTS ] ON [keyspace_name.]table_name
[ WITH [ COLUMNS column_list { option : value } [ , ... ] ]
[ AND ] PROFILES profile_name [ , ... ]
[ AND ] CONFIG { option:value } [ , ... ]
[ AND ] OPTIONS { option:value } [ , ... ] ] ;

Learn more.
**CREATE TABLE**

```sql
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' ] ] ;
```

Learn more.

**CREATE TYPE**

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

Learn more.

**DELETE**

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

Learn more.

**DROP AGGREGATE**

```sql
DROP AGGREGATE [ IF EXISTS ] [keyspace_name.]aggregate_name [ (argument_name [ , ... ]) ] ;
```

Learn more.

**DROP FUNCTION**

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

Learn more.

**DROP INDEX**

```sql
DROP INDEX [ IF EXISTS ] [keyspace_name.]index_name ;
```

Learn more.

**DROP KEYSSPACE**

```sql
DROP KEYSSPACE [ IF EXISTS ] keyspace_name ;
```

Learn more.

**DROP MATERIALIZED VIEW**

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

Learn more.
DROP ROLE

DROP ROLE [ IF EXISTS ] role_name;

Learn more.

DROP SEARCH INDEX CONFIG

DROP SEARCH INDEX ON [keyspace_name.]table_name
  OPTIONS { option : value } [ , { option : value } ... ] ;

Learn more.

DROP TABLE

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

Learn more.

DROP TYPE

DROP TYPE [ IF EXISTS ] [keyspace_name.]type_name;

Learn more.

GRANT

GRANT permission
  ON object
  TO role_name;

Learn more.

INSERT

INSERT [ JSON ] 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 ( ALL PERMISSIONS | permission_list )
  [ ON resource_name ]
  [ OF role_name ]
  [ NORECURSIVE ] ;

Learn more.

LIST ROLES

LIST ROLES
  [ OF role_name ]
<table>
<thead>
<tr>
<th>Function</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REBUILD SEARCH INDEX</td>
<td><code>REBUILD SEARCH INDEX ON [keyspace_name.]table_name</code> WITH OPTIONS {</td>
<td>Learn more.</td>
</tr>
<tr>
<td></td>
<td>deleteAll : (true</td>
<td>false) }</td>
</tr>
<tr>
<td>RELOAD SEARCH INDEX</td>
<td><code>RELOAD SEARCH INDEX ON [keyspace_name.]table_name</code> ;</td>
<td>Learn more.</td>
</tr>
<tr>
<td>RESTRICT</td>
<td><code>RESTRICT permission ON [keyspace_name.]table_name TO role_name ;</code></td>
<td>Learn more.</td>
</tr>
<tr>
<td>RESTRICT ROWS</td>
<td><code>RESTRICT ROWS ON [keyspace_name.]table_name USING pk_column_name ;</code></td>
<td>Learn more.</td>
</tr>
<tr>
<td>REVOKE</td>
<td><code>REVOKE permission ON resource_name FROM role_name ;</code></td>
<td>Learn more.</td>
</tr>
<tr>
<td>SELECT</td>
<td><code>SELECT [ JSON ] selectors FROM [keyspace_name.]table_name WHERE</code></td>
<td>CQL for DataStax Enterprise 6.7</td>
</tr>
<tr>
<td></td>
<td>[ primary_key_conditions ] AND [ index_conditions ] GROUP BY`</td>
<td>Latest DSE version Latest 6.7</td>
</tr>
<tr>
<td></td>
<td>column_name [ , ... ] ORDER BY [ PK_column_name [ , ... ] (ASC</td>
<td>patch: 6.7.7</td>
</tr>
<tr>
<td></td>
<td>DESC )</td>
<td>( LIMIT N</td>
</tr>
<tr>
<td>Command</td>
<td>Syntax</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>TRUNCATE</strong></td>
<td>TRUNCATE [ TABLE ] [keyspace_name.]table_name;</td>
<td></td>
</tr>
</tbody>
</table>
| **UNRESTRICT**| UNRESTRICT permission_name  
|              | ON [keyspace_name.]table_name  
|              | FROM role_name;                                                       |
| **UPDATE**    | UPDATE [keyspace_name.]table_name  
|              | [ USING TTL time_value ]  
|              | [ [ AND ] USING TIMESTAMP timestamp_value ]  
|              | SET assignment [ , assignment ... ]  
|              | WHERE row_specification  
|              | [ IF EXISTS | IF condition [ AND condition ] ] ;                            |
| **USE**       | USE keyspace_name ;                                                   |
Chapter 3. 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 Enterprise 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. DataStax Enterprise'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.

DataStax Enterprise'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 DataStax Enterprise 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.

DataStax Enterprise 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 DataStax Enterprise 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.
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 and the number of columns, primary key columns and static columns in each table. Each application will have different efficiency parameters, but a good rule of thumb is to keep the maximum number of values below 100,000 items and the disk size under 100MB.

Data redundancy must be considered as well. Two redundancies that are a consequence of DataStax Enterprise'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 table, 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.

DataStax Enterprise 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 DataStax Enterprise, 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

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 Enterprise (DSE) 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 DSE 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 DSE 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.
CQL data modeling

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.

- Currently, there is no 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)

**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, which causes 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 examples**

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

```sql
CREATE TABLE IF NOT EXISTS cycling.cyclist_base {
  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.

<table>
<thead>
<tr>
<th>cid</th>
<th>age</th>
<th>birthday</th>
<th>country</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ffdfa2a7-5fc6-49a7-bfcd-3fcdcfd7156</td>
<td>18</td>
<td>1997-02-08</td>
<td>Netherlands</td>
<td>Pascal EENKHOORN</td>
</tr>
<tr>
<td>15a116fc-b833-4da6-ab9a-4a7775752836</td>
<td>18</td>
<td>1997-08-19</td>
<td>United States</td>
<td>Adrien COSTA</td>
</tr>
<tr>
<td>e7ae5cfc-d358-4d99-b900-85902da9bb0</td>
<td>22</td>
<td>1993-06-18</td>
<td>New Zealand</td>
<td>Alex FRAME</td>
</tr>
<tr>
<td>c99c484-5e4a-4542-8d047a01b8a8</td>
<td>27</td>
<td>1987-09-04</td>
<td>Brazil</td>
<td>Cristian EGIDIO</td>
</tr>
<tr>
<td>41091ef-bd1b-4efa-8256-b0fd8ab67029</td>
<td>19</td>
<td>1999-01-04</td>
<td>Uzbekistan</td>
<td>Iskandarbek SHODIEV</td>
</tr>
<tr>
<td>d1aad83b-be60-47a4-bd6e-069b8da0d97b</td>
<td>27</td>
<td>1987-09-04</td>
<td>Germany</td>
<td>Johannes HEIDER</td>
</tr>
<tr>
<td>862cc51f-00a1-4d5a-976b-a359cab7300e</td>
<td>20</td>
<td>1994-09-04</td>
<td>Denmark</td>
<td>Joakim BUKGAL</td>
</tr>
<tr>
<td>1c526b49-d3a2-42a3-bcf9-7903c80b3d16</td>
<td>19</td>
<td>1998-12-23</td>
<td>Australia</td>
<td>Kanden GROVES</td>
</tr>
</tbody>
</table>
The `cyclist_mv` table can be the basis of a materialized view that uses `age` in the primary key.

```cql
CREATE MATERIALIZED VIEW IF NOT EXISTS 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';
```

This `CREATE MATERIALIZED VIEW` statement has several features:

- The `AS SELECT` phrase identifies the columns copied from the base table to the materialized view.
- The `FROM` phrase identifies the source table from which 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 the source table `cyclist_mv` 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.

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

```cql
SELECT age, name, birthday
FROM cycling.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>Adrien 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>

(3 rows)

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

```cql
CREATE MATERIALIZED VIEW IF NOT EXISTS cycling.cyclist_by_birthday
AS SELECT age, birthday, name, country
FROM cycling.cyclist_base
```
CQL data modeling

```
WHERE birthday IS NOT NULL
   AND cid IS NOT NULL
PRIMARY KEY (birthday, cid);
```

```
CREATE MATERIALIZED VIEW IF NOT EXISTS cycling.cyclist_by_country
   AS SELECT age, birthday, name, country
   FROM cycling.cyclist_base
   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 cycling.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>1987-09-04</td>
</tr>
<tr>
<td>27</td>
<td>Johannes HEIDER</td>
<td>1987-09-04</td>
</tr>
</tbody>
</table>

(2 rows)

```
SELECT age, name, birthday
FROM cycling.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 KRUIKSWIJK</td>
<td>1987-06-07</td>
</tr>
<tr>
<td>18</td>
<td>Pascal EENKHOORN</td>
<td>1997-02-08</td>
</tr>
</tbody>
</table>

(3 rows)

A materialized view can be created using a filtering statement that includes a restriction on a non-primary key column.

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

This materialized view only stores 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 cycling.cyclist_by_birthday_Netherlands
WHERE birthday = '1997-02-08';

<table>
<thead>
<tr>
<th>age</th>
<th>name</th>
<th>birthday</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>Pascal EENKHOORN</td>
<td>1997-02-08</td>
</tr>
</tbody>
</table>

(1 rows)

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.

DSE 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.

**Altering a materialized view** [Alter the properties of a materialized view with the ALTER MATERIALIZED VIEW command.]

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.

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

**Creating a materialized view** [Create materialized views with the CREATE MATERIALIZED VIEW command.]

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.

```cql
DROP MATERIALIZED VIEW IF EXISTS cycling.cyclist_by_age;
```
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 DSE version for using materialized views?

DataStax recommends using the most recent version of DSE 6.x and later to get the most stable version of materialized views. Several bugs fixes and performance improvements were added for materialized views in these versions.
Chapter 4. Introduction to CQL

CQL is the primary language for managing database resources such as keyspaces, tables, functions, aggregates, user-defined types, roles, and access permissions. Use CQL to insert, update, and query tables, run DSE Search commands, plus much more. You can run CQL commands in DataStax Studio. See Creating a table [in DSE Studio] using CQL.

For production, DataStax supplies a number of drivers so that CQL statements and search commands can be passed from client to cluster and back. Other administrative tasks can be accomplished using DSE OpsCenter.

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 interface for running 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. An example directory location of the script is /usr/bin.
Introduction to CQL

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.

```bash
$ cqlsh
```

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

```bash
$ cqlsh host_name port
```

For example:

```bash
$ cqlsh 10.100.176.166 9042
```

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

```bash
$ cqlsh -u username -p password host_name port
```

For example:

```bash
$ cqlsh -u janeappdev -p j4nesp&swd 10.100.176.166 9042
```

3. Print the help menu for cqlsh.

```bash
$ 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 interface for running CQL commands interactively. The CQL shell supports tab completion.

To connect to a security-enabled cluster, see Providing Kerberos credentials when starting CQL shell.

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

1. Navigate to the DataStax Enterprise installation directory and locate the bin directory that contains cqlsh.

2. Start cqlsh on the Mac OSX, for example.

```bash
$ cqlsh
```

If you use security features, provide a user name and password.
3. Print the help menu for `cqlsh`.

   ```
   $ cqlsh --help
   ```

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

   ```
   $ 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:

```sql
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:

```sql
CREATE TABLE cycling.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

### Queries that Work

<table>
<thead>
<tr>
<th>SELECT &quot;foo&quot; FROM ...</th>
</tr>
</thead>
</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, and so forth.

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$% ...</td>
</tr>
<tr>
<td>CREATE TABLE foo_bar ...</td>
<td>CREATE TABLE foo[^0 ...</td>
</tr>
<tr>
<td>CREATE TABLE foo (&quot;what*&quot; text, ...)</td>
<td>CREATE TABLE foo (what* text, ...)</td>
</tr>
<tr>
<td>ALTER TABLE foo5 ...</td>
<td>ALTER TABLE &quot;foo5$&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#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 (dot notation) in these statements:

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

1. To specify a table when you are not located in the keyspace that contains the table, specify the name of the keyspace followed by a period, then the table name. For example, `cycling.race_winners`, where `cycling` is the keyspace and `race_winners` is the table name.

```cql
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 must be enclosed in double quotation marks in CQL.

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

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

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.

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

Valid literals

Valid literals consist of the following 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:

  ```cql
  ```

  NaN and Infinity are floats.

- **identifier**
  - Names of tables, columns, types, and other objects are identifiers. Because 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.

  ```cql
  identifier
  ```

- **integer**
  - An optional minus sign (-) followed by one or more digits.

- **string literal**
Introduction to CQL

Characters enclosed in single quotation marks. To use a single quotation mark itself in a string literal, escape it using a single quotation mark. For example, use " to make dog possessive: dog’s.

- uuid

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

- timeuuid

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

- whitespace

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

Exponential notation

DataStax Enterprise supports exponential notation. This example shows the use of exponential notation in the double and float values.

```cql
CREATE TABLE cycling.team_race_time (
    id text PRIMARY KEY,
    value_double double,
    value_float float
);

INSERT INTO cycling.team_race_time (
    id, value_float, value_double
) VALUES (
    'Rabobank-Liv Woman Cycling Team', 2.6034345E+38, 2.6034345E+38
);

SELECT * FROM cycling.team_race_time;
```

Output:

```
id                              | value_double | value_float
---------------------------------+--------------+-------------
Rabobank-Liv Woman Cycling Team  |   2.6034e+38 |  2.6034e+38
(1 rows)
```

Code comments

Use the following notation to include comments in CQL code:
• For a single line or end of a line, place a double hyphen (--) before the text to comment out the rest of the line:

```
SELECT * FROM cycling.race_winners; -- End of line comment
```

• For a single line or end of a line, place a double forward slash (//) before the text to comment out the rest of the line:

```
SELECT * FROM cycling.race_winners; // End of line comment
```

• For a block of comments, put a forward slash asterisk (/*) at the beginning of the comment and then asterisk forward slash (*/ ) at the end.

```
/* This is a
   comment that spans multiple
   lines */
SELECT * FROM cycling.race_winners;
```

### Keywords

To use a reserved keyword as an identifier for a keyspace, table, column, function, or user-defined type name, enclose it in double quotes. 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 in double quotes. For example:

```
CREATE TABLE cycling."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
• FOR
• FROM
• FULL
• GRANT
• IF
• IN
• INDEX
• INFINITY
• INSERT
• INTO
• IS
• KEY
• LANGUAGE
• MATERIALIZED
• MBEAN
• MBEANS
• MODIFY
• NAN
• NORECURSIVE
• NOT
• NULL
• OF
• ON
• OR
• ORDER
• PRIMARY
• RENAME
• REPLACE
• RESTRICT
• REVOKE
• SCHEMA
• SELECT
• SET
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
Introduction to CQL

- DISTINCT
- DOUBLE
- DURATION
- EXISTS
- FILTERING
- FINALFUNC
- FLOAT
- FROZEN
- FUNCTION
- FUNCTIONS
- GROUP
- INET
- INITCOND
- INPUT
- INT
- JSON
- KEYS
- KEYSPEC
- KEYSPECIES
- LIKE
- LIMIT
- LIST
- LOGIN
- MAP
- MONOTONIC
- NOLOGIN
- NOSUPERUSER
- OPTIONS
- PARTITION
- PASSWORD
- PER
- PERMISSION
- PERMISSIONS
- RESOURCE
- RETURNS
Introduction to CQL

- ROLE
- ROLES
- SFUNC
- SMALLINT
- STATIC
- STORAGE
- STYPE
- SUBMISSION
- SUPERUSER
- TEXT
- TIME
- TIMESTAMP
- TIMEUUID
- TINYINT
- TRIGGER
- TTL
- TUPLE
- TYPE
- USER
- USERS
- UUID
- VALUES
- VARCHAR
- VARINT
- WORKPOOL
- WRITETIME

Limitations

The upper limits are as follows:

- Cells in a partition: \(2^{31}\); single column value size: 2 GB (1 MB is recommended)
- Clustering column value length of: 65535 \(2^{16} - 1\)
- Key length: 65535 \(2^{16} - 1\)
- Table / column family name length: 222 characters
- Keyspace name length: 222 characters
- Query parameters in a query: 65535 \(2^{16} - 1\)
- Statements in a batch: 65535 \(2^{16} - 1\)
Introduction to CQL

- Fields in a tuple: 32768 ($2^{15}$) (just a few fields, such as 2 to 10, are recommended)
- Collection (list): collection limit is ~2 billion ($2^{31}$); values size is 65535 ($2^{16}$ - 1)
- Collection (set): collection limit is ~2 billion ($2^{31}$); values size is 65535 ($2^{16}$ - 1)
- Collection (map): collection limit for number of keys is 65535 ($2^{16}$ - 1); values size is 65535 ($2^{16}$ - 1)
- Blob size: 2 GB (less than 1 MB is recommended)

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

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 names</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, rows_merged</td>
<td>Information on compaction history.</td>
</tr>
<tr>
<td><em>IndexInfo</em></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, data_center, dse_version, gossip_generation, graph, host_id, jmx_port, listen_address, native_protocol_version, native_transport_address, native_transport_port, native_transport_port_ssl, partitioner, rack, release_version, rpc_address, schema_version, server_id, storage_port, storage_port_ssl, tokens, truncated_at, workload, workloads</td>
<td>Information that a node has about itself and a superset of gossip.</td>
</tr>
<tr>
<td>paxos</td>
<td>row_key, cf_id, in_progress_ballot, most_recent_commit, most_recent_commit_at, most_recent_commit_version, proposal, proposal_ballot, proposal_version</td>
<td>Information on lightweight Paxos transactions.</td>
</tr>
<tr>
<td>peers</td>
<td>peer, data_center, dse_version, graph, host_id, jmx_port, native_transport_address, native_transport_port, native_transport_port_ssl, preferred_ip, rack, release_version, rpc_address, schema_version, server_id, storage_port, storage_port_ssl, tokens, workload, workloads</td>
<td>Each node records what other nodes tell it about themselves over the gossip.</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, partitions_count</td>
<td>Information on partitions.</td>
</tr>
<tr>
<td>sstable_activity</td>
<td>keyspace_name, columnfamily_name, generation, rate_120m, rate_15m</td>
<td></td>
</tr>
</tbody>
</table>
### Table 4: Columns in system_schema tables

<table>
<thead>
<tr>
<th>Table name</th>
<th>Column names</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>views Builds In Progress</td>
<td>keyspace_name, view_name, generation_number, last_token</td>
<td></td>
</tr>
<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, default_time_to_live, extensions, flags, gc_grace_seconds, id, max_index_interval, memtable_flush_period_in_ms, min_index_interval, speculative_retry</td>
<td>Information on columns and column indexes. Used internally for compound primary keys.</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, default_time_to_live, extensions, flags, gc_grace_seconds, include_all_columns, max_index_interval, memtable_flush_period_in_ms, min_index_interval, 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 nodetool repair history.</td>
</tr>
<tr>
<td>nodesync_status</td>
<td>keyspace_name, table_name, range_group, start_token, end_token, last_successful_validation, last_unsuccessful_validation, locked_by</td>
<td>Contains NodeSync status for segments on the local system. See NodeSync Validation process and status.</td>
</tr>
<tr>
<td>nodesync_user_validations</td>
<td>id, node, keyspace_name, table_name, ended_at, metrics, outcomes, segments_to_validate, segments_validated, started_at, status, validated_ranges</td>
<td>Contains details of NodeSyncs operations that were manually started.</td>
</tr>
<tr>
<td>parent_repair_history</td>
<td>parent_id, columnfamily_names, exception_message, exception_stacktrace, finished_at, keyspace_name, options, requested_ranges, started_at, successful_ranges</td>
<td></td>
</tr>
<tr>
<td>view_build_status</td>
<td>keyspace_name, view_name, host_id, status</td>
<td></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>
<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>
<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>

Keyspace, table, and column information
Use CQL to query system_schema tables to view keyspace, table, and column information. The same information is also available using the cqlsh DESCRIBE commands, which is the preferred method.

To secure schema data, enable system_keyspaces_filtering. Users can only access schema information of objects for which they have permission to access.

- Query the defined keyspaces using the SELECT statement.

```
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>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>dse_system</td>
<td>True</td>
<td>{'class': 'org.apache.cassandra.locator.LocalStrategy'}</td>
</tr>
<tr>
<td>dse_system_local</td>
<td>True</td>
<td>{'class': 'org.apache.cassandra.locator.EverywhereStrategy'}</td>
</tr>
<tr>
<td>dse_leases</td>
<td>True</td>
<td>{'class': 'org.apache.cassandra.locator.SimpleStrategy', 'replication_factor': '1'}</td>
</tr>
</tbody>
</table>
• Get the schema information for tables in the `cycling` keyspace.

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

The following results show the record for the `cyclist_name` table formatted with the `cqlsh` `EXPAND ON` option.

<table>
<thead>
<tr>
<th>Row 1</th>
<th>-----------------------------</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-----------------------------</td>
</tr>
<tr>
<td>keyspace_name</td>
<td>cycling</td>
</tr>
<tr>
<td>table_name</td>
<td>cyclist_name</td>
</tr>
<tr>
<td>bloom_filter_fp_chance</td>
<td>0.01</td>
</tr>
<tr>
<td>caching</td>
<td>{'keys': 'ALL', 'rows_per_partition': 'NONE'}</td>
</tr>
<tr>
<td>cdc</td>
<td>null</td>
</tr>
<tr>
<td>comment</td>
<td></td>
</tr>
<tr>
<td>compaction</td>
<td>{'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy', 'max_threshold': '32', 'min_threshold': '4'}</td>
</tr>
<tr>
<td>compression</td>
<td>{'chunk_length_in_kb': '64', 'class': 'org.apache.cassandra.io.compress.LZ4Compressor'}</td>
</tr>
<tr>
<td>crc_check_chance</td>
<td>1</td>
</tr>
<tr>
<td>dcllocal_read_repair_chance</td>
<td>0.1</td>
</tr>
<tr>
<td>default_time_to_live</td>
<td>0</td>
</tr>
<tr>
<td>extensions</td>
<td>()</td>
</tr>
<tr>
<td>flags</td>
<td>{'compound'}</td>
</tr>
<tr>
<td>gc_grace_seconds</td>
<td>864000</td>
</tr>
<tr>
<td>id</td>
<td>8a1f5bf1-0258-11ea-bd8d-9f9b8a53b5f5</td>
</tr>
<tr>
<td>max_index_interval</td>
<td>2048</td>
</tr>
<tr>
<td>memtable_flush_period_in_ms</td>
<td>0</td>
</tr>
<tr>
<td>min_index_interval</td>
<td>128</td>
</tr>
<tr>
<td>nodesync</td>
<td>null</td>
</tr>
<tr>
<td>speculative_retry</td>
<td>99PERCENTILE</td>
</tr>
<tr>
<td>read_repair_chance</td>
<td>0</td>
</tr>
<tr>
<td>replication_factor</td>
<td>'1'</td>
</tr>
</tbody>
</table>
**Introduction to CQL**

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

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

<table>
<thead>
<tr>
<th>keyspace_name</th>
<th>table_name</th>
<th>column_name</th>
<th>clustering_order</th>
<th>column_name_bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>position</td>
<td>required_for_liveness</td>
<td>type</td>
<td></td>
</tr>
<tr>
<td>---------------+--------------+------------------</td>
<td>------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cycling</td>
<td>cyclist_name</td>
<td>comment</td>
<td>regular</td>
<td>-1</td>
</tr>
<tr>
<td>0x636f6d6d656e74</td>
<td>regular</td>
<td>-1</td>
<td>False</td>
<td></td>
</tr>
<tr>
<td>text</td>
<td>cycling</td>
<td>firstname</td>
<td>regular</td>
<td>-1</td>
</tr>
<tr>
<td>0x6669727374616d65</td>
<td>regular</td>
<td>-1</td>
<td>False</td>
<td></td>
</tr>
<tr>
<td>text</td>
<td>cycling</td>
<td>id</td>
<td>regular</td>
<td>-1</td>
</tr>
<tr>
<td>0x6964</td>
<td>partition_key</td>
<td>0</td>
<td>False</td>
<td></td>
</tr>
<tr>
<td>uuid</td>
<td>cycling</td>
<td>lastname</td>
<td>regular</td>
<td>-1</td>
</tr>
<tr>
<td>0x6c6173746e616d65</td>
<td>regular</td>
<td>-1</td>
<td>False</td>
<td></td>
</tr>
<tr>
<td>text</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The `system_schema` table does not show search index or row-level access control settings.

**Querying system tables for cluster topology information**

Query system tables to get cluster topology information. Display the IP address of peer nodes, datacenter and rack names, ports, token values, versions, and other information.

The `ring` or `list` view in OpsCenter Monitoring displays IP address and workload, in addition to node health.

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

```sql
SELECT peer, data_center, dse_version, graph
FROM system.peers;
```

<table>
<thead>
<tr>
<th>peer</th>
<th>data_center</th>
<th>dse_version</th>
<th>graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.10.100.110</td>
<td>Cassandra</td>
<td>6.7.0</td>
<td>False</td>
</tr>
</tbody>
</table>

The following output will vary depending on the version used.

**Functions, aggregates, and user types**

Query `system_schema.functions`, `aggregates`, and `types` tables to get information about user-defined functions (UDFs), user-defined aggregates (UDAs), and user-defined types (UDTs).

Use OpsCenter to view the CQL for defined UDF, UDA, and UDT in a keyspace.
• Show one user-defined function in the `system_schema.functions` table.

```sql
SELECT *
FROM system_schema.functions
LIMIT 1;
```

The following output shows the first record using the `cqlsh` `EXPAND ON` option.

```
<table>
<thead>
<tr>
<th>Row 1</th>
</tr>
</thead>
</table>
| +--------------------------------------------------------+
| Row 1 | cycling
| keyspace_name | avgfinal
| function_name | avgfinal
| argument_types | ['frozen<tuple<int, bigint>>']
| argument_names | ['state']
| body | double r = 0; if (state.getInt(0) == 0) return null; r = state.getLong(1); r /= state.getInt(0);
| return null | Double.valueOf(r);
| called_on_null_input | True
| deterministic | False
| language | java
| monotonic | False
| monotonic_on | []
| return_type | double
```

• List the schema settings for one user-defined aggregate.

```sql
SELECT *
FROM system_schema.aggregates
LIMIT 1;
```

```
<table>
<thead>
<tr>
<th>Row 1</th>
</tr>
</thead>
</table>
| +--------------------------------------------------------+
| Row 1 | average
| keyspace_name | cycling
| aggregate_name | average
| argument_types | ['int']
| deterministic | False
| final_func | avgfinal
| initcond | (0, 0)
| return_type | double
| state_func | avgstate
| state_type | frozen<tuple<int, bigint>>
```

• Show the schema settings for one user-defined type.

```sql
SELECT *
FROM system_schema.types
LIMIT 1;
```

```
<table>
<thead>
<tr>
<th>Row 1</th>
</tr>
</thead>
</table>
| +--------------------------------------------------------+
| Row 1 | basic_info
| keyspace_name | cycling
| type_name | basic_info
| field_names | ['birthday', 'nationality', 'height', 'weight']
| field_types | ['timestamp', 'text', 'text', 'text']
```
About virtual keyspaces and tables

CASSANDRA-7622 introduced support for virtual keyspaces and virtual tables. Virtual tables are exposed by the server to the users as local tables. They are read-only, but can sometime can support updates. They do not support secondary indexes or materialized views. Virtual tables are always part of a virtual keyspaces. The two available virtual keyspaces are system_virtual_schema and system_views Virtual tables are not associated with physical files on disk, and have no memtables or SSTables. They are designed to expose server data such as configuration or metrics in memory.

**system_virtual_schema keyspace**

The system_virtual_schema contains the information about the existing virtual keyspaces and tables. It contains three virtual tables which expose the definitions of the named data object:

**Table 8: keyspaces**

<table>
<thead>
<tr>
<th>Column name</th>
<th>CQL type</th>
<th>Column type</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyspace_name</td>
<td>text</td>
<td>partition key</td>
</tr>
</tbody>
</table>

**Table 9: tables**

<table>
<thead>
<tr>
<th>Column name</th>
<th>CQL type</th>
<th>Column type</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyspace_name</td>
<td>text</td>
<td>partition key</td>
</tr>
<tr>
<td>table_name</td>
<td>text</td>
<td>clustering key</td>
</tr>
<tr>
<td>comment</td>
<td>text</td>
<td>regular</td>
</tr>
</tbody>
</table>

**Table 10: columns**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>CQL Type</th>
<th>Column type</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyspace_name</td>
<td>text</td>
<td>partition key</td>
</tr>
<tr>
<td>table_name</td>
<td>text</td>
<td>clustering key</td>
</tr>
<tr>
<td>column_name</td>
<td>text</td>
<td>clustering key</td>
</tr>
<tr>
<td>clustering_order</td>
<td>text</td>
<td>regular</td>
</tr>
<tr>
<td>column_name_bytes</td>
<td>blob</td>
<td>regular</td>
</tr>
<tr>
<td>position</td>
<td>int</td>
<td>regular</td>
</tr>
<tr>
<td>type</td>
<td>text</td>
<td>regular</td>
</tr>
</tbody>
</table>

**system_views keyspace**

System views expose information about the server internals. The sstable_tasks system view exposes the SSTable tasks that are currently running on the host.

**Table 11: sstable_tasks**

<table>
<thead>
<tr>
<th>Column name</th>
<th>CQL type</th>
<th>Column type</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyspace_name</td>
<td>text</td>
<td>partition key</td>
</tr>
<tr>
<td>table_name</td>
<td>text</td>
<td>clustering key</td>
</tr>
<tr>
<td>task_id</td>
<td>uuid</td>
<td>clustering key</td>
</tr>
</tbody>
</table>
The remaining amount of work can be computer using `total - progress AS remaining` in a `SELECT` clause.

### Virtual tables and `cqlsh`

Virtual tables are exposed as local tables, so `cqlsh` will display data for the currently connected host. To displace schema for virtual tables, use `DESCRIBE FULL SCHEMA` to return schema for both the real keyspaces and the virtual ones.

Users are not allowed to create virtual keyspaces or virtual tables.

To see the schema for the virtual table `sstable_tasks`, use:

```cql
DESCRIBE TABLE system_views.sstable_tasks;
```

/\* Warning: Table system_views.sstable_tasks is a virtual table and cannot be recreated with CQL. Structure, for reference:
VIRTUAL TABLE system_views.sstable_tasks (  
  keyspace_name text,  
  table_name text,  
  task_id uuid,  
  kind text,  
  progress bigint,  
  total bigint,  
  unit text,  
  PRIMARY KEY (keyspace_name, table_name, task_id)  
) WITH CLUSTERING ORDER BY (table_name ASC, task_id ASC)  
AND comment = 'current sstable tasks';\*/

To discover how much time is remaining for SSTable tasks, use the following query:

```cql
SELECT total - progress AS remaining  
FROM system_views.sstable_tasks;
```
Chapter 5. Managing database resources

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.

Managing keyspaces

Create and drop keyspaces, and 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. Because 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 intended for evaluating DataStax Enterprise. 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 Configuring the Amazon EC2 single-region snitch, create the keyspace using EC2 datacenter and rack names. If the cluster uses the Configuring the Google Cloud Platform snitch, create the keyspace using GoogleCloud datacenter and rack names.

- **EveryWhereStrategy** is a specialized DataStax Enterprise strategy used by the dse_system keyspace and is not intended for customer use.

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 datacenter names.
Errors related to the SimpleSnitch

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

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

Creating a keyspace

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

- SimpleStrategy: Applies the same replication setting across the cluster. Only use this setting on single node test or development environments.
- NetworkTopography: Applies the replication setting per datacenter. Use in production environments.
- EverywhereStrategy: The default replication strategy for the dse_system and solr_admin keyspaces. Do not use or alter any instances of EverywhereStrategy.

1. Verify the datacenter name using nodetool status.

   $ nodetool status

   Example output:

   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-c75669d83a0 rack1

2. Verify the datacenter names, which are case-sensitive. Ensure that the datacenter name case matches exactly in the next step.

3. Start cqlsh and create a keyspace.

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

4. Switch to the keyspace.

   USE cycling;

CREATES 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.

Changing the replication factor of a keyspace impacts each node that the keyspace replicates to (or no longer replicates to). Follow this procedure to prepare all affected nodes for this change.
Managing database resources

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 REPLICACTION = {'class': 'NetworkTopologyStrategy', 'dc_name_1': 3,
                     'dc_name_2': 2};
```

Or if using SimpleStrategy:

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

Datacenter names are case-sensitive. Verify the case of the datacenter name using a utility, such as dsetool status.

See Changing keyspace replication strategy.

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

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

```
ALTER KEYSPACE [Changes keyspace replication strategy and enables or disables commit log.]
```

Dropping a keyspace

Use the DROP KEYSPACE 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 the 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.

Use the OpsCenter Backup Service to back up a keyspace.

• Drop the keyspace.

```
DROP KEYSPACE keyspace_name;
```

Managing types, functions, and aggregates

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

Creating a user-defined function (UDF)

Write custom functions using Java or JavaScript to use in SELECT, INSERT, and UPDATE statements or as a building block for a user-defined aggregate (UDA). Functions are only available within the keyspace where it is defined. To preserve fast performance, use user-defined functions for short computations, and create Java UDFs instead of Javascript UDFs.

Use OpsCenter to view the CQL for a defined UDF in a keyspace.

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.

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 (in addition to Java)**: 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.

```cql
CREATE OR REPLACE FUNCTION cycling.fLog(
    input 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 always executes when called.
- **RETURNS NULL ON NULL INPUT** ensures the function always returns `NULL` if any of the input arguments are `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 same signature does not exist in the keyspace. OR REPLACE and IF NOT EXISTS cannot be used in the same command.**

Creating a user-defined aggregate function (UDA)

DataStax Enterprise 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.

Use OpsCenter to view the CQL for a defined **UDA** in a keyspace.

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.

```cql
CREATE OR REPLACE FUNCTION cycling.avgState(
    state tuple<int, bigint>,
    val int
)
CALLED ON NULL INPUT
RETURNS tuple<int, bigint>
LANGUAGE java AS
```
Managing database resources

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

• Create the aggregate function using the state and final functions, and add an STYPE to define the data type for the function. Different STYPES distinguishes between functions 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.

```sql
CREATE OR REPLACE AGGREGATE cycling.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 (UDT)

Use the ALTER TYPE command to add fields to a user-defined type (UDT) or to rename an existing field in a UDT.

Modifying UDTs used in primary keys or index columns is not supported. Changing the field type is not supported.

Use OpsCenter to view the CQL for a defined UDT in a keyspace.

• Add a middlename field of type text to the user-defined type cycling.fullname.

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

The ALTER TYPE command adds the field to the type.
Managing database resources

To verify the changes, use the `DESC TYPE` command.

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

The `middlename` column appears in the type definition.

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

- To change the name of an existing field, use `RENAME`.

```
ALTER TYPE cycling.fullname
RENAME middlename TO middle
    AND lastname TO last
    AND firstname TO first;
```

Verify the changes.

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

The renamed fields appear in the type definition.

```
CREATE TYPE cycling.fullname (
    first text,
    last text,
    middle text
);
```

Dropping a user-defined function (UDF)

Use the `DROP FUNCTION` command to drop a user-defined function (UDF).

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

```
DROP FUNCTION IF EXISTS cycling.fLog;
```

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, and so forth). See `table_options`.

Create schema using `cqlsh`

Create table schema using `cqlsh`. DataStax Enterprise does not support dynamic schema generation because collisions can occur if multiple clients attempt to generate tables simultaneously. To recover from collisions, follow the instructions in `schema collision fix`.

CQL for DataStax Enterprise 6.7 Latest DSE version Latest 6.7 patch: 6.7.7
Managing database resources

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.

The DataStax Enterprise 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** that 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. When selecting the table’s primary key, consider:

- the size of the partitions.
- the order of the data within partitions.
- the distribution of the partitions among the nodes of the cluster.

Table name conventions
The name of a table can be a string of alphanumeric characters and underscores, but it must begin with a letter. Consider the keyspace when naming a table:

- 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 specify the new table name.

Column characteristics
CQL supports several column types. When creating a table, assign a **data type** to each column. 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**:

```cql
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:

```cql
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 is not a recommended best practice to use that 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:

```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));
```

For more information, see **Tuple type**.
Managing database resources

Create a user-defined type (UDTs) as a data type of several fields, using `CREATE TYPE`. It is best to create a UDT for use with multiple table definitions. The user-defined column type (UDT) requires the `frozen` keyword. 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 Enterprise 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 inserts and retrieves data faster if many values for the column can distribute the partitions across many nodes.

Often, your first venture into using DSE involves tables with simple primary keys. Keep in mind that only the primary key can be specified when retrieving data from the table (unless using secondary indexes). If an application needs a simple lookup table using a single unique identifier, then a simple primary key is the best 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 multiple ways, as shown in the following examples.

- 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.

  ```cql
  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 of the primary key is enclosed in parentheses, which is `id` in the example below.

  ```cql
  USE cycling;
  CREATE TABLE cyclist_name (id UUID, lastname text, firstname text, PRIMARY KEY (id))
  ```
Managing database resources

- 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_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 Enterprise 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 resides. 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. DSE 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.

![Composite partition key example](image)

The database stores an entire row of data on a node by partition key. If there is too much data in a partition and data needs to be spread among multiple nodes, use a composite partition key.

Using a composite partition key

Use a composite partition key in a primary key to create a set of columns that can distribute data across multiple partitions, which can be queried to return sorted results.

The following 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 a few different ways as shown in the following examples.

- 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 IF NOT EXISTS 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 IF NOT EXISTS 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 Enterprise uses a partition key that is either simple or composite. In addition, clustering columns 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 benefits reads and writes better than this simplistic choice.

Remember that data is distributed throughout a 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 a clustering column or columns is analogous to JOINs in a relational database, but clustering columns 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>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>One-day-races</td>
<td>367</td>
<td>220844bf-4860-4965-9a4b-6b5d3a79c8fb</td>
<td>TIRALONGO</td>
</tr>
<tr>
<td>One-day-races</td>
<td>198</td>
<td>6b090bec-6e8e-48d9-a5f8-97e6fb4ec38b</td>
<td>KRUIJSWIJK</td>
</tr>
<tr>
<td>Time-trial</td>
<td>182</td>
<td>220844bf-4860-4965-9a4b-6b5d3a79c8fb</td>
<td>TIRALONGO</td>
</tr>
<tr>
<td>Time-trial</td>
<td>3</td>
<td>6b090bec-6e8e-48d9-a5f8-97e6fb4ec38b</td>
<td>KRUIJSWIJK</td>
</tr>
<tr>
<td>Sprint</td>
<td>39</td>
<td>6b090bec-6e8e-48d9-a5f8-97e6fb4ec38b</td>
<td>KRUIJSWIJK</td>
</tr>
<tr>
<td>Sprint</td>
<td>0</td>
<td>220844bf-4860-4965-9a4b-6b5d3a79c8fb</td>
<td>TIRALONGO</td>
</tr>
<tr>
<td>GC</td>
<td>1324</td>
<td>6b090bec-6e8e-48d9-a5f8-97e6fb4ec38b</td>
<td>KRUIJSWIJK</td>
</tr>
<tr>
<td>GC</td>
<td>1269</td>
<td>220844bf-4860-4965-9a4b-6b5d3a79c8fb</td>
<td>TIRALONGO</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 your environment has 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 can be queried to return sorted results. If the preceding example was designed for a relational database, a cyclists table would be created with a foreign key to the races. In DataStax Enterprise, you denormalize the data because joins are not performant in a distributed system. Later, other schemas 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 `lastname`, `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.

• To create a table that has a compound primary key, specify two or more columns as the primary key. This example defines `category` and `points` as the columns of the compound primary key. To sort the points in
Managing database resources

descending order, the example uses the additional WITH CLUSTERING ORDER BY clause. Ascending order is more efficient to store, but descending queries are faster due to the nature of the storage engine.

```
USE cycling;
CREATE TABLE IF NOT EXISTS 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.

```
CREATE TABLE IF NOT EXISTS 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, the points column is defined as a clustering column sorted in descending order.

Using advanced data types for columns

Creating collections

DataStax Enterprise 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. DSE 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, because 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; DataStax Enterprise reads a collection in its entirety, which impacts 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.

  The following example shows the table and the initial rows.

  ```
  CREATE TABLE IF NOT EXISTS cycling.cyclist_career_teams (
    id UUID PRIMARY KEY,
    lastname text,
    teams set<text>
  );
  ```

<table>
<thead>
<tr>
<th>id</th>
<th>lastname</th>
<th>teams</th>
</tr>
</thead>
<tbody>
<tr>
<td>cb07baad-aec8-4f65-b28a-bddc06a0de23</td>
<td>ARMITSTEAD</td>
<td>{'AA Drink - Leontien.nl', 'Boels-Dolmans Cycling Team', 'Team Garmin - Cervelo'}</td>
</tr>
<tr>
<td>5b6962dd-3f90-4c93-8f61-eabfa4a803e2</td>
<td>VOS</td>
<td>{'Nederland bloeit', 'Rabobank Women Team', 'Rabobank-Liv Giant', 'Rabobank-Liv Woman Cycling Team'}</td>
</tr>
<tr>
<td>1c9ebc13-1eab-4ad5-be87-dce433216d40</td>
<td>BRAND</td>
<td>{'AA Drink - Leontien.nl', 'Leontien.nl', 'Rabobank-Liv Giant', 'Rabobank-Liv Woman Cycling Team'}</td>
</tr>
<tr>
<td>e7cd5752-bc0d-4157-a80f-7523add8bcd</td>
<td>VAN DER BREGGEN</td>
<td>{'Rabobank-Liv Woman Cycling Team', 'Sengers Ladies Cycling Team', 'Team Flexpoint'}</td>
</tr>
</tbody>
</table>

| (4 rows) |

**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.
Managing database resources

Each month and 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. The following example shows the table and the initial rows.

```sql
CREATE TABLE IF NOT EXISTS cycling.upcoming_calendar (
  year int,
  month int,
  events list<text>,
  PRIMARY KEY (year, month)
);
```

<table>
<thead>
<tr>
<th>year</th>
<th>month</th>
<th>events</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015</td>
<td>6</td>
<td>['Criterium du Dauphine', 'Tour de Suisse']</td>
</tr>
<tr>
<td>2015</td>
<td>7</td>
<td>['Tour de France']</td>
</tr>
</tbody>
</table>

**(2 rows)**

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.

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. The following example shows the table and the initial row.

```sql
CREATE TABLE IF NOT EXISTS cycling.cyclist_teams (
  id uuid PRIMARY KEY,
  firstname text,
  lastname text,
  teams map<int, text>
);
```

<table>
<thead>
<tr>
<th>id</th>
<th>firstname</th>
<th>lastname</th>
<th>teams</th>
</tr>
</thead>
<tbody>
<tr>
<td>+---------------------------------------------------------------------------+----------+----------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5b6962dd-3f90-4c93-8f61-eabfa4a803e2</td>
<td>Marianne</td>
<td>VOS</td>
<td>{2014: 'Rabobank-Liv Woman Cycling Team', 2015: 'Rabobank-Liv Woman Cycling Team'}</td>
</tr>
</tbody>
</table>

(2 rows)
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.

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

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

```
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.

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

Examples:

```
CREATE TABLE IF NOT EXISTS cycling.race_starts (  
cyclist_name text PRIMARY KEY,  
rnumbers FROZEN<LIST<int>>  
);  
```

```
CREATE TABLE IF NOT EXISTS cycling.race_winners (  
cyclist_name FROZEN<fullname>,  
race_name text,  
race_position int,  
PRIMARY KEY (race_name, race_position)  
);  
```

```
CREATE TABLE IF NOT EXISTS cycling.cyclist_races (  
id UUID PRIMARY KEY,  
lastname text,  
firstname text,  
races list<FROZEN<race>>  
);  
```

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.

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.
Managing database resources

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

```
CREATE TABLE IF NOT EXISTS cycling.nation_rank {
    nation text PRIMARY KEY,
    info tuple<int, text, int>
};
```

- The previous table `cycling.nation_rank` uses the nation as the primary key. It is possible to store the same data using the rank as the primary key, as shown in the following example. The example creates a table named `cycling.popular` using a tuple to store the country name, cyclist name, and points total for a cyclist, with rank set as the primary key.

```
CREATE TABLE IF NOT EXISTS cycling.popular {
    rank int PRIMARY KEY,
    cinfo tuple<text, text, int>
};
```

- Create a table `cycling.route` using a tuple to store each way point location name, latitude, and longitude. Two tuples are used in the following example, with one tuple nested inside the other tuple.

```
CREATE TABLE IF NOT EXISTS cycling.route {
    race_id int,
    race_name text,
    point_id int,
    lat_long tuple<text, tuple<float, float>>,
    PRIMARY KEY (race_id, point_id)
};
```

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. After a UDT is created, it can be used to define a table column.

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

```
CREATE TYPE IF NOT EXISTS cycling.basic_info {
    birthday timestamp,
    nationality text,
    height text,
    weight 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.

```
The frozen keyword is not required for UDTs that contain only non-collection fields.

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.

CREATE TABLE IF NOT EXISTS cycling.cyclist_stats {
    id UUID PRIMARY KEY,
    lastname text,
    basics basic_info
};
```

- A user-defined type can be nested in another column type. The example below nests a UDT in a frozen list named `races`.

```
CREATE TYPE IF NOT EXISTS cycling.race {
    race_title text,
    races list<
        tuple<
            race_title text,
            basics basic_info
        >
    }
};
```
Creating a blob column

A blob (Binary Large OBject) 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 a native type into binary data (blob), and convert a blob back into a native type:

- `typeAsBlob(value)`
- `blobAsType(value)`

For every native non-blob data type supported by CQL, the `typeAsBlob` function takes an 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.

The following example shows the use of a blob column and the `bigintAsBlob` function:

```sql
CREATE TABLE IF NOT EXISTS cycling.lastname_bio (
   lastname varchar PRIMARY KEY,
   bio blob
);

INSERT INTO cycling.lastname_bio (
   lastname, bio
) VALUES
   ('TSATEVICH', bigintAsBlob(3));

SELECT *
FROM cycling.lastname_bio;
```

Output:

<table>
<thead>
<tr>
<th>lastname</th>
<th>bio</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSATEVICH</td>
<td>0x0000000000000003</td>
</tr>
</tbody>
</table>

(1 row)

The following example shows the use of `blobAsBigInt` function:

```sql
ALTER TABLE cycling.lastname_bio
```
Managing database resources

ADD id bigint;

INSERT INTO cycling.lastname_bio (lastname, id) VALUES ('DUVAL', blobAsBigint(0x0000000000000003));

SELECT * FROM cycling.lastname_bio;

Output:

lastname | bio                | id
-----------+--------------------+------
DUVAL |               null |    3
TSATEVICH | 0x0000000000000003 | null
(2 rows)

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. The timeuuid types 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 could 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 the tuple values in parentheses to insert the values into a table, as shown in the following example:

CREATE TABLE IF NOT EXISTS cycling.nation_rank (nation text PRIMARY KEY, info tuple<int, text, int>);

INSERT INTO cycling.nation_rank (nation, info) VALUES ('Spain', (1, 'Alejandro VALVERDE', 9054));
INSERT INTO cycling.nation_rank (nation, info) VALUES ('France', (2, 'Sylvain CHAVANEL', 6339));
INSERT INTO cycling.nation_rank (nation, info) VALUES ('Belgium', (3, 'Phillippe GILBERT', 6222));

INSERT INTO cycling.nation_rank (nation, info) VALUES ('Italy', (4, 'Davide REBELLINI', 6090));

SELECT * FROM cycling.nation_rank;
SELECT * FROM cycling.nation_rank WHERE info = (3, 'Phillippe GILBERT', 6222);

Output:

<table>
<thead>
<tr>
<th>nation</th>
<th>info</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spain</td>
<td>(1, 'Alejandro VALVERDE', 9054)</td>
</tr>
<tr>
<td>Belgium</td>
<td>(3, 'Phillippe GILBERT', 6222)</td>
</tr>
<tr>
<td>France</td>
<td>(2, 'Sylvain CHAVANEL', 6339)</td>
</tr>
<tr>
<td>Italy</td>
<td>(4, 'Davide REBELLINI', 6090)</td>
</tr>
</tbody>
</table>

(4 rows)

Use a tuple to filter a selection, as shown in the following example that creates an index and then queries the table:

CREATE INDEX IF NOT EXISTS ON cycling.nation_rank (info);

SELECT * FROM cycling.nation_rank WHERE info = (3, 'Phillippe GILBERT', 6222);

Output:

<table>
<thead>
<tr>
<th>nation</th>
<th>info</th>
</tr>
</thead>
<tbody>
<tr>
<td>Belgium</td>
<td>(3, 'Phillippe GILBERT', 6222)</td>
</tr>
</tbody>
</table>

(1 rows)

You can nest tuples as shown in the following example that stores each way point location name, along with the latitude and longitude:

CREATE TABLE IF NOT EXISTS cycling.route (race_id int, race_name text, point_id int, lat_long tuple<text, tuple<float, float>>, PRIMARY KEY (race_id, point_id)
Managing database resources

INSERT INTO cycling.route (race_id, race_name, point_id, lat_long) VALUES (500, '47th Tour du Pays de Vaud', 1, ('Onnens', (46.8444, 6.6667)));

INSERT INTO cycling.route (race_id, race_name, point_id, lat_long) VALUES (500, '47th Tour du Pays de Vaud', 2, ('Champagne', (46.8333, 6.65)));

INSERT INTO cycling.route (race_id, race_name, point_id, lat_long) VALUES (500, '47th Tour du Pays de Vaud', 3, ('Novalle', (46.8333, 6.6)));

INSERT INTO cycling.route (race_id, race_name, point_id, lat_long) VALUES (500, '47th Tour du Pays de Vaud', 4, ('Vuiteboeuf', (46.8, 6.55)));

INSERT INTO cycling.route (race_id, race_name, point_id, lat_long) VALUES (500, '47th Tour du Pays de Vaud', 5, ('Baulmes', (46.7833, 6.5333)));

INSERT INTO cycling.route (race_id, race_name, point_id, lat_long) VALUES (500, '47th Tour du Pays de Vaud', 6, ('Les Clées', (46.7222, 6.5222)));

SELECT * FROM cycling.route;

Output:

<table>
<thead>
<tr>
<th>race_id</th>
<th>point_id</th>
<th>lat_long</th>
<th>race_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>1</td>
<td>('Onnens', (46.8444, 6.6667))</td>
<td>47th Tour du Pays de Vaud</td>
</tr>
<tr>
<td>500</td>
<td>2</td>
<td>('Champagne', (46.8333, 6.65))</td>
<td>47th Tour du Pays de Vaud</td>
</tr>
<tr>
<td>500</td>
<td>3</td>
<td>('Novalle', (46.8333, 6.6))</td>
<td>47th Tour du Pays de Vaud</td>
</tr>
<tr>
<td>500</td>
<td>4</td>
<td>('Vuiteboeuf', (46.8, 6.55))</td>
<td>47th Tour du Pays de Vaud</td>
</tr>
<tr>
<td>500</td>
<td>5</td>
<td>('Baulmes', (46.7833, 6.5333))</td>
<td>47th Tour du Pays de Vaud</td>
</tr>
<tr>
<td>500</td>
<td>6</td>
<td>('Les Clées', (46.7222, 6.5222))</td>
<td>47th Tour du Pays de Vaud</td>
</tr>
</tbody>
</table>

(6 rows)

User-defined type

A user-defined type facilitates handling multiple fields of related information in a table. Applications that required multiple tables can be simplified to use fewer tables by using a user-defined type to represent the related fields of information instead of storing the information in a separate table. The address type example demonstrates how to use a user-defined type.
You can create, alter, and drop a user-defined type using these commands:

- **CREATE TYPE**
- **ALTER TYPE**
- **DROP TYPE**

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

- **DESCRIBE TYPE**
- **DESCRIBE TYPES**

**Accessing a UDT in a keyspace**

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 (static column)**

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

In the following example, the `flag` column is static:

```sql
CREATE TABLE IF NOT EXISTS cycling.country_flag (  
country text,  
cyclist_name text,  
flag int STATIC,  
PRIMARY KEY (country, cyclist_name)  
);

INSERT INTO cycling.country_flag (country, cyclist_name, flag) VALUES ('Belgium', 'Jacques', 1);

INSERT INTO cycling.country_flag (country, cyclist_name) VALUES ('Belgium', 'Andre');

INSERT INTO cycling.country_flag (country, cyclist_name, flag) VALUES ('France', 'Andre', 2);

INSERT INTO cycling.country_flag (country, cyclist_name, flag) VALUES ('France', 'George', 3);

SELECT *;
```
Managing database resources

```
FROM cycling.country_flag;
```

Output:

| country | cyclist_name | flag |
|---------|--------------+------|
| Belgium | Andre        | 1    |
| Belgium | Jacques      | 1    |
| France  | Andre        | 3    |
| France  | George       | 3    |

(4 rows)

- A table that does not define any clustering columns cannot have a static column. The table that does not have clustering columns has a one-row partition in which every column is inherently static.
- A column designated to be the partition key cannot be static.

You can **batch conditional updates to a static column**.

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:

- Keeping track of the number of web page views received on a company website
- Keeping 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 or 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-a5f8-97e6f4b4c9b47</td>
<td>62</td>
</tr>
</tbody>
</table>
```

Tracking count in a distributed database presents an interesting challenge. At any given moment in DataStax Enterprise, the counter value could be stored in the memtable, commit log, 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 Enterprise rejects USING TIMESTAMP or USING TTL in the command to update a counter column.

• Create a table for the counter column.

```plaintext
CREATE TABLE IF NOT EXISTS cycling.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.

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

    UPDATE cycling.popular_count
    SET popularity = popularity + 125
    WHERE id = 6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47;

    UPDATE cycling.popular_count
    SET popularity = popularity - 64
    WHERE id = 6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47;

APPLY BATCH;

    UPDATE cycling.popular_count
    SET popularity = popularity + 2
    WHERE id = 6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47;

```

• The `popularity` column has a value of 64.

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

Output:

<table>
<thead>
<tr>
<th>id</th>
<th>popularity</th>
</tr>
</thead>
<tbody>
<tr>
<td>6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47</td>
<td>64</td>
</tr>
</tbody>
</table>

(1 rows)

• Additional increments or decrements changes 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.

Use counter types as described in Using a counter. 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.
Managing database resources

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.

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.

   Perform a rolling restart using OpsCenter Monitoring.

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` is the newest table ID in `system_schema.tables` using CQL.

   ```cql
   SELECT id, keyspace_name, table_name
   FROM system_schema.tables
   WHERE keyspace_name = 'cycling'
   AND table_name = 'cyclist_name';
   ```

   **Output:**

<table>
<thead>
<tr>
<th>id</th>
<th>keyspace_name</th>
<th>table_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>c2784ad1-065d-11ea-bd8d-9f9b8a53b5f5</td>
<td>cycling</td>
<td>cyclist_name</td>
</tr>
</tbody>
</table>

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.

Modify table columns
Use the `ALTER TABLE` command to add new columns, drop non-primary key columns, or rename a primary key column.

To change the table settings, see Altering table properties.

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

  ```cql
  ALTER TABLE cycling.cyclist_alt_stats
  ADD cyclist_age int;
  ```

  The `ALTER TABLE` command 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.

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

The result set shows the first three rows.

<table>
<thead>
<tr>
<th>id</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>e0953617-07eb-4c82-8f91-3b2757981625</td>
<td>null</td>
</tr>
<tr>
<td>a9e96714-2dd0-41f9-8bd0-557196a44ecf</td>
<td>null</td>
</tr>
<tr>
<td>ed584e99-80f7-4b13-9a90-9dc5571e6821</td>
<td>null</td>
</tr>
</tbody>
</table>

(3 rows)

- Drop the `cyclist_age` column from the table.

```
ALTER TABLE cycling.cyclist_alt_stats
DROP cyclist_age;
```

- Rename the `id` primary key column to `cyclist_id`.

```
ALTER TABLE cycling.cyclist_alt_stats
RENAME id TO cyclist_id;
```

Only primary key columns can be renamed.

### Altering a table to add a collection

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

- Alter the table `cycling.upcoming_calendar` to add a collection map named `description` to store a name and description for each race.

```
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 = {
  '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

{'Criterium du Dauphine': 'Easy race', 'Tour du Suisse': 'Hard uphill race'}
```
Managing database resources

Altering table properties

Use `ALTER TABLE` to add or change table properties.

1. Alter a table to change the caching properties. See Enabling and configuring caching for more details.

```
ALTER TABLE cycling.comments
WITH caching = {
  'keys' : 'NONE',
  'rows_per_partition' : 10
};
```

Dropping a keyspace or table

Use the DROP KEYSPACE or DROP TABLE command to drop a keyspace or table.

- Drop the `cycling` keyspace.

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

- Drop the `cycling.comments` table.

```
DROP TABLE IF EXISTS cycling.comments;
```

Indexing tables

After an index is created, data can be queried using the index.

Indexing

An index provides a means to access data in DataStax Enterprise using attributes other than the partition key. The benefit of an index is fast, efficient lookup of data that matches a given condition. The index indexes column values in a hidden table separate from the table 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 based on stale values in an 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 on average is required 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 incurs 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 recording 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. When looking for a row in a large partition, narrow the search to avoid query performance degradation.

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 cannot normally be queried. DataStax Enterprise supports indexing a collection column and indexing a static column.

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.

DSE Search can provide a more robust solution for indexing data. Learn about managing search indexes and how to configure DSE Search. Do not add a secondary index and a search index to the same table.

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

```
CREATE TABLE IF NOT EXISTS 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 in a query because those columns comprise the partition key.

```
SELECT *  
FROM cycling.rank_by_year_and_name  
WHERE race_year = 2014  
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>2014</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>1</td>
<td>Daniel MARTIN</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 with `race_year` and `race_name`, the following query fails because only the `race_year` column is specified in the `WHERE` clause.

```cql
SELECT *
FROM cycling.rank_by_year_and_name
WHERE race_year = 2014;
```

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"

In the following example, an index is created for the race year and the query succeeds. An index name is optional and must be unique within a keyspace. If you do not provide an index name, a name like `race_year_idx` is automatically assigned. Typically, you should explicitly set the index name.

```cql
CREATE INDEX IF NOT EXISTS race_year_idx ON
cycling.rank_by_year_and_name (race_year);
```

```cql
SELECT *
FROM cycling.rank_by_year_and_name
WHERE race_year = 2014;
```

<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>
</tbody>
</table>

(6 rows)

An index can also be added for a clustering column. In the following example, an index is created on the `rank` column, and `rank` is specified in the query’s `WHERE` clause.

```cql
CREATE INDEX IF NOT EXISTS rank_idx
ON cycling.rank_by_year_and_name (rank);
```

```cql
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>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>
</tbody>
</table>

(6 rows)
Managing database resources

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>2014</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>1</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>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>1</td>
<td>Benjamin PRADES</td>
</tr>
</tbody>
</table>

(4 rows)

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 production environment, certain columns might not be good choices, depending on their cardinality.

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

```sql
CREATE TABLE IF NOT EXISTS 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
);
```

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

```sql
CREATE INDEX IF NOT EXISTS birthday_idx
ON cycling.cyclist_alt_stats (birthday);
```

```sql
CREATE INDEX IF NOT EXISTS nationality_idx
ON cycling.cyclist_alt_stats (nationality);
```

- The following query attempts to retrieve the cyclists with a specified `birthday` and `nationality`. The query returns an error.

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

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"

- The indexes have been created on appropriate low-cardinality columns, but the previous query still fails. Why? The answer lies with the partition key, which has not been defined. When you attempt a potentially
Managing database resources

expensive query, such as searching a range of rows, the database requires the \texttt{ALLOW FILTERING} directive. The error is not due to multiple indexes, but the lack of a partition key definition in the query.

\begin{verbatim}
SELECT * 
FROM cycling.cyclist_alt_stats 
WHERE birthday = '1982-01-29' 
    AND nationality = 'Russia'
ALLOW FILTERING;
\end{verbatim}

\begin{verbatim}
id                                   | birthday   | first_race | height |
--------------------------------------|------------|------------|--------|
+------------+----------+-------------+---------+--------|
e0953617-07eb-4c82-8f91-3b2757981625 | 1982-01-29 | 1998-02-15 | 1.78 | 2017-04-16 |    BRUTT |      Russia |      kg |     68
\end{verbatim}

(1 rows)

Indexing a collection

Collections can be indexed and queried to find a collection containing a particular value. \texttt{Sets} and \texttt{lists} are indexed a bit differently from \texttt{maps}, given the key-value nature of \texttt{maps}.

\texttt{Sets} and \texttt{lists} can index all values found by indexing the collection column. \texttt{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 \texttt{FULL} to index the full content of a frozen collection.

All the \texttt{cautions} about using secondary indexes apply to indexing collections.

• For \texttt{set} and \texttt{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.

\begin{verbatim}
CREATE INDEX IF NOT EXISTS teams_idx 
ON cycling.cyclist_career_teams (teams);
\end{verbatim}

\begin{verbatim}
SELECT * 
FROM cycling.cyclist_career_teams 
WHERE teams CONTAINS 'Rabobank-Liv Giant';
\end{verbatim}

\begin{verbatim}
id | lastname | teams
---------------------------------------
+-------------+---------+
1c9ebc13-1eab-4ad5-be87-3ce433216d40 | BRAND   |
   | {'AA Drink - Leontien.nl', 'Leontien.nl', 'Rabobank-Liv Giant', 'Rabobank-Li
v Woman Cycling Team'}
\end{verbatim}

(1 rows)

• For \texttt{map} collections, create an index on the map key, map value, or map entry. Create an index on a \texttt{map key} to find all cyclist/team combinations for a particular year.

\begin{verbatim}
CREATE INDEX IF NOT EXISTS team_year_idx 
ON cycling.cyclist_teams ( KEYS (teams) );
\end{verbatim}

\begin{verbatim}
SELECT * 
FROM cycling.cyclist_teams
\end{verbatim}
WHERE teams CONTAINS KEY 2015;

CREATE TABLE IF NOT EXISTS cycling.birthday_list (  
cyclist_name text PRIMARY KEY,  
blist map<text, text>  
);

CREATE INDEX IF NOT EXISTS blist_idx  
ON cycling.birthday_list ( ENTRIES(blist) )
;

SELECT *  
FROM cycling.birthday_list  
WHERE blist[ 'age' ] = '23';

CREATE TABLE IF NOT EXISTS cycling.birthday_list (  
cyclist_name text PRIMARY KEY,  
blist map<text, text>  
);

CREATE INDEX IF NOT EXISTS blist_idx  
ON cycling.birthday_list ( ENTRIES(blist) )
;

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

CREATE TABLE IF NOT EXISTS cycling.birthday_list (  
cyclist_name text PRIMARY KEY,  
blist map<text, text>  
);
Managing database resources

CREATE INDEX IF NOT EXISTS blist_values_idx
ON cycling.birthday_list ( VALUES(blist) );

SELECT *
FROM cycling.birthday_list
WHERE blist CONTAINS 'NETHERLANDS';

cyclist_name | blist
---------------+--------------------------------------------------------------
Luc HAGENAARS | {'age': '28', 'bday': '27/07/1987', 'nation': 'NETHERLANDS'}
Toine POELS | {'age': '52', 'bday': '27/07/1963', 'nation': 'NETHERLANDS'}
(2 rows)

• 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 IF NOT EXISTS cycling.race_starts (
    cyclist_name text PRIMARY KEY,
    rnumbers FROZEN<LIST<int>>
);

CREATE INDEX IF NOT EXISTS rnumbers_idx
ON cycling.race_starts ( FULL(rnumbers) );

SELECT *
FROM cycling.race_starts
WHERE rnumbers = [39, 7, 14];

cyclist_name   | rnumbers
----------------+-------------
John DEGENKOLB | [39, 7, 14]
(1 rows)

Indexing with SSTable attached secondary indexes (SASI)

SASI indexes in DSE are experimental. DataStax does not support SASI indexes for production.
SASI is significantly less resource-intensive, using less memory, disk, and CPU. It enables querying with prefix and contains on strings, similar to the SQL implementation of LIKE = "foo%" or LIKE = "%foo%", as shown in SELECT. It also supports SPARSE indexing to improve performance of querying large, dense number ranges such as time series data.

SASI takes advantage of the databases's write-once immutable ordered data model to build indexes when data is flushed from the memtable to disk. The SASI index data structures are built in-memory as the SSTable is written and flushed to disk as sequential writes before the SSTable writing completes. One index file is written for each indexed column.

SASI supports all queries already supported by CQL, and supports the LIKE operator using PREFIX, CONTAINS, and SPARSE. If ALLOW FILTERING is used, SASI also supports queries with multiple predicates using AND. With SASI, the performance pitfalls of using filtering are not realized because the filtering is not performed even if ALLOW FILTERING is used.
SASI is implemented using memory-mapped B+ trees, an efficient data structure for indexes. B+ trees allow range queries to perform quickly. SASI generates an index for each SSTable. Some key features that arise from this design are:

- SASI can reference offsets in the data file, bypassing the Bloom filter and partition indexes to go directly to where data is stored.
- When SSTables are compacted, new indexes are generated automatically.

SASI does not support collections. Regular secondary indexes can be built for collections.

### Using an SSTable Attached Secondary Index (SASI)

SSTable Attached Secondary Indexes (SASI) have improved the performance of secondary indexes but should be used with caution.

SASI indexes in DSE are experimental. DataStax does not support SASI indexes for production.

Using CQL, SSTable Attached Secondary Indexes (SASI) can be created on a non-collection column defined in a table. Secondary indexes are used to query a table that uses a column that is not normally possible to query, such as a non-primary key column. SASI implements the following types of indexes: PREFIX, CONTAINS, and SPARSE.

**PREFIX index**

- Create an index `fn_prefix` for the table `cyclist_name` on the column `firstname`. PREFIX is the default mode, so it does not need to be explicitly specified.

```cql
CREATE TABLE IF NOT EXISTS cycling.cyclist_name (
  id UUID PRIMARY KEY,
  lastname text,
  firstname text
);

CREATE CUSTOM INDEX IF NOT EXISTS fn_prefix
ON cycling.cyclist_name (firstname)
USING 'org.apache.cassandra.index.sasi.SASIIndex';

SELECT *
FROM cycling.cyclist_name;
```

<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-3f90-4c93-8f61-8afba4a803e2</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>KRIKSWIJK</td>
</tr>
<tr>
<td>e7cd5752-bc0d-4157-a80f-7523add8dbcd</td>
<td>Anna</td>
<td>VAN DER BREGGEN</td>
</tr>
</tbody>
</table>

(6 rows)

- Queries can find exact matches for values in `firstname`. Note that indexing is used for this query because the primary key `id` is not specified:

```cql
SELECT * FROM cycling.cyclist_name WHERE firstname = 'Marianne';
```

<table>
<thead>
<tr>
<th>id</th>
<th>firstname</th>
<th>lastname</th>
</tr>
</thead>
</table>

CQL for DataStax Enterprise 6.7 Latest DSE version Latest 6.7 patch: 6.7.7
Queries can find matches for values in `firstname` based on partial matches. The use of `LIKE` specifies that the match is looking for a word that starts with the letter `M`. The `%` after the letter `M` matches any characters after `M` and returns any matching values. Note that indexing is used for this query because the primary key `id` is not specified:

```cql
SELECT * FROM cycling.cyclist_name
WHERE firstname LIKE 'M%';
```

Many queries will fail to find matches based on the partial string. The following queries do not return a match or return an error.

The following query does not return a match. The first name is stored as 'Marianne'. The query provides 'MARIANNE', which does not match the alphabetic case sensitivity for 'Marianne'.

```cql
SELECT * FROM cycling.cyclist_name WHERE firstname = 'MARIANNE';
```

The placement of the `%` characters are critical. Because the index uses the `PREFIX` mode, only a trailing `%` yields results when coupled with `LIKE`. This query uses `LIKE` and provides a trailing `%`, but the lowercase `m` at the start does not match any rows:

```cql
SELECT * FROM cycling.cyclist_name WHERE firstname LIKE 'm%';
```

These queries specify patterns that do not match any rows:

```cql
SELECT * FROM cycling.cyclist_name WHERE firstname = 'M%';
SELECT * FROM cycling.cyclist_name WHERE firstname = 'M';
SELECT * FROM cycling.cyclist_name WHERE firstname = 'M%';
SELECT * FROM cycling.cyclist_name WHERE firstname = 'm';
```

These queries generate errors stating that the column is not properly indexed (after the `fn_contains` index is created in the next section, the queries run):

```cql
SELECT * FROM cycling.cyclist_name WHERE firstname LIKE '%m%';
SELECT * FROM cycling.cyclist_name WHERE firstname LIKE '%m%' ALLOW FILTERING;
SELECT * FROM cycling.cyclist_name WHERE firstname LIKE '%M%';
```

**CONTAINS index**

Create an index `fn_contains` for the table `cyclist_name` on the column `firstname`. `CONTAINS` is the specified mode, so that partial patterns specified, and not just the prefix, are matched.

```cql
CREATE CUSTOM INDEX IF NOT EXISTS fn_contains
ON cycling.cyclist_name (firstname)
USING 'org.apache.cassandra.index.sasi.SASIIndex'
```
• Queries can find exact matches for values in `firstname`. Note that indexing is used for the following query, because the primary key `id` is not specified in the query. For queries that use `CONTAINS` indexing, the `ALLOW FILTERING` phrase must be included, although the database does not actually perform the filtering.

```cql
SELECT *
FROM cycling.cyclist_name
WHERE firstname = 'Marianne'
ALLOW FILTERING;
```

```
<table>
<thead>
<tr>
<th>id</th>
<th>firstname</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>5b69626d-3f90-4c93-8f61-eabfa4a803e2</td>
<td>Marianne</td>
<td>VOS</td>
</tr>
</tbody>
</table>
```

The same results are returned as those in the first `PREFIX` index query example previously shown, using a slightly modified query.

• Queries can find matches for values in `firstname` based on partial matches. The use of `LIKE` specifies that the match is looking for a word that contains the letter M. The `%` before and after the letter M matches any characters that can return a matching value. Note that indexing is used for this query because the primary key `id` is not specified:

```cql
SELECT *
FROM cycling.cyclist_name
WHERE firstname LIKE '%M%';
```

```
<table>
<thead>
<tr>
<th>id</th>
<th>firstname</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>fb372533-eb95-4bb4-8ef6-6ef61e994ca</td>
<td>Michael</td>
<td>MATTHEWS</td>
</tr>
<tr>
<td>5b69626d-3f90-4c93-8f61-eabfa4a803e2</td>
<td>Marianne</td>
<td>VOS</td>
</tr>
</tbody>
</table>
```

The same results are returned as those in the second `PREFIX` index query example previously shown, using a slightly modified query.

• The `CONTAINS` index has a more versatile matching algorithm than `PREFIX`. The following queries return the same row. The first query specifies `%arianne`, which matches `firstname` values that end with arianne. The second query specifies `%arian%`, which matches `firstname` values that contain arian.

```cql
SELECT *
FROM cycling.cyclist_name
WHERE firstname LIKE '%arianne';
```

```cql
SELECT *
FROM cycling.cyclist_name
WHERE firstname LIKE '%arian%';
```

```
<table>
<thead>
<tr>
<th>id</th>
<th>firstname</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>5b69626d-3f90-4c93-8f61-eabfa4a803e2</td>
<td>Marianne</td>
<td>VOS</td>
</tr>
</tbody>
</table>
```
Managing database resources

- With a \texttt{CONTAINS} index, inequality pattern matching is possible. Note again the use of \texttt{ALLOW FILTERING} in the following query, which is required but causes no latency in the query response:

  ```
  SELECT *
  FROM cycling.cyclist_name
  WHERE firstname > 'Mar'
  ALLOW FILTERING;
  ```

<table>
<thead>
<tr>
<th>id</th>
<th>firstname</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<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>220844bf-4860-9dbd-9d9b-6bd3a79c4b7f</td>
<td>Paolo</td>
<td>TIRALONGO</td>
</tr>
<tr>
<td>6ab09bec-e6be-48d9-a5f8-97e6ba4c9c47</td>
<td>Steven</td>
<td>KRUIKSWIJK</td>
</tr>
</tbody>
</table>

- Like with \texttt{PREFIX} indexing, many queries will fail to find matches based on the partial string. These queries fail to find matches:

  ```
  SELECT * FROM cycling.cyclist_name WHERE firstname = 'MariAnne' ALLOW FILTERING;
  ```

  ```
  SELECT * FROM cycling.cyclist_name WHERE firstname LIKE '%m%';
  ```

  ```
  SELECT * FROM cycling.cyclist_name WHERE firstname LIKE '%M';
  ```

  ```
  SELECT * FROM cycling.cyclist_name WHERE firstname LIKE 'm%';
  ```

  The first two queries fail because of case sensitivity. ‘MariAnne’ has one uppercase letter, whereas the stored value does not. The other queries fail due to placement of the %.

- Either the \texttt{PREFIX} index or the \texttt{CONTAINS} index can be created with case sensitivity by adding an \texttt{analyzer_class} and \texttt{case_sensitive} option.

  ```
  CREATE CUSTOM INDEX IF NOT EXISTS fn_suffix_allcase
  ON cycling.cyclist_name (firstname)
  USING 'org.apache.cassandra.index.sasi.SASIIndex'
  WITH OPTIONS = {
    'mode': 'CONTAINS',
    'analyzer_class':
    'org.apache.cassandra.index.sasi.analyzer.NonTokenizingAnalyzer',
    'case_sensitive': 'false'
  };
  ```

  The \texttt{analyzer_class} used here is the non-tokenizing analyzer that does not perform analysis on the text in the specified column. The option \texttt{case_sensitive} is set to \texttt{false} to make the indexing case-insensitive.

- With the addition of the analyzer class and option, this query now works with the lowercase m:

  ```
  SELECT *
  FROM cycling.cyclist_name
  WHERE firstname LIKE '%m%';
  ```

<table>
<thead>
<tr>
<th>id</th>
<th>firstname</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>
• If queries are narrowed with an indexed column value, non-indexed columns can be specified. Compound queries can also be created with multiple indexed columns. The following example alters the table to add a column named `age`, which is included in a query at the end. The `age` column is not indexed.

```cql
ALTER TABLE cycling.cyclist_name
ADD age int;

UPDATE cycling.cyclist_name
SET age = 23
WHERE id = 5b6962dd-3f90-4c93-8f61-eabfa4a803e2;

INSERT INTO cycling.cyclist_name (id, age, firstname, lastname)
VALUES (8566eb59-07df-43b1-a21b-666a3c08c08a, 18, 'Marianne', 'DAAE');

SELECT *
FROM cycling.cyclist_name
WHERE firstname = 'Marianne'
AND age > 20
ALLOW FILTERING;
```

<table>
<thead>
<tr>
<th>id</th>
<th>age</th>
<th>firstname</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>5b6962dd-3f90-4c93-8f61-eabfa4a803e2</td>
<td>23</td>
<td>Marianne</td>
<td>VOS</td>
</tr>
</tbody>
</table>

(1 rows)

**SPARSE index**

• The `SPARSE` index is meant to improve performance of querying large, dense number ranges like timestamps for data inserted every millisecond. If the data is numeric, millions of columns values with a small number of partition keys might be included in the data. If range queries will be performed against the index, then `SPARSE` is the best index type. For numeric data that does not meet this criteria, `PREFIX` is the best index type.

```cql
CREATE CUSTOM INDEX IF NOT EXISTS fn_sparse
ON cycling.comments (created_at)
USING 'org.apache.cassandra.index.sasi.SASIIndex'
WITH OPTIONS = { 'mode': 'SPARSE' };
```

Use `SPARSE` indexing for data that is sparse (every term/column value has less than 5 matching keys). Indexing the `created_at` field in time series data (where there is typically few matching rows/events per `created_at` timestamp) is a good use case. `SPARSE` indexing is primarily an optimization for range queries, especially large ranges that span large timespans.

• To illustrate the use of the `SPARSE` index, create a table and insert some time series data:
WITH CLUSTERING ORDER BY (created_at DESC);

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-a29f9809def6, 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-a29f9809def6, '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-a29f9809def6, '2017-03-22 5:16:59.001+0400', 'Great snacks at all rest stops', 'Amy');

INSERT INTO cycling.comments (record_id, id, created_at, comment, commenter) VALUES (now(), c7fceba0-c141-4207-9494-a29f9809def6, '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-eb95-4bb4-8685-6ef61e994ca, totimestamp(now()), 'Great course', 'Michael');

INSERT INTO cycling.comments (record_id, id, created_at, comment, commenter) VALUES (now(), fb372533-eb95-4bb4-8685-6ef61e994ca, '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-eb95-4bb4-8685-6ef61e994ca, '2017-03-16 19:43:01.030-0800', 'Getting ready for the race', 'Michael');

INSERT INTO cycling.comments (record_id, id, created_at, comment, commenter) VALUES (now(), fb372533-eb95-4bb4-8685-6ef61e994ca, '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-eb95-4bb4-8685-6ef61e994ca, '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-d35c-4a8d-83f7-a3c543789e0, 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-a3c543789e0, '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(), Sb6962dd-3f90-4c93-8f61-eabf4a803e2, totimestamp(now()), 'Thanks for all your hard work', 'Marianne');
* This query returns the comments made before the timestamp 2017-02-13 11:40:16.123:

```sql
SELECT *
FROM cycling.comments
WHERE created_at < '2017-02-13 11:40:16.123';
```

<table>
<thead>
<tr>
<th>id</th>
<th>created_at</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>9011d3be-d35c-4a8d-83f7-a3c543789ee7</td>
<td>2017-01-01 17:20:17.020000+0000</td>
<td>Gearing up for the season</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Katarzyna</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c591c6e</td>
</tr>
<tr>
<td>38ab64b6-26cc-4de9-ab28-c257cf011659</td>
<td>2017-02-11 14:09:56.000000+0000</td>
<td>First race was amazing, can't wait for more</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Marcia</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c597934</td>
</tr>
<tr>
<td>6ab09bec-e68e-48d9-a5f8-97e6fb4cb9a7</td>
<td>2017-02-02 01:49:00.020000+0000</td>
<td>Best of luck everybody I can't make it</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Steven</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c59aa08</td>
</tr>
<tr>
<td>1-0c9f-1eab-db8d-9f9b8a53bf55</td>
<td>2017-02-13 11:40:16.123-0600</td>
<td>I can do without the rain@#$@</td>
</tr>
</tbody>
</table>

The query returns all the results where `created_at` is found to be less than the timestamp supplied. The inequalities `>=`, `>`, and `<` are all valid operators.

**SPARSE** indexing is used only for numeric data, so LIKE queries do not apply.
Managing database resources

Using analyzers

- Analyzers can be specified that will analyze the text in the specified column. The `NonTokenizingAnalyzer` is used for cases where the text is not analyzed, but case normalization or sensitivity is required. The `StandardAnalyzer` is used for analysis that involves stemming, case normalization, case sensitivity, skipping common words like "and", "the", and "or", and localization of the language used to complete the analysis. This example adds a text column named `comment` to the `cyclist_name` table to provide details about the cyclist, adds some comments, and then creates the index:

```sql
ALTER TABLE cycling.cyclist_name
ADD comment text;

UPDATE cycling.cyclist_name
SET comment = 'Rides hard, gets along with others, a real winner'
WHERE id = fb372533-eb95-4bb4-8685-6ef61e994caa;

UPDATE cycling.cyclist_name
SET comment = 'Rides fast, does not get along with others, a real dude'
WHERE id = 5b6962dd-3f90-4c93-8f61-eabfa4a803e2;

CREATE CUSTOM INDEX IF NOT EXISTS stdanalyzer_idx
ON cycling.cyclist_name (comment) USING 'org.apache.cassandra.index.sasi.SASIIndex'
WITH OPTIONS = {
    'mode': 'CONTAINS',
    'analyzer_class': 'org.apache.cassandra.index.sasi.analyzer.StandardAnalyzer',
    'analyzed': 'true',
    'tokenization_skip_stop_words': 'and, the, or',
    'tokenization_enable_stemming': 'true',
    'tokenization_normalize_lowercase': 'true',
    'tokenization_locale': 'en'
};
```

- This query searches for the presence of a designated string, using the analyzed text to return a result:

```sql
SELECT *
FROM cycling.cyclist_name
WHERE comment LIKE 'ride';
```

```
+----------------------------------------+---------------------------------------------------------+-----------+----------+
<table>
<thead>
<tr>
<th>id</th>
<th>comment</th>
<th>firstname</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>fb372533-eb95-4bb4-8685-6ef61e994caa</td>
<td>Rides hard, gets along with others, a real winner</td>
<td>Michael</td>
<td>MATTHEWS</td>
</tr>
<tr>
<td>5b6962dd-3f90-4c93-8f61-eabfa4a803e2</td>
<td>Rides fast, does not get along with others, a real dude</td>
<td>Marianne</td>
<td>VOS</td>
</tr>
</tbody>
</table>
```

The query returns all the results where `ride` is found either as an exact word or as a stem for another word, which is `rides` in this example.

Building and maintaining indexes

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

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

Managing search indexes

In DSE Search, a search index is an Apache Solr™ core. Each DSE Search index uses an internally stored index configuration pair (schema.xml and solrconfig.xml) that is automatically generated when the index is created.

About search commands

Manage DSE search indexes using the following CQL commands:

- **CREATE SEARCH INDEX** Generates a new search index on an existing table with default schema and config.
- **DESCRIBE SEARCH INDEX** Displays the active or pending schema or config in XML format.
- **ALTER SEARCH INDEX CONFIG** Modifies the search index config. After modifying, use `reload` to push the changes live.
- **ALTER SEARCH INDEX SCHEMA** Modifies the search index schema. After modifying, use `reload` to push the changes live.
- **RELOAD SEARCH INDEX** Loads pending changes to the index schema and config. Some changes such as adding or removing indexed fields require a rebuild.
- **REBUILD SEARCH INDEX** Reconstructs the search index using the active schema and config.
- **COMMIT SEARCH INDEX** Forces a reload of data into the index after data is added, modified, or removed from the corresponding CQL table.
- **DROP SEARCH INDEX** Removes the search index and corresponding files.

Run search index commands only on the local node using the `dsetool` commands.

DSE stores the Solr search index configuration pair (schema and config) in the `solr_admin.solr_resources` table and persists it to other search nodes in the cluster.

In DSE authorization-enabled environments, you must grant permission to run search index commands; see Controlling access to search indexes.

Adjusting timeout for index management

When running search index management commands on large datasets using `cqlsh` or `dsetool`, the process might take longer than the default timeout period (10 minutes).

Temporarily increase the timeout period for index management commands by setting an environment variable:

- **cqlsh**: Before starting a `cqlsh` session, set the `CQLSH_SEARCH_MANAGEMENT_TIMEOUT_SECONDS` environment variable:

  ```
  export CQLSH_SEARCH_MANAGEMENT_TIMEOUT_SECONDS=900;
  ```

  Overrides the `cqlsh` **--request-timeout** setting.

- **dsetool**: Before running an index management command, set the `dse.search.client.timeout.secs`:

  ```
  export JVM_OPTS="-Ddse.search.client.timeout.secs=900"
  ```

  Overrides the default timeout.
Managing database resources

Creating an index

Use the `CREATE SEARCH INDEX` command to generate a search index for an existing table.

Indexes created with CQL commands are automatically distributed to all search nodes in the datacenter.

Solr field name policy applies to the indexed field names:

- Every field must have a name.
- Field names must consist of alphanumeric or underscore characters only.
- Fields cannot start with a digit.
- Names with both leading and trailing underscores (for example, `version`) are reserved.

Non-compliant field names are not supported from all components. Backward compatibility is not guaranteed.

Starting cqlsh on a search node

Connect to a search node to use CQL search management commands.

1. Determine which nodes in the cluster are running search:

   ```
   $ dsetool status
   ```

   DSE Search operations are available only on search-enabled nodes. DataStax recommends single workload datacenters.

   The following example shows a development environment where all nodes in the cluster are in the same physical location, on the same rack, and the nodes have been separated into datacenters based on their workloads.

<table>
<thead>
<tr>
<th>DC: Main</th>
<th>Workload: Cassandra</th>
<th>Graph: no</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status-Up/Down</td>
<td>State=Normal/Leaving/Joining/Moving</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>Load</td>
<td>Owns</td>
</tr>
<tr>
<td>Health [0,1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UN 10.10.10.111</td>
<td>15.51 MiB</td>
<td>?</td>
</tr>
<tr>
<td>UN 10.10.10.113</td>
<td>19.51 MiB</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DC: Search</th>
<th>Workload: Search</th>
<th>Graph: no</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status-Up/Down</td>
<td>State=Normal/Leaving/Joining/Moving</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>Load</td>
<td>Owns</td>
</tr>
<tr>
<td>Health [0,1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UN 10.10.10.108</td>
<td>18.13 MiB</td>
<td>?</td>
</tr>
<tr>
<td>rack1</td>
<td>0.90</td>
<td></td>
</tr>
</tbody>
</table>
2. For large datasets, increase the cqlsh timeout:

```bash
export CQLSH_SEARCH_MANAGEMENT_TIMEOUT_SECONDS=900;
```

3. Launch a cqlsh session on a search node:

```bash
$cqlsh hostname
```

A CQL sessions starts on the remote host.

```
Connected to cluster1 at 10.10.10.108:9042.
[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>
```

### Creating a search index with default values

Use the DataStax Enterprise `CREATE SEARCH INDEX` to generate a search index for an existing table that is automatically distributed to all search nodes.

The search index (schema and config) is generated using default values. The schema and config are stored internally in the `solr_admin.resources` table and displayed in XML format.

Create a search index on an existing table.

```cql
CREATE SEARCH INDEX ON keyspace_name.table_name;
```

All columns are indexed using the default settings.

### Setting up default query field

Set up a catch-all field for searches when no field is specified by the query.

1. Create a new index-only field:

```cql
ALTER SEARCH INDEX SCHEMA ON wiki.solr
ADD fields.field[ @name='catch_all',
  @type='TextField',
  @multiValued='true'];
```

Since this new field contains values from two fields, set multiValued to true.

Show the pending schema changes:

```cql
DESCRIBE PENDING SEARCH INDEX SCHEMA ON wiki.solr ;
```

The new field is listed in bold:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<schema name="autoSolrSchema" version="1.5">
  <types>
    <fieldType class="org.apache.solr.schema.TextField" name="TextField">
      <tokenizer class="solr.StandardTokenizerFactory"/>
      <filter class="solr.LowerCaseFilterFactory"/>
    </fieldType>
  </types>
</schema>
```
Managing database resources

2. Set up a copy field directive to collect the data from all CQL columns:

```
ALTER SEARCH INDEX SCHEMA ON wiki.solr
ADD copyField[source='title', @dest='catch_all'];
ALTER SEARCH INDEX SCHEMA ON wiki.solr
ADD copyField[source='body', @dest='catch_all'];
```

Show the pending schema changes:

```
DESCRIBE PENDING SEARCH INDEX SCHEMA ON wiki.solr;
```

The new copy field directives are listed in bold below:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<schema name="autoSolrSchema" version="1.5">
  <types>
    <fieldType class="org.apache.solr.schema.TextField" name="TextField">
      <analyzer>
        <tokenizer class="solr.StandardTokenizerFactory"/>
        <filter class="solr.LowerCaseFilterFactory"/>
      </analyzer>
    </fieldType>
    <fieldType class="org.apache.solr.schema.TrieDateField" name="TrieDateField"/>
    <fieldType class="org.apache.solr.schema.StrField" name="StrField"/>
  </types>
  <fields>
    <field indexed="true" multiValued="false" name="body" stored="true" type="TextField"/>
    <field docValues="true" indexed="true" multiValued="false" name="real_date" stored="true" type="TrieDateField"/>
    <field indexed="true" multiValued="false" name="title" stored="true" type="TextField"/>
    <field indexed="true" multiValued="false" name="id" stored="true" type="StrField"/>
    <field indexed="true" multiValued="false" name="date" stored="true" type="TextField"/>
    <field indexed="true" multiValued="true" name="catch_all" type="TextField"/>
  </fields>
  <uniqueKey>id</uniqueKey>
</schema>
```
3. Define the default field in the search index config:

```sql
ALTER SEARCH INDEX CONFIG ON wiki.solr
SET defaultQueryField = 'catch_all';
```

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<config>
  <luceneMatchVersion>LUCENE_6_0_1</luceneMatchVersion>
  <dseTypeMappingVersion>2</dseTypeMappingVersion>
  <directoryFactory class="solr.StandardDirectoryFactory" name="DirectoryFactory"/>
  <indexConfig>
    <ramBufferSizeMB>512</ramBufferSizeMB>
    <rt>false</rt>
  </indexConfig>
  <jmx/>
  <updateHandler>
    <autoSoftCommit>
      <maxTime>10000</maxTime>
    </autoSoftCommit>
  </updateHandler>
  <query>
    <filterCache class="solr.SolrFilterCache" highWaterMarkMB="2048"
      lowWaterMarkMB="1024"/>
    <enableLazyFieldLoading>true</enableLazyFieldLoading>
    <useColdSearcher>true</useColdSearcher>
    <maxWarmingSearchers>16</maxWarmingSearchers>
  </query>
  <requestDispatcher>
    <requestParsers enableRemoteStreaming="true"
      multipartUploadLimitInKB="2048000"/>
    <httpCaching never304="true"/>
  </requestDispatcher>
  <requestHandler class="solr.SearchHandler" default="true" name="search">
    <lst name="defaults">
      <str name="df">catch_all</str>
    </lst>
  </requestHandler>
  <requestHandler class="com.datastax.bdp.search.solr.handler.component.CqlSearchHandler"
    name="solr_query">
    <lst name="defaults">
      <str name="df">catch_all</str>
    </lst>
  </requestHandler>
  <requestHandler class="solr.UpdateRequestHandler" name="/update"/>
  <requestHandler class="solr.UpdateRequestHandler" name="/update/csv"
    startup="lazy"/>
  <requestHandler class="solr.UpdateRequestHandler" name="/update/json"
    startup="lazy"/>
  <requestHandler class="solr.FieldAnalysisRequestHandler" name="/analysis/field"
    startup="lazy"/>
  <requestHandler class="solr.DocumentAnalysisRequestHandler" name="/analysis/document"
    startup="lazy"/>
  <requestHandler class="solr.admin.AdminHandlers" name="/admin"/>
  <requestHandler class="solr.PingRequestHandler" name="/admin/ping">
    <lst name="invariants">
      <str name="qt">search</str>
      <str name="q">solrpingquery</str>
    </lst>
    <lst name="defaults">
      <str name="echoParams">all</str>
    </lst>
  </requestHandler>
</config>
```
Managing database resources

Generating an index with joins disabled
By default, the partition key fields are combined into a single field, _partitionKey, and stored as a string field to support joins between indexes. When join is not required, create an index with join disabled.

4. Reload the schema and config to make the pending search index schema and config active:

RELOAD SEARCH INDEX ON wiki.solr ;

5. Rebuild the index to update the search index for the existing data:

REBUILD SEARCH INDEX ON wiki.solr ;

Generating an index with joins disabled
By default, the partition key fields are combined into a single field, _partitionKey, and stored as a string field to support joins between indexes. When join is not required, create an index with join disabled.

To disable joins after an index has been created, see Configuring search index joins.

1. Create a search index with join disabled:
The PROFILES spaceSavingNoJoin option disables joins when creating a search index. For example:

CREATE SEARCH INDEX ON demo.health_data WITH PROFILES spaceSavingNoJoin;

2. Verify that joins are disabled:

DESCRIBE ACTIVE SEARCH INDEX SCHEMA ON demo.health_data ;

Managing search index fields
Add, remove, and change indexing definitions for table columns in the search index schema.

Syntax for changing schema settings
Use the ALTER SEARCH INDEX to add, set, or drop settings of an existing search index schema.

The search index schema is in XML format and supports most Solr schema.xml elements:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<schema name="autoSolrSchema" version="1.5">
  <types>
    <fieldType class="class_name" name="type_name"/>
    <analyzer>
      <tokenizer class="class_name"/>
      <filter class="class_name"/>
    </analyzer>
  </types>
  <fields>
  </fields>
</schema>
```
CQL ALTER SEARCH INDEX SCHEMA basic syntax:

```
ALTER SEARCH INDEX SCHEMA ON keyspace_name.table_name
ADD ([shortcut] | element_path) [ element_definition | WITH $$jsonsnippet$$];
```

where:

**Using shortcut keywords**

Use shortcuts field, fieldType, and copyField to:

- Add or drop table columns from the index, for example:

  ```
  ALTER SEARCH INDEX SCHEMA ON demo.health_data ADD field gender;
  ```

  If the field name matches a column name the field definition is automatically added to the pending schema.

- Identify the element (field, fieldType, and copyField) and then change the setting using an element path or JSON definition:

  ```
  ALTER SEARCH INDEX SCHEMA ON wiki.solr ADD copyField[@source='title', @dest='catch_all'];
  ```

**Using element paths**

The element path uniquely describes the setting in the schema XML. Enclose attributes after an element in brackets; to define multiple attributes use a comma-separated list. When adding an element, include all of the attributes.

```
top_level_element_name.child_element_name[@attribute_name='value', ...]
```

For example to add a the Text field type definition:

```
types.fieldType[@name='TextField_intl', @class='org.apache.solr.schema.TextField']
```

The element path can also be used to describe a sub-element in the schema.

You can use element_path to change a sub-element in the schema. For example, to change the ASCII FoldingFilterFactory in the search analyzer to a ClassicFilterFactory:

```
ALTER SEARCH INDEX SCHEMA ON demo.users
SET types.fieldType[@name='TextField_intl']
.analyzer[@type='search']
.filter[@class='solr.ASCIIFoldingFilterFactory']@class='solr.ClassicFilterFactory';
```

Changes the fieldType to:

```
<fieldType class="org.apache.solr.schema.TextField" name="TextField_intl">
  ...
```
Managing database resources

```
<analyzer type="search">
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.ClassicFilterFactory"/>
    <tokenizer class="solr.StandardTokenizerFactory"/>
</analyzer>
</fieldType>
```

**Defining complex elements with JSON**

This JSON snippet is translated into XML elements and attributes:

- JSON pair translates to XML attribute.
- JSON object translates to XML element.

The JSON is translated into these XML attributes:

```json
$$
"analyzer": [ 
    { 
        "type": "index",
        "tokenizer": { "class": "solr.StandardTokenizerFactory" },
        "filter": [ 
            { "class": "solr.LowerCaseFilterFactory" },
            { "class": "solr.ASCIIFoldingFilterFactory" }
        ]
    }, 
    { 
        "type": "search",
        "tokenizer": { "class": "solr.StandardTokenizerFactory" },
        "filter": [ 
            { "class": "solr.LowerCaseFilterFactory" },
            { "class": "solr.ASCIIFoldingFilterFactory" }
        ]
    }
], $$
```

The JSON is translated into these XML elements:

```
"analyzer": [ 
    { "type": "index", <analyzer type="index"> 
    "tokenizer": 
        { "class": "solr.StandardTokenizerFactory" }, <tokenizer class="solr.StandardTokenizerFactory"/> 
    "filter": [ 
        { "class": "solr.LowerCaseFilterFactory" }, <filter class="solr.LowerCaseFilterFactory"/> 
        { "class": "solr.ASCIIFoldingFilterFactory" } <filter class="solr.ASCIIFoldingFilterFactory"/> 
    ]
], $$
```

**Removing elements or attributes**

The CQL command syntax to remove the second filter on the search phase analyzer:

```
ALTER SEARCH INDEX SCHEMA ON demo.users
DROP types.fieldType[@name='TextField_intl'].analyzer[@type='search'].filter[@class='solr.ClassicFilterFactory'];
```

Changes the fieldType to:

```
<fieldType class="org.apache.solr.schema.TextField" name="TextField_intl">
    ... 
```
<analyzer type="search">
   <filter class="solr.LowerCaseFilterFactory"/>
   <tokenizer class="solr.StandardTokenizerFactory"/>
</analyzer>
</fieldType>

**Schema**

Describes the CQL columns to index, sets the Solr data type, defines how to index and search each field type, and defines the primary key.

The schema displays in XML format. Use element paths to define and identify elements and attributes:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<schema name="autoSolrSchema" version="1.5">
   <types>
      <fieldType class="class_path" name="fieldtype_name">
         <analyzer>
            <tokenizer class="class_path"/>
            <filter class="class_path"/>
         </analyzer>
      </fieldType>
   </types>
   <fields>
      <field
         attribute_name="value"
         docValues="true|false"
         indexed="true|false"
         multiValued="true|false"
         name="column_name"
         stored="true|false"
         type="fieldtype_name"/>
      <copyField
         source="field_name"
         dest="field_name"/>
   </fields>
   <uniqueKey>pk_column_list</uniqueKey>
</schema>
```

**ADD FIELD column_name**

Adds a column from the CQL table to the pending search index schema using the default mapping.

```
ALTER SEARCH INDEX SCHEMA ON demo.health_data ADD fields.field fips;
```

Adding the leading element `fields` in `ADD fields.field fieldname` is optional and provides only cosmetic structure.

**ADD fields.field[@attribute_name='value', ...]**

Adds a new field to the pending schema and manually set the attributes. For example, to add a column from the table to the index and set the field type to string.

```
ALTER SEARCH INDEX SCHEMA ON demo.health_data
ADD fields.field[@name='fish', @type='StrField', @indexed='true'];
```

**ADD copyField[@attribute_name='value', ...]**
Managing database resources

Copy the value of the source field to a new field. For example, as a workaround to the rule that you cannot use `LowerCaseStrField` on a primary key column, you can use `copyField` to copy the primary key field data to a new field defined as type `LowerCaseStrField`.

```
ALTER SEARCH INDEX SCHEMA ON <table> ADD lowerCaseString key_column_copy;
```

```
ALTER SEARCH INDEX SCHEMA ON <table> ADD copyField[@source='key_column', @dest='key_column_copy'];
```

**DROP field field_name**

Removes a field from the pending search index schema.

```
ALTER SEARCH INDEX SCHEMA ON demo.health_data DROP field fips;
```

**SET fields.field[@name='field_name']@attribute_name='value'**

Changes the field identified by the attribute in brackets by adding or replacing the `attribute_to_change`.

Field attributes:

- **name**: Matches a CQL table column name or the name of a copyField destination.
- **type**: Name of a defined fieldType.
- **indexed**: True indicates that the field is indexed. By default, only the fields that are included in the index on creation are displayed.
  - Primary key columns must be indexed (`indexed="true"`).
- **docValues**: Creates a forward index on the field values.
- **multiValued**: Contains more than one value, such as a set, map, list column, or the destination of multiple `copyField` definitions.

The stored field attribute is not supported.

```
ALTER SEARCH INDEX SCHEMA ON demo.health_data
SET fields.field[@name='gender_s']@multiValued='true';
```

**ADD types.fieldType[@attribute_name='value', ...] WITH $$ { json_map } $$**

Adds a field type definition to the schema for analyzing, tokenizing, and filtering fields in the index.

```
ADD types.fieldType[@name='TrieIntField', @class='solr.TrieIntField']
```

Optionally add the leading element `fields` in `ADD` and `SET fields.field field_name` to follow a naming convention and provide structure.

**Understanding schema field types**

DataStax Enterprise Search index schemas define the CQL to Solr type mapping. A field type definition is required to parse data from a CQL column into the corresponding Solr field in the index. The `fieldType` definitions support processing instructions in the analyzer section.

Decimal and varint are indexed as strings. Apache Lucene® does not support the precision required by these numeric types. Range and sorting queries do not work as expected if a table uses the decimal and varint types; equality searches work as expected.
CQL data type compatibility with Solr field type classes

DSE Search index schema defines the Solr field type of each indexed CQL column. The default mapping provides support for the most common use cases. To perform more advanced searches or a different type of analysis on a column, use the following compatibility matrix to determine the supported CQL to Solr type mappings.
<table>
<thead>
<tr>
<th>CQL data type</th>
<th>Solr field type</th>
<th>Supports docValues</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ascii</td>
<td>AsciiStrField</td>
<td>false</td>
<td>Indexed as a standard Solr StrField.</td>
</tr>
<tr>
<td>bigint</td>
<td>TrieLongField</td>
<td>true</td>
<td>Long field for Lucene TrieRange processing.</td>
</tr>
<tr>
<td>blob</td>
<td>BinaryField</td>
<td>not supported</td>
<td>Binary data.</td>
</tr>
<tr>
<td>boolean</td>
<td>BoolField</td>
<td>false</td>
<td>True (1, t, or T) or False (not 1, t, or T)</td>
</tr>
<tr>
<td>counter</td>
<td>not indexable</td>
<td>N/A</td>
<td>64-bit signed integer.</td>
</tr>
<tr>
<td>date</td>
<td>SimpleDateField (DataStax custom field)</td>
<td>not supported</td>
<td>TrieDateField holding a CQL date.</td>
</tr>
<tr>
<td>DateRangeType</td>
<td>DateRangeField</td>
<td>not supported</td>
<td>Point-in-time with millisecond precision with support for date ranges. See Using date ranges in solr_query.</td>
</tr>
<tr>
<td>decimal</td>
<td>DecimalStrField</td>
<td>false</td>
<td>Indexed as a standard Solr StrField.</td>
</tr>
<tr>
<td>double</td>
<td>TrieDoubleField</td>
<td>true</td>
<td>Double field for Lucene TrieRange processing.</td>
</tr>
<tr>
<td>DseExecutorState</td>
<td>not indexable</td>
<td>N/A</td>
<td>Reserved data type for Spark.</td>
</tr>
<tr>
<td>duration</td>
<td>not indexable</td>
<td>N/A</td>
<td>Range of time, such as 12h30m.</td>
</tr>
<tr>
<td>text, varchar</td>
<td>EnumField</td>
<td>not supposed</td>
<td>A closed set with a pre-determined sort order.</td>
</tr>
<tr>
<td>text, varchar</td>
<td>ExternalFileField</td>
<td>not supported</td>
<td>Values from disk file.</td>
</tr>
<tr>
<td>float</td>
<td>TrieFloatField</td>
<td>true</td>
<td>Floating point field for Lucene TrieRange processing.</td>
</tr>
<tr>
<td>text, varchar</td>
<td>GeoHashField</td>
<td>not supported</td>
<td>Hash of coordinate pair (latitude,longitude) stored as a string.</td>
</tr>
<tr>
<td>inet</td>
<td>InetField</td>
<td>false</td>
<td>InetField is implemented and indexed as a standard Solr StrField.</td>
</tr>
<tr>
<td>int</td>
<td>TrieIntField</td>
<td>true</td>
<td>32-bit signed integer field for Lucene TrieRange processing.</td>
</tr>
<tr>
<td>text, varchar</td>
<td>LatLonType</td>
<td>not supported</td>
<td>Latitude/Longitude 2-D point, latitude first.</td>
</tr>
<tr>
<td>text, varchar</td>
<td>LowerCaseStrField</td>
<td>true</td>
<td>Recommended method for case-insensitive text search, faceting, grouping, and sorting. Sets field values as lowercase and stores them as lowercase in docValues. See Using LowerCaseStrField with search indexes.</td>
</tr>
<tr>
<td>LineStringType</td>
<td>SpatialRecursivePrefixTreeFieldType</td>
<td>true</td>
<td>Spatial field type for a point geospatial context.</td>
</tr>
<tr>
<td>PointType</td>
<td>SpatialRecursivePrefixTreeFieldType</td>
<td>true</td>
<td>Spatial field type for a point geospatial context.</td>
</tr>
<tr>
<td>text, varchar</td>
<td>PointType</td>
<td></td>
<td>Coordinate values, two per point for spatial search.</td>
</tr>
<tr>
<td>CQL data type</td>
<td>Solr field type</td>
<td>Supports docValues</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------------------------------------</td>
<td>--------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PolygonType</td>
<td>SpatialRecursivePrefixTreeFieldType</td>
<td>true</td>
<td>Spatial field type for a point [geospatial] context.</td>
</tr>
<tr>
<td>text, varchar</td>
<td>RandomSortField</td>
<td>false</td>
<td>Dynamic field in random order.</td>
</tr>
<tr>
<td>text, varchar</td>
<td>SpatialRecursivePrefixTreeFieldType</td>
<td>true</td>
<td>Spatial field type for a geospatial context.</td>
</tr>
<tr>
<td>smallint</td>
<td>TrieIntField</td>
<td>true</td>
<td>32-bit signed integer field for Lucene TrieRange processing.</td>
</tr>
<tr>
<td>text, varchar</td>
<td>StrField</td>
<td>false</td>
<td>String (UTF-8 encoded string or Unicode).</td>
</tr>
<tr>
<td>text</td>
<td>TextField</td>
<td>false</td>
<td>Text, usually multiple words or tokens.</td>
</tr>
<tr>
<td>time</td>
<td>TimeField</td>
<td>true</td>
<td>A TrieLongField holding a CQL time.</td>
</tr>
<tr>
<td>timestamp</td>
<td>TrieDateField</td>
<td>true</td>
<td>Date field for Lucene TrieRange processing; supports indexing negative dates.</td>
</tr>
<tr>
<td>timeuuid</td>
<td>TimeUUIDField</td>
<td>true</td>
<td>Type 1 Universally Unique Identifier (UUID).</td>
</tr>
<tr>
<td>N/A</td>
<td>TrieField</td>
<td>true</td>
<td>Same as any Trie field type.</td>
</tr>
<tr>
<td>tinyint</td>
<td>TrieIntField</td>
<td>true</td>
<td>32-bit signed integer field for Lucene TrieRange processing.</td>
</tr>
<tr>
<td>tuple</td>
<td>TupleField</td>
<td>false</td>
<td>Fixed length set of elements of different types.</td>
</tr>
<tr>
<td>user defined type (UDT)</td>
<td>TupleField</td>
<td>false</td>
<td>User-defined type.</td>
</tr>
<tr>
<td>uuid</td>
<td>UUIDField</td>
<td>true</td>
<td>Universally Unique Identifier (UUID).</td>
</tr>
<tr>
<td>varchar</td>
<td>TextField</td>
<td>false</td>
<td>Text, usually multiple words or tokens.</td>
</tr>
<tr>
<td>varint</td>
<td>VarIntStrField</td>
<td>false</td>
<td>Indexed as a standard Solr StrField.</td>
</tr>
</tbody>
</table>
Managing database resources

The list and set collection types are Solr multiValued fields; maps are dynamicFields. The counter, duration, and DseExecutorState types are not Solr indexable.

Adding a new field type

Add the Solr field type definitions to the search index schema, and then use the new type.

1. Add the field type definition if it does not exist:

   ALTER SEARCH INDEX SCHEMA ON [keyspace_name.]table_name
   ADD types.fieldtype[@class='field_class', @name='type_name'];

2. Change the type of the field:

   ALTER SEARCH INDEX SCHEMA ON [keyspace_name.]table_name
   SET field[@name='column_name']@type='fieldtype_name';

   If a field name in the schema matches a table column, the column is indexed.

3. Verify the pending changes:

   DESCRIBE PENDING SEARCH INDEX SCHEMA ON [keyspace_name.]table_name;

4. Activate the changes:

   RELOAD SEARCH INDEX ON [keyspace_name.]table_name;

   Copies the pending schema over the active schema. New transactions, such as data inserted into the table, are processed using the active schema. The existing data is not effected by a schema change.

5. Rebuild the index:

   REBUILD SEARCH INDEX ON [keyspace_name.]table_name;

   The REBUILD SEARCH INDEX regenerates the index using existing data. Rebuilding is required when changing the way that data is indexed, such as changing the type of field or if a field is added to the index.

To run a faceted queries using the gender field, change the type to StrField.

1. Add the Solr string field type to the health_data table:

   ALTER SEARCH INDEX SCHEMA ON demo.health_data
   SET types.fieldtype[@class='org.apache.solr.schema.StrField']@name='StrField';

2. Change the gender field type:

   ALTER SEARCH INDEX SCHEMA ON demo.health_data
   SET field[@name='gender']@type='StrField';

   See Adding a column to the index.
Adding a column to the index

Add a table column to the index. Field types are inferred when fields are added. The field types are added if they do not exist in the schema. Field type names are generated using the field type name as the simple name of the field type.

Adding the leading element fields. in ADD fields.field fieldname is optional and provides only cosmetic structure.

1. Add a table column to the index:
   
   - **Add a regular column**
     
     For example, to add a field to the wiki demo index:

     ```
     ALTER TABLE wiki.solr ADD intfield int;
     ALTER SEARCH INDEX SCHEMA ON wiki.solr ADD fields.field intfield;
     ```

     Adds the following field:

     ```
     <field indexed="true" multiValued="false" name="intfield" stored="true"
     type="TrieIntField" />
     ```

     And the following field type:

     ```
     <fieldType name="TrieIntField" class="org.apache.solr.schema.TrieIntField"/>
     ```

   - **Add a table column that is a Tuple or UDT**
     
     Tuple columns are added as multiple fields:

     ```
     ALTER TABLE solr.wiki ADD fieldname tuple<text,int>;
     ALTER SEARCH INDEX SCHEMA ON solr.wiki ADD fields.field fieldname;
     ```

     Adds the following to the schema:

     ```
     <field indexed="true" multiValued="false" name="fieldname" stored="true"
     type="TupleField" />
     <field indexed="true" multiValued="false" name="fieldname.field1" stored="true"
     type="TextField" />
     <field indexed="true" multiValued="false" name="fieldname.field2" stored="true"
     type="TrieIntField" />
     ```

     Adding the leading element fields. in ADD fields.field fieldname is optional and provides only cosmetic structure.

2. Verify the pending changes:

   ```
   DESCRIBE PENDING SEARCH INDEX SCHEMA ON [keyspace_name.]table_name;
   ```

3. Activate the changes:

   ```
   RELOAD SEARCH INDEX ON [keyspace_name.]table_name;
   ```

   Copies the pending schema over the active schema. New transactions, such as data inserted into the table, are processed using the active schema. The existing data is not effected by a schema change.
Managing database resources

4. Rebuild the index:

    REBUILD SEARCH INDEX ON [keyspace_name.]table_name;

The REBUILD SEARCH INDEX regenerates the index using existing data. Rebuilding is required when changing the way that data is indexed, such as changing the type of field or if a field is added to the index.

Indexing tuples and UDTs fields

Guidelines

Guidelines for advanced data types, including tuples and user-defined types (UDT):

- The tuple data type holds fixed-length sets of typed positional fields. Use a tuple as an alternative to a UDT.
- A UDT facilitates handling multiple fields of related information in a table. UDTs are a specialization of tuples. All examples and documentation references to tuples apply to both tuples and UDTs.
  
  Simplify applications that require multiple tables by using UDTs to represent the related fields of information, instead of storing the information in a separate table.

DSE Search does not support:

- Tuples and UDTs that are used inside primary key declarations.
- Tuples and UDTs that are used as CQL map values. Instead, use a workaround to simulate a map-like data model.
- Dynamic fields as tuples or UDTs.
- Tuple/UDT subfield sorting and faceting.

Performance and memory

Tuples and UDTs are read and written as a single unit of information. Consider performance and memory impact when working with tuples and UDTs. Subfields are managed as the full tuple or UDT, and are not handled individually.

Highlights

Add CQL tuple and user-defined type (UDT) columns to an existing search index.

- Define a field for the table column using the DataStax Tuple class
  (com.datastax.bdp.search.solr.core.types.TupleField).
- Define a field for each value in the CQL tuple or UDT column using the corresponding Solr field type.
  
  The schema field name is column_name.fieldN where the column_name matches the CQL column and 
  N is the field position starting at 1.

Tuple configuration example

- Tuples
  
  Tuple columns are added as multiple fields:

  ALTER TABLE solr.wiki ADD fieldname tuple<text,int>;
  ALTER SEARCH INDEX SCHEMA ON solr.wiki ADD fields.field fieldname;

  Adds the following to the schema:

  <field indexed="true" multiValued="false" name="fieldname" stored="true"
  type="TupleField" />

CQL for DataStax Enterprise 6.7 Latest DSE version Latest 6.7 patch: 6.7.7
Adding the leading element fields. in ADD fields.field fieldname is optional and provides only cosmetic structure.

Drops the TupleField and all the child fields when dropping the base field name:

```
ALTER SEARCH INDEX SCHEMA ON solr.wiki DROP field fieldname;
```

To drop individual child fields:

```
ALTER SEARCH INDEX SCHEMA ON solr.wiki DROP field "fieldname.field1";
```

- Tuples
  
  Tuple columns are added as multiple fields:

```
ALTER TABLE solr.wiki ADD fieldname tuple<text,int>;
ALTER SEARCH INDEX SCHEMA ON solr.wiki ADD fields.field fieldname;
```

Adds the following to the schema:

```
<field indexed="true" multiValued="false" name="fieldname" stored="true" type="TupleField" />
<field indexed="true" multiValued="false" name="fieldname.field1" stored="true" type="TextField" />
<field indexed="true" multiValued="false" name="fieldname.field2" stored="true" type="TrieIntField" />
```

Adding the leading element fields. in ADD fields.field fieldname is optional and provides only cosmetic structure.

Drops the TupleField and all the child fields when dropping the base field name:

```
ALTER SEARCH INDEX SCHEMA ON solr.wiki DROP field fieldname;
```

To drop individual child fields:

```
ALTER SEARCH INDEX SCHEMA ON solr.wiki DROP field "fieldname.field1";
```

**UDT configuration example**

Example steps to configure a UDT for DSE Search.

**In the search schema, declare the UDTField class**

```
<fieldType class="com.datastax.bdp.search.solr.core.types.TupleField"
name="UDTField"/>
```

Use CQL commands to manage search indexes.
Managing database resources

Create a type with the UDT

You must create a type for UDTs.

```sql
CREATE TYPE Address (street text, city text)
```

Create a table with the tuple

```sql
CREATE TABLE Location ( id text primary key, address frozen<Address> );
```

Configure the UDTField in the search schema

```xml
<field name="address" type="UDTField" indexed="true" stored="true"/>
<field name="address.street" type="text" indexed="true" stored="true"/>
<field name="address.city" type="text" indexed="true" stored="true"/>
```

Nesting tuples and UDTs

DSE Search supports queries for nested tuples and UDTs. For example, you can nest and declare tuples and UDTs inside CQL lists and sets. You cannot nest tuples and UDTs inside maps or keys.

Create a type with the Address tuple

```sql
CREATE TYPE Address (street text, city text, residents set<tuple<text, text>>)
```

Create a table with the Address tuple

```sql
CREATE TABLE Location (id text, address Address)
```

In the search schema, declare the TupleField and the nested TupleField

```xml
<field name="address" type="TupleField" indexed="true" stored="true"/>
<field name="address.street" type="text" indexed="true" stored="true"/>
<field name="address.city" type="text" indexed="true" stored="true"/>
<field name="address.residents" type="TupleField" indexed="true" stored="true" multiValued="true"/>
<field name="address.residents.field1" type="text" indexed="true" stored="true"/>
<field name="address.residents.field2" type="text" indexed="true" stored="true"/>
```

The residents nested tuple is TupleField. Each nested field is concatenated with each parent tuple or UDT by using periods.

Tuples and UDTs as CQL map values

DSE Search does not support using tuples and UDTs as CQL map values. Use this workaround to simulate a map-like data model.

1. Declare a collection of tuples or UDTs that have a type field that represents what would have been the map key:
Create the tuple type. The tuple type applies to tuples and UDTs.

```
CREATE TYPE Address (type text, street text, city text)
```

Create table for UDT:

```
CREATE TABLE Person (name text primary key, addresses set<frozen<Address>>)
```

Or create a table for a tuple:

```
CREATE TABLE Person (name text primary key, addresses set<frozen<tuple<text, text, text>>>)
```

2. Using this collection of tuples or UDTs as a map-like data model, it is possible to query for person addresses of a given type (key).

For example, to query for persons whose home address is in London:

```
{!tuple}addresses.type:Home AND addresses.city:London
```

**Indexing map columns**

DataStax Enterprise (DSE) Search indexes a CQL map column using a Solr dynamic field. Dynamic fields apply the field definition using a wildcard match on the name. In the search index schema, DSE sets the dynamic field name to the CQL column name with an asterisk appended. DSE parses the data from a map using the key name and Solr will index only the keys that have the column name as the prefix. Keys that do not have the column name as a prefix are ignored.

For example, when creating a search index with the default settings on the cycling birthday_list table, the blist_map column definition is:

```
<dynamicField indexed="true" multiValued="false" name="blist_*" type="StrField"/>
```

When DSE builds the index from the CQL rows, the key name is used (not the column name). Therefore, all keys that have the blist_ as the prefix in the example are indexed and the rest are ignored. Only blist_age and blist_nation are indexed when the following data is inserted:

```
INSERT INTO cycling.birthday_list (cyclist_name, blist_)
VALUES (
  'Allan DAVIS', { 'blist_age':'35', 'bday':'27/07/1980', 'blist_nation':'AUSTRALIA' })
```

All key-value pairs in CQL maps have the same data type. The map in the example above sets all values to text (blist_map<text,text>). Because DSE Search loads the data by mapping the key name to the Solr dynamic field name, you can customize field type for each key.

**Prerequisites:**

This section walks you through the process of customizing the search index for data that has the same three map keys in every record, blist_age, bday (birth date), and blist_nation where only blist_age and blist_nation are indexed.

Set up the following keyspace and table to use this example:

- Create the cycling keyspace
- Add the birthday_list table and data
Managing database resources

1. Create an index that excludes the `blist_map` column:
   
   ```
   CREATE SEARCH INDEX IF NOT EXISTS
   ON cycling.birthday_list
   WITH COLUMNS blist_ { excluded:true };
   ```

2. View the active schema:
   
   ```
   DESC ACTIVE SEARCH INDEX SCHEMA ON cycling.birthday_list;
   ```

   DSE sets CQL text to Solr StrField type.

   ```
   <?xml version="1.0" encoding="UTF-8" standalone="no"?>
   <schema name="autoSolrSchema" version="1.5">
   <types>
      <fieldType class="org.apache.solr.schema.StrField" name="StrField"/>
   </types>
   <fields>
      <field indexed="true" multiValued="false" name="cyclist_name" type="StrField"/>
   </fields>
   <uniqueKey>cyclist_name</uniqueKey>
   </schema>
   ```

   To set `blist_age` to an integer, the type definition is also required.

3. Define the `blist_age` type and configure a field definition:
   
   ```
   // Add type
   ALTER SEARCH INDEX SCHEMA
   ON cycling.birthday_list
   ADD types.fieldType[ @class='org.apache.solr.schema.TrieIntField',
      @name='TrieIntField' ];
   
   // Control the data types of map fields by name
   ALTER SEARCH INDEX SCHEMA
   ON cycling.birthday_list
   ADD fields.field[ @indexed='true', @multiValued='false', @name='blist_age',
      @type='TrieIntField' ];
   ```

4. Define the `blist_nation` field as a string type, which has a corresponding type definition of @type='StrField':
   
   ```
   ALTER SEARCH INDEX SCHEMA
   ON cycling.birthday_list
   ADD fields.field[ @name='blist_nation', @indexed='true', @multiValued='false',
      @type='StrField' ];
   ```

5. View the pending changes to the schema to ensure that the syntax is correct:
   
   ```
   DESC PENDING SEARCH INDEX SCHEMA ON cycling.birthday_list;
   ```

   ```
   <?xml version="1.0" encoding="UTF-8" standalone="no"?>
   <schema name="autoSolrSchema" version="1.5">
   <types>
      <fieldType class="org.apache.solr.schema.StrField" name="StrField"/>
      <fieldType class="org.apache.solr.schema.TrieIntField" name="TrieIntField"/>
   </types>
   <fields>
   </fields>
   ```
Managing database resources

6. Reload the index configuration and schema to push the changes live:

   `RELOAD SEARCH INDEX ON cycling.birthday_list;`

7. Rebuild the index whenever fields are added:

   `REBUILD SEARCH INDEX ON cycling.birthday_list;`

8. Use the map fields to filter queries:

   • Limit by age 23:

     ```
     SELECT *
     FROM cycling.birthday_list
     WHERE solr_query = 'blist_age:23';
     ```

     | cyclist_name | blist_ | solr_query |
     |--------------|--------|------------|
     | Claudio HEINEN | {'bday': '27/07/1992', 'blist_age': '23', 'blist_nation': 'GERMANY'} | null |
     | Laurence BOURQUE | {'bday': '27/07/1992', 'blist_age': '23', 'nation': 'CANADA'} | null |

     (2 rows)

   • Limit by nation GERMANY, which is case-sensitive because the type is a string:

     ```
     SELECT *
     FROM cycling.birthday_list
     WHERE solr_query = 'blist_nation:GERMANY';
     ```

     | cyclist_name | blist_ | solr_query |
     |--------------|--------|------------|
     | Claudio HEINEN | {'bday': '27/07/1992', 'blist_age': '23', 'blist_nation': 'GERMANY'} | null |
Dropping columns from the index

Remove a CQL column using the `ALTER SEARCH INDEX SCHEMA` field shortcut. The field is removed based on the name that is defined in the schema.

1. Remove a CQL column from the index:
   - Remove a regular column:
     ```
     ALTER SEARCH INDEX SCHEMA
     ON wiki.solr
     DROP field intfield;
     
     ALTER TABLE wiki.solr
     DROP intfield int;
     ```
   - Remove a tuple column from the index:
     When dropping the base field name, drop the TupleField and all the child fields:
     ```
     ALTER SEARCH INDEX SCHEMA
     ON wiki.solr
     DROP field fieldname;
     ```
     To drop individual child fields:
     ```
     ALTER SEARCH INDEX SCHEMA
     ON wiki.solr
     DROP field "fieldname.field1";
     ```
   - Remove a field that contains an asterisk in the name:
     To remove a field that contains an asterisk in the name, enclose the field name in double quotation marks:
     ```
     ALTER SEARCH INDEX SCHEMA
     ON wiki.solr
     DROP field "fieldname";
     ```
     Alternatively, use the long form of the command:
     ```
     ALTER SEARCH INDEX SCHEMA
     ON wiki.solr
     DROP fields.field[@name='fieldname'];
     ```
   - Remove a field that contains an incorrect XML path:
     For example, assume a field was added with an incorrect XML path that is not nested inside the `<fields>` tag:
     ```
     ALTER SEARCH INDEX SCHEMA
     ON wiki.solr
     ADD field[@name='fieldname', @type='StrField'];
     ```
     Example XML fragment showing that the new field is outside of the `<fields>` tag:
     ```
     <fields>
       ...
     </fields>
     ```
If you use the short form of the `ALTER SEARCH INDEX SCHEMA` command, the command returns an error because it is assumed that the field is nested inside of the `<fields>` tag in the XML:

```sql
ALTER SEARCH INDEX SCHEMA
ON wiki.solr
DROP field fieldname;
```

InvalidRequest: Error from server: code=2200 [Invalid query] message=
"The search index schema could not be updated because: 
Cannot drop resource element fields.field[@name='fieldname'] 
because it doesn't exist"

To drop the field, use the long form of the `ALTER SEARCH INDEX SCHEMA` command:

```sql
ALTER SEARCH INDEX SCHEMA
ON wiki.solr
DROP field[@name='fieldname'];
```

• Remove a dynamic field:

To remove a dynamic field, use the long form of the `ALTER SEARCH INDEX SCHEMA` command because the field name always contains an asterisk:

```sql
ALTER SEARCH INDEX SCHEMA
ON wiki.solr
DROP fields.dynamicField[@name='fieldname*'];
```

• Remove a copy field:

To remove a copy field, use the long form of the `ALTER SEARCH INDEX SCHEMA` command because the source and destination fields must be specified:

```sql
ALTER SEARCH INDEX SCHEMA
ON wiki.solr
DROP copyField[@source='sourcefieldname', @dest='destfieldname'];
```

• Remove associated copy fields and resolve dependencies:

If you attempt to remove a field that has an associated copy field, the `ALTER SEARCH INDEX SCHEMA` command returns an invalid request error:

```sql
ALTER SEARCH INDEX SCHEMA
ON wiki.solr
DROP fields.dynamicField[@name='copyfieldname*'];
```

InvalidRequest: Error from server: code=2200 [Invalid query] message=
"The search index schema is not valid because: 
Can't load schema schema.xml: copyField dest :'copyfieldname'' 
is not an explicit field and doesn't match a dynamicField."

To remove the field, remove all of the underlying child elements first and then remove the field.
Managing database resources

2. Verify the pending changes:

   DESCRIBE PENDING SEARCH INDEX SCHEMA ON [keyspace_name.]table_name;

3. Activate the changes:

   RELOAD SEARCH INDEX ON [keyspace_name.]table_name;

   Copies the pending schema over the active schema. New transactions, such as data inserted into the table, are processed using the active schema. The existing data is not effected by a schema change.

4. Rebuild the index:

   REBUILD SEARCH INDEX ON [keyspace_name.]table_name;

   The REBUILD SEARCH INDEX regenerates the index using existing data. Rebuilding is required when changing the way that data is indexed, such as changing the type of field or if a field is added to the index.

Indexing a column for different analysis

DSE Search supports indexing a CQL table column for different types of analysis using the Solr `copyField` directive.

For a complete explanation, see the Solr Reference Guide Copying fields.

When specified during search index creation, DSE automatically defines a new index string field and sets up the data copy. The new field is not stored in the database or returned in query results.

Copying from/to the same dynamic field and setting the maximum number of characters (maxChars) in the `copyField` definition are unsupported.

The following example uses copy fields to copy various CQL columns, such as a twitter name and email, to a `multiValued` field. You can then query the `multiValued` field using a term to search for all columns in a single query.

The DSE per-segment filter cache is moved off-heap by using native memory to reduce on-heap memory consumption and garbage collection overhead. The off-heap filter cache is enabled by default. To disable, pass the `offheap JVM system property at startup time: -Dsolr.offheap.enable`

1. Create a keyspace using the replication strategy and replication factor that makes sense for your environment. The following example is for a single node test cluster:

   CREATE KEYSPACE user_info
   WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : 1 };

2. Create a table:

   CREATE TABLE user_info.users (
   id text PRIMARY KEY,
   name text,
   email text,
   skype text,
   irc text,
   twitter text
   ) ;

3. Insert some data:

   INSERT INTO user_info.users (id, name, email, skype, irc, twitter) VALUES
INSERT INTO user_info.users (id, name, email, skype, irc, twitter) VALUES
('user1', 'john smith', 'jsmith@abc.com', 'johnsmith', 'smitty', '@johnsmith');

INSERT INTO user_info.users (id, name, email, skype, irc, twitter) VALUES
('user2', 'elizabeth doe', 'lizzy@swbell.net', 'roadwarriorliz', 'elizdoe', '@edoe576');

INSERT INTO user_info.users (id, name, email, skype, irc, twitter) VALUES
('user3', 'dan graham', 'etnaboy1@aol.com', 'danielgra', 'dgraham', '@dannyboy');

INSERT INTO user_info.users (id, name, email, skype, irc, twitter) VALUES
('user4', 'john smith', 'jonsmit@fyc.com', 'johnsmith', 'jsmith345', '@johnrsmith');

INSERT INTO user_info.users (id, name, email, skype, irc, twitter) VALUES
('user5', 'john smith', 'jds@adeck.net', 'jdsmith', 'jdansmith', '@smithjd999');

INSERT INTO user_info.users (id, name, email, skype, irc, twitter) VALUES
('user6', 'dan graham', 'hacker@legalb.com', 'dangrah', 'dgraham', '@graham222');

4. Create a search index on the table:

```
CREATE SEARCH INDEX ON user_info.users;
```

5. Create a field that is only in the index that will contain all the data:

```
ALTER SEARCH INDEX SCHEMA ON user_info.users
ADD fields.field[ @name='all',
                   @type='StrField',
                   @multiValued='true'];
```

6. Use `copyField` to copy the data from all the CQL columns into the new `all` field of the index:

```
ALTER SEARCH INDEX SCHEMA ON user_info.users
  ADD copyField[@source='id', @dest='all'];

ALTER SEARCH INDEX SCHEMA ON user_info.users
  ADD copyField[@source='name', @dest='all'];

ALTER SEARCH INDEX SCHEMA ON user_info.users
  ADD copyField[@source='email', @dest='all'];

ALTER SEARCH INDEX SCHEMA ON user_info.users
  ADD copyField[@source='skype', @dest='all'];

ALTER SEARCH INDEX SCHEMA ON user_info.users
  ADD copyField[@source='irc', @dest='all'];

ALTER SEARCH INDEX SCHEMA ON user_info.users
  ADD copyField[@source='twitter', @dest='all'];
```

7. To allow faceting on the name column, set `docValues` to `true`:

```
ALTER SEARCH INDEX SCHEMA ON user_info.users
```

(CQL for DataStax Enterprise 6.7 Latest DSE version Latest 6.7 patch: 6.7.7)
Managing database resources

8. Reload the schema to make the pending changes active:

```cql
RELOAD SEARCH INDEX ON user_info.users;
```

9. Rebuild the index to apply the new schema to the existing data:

```cql
REBUILD SEARCH INDEX ON user_info.users;
```

10. Filter the query using the index to return all records that contain smitty in any of the columns.

```cql
SELECT * FROM user_info.users WHERE solr_query = 'all:smitty';
```

The output is:

```
| id     | email       | irc | name       | skype     | solr_query | twitter |
|--------|-------------+-----+------------+-----------+------------+---------|
| user1  | jsmith@abc.com | smitty | john smith | johnsmith | null       | @johnsmith|
```

(1 rows)

11. Get a count of unique names (skip nulls):

```cql
SELECT name FROM user_info.users
WHERE solr_query='{"q":"*","facet":{"field":"name","mincount":"1"}}';
```

At the bottom of the output, the facet results appear: 3 instances of john smith, 2 instances of dan graham, and 1 instance of elizabeth doe.

```
facet_fields

| {"name":{"john smith":3,"dan graham":2,"elizabeth doe":1}} |
```

(1 rows)

### Configuring search index joins

DataStax Enterprise supports solr_query joins on the partition key field (_partitionKey). By default, the solr_query join functionality is enabled and DSE indexes the partitioning columns in this additional field. This field, _partitionKey, increases search index size. Disabling joins can decrease the amount of disk space the search indexes uses.

#### Join settings in the schema

`DESCRIBE ACTIVE SEARCH INDEX SCHEMA` displays the schema settings of a search index. DSE hides the definition of the _partitionKey when joins are enabled.

If the schema contains a field named _partitionKey, support for joins is:

- **Enabled**: attributes `docValues` and `indexed` are set to true. For example:

```xml
<field name="_partitionKey" docValues="true" indexed="true" stored="false" type="StrField"/>
```
Managing database resources

- **Disabled**: attributes `docValues` and `indexed` are set to false. For example:

```xml
<field docValues="false" indexed="false" multiValued="false" name="_partitionKey"
omitNorms="true" stored="false" type="StrField"/>
```

If the schema contains no field definition for `_partitionKey`, then joins are enabled.

**Prerequisite**

This section uses the Term and phrase searches using the wikipedia demo.

**Disable joins**

Disable join on a search index by setting the `_partitionKey` field attributes `indexed` and `docValues` to false in the schema.

1. **Verify if schema has the field `_partitionKey` and `fieldType` StrField definitions.**

   ```sql
   DESCRIBE ACTIVE SEARCH INDEX SCHEMA ON wiki.solr;
   ```

   The example search index has joins enabled with no `_partitionKey` definition:

   ```xml
   <?xml version="1.0" encoding="UTF-8" standalone="no"?>
   <schema name="autoSolrSchema" version="1.5">
   <types>
   <fieldType class="org.apache.solr.schema.TextField" name="TextField">
   <analyzer>
   <tokenizer class="solr.StandardTokenizerFactory"/>
   <filter class="solr.LowerCaseFilterFactory"/>
   </analyzer>
   </fieldType>
   <fieldType class="org.apache.solr.schema.TrieDateField" name="TrieDateField"/>
   <fieldType class="org.apache.solr.schema.StrField" name="StrField"/>
   </types>
   <fields>
   <field indexed="true" multiValued="false" name="body" stored="true" type="TextField"/>
   <field docValues="true" indexed="true" multiValued="false" name="real_date" stored="true" type="TrieDateField"/>
   <field indexed="true" multiValued="false" name="title" stored="true" type="TextField"/>
   <field indexed="true" multiValued="false" name="id" stored="true" type="StrField"/>
   <field indexed="true" multiValued="false" name="date" stored="true" type="TextField"/>
   </fields>
   <uniqueKey>id</uniqueKey>
   </schema>
   ```

2. **If required, add the string type definition:**

   ```sql
   ALTER SEARCH INDEX SCHEMA ON wiki.solr
   ADD types.fieldType[@class='org.apache.solr.schema.StrField', @name='StrField'];
   ```

   The definition is added to the pending schema and is not immediately applied.

3. **Define the partition key field:**

   - If the search index already has the partition key field, change the `indexed` and `docValues` to false:

   ```sql
   ALTER SEARCH INDEX SCHEMA ON wiki.solr
   SET field[@name='_partitionKey']@docValues='false';
   ```
Managing database resources

ALTER SEARCH INDEX SCHEMA ON wiki.solr
SET field[@name='_partitionKey']@indexed='false';

- If the schema does not have a _partitionKey definition, add one to override the default settings:

ALTER SEARCH INDEX SCHEMA ON wiki.solr
ADD fields.field[@name='_partitionKey', @type='StrField', @docValues='false', @indexed='false'];

The type definition StrField is also required.

4. Verify that the schema definition was correctly modified:

DESCRIBE PENDING SEARCH INDEX SCHEMA ON wiki.solr;

For example, a simple table with three fields and a single partition key:

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<schema name="autoSolrSchema" version="1.5">
  <types>
    <fieldType class="org.apache.solr.schema.TextField" name="TextField">
      <analyzer>
        <tokenizer class="solr.StandardTokenizerFactory"/>
        <filter class="solr.LowerCaseFilterFactory"/>
      </analyzer>
    </fieldType>
    <fieldType class="org.apache.solr.schema.TrieDateField" name="TrieDateField"/>
    <fieldType class="org.apache.solr.schema.StrField" name="StrField"/>
  </types>
  <fields>
    <field indexed="true" multiValued="false" name="body" stored="true" type="TextField"/>
    <field docValues="true" indexed="true" multiValued="false" name="real_date" stored="true" type="TrieDateField"/>
    <field indexed="true" multiValued="false" name="title" stored="true" type="TextField"/>
    <field indexed="true" multiValued="false" name="id" stored="true" type="StrField"/>
    <field indexed="true" multiValued="false" name="date" stored="true" type="TextField"/>
    <field docValues="false" indexed="false" name="_partitionKey" type="StrField"/>
  </fields>
  <uniqueKey>id</uniqueKey>
</schema>

5. Reload the schema to make it active:

RELOAD SEARCH INDEX ON wiki.solr;

6. Optional, rebuild the search index:

REBUILD SEARCH INDEX ON wiki.solr;

Rebuilding from CQL regenerates the index from the existing data on all search nodes, which use significant resources and is not required when disabling joins. When no rebuild command is executed after a schema change, new data in the field is not be duplicated and indexed. Use dsetool rebuild_indexes to regenerate the index on a node-by-node basis.
Enable joins

To enable join on a search index that previously had join disabled, set the `_partitionKey`, `docValues`, and `indexed` attributes to `true`, reload the schema, and rebuild the index.

Rebuilding the search index on a large dataset might take longer than the default timeout for `cqlsh`. Before launching `cqlsh`, you can override the timeout. See Adjusting timeout for index management.

1. Start `cqlsh` on a node that is running DSE Search.

2. Set the `docValues` and `indexed` attributes to `true`:

   ```
   ALTER SEARCH INDEX SCHEMA ON wiki.solr
   SET field[@name='_partitionKey']@docValues='true';
   ALTER SEARCH INDEX SCHEMA ON wiki.solr
   SET field[@name='_partitionKey']@indexed='true';
   ```

3. Verify that the schema definition was correctly modified:

   ```
   DESCRIBE PENDING SEARCH INDEX SCHEMA ON wiki.solr;
   ```

For example, a simple table with three fields and a single partition key:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<schema name="autoSolrSchema" version="1.5">
    <types>
        <fieldType class="org.apache.solr.schema.TextField" name="TextField">
            <analyzer>
                <tokenizer class="solr.StandardTokenizerFactory"/>
                <filter class="solr.LowerCaseFilterFactory"/>
            </analyzer>
        </fieldType>
        <fieldType class="org.apache.solr.schema.TrieDateField" name="TrieDateField"/>
        <fieldType class="org.apache.solr.schema.StrField" name="StrField"/>
    </types>
    <fields>
        <field indexed="true" multiValued="false" name="body" stored="true" type="TextField"/>
        <field docValues="true" indexed="true" multiValued="false" name="real_date" stored="true" type="TrieDateField"/>
        <field indexed="true" multiValued="false" name="title" stored="true" type="TextField"/>
        <field indexed="true" multiValued="false" name="id" stored="true" type="StrField"/>
        <field indexed="true" multiValued="false" name="date" stored="true" type="TextField"/>
        <field docValues="false" indexed="false" name="_partitionKey" type="StrField"/>
    </fields>
    <uniqueKey>id</uniqueKey>
</schema>
```
Managing database resources

4. Reload the schema to make it active:

```
RELOAD SEARCH INDEX ON wiki.solr;
```

5. Rebuild the search index:

```
REBUILD SEARCH INDEX ON wiki.solr;
```

Reloading the search index

After you modify the search index schema, config, or upload custom resource files (like a synonym file), reload the search index to make the pending search index active.

Changing search index config

To create and make changes to the search index config, follow these basic steps:

1. Create a search index. For example:

```
CREATE SEARCH INDEX ON demo.health_data;
```

2. Alter the search index. For example:

```
ALTER SEARCH INDEX CONFIG ON demo.health_data SET autoCommitTime = 30000;
```

3. Optionally view the XML of the pending search index. For example:

```
DESCRIBE PENDING SEARCH INDEX CONFIG on demo.health_data;
```

4. Make the pending changes active. For example:

```
RELOAD SEARCH INDEX ON demo.health_data;
```

The CQL command `RELOAD SEARCH INDEX` rebuilds the search index.

For operations, you can optionally reload a search index (also called a search core) on a single node using `dsetool reload_core`.

If one or more nodes fail to reload the core in distributed operations, an error message indicates a list of the failing node or nodes. Issue the reload again only on those failing nodes using distributed=false.

Reindexing in place

Setting `reindex=true` and `deleteAll=false` reindexes data and keeps the existing index. During the uploading process, user searches yield inaccurate results. To perform an in-place reindex, use this syntax:

```
$ dsetool reload_core keyspace_name.table_name reindex=true deleteAll=false
```

Reindexing in full

Setting `reindex=true` and `deleteAll=true` deletes the index and reindexes the dataset. User searches initially return no or partial documents as the search cores reload and data is reindexed.

```
$ dsetool reload_core keyspace_name.table_name reindex=true deleteAll=true
```

During reindexing, a series of criteria routes sub-queries to the nodes most capable of handling them. See Shard routing for distributed queries.
Managing database resources

Removing a search index

Drop a search index from a table and delete all related data using the DROP SEARCH INDEX command.

The CQL syntax:

```
DROP SEARCH INDEX on [keyspace_name.]table_name;
```

Keyspace and table names are case-sensitive. Enclose names that contain uppercase in double quotation marks.

Updating the index after data expires (TTL)

Time-To-Live (TTL) set on a CQL field also applies to the indexed values. The DSE Search engine purges expired and deleted data by rebuilding the index as defined by the `ttl_index_rebuild_options` in the `dse.yaml` file. The default rebuild interval is 300 seconds (5 minutes).

Setting data expiration in CQL

1. Using CQL **INSERT** or **UPDATE**, set the TTL property.
   
   For example, insert a row with a life of 60 seconds into the **health_data demo**:

   ```
   INSERT INTO demo.health_data (id, age, gender)
   VALUES (9999,88,'female') USING TTL 60;
   ```

2. Force the index to update with the new data:

   ```
   COMMIT SEARCH INDEX ON demo.health_data;
   ```

3. After 60 seconds, the row is removed from the CQL table and the search index.

Inserting, updating, and deleting data

For DSE Search, inserting and updating data uses the same CQL statements like any update to the database. Updates to a CQL-based search index replaces the entire row. You cannot replace only a field in a CQL table.

To update a CQL-based search index:

1. Building on the **Querying CQL collections** example, insert data into mykeyspace.mysolr and the search index.

   ```
   INSERT INTO mykeyspace.mysolr ('id', 'quotes', 'name', 'title') VALUES ('130', 'Life is a beach', 'unknown', 'Life');
   ```

   When using CQL to update a field, DSE Search implicitly updates individual fields in the Solr document. The reindexing of data occurs automatically.
Chapter 6. Inserting and updating data

Data can be inserted into tables using the INSERT command. Starting with DataStax Enterprise 5.0, JSON data can be inserted.

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').`
Inserting and updating data

The values returned by `minTimeuuid` and `maxTimeuuid` functions are not true UIDs 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.

DataStax Enterprise 5.0 and later support some additional timeuuid and timestamp functions to manipulate dates. The functions can be used in INSERT, UPDATE, and SELECT statements.

- `toDate(timeuuid)`
  Converts timeuuid to date in YYYY-MM-DD format.

- `toTimestamp(timeuuid)`
  Converts timeuuid to timestamp format.

- `toUnixTimestamp(timeuuid)`
  Converts timeuuid to UNIX timestamp format.

- `toDate(timestamp)`
  Converts timestamp to date in YYYY-MM-DD format.

- `toUnixTimestamp(timestamp)`
  Converts timestamp to UNIX timestamp format.

- `toTimestamp(date)`
  Converts date to timestamp format.

- `toUnixTimestamp(date)`
  Converts date to UNIX timestamp format.

An example of the new functions creates a table and inserts various time-related values:

```
CREATE TABLE sample_times (a int, b timestamp, c timeuuid, d bigint, PRIMARY KEY (a,b,c,d));
INSERT INTO sample_times (a,b,c,d) VALUES (1, toUnixTimestamp(now()), 50554d6e-29bb-11e5-b345-feff819cdc9f, toTimestamp(now()));
```

Select data and convert it to a new format:

```
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.

- Insert data into the **set**, enclosing values in curly brackets.
Inserting and updating data

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 biecht' } );
```

- **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;
```

- **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.
Inserting and updating data

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

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

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

```
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.

```
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.

```
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.

• 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.

```
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.

```
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.

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

CQL for DataStax Enterprise 6.7 Latest DSE version Latest 6.7 patch: 6.7.7
Inserting and updating data

'VOS',
'Marianne',

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 - {'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
Inserting and updating data

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

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

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

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

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

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 Enterprise 5.1 and later, 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 `UPDATE` command. The desired key-value pair are defined in the command. In order to update, the UDT must be defined in the `CREATE TABLE` command as an unfrozen data type.

```sql
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-6b5d3a79cbeb, '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-6b5d3a79cbeb;
```

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

```sql
SELECT * FROM cycling.cyclist_stats WHERE id = 220844bf-4860-49d6-9a4b-6b5d3a79cbeb;
```

<table>
<thead>
<tr>
<th>id</th>
<th>basics</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>220844bf-4860-49d6-9a4b-6b5d3a79cbeb</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 `Setting up 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
WHERE id = ed584e99-80f7-4b13-9a90-9dc5571e6821;
```

<table>
<thead>
<tr>
<th>first_race</th>
<th>last_race</th>
<th>birthday</th>
</tr>
</thead>
<tbody>
<tr>
<td>2006-03-15</td>
<td>2019-11-05</td>
<td>null</td>
</tr>
</tbody>
</table>

(1 rows)

Time column

Write values into a time column.

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

String format
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 subseconds; 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' },
    { race_time : '08:00:00' } ]
}"
```
Inserting and updating data

• To UPDATE a time field:

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

**Timestamp column**

Upsert 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.  
  # (Optional) **.fff**: up to three-digit subseconds; 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 Setting up 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.

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

# Full timestamp

```sql
INSERT INTO cycling.comments (id, created_at) VALUES (e7ae5cf3-d358-4d99-b900-85902fda9bb0, toTimeStamptoNow());
```

- Using string format:

  # Date with time and no timezone sets the timezone to UTC.

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

  # Full timestamp using UTC.

  ```sql
  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:

```
NyNmoNwNhNsNmsNusNns
```

Where:

- \( N \): Number
- \( y \): Years
- \( mo \): Months
Inserting and updating data

- **w**: Weeks
- **d**: Days
- **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:

```
PNYMNDT12H30M5S
```

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:

```
P6W
```

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** duration values.

```cql
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** duration values.

```cql
UPDATE cycling.race_times
SET finish_time = '2h5m22s14ms'
WHERE race_name = '17th Santos Tour Down Under'
  AND cyclist_name = 'Rohan DENNIS'
  AND race_date = '2017-04-14';
```

- **SELECT** duration values.

```cql
SELECT *
FROM cycling.race_times
WHERE race_name = '17th Santos Tour Down Under'
  AND cyclist_name = 'Rohan DENNIS'
  AND race_date = '2017-04-14';
```

Output:

```
<table>
<thead>
<tr>
<th>race_date</th>
<th>race_name</th>
<th>cyclist_name</th>
<th>finish_time</th>
<th>race_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017-04-14</td>
<td>17th Santos Tour Down Under</td>
<td>Rohan DENNIS</td>
<td>2h5m22s14ms</td>
<td>19:15:18.0000000000</td>
</tr>
</tbody>
</table>
```
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**.

```
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.

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

Inserting JSON formatted values

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

Use the following syntax:

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

- 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 `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"
}';
```

```
category | points | id | lastname
----------|--------|----|---------
Sprint    | 700    | 829aa84a-4bba-411f-a4fb-38167a987cda | null
GC        | 1269   | 228844bf-4860-49d6-9a4b-6b5d3a79cbfb | TIRALONGO
GC        | 780    | 829aa84a-4bba-411f-a4fb-38167a987cda | SUTHERLAND
```

**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
```
Inserting and updating data

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

```bash
COPY cycling.cyclist_category 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`.

```cql
DELETE lastname FROM cycling.cyclist_name WHERE id = c?fceba0-c141-4207-9494-a29f9809de6f;
```

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

```cql
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 Expiring data with Time-To-Live.

Expiring data with time-to-live

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

- TTL precision is one second, which is calculated by the coordinator node. When setting a TTL, ensure that all nodes in the cluster have synchronized clocks.

- A very short TTL is not useful.

- Expiring data uses 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 requests with expiration timestamps later than the maximum date are rejected or inserted. See `Dcassandra.expiration_date_overflow_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 TTL, the entire row is deleted.
Inserting and updating data

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

Expiring data with TTL example

Both the INSERT and UPDATE commands support setting a time for data in a column to expire, including collections and user-defined types. Use CQL to set the expiration time-to-live (TTL).

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

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

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

```sql
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.

```sql
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 or update data into a table for a specific duration of time. To determine the current time-to-live for a record, use the TTL function.

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.
Inserting and updating data

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

```sql
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.

```sql
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.

```sql
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';
```

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 side as a single mutation for improved performance, provided the number of operations do not exceed the **maximum size of a single operation** or cause the query to time out.
Multiple partition batch operations often have performance issues and should only be used if atomicity must be ensured.
A cassandra.yaml option lets you choose the `batchlog_endpoint_strategy`.

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 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, as shown in the following examples. The examples use the table `cyclist_expenses`:

```
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.
Inserting and updating data

```cql
'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:

```cql
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.

```
[applied]
True
```

The resulting table will only have one record so far.

<table>
<thead>
<tr>
<th>cyclist_name</th>
<th>expense_id</th>
<th>balance</th>
<th>amount</th>
<th>description</th>
<th>paid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vera ADRIAN</td>
<td>1</td>
<td>0</td>
<td>7.95</td>
<td>Breakfast</td>
<td>False</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>cyclist_name</th>
<th>expense_id</th>
<th>balance</th>
<th>amount</th>
<th>description</th>
<th>paid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vera ADRIAN</td>
<td>1</td>
<td>-7.95</td>
<td>7.95</td>
<td>Breakfast</td>
<td>False</td>
</tr>
</tbody>
</table>

- 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
```
Inserting and updating data

```
INSERT INTO cycling.cyclist_expenses (cyclist_name, expense_id, amount, description, paid)
VALUES ('Vera ADRIAN', 2, 13.44, 'Lunch', true);

INSERT INTO cycling.cyclist_expenses (cyclist_name, expense_id, amount, description, paid)
VALUES ('Vera ADRIAN', 3, 25.00, 'Dinner', false);

UPDATE cycling.cyclist_expenses
SET balance = -32.95
WHERE cyclist_name = 'Vera ADRIAN'
IF balance = -7.95;
APPLY BATCH;
```

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

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

```
BEGIN BATCH

UPDATE cycling.cyclist_expenses
SET balance = 0
WHERE cyclist_name = 'Vera ADRIAN'
IF balance = -32.95;

UPDATE cycling.cyclist_expenses
SET paid = true
WHERE cyclist_name = 'Vera ADRIAN'
AND expense_id = 1 IF paid = false;

UPDATE cycling.cyclist_expenses
SET paid = true
WHERE cyclist_name = 'Vera ADRIAN'
AND expense_id = 3
IF paid = false;

APPLY BATCH;
```

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

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

**Multiple partition logged batch**
Inserting and updating data

- Another example is using **BATCH** to perform a multiple partition insert that 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:

  ```sql
  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 batch operation is updating usernames and passwords.

Misuse of **BATCH** statement

Misused **BATCH** statements can cause many problems in DataStax Enterprise. 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 significant 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**.

  ```sql
  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 (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 7. 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 prirmary_key_conditions| AND clustering_columns_conditions]] | PRIMARY KEY LIMIT
```

Setting consistency levels

In a distributed system such as DataStax Enterprise, 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 set up 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 using tracing

Tracing records all activity related to a request. These steps use tracing to show queries on a keyspace with a replication factor of 3 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](https://support.datastax.com/hc/en-us/articles/360034220882) 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 row)
   ```
<table>
<thead>
<tr>
<th>source</th>
<th>timestamp</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute CQL query</td>
<td>2017-02-28 21:40:59.918000</td>
<td>10.200.176.228</td>
</tr>
<tr>
<td>Parsing SELECT * FROM cycling_alt.tester</td>
<td>2017-02-28 21:40:59.918000</td>
<td>10.200.176.228</td>
</tr>
<tr>
<td>Executing single-partition query on tester [ReadStage-3]</td>
<td>2017-02-28 21:40:59.923000</td>
<td>10.200.176.228</td>
</tr>
<tr>
<td>Acquiring sstable references [ReadStage-3]</td>
<td>2017-02-28 21:40:59.923000</td>
<td>10.200.176.228</td>
</tr>
<tr>
<td>Merging memtable contents [ReadStage-3]</td>
<td>2017-02-28 21:40:59.923000</td>
<td>10.200.176.228</td>
</tr>
<tr>
<td>Read 1 live and 0 tombstone cells [ReadStage-3]</td>
<td>2017-02-28 21:40:59.923000</td>
<td>10.200.176.228</td>
</tr>
<tr>
<td>REQUEST_RESPONSE message received from /10.200.176.229</td>
<td>2017-02-28 21:40:59.924000</td>
<td>10.200.176.228</td>
</tr>
<tr>
<td>READ message received from /10.200.176.228</td>
<td>2017-02-28 21:40:59.968000</td>
<td>10.200.176.229</td>
</tr>
<tr>
<td>Executing single-partition query on tester [ReadStage-1]</td>
<td>2017-02-28 21:40:59.969000</td>
<td>10.200.176.229</td>
</tr>
<tr>
<td>Acquiring sstable references [ReadStage-1]</td>
<td>2017-02-28 21:40:59.969000</td>
<td>10.200.176.229</td>
</tr>
<tr>
<td>Merging memtable contents [ReadStage-1]</td>
<td>2017-02-28 21:40:59.969000</td>
<td>10.200.176.229</td>
</tr>
<tr>
<td>Read 1 live and 0 tombstone cells [ReadStage-1]</td>
<td>2017-02-28 21:40:59.969000</td>
<td>10.200.176.229</td>
</tr>
</tbody>
</table>
Querying data

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.

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

```
id | col1 | col2
----+------|------
0  | 0    | 0
(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>Execute CQL3 query</td>
<td>2017-02-28 21:42:22.394000</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>10.200.176.23</td>
<td>10.200.176.23</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Executing single-partition query on tester [ReadStage-2]</td>
<td>2017-02-28 21:42:22.395000</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>10.200.176.23</td>
<td>10.200.176.23</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Acquiring sstable references [ReadStage-2]</td>
<td>2017-02-28 21:42:22.395000</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>10.200.176.23</td>
<td>10.200.176.23</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Merging memtable contents [ReadStage-2]</td>
<td>2017-02-28 21:42:22.395000</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>10.200.176.23</td>
<td>10.200.176.23</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Read 1 live and 0 tombstone cells [ReadStage-2]</td>
<td>2017-02-28 21:42:22.395000</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>10.200.176.23</td>
<td>10.200.176.23</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>10.200.176.23</td>
<td>10.200.176.23</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>2017-02-28 21:42:22.395000</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>10.200.176.23</td>
<td>10.200.176.23</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Parsing SELECT * FROM cycling_alt.tester WHERE id = 0; [Native-Transport-Requests-1]</td>
<td>2017-02-28 21:42:22.395000</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>10.200.176.23</td>
<td>10.200.176.23</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>10.200.176.23</td>
<td>10.200.176.23</td>
<td>127.0.0.1</td>
</tr>
</tbody>
</table>
Querying data

Acquiring sstable references [ReadStage-2] | 2017-02-28 21:42:22.395000 | 10.200.176.228 | 1288 | 127.0.0.1


Read 1 live and 0 tombstone cells [ReadStage-2] | 2017-02-28 21:42:22.396000 | 10.200.176.228 | 1584 | 127.0.0.1


Sending READ message to /10.200.176.229 message size 135 bytes


Sending READ message to /10.200.176.23 message size 135 bytes


REQUEST_RESPONSE message received from /10.200.176.229


REQUEST_RESPONSE message received from /10.200.176.23


READ message received from /10.200.176.228


Acquiring sstable references [ReadStage-1] | 2017-02-28 21:42:22.444000 | 10.200.176.229 | 418 | 127.0.0.1


Read 1 live and 0 tombstone cells [ReadStage-1] | 2017-02-28 21:42:22.444000 | 10.200.176.229 | 854 | 127.0.0.1

Read 1 live and 0 tombstone cells [ReadStage-1] | 2017-02-28 21:42:22.444000 | 10.200.176.229 | 975 | 127.0.0.1


Sending REQUEST_RESPONSE message to /10.200.176.228 message size 96 bytes

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

```
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 rows)

Tracing session: ea9a9aa0-fdfe-11e6-8b40-23a5e4e49022
Sending READ message to /10.200.176.23 message size 135 bytes
10.200.176.228 | 1941 | 127.0.0.1

Executing
10.200.176.228 | 4066 | 127.0.0.1

Acquiring sstable references [ReadStage-3] | 2017-02-28 21:43:18.094000 |
10.200.176.228 | 4213 | 127.0.0.1

10.200.176.228 | 4323 | 127.0.0.1

Read 1 live and 0 tombstone cells [ReadStage-3] | 2017-02-28 21:43:18.095000 |
10.200.176.228 | 4595 | 127.0.0.1

REQUEST_RESPONSE message received from /10.200.176.23

10.200.176.228 | 8493 | 127.0.0.1

REQUEST_RESPONSE message received from /10.200.176.229

10.200.176.228 | 9062 | 127.0.0.1

READ message received from /10.200.176.228

Executing
10.200.176.229 | 1182 | 127.0.0.1

10.200.176.229 | 1280 | 127.0.0.1

10.200.176.229 | 906 | 127.0.0.1

Read 1 live and 0 tombstone cells [ReadStage-2] | 2017-02-28 21:43:18.144000 |
10.200.176.229 | 1182 | 127.0.0.1

Read 1 live and 0 tombstone cells [ReadStage-2] | 2017-02-28 21:43:18.144000 |
10.200.176.229 | 1280 | 127.0.0.1

10.200.176.229 | 1326 | 127.0.0.1

Sending REQUEST_RESPONSE message to /10.200.176.228 message size 96 bytes
10.200.176.229 | 1521 | 127.0.0.1

Request complete | 2017-02-28 21:43:18.100053 |
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:
Querying data

• **Partition key (PK) columns**: The first part of primary key defines the hash that is used to spread data evenly across the data center.

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

Partition keys, clustering, and normal columns have different sets of restrictions within the WHERE clause. Those restrictions differ depending on 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 issues 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 procedure uses the `rank_by_year_and_name` example.

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

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

  ```cql
  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:

  ```cql
  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 might 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 >, >=, <=, or < 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
FROM cycling.rank_by_year_and_name
WHERE TOKEN(race_year, race_name) >= -3074457345618258603
  AND TOKEN(race_year, race_name) <= 3074457345618258602;
```

**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:

```
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>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>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</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>
<tr>
<td>100</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
The database stores and locates the data using a nested sort order. The data is stored in hierarchy that the query must traverse:

```
{ "key" : "100"  {
    "col_1" : "1"  {
        "col_2" : "1"  {
            "col_3" : "1"  {
                "col_4" : "1",
                "col_4" : "2",
                "col_4" : "3" },
            "col_3" : "2"  {
                "col_4" : "1",
                "col_4" : "2",
                "col_4" : "3" } },
        "col_2" : "2"  {
            "col_3" : "2"  {
                "col_4" : "1",
                "col_4" : "2",
                "col_4" : "3" } }
    },
    "col_1" : "2"  {
        "col_2" : "1"  {
            "col_3" : "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:

```
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>

(2 rows)

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:

```
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:
Figure 7:

<table>
<thead>
<tr>
<th>Key = 100</th>
<th>Col_1 IN(1,2)</th>
<th>Col_2=1</th>
<th>Col_3=1</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>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
Querying data

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:

```sql
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:

```sql
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 you 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 8:

<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></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></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>100</td>
<td>2</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></td>
<td></td>
</tr>
</tbody>
</table>
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>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>100</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>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
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 9:**

<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></td>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></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>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>2</td>
<td>3</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>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>3</td>
<td></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>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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:

```sql
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.

```sql
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:

```sql
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:

| month | day | race                                                      |
|-------+-----+----------------------------------------------------------|
| 1     | 23  | Vuelta Ciclista a la Provincia de San Juan               |
| 1     | 26  | Cadel Evans Great Ocean Road Race – Towards Zero Race Melbourne |
| 1     | 26  | Challenge Mallorca: Trofeo Porreres-Felanitx-Ses Salines-Campos |
| 1     | 28  | Cadel Evans Great Ocean Road Race                         |
| 1     | 28  | Challenge Mallorca: Trofeo Andratx-Mirador des Colomer    |
| 1     | 28  | Challenge Mallorca: Trofeo Serra de Tramuntana -2017     |
| 1     | 29  | Cadel Evans Great Ocean Road Race                         |
| 1     | 29  | Grand Prix Cycliste la Marseillaise                       |
| 1     | 29  | Mallorca Challenge: Trofeo Palma                          |
| 1     | 31  | Ladies Tour of Qatar                                      |
| 2     | 1   | Etoile de Bessages                                        |
| 2     | 1   | Jayco Herald Sun Tour                                     |
| 2     | 1   | Volta a la Comunitat Valenciana                           |
| 2     | 5   | G.P. Costa degli Etruschi                                 |
| 2     | 6   | Tour of Qatar                                             |
| 2     | 9   | South African Road Championships                          |
| 2     | 11  | Trofeo Laigueglia                                         |
| 2     | 12  | Clasica de Almeria                                       |
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 can 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` require turning off paging with the `PAGING OFF` command in `cqlsh`.

- Turn off paging.

```
PAGING OFF
```

- To retrieve results, use the `SELECT` command. 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-a5f8-97e6fb4c9b47</td>
<td>39</td>
<td>KRUIJSWIJK</td>
</tr>
<tr>
<td>Time-trial</td>
<td>6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47</td>
<td>3</td>
<td>KRUIJSWIJK</td>
</tr>
<tr>
<td>Sprint</td>
<td>220844bf-4860-49d6-9a4b-6b5d3a79cbfb</td>
<td>0</td>
<td>TIRALONGO</td>
</tr>
<tr>
<td>Time-trial</td>
<td>220844bf-4860-49d6-9a4b-6b5d3a79cbfb</td>
<td>182</td>
<td>TIRALONGO</td>
</tr>
</tbody>
</table>

- Alternatively, retrieve and sort results in ascending order.

```
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-4860-49d6-9a4b-6b5d3a79cbfb</td>
<td>182</td>
<td>TIRALONGO</td>
</tr>
<tr>
<td>Sprint</td>
<td>220844bf-4860-49d6-9a4b-6b5d3a79cbfb</td>
<td>0</td>
<td>TIRALONGO</td>
</tr>
<tr>
<td>Time-trial</td>
<td>6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47</td>
<td>3</td>
<td>KRUIJSWIJK</td>
</tr>
<tr>
<td>Sprint</td>
<td>6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47</td>
<td>39</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');
```

### Sorting and limiting results

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 calculates new values based on user-defined functions. This section uses several example tables in the cycling keyspace.

For SASI indexing, see queries in Using SASI.

#### Controlling the number of rows returned using PER PARTITION LIMIT

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.

```
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>Phillipe 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>

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>Phillipe GILBERT</td>
</tr>
</tbody>
</table>
Querying data

<table>
<thead>
<tr>
<th>Year</th>
<th>Race Description</th>
<th>Result</th>
<th>Rider Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2014</td>
<td>4th Tour of Beijing</td>
<td>2</td>
<td>Daniel MARTIN</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>2015</td>
<td>Giro d'Italia - Stage 11 - Forli &gt; Imola</td>
<td>1</td>
<td>Ilur 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.

```sql
CREATE TABLE cycling.cyclist_category (  
category text,  
points int,  
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-6b5d3a79c9fb</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-6b5d3a79c9fb</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-6b5d3a79c9fb</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-6b5d3a79c9fb</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-6b5d3a79c9fb</td>
<td>TIRALONGO</td>
</tr>
</tbody>
</table>

The database rejects this query if category is not a partition key or clustering column, because the query cannot locate the data on disk. Queries require a sequential retrieval across the entire cyclist_category table. In DataStax Enterprise, 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.

- 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)
);
Querying data

```sql
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>780</td>
<td>829aa84a-4bba-411f-a4fb-38167a987cda</td>
<td>SUTHERLAND</td>
</tr>
<tr>
<td>GC</td>
<td>1269</td>
<td>220844bf-4860-49d6-9a4b-6b5d3a79cbfb</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.

```sql
SELECT race_name, point_id, lat_long AS CITY_LATITUDE_LONGITUDE 
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>('Noville', (46.833, 6.6))</td>
</tr>
<tr>
<td>47th Tour du Pays de Vaud</td>
<td>4</td>
<td>('Vuiteboeuf', (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>

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</td>
</tr>
<tr>
<td>GILBERT</td>
<td>4th Tour of Beijing</td>
<td>2</td>
<td>Daniel</td>
</tr>
<tr>
<td>MARTIN</td>
<td>4th Tour of Beijing</td>
<td>3</td>
<td>Johan Esteban</td>
</tr>
<tr>
<td>CHAVES</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>1</td>
<td>Daniel</td>
</tr>
<tr>
<td>MARTIN</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>2</td>
<td>Johan Esteban</td>
</tr>
<tr>
<td>CHAVES</td>
<td>Tour of Japan - Stage 4 - Minami &gt; Shinshu</td>
<td>3</td>
<td>Benjamin</td>
</tr>
<tr>
<td>PRADES</td>
<td></td>
<td></td>
<td></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>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>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>

---

**DSE Search index enhancements**

DataStax Enterprise supports production-grade implementation of CQL Solr queries in DSE Search.

**Search index syntax**

DataStax Enterprise Search nodes can perform a wide range of functionality, and it's important to understand the syntax required for such tasks as filtering CQL queries, using the `solr_query` JSON formats, and escaping special characters.

**Filtering synopsis**

Applications connecting to the DSE database can use full-text search to filter on these fields within CQL queries. Applications no longer need to use Apache Solr™ APIs to filter queries on search index columns.

In a CQL `SELECT` statement, you can filter results using the following syntax:

```sql
SELECT selectors
FROM table
WHERE (indexed_column_expression | solr_query = 'search_expression')
[ LIMIT n ]
```
Write filters on search index columns as follows:

- Full text search using column names and CQL operators. See Filtering on search indexed columns.
- Recommended for most queries: Native CQL query expressions.
- Still supported: the Solr syntax in the solr_query option:
  
  # Basic expressions (Solr q parameter) in plain text, see Filtering queries with the solr_query q format. For improved performance, you should use Solr filter queries (fq) whenever possible. See Search index filtering best practices.
  
  # Advanced expressions using Solr JSON syntax, see Using the solr_query JSON format.

Filtering restrictions and best practices

The following restrictions apply to filtering a CQL query on a search indexed field:

- Search index queries are defaulted to an equivalent of LIMIT 10.
- Pagination is off by default.
  
  In dse.yaml, the cql_solr_query_paging option specifies when to use pagination (also called cursors).
- Apache Solr and Apache Lucene limitations for field names apply to pagination.
- Queries with smaller result sets will see increased performance with paging off.
- Limitations and known Apache Solr issues apply to DSE Search queries. For example: incorrect SORT results for tokenized text fields.
- Column aliases are not supported in search index queries.
- All of the fields that are queried on DSE SearchAnalytics clusters must be defined in the search index schema definition. Fields that are not defined in the search index schema columns are excluded from the results returned from Spark queries.
- Solr allows the LOCAL_ONE and ONE consistency levels for read operations.

Filtering on search indexed columns

On a DataStax Enterprise search node, when a query is filtered on an indexed column the corresponding Solr document is used. Filtering on indexed columns in queries that do not have the partition key fully defined do not require ALLOW FILTERING.

The search index now supports CQL operators, such as LIKE, IS NOT NULL, range, and =/=/!= to filter. See Native CQL search queries.

Combine filters to form more complex queries:

```
SELECT * FROM test.test WHERE
title IS NOT NULL AND
collection CONTAINS 'Anaisa Pye'
AND age > 35;
```

Sort using ORDER BY.
Filtering queries with the solr_query q format

Filter a CQL query using solr_query option using the syntax of the Solr q parameter in plain text. See Filtering on terms for examples.

```solr_query = 'q_expression'```

Examples

Filtering on a single column

To find rows in the pets table where the name column contains cat, dog but not fish:

```SELECT * FROM pets WHERE solr_query='name: cat name: dog -name:fish';```

To use punctuation such as a single quote in the Solr q expression, see Escaping characters in a solr_query.

Limiting results

When you name specific columns, DSE Search retrieves only the specified columns and returns the columns as part of the resulting rows. DSE Search supports projections (SELECT a, b, c...) only, not functions, for the select expression. The following example retrieves only the name column:

```SELECT name FROM keyspace.table WHERE solr_query='name:cat name:dog -name:fish'```

Use the LIMIT clause to specify how many rows to return. The following example retrieves only 1 row:

```SELECT * FROM keyspace.table WHERE solr_query='name:cat name:dog -name:fish' LIMIT 1```

Using the count function

Use the count() function in CQL Solr queries to return the number of rows that satisfy the Solr query:

```SELECT count(*) FROM table WHERE solr_query = '...';```

Using count() in combination with LIMIT or facets results in an error.

All response queries of the drivers have a custom payload where the total number of documents found is returned. This number is keyed as DSESearch.numFound.

Using the solr_query JSON format

DSE Search supports advanced Solr search features to apply filters to CQL queries using JSON-formatted expressions. Also see Overriding the default TimeZone (UTC) in search queries.

JSON query syntax

The JSON query expression syntax is a JSON string. The JSON-based query expression supports local parameters in addition to the following parameters:

```{
   "q": query_expression (string),
   "fq": filter_query_expression(s) (string_or_array_of_strings, ...),
   "facet": facet_query_expression (object)
}```
Querying data

```json
"sort": sort_expression (string),
"start": start_index(number),
timeAllowed: search_time_limit_ms,
"TZ": zoneID, // Any valid zone ID in java TimeZone class

"paging": "driver" (string),
"distrib.singlePass": true|false (boolean),
"shards.failover": true|false (boolean), // Default: true
"shards.tolerant": true|false (boolean), // Default: false
"commit": true|false (boolean),
"route.partition": partition_routing_expression (array_of_strings),
"route.range": range_routing_expression (array_of_strings),
"query.name": query_name (string),
```

For example:

```sql
SELECT id FROM nhanes_ks.nhanes WHERE solr_query='{"q":"ethnicity:Asian"}';
```

```sql
SELECT id FROM nhanes_ks.nhanes WHERE solr_query='{"q":"ethnicity:Mexi\*", "sort":"id asc"}'} LIMIT 3;
```

```sql
SELECT * FROM mykeyspace.mytable WHERE solr_query='{"q":"{!edismax}quotes:yearning or kills"}';
```

To use Apache Solr™ Extended DisMax Query Parser (eDisMax) with solr_query, you must include defaultSearchField in your schema.

Making distributed queries tolerant of shard failures

Because distributed queries contact many shards, making queries more tolerant of shard failures ensures more successful completions. Use shards.failover and shards.tolerant parameters to define query failover and tolerance of shard failures during JSON queries:

<table>
<thead>
<tr>
<th>Valid configurations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;shards.failover&quot;: true, &quot;shards.tolerant&quot;: false,</td>
<td>This default configuration enables query failover and disables fault tolerance. Attempt to retry the failed shard requests when errors indicate that there is a reasonable chance of recovery.</td>
</tr>
<tr>
<td>&quot;shards.failover&quot;: false, &quot;shards.tolerant&quot;: true,</td>
<td>Disable query failover. Enable fault tolerance. Make the query succeed, even if the query only partially succeeded, and did not succeed for all nodes.</td>
</tr>
<tr>
<td>&quot;shards.failover&quot;: false, &quot;shards.tolerant&quot;: false,</td>
<td>Disable query failover. Disable fault tolerance.</td>
</tr>
</tbody>
</table>

Failover and tolerance of partial results cannot coexist in the same JSON query. Queries support enabling tolerance for only one parameter.

The shards.tolerant parameter is not supported when deep paging is on.

Other fault tolerance configuration options include: netty_client_request_timeout in dse.yaml and read_request_timeout_in_ms in cassandra.yaml.

**JSON queries with literal characters that are Apache Solr™/Apache Lucene® special characters**

Solr and Lucene support escaping special characters that are part of the query syntax.
Table 13: Solr/Lucene escaped special characters

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Plus. Required search term operator.</td>
</tr>
<tr>
<td>-</td>
<td>Minus. Prohibited search term operator.</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Double ampersand. AND operator. Both terms either side of the operator are required for a match.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>Exclamation mark. NOT operator.</td>
</tr>
<tr>
<td>(</td>
<td>Left parenthesis</td>
</tr>
<tr>
<td>)</td>
<td>Right parenthesis</td>
</tr>
<tr>
<td>*</td>
<td>Double quote</td>
</tr>
<tr>
<td>~</td>
<td>Tilde</td>
</tr>
<tr>
<td>*</td>
<td>Asterisk</td>
</tr>
<tr>
<td>?</td>
<td>Question mark</td>
</tr>
<tr>
<td>:</td>
<td>Colon</td>
</tr>
</tbody>
</table>

Using JSON with `solr_query` requires additional syntax for literal characters that are Lucene special characters.

Syntax for a simple search string:

<table>
<thead>
<tr>
<th>Simple search string</th>
<th>Solr query</th>
<th>CQL Solr query</th>
</tr>
</thead>
<tbody>
<tr>
<td>mytestuser1?</td>
<td>name:mytestuser1?</td>
<td>solr_query='&quot;q&quot;:&quot;name:mytestuser1?&quot;'</td>
</tr>
</tbody>
</table>

Syntax for a complex search string:

<table>
<thead>
<tr>
<th>Complex search string</th>
<th>Solr query</th>
<th>CQL Solr query</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1+1):2</td>
<td>e:(1+1):2</td>
<td>solr_query='&quot;q&quot;:&quot;e:(1+1):2&quot;'</td>
</tr>
</tbody>
</table>

Field, query, and range faceting with a JSON query

Specify the facet parameters inside a facet JSON object to perform field, query, and range faceting inside Solr queries. Distributed pivot faceting is supported. The query syntax is less verbose to specify facets by:

- Specifying each facet parameter without the facet prefix that is required by HTTP APIs.
- Expressing multiple facet fields and queries inside a JSON array.
Querying data

Faceted search example
SELECT * FROM solr WHERE solr_query='{"q":"id:*","facet":{"field":"type"}}';

Query facet example
SELECT * FROM solr WHERE solr_query='{"q":"id:*","facet":{"query":"type:0"}}';

Multiple queries example
SELECT * FROM solr WHERE solr_query='{"q":"id:*","facet":{"query":
["type:0","type:1"]}}';

Distributed pivot faceting example
SELECT id FROM table WHERE solr_query='{"q":"id:*","facet":
{"pivot":"type,value","limit":"-1"}}'

Range facet example
SELECT * FROM solr WHERE solr_query='{"q":"id:*","facet":{"range":"type",
"f.type.range.start":-10, "f.type.range.end":10, "range.gap":1}}}';

The returned result is formatted as a single row with each column corresponding to the output of a
facet (either field, query, or range). The value is represented as a JSON blob because facet results
can be complex and nested. For example:
facet_fields
| facet_queries
------------------------+------------------------{"type":{"0":2,"1":1}} | {"type:0":2,"type:1":1}

Range by date facet example
SELECT * FROM solr WHERE solr_query='{"q":"business_date:*","facet":
{"range":"business_date", "f.business_date.range.start":"2015-01-01T00:00:00Z",
"f.business_date.range.end":"2015-08-01T00:00:00Z",
"f.business_date.range.gap":"+1MONTH"}}';

Solr range facets before, after, and between might return incorrect and inconsistent results on multi-node
clusters. See SOLR-6187 and SOLR-6375.
Interval facet example
SELECT * FROM solr WHERE solr_query='{"q":"id:*","facet":{"interval":"id",
"interval.set":"[*,500]"}';

Tracing distributed queries
During a distributed query, every node is responsible for a set of token ranges. A shard is the node/ranges
combination. The shard token range is reported:
• In the shards.info response for HTTP queries.
• In the system_traces.events table for HTTP queries that provide cassandra.trace=true and CQL Solr
queries that enable tracing at the driver level.

Page
170

CQL for DataStax Enterprise 6.7 Latest DSE version Latest 6.7 patch: 6.7.7


**Solr single-pass CQL queries**

Single-pass distributed queries are supported in CQL Solr queries.

There are a number of ways to use a single-pass distributed query instead of the standard two-pass query. You can:

- Specify the `distrib.singlePass` Boolean parameter in the query expression. Example:

  ```cql
  SELECT * FROM keyspace.table WHERE solr_query = '{"q" : ":*:", "distrib.singlePass" : true}'
  ```

- Use a `token()` or partition key restriction in the `WHERE` clause. Example:

  ```cql
  SELECT * FROM keyspace.table WHERE token(id) >= 3074457345618258604 AND solr_query = '{"q" : ":*:"}'
  ```

- Execute a `COUNT` query. Example:

  ```cql
  SELECT count(*) FROM keyspace.table WHERE solr_query = '{"q" : ":*:"}'
  ```

- Specify the primary key elements in the `SELECT` clause. Example:

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

Using a single-pass distributed query has an operational cost that includes potentially more disk and network overhead. With single-pass queries, each node reads all rows that satisfy the query and returns them to the coordinator node. An advanced feature, a single-pass distributed query saves one network round trip transfer during the retrieval of queried rows. A regular distributed query performs two network round trips:

1. The first one to retrieve IDs from DSE Search that satisfy the query.
2. The second one to retrieve only the rows that satisfy the query from the database, based on IDs from the first step.

Single-pass distributed queries are most efficient when most of the documents found are returned in the search results. Single-pass distributed queries are inefficient when most of the documents found are not returned to the coordinator node.

For example, a distributed query that only fans out to a single node from the coordinator node will likely be most efficient as a single-pass query.
Querying data

With single-pass queries, there is a limitation that returns only the table columns that are defined in the Solr schema.xml in the query results. This limitation also applies to map entries that do not conform to the dynamic field mapping.

Consider the cycling.cyclist_name table example above. If the column firstname was not indexed, but needed to be retrievable by queries, the entry in the schema.xml for the firstname field could be:

```xml
<field indexed="false" multiValued="false" docValues="false" name="firstname" type="StrField"/>
```

If the Solr core was created without a customized schema.xml file and generateResources=true was included, the default schema.xml file includes all fields that exist in the table. For more information, see dsetool create_core.

JSON query name option

Using the following syntax to name your queries to support metrics and monitoring for performance objects. Naming queries can be useful for tagging and JMX operations, for example.

```
SELECT id FROM nhanes_ks.nhanes WHERE solr_query=' {"query.name":"Asian subjects", "q":"ethnicity:Asia*"}' LIMIT 50;
```

JSON query commit option

When executing custom queries after bulk document loading, with auto soft commit disabled (or is an extremely infrequent configured value), use the JSON query commit option to ensure that all pending updates are soft-committed before the running a query. The commit makes the latest data visible to a query.

Do not use the JSON commit option for live operations against a production cluster. The commit option is not a replacement for the normal auto soft commit process or the COMMIT SEARCH INDEX command.

By default, the commit option is set to false. The example below sets the commit option to true:

```
SELECT * FROM wiki.solr WHERE solr_query='{"q":"title:Asia", "commit":true} LIMIT 50;
```

Queries that dynamically enable paging using a driver

To dynamically enable paging when cql_solr_query_paging is set to off in dse.yaml, set the Solr paging parameter to driver ("paging":"driver"). For example:

```
SELECT id FROM wiki.solr WHERE solr_query='{"q":"", "sort":"id asc", "paging":"driver"}';
```

Limiting queries by time

DSE Search supports limiting queries by time by using the Solr timeAllowed parameter.

DSE Search differs from native Solr:

- If timeAllowed is exceeded, an exception is thrown.
- If timeAllowed is exceeded, and the additional shards.tolerant parameter is set to true, the application returns the partial results collected so far.

When partial results are returned, the CQL custom payload contains the DSESearch.isPartialResults key.

Example with a 30 second timeout:

```
SELECT * FROM users
```
Querying data

WHERE solr_query = '{ "q": ":*:", "timeAllowed":30000}';

As of DSE 6.7.7 and later, the Solr timeAllowed parameter is enabled by default to prevent long running shard queries, such as complex facets and Boolean queries, from using system resources after they have timed out from the DSE Search coordinator.

By default, the value for timeAllowed is the same as the internode_messaging_options.client_request_timeout_seconds parameter in dse.yaml. Queries that breach client_request_timeout_seconds fail by default.

The 50th percentile latency for queries using timeAllowed should be within 5% of the same query that does not use timeAllowed.

The timeAllowed parameter applies to these queries:

- Standard queries and filtering queries, where the timeout is applied on a match collection.
- Facet queries, including pivot facets.

The json.facet queries are not supported.

Queries that are terminated due to a timeout are logged with a warning.

Using the timeAllowed parameter can cause a small request latency cost for each request. If the latency cost is too high, disable timeAllowed during DSE start by using "-Ddse.timeAllowed.enabled.default=false", or set timeAllowed.enable to false in the query.

Escaping characters in a solr_query

Solr queries require escaping special characters that are part of the query syntax.

Table 14: Solr/Lucene escaped special characters

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Plus. Required search term operator.</td>
</tr>
<tr>
<td>-</td>
<td>Minus. Prohibited search term operator.</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Double ampersand. AND operator. Both terms either side of the operator are required for a match.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>Exclamation mark. NOT operator.</td>
</tr>
<tr>
<td>(</td>
<td>Left parenthesis</td>
</tr>
<tr>
<td>)</td>
<td>Right parenthesis</td>
</tr>
<tr>
<td>*</td>
<td>Double quote</td>
</tr>
<tr>
<td>~</td>
<td>Tilde</td>
</tr>
<tr>
<td>*</td>
<td>Asterisk</td>
</tr>
<tr>
<td>?</td>
<td>Question mark</td>
</tr>
<tr>
<td>:</td>
<td>Colon</td>
</tr>
</tbody>
</table>

To escape these characters, use a slash (\) before the character to escape. For example, to search for a literal double quotation mark (" character, escape the " for Solr with \".

When using solr_query, you can escape special characters using either CQL Solr or JSON forms:

CQL Solr

...WHERE solr_query='field:value'

JSON

WHERE solr_query='{ "q": "field:value"}'
Querying data

JSON-encoded queries require that values must also be JSON-escaped for special characters.

For queries that contain double quotation marks, use triple slashes `\\`:

- For query syntax: One slash `\` to escape the "
- For the JSON string syntax: Two slashes `\\` to escape the \\
  Triple slashes `\\\` escape both characters in " to produce `\"` (an escaped escape) and `\"` (an escaped double quote).

**Escaping single quotation marks**

- Double the single quotation mark (‘)
  
  **CQL**
  ...
  WHERE solr_query='name:Walter''s'
  
  **JSON**
  ...
  WHERE solr_query='{ "q": "Walter''s"}'}

- Use dollar-quotes for the string constant
  
  **CQL**
  ...
  WHERE solr_query='$name:Walter's$
  
  **JSON**
  ...
  WHERE solr_query='${ "q": "Walter's"}$'

**Escaping double quotation marks**

**CQL**

Double the single quotation mark (‘) and add the backslash (\) for Solr escaping

```
...WHERE solr_query='name:Walter''s'
```

**JSON**

Escape " to `\"` to escape both special characters for JSON

```
...WHERE solr_query='{ "q": "Walter\\"s"}'}
```

**Exact and fuzzy query examples**

**Exact phrase query**

For a row that looks like this, with an email address that includes a double quotation mark

```
greenr"q@example.com
```

```sql
INSERT INTO users(id, email) VALUES(1, 'greenr"q@example.com')"
```

Perform a phrase query to search for the email address that is enclosed in double quotation marks:

```sql
SELECT * FROM users WHERE solr_query = '
{ "q": "*:*", "fq": "email:\"greenr\\\"q@example.com\""}
';
```

**Fuzzy query**

For a row that looks like this, with the same email address that includes a double quotation mark

```
greenr"q@example.com
```

```sql
select * from test.users where solr_query='("q":"email:r\\\"q@example")' ;
```

```
id | email | solr_query
-----------------|--------|------------------------
1   | greenr"q@example.com | null
```
For a term query (fuzzy search) for all email addresses that include r"q@example, remove the double quotation marks but retain triple quotation marks for the escaped double quotation character that is part of the email address:

SELECT * FROM users where solr_query = ' {
  "q": "*:*",
  "fq": "email:r\\"q@example""
}
';

Using JSON with solr_query requires additional syntax for literal characters that are Lucene special characters. See JSON queries with literal characters that are Solr special characters.

**Native CQL search queries**

You can query the search index using native CQL statements, without using solr_query syntax with ALLOW FILTERING as required in earlier versions of DSE.

Use native CQL SELECT statements to filter results using the following syntax:

```cql
SELECT selectors FROM table
WHERE indexed_column_expression
[LIMIT n]
[ORDER BY column_name]
```

where `indexed_column_expression` is the WHERE statement in the following native CQL search query examples.
Querying data

Querying all columns equally
Query directly any search-indexed column, even when you don't know if the column is part of the primary key or if it has been specified in full.

```sql
SELECT * FROM customer.region1 WHERE title='Anaisa Pye'
```

```sql
SELECT * FROM customer.region1 WHERE id LIKE '237%
```

Operators
Query any column that is indexed using standard CQL operators, including LIKE, IS NOT NULL, and != operators.

```sql
SELECT * FROM customer.region1 WHERE id LIKE '237%
```

```sql
SELECT * FROM customer.region1 WHERE title IS NOT NULL
```

```sql
SELECT * FROM customer.region1 WHERE title!= 'Anaisa Pye'
```

```sql
SELECT * FROM customer.region1.solr WHERE title='Anaisa Pye'
```

```sql
SELECT * FROM customer.region1 WHERE field >0 and field <=100
```

```sql
SELECT * FROM customer.region1 WHERE title LIKE 'Anais%'
```

```sql
SELECT * FROM customer.region1 WHERE title LIKE '%naisa'
```

```sql
SELECT * FROM customer.region1 WHERE title LIKE '%nais%'
```

```sql
SELECT * FROM customer.region1 WHERE title IN ('Anaisa Pye', 'Karlsbrunn')
```

```sql
SELECT * FROM customer.region1 WHERE collection CONTAINS 'Anaisa Pye'
```

Collections
Lists and sets support only the CONTAINS operator.

```sql
SELECT * FROM customer.region1 WHERE title IS NOT NULL AND collection CONTAINS 'Anaisa Pye' AND age > 35
```

Maps
Query maps using dynamicfield with a map entry equality syntax:

```sql
SELECT * FROM customer.region1 WHERE dynamicfield['dynamicfield_key1'] = 'value1'
```

Prepared statements
To reduce the workload on the coordinator by removing the overhead of parsing the query, use two question marks as input parameters.

```sql
SELECT id FROM customer.region1 WHERE title LIKE ? limit ?
```

Tokenized text
The default for CQL text type is solr.StrField, in contrast with the solr.TextField text type in previous DSE versions. Queries use this criteria for text columns to be considered tokenized/analyzed:
Querying data

<table>
<thead>
<tr>
<th>Indexed by</th>
<th>Tokenized</th>
<th>Query behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>solr.StrField</td>
<td>no</td>
<td>An exact or wildcard match performs as expected</td>
</tr>
<tr>
<td>solr.TextField</td>
<td>yes</td>
<td>Uses single term phrase query matching.</td>
</tr>
</tbody>
</table>

Tokenized text is restricted with LIKE, IS NOT NULL, and single token match. Multi-token match is not supported.

### StrField and TextField

In these examples, `id` is indexed by StrField and `title` is indexed by TextField

<table>
<thead>
<tr>
<th>id</th>
<th>body</th>
<th>date</th>
<th>solr_query</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>test1</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>This is a test1</td>
</tr>
<tr>
<td>test2</td>
<td>text</td>
<td>null</td>
<td>null</td>
<td>This is a test2</td>
</tr>
</tbody>
</table>

This query matches nothing, since there is not an exact match against `id`.

```sql
SELECT * FROM customer.region1 WHERE id LIKE 'test';
```

This query uses wildcard `*` to match both rows.

```sql
SELECT * FROM customer.region1 where id LIKE 'test%';
```

This query has undefined behavior for `title` multi-token match.

```sql
SELECT * from customer.region1 where id LIKE 'test%' and title LIKE 'is a';
```

Matches no rows, `id` matches but `title` doesn't.

```sql
SELECT * FROM customer.region1 WHERE id LIKE 'test%' AND title LIKE 'is a';
```

Matches both rows as expected.

```sql
SELECT * FROM customer.region1 WHERE id LIKE 'test%' AND title LIKE 'is';
```

Matches both rows as expected, with single token match for `title`.

```sql
SELECT * FROM customer.region1 WHERE id LIKE 'test%' AND title LIKE 'test%';
```

### Tuples and UDTs

Queries on tuples and user-defined types (UDT) support only the equal sign (`=`) operator. The query must provide all elements of the tuple or UDT. Queries with nested Tuple or UDTs or collections are not supported due to operator scope constraint.
Querying data

This example query provides all elements of the tuple.

```cql
SELECT * FROM customer.region1 WHERE skill={name:'masonry',type:'type1'}
```

Search index filtering best practices

DataStax recommends following these best practices for running queries in DSE Search:

- Use **CQL** to run search queries. Perform all data manipulation with CQL, except for deleting by query.

- Use the simplest and best fit Solr types to fulfill the required type for your query. See **Understanding schema field types**.

- For improved performance, use Solr filter query (fq) parameters instead of q parameters whenever possible. The results from filter queries are stored in a cache. You can reduce the average response time from seconds to milliseconds. The following example queries the cyclist first name and last name:

  ```cql
  '{"q":"*:*", "fq":"firstname:Alex AND lastname:FRAME"}"
  ```

  Each fq name and value string pair can be a member of an fq array. The fq name and value pairs are treated as if they are separated by AND. For example:

  ```cql
  '{"q":"*:*", "fq":["lastname:BELKOV", "nationality:Russia"]}'
  ```

  Adjust your queries so that the results fit into the memory cache.

- Use profiles when **creating** a search index.

- Avoid querying nodes that are indexing.
  For responding to queries, DSE Search ranks the nodes that are not performing search indexing higher than nodes that are indexing. If nodes that are indexing are the only nodes that can satisfy the query, the query does not fail but can return only partial results.

- Avoid wildcard queries that contain multiple search tokens, unless you are making those queries in conjunction with the Lucene KeywordTokenizer class. If the analyzer creates multiple search tokens from your original input, then you can perform wildcard queries on those tokens. You can also use the KeywordTokenizer class and define a multiple term wildcard query.

- If you specify time-to-live (TTL) settings for data, to ensure that a Solr index synchronizes with the data, add a query condition similar to the following example. The query condition filters out the expired data that is older than TTL boundary in the query. The query condition does not remove expired data from the index; instead, the condition improves the query consistency. In the example, the `epoch time in seconds` is the time in the past rounded to the nearest minute, hour, and so on.

  ```cql
  "q":"-_ttl_expire:[* TO <epoch time in seconds>]"
  ```

  In addition to the TTL guidance shown in the previous point, do not use the current time with every query. You can create a window of staleness; for example, set your TTL filter value every 60 seconds or other time period that the data can be out of synchronization, but the filter cache is cleared after the time period. This avoids caching filters that cannot be reused. You can also use `{!cache=false}` with your TTL filter, which does not cache the data. For example:

  ```cql
  "fq":{"cache=false"-_ttl_expire:[0 TO <epoch time in seconds>]"
  ```

- For optimal CQL single-pass queries, including queries where solr_query is used with a partition restriction, and queries with partition restrictions and a search predicate, ensure that the columns to SELECT are not indexed in the search index schema.
Auto-generation indexes all columns by default. You can ensure that the field is not indexed but still returned in a single-pass query. For example, this statement indexes everything except for column c3, and informs the search index schema about column c3 for efficient and correct single-pass queries.

```sql
CREATE SEARCH INDEX
ON test_search.abc
WITH COLUMNS * { indexed : true }, c3 { indexed : false };
```

- When vnodes are not configured in a cluster, distributed queries in DSE Search are most efficient when the number of nodes in the queried data center (DC) is a multiple of the replication factor (RF) in that DC.

- Avoid using too many terms in the query, like:

  ```sql
  SELECT request_id, store_id
  FROM store_search.transaction_search
  WHERE solr_query = '{"q":"*:*","shards.failover":true,
  "shards.tolerant":false,"fq":"store_id:store1a
  store_id:store2b store_id:store2c ... store_id:store19987d"}';
  ```

  Instead, use a terms filter query.

- When writing collections with few collection updates, DataStax recommends frozen collections over non-frozen collections to address query latency.

  For example, a simple frozen set of text elements:

  ```sql
  CREATE TABLE foo (
    id text, values frozen<set<text>>, PRIMARY KEY (id)
  );
  CREATE TYPE name (
    first text, last text
  );
  ```

  A frozen list of UDTs:

  ```sql
  CREATE TABLE tableWithList (
    id text, names frozen<list<frozen<name>>>), PRIMARY KEY (id)
  );
  ```

- JSON query limitations

  Failover and tolerance of partial results cannot coexist in the same JSON query. Queries support enabling tolerance for only one parameter.

  The `shards.tolerant` parameter is not supported when deep paging is on.

### Limiting results and paging

DSE Search integrates native driver paging with Apache Solr™ cursor-based paging. Pagination, also called cursors, supports using a cursor to scan results. Solr pagination restrictions apply.

#### Using paging with CQL SELECT

A CQL SELECT run by DSE Search query uses deep paging when:

<table>
<thead>
<tr>
<th>Required</th>
<th>One of these applies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Driver is using paging</td>
<td>No LIMIT is provided.</td>
</tr>
<tr>
<td>Driver is using paging</td>
<td>The number of rows to return is greater than the driver's page size.</td>
</tr>
</tbody>
</table>

Deep paging is not used if a LIMIT is provided, and the LIMIT is less than the driver's fetch size.
**Using paging with CQL Solr queries (solr_query)**

When using CQL Solr queries with pagination enabled, your queries might experience a performance slowdown because Solr is not able to use its query result cache when pagination is configured. If you do not want to paginate through large result sets, disable pagination when running CQL Solr queries. See the driver documentation.

In dse.yaml, the `cql_solr_query_paging` option specifies when to use pagination:

- To use the driver pagination settings by default when a driver connects to the database and executes a CQL SELECT statement using a search index `solr_query` option, set the `cql_solr_query_paging` to `driver`.

- To enable pagination persistently with CQL Solr queries, set `cql_solr_query_paging: on` in dse.yaml and restart the node.

- To dynamically enable paging when `cql_solr_query_paging` is set to `off` in dse.yaml, set the Solr paging parameter to `driver` ("paging":"driver"). For example:

```sql
SELECT id FROM wiki.solr WHERE solr_query= '{"q":"*", "sort":"id asc", "paging":"driver"}';
```

SearchAnalytics nodes always use driver paging settings. See DSE Analytics and Search integration.

See the documentation for the CQL shell `PAGING` command and the `driver`.

If cqlsh PAGING is enabled, query results display in 100-line chunks followed by the more prompt. Press the space bar to move to the next chunk. When disabled with PAGING OFF, the entire results are displayed.

It is not mandatory to use a sort clause. However, if a sort clause is not provided, sorting is undefined.

**Example of CQL SELECT query**

```sql
SELECT * FROM wiki.solr WHERE id LIKE 'Journal%';
```

**Examples of CQL Solr queries (solr_query)**

The word Journal is contained in ~159 entries in the body or title. Use count to determine the number of rows that match:

```sql
SELECT count(*) FROM wiki.solr WHERE solr_query = 'Journal';
```

Count returns only a single row; it is not affected by the 10 row limit.

```
count
------
  159
```

(1 rows)

Run the same query without count (and `cql_solr_query_paging: off`):

```sql
SELECT id FROM wiki.solr WHERE solr_query = 'Journal';
```

Only 10 rows are returned.

```
id
-----
23759487
23732986
23759527
```
To return all matching IDs, override the `cql_solr_query_paging` setting:

```sql
SELECT id FROM wiki.solr
WHERE solr_query='{"q":"Journal", "paging":"driver"}';
```

### Identifying the partition key

Solr CQL queries support restriction to a single partition key. Partition key restrictions work only when `partitionKey` is explicitly indexed or the schema explicitly includes all of the components of the database partition key. In your schema, you can override `partitionKey` when not using joins.

Example:

```sql
SELECT id, date, value FROM keyspace.table WHERE id = 'series1' AND
solr_query='value:bar*';
```

CQL partition key restrictions work only with fully specified partition keys.

```sql
CREATE TABLE vtbl (k1 text, k2 text, value text, PRIMARY KEY (k1, k2));
```

For example, with the above table, avoid using a query such as shown below:

```sql
SELECT * FROM vtbl WHERE k1 = '50' AND solr_query='value:*'
```

Use a filter query against the partially specified composite partition key, "fq":"k1:50":

```sql
SELECT * FROM valuetable WHERE solr_query='{"q":"value:*", "fq":"k1:50"}'
```

### Restricting nodes queried using the Solr token function

Solr CQL queries support limited use of the token function. The token function enables targeted search that restricts the nodes queried to reduce latency.

Using the Solr token function is for advanced users only and is supported only in specific use cases.

Example:

```sql
SELECT id, value FROM keyspace.table WHERE token(id) >= -3074457345618258601 AND token(id)
<= 3074457345618258603 AND solr_query='id:*'
```

Example with an open range:

```sql
SELECT id, value FROM keyspace.table WHERE token(id) >= 3074457345618258604 AND
solr_query='id::*'
```

Constraints apply to using the token function with Solr CQL queries:
Querying data

- `token()` cannot be used with `route.range` or `route.partition`
- Wrapping `token()` ranges are not supported
- A specified `token()` range must be owned by a single node; ranges cannot span multiple nodes
- Because DSE uses the Solr single-pass queries, only the fields that are declared in the search schema are returned in the query results. If you have columns that do not need to be indexed but still need to be returned by using a token-restricted query, you can declare the columns as stored non-indexed fields in your `schema.xml` file.

Overriding `_partitionKey` when not using joins
The special `_partitionKey` field is used internally for joins. If you do not plan on using joins, you can override this field declaration in the `schema.xml` file for only the indexed and `docValues` properties.

To prevent the `_partitionKey` field from being indexed, set `indexed` to `false`:

```xml
<field name="_partitionKey" type="string" indexed="false"/>
```

To prevent building `docValues` for the `_partitionKey` field, set `docValues` to `false`:

```xml
<field name="_partitionKey" type="string" docValues="false"/>
```

Filtering on terms
Filter rows returned by a CQL SELECT statement on terms using the Solr Standard Parser syntax.

The basic syntax to limit queries has the following syntax:

```cql
SELECT column_list FROM table_name
WHERE solr_query = 'standard_term_expression ...';
```

The Solr Standard Parser is a case-sensitive term search that supports boolean expressions with wildcards. CQL for DSE Search also supports more complex searches using JSON-formatted query strings.

This section uses the Wikipedia Demo included in DataStax Enterprise. Replace `standard_term_expression` with the `solr_query` value from corresponding tables below to return the results:

```cql
SELECT count(*) FROM solr
WHERE solr_query = 'q_search_expression';
```

CQL Solr queries do not support native functions or column aliases as selectors. Only `count(*)` is supported with search index queries. Results use the Solr count process. Results might vary from the native CQL count function.

Filtering on words, phrases, or substrings
Find rows that contain words, phrases, or substrings in indexed fields, similar to `LIKE` in SQL.

- **Term**: A word that contains no spaces or punctuation and is separated from other content by a beginning or end of line, space, or punctuation mark.
- **Substring**: Match character patterns in a term. Use asterisk (*) for zero or more characters. Use question mark (?) for zero or one character in a term search.
- **Phrase**: Exact string that contains spaces or punctuation or both. Wrap phrases in double-quotes to search for the complete string when separated from other content by a space, punctuation mark, or beginning or end of a line.
Prerequisites: To run the following examples, set up Term and phrase searches using the wikipedia demo. Use cqlsh on a search node and replace the search_expression in the following statement with the example string.

```cql
SELECT count(*) FROM wiki.solr
WHERE solr_query = search_expression;
```

- Search for a single term on any indexed column or a specific column:

<table>
<thead>
<tr>
<th>Search in</th>
<th>Syntax</th>
<th>Example</th>
<th>Results</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any column</td>
<td>'term'</td>
<td>'Journal'</td>
<td>count</td>
<td>159 Count of rows that contain the word African in the title or body fields.</td>
</tr>
<tr>
<td>Specific column</td>
<td>'column_name:term'</td>
<td>'title: Journal'</td>
<td>count</td>
<td>9 Count of rows that contain the word Journal in the title. Use column names in the query syntax to limit searches to specific columns.</td>
</tr>
</tbody>
</table>

- Search for substrings:
  Asterisk indicates zero or more characters.
  Question mark indicates zero or one character.

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
<th>Results</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beginning of term</td>
<td>'title:Africa?'</td>
<td>count</td>
<td>16 Count of rows that have a term that begins with Africa in the title, but can have only one additional character.</td>
</tr>
<tr>
<td>Anywhere in term</td>
<td>'title:<em>at</em>'</td>
<td>count</td>
<td>559 Count of rows that have the term at or a term that contains at.</td>
</tr>
</tbody>
</table>

- Searching for a phrase in any column or a specific column:

<table>
<thead>
<tr>
<th>Search in</th>
<th>Syntax</th>
<th>Example</th>
<th>Results</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any column</td>
<td>&quot;phrase&quot;</td>
<td>&quot;African Journal&quot;</td>
<td>count</td>
<td>8 Count of rows that contain the complete phrase in the title or body.</td>
</tr>
<tr>
<td>Specific column</td>
<td>'column_name:&quot;phrase&quot;'</td>
<td>'title:&quot;African Journal&quot;'</td>
<td>count</td>
<td>8 Count of rows that contain the complete phrase only in the title.</td>
</tr>
</tbody>
</table>

- Search for multiple words or phrases in an indexed field, such as title, using operators:
## Advanced term and phrase searches

- Find terms with similar spelling:

<table>
<thead>
<tr>
<th>Type of query</th>
<th>solr_query value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Similar spelling</td>
<td>'Kenya~'</td>
<td>The Solr fuzzy search syntax uses the Damerau-Levenshtein Distance algorithm to determine similarity of spelling based on distance. A range of 0-2 can be specified, such as Kenya~1. Default distance: 2</td>
</tr>
<tr>
<td>Proximity search</td>
<td>&quot;football Bolivia&quot;~10</td>
<td>Searches for football and Bolivia within 10 words of each other.</td>
</tr>
<tr>
<td>Range searches</td>
<td>'title:football TO soccer'</td>
<td>Supports both inclusive and exclusive bounds using square brackets and curly braces, respectively.</td>
</tr>
<tr>
<td>Term boosting</td>
<td>&quot;football&quot;^4 &quot;soccer&quot;</td>
<td>By default, the boost factor is 1. Must be a positive number.</td>
</tr>
</tbody>
</table>

# The following example shows a search for Science in the title field using the default distance 2:

```sql
SELECT title FROM solr WHERE solr_query = 'title:Science~';
```

The first row is unexpected, given the difference of *ci in Science* to *p in Spence*; the letters are similar, but the meaning is dissimilar. The fifth row has *Schenck*, even more dissimilar, although within the distance of 2 just like *Spence*.

<table>
<thead>
<tr>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aitken Spence</td>
</tr>
<tr>
<td>African Journal of Marine Science</td>
</tr>
<tr>
<td>African Journal of Range and Forage Science</td>
</tr>
<tr>
<td>African Journal of Science and Technology</td>
</tr>
<tr>
<td>Maarten Schenck van Nydeggen</td>
</tr>
</tbody>
</table>
This example shows a search for Science in the title field using a smaller distance of 1, which produces a more accurate result:

```
SELECT title FROM solr WHERE solr_query = 'title:Science~1';
```

In this case, only the title with Sciences is different from all the other titles with Science in the title.

**Geospatial queries for Point and LineString**

Perform geospatial queries for Point and LineString.

**Create a CQL table with geospatial columns**

For geospatial field types, enclose the data type name in single quotes when creating or altering a table. For example:

```
CREATE TABLE test (
    id text PRIMARY KEY,
    point 'PointType',
    linestring 'LineStringType');
```

**Create a search index**

Create a basic search index. Geospatial types will be included in the created index, indexed as SpatialRecursivePrefixTreeFieldType, and no lenient mode required for PolygonType.

**Inserting or updating geospatial data**

To insert or update data in the database, specify geotypes in the INSERT or UPDATE command. For example:

```
INSERT INTO test (id, point, linestring) VALUES ('1', 'POINT(5 50)', 'LINESTRING (30 10, 10 30, 40 40)');
INSERT INTO test (id, point, linestring) VALUES ('2', 'POINT(100 100)', 'LINESTRING (50 20, 20 40, 50 50)');
```

**Querying geospatial data**

Find points within a 10 unit radius from point (4, 49):

```
SELECT * FROM test
```
Querying data

WHERE solr_query= '{ "q":"*:*", "fq":"point:"IsWithin(BUFFER(POINT(4.0 49.0), 10.0))\""

<table>
<thead>
<tr>
<th>id</th>
<th>linestring</th>
<th>point</th>
<th>solr_query</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LINESTRING (30.0 10.0, 10.0 30.0, 40.0 40.0)</td>
<td>POINT (5.0 50.0)</td>
<td>null</td>
</tr>
</tbody>
</table>

Find linestring that contains the point (10, 30):

SELECT * FROM test WHERE solr_query='linestring:"Intersects(POINT(10 30))"';

<table>
<thead>
<tr>
<th>id</th>
<th>linestring</th>
<th>point</th>
<th>solr_query</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LINESTRING (30.0 10.0, 10.0 30.0, 40.0 40.0)</td>
<td>POINT (5.0 50.0)</td>
<td>null</td>
</tr>
</tbody>
</table>

Find all points on a linestring and return a heatmap:

SELECT * FROM test
WHERE solr_query='{"q":id:*", "facet": {"heatmap": "point"}}';

facet_heatmaps

```json
{
  "point":{
    "gridLevel":4,
    "columns":16,
    "rows":16,
    "minX":-1000.0,
    "maxX":1000.0,
    "minY":-1000.0,
    "maxY":1000.0,
    "counts_ints2D":[
      null,
      null,
      null,
      [0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0],
      null,
      [0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,7,3,0,0,0,0,0,0,0],
      null,
      null,
      null,
      null,
      null,
      null,
      null,
      null
    ]
  }
}
```

See this tutorial for details on how to index and query geospatial Polygons and MultiPolygons.

**Using dynamic fields**

Using dynamic fields, you can index content in fields that are not explicitly defined by the schema. A common use case for dynamic fields is to identify fields that should not be indexed or to implement a schema-less index.

Search schema fields that are dynamic and multiValued are not supported in CQL-based search indexes.

**Best practices**

- Avoid or limit the use of dynamic fields.
  - Apache Lucene® allocates memory for each unique field (column) name. For example, for a row with columns A, B, C, and another row with B, D, E, Lucene allocates 5 chunks of memory. For millions of rows, the heap is unwieldy.

- Instead of using dynamic fields, use a default query field, and then perform queries against the combined field.

- Use the FieldInputTransformer (FIT) API.

**Spatial subfields prefix naming conventions**

Dynamic fields for spatial subfields use prefix naming conventions to enable using map types to store geospatial data:

```xml
<types>
  <fieldType class="solr.LatLonType" multiValued="false" name="LatLonType"
    subFieldPrefix="llt_"/>
  <fieldType name="tdouble" class="solr.TrieDoubleField" precisionStep="8"
    positionIncrementGap="0"/>
</types>
<fields>
  <dynamicField indexed="true" name="latmap*" stored="true" type="LatLonType"/>
  <dynamicField name="llt_*" type="tdouble" indexed="true" stored="true"/>
</fields>
```

**To use a dynamic field**

- Include an Apache Solr™ dynamic field in the search index schema.

  Name the field using a wildcard at the beginning or end of the field. For example, an asterisk prefix or suffix in the field name in the schema designates a dynamic field:

  ```
  # dyna_*
  # `_s
  ```

- To define the map collection column in CQL, use the same base name (no asterisk) that you used for the field in search index schema.

  For example, use `dyna_*` in the search index schema and `dyna_` for the name of the CQL map collection.

- Use type text for the map key:

  ```
  CREATE TABLE my_dynamic_table ( 
    ... 
    dyna_ map<text, int>, 
    ... 
  );
  ```

- Using CQL, insert data into the map using the base name as a prefix or suffix in the first component of each map pair:

  ```
  { prefix_literal : literal, prefix_literal : literal, ... }
  ```
Querying data

The CQL map looks like:

```
'dyna_' : {'dyna_1' : 1, 'dyna_2' : 2, 'dyn_3' : 3}
```

DSE Search maps the dynamic field to a map collection column.

Joining cores

DSE Search supports the OS Solr query time join through a custom implementation. You can join search documents, including those having different search indexes under these conditions:

- Search indexes must have the same keyspace and same database partition key.
- Both tables that support the search indexes to be joined must be CQL-compatible.
- The type of the unique key (database key validator of the partition key) are the same in both search documents.
- The order of table partition keys and schema unique keys are the same in both search documents.

Using the simplified syntax automatically takes advantage of joins.

Simplified syntax

DataStax recommends this simplified syntax to join search indexes:

```
q={!join fromIndex=test.from}field:value
```

The custom DSE Search implementation does not use the to/from parameters that are required by OS Apache Solr™. Based on the key structure, DSE Search determines the parameters. For backward compatibility with applications, the verbose legacy syntax is also supported.

Example of using a query time join

This example creates two tables:

- The songs table uses a simple primary key: the UUID of a song.
- The primary key of the songs table is its partition key.
- The lyrics table uses a compound primary: id and song, both of type UUID.
- Both tables use the same partition key.

After joining search indexes, you can construct a single query to retrieve information about songs having lyrics that include "love".

To join the search indexes:

1. Download and unzip the file.
   This action creates `/songs` and `/lyrics` directories, schemas, and config files for indexing data in the songs and lyrics tables.

2. Start cqlsh, and then create and use a keyspace named internet.
   You can copy from the downloaded `commands.txt` file.

3. Create two tables, song and lyrics, that share the internet keyspace and use the same partition key.

   ```
cqlsh> CREATE TABLE songs (song uuid PRIMARY KEY, title text, artist text);
cqlsh> CREATE TABLE lyrics (song uuid, id uuid, words text, PRIMARY KEY (song, id));
   ```

   Both tables share the song partition key, a uuid. The second table also contains the id clustering column.
4. Insert the data from the downloaded file into the songs table.

5. Insert data into the lyrics table.
   The lyrics of songs by Big Data and John Cedrick mention love.

6. Navigate to the songs directory that you created in step 1, and take a look at the schema.xml. Navigate to the lyrics directory and take a look at the schema. Notice that the order of the unique key in the schema and the partition key of the lyrics table are the same: (song, id). Using (id, song) does not work.

```xml
<schema name="songs_schema" version="1.5">
  <types>
    <fieldType name="uuid" class="solr.UUIDField" />
    <fieldType name="text" class="solr.TextField">
      <analyzer>
        <tokenizer class="solr.StandardTokenizerFactory"/>
      </analyzer>
    </fieldType>
  </types>
  <fields>
    <field name="song" type="uuid" indexed="true" stored="true"/>
    <field name="title" type="text" indexed="true" stored="true"/>
    <field name="artist" type="text" indexed="true" stored="true"/>
  </fields>
  <defaultSearchField>artist</defaultSearchField>
  <uniqueKey>song</uniqueKey>
</schema>

<schema name="lyrics_schema" version="1.5">
  <types>
    <fieldType name="uuid" class="solr.UUIDField" />
    <fieldType name="text" class="solr.TextField">
      <analyzer>
        <tokenizer class="solr.StandardTokenizerFactory"/>
      </analyzer>
    </fieldType>
  </types>
  <fields>
    <field name="song" type="uuid" indexed="true" stored="true"/>
    <field name="id" type="uuid" indexed="true" stored="true"/>
    <field name="words" type="text" indexed="true" stored="true"/>
  </fields>
  <defaultSearchField>words</defaultSearchField>
  <uniqueKey>(song, id)</uniqueKey>
</schema>
```

7. In the songs directory, create the search index config and schema for the internet.songs table.

8. In the lyrics directory, create the search index config and schema for the internet.lyrics core, and create the search index for internet.lyrics.

9. Search for songs that have lyrics about love.

   http://localhost:8983/solr/internet.songs/select/?q={!join+fromIndex=internet.lyrics}words:love&indent=true&wt=json

The output includes two songs having the word "love" in the lyrics, one by Big Data and the other by John Cedrick:

```
"response":{"numFound":2,"start":0,"docs":[
  {
    "song":"a3e64f8f-bd44-4f28-b8d9-6938726e34d4",
    "title":"Dangerous",
```
Querying data

Recursive join support
You can nest a join query to use the result of one join as an input for another join, and another, recursively. All joined data must reside on the same partition. To embed one query in the query string of another, use the magic field name `_query_`.

Use this syntax to construct a query that recursively joins search indexes:

```
F1:V1 AND _query_:"{{join fromIndex=keyspace.table}(F2:V2
AND _query_:"{{join fromIndex=keyspace.table}(F3:V3)"}
```

Where the top level from query includes a nested join query. The nested join in this example is:

```
_query_:"{{join fromIndex=keyspace.table}(F3:V3)"
```

Like an SQL SELECT IN ... (SELECT IN ...) query, the nested join queries run first, enabling multiple nested join queries if required.

A recursive join query is not a relational join where the values from the nested join queries are returned in the results.

Example of a recursive join query
This example builds on the Solr query time join example. In the query to join songs and lyrics that contain the word “love”, embed another query to join award-winning videos using AND _query_:“award:true”.

You can copy CQL commands, Solr HTTP requests, and the query from the downloaded commands.txt file.

1. In cqlsh, create a videos table that shares the internet keyspace and uses the same partition key as the songs and lyrics tables.
   ```
cqlsh> CREATE TABLE videos (song uuid, award boolean, title text, PRIMARY KEY (song));
   ```

All three tables (songs, lyrics, videos) use the song partition key, a uuid.

2. Insert the data from the downloaded file into the videos table. The video data sets the award field to true for the videos featuring songs by Big Data and Brad Paisley.

3. Navigate to the videos directory that was created when you unzipped the downloaded file.

4. In the videos directory, post solrconfig.xml and schema.xml, and create the Search core for internet.videos.

5. Use a nested join query to recursively join the songs and lyrics documents with the videos document, and to select the song that mentions love and also won a video award.

```
http://localhost:8983/solr/internet.songs/select/?q=
{{join+fromIndex=internet.lyrics}words:love AND _query_:
{{join+fromIndex=internet.videos}award:true&indent=true&wt=json
```

Output is:

```
"response":{"numFound":1,"start":0,"docs":[
```
Support for the legacy join query

DataStax Enterprise supports using the legacy syntax that includes to and from fields in the query.

The legacy join query feature is deprecated and will be removed in a future software release.

The requirements for using the legacy syntax are:

- The tables do not use **composite partition key**.
- The query includes the force=true local parser parameter, as shown in the following example that joins mytable1 and mytable2 in mykeyspace.

Legacy syntax example

```
curl 'http://localhost:8983/solr/mykeyspace.mytable1/select/?q=
\{|!join+from=id+to=id+fromIndex=mykeyspace.mytable2+force=true\}'
```

Spatial queries with polygons require JTS

JTS (Java Topology Suite) is required to index polygon and multi-polygon shapes and perform queries using polygon shapes. Dynamic fields for spatial subfields use prefix naming conventions to enable map types to store geospatial data. DSE Search includes the Apache Solr™ Spatial4j library that adds advanced spatial types like polygons to search indexes.

**Spatial field type with JTS enabled**

For optimal indexing of multipolygon shapes, you must set useJtsMulti="false". For example:

```
<fieldType autoIndex="true" useJtsMulti="false"
class="solr.SpatialRecursivePrefixTreeFieldType" distErrPct="0.0125"
distanceUnits="kilometers" geo="true" name="WktField"
spatialContextFactory="org.locationtech.spatial4j.context.jts.JtsSpatialContextFactory"/>
```

Advanced spatial queries using spatial predicates

DSE Search supports these spatial predicates:

- Intersects
- IsWithin
- IsDisjointTo
- Contains
### Querying data

#### Examples

**Intersects**

```
fq=geo:"Intersects(-74.093 41.042 -69.347 44.558)"
```

**IsWithin**

```
fq=geo:"IsWithin(POLYGON((-10 30, -40 40, -10 -20, 40 20, 0 0, -10 30))) distErrPct=0"
```

**IsDisjointTo**

```
fq=geo:"IsDisjointTo(POLYGON((-10 30, -40 40, -10 -20, 40 20, 0 0, -10 30))) distErrPct=0"
```

**Contains**

```
fq=geo:"Contains(POLYGON((-10 30, -40 40, -10 -20, 40 20, 0 0, -10 30))) distErrPct=0"
```

### UDT query examples

You can query nested tuples and UDTs inside CQL lists and sets. A UDT facilitates handling multiple fields of related information in a table. UDTs are a specialization of tuples. In these examples, `{!tuple}` applies to both UDTs and tuples.

Selecting an entire UDT column in the CQL SELECT clause is supported. Selecting individual fields of a UDT is supported for unfrozen tuples.

#### Querying fields

```
{"tuple}address.street:sesame
```

#### Querying dynamic fields

```
<dynamField name="user.position_*/" type="text" indexed="true" stored="true"/>
{"tuple}user.position_day1:second
{"tuple}user.position_day2:first
```

#### Querying collections

```
{"tuple}user.hobbies:swim
```

#### Querying across different UDT/tuple fields

```
+{"tuple v='father.name.firstname:Sam'} +{"tuple v='mother.name.firstname:Anne'}
```

In CQL, to use a single quotation mark in a string literal, you must escape it using a single quotation mark (so you'll need to double the single quotation marks). See CQL escaping characters.
You can also use the discouraged syntax:

```
({!tuple}father.name.firstname:Sam AND {!tuple}mother.name.firstname:Anne)
```

### Querying UDT/tuple fields with several conditions

You can find a tuple that satisfies several conditions. Notice how all the conditions are on the same tuple all the time. For example:

```
{:tuple v='address.residents.field1:Alice AND address.residents.field2:Smith'}
```

You can also use the discouraged syntax:

```
{!tuple}address.residents.field1:Alice AND address.residents.field2:Smith
```

The difference in syntax specifies to search across tuples or within a tuple.

- **Across tuples:**
  
  ```
  +{!tuple v='condition1'} +{!tuple v='condition2'} +{!tuple v='conditionN'}
  ```

  searches for documents that satisfy all conditions, but are not necessarily satisfied by the same single tuple/UDT.

- **Within a tuple:**
  
  ```
  {!tuple v='condition1 AND condition2 AND conditionN'}
  ```

  searches for documents that satisfy all conditions within a single tuple/UDT.

### Querying nested tuples and UDTs

To query nested tuples and UDTs, use the same dot notation and the tuple query parser. Because UDTs are a specialization of tuples, use the tuple query parser for tuples and UDTs. In this example, the dot notation identifies `address.resident` as a UDT.

**Query for locations that have a resident with the first name Alice using the nested address.residents tuple:**

```
{:tuple}address.residents.field1:Alice
```

**Query for locations with a resident that has the first name Alice and second name Smith:**

```
+{:tuple v='address.residents.field1:Alice AND address.residents.field2:Smith'}
```

Tuples and UDTs are modelled internally as nested documents. The Apache Solr™ block join is used internally to query them. Parents are identified with the `_parent_=true` field. Children are identified with `_parent_=false`. For certain types of queries, including negative queries and empty field queries, you might need to use the `_parent_` field.
Querying data

Querying for empty firstnames

The negation (-) and inclusion (+) operators must precede the {!tuple} directive:

```
-{!tuple}_parent_:false AND user.name.firstname:* TO *
```

Negative queries

Negative queries use this syntax:

```
select * from demo where solr_query='-{!tuple}name.firstname:*
```

Negative queries with more than one condition must follow the Solr rules. Use this syntax:

```
{!tuple v='address.street:* NOT (address.street:sesame AND address.number:32)'}
```

or

```
-{!tuple v='address.street:sesame AND address.number:32'}
```

or

```
{!tuple}address.street:* NOT (address.street:sesame AND address.number:32)
```

Querying CQL collections

DSE Search supports CQL collections. In this example, you create a table containing a CQL set collection of famous quotations.

1. Start DataStax Enterprise as a DSE Search node.
2. Start cqlsh.
3. Create a keyspace and a table for a collection column and other columns, and then insert data.

```cql
CREATE KEYSPACE mykeyspace
  WITH REPLICATION = {'class':'NetworkTopologyStrategy', 'Solr':1};
USE mykeyspace;
CREATE TABLE mysolr (
  id text PRIMARY KEY,
  name text,
  title text,
  quotes set<text>
);
```

4. Download the quotations.zip file.
5. Extract the quotations.zip file, copy the insert commands, and paste each command on the cqlsh command line.
6. Run the following command:

```
$ installation_location/bin/dsetool create_core mykeyspace.mysolr
genenerateResources=true reindex=true
```

If you are recreating the mykeyspace.mysolr core, use the `reload_core` command instead of the `create_core` command.

There is no output from this command. You can search data after indexing finishes.

7. In `cqlsh`, search the indexed data to find quotes like `succ*`.

```
SELECT * FROM mykeyspace.mysolr WHERE solr_query='quotes:succ*';
```

Because you created the core using automatically generated resources, the search index config defines the request handler for using CQL for search queries.

8. Using a browser, search-indexed data using the Solr HTTP API to find titles like `succ*`.

```
http://localhost:8983/solr/mykeyspace.mysolr/
select?q=quotes%3Asucc*&wt=json&indent=on&omitHeader=on
```

```
{
  "response": {
    "numFound": 2,
    "start": 0,
    "docs": [
      {
        "id": "126",
        "title": "Success",
        "quotes": ["If A is success in life, then A equals x plus y plus z. Work is x; y is play; and z is keeping your mouth shut."],
        "name": "Albert Einstein"),
        {
        "id": "125",
        "title": "Success",
        "quotes": ["Always bear in mind that your own resolution to succeed is more important than any one thing.",
        "Better to remain silent and be thought a fool than to speak out and remove all doubt."],
        "name": "Abraham Lincoln")
    ]
  }
}
```

**Using date ranges in solr_query**

The Solr `DateRangeField` is supported in DSE Search with mapping of Solr DateRangeField to the CQL type `DateRangeType`.

The CQL type `DateRangeType` is supported for use with the latest Java driver, the DSE Python driver, and `cqlsh` commands.

**Overriding the default TimeZone (UTC) in search queries**

Specify the `TZ` parameter to overwrite the default TimeZone (UTC) that is used for adding and rounding in date math. The local rules for the specified time zone, including the start and end of daylight saving time (DST) if any, determine when each arbitrary day starts. The time zone rules impact the rounding and adding of DAYS, but also cascades to rounding of HOUR, MIN, MONTH, and YEAR. For example, specifying a different time zone changes the result:

<table>
<thead>
<tr>
<th>Date math</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016-03-10T12:34:56Z/YEAR</td>
<td>Default TZ 2016-01-01T00:00:00Z</td>
</tr>
</tbody>
</table>
## Querying data

<table>
<thead>
<tr>
<th>Date math</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>TZ=America/Los_Angeles 2016-01-01T08:00:00Z</td>
<td>2016-03-10T08:00:00Z+1DAY</td>
</tr>
<tr>
<td>Default TZ 2016-03-11T08:00:00Z</td>
<td>TZ=America/Los_Angeles 2016-03-11T07:00:00Z</td>
</tr>
</tbody>
</table>

The value of the TZ parameter can be any zone ID that is supported by the java TimeZone class.

### Primary key or ordinary column

DateRangeType can be used as a primary key or ordinary column:

```sql
CREATE TABLE taxi_trips(id int PRIMARY KEY, pickup_dropoff_range 'DateRangeType');
```

```sql
CREATE TABLE weather_sensors(weatherstation_id text, event_time 'DateRangeType', temperature text, PRIMARY KEY (weatherstation_id,event_time));
```

### CQL representation

The CQL representation uses the same syntax as Solr DateRangeField:

```sql
INSERT INTO taxi_trips(id, pickup_dropoff_range) VALUES (1, '[2017-02-02T14:57:00 TO 2017-02-02T15:10:17]');
```

```sql
INSERT INTO taxi_trips(id, pickup_dropoff_range) VALUES (2, '[2017-02-01T09:00:03 TO 2017-02-01T09:32:00.001]');
```

```sql
INSERT INTO taxi_trips(id, pickup_dropoff_range) VALUES (3, '[2017-02-03T12:10:01.358 TO 2017-02-03T12:19:57]');
```

### dateTime precision

The dateTime precision is preserved from user input. Milliseconds are displayed only when millisecond precision is provided on input.

```sql
SELECT * FROM taxi_trips;
```

```
<table>
<thead>
<tr>
<th>id</th>
<th>pickup_dropoff_range</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[2017-02-02T14:57:00Z TO 2017-02-02T15:10:17Z]</td>
</tr>
<tr>
<td>2</td>
<td>[2017-02-01T09:00:03Z TO 2017-02-01T09:32:00.001Z]</td>
</tr>
</tbody>
</table>
```
Create search index:

```
CREATE SEARCH INDEX ON taxi_trips ;
```

Select all trips from February 2017:

```
SELECT * FROM taxi_trips WHERE solr_query = 'pickup_dropoff_range:2017-02';
```

Select all trips started after 2017-02-01 12:00 PM (inclusive) and ended before 2017-02-02 (inclusive):

```
SELECT * FROM taxi_trips WHERE solr_query = 'pickup_dropoff_range:[2017-02-01T12 TO 2017-02-02]';
```

Select all trips started after 2017-02-01 12:00 PM (inclusive) and ended before 2017-02-01:23:59:59.999 (inclusive):

```
SELECT * FROM taxi_trips WHERE solr_query = 'pickup_dropoff_range:[2017-02-01T12 TO 2017-02-01]';
```

**Single point in time**

DateRangeField can represent a single point in time:

```
INSERT INTO weather_sensors (weatherstation_id, event_time, temperature) VALUES ('A1', '2017-10-02T00:00:05', '12C');
INSERT INTO weather_sensors (weatherstation_id, event_time, temperature) VALUES ('A1', '2017-10-02T00:00:10', '12C');
INSERT INTO weather_sensors (weatherstation_id, event_time, temperature) VALUES ('A1', '2017-10-02T00:00:15', '13C');
INSERT INTO weather_sensors (weatherstation_id, event_time, temperature) VALUES ('A1', '2017-10-02T00:00:20', '13C');
INSERT INTO weather_sensors (weatherstation_id, event_time, temperature) VALUES ('A1', '2017-10-02T00:00:25', '12C');
```

Select all from weather_sensors:

```
SELECT * FROM weather_sensors;
```

```
<table>
<thead>
<tr>
<th>weatherstation_id</th>
<th>event_time</th>
<th>temperature</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>2017-10-02T00:05Z</td>
<td>12C</td>
</tr>
<tr>
<td>A1</td>
<td>2017-10-02T00:10Z</td>
<td>12C</td>
</tr>
<tr>
<td>A1</td>
<td>2017-10-02T00:15Z</td>
<td>13C</td>
</tr>
<tr>
<td>A1</td>
<td>2017-10-02T00:20Z</td>
<td>13C</td>
</tr>
</tbody>
</table>
```
Querying data

Create a search index on `weather_sensors`:

```
CREATE SEARCH INDEX ON weather_sensors;
```

Select a specific point in time:

```
SELECT * FROM weather_sensors WHERE solr_query = 'event_time:[2017-10-02T00:00:10 TO 2017-10-02T00:00:20]';
```

```
weatherstation_id | event_time           | solr_query | temperature
-------------------+----------------------+------------+-------------
A1 | 2017-10-02T00:00:10Z |       null |         12C
A1 | 2017-10-02T00:00:15Z |       null |         13C
A1 | 2017-10-02T00:00:20Z |       null |         13C
```

Open bounds

`DateRangeField` can have open bounds.

Select from a point in time to an open bound:

```
SELECT * FROM weather_sensors WHERE solr_query = 'event_time:[2017-10-02T00:00:10 TO *]';
```

```
weatherstation_id | event_time           | solr_query | temperature
-------------------+----------------------+------------+-------------
A1 | 2017-10-02T00:00:25Z |       null |         12C
A1 | 2017-10-02T00:00:10Z |       null |         12C
A1 | 2017-10-02T00:00:15Z |       null |         13C
A1 | 2017-10-02T00:00:20Z |       null |         13C
```

Select from an open bound up to a point in time:

```
SELECT * FROM weather_sensors WHERE solr_query = '* TO 2017-10-02T00:00:20]';
```

```
weatherstation_id | event_time           | solr_query | temperature
-------------------+----------------------+------------+-------------
A1 | 2017-10-02T00:00:10Z |       null |         12C
A1 | 2017-10-02T00:00:15Z |       null |         13C
A1 | 2017-10-02T00:00:20Z |       null |         13C
A1 | 2017-10-02T00:00:05Z |       null |         12C
```

Select from all points in time:

```
SELECT * FROM weather_sensors WHERE solr_query = '* TO *';
```

```
weatherstation_id | event_time           | solr_query | temperature
-------------------+----------------------+------------+-------------
A1 | 2017-10-02T00:00:25Z |       null |         12C
A1 | 2017-10-02T00:00:10Z |       null |         12C
A1 | 2017-10-02T00:00:15Z |       null |         13C
A1 | 2017-10-02T00:00:20Z |       null |         13C
```
Insert an open-bounded range into a table:

```sql
INSERT INTO weather_sensors (weatherstation_id, event_time, temperature) VALUES ('A1', '[2017-10-02T00:00:25 TO *]', '12C');
SELECT * FROM weather_sensors WHERE solr_query = 'event_time:[* TO *]';
```

<table>
<thead>
<tr>
<th>weatherstation_id</th>
<th>event_time</th>
<th>solr_query</th>
<th>temperature</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>2017-10-02T00:00:05Z</td>
<td>null</td>
<td>12C</td>
</tr>
<tr>
<td>A1</td>
<td>2017-10-02T00:00:10Z</td>
<td>null</td>
<td>12C</td>
</tr>
<tr>
<td>A1</td>
<td>2017-10-02T00:00:15Z</td>
<td>null</td>
<td>13C</td>
</tr>
<tr>
<td>A1</td>
<td>2017-10-02T00:00:20Z</td>
<td>null</td>
<td>13C</td>
</tr>
<tr>
<td>A1</td>
<td>2017-10-02T00:00:05Z</td>
<td>null</td>
<td>12C</td>
</tr>
<tr>
<td>A1</td>
<td>[2017-10-02T00:00:25Z TO *}</td>
<td>null</td>
<td>12C</td>
</tr>
</tbody>
</table>

Restricted query routing

This feature is for experts only and should be used with care.

DSE Search restricted query routing is designed for applications that have a data model that supports restricting common queries to a single partition.

route.partition and route.range filter only which endpoints to query.

To restrict queries to a token or partition, use the CQL `solr_query` instead.

For example:

```sql
SELECT aid, bkt, ts, rid, mt FROM tt.accounttransactions WHERE aid=1096 AND bkt=0 AND solr_query='{"q":"*:*", "sort":"ts asc"}';
```

to filter:

`;`

Partition key routing

You can restrict routing queries to a limited number of nodes based on a list of partition keys. You can also restrict queries based on a single token range. To specify routing by partition keys, use the route.partition query parameter and set its value to one or more partition keys. DSE Search queries only the nodes that own the given partition keys. The vertical line delimiter separates components of a composite key. The comma delimiter separates different partition keys.

For example:

```sql
route.partition=k1c1|k1c2,k2c1|k2c2 . . .
```

If the actual partition key value contains a delimiter character, use a backslash character to escape the delimiter.

Examples
Querying data

You can route Solr HTTP API and CQL queries. This example shows how to use the route queries on a table with a composite partition key, where "nike" and "2" are composite key parts.


In CQL:

```
SELECT * FROM test.route WHERE solr_query='{"q" : ":*:" , "route.partition" : ["nike\|2","reebok\|2"]}'
```

Token range routing

Only use token range routing if you thoroughly understand cluster token placement. For simplicity, DataStax recommends routing queries by partition range instead of routing by token range. To specify routing by token range, use the route.range query parameter and set its value to the two token values that represent the range, separated by comma.

For example:

```
route.range=t1,t2
```

DSE Search queries only the nodes in the given token range.

**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**.

```
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**.

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

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

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

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

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

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

```sql
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

Starting with DataStax Enterprise 5.0, 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.

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

- 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.

```sql
SELECT sum(race_points) FROM cycling.cyclist_points WHERE id=e3b19ec4-774a-4d1c-9e5a-decec1e30aac;
```

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

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

```plaintext
<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>
```
Querying data

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

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

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.

1. List all the data in the table.

   ```sql
   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 presented by Sho-Air</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>UnitedHealthCare Pro Cycling Womens Team</td>
<td>Katie HALL</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>UnitedHealthCare Pro Cycling Womens Team</td>
<td>Alena AMALIUSIK</td>
<td>Amgen Tour of California Women's Race presented by SRAM - Stage 1 - Lake Tahoe</td>
</tr>
<tr>
<td>VeloCito-SRAM</td>
<td>Tricia 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.

   ```sql
   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';
   ```

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) \texttt{fLog()} created previously to retrieve data from a table \texttt{cycling.cyclist_points}.

\[
\begin{array}{|c|c|c|}
\hline
\text{id} & \text{lastname} & \text{cycling.flog(race_points)} \\
\hline
220844bf-4860-49d6-9a4b-6b5d3a79c6f6 & TIRALONGO & 0.693147 \\
e3b19ec4-774a-4d1c-9e5a-decece1e30aac & BRONZINI & 1.79176 \\
e3b19ec4-774a-4d1c-9e5a-decece1e30aac & BRONZINI & 4.31749 \\
e3b19ec4-774a-4d1c-9e5a-decece1e30aac & BRONZINI & 4.78749 \\
\hline
\end{array}
\]

**Returning the write timestamp**

A table contains a timestamp representing the date and time that a write occurred to a column. Using the \texttt{writeTime} 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 \texttt{Paolo} was written to the \texttt{firstname} column in the table \texttt{cyclist_points}.
  Use the \texttt{WRITETIME} function in a SELECT statement, followed by the name of a column in parentheses:

\[
\text{SELECT WRITETIME (firstname) FROM cycling.cyclist_points WHERE id=220844bf-4860-49d6-9a4b-6b5d3a79c6f6;}
\]

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 \texttt{What's New in Cassandra 2.2: JSON Support}.

**Retrieving all results in the JSON format**

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

\[
cqlsh:cycling> select json name, checkin_id, timestamp from checkin;
\]

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE map PUBLIC "-//OASIS//DTD DITA Map//EN" "map.dtd">
<map>
  <title>meta</title>
  <topicref href="dse-wip.ditamap" format="ditamap"/>
  <topicref href="install.ditamap" format="ditamap"/>
  <topicref href="security.ditamap" format="ditamap"/>
  <topicref href="cql.ditamap" format="ditamap"/>
  <topicref href="opscuserguide.ditamap" format="ditamap"/>
  <topicref href="studio.ditamap" format="ditamap"/>
  <topicref href="insights.ditamap" format="ditamap"/>
</map>
```
Querying data

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

```cql
select name, checkin_id, toJson(timestamp) from checkin;
```

<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-ffe8194dc9f</td>
<td>&quot;2016-08-28 21:45:10.406Z&quot;</td>
</tr>
<tr>
<td>VOSS</td>
<td>50554d6e-29bb-11e5-b345-ffe819c9f</td>
<td>&quot;2016-08-28 21:44:04.113Z&quot;</td>
</tr>
</tbody>
</table>
```
(2 rows)
```

DataStax Enterprise supports returning a JSON-formatted `timestamp` with complete time zone information.
Chapter 8. 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 interface for running CQL commands interactively. The CQL shell supports tab completion.

Synopsis

$ cqlsh [ options ] [ host_name[:port_number] ]

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>
<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 shell (cqlsh) reference

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>

--browser=""launch_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=DSE_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]
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:

```cql
SELECT *
FROM calendar
WHERE race_id = 201;
```

The returned results are shown in the standard output:

<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>201</td>
<td>2015-02-18 08:00:00.000000+0000</td>
<td>2015-02-22 08:00:00.000000+0000</td>
<td>Women's Tour of New Zealand</td>
</tr>
</tbody>
</table>

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]
Use HELP for help.
```

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.

Command line options always override the settings in a `cqlshrc` file.

You can configure these settings in the `cqlshrc` file:

- 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 and COPY 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 20: 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>
Using a sample cqlshrc file

A sample file is installed with DataStax Enterprise cqlshrc sample files. These files contain all the available settings. Some settings are commented out using a single semicolon.

DataStax Enterprise has these sample cqlshrc files:

- cqlshrc.sample: Contains all available settings.
- cqlshrc.sample.kerberos: Contains settings related to Kerberos authentication for the CQL shell connection.
- cqlshrc.sample.ssl: Contains settings related to using SSL for the CQL shell for the connection.
- cqlshrc.sample.kerberos_ssl: Contains settings related to using SSL and Kerberos authentication for the CQL shell connection.

The location of these files depend on the type of installation:

- Package installations: /etc/dse/cassandra
- Tarball installations: install_location/resources/cassandra/conf

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 is typically 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 shows a message that the file moved.
3. Remove the semicolon to uncomment an option (options must be in brackets) and the corresponding settings. This example 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 the CQL shell starts and choose an optional keyspace.

Only set a user name and password for hosts that use internal authentication. See Encrypting with SSL.

```
[authentication]
username
Example: 'example'
password
Example: 'example'
keyspace
Example: 'example'
```

Optional. Uses 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 following options apply to the cqlsh console display settings and COPY TO date parsing settings.

[ui]

color
  Shows query results with color.
  on  - use color
  off - no color

datetimeformat
  Configure the format of timestamps using Python strftime syntax.

timezone
  Display timestamps in Etc/UTC format.

float_precision, double_precision
  Sets the number of digits displayed after the decimal point for single and double precision numbers.
  Increasing these options to large numbers might produce unusual results when the values are displayed.

completekey
  Set the key for automatic completion of a cqlsh shell entry. Default: Tab key.

encoding

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
  Timeout in seconds when opening new connections.

request_timeout
  Request timeout in seconds for executing queries. Sets the number of seconds of inactivity.

factory
  For SSL, set to cqlshlib.ssl.ssl_transport_factory.

Limiting the field size

[csv]

field_size_limit
  Set to a particular field size, for example field_size_limit = 1000000000.

DSE and Native protocols

[protocol]

version
  Specify default native protocol version, which is mutually exclusive from dse_version.
  Default: 4

dse_version
  Specify default DataStax Enterprise protocol version.
  Default: 2
CQL shell (cqlsh) reference

Setting tracing timeout
Specify the wait time for tracing.

```
[tracing]
max_trace_wait
```
Maximum number of seconds to wait for a trace to complete.

Configuring SSL
Specify connection SSL settings.

For more information, see Connecting to SSL-enabled nodes using cqlsh.

```
[ssl]
certfile
validate
userkey
usercert
```
Path to the DataStax Enterprise certificate. See Connecting to SSL-enabled nodes using cqlsh.
Optional validation. Default: true.
User key must be provided when require_client_auth=true in cassandra.yaml.
User certificate must be provided when require_client_auth=true in cassandra.yaml.

Overriding SSL local settings
Overrides the 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 computer.

Setting common COPY TO and COPY FROM options
Settings common to both the COPY TO and COPY FROM commands.

```
[copy]
  option_name = value
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_name</td>
<td>Table for the copy operation.</td>
<td></td>
</tr>
<tr>
<td>column_list</td>
<td>List of columns in the table. All fields are included when no column names are specified. To omit columns, specify a column list with only the columns to include.</td>
<td></td>
</tr>
<tr>
<td>file_name, file2_name</td>
<td>CSV file name.</td>
<td></td>
</tr>
<tr>
<td>BOOLSTYLE</td>
<td>Boolean indicators for true and false. The values are case-insensitive. For example: yes, no and YES, NO are the same.</td>
<td>True, False</td>
</tr>
<tr>
<td>CONFIGFILE</td>
<td>Directory that contains the cqlshrc configuration file.</td>
<td></td>
</tr>
<tr>
<td>DATETIMEFORMAT</td>
<td>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.</td>
<td>%Y-%m-%d %H:%M:%S%z</td>
</tr>
<tr>
<td>DECIMALSEP</td>
<td>Decimal value separator.</td>
<td>. (period)</td>
</tr>
<tr>
<td>DELIMITER</td>
<td>Field separator.</td>
<td>, (comma)</td>
</tr>
</tbody>
</table>
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.
Default: None

Setting COPY TO specific options

BEGINTOKEN
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

ENDTOKEN
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
CQL shell (cqlsh) reference

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

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 and COPY FROM options

Use these options to configure table-specific settings. Create a new entry for each table. For example, to set the chunk size for cyclist names and rank:

```
[copy:cycling.cyclist_names]
chunksize = 1000
```

```
[copy:cycling.rank_by_year_and_name]
chunksize = 10000
```

```
[copy:keyspace_name.table_name]
chunks
Chunk size passed to worker processes. Default value: 1000

[copy-from:keyspace_name.table_name]
ingestrate
Approximate ingest rate in rows per second. Must be greater than the chunk size.

[copy-to:keyspace_name.table_name]
pagetimeout
Page timeout for fetching results.
```
CAPTURE
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 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>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.
Examples
Capture results to the winners text file:

CAPTURE '~/results/winners.txt'

The results directory must exist in your home directory. The winners text file is created if it does not exist.

Now capturing query output to '/Users/local_system_user/results/winners.txt'.

Execute a query that selects all cycling race winners:

SELECT *
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+l.

Synopsis

CLEAR

or

CLS

<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>
<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>
</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 23: 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 surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td><strong>{ key : value }</strong></td>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon ( : ) separates the key and the value.</td>
</tr>
</tbody>
</table>
CQL shell (cqlsh) reference

<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 (<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>

**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.

```
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 24: 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</td>
<td></td>
</tr>
<tr>
<td>QUORUM</td>
<td>Quorum of all nodes across all datacenters. Some level of failure is possible.</td>
<td>Low in multi-datacenter</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></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>Level</td>
<td>Replicas</td>
<td>Consistency</td>
<td>Availability</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------------------------------------------------------</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>
<tr>
<td>ANY</td>
<td>Closest replica, as determined by the snitch. If all replica nodes are down, write succeeds after a hinted handoff. Provides low latency and guarantees that writes never fail.</td>
<td>Lowest (WRITE)</td>
<td>Highest (WRITE)</td>
</tr>
<tr>
<td>SERIAL</td>
<td>Returns results with the most recent data including an inflight LWT (uncommitted). Commits an inflight LWT as part of the read. Writes NOT supported.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOCAL_SERIAL</td>
<td>Same as SERIAL, but confined to the datacenter. Writes NOT supported.</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:

```
CONSISTENCY QUORUM
```

Set the level to serial for LWT read requests:

```
CONSISTENCY SERIAL
```

Consistency level set to SERIAL.

Query that returns all cycling race winners:

```
SELECT * FROM cycling.race_winners;
```

The query results are as follows:

```
@ Row 1
---------------+--------------------------------------------------
race_name     | National Championships South Africa WJ-ITT (CN)
race_position | 1                                                
cyclist_name  | {firstname: 'Frances', lastname: 'DU TOUT'}

@ Row 2
---------------+--------------------------------------------------
race_name     | National Championships South Africa WJ-ITT (CN)
race_position | 2                                                
cyclist_name  | {firstname: 'Lynette', lastname: 'BENSON'}

@ Row 3
---------------+--------------------------------------------------
race_name     | National Championships South Africa WJ-ITT (CN)
race_position | 3                                                
cyclist_name  | {firstname: 'Anja', lastname: 'GERBER'}

@ Row 4
---------------+--------------------------------------------------
```
CQL shell (cqlsh) reference

<table>
<thead>
<tr>
<th>race_name</th>
<th>National Championships South Africa WJ-ITT (CN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>race_position</td>
<td>4</td>
</tr>
<tr>
<td>cyclist_name</td>
<td>{firstname: 'Ame', lastname: 'VENTER'}</td>
</tr>
</tbody>
</table>

0 Row 5

<table>
<thead>
<tr>
<th>race_name</th>
<th>National Championships South Africa WJ-ITT (CN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>race_position</td>
<td>5</td>
</tr>
<tr>
<td>cyclist_name</td>
<td>{firstname: 'Danielle', lastname: 'VAN NIEKERK'}</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 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>&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>

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.

table_name
Table for the copy operation.

column_list
List of columns in the table. All fields are included when no column names are specified. To omit
columns, specify a column list with only the columns to include.

file_name, file2_name
CSV file name.

BOOLSTYLE
CQL shell (cqlsh) reference

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
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: , (comma)

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.
Default: None

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

ENDTOKEN
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 REPLICATION = {
     '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');
   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');
   ```
CQL shell (cqlsh) reference

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

Export data from the `cyclist_name` table

1. Export only the `id` and `lastname` columns from the `cyclist_name` table to a CSV file:

```
COPY cycling.cyclist_name (id,lastname)
TO '../cyclist_lastname.csv' WITH HEADER = TRUE;
```

The `cyclist_lastname.csv` file is created in the directory above the current working directory (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:

```
 id | firstname | lastname
------|-----------|----------
 (0 rows)
```

5. Import the cyclist first names:

```
COPY cycling.cyclist_name (id,firstname)
```
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:

```
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-eb95-4bb4-8685-6e66e994ca</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>220844bf-4860-49d6-9a4b-6b5d3a79cbfb</td>
<td>Paolo</td>
<td>null</td>
</tr>
<tr>
<td>6ab09b0ec-e89e-48d9-a5f8-97e6b4c9b47</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:

```
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:

```
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-b900-859!2fda9bb0</td>
<td>Alex</td>
<td>FRAME</td>
</tr>
<tr>
<td>fb372533-eb95-4bb4-8685-6e66e994ca</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>6ab09b0ec-e89e-48d9-a5f8-97e6b4c9b47</td>
<td>Steven</td>
<td>KRUIKWIJK</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]`:

   ```
   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:

   ```
   [copy] e7cd5752-bc0d-4157-a80f-7523add8dbcd,Anna,VAN DER BREGGEN
   [copy] .
   ```

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:

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

   ```
   id                                   | firstname | lastname
   --------------------------------------+-----------+-----------------
   e7cd5752-bc0d-4157-a80f-7523add8dbcd |      Anna | VAN DER BREGGEN
   ```

   (1 rows)

---

**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>
<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>

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.

- **table_name**
  - Table for the copy operation.
- **column_list**
  - List of columns in the table. All fields are included when no column names are specified. To omit columns, specify a column list with only the columns to include.
- **file_name, file2_name**
  - CSV file name.
- **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**
  - 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.
Default: None

**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:

```sql
CREATE KEYSPACE cycling
WITH REPLICATION = {
  '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');

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-6b5d3a79c8b', 'TIRALONGO', 'Paolo');

INSERT INTO cycling.cyclist_name (id, lastname, firstname)
VALUES ('6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47', 'KRUWKSWIJK', 'Steven');

INSERT INTO cycling.cyclist_name (id, lastname, firstname)
VALUES ('fb372533-eb95-4bb4-8685-6ef61e994ca', '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:

```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].
2. Copy the **id** and **firstname** to a different CSV file named **cyclistFirstname.csv**:

```sql
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:

```sql
TRUNCATE cycling.cyclist_name;
```

4. Verify that there are no rows:

```sql
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:

```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:

```
<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-eb95-4bb4-8685-6ef61e994ca9</td>
<td>Michael</td>
<td>null</td>
</tr>
<tr>
<td>5b6962dd-3f90-4c93-8f61-eabfa4a803e2</td>
<td>Marianne</td>
<td>null</td>
</tr>
</tbody>
</table>
```
7. Import the last names:

COPY cycling.cyclist_name (id, lastname) FROM '../cyclist_lastname.csv' WITH HEADER = TRUE;

The records are imported but no new records 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:

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>569b3311-3b8b-4d99-8962-7d93d9749262</td>
<td>Alex</td>
<td>FRAME</td>
</tr>
<tr>
<td>e7cd5752-bc0d-4157-a80f-7523add8dbcd</td>
<td>Anna</td>
<td>VAN DER BREGGEN</td>
</tr>
<tr>
<td>220844bf-4860-49d6-9a4b-6b5d3a79cbfb</td>
<td>Paolo</td>
<td>null</td>
</tr>
<tr>
<td>6ab09bec-e68e-48d9-a5f8-97e6fb49b47</td>
<td>Steven</td>
<td>null</td>
</tr>
</tbody>
</table>

Copy data from standard input to a table.

1. Clear the data from the cyclist_name table:

TRUNCATE cycling.cyclist_name;

2. Start the copy input operation using the FROM STDIN option:

COPY cycling.cyclist_name FROM STDIN;

The line prompt changes to [copy]:

Using 7 child processes

Starting copy of cycling.cyclist_name with columns [id, firstname, lastname].
[Use . on a line by itself to end input]
[copy]

3. Next to the [copy] prompt, enter the field values in a common-separated list; on the last line of data, enter a period:

[copy] e7cd5752-bc0d-4157-a80f-7523add8dbcd,Anna,VAN DER BREGGEN
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>
```

**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 27: 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>
</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='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>

### 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:

```
USE cycling; DESC AGGREGATES;
```

`average(int)`

Show the definition of the cycling average aggregate:

```
DESC AGGREGATE cycling.average
```

The result is an executable CQL statement:

```
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 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>
</tbody>
</table>
### Syntax conventions

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</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='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

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 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>
</tbody>
</table>
### Syntax conventions

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</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></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><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>' &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>

### 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.
CQL shell (cqlsh) reference

- 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 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><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; &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>

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:

```cql
USE cycling; DESC FUNCTIONS
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flog(double)</td>
<td>avgfinal(frozen&lt;tuple&lt;int, bigint&gt;&gt;, int)</td>
</tr>
<tr>
<td>avgstate(frozen&lt;tuple&lt;int, bigint&gt;&gt;, int)</td>
<td>left(text,int)</td>
</tr>
</tbody>
</table>

Show the definition of the cycling flog function:

```cql
DESC FUNCTION cycling.flog
```

```cql
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

```cql
DESCRIBE INDEX [keyspace_name.]index_name
```

Table 31: 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>
</tbody>
</table>
### Syntax conventions

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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

#### Create an index:

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

#### Show the definition of the index:

```cql
DESC INDEX cycling.nationality_idx
```

#### Output:

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

### DESCRIBE KEYSASPE

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 KEYSASPE | KEYSASPE keyspace_name
```

### 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><strong>italics</strong></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 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>

**KEYSPACES**
All keyspaces.

**KEYSPACE**
One keyspace.

**keyspace_name**
Name of the keyspace.

**Examples**
Show all keyspaces:

```
DESC KEYSPACES
```

All the keyspaces on the cluster are listed:

```
dse_system_local  cycling  system  keyspace1  system_traces
dse_security      system_schema  dse_leases  system_distributed  dse_perf
solr_admin        system_auth  dse_audit  test  dse_system
```

**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 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></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.

**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 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 SEARCH INDEX**

Shows the definition of specified search index in XML format.

Command available only on DSE Search nodes. Running search index management commands on large datasets can take longer than the CQLSH default timeout of 10 minutes. Increase the **CQLSH client timeout** as required.

**Synopsis**

```
DESCRIBE ( PENDING | ACTIVE )
SEARCH INDEX ( SCHEMA | CONFIG )
ON [keyspace_name.]table_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>'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>Syntax conventions</td>
<td>Description</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=&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>

**EBNF**

**EBNF syntax:**

```plaintext
describeSearchIndex ::= 'DESCRIBE' ( 'PENDING' | 'ACTIVE' ) 'SEARCH' 'INDEX'
tableName ::= (keyspace '.')? Table

Railroad diagram:

Figure 11: describeSearchIndex
ACTIVE
Index configuration resources that are currently applied.

PENDING
Index configuration resources that pending but not yet applied.

SCHEMA
Show the fields to index in Solr and types associated with them. These fields map to database columns.

CONFIG
Show the configuration information for the index.

keyspace_name
Name of the keyspace.

table_name
Name of the table.

Examples
Show the active index configuration for wiki.solr:

```cql
DESCRIBE ACTIVE SEARCH INDEX CONFIG ON wiki.solr;
```

The returned results are in XML.
Show the pending index schema for wiki.solr:

DESCRIBE PENDING SEARCH INDEX SCHEMA ON wiki.solr;

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 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>

TABLES

All tables.

TABLE

One table.
**keyspace_name**
Name of the keyspace.

**table_name**
Name of the table.

**Examples**
Show a list of tables in the system keyspace:

```cql
USE system; DESC TABLES
```

A list of all the tables in the keyspace is returned:

<table>
<thead>
<tr>
<th>repairs</th>
<th>view_builds_in_progress</th>
<th>paxos</th>
<th>transferred_ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>available_ranges</td>
<td>peers</td>
<td>size_estimates</td>
<td></td>
</tr>
<tr>
<td>batches</td>
<td>compaction_history</td>
<td>built_views</td>
<td></td>
</tr>
<tr>
<td>prepared_statements</td>
<td>sstable_activity</td>
<td>range_xfers</td>
<td></td>
</tr>
<tr>
<td>&quot;IndexInfo&quot;</td>
<td>peer_events</td>
<td>local</td>
<td></td>
</tr>
</tbody>
</table>

Show the CQL for the cycling calendar table:

```cql
DESC cycling.calendar
```

A complete CQL table description is returned, which can be used to recreate the table:

```cql
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 nodesync = {'enabled': 'true'}
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.
CQL shell (cqlsh) reference

To select a keyspace, use the **USE** command.

**Synopsis**

```
DESCRIBE TYPES [ keyspace_name ] | TYPE [keyspace_name.]udt_name
```

**Table 36: 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>italic</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=&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>

**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 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>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 (<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>

### 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:

```sql
EXECUTE AS read_race;
```

Executing queries as read_race.

Perform a query:

```sql
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:

```sql
EXECUTE AS
```

Disabling proxy execution

### 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.
**Synopsis**

EXPAND [ 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><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=&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>

**ON**

Enable expand.

**OFF**

Disable expand.

**Examples**

View rows vertically:

EXPAND ON

Select all rows from the cycling race winners table:

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

Each field is shown in a vertical row table:

```
@ Row 1
---------------------------------|
race_name  | National Championships South Africa WJ-ITT (CN)
```
### Race Results

<table>
<thead>
<tr>
<th>race_name</th>
<th>National Championships South Africa WJ-ITT (CN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>race_position</td>
<td>National Championships South Africa WJ-ITT (CN)</td>
</tr>
<tr>
<td>race_position</td>
<td>National Championships South Africa WJ-ITT (CN)</td>
</tr>
<tr>
<td>race_position</td>
<td>National Championships South Africa WJ-ITT (CN)</td>
</tr>
<tr>
<td>race_position</td>
<td>National Championships South Africa WJ-ITT (CN)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cyclist_name</th>
<th>{firstname: 'Frances', lastname: 'DU TOUT'}</th>
</tr>
</thead>
<tbody>
<tr>
<td>cyclist_name</td>
<td>{firstname: 'Lynette', lastname: 'BENSON'}</td>
</tr>
<tr>
<td>cyclist_name</td>
<td>{firstname: 'Anja', lastname: 'GERBER'}</td>
</tr>
<tr>
<td>cyclist_name</td>
<td>{firstname: 'Ame', lastname: 'VENTER'}</td>
</tr>
<tr>
<td>cyclist_name</td>
<td>{firstname: 'Danielle', lastname: 'VAN NIEKERK'}</td>
</tr>
</tbody>
</table>

Turn off vertical expansion:

```
EXPAND OFF
```

**EXIT**

Terminates the CQL shell.

### Syntax Summary

**Synopsis**

```
EXIT
```

### Table 39: 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>
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 40: 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 (<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

Log in as the cycling administrator:

```cql
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 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>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>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>

---

**ON**

Enables paging.

**OFF**

Disables paging. Causes subsequent result sets to be displayed in their entirety.

**page_size**

Number of lines in each page.

**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**

```plaintext
SERIAL CONSISTENCY [ consistency_level ]
```

**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>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=&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>

**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:

```
<table>
<thead>
<tr>
<th>applied</th>
<th>id</th>
<th>firstname</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>e7ae5cf3-d358-4d99-b900-85902fda9bb0</td>
<td>Alex</td>
<td>FRAME</td>
</tr>
</tbody>
</table>
```

**SHOW**

Shows this information depending on the options selected:

- Software version.
- Current session node.
- Tracing session details captured in the past 24 hours.
CQL shell (cqlsh) reference

Synopsis

| SHOW VERSION | HOST | SESSION tracing_session_id |

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>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>'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 (\n empID int,\n deptID int,\n first_name varchar,\n last_name varchar,\n PRIMARY KEY (empID, deptID)\n);</td>
<td>12:19:52,372</td>
<td>127.0.0.1</td>
<td>153</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.
CQL shell (cqlsh) reference

Synopsis

SOURCE 'file_name'

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></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>

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 all CQL statements in the current CQLSH session. Tracing can show tracing input from any or 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.
To enable or disable tracing for the entire cluster, run `nodetool settraceprobability` for each node. For information about probabilistic tracing, see `nodetool settraceprobability`.

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

**Synopsis**

```shell
TRACING [ ON | OFF ]
```

**Table 45: 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><code>{}</code></td>
<td>Optional. Square brackets <code>{}</code> surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td><code>()</code></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><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 <code>‘‘</code> 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 <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>

**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:

```
Tracing session: 9b378c70-b114-11e6-89b5-b7fad52e1885

<table>
<thead>
<tr>
<th>activity</th>
<th>timestamp</th>
<th>source</th>
<th>source_elapsed</th>
<th>client</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute 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-eb95-4db4-8685-6ef61e994c0a</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-6b5d3a79cbf6</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

<table>
<thead>
<tr>
<th>activity</th>
<th>timestamp</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute CQL3 query</td>
<td>2016-11-22 16:45:02.212000</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Parsing SELECT * FROM cycling.cyclist_name ;</td>
<td>2016-11-22 16:45:02.212000</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Preparing statement</td>
<td>2016-11-22 16:45:02.212000</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Computing ranges to query</td>
<td>2016-11-22 16:45:02.212000</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Submitting range requests on 257 ranges with</td>
<td>2016-11-22 16:45:02.212000</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>a concurrency of 257 (0.3 rows per range</td>
<td>2016-11-22 16:45:02.212000</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>expected)</td>
<td>2016-11-22 16:45:02.212000</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Submitted 1 concurrent range</td>
<td>2016-11-22 16:45:02.212000</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>Executing seq scan across 1 sstables for</td>
<td>2016-11-22 16:45:02.212000</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>(min(-9223372036854775808),</td>
<td>2016-11-22 16:45:02.212000</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>min(-9223372036854775808)) [ReadStage-2]</td>
<td>2016-11-22 16:45:02.212000</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>3130</td>
<td>127.0.0.1</td>
<td></td>
</tr>
<tr>
<td>Read 6 live</td>
<td>2016-11-22 16:45:02.212000</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>3928</td>
<td>127.0.0.1</td>
<td></td>
</tr>
<tr>
<td>Request complete</td>
<td>2016-11-22 16:45:02.216252</td>
<td>127.0.0.1</td>
</tr>
<tr>
<td>4252</td>
<td>127.0.0.1</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 9. Cassandra Query Language (CQL) reference

Provides a brief description and syntax for CQL statements, types, operators, and functions supported by the DataStax Enterprise database.

Introduction

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

This reference covers CQL statements based on the specification 3.4.5 with additional DSE Search and security-related enhancements.

Data types

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

String types

Wrap strings in single quotes or double dollar signs (\$\$) in INSERT, UPDATE, and the SELECT statement WHERE clause.

When using single quotes, additional escaping is required for field values that contain single quotes or reserved characters, see Escaping characters.

For example, to insert a comment into a text field that contains a single quote and emoji:

```
INSERT INTO cycling.comments (id, created_at, comment)
VALUES (e7ae5cf3-d358-4d99-b900-85902fda9bb0, currentTimestamp(),
        $$$ It's pouring rain, race should have been postponed :'( $$
    );
```

ascii

US-ASCII characters.

text

UTF-8 encoded string.

varchar

UTF-8 encoded string.

Numeric types

Enter numeric types in plain text in the INSERT, UPDATE, and the SELECT statement WHERE clause.

For example, age = 31.

Integers

tinyint

8-bit signed integer.
smallint 16-bit signed integer.
int 32-bit signed integer.
bигint 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 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' versus 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 the 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.

\[ beginning_date TO end_date \]

Insert the custom ranges between square brackets. For example:

- 2018-01 – Beginning of the first day to the end of the last day in January 2018.
- 2018-01T15 – Range includes hours of the day. 1500 to before 1600 (3pm to 4pm).
- [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 until the end of the first day of November.
- [* TO 2018-01-31] – From the earliest representable time through to the end of the day on 2018-01-31.

If you specify a date instance using a date function, like currentDate(), you get the first millisecond of that day, not the entire day’s range.
The data type name is case sensitive. Use single quotes to specify this type in CQL statement. For example, CREATE TABLE test.example (id UUID, daterange 'DateRangeField');.

duration
Encoded as these 3 signed integers of variable lengths, where the integers represent the number of:

1. Months
2. Days
3. Nanoseconds

The number of months and days are decoded as 32-bit integers. The number of nanoseconds is decoded as a 64-bit integer. Provide the duration value using one of these formats:

- Duration format: yymowwdwnmsusns. For example, 12h30m. The units are:
  # y - years (12 months)
  # mo - months (1 month)
  # w - weeks (7 days)
  # d - days (1 day)
  # h - hours (3,600,000,000,000 nanoseconds)
  # m - minutes (60,000,000,000 nanoseconds)
  # s - seconds (1,000,000,000 nanoseconds)
  # ms - milliseonds (1,000,000 nanoseconds)
  # us or μs - microseconds (1000 nanoseconds)
  # ns - nanoseconds (1 nanosecond)

- ISO 8601 alternative format: P[YYYY]-[MM]-[DD]T[hh]:[mm]:[ss]

The PRIMARY KEY does not support duration type because it is not possible to determine if 1mo is greater than 29d without a date context.

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 millisecond (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 boolean 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.

DseExecutorStateType
Reserved type used for Spark clients on analytic workload nodes.

inet
IP address string in IPv4 or IPv6 format.

Geo-spatial types

PointType
Contains two coordinate values for latitude and longitude. See Geospatial queries for Point and LineString for details on entering point information.

LineStringType
Comma separate list of points. See Geospatial queries for Point and LineString for details on entering linestring information.

PolygonType
Set of two linestrings.

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 a `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 or 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. Only supports replacing the entire set using **INSERT** and **UPDATE**.

**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<text, text>, boolean>` and also be specified as a data type of another collection type, for example `set<tuple<text, inet>>`.

**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**

`Deprecation` 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 subseconds; 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:

- (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 subseconds; 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:

<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>yyyymm-dd HH:mmXX</td>
<td>2018-04-26 12:59Z</td>
<td>2018-04-26 14:59 +02:00</td>
<td>2018-04-26 05:59-07:00</td>
</tr>
<tr>
<td>Format</td>
<td>Date</td>
<td>Time</td>
<td>Offset</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------------</td>
<td>-----------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>yyyy-mm-dd'T'HH:MM</td>
<td>2018-04-26</td>
<td>05:59 GMT-07:00</td>
<td>2018-04-26T05:59 GMT</td>
</tr>
<tr>
<td>yyyy-mm-dd'T'HH:MMX</td>
<td>2018-04-26</td>
<td>05:59 GMT-07:00</td>
<td>2018-04-26T05:59 GMT</td>
</tr>
<tr>
<td>yyyy-mm-dd'T'HH:MM:SS</td>
<td>2018-04-26</td>
<td>05:59 GMT-07:00</td>
<td>2018-04-26T05:59 GMT</td>
</tr>
<tr>
<td>yyyy-mm-dd'T'HH:MM:SS</td>
<td>2018-04-26</td>
<td>05:59 GMT-07:00</td>
<td>2018-04-26T05:59 GMT</td>
</tr>
<tr>
<td>yyyy-mm-dd'T'HH:MM:SS</td>
<td>2018-04-26</td>
<td>05:59 GMT-07:00</td>
<td>2018-04-26T05:59 GMT</td>
</tr>
<tr>
<td>yyyy-mm-dd'T'HH:MM:SS</td>
<td>2018-04-26</td>
<td>05:59 GMT-07:00</td>
<td>2018-04-26T05:59 GMT</td>
</tr>
<tr>
<td>yyyy-mm-dd'T'HH:MM:SS</td>
<td>2018-04-26</td>
<td>05:59 GMT-07:00</td>
<td>2018-04-26T05:59 GMT</td>
</tr>
<tr>
<td>yyyy-mm-dd'T'HH:MM:SS</td>
<td>2018-04-26</td>
<td>05:59 GMT-07:00</td>
<td>2018-04-26T05:59 GMT</td>
</tr>
<tr>
<td>yyyy-mm-dd'T'HH:MM:SS</td>
<td>2018-04-26</td>
<td>05:59 GMT-07:00</td>
<td>2018-04-26T05:59 GMT</td>
</tr>
</tbody>
</table>

### Solr field type reference for DSE Search

CQL data is parsed into DataStax Enterprise Search indexes using a corresponding Solr field type. The index schema for each table must contain the Solr field type definition along with optional advanced processing instructions.
Cassandra Query Language (CQL) reference

See compatibility matrix for CQL to Solr type mappings.

Trie field types

**TrieDateField**
Date field for Lucene TrieRange processing, supports indexing negative date. For example: -28011-12-02T00:00:00.002Z. To insert negative dates for the CQL timestamp, insert an epoch time in milliseconds. The TimestampType does not accept a textual representation of negative dates.

```sql
ALTER SEARCH INDEX SCHEMA ON table_name
ADD types.fieldType[@class='org.apache.solr.schema.TrieDateField',
  @name='TrieDateField'];
```

Results in the following schema XML:

```xml
<fieldType class="org.apache.solr.schema.TrieDateField" name="TrieDateField"/>
```

**TrieDoubleField**

```sql
ALTER SEARCH INDEX SCHEMA ON table_name
ADD types.fieldType[@class=org.apache.solr.schema.TrieDoubleField',
  @name='TrieDoubleField'];
```

Results in the following schema XML:

```xml
<fieldType class="org.apache.solr.schema.TrieDoubleField" name="TrieDoubleField"/>
```

**TrieFloatField**

```sql
ALTER SEARCH INDEX SCHEMA ON table_name
ADD types.fieldType[@class='org.apache.solr.schema.TrieFloatField',
  @name='TrieFloatField'];
```

Results in the following schema XML:

```xml
<fieldType class="org.apache.solr.schema.TrieFloatField" name="TrieFloatField"/>
```

**TrieIntField**

```sql
ALTER SEARCH INDEX SCHEMA ON table_name
ADD types.fieldType[@class='org.apache.solr.schema.TrieIntField',
  @name='TrieIntField'];
```

Results in the following schema XML:

```xml
<fieldType class="org.apache.solr.schema.TrieIntField" name="TrieIntField"/>
```

**TrieLongField**

```sql
ALTER SEARCH INDEX SCHEMA ON table_name
ADD types.fieldType[@class='org.apache.solr.schema.TrieLongField',
  @name='TrieLongField'];
```

Results in the following schema XML:

```xml
<fieldType class="org.apache.solr.schema.TrieLongField" name="TrieLongField"/>
```
ADD types.fieldType[@class='org.apache.solr.schema.TrieLongField',
@name='TrieLongField'];

Results in the following schema XML:

```
<fieldType class="org.apache.solr.schema.TrieLongField" name="TrieLongField"/>
```

String field types

**AsciiStrField**

Converts a CQL ascii into a standard Solr StrField.

```
ALTER SEARCH INDEX SCHEMA ON table_name
ADD types.fieldType[@class='com.datastax.bdps.search.solr.core.types.AsciiStrField',
@name='AsciiStrField'];
```

Results in the following Schema XML:

```
<fieldType class="com.datastax.bdp.search.solr.core.types.AsciiStrField"
name="AsciiStrField"/>
```

**VarIntStrField**

Define with the DataStax class to convert a CQL varint.

```
ALTER SEARCH INDEX SCHEMA ON table_name
ADD types.fieldType[@class='com.datastax.bdp.search.solr.core.types.VarIntStrField',
@name='VarIntStrField'];
```

Results in the following schema XML:

```
<fieldType class="com.datastax.bdp.search.solr.core.types.VarIntStrField"
name="VarIntStrField"/>
```

**BinaryField**

```
ALTER SEARCH INDEX SCHEMA ON table_name
ADD types.fieldType[@class='org.apache.solr.schema.BinaryField',
@name='BinaryField'];
```

Results in the following Schema XML:

```
<fieldType class="org.apache.solr.schema.BinaryField" name="BinaryField"/>
```

**BoolField**

Due to SOLR-7264, setting docValues to true on a boolean field in the Solr schema does not work. A workaround for boolean docValues is to use 0 and 1 with a TrieIntField.

```
ALTER SEARCH INDEX SCHEMA ON table_name
ADD types.fieldType[@class='org.apache.solr.schema.TrieIntField',
@name='TrieIntField'];
```
ADD types.fieldType[@class='org.apache.solr.schema.BoolField', @name='BoolField'];

Results in the following Schema XML:

```xml
<fieldType class="org.apache.solr.schema.BoolField" name="BoolField"/>
```

**SimpleDateField**

Define with the DataStax class to convert a CQL date field into a compatible Solr date field.

```sql
ALTER SEARCH INDEX SCHEMA ON table_name
ADD
types.fieldType[@class='com.datastax.bdp.search.solr.core.types.SimpleDateField', @name='SimpleDateField'];
```

Results in the following Schema XML:

```xml
<fieldType class="com.datastax.bdp.search.solr.core.types.SimpleDateField"
    name="SimpleDateField"/>
```

**TextField**

```sql
ALTER SEARCH INDEX SCHEMA ON table_name
ADD types.fieldType[@class='org.apache.solr.schema.TextField', @name='TextField'];
```

To perform searches containing wildcard characters like % in TextField data, do not use the Lucene StandardAnalyzer class. Instead, define the tokenizer and filters for use in a multiterm search.

**UUIDField**

UUIDField is a type 1 UUID data type, which includes the time of its generation. Values are sorted by conflict-free timestamps. For example, use the TimeUUID 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. To find data mapped from a TimeUUID to a UUIDField, search for the entire UUID value, not just its time component.

```sql
ALTER SEARCH INDEX SCHEMA ON table_name
ADD types.fieldType[@class='org.apache.solr.schema.UUIDField', @name='UUIDField'];
```

Results in the following Schema XML:

```xml
<fieldType class="org.apache.solr.schema.UUIDField" name="UUIDField"/>
```

**Operators**

**Arithmetic operators**

CQL supports these operators:

- Use the minus symbol for either the unary negate operand or subtraction.
- Use the plus symbol for addition.
- Use the asterisk symbol for multiplication.
- Use the forward slash symbol for division.
- Use the percent symbol for division.

---

CQL for DataStax Enterprise 6.7 Latest DSE version Latest 6.7 patch: 6.7.7
Use the percent symbol to return the remainder from division.
The *, /, and % operators have a higher precedence than the + and - operators. Higher precedence operators are evaluated before lower precedence operators. If two operators in an expression have the same precedence, they are evaluated from left to right based on the operator position in the expression.

Return types
The return type of an arithmetic expression is based on the types of the input operands. In the following table, left and right refer to the type of the operand on the left and right of the operator. For example, in the expression a + b, a is the left operand and b is the right operand; if a is an int operand and b is a float operand, a float is returned.

<table>
<thead>
<tr>
<th></th>
<th>right</th>
<th>tinyint</th>
<th>smallint</th>
<th>int</th>
<th>bigint</th>
<th>counter</th>
<th>float</th>
<th>double</th>
<th>varint</th>
<th>decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tinyint</td>
<td>tinyint</td>
<td>smallint</td>
<td>int</td>
<td>bigint</td>
<td>-bigint</td>
<td>float</td>
<td>double</td>
<td>varint</td>
<td>decimal</td>
<td></td>
</tr>
<tr>
<td>smallint</td>
<td>smallint</td>
<td>smallint</td>
<td>int</td>
<td>bigint</td>
<td>bigint</td>
<td>float</td>
<td>double</td>
<td>varint</td>
<td>decimal</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>int</td>
<td>int</td>
<td>bigint</td>
<td>bigint</td>
<td>float</td>
<td>double</td>
<td>varint</td>
<td>decimal</td>
<td></td>
</tr>
<tr>
<td>bigint</td>
<td>bigint</td>
<td>bigint</td>
<td>bigint</td>
<td>bigint</td>
<td>bigint</td>
<td>double</td>
<td>double</td>
<td>varint</td>
<td>decimal</td>
<td></td>
</tr>
<tr>
<td>counter</td>
<td>bigint</td>
<td>bigint</td>
<td>bigint</td>
<td>bigint</td>
<td>bigint</td>
<td>double</td>
<td>double</td>
<td>varint</td>
<td>decimal</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>float</td>
<td>float</td>
<td>double</td>
<td>double</td>
<td>float</td>
<td>double</td>
<td>decimal</td>
<td>decimal</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>double</td>
<td>double</td>
<td>double</td>
<td>double</td>
<td>double</td>
<td>double</td>
<td>decimal</td>
<td>decimal</td>
<td></td>
</tr>
<tr>
<td>varint</td>
<td>varint</td>
<td>varint</td>
<td>decimal</td>
<td>decimal</td>
<td>decimal</td>
<td>decimal</td>
<td>decimal</td>
<td>decimal</td>
<td>decimal</td>
<td></td>
</tr>
<tr>
<td>decimal</td>
<td>decimal</td>
<td>decimal</td>
<td>decimal</td>
<td>decimal</td>
<td>decimal</td>
<td>decimal</td>
<td>decimal</td>
<td>decimal</td>
<td>decimal</td>
<td></td>
</tr>
</tbody>
</table>

Dates and times
A time duration can be added (+) or subtracted (-) from a timestamp or a date to create a new timestamp or date. For example, the following query returns cycling races that occur one day before the supplied date:

```
SELECT *
FROM cycling.race_times
WHERE race_date = '2017-04-15' - 1d;
```

Multiple columns
Multiple columns can be used in an expression, as shown in the following queries:

```
SELECT (emp_id + dept_id) / age
FROM cycling.mechanic;

SELECT emp_id + dept_id / age
FROM cycling.mechanic;
```

CQL native functions
CQL supports several functions that transform a column value into a new value.

In addition, users can define functions and aggregates.

Scalar functions
The DataStax Enterprise database supports the following native functions.
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.

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 46: 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>
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 then) 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.

**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.

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](https://datastax.github.io/cassandra/doc/latest/core/configuration/configuration-keyspace.html#dcassandratl dov).

**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.

**Date and time functions**

**Current date and time**

Return the (current) system time of the coordinator node in a CQL type format. These functions have no arguments.

**currentDate**
Cassandra Query Language (CQL) reference

- **currentTime**
  - Returns the system time in **date** format.

- **currentTimestamp**
  - Returns the system time in **timestamp** format.

- **currentTimeUuid**
  - Returns the system time in **timeuuid** format.

- **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.
  - 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.

- **floor**
  - Rounds date and time to the nearest value.
  - **timestamp**
    - `(timestamp, duration [, start_timestamp])`
    - If the **start_timestamp** is not used, then the start timestamp is January 1, 1970 00:00:00.000 GMT.
  - **timeuuid**
    - `floor(timeuuid, duration [, start_timestamp])`
    - If the **start_timestamp** is not used, then the start timestamp is January 1, 1970 00:00:00.000 GMT.
  - **date**
    - `floor(date, duration [, start_date])`
    - If the **start_date** is not used, then the start date is January 1, 1970 GMT.
  - **time**
    - `floor(time, duration[, start_time])`
    - If the **start_time** is not used, then the start time is 00:00:00[000000000].

Use the duration syntax as follows:
- **N y** - Number of years
- **N mo** - Number of months
- **N w** - Number of weeks
- **N d** - Number of days
- **N h** - Number of hours
- **N m** - Number of minutes
- **N s** - Number of seconds
- **N ms** - Number of milliseconds
Cassandra Query Language (CQL) reference

- **N us or µs** - Number of microseconds
- **N ns** - Number of nanoseconds

**now**

Alias of `currentTimeUuid`.

**TimeUuid calculations**

**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')`.

**Date and time conversion**

Converts the data in the defined argument from a **timestamp, date**, and **timeuuid** to another type.

**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 )
```

**Blob conversion**

Convert CQL native types to binary (blob).

```
blobAs
```
Cassandra Query Language (CQL) reference

Converts the target column or literal (enclose strings in single quotes) from a blob to the specified type.

```
blobAsType(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`.

**AsBlob**

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.

**Deprecated functions**

The following functions are supported for backward compatibility only.

- **dateOf**
  Similar to `toTimestamp`.
- **unixTimestampOf**
  Similar to `toUnixTimestamp`

**CQL native aggregates**

Aggregate functions work on a set of rows matching a `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 `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 `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 `CREATE AGGREGATE`.

**Synopsis**

```
aggregate_name(column_name)
```

Where the system aggregate names are listed below:

- **AVG**
  `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 `tinyint`, `smallint`, `int`, `bigint`, `decimal`, `float`, and `double`.

- **COUNT**
  `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 CQL data types columns.
**MIN(column_name)**: Provides the smallest value. Null values are ignored. Only works on numeric columns, that is tinyint, smallint, int, bigint, decimal, float, and double.

**MAX**

**MAX(column_name)**: Provides the largest value. Null values are ignored. Only works on numeric columns, that is tinyint, smallint, int, bigint, decimal, float, and double.

**SUM(column_name)**: Provides the total of the target column; nulls are ignored. Only works on numeric columns, that is tinyint, smallint, int, bigint, decimal, float, and double.

**Examples**
The following examples show how to use aggregates using the cyclist examples.

### Find Average (AVG)

Get average time in seconds for the team:

```cql
SELECT AVG(cyclist_time_sec) AS Average
FROM cycling.team_average
WHERE team_name = 'UnitedHealthCare Pro Cycling Womens Team';
```

Results:

### Find Count (COUNT)

Find the number rows for the United Health Care Pro Cycling Women’s Team:

```cql
SELECT COUNT(cyclist_name) AS Row_Count
FROM cycling.team_average
WHERE team_name = 'UnitedHealthCare Pro Cycling Womens Team';
```

Results:

### Find lowest value (MIN)

Find the slowest time recorded for the United Health Care Pro Cycling Women’s Team:

```cql
SELECT MIN(cyclist_time_sec) AS Fastest
FROM cycling.team_average
WHERE team_name = 'UnitedHealthCare Pro Cycling Womens Team';
```

Results:

### Find highest value (MAX)

Find the fastest time recorded for the United Health Care Pro Cycling Women’s Team:

```cql
SELECT MAX(cyclist_time_sec) AS Fastest
FROM cycling.team_average
WHERE team_name = 'UnitedHealthCare Pro Cycling Womens Team';
```
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:

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:

Commands

This section describes the Cassandra Query Language (CQL) commands supported by the DataStax Enterprise database.

**ALTER KEYSSPACE**

Modifies the keyspace replication strategy, which is the number of copies of the data in each datacenter, Table 2, and/or disable the commit log for writes, Durable Writes.

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

Changing the keyspace name is not supported.

**Synopsis**

```
ALTER KEYSSPACE keyspace_name
    WITH REPLICATION = { replication_map }
    [ AND DURABLE_WRITES = ( true | false ) ] ;
```

**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>
</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 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. |

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 DSE.

**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 1.

```
ALTER KEYSPACE cycling
WITH REPLICATION = {
    'class' : 'NetworkTopologyStrategy',
    'SearchAnalytics' : 1
}
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 48: 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>
<tr>
<td><strong>(key : value)</strong></td>
<td>Map collection. Braces ( {} ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<td><strong>&lt;datatype1,datatype2&gt;</strong></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><strong>cql_statement;</strong></td>
<td>End CQL statement. A semicolon (;) terminates all CQL statements.</td>
</tr>
<tr>
<td><strong>[--]</strong></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><strong>' &lt;schema&gt; ... &lt;/schema&gt; '</strong></td>
<td>Search CQL only: Single quotation marks ( ’ ) surround an entire XML schema declaration.</td>
</tr>
</tbody>
</table>
CQL for DataStax Enterprise 6.7 Latest DSE version Latest 6.7 patch: 6.7.7

### 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>

### 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

This section uses the cyclist_base and cyclist_by_age-mv.

#### Modifying table properties

Add a comment and set the bloom filter properties:

```cql
ALTER MATERIALIZED VIEW cycling.cyclist_by_age
WITH comment = 'A most excellent and useful view'
AND bloom_filter_fp_chance = 0.02;
```

For an overview of properties that apply to materialized views, see `table_options`.

#### Modifying compression and compaction

Use a property map to specify new properties for compression or compaction:

```cql
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`.

```cql
ALTER MATERIALIZED VIEW cycling.cyclist_by_age
WITH caching = {
    'keys' : 'NONE',
    'rows_per_partition' : '15'
};
```
Use **DESCRIBE MATERIALIZED VIEW** to see all current properties.

```sql
DESCRIBE MATERIALIZED VIEW cycling.cyclist_by_age;
```

A CQL executable script is displayed:

```sql
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 bloom_filter_fp_chance = 0.02
  AND caching = {'keys': 'NONE', 'rows_per_partition': '15'}
  AND comment = 'A most excellent and useful view'
  AND compaction = {'class':
    'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy',
    'max_threshold': '64',
    'min_threshold': '4'}
  AND compression = {'chunk_length_in_kb': '64', 'class':
    'org.apache.cassandra.io.compress.DeflateCompressor'}
  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';
```

**ALTER ROLE**

Changes password and sets superuser or login options.

**Synopsis**

```sql
ALTER ROLE role_name
  [ WITH [ PASSWORD = role_password ]
  [ [ AND ] LOGIN = ( true | false ) ]
  [ [ AND ] SUPERUSER = ( true | false ) ]
  [ [ AND ] OPTIONS = { option_map } ] ] ;
```

### Table 49: 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 surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
</tbody>
</table>
Cassandra Query Language (CQL) reference

Syntax conventions

<table>
<thead>
<tr>
<th>Syntax conventions</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='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>

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 sandy WITH PASSWORD = 'bestTeam';
```

Alter Search Index Config

Modify the search index pending configuration. Use the CQL shell command DESCRIBE SEARCH INDEX to view the pending and active search index. Use the RELOAD SEARCH INDEX command to apply changes to the active configuration.

Command available only on DSE Search nodes. Running search index management commands on large datasets can take longer than the CQLSH default timeout of 10 minutes. Increase the CQLSH client timeout as required.

Synopsis

```
ALTER SEARCH INDEX CONFIG ON [keyspace_name.]table_name
( ADD element_path [ attribute_list ] WITH $$ json_map $$
| SET element_identifier = 'value'
| SET shortcut = value
| DROP element_identifier
| DROP shortcut ) ;
```

Table 50: 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>
</tbody>
</table>
## Syntax conventions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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, 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>

**keyspace_name.table_name**

Identifies the table of the search index; keyspace name is required when the table is not in the active keyspace.

**element_path**

Identifies the XML path to the setting. Separate child elements using a period. For example:

```plaintext
types.fieldTypes
```

**attribute_list**

A comma-separated list of attributes value pairs enclosed in braces using the following syntax:

```plaintext
[@attribute_name = 'value',
  @attribute_name = 'value', ...
]
```

**json_map**

Advanced. Use JSON format to define child elements, such as analyzer tokenizer and filter definitions of field type.

```plaintext
{ "element_name" : {
  "child_element_name" : {
    "child_attribute_name" : "value",
  },
  "child_element_name" : {
    "child_attribute_name" : "value",
  }, ...
},
"element_name" : {
  "child_element_name" : {
    "child_attribute_name" : "value"
  },
  "child_element_name" : {
    "child_attribute_name" : "value"
  }, ...
}
```
element_identifier
Identifies the XML path to the setting. To locate an element with specific attribute, use the following syntax.

```
(element_name[@attribute_name='value'])
```

shortcut
Shortcuts to configuration element values using SET:

- `autoCommitTime` Default value is 10000.
- `defaultQueryField` Name of the field. Default not set.
  Use SET to add. Use DROP to remove.
- `directoryFactory` Can be used as an alternative to the directoryFactoryClass option. The options are:
  # 'standard'
  # 'encrypted'
- `filterCacheLowWaterMark` Default is 1024.
- `filterCacheHighWaterMark` Default is 2048.
- `directoryFactoryClass` Specifies the fully-qualified name of the directory factory. Use in place of the directoryFactory option for directory factories other than the standard or encrypted directory factory.
- `mergeMaxThreadCount` Must configure with mergeMaxMergeCount. The default is the number of tpc_cores as configured in cassandra.yaml.
- `mergeMaxMergeCount` Must configure with mergeMaxThreadCount. The default calculated value is:
  ```
  max(max(<maxThreadCount * 2>, <num_tokens * 8>), <maxThreadCount + 5>)
  ```
  where `num_tokens` is the number of token ranges to assign to the virtual node (vnode) as configured in cassandra.yaml.
- `ramBufferSize` Default is 512.
- `realtime` Default is false.

EBNF
EBNF syntax:

```
alterSearchIndex ::= 'ALTER' 'SEARCH' 'INDEX' 'CONFIG'
  'ON' tableName
  ( ('ADD' elementPath 'WITH' json) | ('SET' elementPath('@' attribute)? '=' value) | ('DROP' elementPath('@' attribute)?) )

tableName ::= (keyspace '.')? table
elementPath ::= elementName
  ('[ '@' attributeName '=' attributeValue ',' '@' attributeName '=' attributeValue ... ']')?
```
Railroad diagram:

Figure 13: alterSearchIndex
Figure 14:
Figure 15: tableName
Examples

The search index configuration is altered for the wiki.solr keyspace and table, and the specified options.

Enable encryption on search index

Enable encryption on search index:

1. Change the configuration schema:

   ```
   ALTER SEARCH INDEX CONFIG
   ON cycling.comments
   ```
Cassandra Query Language (CQL) reference

```cql
SET directoryFactory = 'encrypted';
```

2. Verify the change is correct in the pending schema:

```cql
DESC PENDING SEARCH INDEX CONFIG ON cycling.comments;
```

3. Make the configuration active:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<config>
  <luceneMatchVersion>LUCENE_6_0_1</luceneMatchVersion>
  <dseTypeMappingVersion>2</dseTypeMappingVersion>
  <directoryFactory class="solr.EncryptedFSDirectoryFactory" name="DirectoryFactory"/>
  <indexConfig>
    <rt>false</rt>
  </indexConfig>
  <jmx/>
  <updateHandler>
    <autoSoftCommit>
      <maxTime>10000</maxTime>
    </autoSoftCommit>
  </updateHandler>
  <query>
    <filterCache class="solr.SolrFilterCache" highWaterMarkMB="2048" lowWaterMarkMB="1024"/>
    <enableLazyFieldLoading>true</enableLazyFieldLoading>
    <useColdSearcher>true</useColdSearcher>
    <maxWarmingSearchers>16</maxWarmingSearchers>
  </query>
  <requestDispatcher>
    <requestParsers enableRemoteStreaming="true" multipartUploadLimitInKB="2048000"/>
    <httpCaching never304="true"/>
  </requestDispatcher>
  <requestHandler class="solr.SearchHandler" default="true" name="search"/>
  <requestHandler class="com.datastax.bdp.search.solr.handler.component.CqlSearchHandler" name="solr_query"/>
  <requestHandler class="solr.UpdateRequestHandler" name="/update" startup="lazy"/>
  <requestHandler class="solr.UpdateRequestHandler" name="/update/csv" startup="lazy"/>
  <requestHandler class="solr.FieldAnalysisRequestHandler" name="/analysis/field" startup="lazy"/>
  <requestHandler class="solr.DocumentAnalysisRequestHandler" name="/analysis/document" startup="lazy"/>
  <requestHandler class="solr.admin.AdminHandlers" name="/admin/"/>
  <requestHandler class="solr.PingRequestHandler" name="/admin/ping">
    <lst name="invariants">
      <str name="gt">search</str>
      <str name="g">solrpingquery</str>
    </lst>
    <lst name="defaults">
      <str name="echoParams">all</str>
    </lst>
  </requestHandler>
  <requestHandler class="solr.DumpRequestHandler" name="/debug/dump">
    <lst name="defaults">
      <str name="echoParams">explicit</str>
      <str name="echoHandler">true</str>
    </lst>
  </requestHandler>
</config>
```
4. Apply the configuration and rebuild the index:

```cql
RELOAD SEARCH INDEX
ON cycling.comments;
```

Auto soft commit max time

To set index configuration with shortcut for automatic soft commit max time to 10000ms:

```cql
ALTER SEARCH INDEX CONFIG
ON cycling.comments
SET autoCommitTime = 10000;
```

Make the pending changes active (use `DESCRIBE SEARCH INDEX` to review changes):

```cql
RELOAD SEARCH INDEX
ON cycling.comments;
```

Request handler

Add a configuration request handler

```cql
ALTER SEARCH INDEX CONFIG
ON cycling.comments
ADD requestHandler[@name='/elevate',@class='solr.SearchHandler', @startup='lazy']
WITH $$ {"defaults":[{"echoParams":"explicit"}],"last-components":["elevator"]} $$;
```

Reload the search index:

```cql
RELOAD SEARCH INDEX
ON cycling.comments;
```

which adds the requestHandler element to the config:

```xml
<requestHandler name="/elevate" class="solr.SearchHandler">
    <lst name="defaults">
        <str name="echoParams">explicit</str>
    </lst>
    <arr name="last-components">
        <str>elevator</str>
    </arr>
</requestHandler>
```

To extend TieredMergePolicy to support automatic removal of deletes:

```cql
ALTER SEARCH INDEX CONFIG
ON cycling.comments
SET
    indexConfig.mergePolicyFactory[@class='org.apache.solr.index.AutoExpungeDeletesTieredMergePolicyFactory'].bool[@name='mergeSingleSegments'] = true;

ALTER SEARCH INDEX CONFIG
ON cycling.comments
SET
    indexConfig.mergePolicyFactory[@class='org.apache.solr.index.AutoExpungeDeletesTieredMergePolicyFactory'].int[@name='maxMergedSegmentMB'] = 1005;
```
ALTER SEARCH INDEX CONFIG
ON cycling.comments
SET
  indexConfig.mergePolicyFactory[@class='org.apache.solr.index.AutoExpungeDeletesTieredMergePolicyFactory'].int[@name='forceMergeDeletesPctAllowed'] = 25;

Reload the search index:

RELOAD SEARCH INDEX
ON cycling.comments;

Change the auto-commit time

To change the automatic commit time, ALTER SEARCH INDEX CONFIG. For example:

ALTER SEARCH INDEX CONFIG
ON cycling.comments
SET autoCommitTime = 1000;

ALTER SEARCH INDEX SCHEMA

Modify the search index pending schema.

See Managing search indexes.

Use the RELOAD SEARCH INDEX command to apply changes to the active schema.

Space saving profiles apply only to the initial creation of the search index. For example, if the index was created using a resource_generation_profiles, like spaceSavingSlowTriePrecision, and later a numeric column is added to the index using the ALTER command, the trie fields precisionStep of the new column is not set to '0'.

Command available only on DSE Search nodes. Running search index management commands on large datasets can take longer than the CQLSH default timeout of 10 minutes. Increase the CQLSH client timeout as required.

Synopsis

ALTER SEARCH INDEX SCHEMA ON [keyspace_name.]table_name
  ( ADD field column_name
  | ADD element_path [ attribute_list ] WITH $$ json_map $$
  | SET element_identifier = 'value'
  | DROP field field_name
  | DROP element_identifier ) ;

Table 51: 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><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>Syntax conventions</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------</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>

**Variables**

**keyspace_name.table_name**
Identifies the table of the search index; keyspace name is required when the table is not in the active keyspace.

**column_name**
Identifies a table column. The search index field and associated type are automatically defined.

**element_path**
Identifies the XML path to the setting. Separate child elements using a period. For example:

```plaintext
    types.fieldTypes
```

**attribute_list**
A comma-separated list of attributes value pairs enclosed in braces using the following syntax:

```plaintext
[ @attribute_name = 'value',
  @attribute_name = 'value', ...
]
```

**json_map**
Advanced. Use JSON format to define child elements, such as analyzer tokenizer and filter definitions of field type.

```json
{   "element_name" : [
    { "child_element_name" : { "child_attribute_name" : "value" } },
    { "child_element_name" : { "child_attribute_name" : "value" } }, ...
],
   "element_name" : [
    { "child_element_name" : { "child_attribute_name" : "value" } },
    { "child_element_name" : { "child_attribute_name" : "value" } }, ...
]
```

**element_identifier**
Cassandra Query Language (CQL) reference

Identifies the XML path to the setting. To locate an element with specific attribute, use the following syntax.

```
  element_name[@attribute_name='value']
```

**ADD**

Insert a new type, field, or other settings in the pending schema.

**DROP**

Remove a table column that corresponds directly to a field or one of the following configurations from the pending schema. The required attributes by element are:

- `field - name attribute`
- `fieldType - name attribute`
- `dynamicField - name attribute`
- `copyField - source` and `dest`

See [Managing search index fields](#) and [Dropping columns from the index](#) in the documentation.

**SET**

Change the configuration of a setting in the pending schema.

**EBNF**

**EBNF** syntax:

```
alterSearchIndex ::= 'ALTER' 'SEARCH' 'INDEX' 'SCHEMA'
                  'ON' tableName
                  ( ('ADD' elementPath 'WITH' json) |
                   ('ADD' 'FIELD' fieldName) |
                   ('SET' elementPath('@' attribute)? '=' value)? |
                   ('DROP' elementPath('@' attribute)?) |
                   ('DROP' 'FIELD' fieldName) )

tableName ::= (keyspace '.')? table

elementPath ::= elementName
              ( ['' '@' attributeName '=' attributeValue |
                  ('',' '@' attributeName '=' attributeValue)* ']')?
              ( '.' elementName ( ['' '@' attributeName '=' attributeValue |
                                 ('',' '@' attributeName '=' attributeValue)* ']')*)
```

**Railroad diagram:**
Figure 17: alterSearchIndex
Figure 18:
Figure 19: `tableName`

Diagram showing the relationship between keyspace and `tableName`.
Examples

The search index schema is altered for the `cycling.comments` keyspace and table, and the specified options.

For extensive information and examples on search indexes, including adding and dropping search index fields, field types, field classes, tuples, UDTs, and map columns, see Managing search index fields.

You must add the search index before you can alter it.

Add a new field using the element path and attribute list

Fields that do not exist in the table can be added to index to combine multiple columns into a single indexed field for searches.
Adding the leading element `fields.` in ADD `fields.field` `fieldname` is optional and provides only cosmetic structure.

```cql
ALTER SEARCH INDEX SCHEMA
ON cycling.comments
ADD fields.field[@name='fieldname', @type='StrField', @multiValued = 'false', @indexed='true'];
```

To apply the schema changes:

```cql
RELOAD SEARCH INDEX
ON cycling.comments;
```

Add a table column to the index

Automatically creates a field definition and adds the field type if required for a field that is not already indexed.

```cql
ALTER SEARCH INDEX SCHEMA
ON cycling.comments
ADD FIELD record_id;
```

To apply the schema changes and rebuild the index:

```cql
RELOAD SEARCH INDEX
ON cycling.comments;
REBUILD SEARCH INDEX
ON cycling.comments;
```

Change a field name

DSE maps CQL columns to search index fields by matching the column name to the field name. Use unmapped fields for copy fields. If the field does not already exist it is added.

```cql
ALTER SEARCH INDEX SCHEMA
ON cycling.comments
SET field[@name='fieldname'] @name = 'anotherFieldName';
```

To apply the schema changes:

To apply the schema changes and rebuild the index:

```cql
RELOAD SEARCH INDEX
ON cycling.comments;
REBUILD SEARCH INDEX
ON cycling.comments;
```

Change the field type

Change the field type to another type that is already defined in the schema:

```cql
ALTER SEARCH INDEX SCHEMA
ON cycling.comments
SET field[@name='fieldname'] @type = 'UUIDField';
```

To apply the schema changes:
Cassandra Query Language (CQL) reference

To apply the schema changes and rebuild the index:

```
RELOAD SEARCH INDEX
ON cycling.comments;

REBUILD SEARCH INDEX
ON cycling.comments;
```

Drop a field

To remove a field from the search index, but not from the table:

```
ALTER SEARCH INDEX SCHEMA
ON cycling.comments
DROP field anotherFieldName;
```

To apply the schema changes and rebuild the index:

```
RELOAD SEARCH INDEX
ON cycling.comments;

REBUILD SEARCH INDEX
ON cycling.comments;
```

Set a field type and a text field

The first command sets the `TextField` type, which is required for the second command that sets the text field:

```
ALTER SEARCH INDEX SCHEMA
ON cycling.comments
SET types.fieldType[@name='TextField']@class='org.apache.solr.schema.TextField';

ALTER SEARCH INDEX SCHEMA
ON cycling.comments
SET fields.field[@name='comment']@type='TextField';
```

To apply the schema changes and rebuild the index:

```
RELOAD SEARCH INDEX
ON cycling.comments;

REBUILD SEARCH INDEX
ON cycling.comments;
```

Drop a field and a field type

The first command drops the field from the search index and the second command drops the field type:

```
ALTER SEARCH INDEX SCHEMA
ON cycling.comments
DROP field comment;

ALTER SEARCH INDEX SCHEMA
ON cycling.comments
DROP types.fieldType[@name='TextField'];
```

To apply the schema changes and rebuild the index:

```
RELOAD SEARCH INDEX
```
Add a field type and a dynamic field

The first command adds the **TextField** type, which must exist before the dynamic field is added by the second command:

```cql
ALTER SEARCH INDEX SCHEMA
ON cycling.comments
ADD types.fieldType[@class='org.apache.solr.schema.TextField', @name='TextField']
WITH '{"analyzer":{"class":"org.apache.lucene.analysis.standard.StandardAnalyzer"}}';
```

```cql
ALTER SEARCH INDEX SCHEMA
ON cycling.comments
ADD dynamicField[@name='*fieldname', @type='TextField'];
```

Ensure your dynamic field name has a leading or a trailing asterisk character.

To apply the schema changes and rebuild the index:

```cql
RELOAD SEARCH INDEX
ON cycling.comments;
```

```cql
REBUILD SEARCH INDEX
ON cycling.comments;
```

**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**

```cql
ALTER TABLE [keyspace_name.]table_name
  [ ADD ( column_definition | column_definition_list ) [ , ... ] ]
  [ DROP column_name [ , ... ] ]
  [ RENAME column_name TO column_name ]
```
Table 52: 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>

ADD column_definition | (column_definition_list)
Add one or more columns and set the column data types. Specify the column names followed by the data types. The column 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.

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 Cassandra Enterprise Search indexes.

```table_properties```
You can modify an existing table’s properties. Some properties are single options that are set to a value:

```
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:

```
option_name = { subproperty_name : value [ , ... ] }
```

See `table_options` for more details.

**Examples**

This section uses the `cyclist_races` table.

**Adding a column**

To add a column, use the `ADD` instruction:

```
ALTER TABLE cycling.cyclist_races
ADD manager UUID;
```

To add a column of a collection type:

```
ALTER TABLE cycling.cyclist_races
ADD completed 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.

**Dropping a column**

To remove a column from the table, use the `DROP` instruction:

```
ALTER TABLE cycling.cyclist_races
DROP manager;
```

`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 Enterprise 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.

**Renaming a column**

To rename a column in the `race_times` table:

```
ALTER TABLE cycling.race_times
```

CQL for DataStax Enterprise 6.7 Latest DSE version Latest 6.7 patch: 6.7.7
RENAME race_date TO date;

The following restrictions apply to RENAME:

- 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 the data 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.

For example, to add a comment to the cyclist_base table using WITH:

```
ALTER TABLE cycling.cyclist_base
WITH comment = 'basic cyclist information';
```

Enclose a text property value in single quotation marks.

Modifying compression and compaction
Use a property map to alter the comments 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 the string 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 the comments table to 10 rows:

```
ALTER TABLE cycling.comments
WITH caching = {
  'keys' : 'NONE',
  'rows_per_partition' : 10
};
```

Change the speculative retries
Modify the `cyclist_base` table to 95th percentile for speculative retry:

```
ALTER TABLE cycling.cyclist_base
WITH speculative_retry = '95percentile';
```

Modify the `cyclist_base` table to use 10 milliseconds for speculative retry:

```
ALTER TABLE cycling.cyclist_base
WITH speculative_retry = '10ms';
```

Enabling and disabling background compaction

The following example sets the `enabled` property to `false` to disable background compaction:

```
ALTER TABLE cycling.comments
WITH COMPACTION = {
  'class' : 'SizeTieredCompactionStrategy',
  'enabled' : 'false'
};
```

Disabling background compaction can be harmful: without it, the database does not regain disk space, and could 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 compaction 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 the following types of compaction events:

- **type: enable**
  Lists SSTables that have been flushed previously.

```
{"type":"enable","keyspace":"test","table":"t","time":1470071098866,"strategies":
  [{"strategyId":"0","type":"LeveledCompactionStrategy","tables":[]},
   {"strategyId":"1","type":"LeveledCompactionStrategy","tables":[]}
  ],
  "repaired":true,"folders":
  ["/home/carl/oss/cassandra/bin/..data/data"],
  "details":
  {"level":0,"min_token":-9221834874718566760,"max_token":9221396997139245178"
}
```

- **type: flush**
  Logs a flush event from a memtable to an SSTable on disk, including the CompactionStrategy for each table.

```
{"type":"flush","keyspace":"test","table":"t","time":1470083335639,"strategies":
  [{"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": [
    {
      "strategyId": "1",
      "table": {
        "generation": 1404,
        "version": "mb",
        "size": 1064306,
        "details": {
          "level": 2,
          "min_token": "7199305267944662291",
          "max_token": "732343447996777057"
        }
      }
    }
  ]
}
```

• type: pending

Lists the number of pending tasks for a compaction strategy.

```json
{
  "type": "pending",
  "keyspace": "test",
  "table": "t",
  "time": 1470083447967,
  "strategyId": "1",
  "pending": 200
}
```

Reviewing the table definition

Use `DESCRIBE` or `DESC` to view the table definition.

```sql
DESC cycling.comments;
```

The table details including the column names are returned.

```sql
CREATE TABLE cycling.comments (  
id uuid,  
  created_at timestamp,  
  comment text,  
  commenter text,  
  record_id timeuuid,  
PRIMARY KEY (id, created_at)  
) WITH CLUSTERING ORDER BY (created_at DESC)  
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',  
  'enabled': 'true',  
  '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 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
```
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

```
ALTER TYPE field_name
   ( ADD field_name cql_datatype
   | RENAME field_name TO new_field_name [ AND field_name TO new_field_name ... ] ) ;
```

Table 53: Legend

<table>
<thead>
<tr>
<th></th>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE</td>
<td></td>
<td>Literal keyword.</td>
</tr>
<tr>
<td>Lowercase</td>
<td></td>
<td>Not literal.</td>
</tr>
<tr>
<td>Italic</td>
<td></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td>[]</td>
<td></td>
<td>Optional. Square brackets ([ ]) surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td>{}</td>
<td></td>
<td>Group. Parentheses ({ }) identify a group to choose from. Do not type the parentheses.</td>
</tr>
<tr>
<td>[]</td>
<td></td>
<td>Or. A vertical bar (</td>
</tr>
<tr>
<td>...</td>
<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></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></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></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></td>
<td>End CQL statement. A semicolon (;) terminates all CQL statements.</td>
</tr>
<tr>
<td>[--]</td>
<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></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></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 (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

This section uses the `fullname` type.

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 cycling.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`.

Rename the fields in the `cycling.fullname` UDT.

```cql
ALTER TYPE cycling.fullname
RENAME middlename TO middle
AND lastname TO last
AND firstname TO first;
```

Verify the changes using `describe`:

```cql
DESC TYPE cycling.fullname;
```

The new field names appear in the description.

```cql
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

```cql
ALTER USER user_name
  WITH PASSWORD user_password
  [ ( SUPERUSER | NOSUPERUSER ) ] ;
```

Table 54: 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>
</tbody>
</table>
### Syntax conventions

| **Italic**  | Variable value. Replace with a user-defined value. |
| [ ]         | Optional. Square brackets ([ ]) surround optional command arguments. Do not type the square brackets. |
| { }         | Group. Parentheses ({ }) identify a group to choose from. Do not type the parentheses. |
| | Or. A vertical bar (|) separates alternative elements. Type any one of the elements. Do not type the vertical bar. |
| [...]       | Repeatable. An ellipsis ( [... ] ) indicates that you can repeat the syntax element as often as required. |
| '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. |

### 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.

See [Batching inserts, updates, and deletes](#).

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 | LOGGED ) ] BATCH
   [ USING TIMESTAMP [ epoch_microseconds ] ]
   dml_statement [ USING TIMESTAMP [ epoch_microseconds ] ] ;
   [ dml_statement [ USING TIMESTAMP [ epoch_microseconds ] ] ; ... ] ;
APPLY BATCH ;

<table>
<thead>
<tr>
<th>Table 55: Legend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax conventions</td>
</tr>
<tr>
<td>UPPERCASE</td>
</tr>
<tr>
<td>Lowercase</td>
</tr>
<tr>
<td>Italics</td>
</tr>
<tr>
<td>[ ]</td>
</tr>
<tr>
<td>( )</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>'Literal string'</td>
</tr>
<tr>
<td>( key : value )</td>
</tr>
<tr>
<td>&lt;datatype1,datatype2&gt;</td>
</tr>
<tr>
<td>cql_statement ;</td>
</tr>
<tr>
<td>[---]</td>
</tr>
<tr>
<td>' &lt;schema&gt; ... &lt;/schema&gt; '</td>
</tr>
<tr>
<td>&amp;xml_entity='xml_entity_type'</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**

This section uses the cyclist_expenses and popular_count tables.

Applying a client supplied timestamp to all DMLs
Insert meals paid for Vera Adrian using the user-defined date when inserting the records:

```
BEGIN BATCH USING TIMESTAMP 1481124356754405

INSERT INTO cycling.cyclist_expenses (cyclist_name, expense_id, amount, description, paid)
VALUES ('Vera ADRIAN', 2, 13.44, 'Lunch', true);

INSERT INTO cycling.cyclist_expenses (cyclist_name, expense_id, amount, description, paid)
VALUES ('Vera ADRIAN', 3, 25.00, 'Dinner', true);

APPLY BATCH;
```

Combining two statements for the same partition results in a single table mutation.

View the records vertically in `cqlsh`:

```
EXPAND ON
```

Verify that the timestamps are all the same:

```
SELECT
  cyclist_name, expense_id,
  amount, WRITETIME(amount),
  description, WRITETIME(description),
  paid, WRITETIME(paid)
FROM cycling.cyclist_expenses
WHERE cyclist_name = 'Vera ADRIAN';
```

Both records were entered with the same timestamp.

```
@ Row 1
------------------------+------------------
cyclist_name           | Vera ADRIAN
expense_id             | 2
amount                 | 13.44
writetime(amount)      | 1481124356754405
description            | Lunch
writetime(description) | 1481124356754405
paid                   | True
writetime(paid)        | 1481124356754405

@ Row 2
------------------------+------------------
cyclist_name           | Vera ADRIAN
expense_id             | 3
amount                 | 25
writetime(amount)      | 1481124356754405
description            | Dinner
writetime(description) | 1481124356754405
paid                   | True
writetime(paid)        | 1481124356754405
```
If any DML statement in the batch uses compare-and-set (CAS) logic, an error is returned. For example, the following batch with the CAS IF NOT EXISTS option returns an error:

BEGIN BATCH USING TIMESTAMP 1481124356754405
    INSERT INTO cycling.cyclist_expenses (cyclist_name, expense_id, amount, description, paid)
    VALUES ('Vera ADRIAN', 2, 13.44, 'Lunch', true);
    INSERT INTO cycling.cyclist_expenses (cyclist_name, expense_id, amount, description, paid)
    VALUES ('Vera ADRIAN', 3, 25.00, 'Dinner', false)
    IF NOT EXISTS;
APPLY BATCH;

InvalidRequest: Error from server: code=2200 [Invalid query]
message="Cannot provide custom timestamp for conditional BATCH"

Batching conditional updates

Batch conditional updates introduced as lightweight transactions. However, a batch containing conditional updates can operate only 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.

The following example shows batching of conditional updates. The first statement uses the IF NOT EXISTS conditional clause.

BEGIN BATCH
    INSERT INTO cycling.cyclist_expenses (cyclist_name, expense_id)
    VALUES ('Joe WALLS', 1)
    IF NOT EXISTS;
    INSERT INTO cycling.cyclist_expenses (cyclist_name, expense_id, amount, description, paid)
    VALUES ('Joe WALLS', 1, 8, 'burrito', 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 Enterprise, a counter update is not an idempotent operation.

```cql
BEGIN COUNTER BATCH

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

UPDATE cycling.popular_count
SET popularity = popularity + 125
WHERE id = 6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47;

UPDATE cycling.popular_count
SET popularity = popularity - 64
WHERE id = 6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47;

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.

**COMMIT SEARCH INDEX**

Forces an update of the search index with the most recent data after executing an `INSERT`, `UPDATE`, or `DELETE` statement.

By default, changes are automatically committed every 10000 milliseconds. To change the default setting see Changing the autocommit time.

Command available only on DSE Search nodes. Running search index management commands on large datasets can take longer than the CQLSH default timeout of 10 minutes. Increase the CQLSH client timeout as required.

**Synopsis**

```
COMMIT SEARCH INDEX ON [keyspace_name.]table_name ;
```

**Table 56: 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>&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 ( {} ) 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.

**EBNF**

**EBNF syntax:**

```
commitSearchIndex ::= 'COMMIT' 'SEARCH' 'INDEX'
                   'ON' tableName
tableName ::= (keyspace '.')? table
```

Figure 21: commitSearchIndex
Examples

The search index is committed for the comments search index table.

```cql
COMMIT SEARCH INDEX
ON cycling.comments;
```

**CREATE AGGREGATE**

Defines a user-defined aggregate. An aggregate executes a user-define 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**

```cql
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
```
### Table 57: 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>Italic</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><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>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>

**OR REPLACE**
Overwrites existing aggregate (with the same name). When OR REPLACE is not specified the operations fail 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]
Cassandra Query Language (CQL) reference

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).

GROUP BY only supports aggregates that are deterministic.

**Examples**
Create an aggregate that calculates average in the cycling keyspace and test on the team_average.

This section explains how to create a function that has the same functionality as the native AVG function.

1. 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.

   ```java
   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.

```cql
CREATE TABLE IF NOT EXISTS 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
};

INSERT INTO cycling.test_avg {
  id, state, val
} VALUES {
  2, (79, 10000), 9999
};

SELECT state, avgstate(state, val), val
FROM cycling.test_avg;
```

The first value was incremented by one and the second value is the results of the initial state value and val.

<table>
<thead>
<tr>
<th>state</th>
<th>cycling.avgstate(state, val)</th>
<th>val</th>
</tr>
</thead>
</table>
2. 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
$$
    double r = 0;
    if (state.getInt(0) == 0) return null;
    r = state.getLong(1);
    r /= state.getInt(0);
    return Double.valueOf(r);
$$
;
```

3. 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);
```

4. 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'
    AND race_title = 'Amgen Tour of California Women''s Race presented by SRAM - Stage 1 - Lake Tahoe > Lake Tahoe';
```

**CREATE CUSTOM INDEX (SASI)**

Generates SSTable Attached Secondary Index (SASI) on a table column.

**Upgrade impact:** Changes in DSE 6.0 could effect your implementation. See 6.0 release notes and be sure to follow the upgrade instructions for required actions.

- The o.a.c.index.Index interface was modified to comply with core storage engine changes. Updated implementations are required. If unsure, drop all existing custom secondary indexes before upgrading, except DSE Search indexes, which do not need to be replaced. Because a rewrite of custom index implementations is necessary in DSE, DataStax Support can help you find a solution.

SASI uses significantly using fewer memory, disk, and CPU resources. It enables querying with PREFIX and CONTAINS on strings, similar to the SQL implementation of `LIKE = "foo*"` or `LIKE = "*foo*"`.

SASI indexes in DSE are experimental. DataStax does not support SASI indexes for production.

For more information about SASI, see Using SASI.
### Synopsis

CREATE CUSTOM INDEX [ IF NOT EXISTS ] [ index_name ]
ON [ keyspace_name.|table_name (column_name)
USING 'org.apache.cassandra.index.sasi.SASIIndex'
[ WITH OPTIONS = { option_map } ] ;

<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>

**index_name**

Optional identifier for index. If no name is specified, the default is used, `table_name.column_name_idx`. Enclose in quotes to use special characters or preserve capitalization.

**OPTIONS**

Define options in JSON simple format.

Specifying an analyzer allows:

- Analyzing and indexing text column data
- Using word stemming for indexing
- Specifying words that can be skipped
- Applying localization based on a specified language
- Case normalization, like the non-tokening analyzer
 Analyzer class option

The SASI indexer has two analyzer classes (analyzer_class):

- org.apache.cassandra.index.sasi.analyzer.StandardAnalyzer (default analyzer)
- org.apache.cassandra.index.sasi.analyzer.NonTokenizingAnalyzer

Specify the class:

'class' : 'org.apache.cassandra.index.sasi.analyzer.NonTokenizingAnalyzer'

There are global options that apply to both and class specify options, **Standard Analyzer** and **Non-tokenizing Analyzer**.

Global options

The following options apply to all analyzer classes:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>analyzed</td>
<td>True indicates if the literal column is analyzed using the specified analyzer.</td>
</tr>
<tr>
<td>is_literal</td>
<td>Designates a column as literal.</td>
</tr>
<tr>
<td>max_compaction_flush_memory_in_mb</td>
<td>Enter the size.</td>
</tr>
</tbody>
</table>

Standard analyzer options

Default analyzer class. The following options are available for the org.apache.cassandra.index.sasi.analyzer.StandardAnalyzer.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tokenization_enable_stemming</td>
<td>Reduce words to their base form, for example &quot;stemmer&quot;, &quot;stemming&quot;, &quot;stemmed&quot; are based on &quot;stem&quot;. Default: false.</td>
</tr>
<tr>
<td>tokenization_skip_stop_words</td>
<td>Comma-separate list of words to ignore, for example 'and, the, or'.</td>
</tr>
<tr>
<td>tokenization_locale</td>
<td>Language code of the column, see List of localization codes. Default: en.</td>
</tr>
<tr>
<td>tokenization_normalize_lowercase</td>
<td>Use lowercase. Default false.</td>
</tr>
<tr>
<td>tokenization_normalize_uppercase</td>
<td>Use uppercase. Default false.</td>
</tr>
</tbody>
</table>

Non-tokenizing analyzer options

The following options are available for the org.apache.cassandra.index.sasi.analyzer.NonTokenizingAnalyzer.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>normalize_lowercase</td>
<td>Index all strings as lowercase. Default: false.</td>
</tr>
<tr>
<td>normalize_uppercase</td>
<td>Index all strings as uppercase. Default: false.</td>
</tr>
<tr>
<td>case_sensitive</td>
<td>Ignore case in matching. Default is case-sensitive indexing, setting: true.</td>
</tr>
</tbody>
</table>
Examples
All examples use the cycling.cyclist_name table.

Creating a SASI PREFIX index on a column
Create an SASI index on the column firstname:

```
CREATE CUSTOM INDEX IF NOT EXISTS fn_prefix
ON cycling.comments (commenter)
USING 'org.apache.cassandra.index.sasi.SASIIndex';
```

The SASI mode PREFIX is the default, and does not need to be specified.

Creating a SASI CONTAINS index on a column
Create an SASI index on the column firstname:

```
CREATE CUSTOM INDEX IF NOT EXISTS fn_contains
ON cycling.comments (comment)
USING 'org.apache.cassandra.index.sasi.SASIIndex'
WITH OPTIONS = {
    'mode' : 'CONTAINS'
};
```

The SASI mode CONTAINS must be specified.

Creating a SASI SPARSE index on a column
Define a table and then create an SASI index on the column age:

```
CREATE CUSTOM INDEX IF NOT EXISTS fn_sparse
ON cycling.comments (record_id)
USING 'org.apache.cassandra.index.sasi.SASIIndex'
WITH OPTIONS = {
    'mode' : 'SPARSE'
};
```

The SASI mode SPARSE must be specified. This mode is used for dense number columns that store timestamps or millisecond sensor readings.

Creating a SASI PREFIX index on a column using the non-tokenizing analyzer
Define a table, then create an SASI index on the column age:

```
CREATE CUSTOM INDEX IF NOT EXISTS fn_notcasesensitive
ON cycling.comments (comment)
USING 'org.apache.cassandra.index.sasi.SASIIndex'
WITH OPTIONS = {
    'analyzer_class' : 'org.apache.cassandra.index.sasi.analyzer.NonTokenizingAnalyzer',
    'case_sensitive' : 'false'
};
```

Using the non-tokenizing analyzer is a method to specify case sensitivity or character case normalization without analyzing the specified column.

Creating a SASI analyzing index on a column
Define a table and then create an SASI index on the column comments:

```
CREATE CUSTOM INDEX IF NOT EXISTS stdanalyzer_idx
ON cycling.comments (comment)
USING 'org.apache.cassandra.index.sasi.SASIIndex'
WITH OPTIONS = {
    'mode' : 'CONTAINS',
};
```
CREATE FUNCTION
Executes user-provided Java or Javascript code in SELECT, UPDATE, INSERT or provides a building block for user-defined aggregate. Functions are only available in the keyspace where they were created.

UDF supports Java generic methods or Javascript in the user provided codeblock. 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 of problems. An exception during function execution results in the entire statement failing.

By default, the database does not allow UDFs. To enable, change the following settings in the cassandra.yaml and restart all nodes:

- Set enable_user_defined_functions to true - allow users to create custom functions. Only Java is allowed in codeblocks if enable_scripted_user_defined_functions is false.
- (Optional) Set enable_scripted_user_defined_functions to true - allow function codeblocks to use Javascript.
- (Optional) Set enable_user_defined_functions_threads to false to allow functions in GROUP BY clauses.

Synopsis

CREATE [ OR REPLACE ] FUNCTION [ IF NOT EXISTS ] [keyspace_name.]function_name (argument_list [ , ... ])
( CALLED | RETURNS NULL ) ON NULL INPUT RETURNS type
[ DETERMINISTIC ]
[ MONOTONIC [ ON argument_name ] ]
LANGUAGE ( java | javascript ) AS $$ code_block $$ ;

Table 59: 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 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>
Cassandra Query Language (CQL) reference

<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 ( <code>&lt;</code> &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 (<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>

**function_name**
Specify a keyspace-qualified function name. Names must start with a letter or number. To preserve case, enclose the name in double-quotes.

**OR REPLACE**
Overwrite the function, if one already exists with the same name.

**IF NOT EXISTS**
Performs no operation and suppresses the error message if a function with the same name already exists.

**argument_list**
Comma separated list of arguments with data types passed to the code block for processing:

```cql
arg_name cql_type [, ...]
```

In the list, specify an argument name followed by the CQL data type.

For requests, an argument value can be read from a column with the corresponding data type or manually entered (literal).

**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, while now() and currentDate() are not.

Default: `false` (non-deterministic).

GROUP BY only supports functions that are both deterministic and monotonic.

**MONOTONIC [ ON argument_name ]**
All arguments or the specified argument are monotonic if they are either entirely non-increasing or non-decreasing.

GROUP BY only supports functions that are both deterministic and monotonic.

**LANGUAGE language_name**
Supported types are `java` or `javascript`.

When `enable_scripted_user_defined_functions` is false and `enable_user_defined_functions` is true, Java is the only supported language.

`'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.
Examples

Use Java to create FLOG function

Overwrite or create the fLog function that computes the logarithm of an input value. CALLED ON NULL INPUT ensures that the function will always be executed.

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

Use Javascript to create SQL-like LEFT function

Create a function that returns the first \( N \) characters from a text field in Javascript. RETURNS NULL ON NULL INPUT ensures that if the input value is null then the function is not executed.

```
CREATE 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:

```
SOURCE 'left-function.cql';
CAPTURE 'cyclist_name-select_left.results';
SELECT left(firstname, 1), lastname
FROM cycling.cyclist_name;
CAPTURE OFF;
```

<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>null</td>
<td>MATTHEWS</td>
</tr>
<tr>
<td>null</td>
<td>VOS</td>
</tr>
<tr>
<td>P</td>
<td>TIRALONGO</td>
</tr>
<tr>
<td>M</td>
<td>VOS</td>
</tr>
<tr>
<td>S</td>
<td>KRIUKSWIJK</td>
</tr>
<tr>
<td>A</td>
<td>VAN DER BREGGEN</td>
</tr>
</tbody>
</table>

(7 rows)

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. After an index has been created, it is automatically updated when data in the column changes. DataStax Enterprise 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 the key, value, or entries.

**Synopsis**

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

**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 IF NOT EXISTS cycling.rank_by_year_and_name (
    race_year int,
    race_name text,
    cyclist_name text,
    rank int,
);```
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 teams to the `cyclist_career_teams` table to index the data in the teams set.

```
CREATE TABLE IF NOT EXISTS cycling.cyclist_career_teams (  
id UUID PRIMARY KEY,  
lastname text,  
teams set<text>
);
```

```
CREATE INDEX IF NOT EXISTS teams_idx
ON cycling.cyclist_career_teams (teams);
```

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. Assume a cyclist table contains this map data:

```
{ 'nation': 'CANADA' }
```

The map key is located to the left of the colon, and the map value is located to the right of the colon.

To index map keys, use the `KEYS` keyword and map name in nested parentheses:

```
CREATE INDEX IF NOT EXISTS team_year_idx  
ON cycling.cyclist_teams ( KEYS (teams) );
```

To query the table, you can use `CONTAINS KEY` in `WHERE` clauses.

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

The example returns cyclist teams that have an entry for the year 2015.

```
+----------------+---------------+----------------+-------------------------+
| id             | firstname     | lastname       | teams                   |
|----------------|---------------+----------------+-------------------------|
| 5b6962dd-3f90-4c93-8f61-eabfa4a803e2 | Marianne      | VOS            | {2015: 'Rabobank-Liv Woman Cycling Team'} |
```

Creating an index on map entries

```
CREATE INDEX IF NOT EXISTS team_year_idx  
ON cycling.cyclist_teams ( KEYS (teams) );
```

To query the table, you can use `CONTAINS KEY` in `WHERE` clauses.

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

The example returns cyclist teams that have an entry for the year 2015.

```
+----------------+---------------+----------------+-------------------------+
| id             | firstname     | lastname       | teams                   |
|----------------|---------------+----------------+-------------------------|
| 5b6962dd-3f90-4c93-8f61-eabfa4a803e2 | Marianne      | VOS            | {2015: 'Rabobank-Liv Woman Cycling Team'} |
```

(2 rows)
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, use the **ENTRIES** keyword and map name in nested parentheses:

```cql
CREATE INDEX IF NOT EXISTS blist_idx
ON cycling.birthday_list ( ENTRIES(blist) );
```

To query the map entries in the table, use a **WHERE** clause with the map name and a value.

```cql
SELECT *
FROM cycling.birthday_list
WHERE blist[ 'age' ] = '23';
```

The example finds cyclists who are the same age.

<table>
<thead>
<tr>
<th>cyclist_name</th>
<th>blist</th>
</tr>
</thead>
<tbody>
<tr>
<td>Claudio HEINEN</td>
<td>{'age': '23', 'bday': '27/07/1992', 'nation': 'GERMANY'}</td>
</tr>
<tr>
<td>Laurence BOURQUE</td>
<td>{'age': '23', 'bday': '27/07/1992', 'nation': 'CANADA'}</td>
</tr>
</tbody>
</table>

(2 rows)

Use the same index to find cyclists from the same country.

```cql
SELECT *
FROM cycling.birthday_list
WHERE blist[ 'nation' ] = '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>

(2 rows)

Creating an index on map values

To create an index on map values, use the **VALUES** keyword and map name in nested parentheses:

```cql
CREATE INDEX IF NOT EXISTS blist_values_idx
ON cycling.birthday_list ( VALUES(blist) );
```

To query the table, use a **WHERE** clause with the map name and the value it contains.

```cql
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>
Creating an index on the full content of a frozen collection

You can create an index on a full FROZEN collection. A FULL index can be created on a set, list, or map column of a table that doesn't have an existing index.

Create an index on the full content of a FROZEN list. The table in this example stores the number of Pro wins, Grand Tour races, and Classic races that a cyclist has competed in.

```
CREATE TABLE IF NOT EXISTS cycling.race_starts (
    cyclist_name text PRIMARY KEY,
    rnumbers FROZEN<LIST<int>>
);
```

To index collection entries, use the FULL keyword and collection name in nested parentheses. For example, index the frozen list rnumbers.

```
CREATE INDEX IF NOT EXISTS rnumbers_idx
ON cycling.race_starts ( FULL(rnumbers) );
```

To query the table, use a WHERE clause with the collection name and values:

```
SELECT *
FROM cycling.race_starts
WHERE rnumbers = [39, 7, 14];
```

```
cyclist_name | rnumbers
--------------|------------------
John DEGENKOLB | [39, 7, 14]
```

CREATE KEYSAPCE

Creates a top-level keyspace. Configure the replica placement strategy, replication factor, and durable writes setting.

Use only replication strategy implementations bundled with DSE.

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

Table 61: 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>

### keyspace_name

Maximum of 222 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 62: Replication strategy class and factor settings

<table>
<thead>
<tr>
<th>Class</th>
<th>Replication factor</th>
<th>Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'SimpleStrategy'</td>
<td>'replication_factor': N</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>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 Topolgy 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:

```cql
CREATE KEYSPACE cycling
WITH REPLICA`
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 REPLICAION = {
  '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.
- Multiple non-primary key columns from the base table are supported when the partition key is the same as in the base table, otherwise only a single non-primary key from the base table is allowed in the materialized view's PRIMARY KEY.
- Static columns are not supported.

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 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>
<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>
</tbody>
</table>
### Syntax conventions

<table>
<thead>
<tr>
<th>Syntax conventions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>

### 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, set the 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 columns from the base table to include in the materialized view. Static columns, even when specified, are not supported and not included in the materialized view.

#### column_name IS NOT NULL

Test all columns for null values in the WHERE clause. Separate each condition with AND. Rows with null values in any column 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 non-primary key columns 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:

```plaintext
PRIMARY KEY {
( PK_column1 [, PK_column2... ] ),
clustering_column1 [, clustering_column2 ... ] )
```

#### table_properties
Cassandra Query Language (CQL) reference

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 examples of materialized views

These tables are used in the first example scenario:

```cql
CREATE TABLE IF NOT EXISTS cycling.cyclist_base (
    cid UUID PRIMARY KEY,
    name text,
    age int,
    birthday date,
    country text
);

CREATE TABLE IF NOT EXISTS cycling.cyclist_base_ext (
    cid UUID,
    name text,
    age int,
    birthday date,
    country text,
    PRIMARY KEY (cid, age, birthday)
);
```

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.

```cql
CREATE MATERIALIZED VIEW IF NOT EXISTS 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:

```cql
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>
</tbody>
</table>
The following materialized view **cyclist_by_birthday_and_age** uses the base table **cyclist_base**. The **WHERE** clause ensures that only rows whose **age**, **birthday**, and **cid** columns are non-NULL are added to the materialized view. In the materialized view, **cid** is the partition key, **birthday** is the first clustering column, and **age** is the second clustering column. In the base table, **cid** is the partition key.

```
CREATE MATERIALIZED VIEW IF NOT EXISTS cycling.cyclist_by_birthday_and_age AS
SELECT age, cid, birthday, country, name
FROM cycling.cyclist_base
WHERE age IS NOT NULL
AND birthday IS NOT NULL
AND cid IS NOT NULL
PRIMARY KEY (cid, birthday, age);
```

The results of this query:

```
SELECT *
FROM cycling.cyclist_by_birthday_and_age;
```

are:

<table>
<thead>
<tr>
<th>cid</th>
<th>birthday</th>
<th>age</th>
<th>country</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ffdfa2a7-5fc6-49a7-bfdd-c3fcdcfdd7156</td>
<td>1997-02-08</td>
<td>18</td>
<td>Netherlands</td>
<td>Pascal</td>
</tr>
<tr>
<td>15a116fc-b833-4da6-ab9a-4a7776572836</td>
<td>1997-08-19</td>
<td>18</td>
<td>United States</td>
<td>Adrien</td>
</tr>
<tr>
<td>e7aa65cf3-d358-4d99-b900-85902f9b00</td>
<td>1993-06-18</td>
<td>22</td>
<td>New Zealand</td>
<td>Alex</td>
</tr>
<tr>
<td>c9c9c484-5e4a-4542-8203-8d047a01b8a8</td>
<td>1987-09-04</td>
<td>27</td>
<td>Brazil</td>
<td>Cristian</td>
</tr>
<tr>
<td>862cc51f-00a1-4d5a-976b-a359cab7300e</td>
<td>1994-09-04</td>
<td>20</td>
<td>Denmark</td>
<td>Joakim</td>
</tr>
<tr>
<td>220844bf-4860-49d6-9a4b-6b5d3a79cbf</td>
<td>1977-07-08</td>
<td>38</td>
<td>Italy</td>
<td>Paolo</td>
</tr>
<tr>
<td>96c4c40d-58c8-4710-b73f-681e9b1f70ae</td>
<td>1989-04-20</td>
<td>18</td>
<td>Australia</td>
<td>Benjamin</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>------------</td>
<td>-----</td>
<td>---------------</td>
<td>--------------------</td>
</tr>
</tbody>
</table>

(12 rows)
The following materialized view `cyclist_by_country_and_birthday` uses the base table `cyclist_base`. The `WHERE` clause ensures that only rows whose `birthday` and `cid` columns are non-NULL and `country` equals `Australia` are added to the materialized view.

```cql
CREATE MATERIALIZED VIEW IF NOT EXISTS cycling.cyclist_by_country_and_birthday AS
SELECT age, cid, birthday, country, name
FROM cycling.cyclist_base
WHERE birthday IS NOT NULL
AND cid IS NOT NULL
AND country = 'Australia'
PRIMARY KEY (cid, country, birthday);
```

The results of this query:

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

are:

<table>
<thead>
<tr>
<th>cid</th>
<th>country</th>
<th>birthday</th>
<th>age</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1c526849-d3a2-42a3-bcf9-7903c80b3d16</td>
<td>Australia</td>
<td>1998-12-23</td>
<td>19</td>
<td>Kanden GROVES</td>
</tr>
<tr>
<td>96c4c40d-58c8-4710-b73f-681e9b1f70ae</td>
<td>Australia</td>
<td>1989-04-20</td>
<td>29</td>
<td>Benjamin DYBALL</td>
</tr>
</tbody>
</table>

(2 rows)

The following materialized view `cyclist_by_birthday_and_age19` uses the base table `cyclist_base`. The `WHERE` clause ensures that only rows whose `birthday` and `cid` columns are non-NULL and `age` equals the value 19 are added to the materialized view.

```cql
CREATE MATERIALIZED VIEW IF NOT EXISTS cycling.cyclist_by_birthday_and_age19 AS
SELECT age, cid, birthday, country, name
FROM cycling.cyclist_base
WHERE birthday IS NOT NULL
AND cid IS NOT NULL
AND age = 19
PRIMARY KEY (cid, birthday, age);
```

The results of this query:

```cql
SELECT *
FROM cycling.cyclist_by_birthday_and_age19;
```
FROM cycling.cyclist_by_birthday_and_age19;

are:

<table>
<thead>
<tr>
<th>cid</th>
<th>birthday</th>
<th>age</th>
<th>country</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>410919ef-bd1b-4efa-8256-b0fd8ab67029</td>
<td>1999-01-04</td>
<td>19</td>
<td>Uzbekistan</td>
<td>Iskandarbek SHODIEV</td>
</tr>
<tr>
<td>1c526849-d3a2-42a3-bcf9-7903c80b3d16</td>
<td>1998-12-23</td>
<td>19</td>
<td>Australia</td>
<td>Kanden GROVE</td>
</tr>
</tbody>
</table>

(2 rows)

The following materialized view `cyclist_by_age_birthday_cid` uses the base table `cyclist_base_ext`. The `WHERE` clause ensures that only rows whose `age`, `birthday`, and `cid` columns are non-NULL are added to the materialized view.

```
CREATE MATERIALIZED VIEW IF NOT EXISTS cycling.cyclist_by_age_birthday_cid AS
SELECT age, cid, birthday, country, name
FROM cycling.cyclist_base_ext
WHERE age IS NOT NULL
    AND birthday IS NOT NULL
    AND cid IS NOT NULL
PRIMARY KEY (age, birthday, cid);
```

The results of this query:

```
SELECT *
FROM cycling.cyclist_by_age_birthday_cid
WHERE age = 19;
```

are:

<table>
<thead>
<tr>
<th>cid</th>
<th>birthday</th>
<th>age</th>
<th>country</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>410919ef-bd1b-4efa-8256-b0fd8ab67029</td>
<td>1999-01-04</td>
<td>19</td>
<td>Uzbekistan</td>
<td>Iskandarbek SHODIEV</td>
</tr>
<tr>
<td>1c526849-d3a2-42a3-bcf9-7903c80b3d16</td>
<td>1998-12-23</td>
<td>19</td>
<td>Australia</td>
<td>Kanden GROVE</td>
</tr>
</tbody>
</table>

(2 rows)

The results of this query using a `WHERE` clause for two values:

```
SELECT *
FROM cycling.cyclist_by_age_birthday_cid
WHERE age = 19
    AND birthday = '1998-12-23';
```

are:

<table>
<thead>
<tr>
<th>age</th>
<th>birthday</th>
<th>cid</th>
<th>country</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>1998-12-23</td>
<td>1c526849-d3a2-42a3-bcf9-7903c80b3d16</td>
<td>Australia</td>
<td>Kanden GROVES</td>
</tr>
</tbody>
</table>
Notice that clustering columns must still be included in order. This query violates the rule:

```sql
SELECT *
FROM cycling.cyclist_by_age_birthday_cid
WHERE birthday = '1998-12-23';
```

Result:

```
cyclist_by_age-mv.cql:195: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."
```

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 IF NOT EXISTS 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
```
This query retrieves the rows where the age is 21:

```
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 rows)

**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.
- 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' ]
[ [ AND ] OPTIONS = { option_map } ] ];
```

**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><code>Italic</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>'Literal string'</td>
<td>Single quotation ( <code> </code> ) 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 conventions</th>
<th>Description</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>

### role_name

- Use a unique name for the role. DataStax Enterprise forces all names to lowercase; enclose in quotes to preserve case or use special characters in the name.

- To automatically map external users to roles with DSE Unified Authenticator, the role name must exactly match the LDAP group name, including case.

### 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, or DSE Unified Authenticator.

  Default: false.

### PASSWORD

- Enclose the password in single quotes. *Internal authentication* requires a password.

- Roles for users authenticated by an external directory, such as DSE Unified Authenticator, must have login enabled with no 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.

   ```cql
   CREATE ROLE IF NOT EXISTS 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:

   \texttt{LOGIN coach}

3. Enter the password at the prompt.

   \texttt{Password:}

4. The \texttt{cqlsh} prompt includes the role name:

   \texttt{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 \texttt{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 \texttt{keyspace} administrator, that is a role that has full permission to only the cycling keyspace.

1. Create the role:

   \texttt{CREATE ROLE IF NOT EXISTS cycling_admin;}

   At this point the role has no permissions. Manage permissions using \texttt{GRANT} and \texttt{REVOKE}.

   A role can only modify permissions of another role and can only modify (\texttt{GRANT} or \texttt{REVOKE}) role permissions that it also has.

2. Assign the role full access to the cycling keyspace:

   \texttt{GRANT ALL PERMISSIONS ON KEYSSPACE cycling TO cycling_admin;}

3. Now assign the role to the coach.

   \texttt{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.

   \texttt{LIST ALL PERMISSIONS OF coach;}

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 \texttt{ALTER} to change a role's password:

\texttt{ALTER ROLE sandy WITH PASSWORD = 'bestTeam';}

\textbf{CREATE SEARCH INDEX}

Defines a new search index for an existing table. Automatically creates the search index schema and configuration, then generates an index.
Command available only on DSE Search nodes. Running search index management commands on large datasets can take longer than the CQLSH default timeout of 10 minutes. Increase the CQLSH client timeout as required.

This command runs with a consistency level of LOCAL_QUORUM because this command is only executed in the current datacenter.

**Synopsis**

```
CREATE SEARCH INDEX [ IF NOT EXISTS ] ON [keyspace_name.]table_name
  [ WITH [ COLUMNS column_list { option : value } [ , ... ] ]
  [ [ AND ] PROFILES profile_name [ , ... ] ]
  [ [ AND ] CONFIG { option:value } [ , ... ] ]
  [ [ AND ] OPTIONS { option:value } [ , ... ] ] ] ;
```

If the CREATE SEARCH INDEX statement specifies no options, all columns are indexed using the default values.

**Table 65: 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>'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, 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>

**EBNF**

**EBNF** syntax:

```
createSearchIndex ::= 'CREATE' 'SEARCH' 'INDEX' ('IF' 'NOT' 'EXISTS')?
  'ON' tableName
  ('WITH' indexOptions)?
```
Cassandra Query Language (CQL) reference

```
tableName ::= (keyspace '.')? table
indexOptions ::= indexOption ('AND' indexOption)*
indexOption ::= 'COLUMNS' columnList
               | 'PROFILES' profileName (',' profileName)*
               | 'CONFIG' optionMap
               | 'OPTIONS' optionMap
columnList ::= column (',' column)*
column ::= (columnName | '*')('{' optionMap '}')?
optionMap ::= '{' (optionName ':' optionValue (',' optionName ':'
               optionValue)* )? '}'
```

Railroad diagram:

Figure 23: createSearchIndex
Figure 24: tableName
Figure 25: indexOptions

![Diagram showing indexOptions with an AND connection]
Figure 27: columnList
Figure 28: column
COLUMNS
Defines which fields to include in index, sets index type, and creates non-tokenized fields for faceted search.

COLUMNS column_list { copyField : true | false },
column_list { docValues : true | false },
column_list { excluded : true | false },
column_list { indexed : true | false }

When the COLUMNS option is used, any column not listed is excluded from the index by default, except PRIMARY KEY columns which must be indexed.

column_list
A comma-separated list or * (for all columns). You can include tuple fields and subfields. Any column not listed is excluded from the index by default, except PRIMARY KEY columns, which must be indexed.

- A comma-separated list of all of the column_names, tuplefield, or tuplefield.subfield to include in the search index. When a subfield is selected for inclusion, parent fields are always included.
- For each column in the column_list, optionally specify true or false for copyField, docValues, excluded, or indexed.
- Asterisk (*) to select all columns.
Cassandra Query Language (CQL) reference

For example:

COLUMNS column_name1, column_name2

COLUMNS column_name1, column_name2 {copyField:true}

copyField: (true | false)
Set to true to create a new field copied from the specified columns with type StrField. Duplicates the data from the original field into the new field. Use for columns that require both search and faceting.
Default value is false.

docValues: (true | false)
Creates a forward index on each specified column. Setting is only valid on Solr types that extend TrieField, UUIDField, and StrField. Use on columns that are sorted or grouped (faceted). Default is true for TrieField and UUIDField types and false StrField types.
Due to SOLR-7264, setting docValues to true on a boolean field in the Solr schema does not work. A workaround for boolean docValues is to use 0 and 1 with a TrieIntField.

Using spaceSavings profiles disables auto generation of DocValues.

excluded: (true | false)
When using the COLUMNS option, exclude column from index:

• true - exclude the listed columns and all fields in the columns from the index.
• false - do not exclude the listed columns from the index. You must specify columns to include to include the columns in the index. Default when not specified.

Excluded columns are not present in HTTP query results and singlePass queries.

indexed: (true | false)
When using the COLUMNS option:

• true - include the specified fields in the index. Default when not specified.
• false - exclude the specified fields from the index.

Non-indexed columns are present in HTTP query results and singlePass queries only if they are included in the Solr schema.xml file. For more information, see Solr single-pass CQL queries.

PROFILES
Apply space saving options to minimize index size on initial creation. Specify spaceSavingAll or a comma separated list of profiles to apply.

PROFILES profile_name [, profile_name, ...]

Profiles only apply to the initial index generation, and do not apply to the ALTER SEARCH INDEX SCHEMA command.

spaceSavingAll
Applies all profiles.

spaceSavingNoJoin
Excludes the hidden partition key required for joins across tables on search queries from the index. When used table joins on search index queries are not allowed.

spaceSavingSlowTriePrecision
Sets trie fields precisionStep to '0', allowing for greater space saving but slower querying.
CONFIG
Configuration options override values in the Search index config file. The CONFIG option map can pass options with this syntax:

```csharp
CONFIG { shortcut_name:value [, shortcut_name:value, ...] }
```

shortcuts
Shortcuts to configuration element values using SET:

- `autoCommitTime` Default value is 10000.
- `defaultQueryField` Name of the field. Default not set.
  - Use SET to add. Use DROP to remove.
- `directoryFactory` Can be used as an alternative to the directoryFactoryClass option. The options are:
  - `'standard'`
  - `'encrypted'`
- `filterCacheLowWaterMark` Default is 1024.
- `filterCacheHighWaterMark` Default is 2048.
- `directoryFactoryClass` Specifies the fully-qualified name of the directory factory. Use in place of the directoryFactory option for directory factories other than the standard or encrypted directory factory.
- `mergeMaxThreadCount` Must configure with mergeMaxMergeCount. The default is the number of tpc_cores as configured in cassandra.yaml.
- `mergeMaxMergeCount` Must configure with mergeMaxThreadCount. The default calculated value is:

```
max(max(<maxThreadCount * 2>, <num_tokens * 8>), <maxThreadCount + 5>)
```

where `num_tokens` is the number of token ranges to assign to the virtual node (vnode) as configured in cassandra.yaml.
- `ramBufferSize` Default is 512.
- `realtime` Default is false.

OPTIONS
Request options configure the entire request. The OPTIONS map can pass options with this syntax:

```csharp
OPTIONS { option:value [, option:value, ...] }
```

The request options are boolean values:

- `recovery`
  - `true` - if the search core is unable to load due to corrupted index, recovers it by deleting and recreating the index. The deleteAll flag is set based on the recovery flag unless deleteAll is specifically set.
  - `false` - no recovery. Default.

- `reindex`
Cassandra Query Language (CQL) reference

- true - reindexes the data. Keeps the current index (accepting reads) while the new index is building. Default.
- false - does not reindex the data.

**lenient**

- true - Silently ignores fields if you encounter an unsupported column type during schema autogeneration.
- false - Raise error if you encounter an unsupported column type during schema autogeneration.

This option is no longer needed for SpatialRecursivePrefixTreeFieldType fields, which are now automatically indexed.

**Examples**

The search index is created with the wiki.solr keyspace and table, and the specified options.

**Create search index if it does not exist**

```cql
CREATE SEARCH INDEX IF NOT EXISTS
ON wiki.solr
WITH COLUMNS id, body {excluded : false};
```

**Create real time (RT) search index, but don't reindex the data**

```cql
CREATE SEARCH INDEX
ON wiki.solr
WITH CONFIG { realtime:true }
AND OPTIONS { reindex : false };
```

**Create search index with transparent data encryption (TDE)**

```cql
CREATE SEARCH INDEX IF NOT EXISTS
ON wiki.solr
WITH COLUMNS c1,c2 {docValues:true}
AND PROFILES spaceSavingAll
AND CONFIG {directoryFactory:'encrypted'};
```

**Create search index with docValues set for all columns with a supported type**

```cql
CREATE SEARCH INDEX
ON wiki.solr
WITH COLUMNS * { docValues:true };
```

**Create search index to specify the columns to include and exclude from index**

```cql
CREATE SEARCH INDEX
ON wiki.solr
WITH COLUMNS field1 { indexed:true }, field2 { indexed:false };
```

Non-indexed columns are included in present in HTTP query results and single pass query results. To exclude, use the excluded option.

**Create search index with controls for tuple and UDT fields**

```cql
CREATE SEARCH INDEX
ON wiki.solr
```
WITH COLUMNS tuplefield.field1 {docValues:true};

Parent fields are included since the subfield is selected for inclusion.
Create search index to specify the columns to exclude from HTTP query results and single pass queries

CREATE SEARCH INDEX
ON wiki.solr
WITH COLUMNS field1 { excluded:true }, field2 { excluded:false };

Excluded columns are not present in HTTP query results, but non-indexed columns are included.
Create search index with space saving no join option

CREATE SEARCH INDEX
ON wiki.solr
WITH PROFILES spaceSavingNoJoin;

The example avoids indexing the _partitionKey field. See Identifying the partition key.

CREATE TABLE

Creates a new table in the selected keyspace. Use IF NOT EXISTS to 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' ] ] ;
```

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>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>
</tbody>
</table>
Cassandra Query Language (CQL) reference

<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>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><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>

**column_definition**

Enclosed in parentheses 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 a 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 orders data (clustering columns) within a partition. Primary keys cannot have the following data types: 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 parentheses. 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 CREATE TABLE (or ALTER TABLE) CQL statement, use a `WITH` clause to define table property options. Separate multiple values with `AND`.

```cql
CREATE 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 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**

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’**

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, limited to the same datacenter as the coordinator node.

This option is deprecated and should be set to 0.0.

**default_time_to_live**
Cassandra Query Language (CQL) reference

TTL (Time To Live) in seconds, where zero is disabled. The maximum configurable value is 63072000 (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.

gc_grace_seconds
Seconds after data is marked with a tombstone (deletion marker) before it is eligible for garbage-collection. Default value: 864000 (10 days). The default value allows time for 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, DataStax Enterprise 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.
Manages the table repair settings using the following syntax:

```javascript
{ 'enabled' : value, 'deadline_target_sec' : seconds }
```

- **enabled**: Set to `true` or `false` to enable/disable NodeSync on the table.
  - Default: `true`.

- **deadline_target_sec**: Specify the maximum number of seconds to validate all segments of the table. Set to less than the `gc_grace_seconds` to avoid resurrecting deleted data. DataStax recommends using the `nodetool nodesyncservice ratesimulator` to calculate the appropriate setting.
  - Default: Highest value of `gc_grace_seconds` or 4 days.

**read_repair_chance**

The probability that a successful read operation triggers a read repair.

This option is deprecated and should be set to 0.0.

**speculative_retry**

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.

**Compaction Strategy**

<table>
<thead>
<tr>
<th>LCS</th>
<th>STCS</th>
<th>TWCS</th>
<th>DTCS (deprecated)</th>
</tr>
</thead>
</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.

```javascript
compression = {
    'class' : 'compression_algorithm_name',
    'chunk_length_in_kb' : 'value',
    'crc_check_chance' : 'value',
}
```

**class**
Cassandra Query Language (CQL) reference

Sets the compressor name. DataStax Enterprise provides the following built-in classes:

- LZ4Compressor
- SnappyCompressor
- DeflateCompressor

Use only compression implementations bundled with DSE.

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 bit rot 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.

Syntax uses a simple JSON format:

```json
compaction = {
    'class' : 'compaction_strategy_name',
    'property_name' : value [, ..., ]
}
```

where the compaction_strategy_name is SizeTieredCompactionStrategy, TimeWindowCompactionStrategy, or LeveledCompactionStrategy.

Use only compaction implementations bundled with DSE. See How is data maintained? for more details.

Common properties

The following properties apply to all compaction strategies.

```json
compaction = {
    'class' : 'compaction_strategy_name',
    'enabled' : (true | false),
    'log_all' : (true | false),
    'only_purge_repaired_tombstone' : (true | false),
    'tombstone_threshold' : ratio,
    'tombstone_compaction_interval' : sec,
}
```
Cassandra Query Language (CQL) reference

enabled
Enable background compaction.

- true runs minor compactions.
- false disables minor compactions.

Use `nodetool enableautocompaction` to start running compactions.

Default: true

log_all
Activates advanced logging for the entire cluster.

Default: false

only_purge_repaired_tombstone
Enabling this property prevents data from resurrecting when repair is not run within the `gc_grace_seconds`. When it has been a long time between repairs, the database keeps all tombstones.

- true: Only allow tombstone purges on repaired SSTables.
- false: Purge tombstones on SSTables during compaction even if the table has not been repaired.

Default: false

tombstone_threshold
The ratio of garbage-collectable tombstones to all contained columns. If the ratio exceeds this limit, compactions start only on that table to purge the tombstones.

Default: 0.2

tombstone_compaction_interval
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 based 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.

Default: 86400 (1 day)

unchecked_tombstone_compaction
Setting to true allows tombstone compaction to run without pre-checking which tables are eligible for the operation. Even without this pre-check, DSE checks an SSTable to make sure it is safe to drop tombstones.

Default: false

min_threshold
The minimum number of SSTables to trigger a minor compaction.

Not used in `LeveledCompactionStrategy`.

Default: 4

max_threshold
The maximum number of SSTables before a minor compaction is triggered.

Not used in `LeveledCompactionStrategy`.

Default: 32

SizeTieredCompactionStrategy
The compaction class `SizeTieredCompactionStrategy (STCS)` triggers a minor compaction when a table meets the `min_threshold`. Minor compactions do not involve all the tables in a keyspace. See `SizeTieredCompactionStrategy (STCS)`.

Default compaction strategy.
The following properties only apply to SizeTieredCompactionStrategy:

```json
compaction = { 'class' : 'SizeTieredCompactionStrategy', 'bucket_high' : factor, 'bucket_low' : factor, 'min_sstable_size' : int }
```

**bucket_high**
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 \([\text{average-size} \times \text{bucket_high}]\).
Default: 1.5

**bucket_low**
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 \([\text{average-size} \times \text{bucket_high}]\).
Default: 0.5

**min_sstable_size**
STCS groups SSTables into buckets. The bucketing process groups SSTables that differ in size by less than 50%. This bucketing process is too fine-grained for small SSTables. If your SSTables are small, use this option to define a size threshold in MB below which all SSTables belong to one unique bucket.
Default: 50 (MB)

The **cold_reads_to_omit** property for **SizeTieredCompactionStrategy (STCS)** is no longer supported.

**TimeWindowCompactionStrategy**

The compaction class **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 (TWCS)**.

All of the properties for STCS are also valid for TWCS.

The following properties apply only to **TimeWindowCompactionStrategy**:

```json
compaction = { 'class' : 'TimeWindowCompactionStrategy', 'compaction_window_unit' : days, 'compaction_window_size' : int, 'split_during_flush' : (true | false) }
```

**compaction_window_unit**
Time unit used to define the bucket size. The value is based on the Java **TimeUnit**. For the list of valid values, see the Java API **TimeUnit** page located at https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TimeUnit.html.
Default: days

**compaction_window_size**
Units per bucket.
Default: 1

**split_during_flush**
Prevents mixing older data from repairs and hints with newer data from the current time window. During a flush operation, determines whether data partitions are split based on the configured time window.
false - the data partitions are not split based on the configured time window.

true - ensure that data repaired by NodeSync is placed in the correct TWCS window. Enable split_during_flush when using NodeSync with TWCS or when running node repairs.

Default: false

During the flush operation, the data is split into a maximum of 12 windows. Each window holds the data in a separate SSTable. If the current time is $t_0$ and each window has a time duration of $w$, the data is split in the SSTables as follows:

- SSTable 0 contains data for the time period $< t_0 - 10 \times w$
- SSTables 1 to 10 contain data for the 10 equal time periods from $(t_0 - 10 \times w)$ through to $(t_0 - 1 \times w)$
- SSTable 11, the 12th table, contains data for the time period $> t_0$

LeveledCompactionStrategy

The compaction class 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).

For more guidance, see When to Use Leveled Compaction and Leveled Compaction blog.

The following properties only apply to LeveledCompactionStrategy:

```
compaction = {
  'class' : 'LeveledCompactionStrategy,
  'sstable_size_in_mb' : int }
```

sstable_size_in_mb

The target size for SSTables that use the LeveledCompactionStrategy. Although SSTable sizes should be less or equal to sstable_size_in_mb, it is possible that compaction could produce a larger SSTable during compaction. This occurs when data for a given partition key is exceptionally large. The DSE database does not split the data into two SSTables.

Default: 160

DateTieredCompactionStrategy (deprecated)

Use TimeWindowCompactionStrategy instead.

Stores data written within a certain period of time in the same SSTable.

base_time_seconds

The size of the first time window.

Default: 3600

max_sstable_age_days (deprecated)

DSE does not compact SSTables if its most recent data is older than this property. Fractional days can be set.

Default: 1000

max_window_size_seconds

The maximum window size in seconds.

Default: 86400

timestamp_resolution

Units, MICROSECONDS or MILLISECONDS, to match the timestamp of inserted data.

Default: MICROSECONDS

Table keywords

CLUSTERING ORDER BY ( column_name ASC | DESC)
Cassandra Query Language (CQL) reference

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

**ID**

If a table is accidentally dropped with `DROP TABLE`, use this option to recreate the table and run a commit log replay to retrieve the data.

**Examples**

**Creating a table with a frozen UDT**

Create the `race_winners` table that has a frozen user-defined type (UDT):

```cql
CREATE TABLE IF NOT EXISTS cycling.race_winners (
  cyclist_name FROZEN<fullname>,
  race_name text,
  race_position int,
  PRIMARY KEY (race_name, race_position)
);
```

See [Creating a user-defined type](#) for information on creating UDTs. 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.

**Creating a table with UUID as the primary key**

Create the `cyclist_name` table with UUID as the primary key:

```cql
CREATE TABLE IF NOT EXISTS 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 IF NOT EXISTS 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 IF NOT EXISTS cycling.rank_by_year_and_name (
  race_year int,
  race_name text,
  cyclist_name text,
  rank int,
  PRIMARY KEY ((race_year, race_name), rank)
);
```

**Creating a table with a CDC log**
Create a change data capture log for the `cyclist_id` table:

```cql
CREATE TABLE IF NOT EXISTS cycling.cyclist_id (
    lastname text,
    firstname text,
    age int,
    id UUID,
    PRIMARY KEY ((lastname, firstname), age)
);
```

CDC logging must be enabled in `cassandra.yaml`.

Before enabling CDC logging, have a plan for moving and consuming the log information. After the disk space limit is reached, writes to CDC-enabled tables are rejected until more space is freed. See Change-data-capture (CDC) space settings for information about available CDC settings.

Storing data in descending order

The following example shows a table definition that stores the categories with the highest points first.

```cql
CREATE TABLE IF NOT EXISTS cycling.cyclist_category (
    category text,
    points int,
    id UUID,
    lastname text,
    PRIMARY KEY (category, points)
) WITH CLUSTERING ORDER BY (points DESC);
```

Restoring from the table ID for commit log replay

Recreate a table with its original ID to facilitate restoring table data by replaying commit logs:

```cql
CREATE TABLE IF NOT EXISTS cycling.cyclist_emails (
    userid text PRIMARY KEY,
    id UUID,
    emails set<text>
) WITH ID = '1bb7516e-b140-11e8-96f8-529269fb1459';
```

To retrieve a table's ID, query the `id` column of `system_schema.tables`. For example:

```cql
SELECT id
FROM system_schema.tables
WHERE keyspace_name = 'cycling'
AND table_name = 'cyclist_emails';
```

To perform a point-in-time restoration of the table, see Restoring a backup to a specific point-in-time.

**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).

UDTs cannot contain counter fields.

**Synopsis**

```cql
CREATE TYPE [ IF NOT EXISTS ] [keyspace_name].type_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>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>--</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.

```
CREATE TYPE IF NOT EXISTS cycling.basic_info (  
    birthday timestamp,  
    nationality text,  
    height text,  
    weight 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.

This example creates a user-defined type (UDT) `cycling.basic_info_withTTL` that consists of personal data about an individual cyclist, which includes the `next_race` column that will be set with a timestamp and time-to-live (TTL).

```cql
CREATE TYPE IF NOT EXISTS cycling.basic_info_expire {
    birthday timestamp,
    nationality text,
    height text,
    weight text,
    next_race text
};
```

To insert an entire row of data into a table using a timestamp and TTL, specify the values with an `INSERT` command:

```cql
INSERT INTO cycling.basic_info_TTL_expire (
    id, lastname, basics
) VALUES (
    e7ae5cf3-d358-4d99-b900-85902fda9bb0,
    'FRAME',
    {
        birthday : '1993-06-18',
        nationality : 'New Zealand',
        weight : '175',
        height : '72',
        next_race : 'Amgen Tour of California'
    }
) USING TIMESTAMP 100 AND TTL 10000;
```

To insert a single value with a TTL, use a `UPDATE` command:

```cql
UPDATE cycling.basic_info_TTL_expire
    USING TTL 100
SET basics.next_race = 'Tour de France'
WHERE id = e7ae5cf3-d358-4d99-b900-85902fda9bb0;
```

To check the write time and TTL values of a UDT use the WRITETIME and TTL functions in a `SELECT` command:

```cql
SELECT WRITETIME(basics), TTL(basics)
FROM cycling.basic_info_TTL_expire
WHERE id = e7ae5cf3-d358-4d99-b900-85902fda9bb0;
```

A string with a single quote in a UDT text field is returned with two quotes in queries and `COPY` commands. For example, if you store the string `Single ' quote` in a UDT text field, the string is returned as `Single '" Quote`.

**CREATE USER (Deprecated)**

`CREATE USER` is supported for backwards compatibility only. Authentication and authorization for DataStax Enterprise 5.0 and later are based on `ROLES`, and use `CREATE ROLE` instead.
CREATE USER defines a new database user account. By default, users' accounts do not have superuser status. Only a superuser can issue CREATE USER requests. See CREATE ROLE for more information about SUPERUSER and NOSUPERUSER.

User accounts are required for logging in under internal authentication and authorization. Enclose the user name in single quotation marks if it contains non-alphanumeric characters. You cannot recreate an existing user. To change the superuser status or password, use ALTER USER.

**Synopsis**

```
CREATE USER [ IF NOT EXISTS ] user_name
WITH PASSWORD user_password
[ ( SUPERUSER | NOSUPERUSER ) ] ;
```

<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 ( [ ] ) separates alternative elements. Type any one of the elements. Do not type the 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; &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**

Creating internal user accounts

Use WITH PASSWORD to create a user account for internal authentication. Enclose the password in single quotation marks.

```
CREATE USER spillman WITH PASSWORD 'Niner27';
CREATE USER akers WITH PASSWORD 'Niner2' SUPERUSER;
```
CREATE USER boone WITH PASSWORD 'Niner75' NOSUPERUSER;

If internal authentication has not been set up, WITH PASSWORD is not required.

CREATE USER test NOSUPERUSER;

Creating a user account conditionally

You can test that the user does not have an account before attempting to create one. Attempting to create an existing user results in an invalid query condition unless the IF NOT EXISTS option is used. If the option is used, the statement will be a no-op if the user exists.

1. Login with the default account (this account uses QUOROM, only use to create a new account once).

```
$ cqlsh -u cassandra -p cassandra
```

The DSE and component versions display, and then the command prompt.

```
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>
```

2. Create the new user:

```
CREATE USER newuser WITH PASSWORD 'password';
```

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. Tombstones are created, causing stale data to be read until removed by a compaction.

**Synopsis**

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

**Table 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><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>
Cassandra Query Language (CQL) reference

<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>'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:
- 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 in specified columns from a row
The examples in this section use this table:

```cql
CREATE TABLE IF NOT EXISTS cycling.cyclist_name (
    id UUID PRIMARY KEY,
    lastname text,
    firstname text
);
```

<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>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>TIRALONGO</td>
</tr>
<tr>
<td>e7cd5752-bc0d-4157-a80f-7523add8dbc</td>
<td>Anna</td>
<td>VAN DER BREGGEN</td>
</tr>
</tbody>
</table>

(6 rows)

Delete the data in specific columns by listing them after the DELETE command, separated by commas. Change the data in first and last name columns to null for the cyclist specified by id.

```cql
DELETE firstname, lastname
FROM cycling.cyclist_name
WHERE id = e7ae5cf3-d358-4d99-b900-85902fda9bb0;
```

Result:

<table>
<thead>
<tr>
<th>id</th>
<th>firstname</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>e7ae5cf3-d358-4d99-b900-85902fda9bb0</td>
<td>null</td>
<td>null</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>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>TIRALONGO</td>
</tr>
<tr>
<td>e7cd5752-bc0d-4157-a80f-7523add8dbc</td>
<td>Anna</td>
<td>VAN DER BREGGEN</td>
</tr>
</tbody>
</table>

(6 rows)

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 by specifying IF EXISTS.

```cql
DELETE FROM cycling.cyclist_name
WHERE id = e7ae5cf3-d358-4d99-b900-85902fda9bb0
IF EXISTS;
```

Without IF EXISTS, the command proceeds with no standard output. If IF EXISTS returns True (if a row with this primary key does exist), standard output displays a table like the following:

```
[applied]
---------
True
```

The cyclist named Alex Frame has been completely removed from the table.

<table>
<thead>
<tr>
<th>id</th>
<th>firstname</th>
<th>lastname</th>
</tr>
</thead>
</table>

CQL for DataStax Enterprise 6.7 Latest DSE version Latest 6.7 patch: 6.7.7
Delete a row based on a static column condition

The **IF** condition limits the **WHERE** clause, allowing selection based on values in non-PRIMARY KEY columns, such as first and last name. This example removes the cyclist record if the first and last names do not match.

```
DELETE FROM cycling.cyclist_name
WHERE id = fb372533-eb95-4bb4-8685-6ef61e994caa
IF firstname = 'Michael'
   AND lastname = 'Smith';
```

The results show all the applicable data. The condition was not met because the last names did not match; therefore the operation was not applied.

```
[applied] | firstname | lastname
-----------+-----------+----------
False | Michael  | MATTHEWS
```

Conditionally deleting data from a column

Use the **IF** or **IF EXISTS** clauses to conditionally delete data from columns. Deleting column data conditionally is similar to making an **UPDATE** conditionally.

Add **IF EXISTS** to the command to ensure that an operation is not performed if the specified row does not exist:

```
DELETE id FROM cycling.cyclist_id
WHERE lastname = 'JONES'
   AND firstname = 'Bram'
IF EXISTS;
```

No such row exists so the conditions returns **False** and the command fails. In this case, standard output looks like:

```
[applied]  
----------
False
```

Use **IF condition** to apply tests to one or more column values in the selected row:

```
DELETE id FROM cycling.cyclist_id
WHERE lastname = 'WELTEN'
   AND firstname = 'Bram'
IF age = 20;
```

If all the conditions return True, the standard output is the same as if **IF EXISTS** returned True. 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
```
Conditional deletions incur a non-negligible performance cost and should be used sparingly.

Deleting one or more rows

The WHERE clause specifies which row or rows to delete from a specified table.

```
DELETE FROM cycling.cyclist_name
WHERE id = 6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47;
```

The cyclist named Steven Kruikswijk was deleted from the table.

```
id                                   | firstname | lastname
--------------------------------------+-----------+-----------------
fb372533-eb95-4bb4-8685-6ef61e994caa |   Michael |        MATTHEWS
5b6962dd-3f90-4c93-8f61-eabfa4a803e2 |  Marianne |             VOS
22084bf-4860-49d6-9a4b-6b5d3a79cbfb |    Paolo  |       TIRALONGO
e7cd5752-bc0d-4157-a80f-7523add8d9cb |      Anna | VAN DER BREGGEN
```

(4 rows)

To delete more than one row, use the keyword IN and supply a list of comma-separated values in parentheses:

```
DELETE FROM cycling.cyclist_name
WHERE id IN (
  5b6962dd-3f90-4c93-8f61-eabfa4a803e2, 22084bf-4860-49d6-9a4b-6b5d3a79cbfb
);
```

Using IN predicates on non-primary keys is not supported.

CQL supports an empty list of values in the IN clause, which is useful in Java Driver applications.

The cyclists Marianne Vos and Paolo Tiralongo were deleted from the table.

```
id                                   | firstname | lastname
--------------------------------------+-----------+-----------------
fb372533-eb95-4bb4-8685-6ef61e994caa |   Michael |        MATTHEWS
e7cd5752-bc0d-4157-a80f-7523add8d9cb |      Anna | VAN DER BREGGEN
```

(2 rows)

Deleting old data using a timestamp

The TIMESTAMP is an integer representing microseconds. Use TIMESTAMP to identify data to delete. The query deletes any rows from a partition older than the timestamp.

```
DELETE firstname, lastname
FROM cycling.cyclist_name
USING TIMESTAMP 1318452291034
WHERE lastname = 'VOS';
```

Deleting from a collection set, list, or map

To delete all elements from a set for a race, specify the column_name, which is sponsorship:

```
DELETE sponsorship
FROM cycling.race_sponsors
```
WHERE race_name = 'Giro d’Italia';

To delete an element from a list, specify the column_name followed by the list index position in square brackets:

DELETE sponsorship[2]
FROM cycling.race_sponsors
WHERE race_year = 2018
AND race_name = 'Tour de France';

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.

Using an UPDATE command with a subtraction operator to remove a list element in a safer and faster manner is recommended. See List fields.

To delete an element from a map, specify the column_name followed by the key of the element in square brackets:

DELETE teams[2014]
FROM cycling.cyclist_teams
WHERE id = 5b6962dd-3f90-4c93-8f61-eabfa4a803e2;

DROP AGGREGATE

Deletes a user-defined aggregate (UDA) from a keyspace.

Synopsis

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

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>
<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>
</tbody>
</table>
Cassandra Query Language (CQL) reference

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>

argument_list

Comma separated list of arguments.

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 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>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>
</tbody>
</table>
Cassandra Query Language (CQL) reference

Syntax conventions

<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=&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>

argument_list

Comma separated list of arguments.

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_name.]index_name;
```

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>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>'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
Drop the index `rank_idx` from the `cycling.rank_by_year_and_name` table.

```
DROP INDEX IF EXISTS cycling.rank_idx;
```

**DROP KEYSAPCE**
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**

```
DROP KEYSPACE [ IF EXISTS ] keyspace_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><code>italics</code></td>
<td>Variable value. Replace with a user-defined value.</td>
</tr>
<tr>
<td><code>[]</code></td>
<td>Optional. Square brackets <code>([])</code> surround optional command arguments. Do not type the square brackets.</td>
</tr>
<tr>
<td><code>{ }</code></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><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 <code>( ’ )</code> 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 <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>
Example
Drop the cycling keyspace:

```cql
DROP KEYSPACE IF EXISTS 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;
```

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></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>

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 the `cyclist_by_age` materialized view from the `cyclist` keyspace.

```
DROP MATERIALIZED VIEW IF EXISTS 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 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>
<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
Drop the team manager role.

```
DROP ROLE IF EXISTS team_manager;
```

**DROP SEARCH INDEX**

Drops a search index from a table and optionally deleted the resources associated with the search index.

Command available only on DSE Search nodes. Running search index management commands on large datasets can take longer than the CQLSH default timeout of 10 minutes. Increase the CQLSH client timeout as required.

**Synopsis**

```
DROP SEARCH INDEX ON [keyspace_name.]table_name
OPTIONS { option : value } [ , { option : value } ... ] ;
```

**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=&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>

**EBNF**

**EBNF syntax:**

```
dropSearchIndex ::= 'DROP' 'SEARCH' 'INDEX'
                 'ON' tableName
                 ('WITH' 'OPTIONS' optionMap)?
```
CQL for DataStax Enterprise 6.7 Latest DSE version Latest 6.7 patch: 6.7.7

**Table Definition:**
```
tableName ::= (keyspace ',')? table
optionMap ::= '{' (key ':' value (',' key ':' value)*)? '}
```

**Railroad diagram:**

Figure 30: dropSearchIndex
Figure 31: tableName

```
| keyspace | . | tableName |
```

Cassandra Query Language (CQL) reference

Page 382
The request options are boolean values:

**deleteResources**
- true - deletes the resources associated with the search index.
- false - does not delete the resources.

Default: true

**deleteDataDir**
- true - deletes index data and any other artifacts in the solr.data directory. It does **not** delete data from the database.
- false - does not delete index data.

Default: false

**Examples**
The search index is dropped for the wiki.solr keyspace and table, and the specified options.
Delete the resources associated with the search index

```
DROP SEARCH INDEX ON wiki.solr;
```

Keep the resources associated with the search index

```
DROP SEARCH INDEX ON wiki.solr WITH OPTIONS { deleteResources:false };
```

**DROP TABLE**

Immediate, irreversible removal of a table, including all data contained in the table.

Drop all materialized views associated with the table before dropping the table. An error message lists any materialized views that are based on the table:

```
InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot drop table when materialized views still depend on it (cycling.{cyclist_by_age})"
```

**Synopsis**

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

**Table 77: 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>Lowcase</strong></td>
<td>Not literal.</td>
</tr>
<tr>
<td><code>?</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>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><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>
Example
Drop the cyclist_name table:

DROP TABLE IF EXISTS cycling.cyclist_name;

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

DROP TYPE [ IF EXISTS ] [ keyspace_name.]type_name ;

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>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=&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

Attempting to drop a type that is in use by a table:

```
DROP TYPE IF EXISTS 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 that uses the type:

```
DROP TABLE IF EXISTS cycling.cyclist_stats;
```

Drop the type:

```
DROP TYPE IF EXISTS cycling.basic_info;
```

**DROP USER (Deprecated)**

Removes a user.

DROP USER is supported for backwards compatibility. Authentication and authorization for DataStax Enterprise 5.1 and later are based on ROLES, and use DROP ROLE.

Synopsis

```
DROP USER [ IF EXISTS ] user_name;
```

Table 79: 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><em>Lowercase</em></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><code>'literal string'</code></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 ({ }) 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>[---]</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

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, and functions.

Permissions apply immediately, even to active client sessions.

Enable authentication and authorization to control access to database resources. See Enabling DSE Unified Authentication.

### Synopsis

```
GRANT permission
ON object
TO role_name ;
```

## Table 80: 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>&quot;Literal string&quot;</code></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 conventions</th>
<th>Description</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> </code>&lt; ``) 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=&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>

### permission

Type of access a role has on a database resource. Use **ALL PERMISSIONS** or a comma separated list of permissions.

Permissions are resource specific as follows:

- **Data** - **ALL PERMISSIONS** or **ALTER**, **AUTHORIZE** [FOR permission_list], **CREATE**, **DESCRIBE**, **DROP**, **MODIFY**, and **SELECT**

- **Functions (and aggregates)** - **ALL PERMISSIONS** or **ALTER**, **AUTHORIZE** [FOR permission_list], **CREATE**, and **DROP**

- **Search indexes** - **AUTHORIZE** [FOR permission_list], **SEARCH.ALTER**, **SEARCH.COMMIT**, **SEARCH.CREATE**, **SEARCH.DROP**, **SEARCH.REBUILD**, and **SEARCH.RELOAD**

- **Roles** - **ALL PERMISSIONS** or **ALTER**, **AUTHORIZE** [FOR permission_list], **CREATE**, **DESCRIBE**, **DROP**, **PROXY.EXECUTE**, and **PROXY.LOGIN**

- **JMX (MBeans)** - **ALL PERMISSIONS** or **AUTHORIZE** [FOR permission_list], **DESCRIBE**, **EXECUTE**, **MODIFY**, and **SELECT**

- **Remote procedure calls (RPC)** - **ALL PERMISSIONS** or **AUTHORIZE** [FOR permission_list], **EXECUTE**, **MODIFY**, and **SELECT**

- **Authentication schemes** - **ALL PERMISSIONS** or **AUTHORIZE** [FOR permission_list] and **EXECUTE**

- **Spark workpools** - **ALL PERMISSIONS** or **AUTHORIZE** [FOR permission_list], **CREATE**, and **DESCRIBE**

- **Spark submissions** - **ALL PERMISSIONS** or **AUTHORIZE** [FOR permission_list], **DESCRIBE**, and **MODIFY**

To manage access control the role must have authorize permission on the resource for the type of permission. When **AUTHORIZE** is granted without specifying **FOR permission**, the role can manage all permissions on the object.

### resource_name

DataStax Enterprise database objects on which permissions are applied. Database resources have modelled hierarchy, the permission on a top level object gives the role the same permission on the objects ancestors. Identify the resource using the following keywords:

- **Data** - **ALL KEYSPACES > KEYSSPACE keyspace_name > TABLE table_name > 'filtering_data' ROWS IN table_name**

- **Function (including aggregates)** - **ALL FUNCTIONS, ALL FUNCTIONS IN KEYSSPACE keyspace_name, and FUNCTION keyspace_name.function_name( argument_types)**
• **Search indexes** - ALL SEARCH INDICES > SEARCH KEYSPACE `keyspace_name` > SEARCH INDICES `{keyspace_name}.table_name`

• **JMX MBeans** - ALL MBEANS > MBEAN `mbean_name` and MBEANS `pattern`

• **Remote procedure calls (RPC)** - ALL REMOTE CALLS > REMOTE METHOD `name` | REMOTE OBJECT `name`

• **Roles** - ALL ROLES > ROLE `role_name`

• **Authentication schemes** - ALL SCHEMES > LDAP | KERBEROS | INTERNAL

• **Analytic applications**

  # Workpools - ANY WORKPOOL > WORKPOOL '{dc_name}.*' > WORKPOOL '{dc_name}.workpool_name'

  # Submissions - ANY SUBMISSION > ANY SUBMISSION IN WORKPOOL '{datacenter_name}.*' >
  '{datacenter_name}.workpool_name' > SUBMISSION ID

**Examples**

In most environments, user authentication is handled by a plug-in that verifies users credentials against an external directory service such as LDAP. For simplicity, the following examples use internal users.

**Manage object permissions**

Use `AUTHORIZE` to allow a role to manage access control of specific resources.

- **Allow role to grant any permission type, including `AUTHORIZE`, on all objects in the cycling keyspace:**

  ```cql
  GRANT AUTHORIZE
  ON KEYSpace cycling
  TO cycling_admin;
  ```

  This makes the role a superuser in the cycling keyspace because roles can modify their own permissions as well as roles that they inherit permissions from.

  - **Allow the `sam` role to assign permission to run queries and change data in the cycling keyspace:**

    ```cql
    GRANT AUTHORIZE FOR SELECT, MODIFY
    ON KEYSpace cycling
    TO sam;
    ```

    The `sam` role cannot grant other permissions such as `AUTHORIZE`, `AUTHORIZE FOR ...`, `ALTER`, `CREATE`, `DESCRIBE`, and `DROP` to another role.

**Access to data resources**

Use the `data resource` permissions to manage access to keyspaces, tables, rows, and types.

- **Give the role `cycling_admin` all permissions to the cycling keyspace:**

  ```cql
  GRANT ALL PERMISSIONS
  ON KEYSpace cycling
  TO cycling_admin;
  ```

- **Give the role `coach` permission to perform `SELECT` statements and modify data on all tables in the cycling keyspace:**

  ```cql
  GRANT SELECT, MODIFY
  ON KEYSpace cycling
  ```
Give the role **coach** permission to perform `ALTER KEYSPACE` statements on the cycling keyspace, and also `ALTER TABLE`, `CREATE INDEX`, and `DROP INDEX` statements on all tables in the cycling keyspace:

```
GRANT ALTER
ON KEYSPACE cycling
TO coach;
```

Give the role **martin** permission to perform `SELECT` statements on rows that contain 'Sprint' in the `cycling.cyclist_category` table:

```
GRANT SELECT
ON 'Sprint' ROWS IN cycling.cyclist_category
TO martin;
```

The **filtering_data** string is case-sensitive.

To view permissions see **LIST PERMISSIONS**.

**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 columns are unset by default and do not create tombstones in the database. 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**

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

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 81: 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

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Or. A vertical bar (</td>
</tr>
<tr>
<td>Repeatable. An ellipsis ( ... ) indicates that you can repeat the syntax element as often as required.</td>
</tr>
<tr>
<td>Single quotation ( ‘ ) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.</td>
</tr>
<tr>
<td>Map collection. Braces ( { } ) enclose map collections or key value pairs. A colon separates the key and the value.</td>
</tr>
<tr>
<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>End CQL statement. A semicolon ( ; ) terminates all CQL statements.</td>
</tr>
<tr>
<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>Search CQL only: Single quotation marks ( ‘ ) surround an entire XML schema declaration.</td>
</tr>
<tr>
<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.

### 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 time-to-live (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

Specifying time-to-live (TTL) and timestamp in collections.

Insert a cyclist name using both a TTL and timestamp for a sponsorship set collection.

```cql
INSERT INTO cycling.cyclist_sponsors_expire (cyclist_name, sponsorship)
VALUES ('PRIETO, Marcela', 
{ 'Castrelli', 'Alfa Romeo' })
USING TIMESTAMP 100
AND TTL 10000;
```

- **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 teams as a set for the cyclist VOS. The set is defined in the table as teams set<text>.

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

Insert data into a map named teams that lists two recent team memberships for the cyclist VOS. The map is defined in the table as teams map<int, text>.

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

```cql
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**

List of permissions. Filter list by resource and/or role.

- Only superusers can list all permissions.
- Requires DESCRIBE permission on the target resources and roles.

**Synopsis**

```
LIST ( ALL PERMISSIONS | permission_list )
              [ ON resource_name ]
              [ OF role_name ]
              [ NORECURSIVE ] ;
```

Omit ON resource_name to display all related resources or omit OF role_name to display all role permissions.
### Table 82: 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>

### List options

**permission**

Type of access a role has on a database resource. Use ALL PERMISSIONS or a comma separated list of permissions.

Permissions are resource specific as follows:

- **Data** - ALL PERMISSIONS or ALTER, AUTHORIZE [FOR permission_list], CREATE, DESCRIBE, DROP, MODIFY, and SELECT
- **Functions (and aggregates)** - ALL PERMISSIONS or ALTER, AUTHORIZE [FOR permission_list], CREATE, and DROP
- **Search indexes** - AUTHORIZE [FOR permission_list], SEARCH.ALTER, SEARCH.COMMIT, SEARCH.CREATE, SEARCH.DROP, SEARCH.REBUILD, and SEARCH.RELOAD
- **Roles** - ALL PERMISSIONS or ALTER, AUTHORIZE [FOR permission_list], CREATE, DESCRIBE, DROP, PROXY.EXECUTE, and PROXY.LOGIN
- **JMX (MBeans)** - ALL PERMISSIONS or AUTHORIZE [FOR permission_list], DESCRIBE, EXECUTE, MODIFY, and SELECT
- **Remote procedure calls (RPC)** - ALL PERMISSIONS or AUTHORIZE [FOR permission_list], EXECUTE, MODIFY, and SELECT
- **Authentication schemes** - ALL PERMISSIONS or AUTHORIZE [FOR permission_list] and EXECUTE
Cassandra Query Language (CQL) reference

- Spark workpools - ALL PERMISSIONS or AUTHORIZE [FOR permission_list], CREATE, and DESCRIBE
- Spark submissions - ALL PERMISSIONS or AUTHORIZE [FOR permission_list], DESCRIBE, and MODIFY

To manage access control the role must have authorize permission on the resource for the type of permission. When AUTHORIZE is granted without specifying FOR permission, the role can manage all permissions on the object.

resource_name

DataStax Enterprise database objects on which permissions are applied. Database resources have modelled hierarchy, the permission on a top level object gives the role the same permission on the objects ancestors. Identify the resource using the following keywords:

- Data - ALL KEYSPACES > KEYSPACE keyspace_name > TABLE table_name > 'filtering_data' ROWS IN table_name
- Function (including aggregates) - ALL FUNCTIONS, ALL FUNCTIONS IN KEYSPACE keyspace_name, and FUNCTION keyspace_name.function_name (argument_types)
- Search indexes - ALL SEARCH INDICES > SEARCH KEYSPACE keyspace_name > SEARCH INDICES [keyspace_name.]table_name
- JMX MBeans - ALL MBEANS > MBEAN mbean_name and MBEANS pattern
- Remote procedure calls (RPC) - ALL REMOTE CALLS > REMOTE METHOD name | REMOTE OBJECT name
- Roles - ALL ROLES > ROLE role_name
- Authentication schemes - ALL SCHEMES > LDAP | KERBEROS | INTERNAL
- Analytic applications
  # Workpools - ANY WORKPOOL > WORKPOOL 'dc_name.*' > WORKPOOL 'dc_name.workpool_name'
  # Submissions - ANY SUBMISSION > ANY SUBMISSION IN WORKPOOL 'datacenter_name.*' > 'datacenter_name.workpool_name' > SUBMISSION ID

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.

List output

The list command shows the following information:

```
list all permissions of role1;

role | username | resource                      | permission | granted | restricted | grantable
------|----------|-------------------------------|------------|---------|------------|-----------
role1 | role1    | <keyspace cycling>           | DROP       | False   | True       | True
role1 | role1    | <keyspace cycling>           | AUTHORIZE  | True    | True       | False
role2 | role2    | <keyspace cycling>           | CREATE     | True    | False      | False
role3 | role3    | <keyspace cycling>           | DROP       | False   | False      | True
role3 | role3    | <keyspace cycling>           | MODIFY     | True    | False      | False
```

(5 rows)

Output columns

role
Cassandra Query Language (CQL) reference

username
The name of the role that the permission was granted or authorized on.

resource
The resource name in angle brackets.

permission
The name of the permission.

When ALL PERMISSIONS is used, each type of permission associated with the resource is granted.

granted
- True - Execute commands granted by the permission on the resource. When AUTHORIZE is granted equals true, the users with the role can grant other permissions that have granted to them on the resource to other roles.
- False - Users cannot execute the permission commands.

restricted
- True - Denies execution of the commands associated with the permission on the resource even if granted is true. If grantable is true, users with the role can still AUTHORIZE roles other than their own.
- False - Users can execute commands that have granted equal to true.

grantable
- True - Allows grant or revoke of the permission on the resource to another role, other than any of their own roles.
- False - AUTHORIZE FOR permission has not been granted.

Example

All permissions for all roles and resources
List permissions given to all the roles on all resources:

LIST ALL PERMISSIONS;
LIST ROLES;

Individual role permissions
List all permissions given to sam:

LIST ALL PERMISSIONS OF sam;

Output is:

<table>
<thead>
<tr>
<th>role</th>
<th>username</th>
<th>resource</th>
<th>permission</th>
<th>granted</th>
<th>restricted</th>
<th>grantable</th>
</tr>
</thead>
<tbody>
<tr>
<td>sam</td>
<td>sam</td>
<td>&lt;keyspace cycling&gt;</td>
<td>SELECT</td>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>sam</td>
<td>sam</td>
<td>&lt;keyspace cycling&gt;</td>
<td>MODIFY</td>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>

(2 rows)

All permissions on a resource

List all permissions on the **cyclist_name** table:

```cql
LIST ALL PERMISSIONS ON cycling.cyclist_name;
```

Output is:

<table>
<thead>
<tr>
<th>role</th>
<th>username</th>
<th>resource</th>
<th>permission</th>
<th>granted</th>
</tr>
</thead>
<tbody>
<tr>
<td>restricted</td>
<td>grantable</td>
<td>------------------------------</td>
<td>------------</td>
<td>---------</td>
</tr>
<tr>
<td>cassandra</td>
<td>cassandra</td>
<td>&lt;keyspace cycling&gt;</td>
<td>CREATE</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>cassandra</td>
<td>&lt;keyspace cycling&gt;</td>
<td>ALTER</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>cassandra</td>
<td>&lt;keyspace cycling&gt;</td>
<td>DROP</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>cassandra</td>
<td>&lt;keyspace cycling&gt;</td>
<td>SELECT</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>cassandra</td>
<td>&lt;keyspace cycling&gt;</td>
<td>MODIFY</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>cassandra</td>
<td>&lt;keyspace cycling&gt;</td>
<td>AUTHORIZE</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>cassandra</td>
<td>&lt;keyspace cycling&gt;</td>
<td>DESCRIBE</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>cassandra</td>
<td>&lt;table cycling.cyclist_name&gt;</td>
<td>ALTER</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>cassandra</td>
<td>&lt;table cycling.cyclist_name&gt;</td>
<td>DROP</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>cassandra</td>
<td>&lt;table cycling.cyclist_name&gt;</td>
<td>SELECT</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>cassandra</td>
<td>&lt;table cycling.cyclist_name&gt;</td>
<td>MODIFY</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>cassandra</td>
<td>&lt;table cycling.cyclist_name&gt;</td>
<td>AUTHORIZE</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>coach</td>
<td>cassandra</td>
<td>&lt;keyspace cycling&gt;</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>coach</td>
<td>cassandra</td>
<td>&lt;keyspace cycling&gt;</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>coach</td>
<td>cassandra</td>
<td>&lt;keyspace cycling&gt;</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>coach</td>
<td>cassandra</td>
<td>&lt;keyspace cycling&gt;</td>
</tr>
<tr>
<td></td>
<td>coach</td>
<td>&lt;keyspace cycling&gt;</td>
<td>MODIFY</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>cycling_admin</td>
<td>cycling_admin</td>
<td>&lt;keyspace cycling&gt;</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>cycling_admin</td>
<td>cycling_admin</td>
<td>&lt;keyspace cycling&gt;</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>cycling_admin</td>
<td>cycling_admin</td>
<td>&lt;keyspace cycling&gt;</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>cycling_admin</td>
<td>cycling_admin</td>
<td>&lt;keyspace cycling&gt;</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>cycling_admin</td>
<td>cycling_admin</td>
<td>&lt;keyspace cycling&gt;</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>cycling_admin</td>
<td>cycling_admin</td>
<td>&lt;keyspace cycling&gt;</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>cycling_admin</td>
<td>cycling_admin</td>
<td>&lt;keyspace cycling&gt;</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>db_admin</td>
<td>db_admin</td>
<td>&lt;keyspace cycling&gt;</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>db_admin</td>
<td>db_admin</td>
<td>&lt;keyspace cycling&gt;</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>db_admin</td>
<td>db_admin</td>
<td>&lt;all keyspaces&gt;</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>db_admin</td>
<td>db_admin</td>
<td>&lt;all keyspaces&gt;</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>db_admin</td>
<td>db_admin</td>
<td>&lt;all keyspaces&gt;</td>
</tr>
</tbody>
</table>
LIST Roles

Lists roles and shows superuser and login status. This command runs with a consistency level of QUOROM.

Roles have describe permission on their own and any inherited roles. Only roles that the current user has permission to see is listed.

Synopsis

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

Table 83: 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 require 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, 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>
**Syntax conventions**

<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=&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**

**All roles**

Show all the roles that the current role has permission to describe.

```
LIST ROLES;
```

**(Internal Role Management only) Roles assigned to a role**

Show the roles for a particular role. Sufficient privileges are required to show this information.

```
LIST ROLES OF coach;
```

Returns only the roles assigned to coach:

<table>
<thead>
<tr>
<th>role</th>
<th>username</th>
<th>resource</th>
<th>permission</th>
<th>granted</th>
<th>restricted</th>
</tr>
</thead>
<tbody>
<tr>
<td>grantable</td>
<td>----------</td>
<td>------------------------------</td>
<td>------------</td>
<td>---------</td>
<td>------------</td>
</tr>
<tr>
<td></td>
<td>coach</td>
<td>&lt;keyspace cycling&gt;</td>
<td>ALTER</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>coach</td>
<td>&lt;keyspace cycling&gt;</td>
<td>SELECT</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>coach</td>
<td>&lt;keyspace cycling&gt;</td>
<td>MODIFY</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>cycling_admin</td>
<td>&lt;keyspace cycling&gt;</td>
<td>CREATE</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>cycling_admin</td>
<td>&lt;keyspace cycling&gt;</td>
<td>ALTER</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>cycling_admin</td>
<td>&lt;keyspace cycling&gt;</td>
<td>DROP</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>cycling_admin</td>
<td>&lt;keyspace cycling&gt;</td>
<td>SELECT</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>cycling_admin</td>
<td>&lt;keyspace cycling&gt;</td>
<td>MODIFY</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>cycling_admin</td>
<td>&lt;keyspace cycling&gt;</td>
<td>AUTHORIZE</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>cycling_admin</td>
<td>&lt;keyspace cycling&gt;</td>
<td>DESCRIBE</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

(10 rows)

**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 Enterprise 5.0 and later are based on `ROLES`, and `LIST ROLES` should be used.
Cassandra Query Language (CQL) reference

Synopsis

LIST USERS ;

Table 84: 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>

Example

List the current users:

LIST USERS;

Output is:

<table>
<thead>
<tr>
<th>name</th>
<th>super</th>
</tr>
</thead>
<tbody>
<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>

REBUILD SEARCH INDEX

Rebuilds the search index. Re-constructs the index after changes to the index schema, such as adding or removing fields, adding a copy field or changing field settings. Queries using the solr_query option may return no results, partial, or old data while the index is rebuilding.
Use the RELOAD command to replace the active search index with the pending version.

Command available only on DSE Search nodes. Running search index management commands on large datasets can take longer than the CQLSH default timeout of 10 minutes. Increase the CQLSH client timeout as required.

**Synopsis**

```cql
REBUILD SEARCH INDEX ON [keyspace_name.]table_name
[ WITH OPTIONS { deleteAll : ( true | false ) } ]
```

**OPTIONS**

The request option configures the entire request to specify whether to delete the existing index:

**deleteAll**

- **true** Completely replaces the existing index. Use when changes to the schema affect the way that data is physically stored, such as changes to the field type analyzer, a column data type, etc.
- **false** (Default.) Keeps existing search index data. Reindexing happens in-place; search results will return partially incorrect results while the index is updating. Keep the current index (accepting reads) while you build the new one, then swap over to the new index after it's ready. Use when table columns are removed or added to the index.
**EBNF**

**EBNF syntax:**

```plaintext
rebuildSearchIndex ::= 'REBUILD' 'SEARCH' 'INDEX'
                    'ON' tableName
                    ('WITH' 'OPTIONS' optionMap)?
tableName        ::= (keyspace '.')? table
optionMap        ::= '{' (key ':' value (',' key ':' value)*)? '}'
```

Railroad diagram:

Figure 33: rebuildSearchIndex
Figure 34: tableName
Examples
The search index is rebuilt for the wiki.solr keyspace and table.

```
REBUILD SEARCH INDEX ON wiki.solr WITH OPTIONS { deleteAll: true };
```

**RELOAD SEARCH INDEX**
Changes made using `ALTER SEARCH INDEX CONFIG` or `ALTER SEARCH INDEX SCHEMA` are marked PENDING. Use the RELOAD command to replace the active search index with the pending version. Use the CQL shell command `DESCRIBE SEARCH INDEX` to view pending and active search index schema and config.

Run `REBUILD SEARCH INDEX` after changing the schema to reindex existing data.

Command available only on DSE Search nodes. Running search index management commands on large datasets can take longer than the CQLSH default timeout of 10 minutes. Increase the CQLSH client timeout as required.
Synopsis

RELOAD SEARCH INDEX ON [keyspace_name.]table_name ;

Table 86: 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>&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>--</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>

EBNF

EBNF syntax:

```
reloadSearchIndex ::= 'RELOAD' 'SEARCH' 'INDEX'
                  'ON' tableName
tableName ::= (keyspace '.,')? table
```

Railroad diagram:
Figure 36: reloadSearchIndex

Figure 37: tableName
**Examples**

The search index schema and configuration is reloaded for the wiki.solr keyspace.

```
RELOAD SEARCH INDEX ON wiki.solr;
```

**RESTRICT**

Use RESTRICT to deny access to a role on a data resource, that is a keyspace or table. Restrict denies access even if permission to access the resource has been granted.

RESTRICT permission always take precedence over GRANT permissions.

**Synopsis**

RESTRICT  
permission  
ON popup:keyspace_name.tuple_name  
TO role_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>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>

**permission**

A comma separated list of permissions that the role is prevented from using on the resources even if the permissions is granted. Where the permission types are: ALL PERMISSIONS or ALTER, AUTHORIZE [FOR permission_list], CREATE, DESCRIBE, DROP, MODIFY, and SELECT.

**resource**
Database object to which the permission is denied. Restriction is applied using modeled hierarchy as follows:

- ALL KEYSPACES - restricts access to every keyspace and table.
- KEYSPACE keyspace_name - restricts access on the keyspace and any table it contains
- TABLE table_name - restricts access on the table and all the data it contains

Examples
Prevent the role admin from seeing any data in the cycling keyspace:

```cql
RESTRICT MODIFY, SELECT
ON KEYSPACE cycling
TO role_admin;
```

RESTRICT ROWS
Configures the column used for row-based access control; you can only define one Primary Key column. If the column is already configured, running the RESTRICT ROWS command replaces the definition.

Use `DESCRIBE TABLE` to view the existing restrictions on the table.

Synopsis

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

Table 88: 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>
Syntax conventions | Description
---|---
' <schema> ... </schema>' | Search CQL only: Single quotation marks (’) surround an entire XML schema declaration.
@xml_entity="xml_entity_type" | Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

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 permissions. Permissions are invalidated the next time they are refreshed.

Synopsis

REVOKE permission
ON resource_name
FROM role_name ;

Table 89: 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>'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>
</tbody>
</table>
**Cassandra Query Language (CQL) reference**

### Syntax conventions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>

### permission

Type of access a role has on a database resource. Use ALL PERMISSIONS or a comma separated list of permissions.

Permissions are resource specific as follows:

- **Data** - ALL PERMISSIONS or ALTER, AUTHORIZE [FOR permission_list], CREATE, DESCRIBE, DROP, MODIFY, and SELECT
- **Functions (and aggregates)** - ALL PERMISSIONS or ALTER, AUTHORIZE [FOR permission_list], CREATE, and DROP
- **Search indexes** - AUTHORIZE [FOR permission_list], SEARCH.ALTER, SEARCH.COMMIT, SEARCH.CREATE, SEARCH.DROP, SEARCH.REBUILD, and SEARCH.RELOAD
- **Roles** - ALL PERMISSIONS or ALTER, AUTHORIZE [FOR permission_list], CREATE, DESCRIBE, DROP, PROXY.EXECUTE, and PROXY.LOGIN
- **JMX (MBeans)** - ALL PERMISSIONS or AUTHORIZE [FOR permission_list], DESCRIBE, EXECUTE, MODIFY, and SELECT
- **Remote procedure calls (RPC)** - ALL PERMISSIONS or AUTHORIZE [FOR permission_list], EXECUTE, MODIFY, and SELECT
- **Authentication schemes** - ALL PERMISSIONS or AUTHORIZE [FOR permission_list] and EXECUTE
- **Spark workpools** - ALL PERMISSIONS or AUTHORIZE [FOR permission_list], CREATE, and DESCRIBE
- **Spark submissions** - ALL PERMISSIONS or AUTHORIZE [FOR permission_list], DESCRIBE, and MODIFY

To manage access control the role must have authorize permission on the resource for the type of permission. When AUTHORIZE is granted without specifying FOR permission, the role can manage all permissions on the object.

### resource_name

DataStax Enterprise database objects on which permissions are applied. Database resources have modelled hierarchy, the permission on a top level object gives the role the same permission on the objects ancestors. Identify the resource using the following keywords:

- **Data** - ALL KEYSPACES > KEYSPACE keyspace_name > TABLE table_name > 'filtering_data' ROWS IN table_name
- **Function (including aggregates)** - ALL FUNCTIONS, ALL FUNCTIONS IN KEYSPACE keyspace_name, and FUNCTION keyspace_name.function_name( argument_types)
- **Search indexes** - ALL SEARCH INDICES > SEARCH KEYSPACE keyspace_name > SEARCH INDICES [keyspace_name.]table_name
JMX MBeans - ALL MBEANS > MBEAN mbean_name and MBEANS pattern

Remote procedure calls (RPC) - ALL REMOTE CALLS > REMOTE METHOD name REMOTE OBJECT name

Roles - ALL ROLES > ROLE role_name

Authentication schemes - ALL SCHEMES > LDAP | KERBEROS | INTERNAL

Analytic applications

# Workpools - ANY WORKPOOL > WORKPOOL 'dc_name.*' > WORKPOOL 'dc_name.workpool_name'

# Submissions - ANY SUBMISSION > ANY SUBMISSION IN WORKPOOL 'datacenter_name.*' > 'datacenter_name.workpool_name' > SUBMISSION ID

Example

The role couch can no longer perform queries or modify data in the cycling keyspace.

REVOKE SELECT, MODIFY
ON KEYSPACE cycling
FROM coach;

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 ALTER commands in the cycling keyspace:

REVOKE ALTER
ON KEYSPACE cycling
FROM coach;

SELECT

Returns data from a single table. A SELECT statement without a WHERE clause is not recommended because all rows from all partitions are returned.

DataStax recommends limiting queries to a single partition using the WHERE clause. Queries across multiple partitions can impact performance.

Synopsis

SELECT [ JSON ] 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 ) ]
### Table 90: 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>

### Selectors

Determines the columns returned in the results set.

- `column_list | 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.

- `column_name | function_name( argument_list )`

- **column_name**: Includes a column in the result set.
- **function_name( arguments )**: Execute a function on the specified argument for each row in the result set. See **CQL native functions** and **Creating a user-defined function (UDF)**.
- **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 of columns for a composite partition key.

Run `DESC TABLE table_name` to get the PRIMARY KEY definition and then `SELECT DISTINCT partition_key` FROM `table_name` to get 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 characters, 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 quotation marks; 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 specify just 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.
Cassandra Query Language (CQL) reference

<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>
<tr>
<td>CONTAINS</td>
<td>Matches a value in any type of collection. Only use on indexed collections.</td>
</tr>
<tr>
<td>CONTAINS KEY</td>
<td>Matches a key name in a map. Only use on maps with indexed keys.</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 equals 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 determine the sort order within the partition. Data is sorted by the first clustering column, the second clustering column, and so on.

**ALLOW FILTERING** overrides restrictions on filtering partition columns, clustering columns, and regular columns, but can negatively impact performance, causing read latencies. Avoid **ALLOW FILTERING** in a production environment. In test environments, use **cqlsh TRACING** to analyze performance problems.

**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 these 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 partition key 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 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:

- All clustering columns excluding the last clustering column:
  
  ```
  # Equals (=)
  # IN
  ```

- Last clustering column: All equality and inequality operators, and multi-column comparisons

Clustering column logic statements also support returning slices across clustering segments:

```cql
(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 examples and instructions.

**index_conditions**

DSE supports these index types:

**Secondary index**

Logical statements on secondary index columns support these operators:

- =
- CONTAINS on index collection types
- CONTAINS KEY on index map types

**Solr query**

Filter the query using the `solr_query` option by creating a Solr expression. See Search index syntax.

**SASI index**

To retrieve data using a SSTable Attached Secondary Index, see Using SASI.

**Additional options**

Change the scope and order of the data returned by the query.

**GROUP BY** `{column_name | function_name(arg1, arg2, ... )}`

Condenses the selected rows that share the same values for a set of columns or values returned by a function into a group.

**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 order that they are stored.

**ALLOW FILTERING**

Enables filtering without applying logic statements that identify the primary key. Avoid ALLOW FILTERING in a production environment because a full scan of the cluster is performed.

See Allow Filtering explained.

**LIMIT** `{N | PER PARTITION LIMIT N}`

Limits the number of records returned in the result set.

**Examples**

Using a column alias
Cassandra Query Language (CQL) reference

When your selection list includes functions or other complex expressions, use aliases to make the output more readable. This query uses the alias `best_rank` for `MIN(rank)`:

```cql
CAPTURE 'select_best_rank_from_rank_by_year_and_name.results';
SELECT
  MIN(rank) AS best_rank,
  cyclist_name
FROM cycling.rank_by_year_and_name
WHERE "race_name" = 'Tour of Japan - Stage 4 - Minami > Shinshu'
  AND race_year = 2014;
```

Output:

<table>
<thead>
<tr>
<th>best_rank</th>
<th>cyclist_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Daniel MARTIN</td>
</tr>
</tbody>
</table>

(1 rows)

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 `rank_by_year_and_name` table:

```cql
SELECT COUNT(*)
FROM cycling.rank_by_year_and_name;
```

Controlling the number of rows returned using LIMIT

The `LIMIT` option sets the maximum number of rows that the query returns:

```cql
SELECT cyclist_name
FROM cycling.rank_by_year_and_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 DSE 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
WHERE partition_column = value
```

Simple partition key, select multiple partitions:

```cql
WHERE partition_column IN ( value1, value2 [ , ... ] )
```

For a composite partition key, create a condition for each partition key column separated by AND:

```cql
WHERE partition_column1 = value1
  AND partition_column2 = value2 [ AND ... ]
```

Controlling the number of rows returned using PER PARTITION LIMIT

The `PER PARTITION LIMIT` option sets the maximum number of rows that the query returns from each partition.
For example, the cycling keyspace contains this table:

```cql
CREATE TABLE IF NOT EXISTS cycling.rank_by_year_and_name (  
race_year int,  
race_name text,  
cyclist_name text,  
rank int,  
PRIMARY KEY ((race_year, race_name), rank)
);
```

The table contains these rows:

<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>

(11 rows)

This query returns the top two racers for each race year and race name combination using `PER PARTITION LIMIT 2`:

```cql
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>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>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>

(8 rows)

Filtering data using `WHERE`:

The `WHERE` clause contains one or more relations that filter the rows returned by `SELECT`.

**Column specifications**

The column specification of the relation must be one of these:

- One or more members of the partition key of the table.
Cassandra Query Language (CQL) reference

- 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

This table has id as the partition key (it is the only column in the primary key, and is therefore also the partition key by default):

```cql
CREATE TABLE IF NOT EXISTS cycling.cyclist_career_teams (
    id UUID PRIMARY KEY,
    lastname text,
    teams set<text>
);
```

This query includes the partition key id value in the WHERE clause:

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

A relation that references the partition key can only use an equality operator = or IN. For more details about the IN operator, see the Examples below.

This example table contains a more complex primary key:

```cql
CREATE TABLE IF NOT EXISTS 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
    )
);
```

This query contains a WHERE clause that provides values for the primary key columns that precede the race column:

```cql
SELECT *
FROM cycling.events
WHERE year = 2017
    AND discipline = 'Cyclo-cross'
    AND start_month = 1
```
AND start_day = 1;

Output:

<table>
<thead>
<tr>
<th>year</th>
<th>discipline</th>
<th>start_month</th>
<th>start_day</th>
<th>race</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017</td>
<td>Cyclo-cross</td>
<td>1</td>
<td>1</td>
<td>DVV verzekeringen trofee - GP Sven Nys</td>
</tr>
</tbody>
</table>

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 table in the cycling keyspace (the partition key is the id column, the clustering column is race_points):

```sql
CREATE TABLE IF NOT EXISTS cycling.cyclist_points (id UUID, race_points int, firstname text, lastname text, race_title text, PRIMARY KEY (id, race_points))
```

Example query:

```sql
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                   |
```

It is possible to add ALLOW FILTERING to filter a non-indexed cluster column.

Avoid ALLOW FILTERING because it impacts performance.

The following table definition contains a clustering column named race_start_date and does not have a secondary index.

```sql
CREATE TABLE IF NOT EXISTS 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))
```
Cassandra Query Language (CQL) reference

Example query with ALLOW FILTERING:

```cql
SELECT * FROM cycling.calendar
WHERE race_start_date = '2015-06-13'
ALLOW FILTERING;
```

Output:

```
race_id | race_start_date                 | race_end_date                   | race_name
---------+---------------------------------+---------------------------------+----------------+
102 | 2015-06-13 00:00:00.000000+0000 | 2015-06-21 00:00:00.000000+0000 | Tour de Suisse
(1 rows)
```

Filtering on indexed columns

A WHERE clause in a SELECT on a table with a secondary indexed column must include at least one equality relation to the indexed column. See Indexing a column.

Using the IN operator

Use `IN`, an equals condition operator, to list multiple values for a column in a WHERE clause. This example selects the rows where `race_id` is in a list of values:

```cql
SELECT * FROM cycling.calendar
WHERE race_id IN (101, 102, 103);
```

The values in the list are separated by commas.

Using IN to filter on a compound primary key

Use an `IN` condition on the last column of a compound primary key only when it is preceded by equality conditions for all preceding columns of the primary key.

For example, examine the primary key in this table:

```cql
CREATE TABLE IF NOT EXISTS cycling.cyclist_id (
    lastname text,
    firstname text,
    age int,
    id UUID,
    PRIMARY KEY ((lastname, firstname), age)
);
```

This query contains the appropriate WHERE clause containing equality conditions for the first two columns of the primary key and an `IN` condition for the last column of the primary key:

```cql
SELECT * FROM cycling.cyclist_id
```
WHERE lastname = 'EENKHOORN'
    AND firstname = 'Pascal'
    AND age IN (17, 18);

When using `IN`, you can omit the equality test for clustering columns other than the last clustering column. This may require `ALLOW FILTERING` and should not be used in a production environment.

This table shows an example in which the `race` column is the last clustering column:

```
CREATE TABLE IF NOT EXISTS 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
    )
);
```

This query contains a `WHERE` clause with the equality condition for the `race` column (the last clustering column), an `IN` clause for the `start_month` column, and `ALLOW FILTERING` (avoid in a production environment):

```
SELECT *
FROM cycling.events
WHERE race = 'Superprestige - Hoogstraten -2017'
    AND start_month IN (1, 2)
ALLOW FILTERING;
```

CQL supports an empty list of values in the `IN` clause, which can be useful in driver applications when passing empty arrays as arguments for the `IN` clause. See Connecting to DSE clusters using DSE drivers.

**When not to use `IN`**

Typically, 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 will cause less performance latency because all query actions are performed in a single partition.

See Cassandra Query Patterns: Not using the “in” query for multiple partitions.

**Filtering on collections**

A query can retrieve a collection in its entirety. You can also index the collection column, and then use the `CONTAINS` condition in the `WHERE` clause to filter the data for a particular value in the collection, or use `CONTAINS KEY` to filter by key.
Cassandra Query Language (CQL) reference

This example features a set of text values named teams in the cyclist_career_teams table. This query filters on a value in the teams set.

```
SELECT *
FROM cycling.cyclist_career_teams
WHERE teams CONTAINS 'Rabobank-Liv Giant';
```

Output:

```
id                                   | lastname | teams
--------------------------------------+----------+---------------------------------------------
+-------------------------------------------------------------------------------
-----------------------                     +----------+------------------------------------------------------------------------
1c9ebc13-1eab-4ad5-be87-dce433216d40   |    BRAND| {'AA Drink - Leontien.nl', 'Leontien.nl', 'Rabobank-Liv Giant', 'Rabobank-Liv Woman Cycling Team'}
(1 rows)
```

The cyclist_teams table contains a map of int keys and text values named teams. The teams map keys are indexed:

```
CREATE INDEX IF NOT EXISTS team_year_idx
ON cycling.cyclist_teams ( KEYS (teams) );
```

The index allows a query to filter the map keys:

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

See Indexing a collection and .

Filtering map entries

This example adds an index for map entries.

```
CREATE INDEX IF NOT EXISTS blist_idx
ON cycling.birthday_list ( ENTRIES(blist) );
```

This method only works for maps.

This query finds all cyclists who are 23 years old based on their entry in the blist map in the birthday_list table.

```
SELECT *
FROM cycling.birthday_list
WHERE blist[ 'age' ] = '23';
```

Filtering a full frozen collection
The example in this section uses a table containing a FROZEN list collection named rnumbers. This statement creates an index, which is required for the query:

```cql
CREATE INDEX IF NOT EXISTS rnumbers_idx
ON cycling.race_starts ( FULL(rnumbers) );
```

This query retrieves the row that fully matches the collection's values, specifically a cyclist who has 39 Pro wins, 7 Grand Tour starts, and 14 Classic starts in `rnumbers`:

```cql
SELECT *
FROM cycling.race_starts
WHERE rnumbers = [39, 7, 14];
```

### Range relations

DataStax Enterprise 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.

This query constructs a filter that selects cycling calendar data whose start date is within a specified range and the `race_id` is 101. (If `race_id` were not a component of the primary key, you would need to create an index on `race_id` to use this query.)

```cql
SELECT *
FROM cycling.calendar
WHERE race_id = 101
  AND race_start_date >= '2014-05-27'
  AND race_start_date < '2017-06-16';
```

Output:

```
race_id | race_start_date                 | race_end_date                   | race_name
---------+---------------------------------+---------------------------------+--------
      101 | 2015-06-07 00:00:00.000000+0000 | 2015-06-14 00:00:00.000000+0000 | Criterium du Dauphine
      101 | 2014-06-06 00:00:00.000000+0000 | 2014-06-13 00:00:00.000000+0000 | Criterium du Dauphine
```

To allow selection of a contiguous set of rows, the WHERE clause must apply an equality condition to the `race_id` component of the primary key.

Using compound primary keys and sorting results

These restrictions apply when using an ORDER BY clause with a compound primary key:

1. Only include clustering columns in the ORDER BY clause.

2. In the WHERE clause, provide all the partition key values and clustering column values that precede the column(s) in the ORDER BY clause. In 6.0 and later, the columns specified in the ORDER BY clause must be an ordered subset of the columns of the clustering key; however, columns restricted by the equals operator (=) or a single-valued IN restriction can be skipped.

3. When sorting multiple columns, the columns must be listed in the same order in the ORDER BY clause as they are listed in the PRIMARY KEY clause of the table definition.

4. Sort ordering is limited. For example, if your table definition uses CLUSTERING ORDER BY (start_month ASC, start_day ASC), then you can use ORDER BY start_day, race in your query (ASC is the default).
You can also reverse the sort ordering if you apply it to all of the columns; for example, ORDER BY start_day DESC, race DESC.

5. Refer to a column using the actual name, not an alias.

See the cyclist_category table, which uses a compound primary key. This query retrieves the cyclist categories, in descending order by points.

```
SELECT *
FROM cycling.cyclist_category
WHERE category = 'Time-trial'
ORDER BY points DESC;
```

Output:

<table>
<thead>
<tr>
<th>category</th>
<th>points</th>
<th>id</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time-trial</td>
<td>182</td>
<td>220844bf-4860-49d6-6b5d3a79cbfb</td>
<td>TIRALONGO</td>
</tr>
<tr>
<td>Time-trial</td>
<td>3</td>
<td>6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47</td>
<td>KRUIJSWIJK</td>
</tr>
</tbody>
</table>

(2 rows)

The following example shows a table with a more complex compound primary key.

```
CREATE TABLE IF NOT EXISTS 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  
)
);
```

This query contains a WHERE clause that provides values for all primary key columns that precede the race column and orders the results by race:

```
SELECT *
FROM cycling.events
WHERE year = 2017
    AND discipline = 'Cyclo-cross'
    AND start_month = 1
    AND start_day = 1
ORDER BY race;
```

Output:

<table>
<thead>
<tr>
<th>year</th>
<th>discipline</th>
<th>start_month</th>
<th>start_day</th>
<th>race</th>
<th>end_day</th>
<th>end_month</th>
<th>location</th>
<th>uci_code</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017</td>
<td>Cyclo-cross</td>
<td>1</td>
<td>1</td>
<td>DVV verzekeringen trofee - GP Sven Nys</td>
<td>null</td>
<td>Baal</td>
<td>C1</td>
<td></td>
</tr>
</tbody>
</table>
This query has multiple clustering columns in the **ORDER BY** clause:

```cql
SELECT *
FROM cycling.events
WHERE year = 2017
  AND discipline = 'Cyclo-cross'
  AND start_month = 1
ORDER BY start_day, race;
```

**Grouping results**

The **GROUP BY** clause condenses the selected rows that share the same values for a set of columns into a group.

A **GROUP BY** clause can contain:

- Partition key columns and clustering columns.
- A deterministic monotonic function, including a user-defined function (UDF), on the last clustering column specified in the **GROUP BY** clause. The `FLOOR()` function is monotonic when the duration and start time parameters are constants.
- A deterministic aggregate.

The examples in this section use the **race_times_summary** table:

```cql
CREATE TABLE IF NOT EXISTS cycling.race_times_summary (
  race_date date,
  race_time time,
  PRIMARY KEY (race_date, race_time)
);
```

The table contains these rows:

<table>
<thead>
<tr>
<th>race_date</th>
<th>race_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2019-03-21</td>
<td>10:01:18.0000000000000</td>
</tr>
<tr>
<td>2019-03-21</td>
<td>10:15:20.0000000000000</td>
</tr>
<tr>
<td>2019-03-21</td>
<td>11:15:38.0000000000000</td>
</tr>
<tr>
<td>2019-03-21</td>
<td>12:15:40.0000000000000</td>
</tr>
<tr>
<td>2018-07-26</td>
<td>10:01:18.0000000000000</td>
</tr>
<tr>
<td>2018-07-26</td>
<td>10:15:20.0000000000000</td>
</tr>
<tr>
<td>2018-07-26</td>
<td>11:15:38.0000000000000</td>
</tr>
<tr>
<td>2018-07-26</td>
<td>12:15:40.0000000000000</td>
</tr>
<tr>
<td>2017-04-14</td>
<td>10:01:18.0000000000000</td>
</tr>
<tr>
<td>2017-04-14</td>
<td>10:15:20.0000000000000</td>
</tr>
<tr>
<td>2017-04-14</td>
<td>11:15:38.0000000000000</td>
</tr>
<tr>
<td>2017-04-14</td>
<td>12:15:40.0000000000000</td>
</tr>
</tbody>
</table>

(12 rows)

This query groups the rows by the **race_date** column values:

```cql
SELECT race_date, race_time
FROM cycling.race_times_summary
```
Cassandra Query Language (CQL) reference

GROUP BY race_date;

Each set of rows with the same race_date column value are grouped together into one row in the query output. Three rows are returned because there are three groups of rows with the same race_date column value. The value returned is the first value that is found for the group.

<table>
<thead>
<tr>
<th>race_date</th>
<th>race_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2019-03-21</td>
<td>10:01:18.000000000</td>
</tr>
<tr>
<td>2018-07-26</td>
<td>10:01:18.000000000</td>
</tr>
<tr>
<td>2017-04-14</td>
<td>10:01:18.000000000</td>
</tr>
</tbody>
</table>

(3 rows)

This query groups the rows by race_date and FLOOR(race_time, 1h), which returns the hour. The number of rows in each group is returned by COUNT(*).

SELECT race_date, FLOOR(race_time, 1h), COUNT(*)
FROM cycling.race_times_summary
GROUP BY race_date, FLOOR(race_time, 1h);

Nine rows are returned because there are nine groups of rows with the same race_date and FLOOR(race_time, 1h) values:

<table>
<thead>
<tr>
<th>race_date</th>
<th>system.floor(race_time, 1h)</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>2019-03-21</td>
<td>10:00:00.000000000</td>
<td>2</td>
</tr>
<tr>
<td>2019-03-21</td>
<td>11:00:00.000000000</td>
<td>1</td>
</tr>
<tr>
<td>2019-03-21</td>
<td>12:00:00.000000000</td>
<td>1</td>
</tr>
<tr>
<td>2018-07-26</td>
<td>10:00:00.000000000</td>
<td>2</td>
</tr>
<tr>
<td>2018-07-26</td>
<td>11:00:00.000000000</td>
<td>1</td>
</tr>
<tr>
<td>2018-07-26</td>
<td>12:00:00.000000000</td>
<td>1</td>
</tr>
<tr>
<td>2017-04-14</td>
<td>10:00:00.000000000</td>
<td>2</td>
</tr>
<tr>
<td>2017-04-14</td>
<td>11:00:00.000000000</td>
<td>1</td>
</tr>
<tr>
<td>2017-04-14</td>
<td>12:00:00.000000000</td>
<td>1</td>
</tr>
</tbody>
</table>

(9 rows)

Computing aggregates

DSE provides the built-in functions COUNT(), MIN(), MAX(), SUM(), and AVG() that return aggregate values to SELECT statements. You can also create user-defined aggregates (UDAs). The following sections show examples.

Using COUNT() to get the non-null value count for a column

A SELECT using COUNT(column_name) returns the number of non-null values in a column. COUNT ignores null values.

This query counts 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 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 query returns the number of rows in the `cyclist_name` table:

```
SELECT COUNT(*)
FROM cycling.cyclist_name;
```

This query counts the number of rows and calculates the maximum value for `start_day` in the `events` table:

```
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:

```
SELECT start_month, MAX(start_day)
FROM cycling.events
WHERE year = 2022
ALLOW FILTERING;
```

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 using MAX(column_name)` returns the maximum value in a column. When the column’s data type is numeric (bigint, decimal, double, float, int, or smallint), `MAX(column_name)` returns the maximum value.

For example:

```
SELECT MAX(race_points)
FROM cycling.cyclist_points
WHERE id = e3b19ec4-774a-4d1c-9e5a-decec1e30aac;
```

Output:

```
system.max(race_points)
-----------------------
120
```

(1 row)

If you do not include a `WHERE` clause, a warning message is displayed:

```
Warnings :
Aggregation query used without partition key
```

The `MIN` function returns the minimum value:

```
SELECT MIN(race_points)
```

```
system.min(race_points)
-----------------------
6
```

(1 row)
If the column referenced by the `MAX` or `MIN` functions has an `ascii` or `text` data type, the 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`, the functions return the most recent or least recent times and dates.

If a column has a null value, `MAX` and `MIN` ignore the null value. If the column for an entire set of rows contains null, `MAX` and `MIN` return null.

If the query includes a `WHERE` clause (recommended), `MAX` returns the largest value from the rows that satisfy the `WHERE` condition, and `MIN` returns the smallest value from the rows that satisfy the `WHERE` condition.

Getting the average or sum of a column of numbers

This example computes the average of all values in a column using `AVG`:

```
SELECT AVG(race_points)
FROM cycling.cyclist_points
WHERE id = e3b19ec4-774a-4d1c-9e5a-decec1e30aac;
```

Output:

```
system.avg(race_points)
-------------------------
67
(1 rows)
```

Use `SUM` to get a total:

```
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)
```

If any of the rows returned have a null value for the column referenced in the `AVG` function, the function includes that row in the row count, but uses zero for the null value to calculate the average. The `SUM` and `AVG` functions do not work with `text`, `uuid`, or `date` fields.

This query returns the cyclist team average time using a user-defined aggregate (UDA).

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

See Creating a user-defined aggregate function (UDA) and CREATE AGGREGATE.

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.

This query retrieves the date/time that writes occurred to the `firstname` column of a cyclist:

```cql
SELECT WRITETIME (firstname)
FROM cycling.cyclist_points
WHERE id = e3b19ec4-774a-4d1c-9e5a-decec1e30aac;
```

Output:

<table>
<thead>
<tr>
<th>writetime(firstname)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1538688876521481</td>
</tr>
<tr>
<td>1538688876523973</td>
</tr>
<tr>
<td>1538688876525239</td>
</tr>
</tbody>
</table>

The **WRITETIME** output of the last write 1538688876525239 in microseconds converts to **Thursday, October 4, 2018 4:34:36.525 PM GMT-05:00 DST**.

Retrieving the time-to-live of a column

The time-to-live (TTL) of a column value in a row is the number of seconds before the value is marked with a tombstone.

This example **INSERT** sets the TTL of the column values to 200 seconds:

```cql
INSERT INTO cycling.calendar (
    race_id, race_name, race_start_date, race_end_date
) VALUES (
)
USING TTL 200;
```

This example **UPDATE** sets the TTL of a single `race_name` column value to 200 seconds:

```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';
```

This query retrieves the current TTL of the specified `race_name` column value:

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

Output:

<table>
<thead>
<tr>
<th>ttl(race_name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>276</td>
</tr>
</tbody>
</table>

(1 rows)

Retrieving values in JSON format

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>
<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><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>' 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.

```
TRUNCATE cycling.country_flag;
```
4. Use SELECT to verify the table data has been truncated.

```
SELECT *
```
Cassandra Query Language (CQL) reference

```
FROM cycling.country_flag;
```

<table>
<thead>
<tr>
<th>country</th>
<th>cyclist_name</th>
<th>flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>---------</td>
<td>--------------</td>
<td>------</td>
</tr>
<tr>
<td>(0 rows)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TRUNCATE sends a JMX command to all nodes to delete SSTables that hold the data from the specified table. If any of the nodes are down or do not respond, the command fails and outputs a message like the following:

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

UNRESTRICT

Removes restriction from a role.

**Synopsis**

```
UNRESTRICT permission_name
ON [keyspace_name.]table_name
FROM role_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>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>
Examples
Remove the select restriction from the database admin on the cycling keyspace:

```
UNRESTRICT SELECT ON KEYSpace cycling FROM db_admin;
```

UNRESTRICT ROWS
Removes the column definition for 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 ROWS command.

Synopsis
```
UNRESTRICT ROWS ON [keyspace_name.]table_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>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>
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 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 94: 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>
<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, 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>--</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>

**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 UPDATE command does not return any result unless it includes IF EXISTS.

<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>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;)</td>
</tr>
</tbody>
</table>

The database storage engine can only encode TTL timestamps through January 192038 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 [Cassandra.expiration_date_overflow_policy](#).
### Cassandra Query Language (CQL) reference

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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></td>
<td>Note the use of square brackets.</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></td>
<td>Note the use of curly brackets ({}).</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></td>
<td>Note the use of curly brackets ({}).</td>
</tr>
<tr>
<td>set_item</td>
<td>The literal value included in a set.</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.

```cql
UPDATE cycling.cyclist_name
SET comment = 'Rides hard, gets along with others, a real winner'
WHERE id = fb372533-eb95-4bb4-8685-6ef61e994caa;

UPDATE cycling.cyclist_name
SET comment = 'Rides fast, does not get along with others, a real dude'
WHERE id = 5b6962dd-3f90-4c93-8f61-eabfa4a803e2;

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
SET comment = 'Rides hard, gets along with others, a real winner'
WHERE id = fb372533-eb95-4bb4-8685-6ef61e994caa
IF comment = NULL;
```

Using IF statements impact performance, see linearizable consistency.

For examples, see Conditionally updating columns.

**Examples**

**Updating a column**

Update a column in several rows at once:

```
UPDATE cycling.cyclist_name
SET firstname = NULL
WHERE id IN (
    5b6962dd-3f90-4c93-8f61-eabfa4a803e2,
    fb372533-eb95-4bb4-8685-6ef61e994caa
);
```

Update multiple columns in a single row:

```
UPDATE cycling.cyclist_name
SET
    firstname = 'Marianne',
    lastname = 'VOS'
WHERE id = 88b8fd18-b1ed-4e96-bf79-4280797cba80;
```

**Updating using a timestamp and TTL**

To update a `set` (or any collection) value with a timestamp and TTL, specify the value:

```
UPDATE cycling.cyclist_sponsors_expire
USING TIMESTAMP 200
AND TTL 20000
SET sponsorship += { 'Tag Heuer' }
WHERE cyclist_name = 'PRIETO, Marcela';
```

**Updating a counter column**

To update a counter column value in a counter table, specify the increment or decrement to apply to the current value.

```
UPDATE cycling.popular_count
SET popularity = popularity + 2
WHERE id = 6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47;
```

To use a lightweight transaction on a counter column to ensure accuracy, put one or more counter updates in the batch statement. For details, see Performing conditional updates in a batch.

Creating a partition using UPDATE
Since 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 `(id)`, you can `UPDATE` the partition with id `e7cd5752-bc0d-4157-a80f-7523add8dbcd`, even though it does not exist yet:

```
UPDATE cycling.cyclist_name
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' ]
WHERE year = 2015
  AND month = 06;
```

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[2] = 'Tour de France'
WHERE year = 2015
  AND month = 06;
```

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 - [ 'Tour de France' ]
WHERE year = 2015
  AND month = 06;
```

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
```
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
    AND month = 06;
```

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 = 2015
    AND month = 06;
```

To set the a TTL for each map element:

```
UPDATE cycling.upcoming_calendar
USING TTL 10000000
SET events[2] = 'Vuelta Ciclista a Venezuela'
WHERE year = 2015
    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:

```cql
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.

You may, however, try to add the new entry with a command that overwrites the first two and adds the new one. 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

To change the value of an individual field value in a user-defined type with non-collection fields, use the `UPDATE` command:

```cql
UPDATE cycling.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 only apply the update if the query matches a row:

```cql
UPDATE cycling.cyclist_id
SET id = UUID()
WHERE lastname = 'WELTEN'
AND firstname = 'Bram'
AND age = 18
IF EXISTS;
```

• If the row exists (returns true), the following is output:

```
[applied]
-------------
True
```

• If no row exists (returns false), the command fails and the following is output:

```
[applied]
-------------
False
```

Use `IF condition` to apply tests to one or more other (non-primary key) column values in the matching row.

For example, to set a new UUID only if the id matches.

```cql
UPDATE cycling.cyclist_id
SET id = UUID()
WHERE lastname = 'WELTEN'
AND firstname = 'Bram'
```
Cassandra Query Language (CQL) reference

AND age = 18
IF id = 18f471bf-f631-4bc4-a9a2-d6f6cf5ea503;

• If a record matches and the condition returns TRUE, the update is applied and following is output:

<table>
<thead>
<tr>
<th>[applied]</th>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>863e7103-c03b-48c3-a11c-42376aa77291</td>
</tr>
</tbody>
</table>

• If a record matches and the condition returns false, the query fails and following shows an example of the output:

<table>
<thead>
<tr>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>863e7103-c03b-48c3-a11c-42376aa77291</td>
</tr>
</tbody>
</table>

• If no record matches and the condition is testing for a non-null value such as id = 18f471bf-f631-4bc4-a9a2-d6f6cf5ea503 the query also fails.

When the IF condition tests for a null value, for example:

```
UPDATE cycling.cyclist_id
SET id = UUID()
WHERE lastname = 'Smith'
AND firstname = 'Joe'
AND age = 22
IF id = NULL;
```

• A record matches and the id column has no value, a value is inserted.
• A record matches and the id column has a value (is not null), the statement fails.
• No record matches, then a new record is created.

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:

```
CREATE TABLE cycling.mytable (a int, b int, s int static, d text, PRIMARY KEY (a, b));
BEGIN BATCH
  INSERT INTO cycling.mytable (a, b, d) values (7, 7, 'a');
  UPDATE cycling.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.)

**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 double quotes:

```cql
USE "DataStax";
```

**Synopsis**

```cql
USE keyspace_name ;
```

### Table 95: 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;&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**

```cql
USE cycling;
```
Chapter 10. Examples

CQL description of keyspaces, tables and other database resources used in CQL examples.

Setting up the Cycling keyspace

CQL commands to set up the cycling keyspace and 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 Enterprise deployment is a single node cluster in a development environment.

Update the following command and create a CQL file named 0_create_keyspace.cql in the same directory where you launch cqlsh.

```cql
// START-cyclingks
CREATE KEYSPACE IF NOT EXISTS cycling
WITH replication = {
  'class' : 'SimpleStrategy',
  'replication_factor' : 1
};
// END-cyclingks

// START-usecycling
USE cycling;
// END-usecycling
```

Additional cycling keyspace examples

Shows additional keyspace examples

```cql
// START-single
CREATE KEYSPACE cycling
WITH REPLICATION = {
  'class' : 'SimpleStrategy',
  'replication_factor' : 1
};
// END-single

// START-drop
DROP KEYSPACE cycling;
// END-drop

// START-network
CREATE KEYSPACE cycling
WITH REPLICATION = {
  'class' : 'NetworkTopologyStrategy',
  'datacenter1' : 1
};
// END-network

DROP KEYSPACE cycling;

// START-multi
CREATE KEYSPACE "Cycling"
WITH REPLICATION = {
  'class' : 'NetworkTopologyStrategy',
  'boston' : 3 , // Datacenter 1
  'seattle' : 2 , // Datacenter 2
```
\[
\text{'tokyo' : 2 // Datacenter 3 }
\]

// END-multi

DROP KEYS\$E cycling;

// START-durable
CREATE KEYS\$E cycling
WITH REPLICATION = {
  'class' : 'NetworkTopologyStrategy',
  'datacenter1' : 3
}
AND DURABLE_W\$\$rites = false;
// END-durable

Cycling types
User defined type examples used in the DataStax Enterprise documentation set.

basic_info
User-defined type that contains fields for basic information about a cyclist such as birthday, nationality, height, and weight.

SOURCE '0_create_keyspace.cql';

// START-droptype
DROP TYPE IF EXISTS cycling.basic_info;
// END-droptype

// START-basicinfotype
CREATE TYPE IF NOT EXISTS cycling.basic_info (birthday timestamp, nationality text, height text, weight text);
// END-basicinfotype

DESC TYPE cycling.basic_info;

basic_info_withTTL
User-defined type that contains fields for basic information about a cyclist such as birthday, nationality, height, and weight. It also includes examples of inserting and updating UDTs with time-expired data.

SOURCE '0_create_keyspace.cql';

// START-droptype
DROP TYPE IF EXISTS cycling.basic_info_expire;
// END-droptype

// START-basicinfoTTLexpiretype
CREATE TYPE IF NOT EXISTS cycling.basic_info_expire (birthday timestamp, nationality text, height text, weight text, next_race text);
// END-basicinfoTTLexpiretype

// START-createtable
CREATE TABLE IF NOT EXISTS cycling.basic_info_TTL_expire (
Examples

```cql
id UUID PRIMARY KEY,
   lastname text,
   basics basic_info_expire
); // END-createtable

// START-insertTTLexpiretype
INSERT INTO cycling.basic_info_TTL_expire (id, lastname, basics)
VALUES (e7ae5cf3-d358-4d99-b900-85902fda9bb0,
   'FRAME',
   {
      birthday : '1993-06-18',
      nationality : 'New Zealand',
      weight : '175',
      height : '72',
      next_race : 'Amgen Tour of California'
   }
)
USING TIMESTAMP 100 AND TTL 10000;
// END-insertTTLexpiretype

// START-updateTTLexpiretype
UPDATE cycling.basic_info_TTL_expire
USING TTL 100
SET basics.next_race = 'Tour de France'
WHERE id = e7ae5cf3-d358-4d99-b900-85902fda9bb0;
// END-updateTTLexpiretype

// START-selectTTLexpiretype
SELECT *
FROM cycling.basic_info_TTL_expire;
// END-selectTTLexpiretype

// START-selectWritetimeTTLexpiretype
SELECT WRITETIME(basics), TTL(basics)
FROM cycling.basic_info_TTL_expire
WHERE id = e7ae5cf3-d358-4d99-b900-85902fda9bb0;
// END-selectWritetimeTTLexpiretype

fullname
User-defined type that contains two fields for first name and last name.

SOURCE '0_create_keyspace.cql';

// START-drop
DROP TYPE IF EXISTS cycling.fullname;
// END-drop

// Store first and last in the same column using a custom type

// START-fullname
CREATE TYPE IF NOT EXISTS cycling.fullname (firstname text, lastname text);
// END-fullname

// START-alt
ALTER TYPE cycling.fullname
ADD middlename text;
// END-alt
```
Examples

// START-rename
ALTER TYPE cycling.fullname
RENAME middlename TO middle
AND lastname TO last
AND firstname TO first;
// END-rename

ALTER TYPE cycling.fullname
RENAME middle to middlename
AND last to lastname
AND first TO firstname;

Displaying command results

DESC TYPE cycling.fullname;

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

race

<what is it used for?>

SOURCE '0_create_keyspace.cql';

// START-drop
DROP TYPE IF EXISTS cycling.race;
// END-drop

// START-racetype
CREATE TYPE IF NOT EXISTS cycling.race (  
  race_title text,  
  race_date timestamp,  
  race_time text  
);
// END-racetype

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:

SOURCE '0_create_keyspace.cql';

// START-drop
DROP TABLE IF EXISTS cycling.birthday_list;
// END-drop

/* Map field and search index map fields example */

// START-table
CREATE TABLE IF NOT EXISTS cycling.birthday_list (  
  cyclist_name text PRIMARY KEY,  
  /* Additional columns here */  
);
Examples

```cql
blist_map<text, text>
);
// END-table

// START-insert_allan
INSERT INTO cycling.birthday_list (cyclist_name, blist_ ) VALUES ('Allan DAVIS', { 'blist_age': '35', 'bday': '27/07/1980', 'blist_nation': 'AUSTRALIA' });
// END-insert_allan

// START-insertall
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 search index:

```cql
SOURCE 'birthday_list-table.cql';

// For different key-value pairs that you want specify a data types
// START-index-control-types
CREATE SEARCH INDEX IF NOT EXISTS 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:*';

// Regular queries
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:

// SOURCE 'birthday_list-table.cql';

// START-index
CREATE SEARCH INDEX IF NOT EXISTS
ON cycling.birthday_list
WITH COLUMNS blist_ { excluded:true }; // END-index

// START-desc_active
DESC ACTIVE SEARCH INDEX SCHEMA ON cycling.birthday_list;
// END-desc_active

// START-type
// Add type
ALTER SEARCH INDEX SCHEMA
ON cycling.birthday_list
ADD types.fieldType[ @class='org.apache.solr.schema.TrieIntField',
@name='TrieIntField' ];

// Control the data types of map fields by name
ALTER SEARCH INDEX SCHEMA
ON cycling.birthday_list
ADD fields.field[ @indexed='true', @multiValued='false', @name='blist_age',
@type='TrieIntField' ]; // END-type

// START-alter
ALTER SEARCH INDEX SCHEMA
ON cycling.birthday_list
ADD fields.field[ @name='blist_nation', @indexed='true', @multiValued='false',
@type='StrField' ]; // END-alter

// START-desc_pending
DESC PENDING SEARCH INDEX SCHEMA ON cycling.birthday_list;
// END-desc_pending

// Make the pending schema active and rebuild the index
// START-reload
RELOAD SEARCH INDEX
ON cycling.birthday_list;
// END-reload
Create indexes on map collections to query.

```cql
CREATE TABLE IF NOT EXISTS cycling.birthday_list (
  cyclist_name text PRIMARY KEY,
  blist map<text, text>
);
```

```cql
CREATE INDEX IF NOT EXISTS blist_idx
ON cycling.birthday_list ( ENTRIES(blist) );
```

```cql
CREATE INDEX IF NOT EXISTS blist_values_idx
ON cycling.birthday_list ( VALUES(blist) );
```

```cql
INSERT INTO cycling.birthday_list (
  cyclist_name, blist
) VALUES (  
  'Allan DAVIS', { 'age':'35', 'bday':'27/07/1980', 'nation':'AUSTRALIA' }
);
```

```cql
INSERT INTO cycling.birthday_list (
  cyclist_name, blist
) VALUES (  
  'Claudio VANDELLI', { 'age':'54', 'bday':'27/07/1961', 'nation':'ITALY' }
);
```

```cql
INSERT INTO cycling.birthday_list (
  cyclist_name, blist
) VALUES (  
  'Laurence BOURQUE', { 'age':'23', 'bday':'27/07/1992', 'nation':'CANADA' }
);
```

```cql
INSERT INTO cycling.birthday_list (
  cyclist_name, blist
) VALUES (  
  'Claudio HEINEN', { 'age':'23', 'bday':'27/07/1992', 'nation':'GERMANY' }
);
```

```cql
INSERT INTO cycling.birthday_list (
  cyclist_name, blist
) VALUES (  
  'Allan DAVIS', { 'age':'35', 'bday':'27/07/1980', 'nation':'AUSTRALIA' }
)

```

```
CREATE TABLE IF NOT EXISTS cycling.birthday_list (cyclist_name text PRIMARY KEY, blist map<text, text>);
```

```cql
CREATE INDEX IF NOT EXISTS blist_idx
ON cycling.birthday_list (ENTRIES(blist));
```

```cql
CREATE INDEX IF NOT EXISTS blist_values_idx
ON cycling.birthday_list (VALUES(blist));
```

```
 // START-insertentries
 INSERT INTO cycling.birthday_list (cyclist_name, blist)
 VALUES ('Allan DAVIS', { 'age':'35', 'bday':'27/07/1980', 'nation':'AUSTRALIA' });

 INSERT INTO cycling.birthday_list (cyclist_name, blist)
 VALUES ('Claudio VANDELLI', { 'age':'54', 'bday':'27/07/1961', 'nation':'ITALY' });

 INSERT INTO cycling.birthday_list (cyclist_name, blist)
 VALUES ('Laurence BOURQUE', { 'age':'23', 'bday':'27/07/1992', 'nation':'CANADA' });

 INSERT INTO cycling.birthday_list (cyclist_name, blist)
 VALUES ('Claudio HEINEN', { 'age':'23', 'bday':'27/07/1992', 'nation':'GERMANY' });

```

```
SOURCE '0_create_keyspace.cql';
```

```
DROP TABLE IF EXISTS cycling.birthday_list;
```

```
DROP INDEX IF EXISTS cycling.blist_idx;
DROP INDEX IF EXISTS cycling.blist_values_idx;
```

```
/* Map entries and regular index map fields example */

CREATE TABLE IF NOT EXISTS cycling.birthday_list (cyclist_name text PRIMARY KEY, blist map<text, text>);
```

```
CREATE INDEX IF NOT EXISTS blist_idx
ON cycling.birthday_list (ENTRIES(blist));
```

```
CREATE INDEX IF NOT EXISTS blist_values_idx
ON cycling.birthday_list (VALUES(blist));
```

```
// START-insertentries
INSERT INTO cycling.birthday_list (cyclist_name, blist)
VALUES ('Allan DAVIS', { 'age':'35', 'bday':'27/07/1980', 'nation':'AUSTRALIA' });

INSERT INTO cycling.birthday_list (cyclist_name, blist)
VALUES ('Claudio VANDELLI', { 'age':'54', 'bday':'27/07/1961', 'nation':'ITALY' });

INSERT INTO cycling.birthday_list (cyclist_name, blist)
VALUES ('Laurence BOURQUE', { 'age':'23', 'bday':'27/07/1992', 'nation':'CANADA' });

INSERT INTO cycling.birthday_list (cyclist_name, blist)
VALUES ('Claudio HEINEN', { 'age':'23', 'bday':'27/07/1992', 'nation':'GERMANY' });
```

```cql
SOURCE '0_create_keyspace.cql';
```
This query returns the rows where **age** is 23:

```cql
SELECT *
FROM cycling.birthday_list
WHERE blist[ 'age' ] = '23';
```

<table>
<thead>
<tr>
<th>cyclist_name</th>
<th>blist</th>
</tr>
</thead>
<tbody>
<tr>
<td>Claudio HEINEN</td>
<td>{'age': '23', 'bday': '27/07/1992', 'nation': 'GERMANY'}</td>
</tr>
<tr>
<td>Laurence BOURQUE</td>
<td>{'age': '23', 'bday': '27/07/1992', 'nation': 'CANADA'}</td>
</tr>
</tbody>
</table>

(2 rows)

This query returns the rows where **blist[ 'nation' ]** is NETHERLANDS:

```cql
SELECT *
FROM cycling.birthday_list
WHERE blist CONTAINS 'NETHERLANDS';
```
Examples

WHERE blist[ 'nation' ] = 'NETHERLANDS';

cyclist_name | blist
---------------+--------------------------------------------------------------
  Luc HAGENAARS | {'age': '28', 'bday': '27/07/1987', 'nation': 'NETHERLANDS'}
  Toine POELS  | {'age': '52', 'bday': '27/07/1963', 'nation': 'NETHERLANDS'}
(2 rows)

This query returns the rows where blist contains NETHERLANDS:

SELECT *
FROM cycling.birthday_list
WHERE blist CONTAINS 'NETHERLANDS';

cyclist_name | blist
---------------+--------------------------------------------------------------
  Luc HAGENAARS | {'age': '28', 'bday': '27/07/1987', 'nation': 'NETHERLANDS'}
  Toine POELS  | {'age': '52', 'bday': '27/07/1963', 'nation': 'NETHERLANDS'}
(2 rows)

calendar

Using clustering columns to display data.

SOURCE '0_create_keyspace.cql';

DROP TABLE IF EXISTS cycling.calendar;

// NOT A QUERY, JUST A TABLE FOR QUERIES
// CREATE TABLE WITH LIST FOR UPDATE
// The SELECT statements that use this table can be found below

// START-listColumn
CREATE TABLE IF NOT EXISTS 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
 )
) WITH CLUSTERING ORDER BY (  
 race_start_date DESC, race_end_date DESC
 );
// END-listColumn

INSERT INTO cycling.calendar (  
race_id, race_name, race_start_date, race_end_date
 ) VALUES (  
 100, 'Giro d''Italia', '2015-05-09', '2015-05-31'
 );

INSERT INTO cycling.calendar (  
race_id, race_name, race_start_date, race_end_date
 ) VALUES (  
 );

INSERT INTO cycling.calendar (  
race_id, race_name, race_start_date, race_end_date
 ) VALUES (  
 102, 'Paris-Nice', '2015-03-02', '2015-03-08'
 );

// END-Calendar
Examples

```cql
race_id, race_name, race_start_date, race_end_date
VALUES (102, 'Tour de Suisse', '2015-06-13', '2015-06-21');

INSERT INTO cycling.calendar (race_id, race_name, race_start_date, race_end_date)
VALUES (103, 'Tour de France', '2015-07-04', '2015-07-26');

INSERT INTO cycling.calendar (race_id, race_name, race_start_date, race_end_date)
VALUES (100, 'Giro d''Italia', '2014-05-08', '2014-05-30');

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

INSERT INTO cycling.calendar (race_id, race_name, race_start_date, race_end_date)
VALUES (102, 'Tour de Suisse', '2014-06-12', '2014-06-20');

INSERT INTO cycling.calendar (race_id, race_name, race_start_date, race_end_date)
VALUES (103, 'Tour de France', '2014-07-03', '2014-07-25');

INSERT INTO cycling.calendar (race_id, race_name, race_start_date, race_end_date)
VALUES (100, 'Giro d''Italia', '2013-05-07', '2014-05-29');

INSERT INTO cycling.calendar (race_id, race_name, race_start_date, race_end_date)
VALUES (101, 'Criterium du Dauphine', '2013-06-05', '2013-06-12');

INSERT INTO cycling.calendar (race_id, race_name, race_start_date, race_end_date)
VALUES (102, 'Tour de Suisse', '2013-06-11', '2013-06-19');

INSERT INTO cycling.calendar (race_id, race_name, race_start_date, race_end_date)
VALUES (103, 'Tour de France', '2013-07-02', '2013-07-24');

CAPTURE 'select_all_from_calendar_allow_filtering.results';
// START-select_start_date
SELECT *
FROM cycling.calendar
WHERE race_start_date = '2015-06-13'
ALLOW FILTERING;
// END-select_start_date
```
CAPTURE OFF;

// START-select_with_in
SELECT *
FROM cycling.calendar
WHERE race_id IN (101, 102, 103);
// END-select_with_in

CAPTURE 'select_all_from_calendar_with_date_range.results';

// START-select_with_range
SELECT *
FROM cycling.calendar
WHERE race_id = 101
  AND race_start_date >= '2014-05-27'
  AND race_start_date < '2017-06-16';
// END-select_with_range

CAPTURE OFF;

// START-insert_with_TTL
INSERT INTO cycling.calendar (race_id, race_name, race_start_date, race_end_date)
USING TTL 200;
// END-insert_with_TTL

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

// START-select_with_TTL
SELECT TTL(race_name)
FROM cycling.calendar
WHERE race_id = 200;
// END-select_with_TTL

DELETE
FROM cycling.calendar
WHERE race_id = 200;

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

DELETE
FROM cycling.calendar
WHERE race_id = 201;

// START-insert_with_dollar_characters
INSERT INTO cycling.calendar (race_id, race_name, race_start_date, race_end_date)
VALUES (201, $$Women's Tour of New Zealand$$, '2015-02-18', '2015-02-22')
);
// END-insert_with_dollar_characters
DELETE
FROM cycling.calendar
WHERE race_id = 201;

comments
Example for count, timestamps, limiting results by partition.

SOURCE '0_create_keyspace.cql';

// START-drop
DROP TABLE IF EXISTS cycling.comments;
// END-drop

/* Support fast retrieval of first record on each partition
   and show upserts of timestamp type fields */

// START-comments-table
CREATE TABLE IF NOT EXISTS 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);
// END-comments-table

DROP INDEX IF EXISTS cycling.fn_sparse;

// START-fn_sparse
CREATE CUSTOM INDEX IF NOT EXISTS fn_sparse
ON cycling.comments (created_at)
USING 'org.apache.cassandra.index.sasi.SASIIndex'
WITH OPTIONS = { 'mode': 'SPARSE' };
// END-fn_sparse

// START-insertall
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, '2017-02-17 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');
Examples

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 11:19:44.060-0800', '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-d35c-4a8d-83f7-a3c543789ee7, 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-3f90-4c93-8f61-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-fe58-4846-83e8-f0e1c13d518f, totimestamp(now()), 'Closing ceremony was a little lame', 'Rossella');

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(), 6ab099bc-e68e-48d9-a5f8-9e6fb4c9b47, totimestamp(now()), 'So many great races thanks y'all', 'Steven');

INSERT INTO cycling.comments (record_id, id, created_at, comment, commenter) VALUES
  (now(), 6ab099bc-e68e-48d9-a5f8-9e6fb4c9b47, '2017-02-02 09:49:00.02+0800', 'Best of luck everybody I can't make it', 'Steven');

INSERT INTO cycling.comments (record_id, id, created_at, comment, commenter) VALUES
  (now(), 6ab099bc-e68e-48d9-a5f8-9e6fb4c9b47, '2017-04-05 12:01:00.003', 'Bike damaged in transit bummer', 'Steven');

INSERT INTO cycling.comments (record_id, id, created_at, comment, commenter) VALUES
  (now(), e7cd5752-bc0d-4157-a80f-7523add8bcd, totimestamp(now()), 'Go team, you rocked it', 'Anna');

INSERT INTO cycling.comments (record_id, id, created_at, comment, commenter) VALUES
  (now(), 6d5f1663-89c0-45f6-8cf0-60a373b01622, totimestamp(now()), 'Next year the tour of california!', 'Melissa');

INSERT INTO cycling.comments (record_id, id, created_at, comment, commenter) VALUES
  (now(), 9add4c4-459e-4ed7-b4b5-472f19a67995, totimestamp(now()), 'Next year for sure!', 'Vera');
Examples

INSERT INTO cycling.comments (record_id, id, created_at, comment, commenter) VALUES (now(), 95addc4c-459e-4ed7-b4b5-472f19a67995, '2017-02-13 11:40:16.123-0600', 'I can do without the rain@@@', 'Vera'); // END-insertall

CAPTURE 'select_from_comments.results'; // START-select
SELECT *
FROM cycling.comments
WHERE created_at < '2017-02-13 11:40:16.123'; // END-select
CAPTURE OFF;

// START-alt
ALTER TABLE cycling.comments
WITH compression = {
'sstable_compression' : 'DeflateCompressor',
'chunk_length_kb' : 64
}; // END-alt

// START-cache
ALTER TABLE cycling.comments
WITH caching = {
'keys' : 'NONE',
'rows_per_partition' : 10
}; // END-cache

// START-nocompact
ALTER TABLE cycling.comments
WITH COMPACTION = {
'class' : 'SizeTieredCompactionStrategy',
'enabled' : 'false'
}; // END-nocompact

// START-compaction
ALTER TABLE cycling.comments
WITH COMPACTION = {
'class' : 'SizeTieredCompactionStrategy',
'enabled' : 'true'
}; // END-compaction

Queries against the comment tables
Filter data on the time UUID field.

// SOURCE 'comments-table.cql'

// START-removedata
TRUNCATE cycling.comments;
// END-removedata

// START-insert_timeuuid
INSERT INTO cycling.comments (record_id, id, created_at)
VALUES (now(), e7ae5cf3-d358-4d99-b900-85902fda9bb0, '2017-02-14 12:43:20-0800') ;
Examples

// END-insert_timeuuid

// START-s_insert_timeuuid
SELECT record_id
FROM cycling.comments
WHERE id = e7ae5cf3-d358-4d99-b900-85902fda9bb0
AND created_at = '2017-02-14 12:43:20-0800';
// END-s_insert_timeuuid

/* START-rinsert_timeuuid
record_id
--------------------------------------
e81ae50-4670-11e7-9fd7-81d397057b18
(1 rows)
END-rinsert_timeuuid */

Return the most recent records:

// SOURCE 'comments-table.cql';

CAPTURE 'select_mostrecent_from_comments.results';
// START-select-mostrecent
SELECT toDate(created_at) AS Date, comment, commenter
FROM cycling.comments
PER PARTITION LIMIT 1;
// END-select-mostrecent
CAPTURE OFF;

date       | comment                                           | commenter
------------+---------------------------------------------------+-----------
2017-04-01 | LATE RIDERS SHOULD NOT DELAY THE START            | Alex      
2019-11-05 | The gift certificate for winning was the best     | Maryanne  
2019-11-05 | Fastest womens time ever way to go amy!           | Katarzyna 
2019-11-05 | Great course                                     | Michael   
2019-11-05 | Can't wait for the next race                      | Maryanne  
2019-11-05 | Thanks for all your hard work                     | Marianne  
2019-11-05 | A for effort!                                     | Paolo     
2019-11-05 | Closing ceremony was a little lame                | Rossella  
2019-11-05 | Next time guys!                                  | Marcia    
2019-11-05 | So many great races thanks y'all                  | Steven    
2019-11-05 | Go team, you rocked it                           | Anna      
2019-11-05 | Next year the tour of california!                | Melissa   
2019-11-05 | Next year for sure!                              | Vera      
(13 rows)

Date and time queries:

// SOURCE 'comments-table.cql'

// START-removedata
TRUNCATE cycling.comments;
// END-removedata

/* CURRENT DATE */

// START-now_date
INSERT INTO cycling.comments (id, created_at) VALUES (e81ae50-4670-11e7-9fd7-81d397057b18, '2017-02-14 12:43:20-0800');
// END-now_date

PAGE 456
Examples

```cql
e7ae5cf3-d358-4d99-b900-85902fda9bb0,
toTimeStamp(toDate(now()))
);
// END-now_date

// START-snow_date
SELECT created_at
FROM cycling.comments
WHERE id = e7ae5cf3-d358-4d99-b900-85902fda9bb0
LIMIT 1;
// END-snow_date

/* START-rnow_date
created_at
---------------------------------
2017-06-01 00:00:00.000000+0000
(1 rows)
END-rnow_date */

/* CURRENT TIMESTAMP */

// START-now_timestamp
INSERT INTO cycling.comments (id, created_at) VALUES (e7ae5cf3-d358-4d99-b900-85902fda9bb0,
toTimeStamp(now()))
);
// END-now_timestamp

// START-snow_timestamp
SELECT created_at FROM cycling.comments
WHERE id = e7ae5cf3-d358-4d99-b900-85902fda9bb0
LIMIT 1;
// END-snow_timestamp

/* START-rnow_timestamp
created_at
---------------------------------
2017-06-01 03:28:41.526000+0000
(1 rows)
END-rnow_timestamp */

/* DATE without time or zone */

// START-string_date_no_tz
INSERT INTO cycling.comments (id, created_at) VALUES (e7ae5cf3-d358-4d99-b900-85902fda9bb0,
'2017-04-01')
);
// END-string_date_no_tz

// START-sstring_date_no_tz
SELECT created_at FROM cycling.comments
WHERE id = e7ae5cf3-d358-4d99-b900-85902fda9bb0
AND created_at = '2017-04-01';
// END-sstring_date_no_tz

/* START-rstring_date_no_tz
created_at
---------------------------------
2017-06-01 03:28:41.526000+0000
(1 rows)
END-rstring_date_no_tz */

/ * DATE without time or zone */

// START-string_date_no_tz
INSERT INTO cycling.comments (id, created_at) VALUES (e7ae5cf3-d358-4d99-b900-85902fda9bb0,
'2017-04-01')
);
// END-string_date_no_tz

// START-sstring_date_no_tz
SELECT created_at FROM cycling.comments
WHERE id = e7ae5cf3-d358-4d99-b900-85902fda9bb0
AND created_at = '2017-04-01';
// END-sstring_date_no_tz

/* START-rstring_date_no_tz
created_at
---------------------------------
2017-06-01 03:28:41.526000+0000
(1 rows)
END-rstring_date_no_tz */
```
Examples

---

2017-04-01 00:00:00.000000+0000

(1 rows)
END-rstring_date_no_tz */

/* DATE WITH TIMEZONE */

// START-string_date_tz
INSERT INTO cycling.comments (id, created_at) VALUES (e7ae5cf3-d358-4d99-b900-85902fda9bb0, '2017-04-01+0000');
// END-string_date_tz

// START-sstring_date_tz
SELECT created_at FROM cycling.comments WHERE id = e7ae5cf3-d358-4d99-b900-85902fda9bb0 AND created_at = '2017-04-01+0000';
// END-sstring_date_tz

/* START-rstring_date_tz
created_at
---------------------------------
2017-04-01 00:00:00.000000+0000
(1 rows)
END-rstring_date_tz */

/* FULL TIMESTAMP */

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

// START-sstring_ts
SELECT created_at FROM cycling.comments WHERE id = e7ae5cf3-d358-4d99-b900-85902fda9bb0 AND created_at = '2017-04-01T11:21:59.001-0800';
// END-sstring_ts

/* START-rstring_ts
created_at
---------------------------------
2017-04-01 11:21:59.001000-0800
END-rstring_ts */

country_flag

Static data type examples.

// CREATE TABLE WITH STATIC COLUMN, example uses an integer to identify flag, but it could be a blob
SOURCE '0_create_keyspace.cql';

DROP TABLE IF EXISTS cycling.country_flag;
CREATE TABLE IF NOT EXISTS cycling.country_flag (
   country text,
   cyclist_name text,
   flag int STATIC,
   PRIMARY KEY (country, cyclist_name)
);

INSERT INTO cycling.country_flag (
   country, cyclist_name, flag
) VALUES ('Belgium', 'Jacques', 1);

INSERT INTO cycling.country_flag (
   country, cyclist_name
) VALUES ('Belgium', 'Andre');

INSERT INTO cycling.country_flag (
   country, cyclist_name, flag
) VALUES ('France', 'Andre', 2);

INSERT INTO cycling.country_flag (
   country, cyclist_name, flag
) VALUES ('France', 'George', 3);

CAPTURE 'select_initial_from_country_flag.results';

SELECT *
FROM cycling.country_flag;

CAPTURE OFF;

TRUNCATE cycling.country_flag;

CAPTURE 'select_initial_from_country_flag.results';

SELECT *
FROM cycling.country_flag;

CAPTURE OFF;

INSERT INTO cycling.country_flag (
   country, cyclist_name, flag
) VALUES ('Belgium', 'Jacques', 1);

INSERT INTO cycling.country_flag (
   country, cyclist_name
) VALUES ('Belgium', 'Andre');

CAPTURE OFF;

INSERT INTO cycling.country_flag (
   country, cyclist_name, flag
) VALUES ('Belgium', 'Jacques', 1);

INSERT INTO cycling.country_flag (
   country, cyclist_name
) VALUES ('Belgium', 'Andre');
Examples

```cql
INSERT INTO cycling.country_flag (country, cyclist_name, flag)
VALUES ('France', 'Andre', 2);

INSERT INTO cycling.country_flag (country, cyclist_name, flag)
VALUES ('France', 'George', 3);
```

cyclist_alt_stats

Removes the table and inserts data required for examples.

```cql
// QUERY USING MULTIPLE INDEXES
// DISCUSSION OF THE NEED FOR ALLOW FILTERING
// Showing date upserts and queries
// avg
SOURCE '0_create_keyspace.cql';

// START-drop_cyclist_alt_stats
DROP TABLE IF EXISTS cycling.cyclist_alt_stats;
// END-drop_cyclist_alt_stats

// START-cyclist_alt_stats
CREATE TABLE IF NOT EXISTS 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);
// END-cyclist_alt_stats

// START-index_bday
CREATE INDEX IF NOT EXISTS birthday_idx ON cycling.cyclist_alt_stats (birthday);
// END-index_bday

// START-index_nationality
CREATE INDEX IF NOT EXISTS nationality_idx ON cycling.cyclist_alt_stats (nationality);
// END-index_nationality

USE cycling;

// START-insertnow
INSERT INTO cycling.cyclist_alt_stats (id, last_race)
VALUES (ed584e99-80f7-4b13-9a90-9dc5571e6821, todate(now()));
// END-insertnow
Examples

// START-insertdate
INSERT INTO cycling.cyclist_alt_stats (id, first_race) VALUES (ed584e99-80f7-4b13-9a90-9dc5571e6821, '2006-03-15');
// END-insertdate

CAPTURE 'select_dates_from_cyclist_alt_stats.results';
// START-selectrace
SELECT first_race, last_race, birthday
FROM cycling.cyclist_alt_stats
WHERE id = ed584e99-80f7-4b13-9a90-9dc5571e6821;
// END-selectrace

CAPTURE OFF;

// START-updatestring
UPDATE cycling.cyclist_alt_stats
SET birthday = '1987-03-07'
WHERE id = ed584e99-80f7-4b13-9a90-9dc5571e6821;
// END-updatestring

// START-updatenow
UPDATE cycling.cyclist_alt_stats
SET last_race = toDate(now())
WHERE id = ed584e99-80f7-4b13-9a90-9dc5571e6821;
// END-updatenow

// START-insert
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', '1986-04-22', '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 (823ec386-2a46-45c9-be41-2425a47658e, 'BELKOV', '1985-01-09', 'Russia', 71, 'kg', 1.84, '2002-03-22', '2017-04-16');
INSERT INTO cycling.cyclist_alt_stats (id, lastname, birthday, nationality, weight, w_units, height, first_race, last_race)
VALUES (e0953617-07eb-4c82-89f1-3b2757981623, 'BRUTT', '1982-01-29', 'Russia', 68, 'kg', 1.78, '1998-02-15', '2017-04-16');
INSERT INTO cycling.cyclist_alt_stats (id, lastname, birthday, nationality, weight, w_units, height, first_race, last_race)
VALUES (c09e9451-50a-483d-8108-e6bea2e827b3, 'VEIKKANEN', '1981-01-14', 'Finland', 63, 'kg', 1.82, '1996-05-21', '2017-04-16');
INSERT INTO cycling.cyclist_alt_stats (id, lastname, birthday, nationality, weight, w_units, height, first_race, last_race)
INSERT INTO cycling.cyclist_alt_stats (id, lastname, birthday, nationality, weight, w_units, height, first_race, last_race)
VALUES (1ba0417d-62da-4103-CQL for DataStax Enterprise 6.7 Latest DSE version Latest 6.7 patch: 6.7.7

461
Examples

b710-de6fb27db6f,'PAULINHO', '1990-05-27', 'Portugal', null, null, null, '2006-03-15','2017-03-05');
INSERT INTO cycling.cyclist_alt_stats (id, lastname, birthday, nationality, weight, w_units, height, first_race, last_race) VALUES
// END-insert

CAPTURE 'select_all_from_cyclist_alt_stats_filtering.results';
// START-select_success
SELECT *
FROM cycling.cyclist_alt_stats
WHERE birthday = '1982-01-29'
AND nationality = 'Russia'
ALLOW FILTERING;
// END-select_success
CAPTURE OFF;

// START-select_portugal
SELECT *
FROM cycling.cyclist_alt_stats
WHERE birthday = '1990-05-27'
AND nationality = 'Portugal'
ALLOW FILTERING;
// END-select_portugal

ALTER TABLE cycling.cyclist_alt_stats
ADD cyclist_age int;
// END-add

CAPTURE 'select_id_age_from_cyclist_alt_stats.results';
// START-sadd
SELECT id, cyclist_age AS age
FROM cycling.cyclist_alt_stats
LIMIT 3;
// END-sadd
CAPTURE OFF;

// START-delete
ALTER TABLE cycling.cyclist_alt_stats
DROP cyclist_age;
// END-delete

// START-rename
ALTER TABLE cycling.cyclist_alt_stats
RENAME id TO cyclist_id;
// END-rename

cyclist_base

SOURCE '0_create_keyspace.cql';
DROP TABLE IF EXISTS cycling.cyclist_base;

// START-cyclist_base
CREATE TABLE IF NOT EXISTS cycling.cyclist_base (cid UUID PRIMARY KEY,
name text,
age int,
birthday date,
country text);
// END-cyclist_base
// START-data
INSERT INTO cycling.cyclist_base (cid, name, age, birthday, country)
VALUES ('e7ae5cf3-d358-4d99-b900-85902fda9bb0','Alex FRAME', 22, '1993-06-18', 'New Zealand');

INSERT INTO cycling.cyclist_base (cid, name, age, birthday, country)
VALUES ('22084bf-4860-49d6-9a4b-6b5d3a79cbfb','Paolo TIRALONGO', 38, '1977-07-08', 'Italy');

INSERT INTO cycling.cyclist_base (cid, name, age, birthday, country)
VALUES ('6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47','Steven KRUIKSWIJK', 28, '1987-06-07', 'Netherlands');

INSERT INTO cycling.cyclist_base (cid, name, age, birthday, country)
VALUES ('ffdfa2a7-5fc6-49a7-bfdc-3fcdcfdd7156','Pascal EENKHOORN', 18, '1997-02-08', 'Netherlands');

INSERT INTO cycling.cyclist_base (cid, name, age, birthday, country)
VALUES ('18f471bf-f631-4bc4-a9a2-d6f6cf5ea503','Bram WELTEN', 18, '1997-03-29', 'Netherlands');

INSERT INTO cycling.cyclist_base (cid, name, age, birthday, country)
VALUES ('15a116fc-b833-4da6-ab9a-4a775752836','Adrien COSTA', 18, '1997-08-19', 'United States');

INSERT INTO cycling.cyclist_base (cid, name, age, birthday, country)
VALUES ('862cc51f-00a1-4d5a-976b-a359cab7300e','Joakim BUKDAL', 20, '1994-09-04', 'Denmark');

INSERT INTO cycling.cyclist_base (cid, name, age, birthday, country)
VALUES ('c9c9c484-5e4a-4542-8203-8d047a01b8a8','Cristian EGIDIO', 27, '1987-09-04', 'Brazil');

// END-data
CAPTURE 'select_all_from_cyclist_base.results';
SELECT *
FROM cycling.cyclist_base;
CAPTURE OFF;

// START-alt
ALTER TABLE cycling.cyclist_base
WITH comment = 'basic cyclist information';
// END-alt

// START-specr
ALTER TABLE cycling.cyclist_base
WITH speculative_retry = '95percentile';
ALTER TABLE cycling.cyclist_base
WITH speculative_retry = '10ms';

---

cyclist_career_teams

Set data type example.

SOURCE '0_create_keyspace.cql';

DROP TABLE IF EXISTS cycling.cyclist_career_teams;

// Find all teams that a cyclist has been a member of
// CREATE TABLE WITH SET
// START-setColumn
CREATE TABLE IF NOT EXISTS cycling.cyclist_career_teams (
  id UUID PRIMARY KEY,
  lastname text,
  teams set<text>
);
// END-setColumn

DROP INDEX IF EXISTS cycling.teams_idx;

// Create an index on the set collection
// START-createidxset
CREATE INDEX IF NOT EXISTS teams_idx
ON cycling.cyclist_career_teams (teams);
// END-createidxset

// START-insertdatasetvos
INSERT INTO cycling.cyclist_career_teams (
  id, lastname, teams
) VALUES (
  5b6962dd-3f90-4c93-8f61-eabfa4a03e2,
  'VOS',
  {
    'Rabobank-Liv Woman Cycling Team',
    'Rabobank-Liv Giant',
    'Rabobank Women Team',
    'Nederland bloeit'
  }
);
// END-insertdatasetvos

INSERT INTO cycling.cyclist_career_teams (id, lastname, teams) VALUES (e7cd5752-bc0d-4157-a80f-7523add686cd, 'VAN DER BREGGEN', { 'Rabobank-Liv Woman Cycling Team','Sengers Ladies Cycling Team','Team Flexpoint' } );
INSERT INTO cycling.cyclist_career_teams (id, lastname, teams) VALUES (cb07baad-eac8-4f68-b28a-bddc06a0de23, 'ARMISTEAD', { 'Boels-Dolmans Cycling Team','AA Drink - Leontien.nl','Team Garmin - Cervelo' } );
INSERT INTO cycling.cyclist_career_teams (id, lastname, teams) VALUES (1c9ebc13-1eab-4ad5-be87-dce43126d40, 'BRAND', { 'Rabobank-Liv Woman Cycling Team','Rabobank-Liv Giant','AA Drink - Leontien.nl','Leontien.nl' } );

CAPTURE 'select_all_from_cyclist_career_teams.results';
// START-select
SELECT *
FROM cycling.cyclist_career_teams;
// END-select
CAPTURE OFF;
Examples

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

// START-remove
UPDATE cycling.cyclist_career_teams
SET teams = teams - {'DSB Bank Nederland bloeit'}
WHERE id = 5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
// END-remove

// START-clear
UPDATE cycling.cyclist_career_teams
SET teams = {}
WHERE id = 5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
// END-clear

// START-selectwithpartitionkey
SELECT id, lastname, teams
FROM cycling.cyclist_career_teams
WHERE id = 5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
// END-selectwithpartitionkey

CAPTURE 'select_all_from_cyclist_career_teams_contains_value.results';
// START-select_with_contains_value
SELECT * FROM cycling.cyclist_career_teams
WHERE teams CONTAINS 'Rabobank-Liv Giant';
// END-select_with_contains_value
CAPTURE OFF;

// START-select_with_contains_key
SELECT * FROM cycling.cyclist_career_teams
WHERE teams CONTAINS 'Rabobank-Liv Giant';
// END-select_with_contains_key

cyclist_category
Clustering column simple example with corresponding query.

// Find cyclists that fit a particular category
SOURCE '0_create_keyspace.cql';

// CREATE TABLE CLUSTERING ORDER, PRIMARY KEY: PARTITION KEY + 1 CLUSTERING COLUMN, SIMPLE WHERE QUERY
DROP TABLE IF EXISTS cycling.cyclist_category;

// START-use_and_cpartitionkey
USE cycling;
CREATE TABLE IF NOT EXISTS cyclist_category (
    category text,
    points int,
    id UUID,
    lastname text,
    PRIMARY KEY (category, points)
) WITH CLUSTERING ORDER BY (points DESC);
// END-use_and_cpartitionkey

DROP TABLE IF EXISTS cycling.cyclist_category;
Examples

CREATE TABLE IF NOT EXISTS cycling.cyclist_category (  
category text,  
points int,  
id UUID,  
lastname text,  
PRIMARY KEY (category, points)  
)  
WITH CLUSTERING ORDER BY (points DESC);

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  
('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');  
INSERT INTO cycling.cyclist_category (category, points, id, lastname) VALUES  
('Time-trial',3,6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47,'KRUIJSWIJK');

RESTRICT ROWS ON cycling.cyclist_category USING category;

SELECT *  
FROM cycling.cyclist_category  
WHERE category = 'Time-trial'  
ORDER BY points DESC;

DROP TABLE IF EXISTS cycling.cyclist_emails;

CREATE TABLE IF NOT EXISTS cycling.cyclist_emails (  
userid text PRIMARY KEY,  
id UUID,  
emails set<text>  
)  
WITH ID = '1bb7516e-b140-11e8-96f8-529269fb1459';

DROP TABLE IF EXISTS cycling.cyclist_emails;
// Create table again with a known table id
// START-recreateID
CREATE TABLE IF NOT EXISTS cycling.cyclist_emails (    userid text PRIMARY KEY,    id UUID,    emails set<text> ) WITH ID = '1bb7516e-b140-11e8-96f8-529269fb1459';
// END-recreateID

// Retrieve the table id
// START-select_id_from_system_schema
SELECT id FROM system_schema.tables WHERE keyspace_name = 'cycling' AND table_name = 'cyclist_emails';
// END-select_id_from_system_schema

cyclist_expenses

Empty table to demonstrate batches.

SOURCE '0_create_keyspace.cql';

DROP TABLE IF EXISTS cycling.cyclist_expenses;

CREATE TABLE IF NOT EXISTS cycling.cyclist_expenses (    cyclist_name text,    balance float STATIC,    expense_id int,    amount float,    description text,    paid boolean,    PRIMARY KEY (cyclist_name, expense_id) ) ;

// START-batch
BEGIN BATCH USING TIMESTAMP 1481124356754405
    INSERT INTO cycling.cyclist_expenses (    cyclist_name, expense_id, amount, description, paid ) VALUES (    'Vera ADRIAN', 2, 13.44, 'Lunch', true    );
    INSERT INTO cycling.cyclist_expenses (    cyclist_name, expense_id, amount, description, paid ) VALUES (    'Vera ADRIAN', 3, 25.00, 'Dinner', true    );
APPLY BATCH;
// END-batch

// START-con
BEGIN BATCH
    INSERT INTO cycling.cyclist_expenses (    cyclist_name, expense_id ) VALUES (    'Joe WALLS', 1    ) IF NOT EXISTS;
    INSERT INTO cycling.cyclist_expenses (    cyclist_name, expense_id ) VALUES (    'Jae WALLS', 1    ) IF NOT EXISTS;
APPLY BATCH;
// END-con
Examples

cyclist_name, expense_id, amount, description, paid
) VALUES (
    'Joe WALLS', 1, 8, 'burrito', false
);

APPLY BATCH;
// END-con

insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Anna VAN DER BREGGEN',1,50,'dinner');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Anna VAN DER BREGGEN',2,50,'lunch');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Michael MATTHEWS',3,25,'lunch');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Michael MATTHEWS',4,25,'lunch');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Alex FRAME',5,40,'dinner');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Alex FRAME',6,40,'lunch');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Steven KRUIKSWIJK',7,20,'lunch');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Steven KRUIKSWIJK',8,39,'dinner');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Marianne VOS',9,20,'lunch');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Marianne VOS',10,20,'dinner');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Paolo TIRALONGO',11,10,'dinner');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Paolo TIRALONGO',12,10,'lunch');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Anna VAN DER BREGGEN',13,50,'dinner');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Anna VAN DER BREGGEN',14,25,'lunch');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Anna VAN DER BREGGEN',15,35,'dinner');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Anna VAN DER BREGGEN',16,20,'lunch');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Anna VAN DER BREGGEN',17,70,'dinner');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Anna VAN DER BREGGEN',18,10,'lunch');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Anna VAN DER BREGGEN',19,50,'dinner');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Anna VAN DER BREGGEN',20,25,'lunch');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Paolo TIRALONGO',21,35,'dinner');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Paolo TIRALONGO',22,20,'lunch');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Paolo TIRALONGO',23,70,'dinner');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Paolo TIRALONGO',24,10,'lunch');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Paolo TIRALONGO',25,11,'dinner');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Paolo TIRALONGO',26,12,'lunch');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Paolo TIRALONGO',27,13,'lunch');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Paolo TIRALONGO',28,14,'lunch');
insert into cycling.cyclist_expenses (cyclist_name,expense_id,amount,description) values
    ('Paolo TIRALONGO',29,15,'dinner');

Examples

```cql
insert into cycling.cyclist_expenses (cyclist_name, expense_id, amount, description) values ('Paolo TIRALONGO', 30, 16, 'lunch');
insert into cycling.cyclist_expenses (cyclist_name, expense_id, amount, description) values ('Paolo TIRALONGO', 31, 17, 'dinner');
insert into cycling.cyclist_expenses (cyclist_name, expense_id, amount, description) values ('Paolo TIRALONGO', 32, 18, 'breakfast');
insert into cycling.cyclist_expenses (cyclist_name, expense_id, amount, description) values ('Steven KRUIKSWIJK', 33, 3, 'coffee');
insert into cycling.cyclist_expenses (cyclist_name, expense_id, amount, description) values ('Steven KRUIKSWIJK', 34, 3, 'coffee');
insert into cycling.cyclist_expenses (cyclist_name, expense_id, amount, description) values ('Steven KRUIKSWIJK', 35, 3, 'coffee');
insert into cycling.cyclist_expenses (cyclist_name, expense_id, amount, description) values ('Steven KRUIKSWIJK', 36, 3, 'coffee');
insert into cycling.cyclist_expenses (cyclist_name, expense_id, amount, description) values ('Steven KRUIKSWIJK', 37, 3, 'coffee');
insert into cycling.cyclist_expenses (cyclist_name, expense_id, amount, description) values ('Steven KRUIKSWIJK', 38, 3, 'coffee');
insert into cycling.cyclist_expenses (cyclist_name, expense_id, amount, description) values ('Steven KRUIKSWIJK', 39, 3, 'coffee');
insert into cycling.cyclist_expenses (cyclist_name, expense_id, amount, description) values ('Michael MATTHEWS', 40, 25, 'lunch');
insert into cycling.cyclist_expenses (cyclist_name, expense_id, amount, description) values ('Michael MATTHEWS', 41, 25, 'lunch');
insert into cycling.cyclist_expenses (cyclist_name, expense_id, amount, description) values ('Michael MATTHEWS', 42, 25, 'lunch');
insert into cycling.cyclist_expenses (cyclist_name, expense_id, amount, description) values ('Michael MATTHEWS', 43, 25, 'lunch');
insert into cycling.cyclist_expenses (cyclist_name, expense_id, amount, description) values ('Michael MATTHEWS', 44, 25, 'lunch');
insert into cycling.cyclist_expenses (cyclist_name, expense_id, amount, description) values ('Michael MATTHEWS', 45, 25, 'lunch');
insert into cycling.cyclist_expenses (cyclist_name, expense_id, amount, description) values ('Michael MATTHEWS', 46, 25, 'lunch');
insert into cycling.cyclist_expenses (cyclist_name, expense_id, amount, description) values ('Michael MATTHEWS', 47, 25, 'lunch');
```

cyclist_id

Composite partition key example with a clustering column.

```cql
SOURCE '0_create_keyspace.cql';

DROP TABLE IF EXISTS cycling.cyclist_id;

// Find a cyclist's id given lastname and firstname
// Another CREATE TABLE using COMPOSITE PARTITION KEY
// INDEX ALSO GOOD FOR THIS TABLE

// START-comp_pk
CREATE TABLE IF NOT EXISTS cycling.cyclist_id (  
   lastname text,  
   firstname text,  
   age int,  
   id UUID,  
   PRIMARY KEY ((lastname, firstname), age)  
);
// END-comp_pk

// START-altercdc
ALTER TABLE cycling.cyclist_id  
WITH CDC = false;  
// END-altercdc

INSERT INTO cycling.cyclist_id (  
```
Examples

```cql
lastname, firstname, age, id
) VALUES
    'EENKHOORN', 'Pascal', 18, ffdfa2a7-5fc6-49a7-bf0c-3fcdcfdd7156
);

INSERT INTO cycling.cyclist_id ( lastname, firstname, age, id )
VALUES ( 'WELTEN', 'Bram', 18, 18f471bf-f631-4bc4-a92a-d6f6cf5ea503 )
);

INSERT INTO cycling.cyclist_id ( lastname, firstname, age, id )
VALUES ( 'COSTA', 'Adrien', 17, 15a116fc-b833-4da6-ab9a-4a7775752836 )
);

// START-resetid
UPDATE cycling.cyclist_id
SET id = UUID()
WHERE lastname = 'WELTEN'
    AND firstname = 'Bram'
    AND age = 18
IF EXISTS;
// END-resetid

// START-ifid
UPDATE cycling.cyclist_id
SET id = UUID()
WHERE lastname = 'WELTEN'
    AND firstname = 'Bram'
    AND age = 18
IF id = 18f471bf-f631-4bc4-a92a-d6f6cf5ea503;
// END-ifid

// START-ifnull
UPDATE cycling.cyclist_id
SET id = UUID()
WHERE lastname = 'Smith'
    AND firstname = 'Joe'
    AND age = 22
IF id = NULL;
// END-ifnull

// START-select_with_in_and_equals
SELECT *
FROM cycling.cyclist_id
WHERE lastname = 'EENKHOORN'
    AND firstname = 'Pascal'
    AND age IN (17, 18);
// END-select_with_in_and_equals

Queries

SOURCE 'cyclist_id-table.cql';

// START-basicwhere
SELECT
    firstname AS first,
    lastname AS last
FROM cycling.cyclist_id
WHERE lastname = 'COSTA'
    AND firstname = 'Adrien';
```
Composite partition key example without a clustering column.

```cql
SOURCE '0_create_keyspace.cql';
// Composite partition key example without a clustering column
DROP TABLE IF EXISTS cycling.cyclist_id;
// create table without age as clustering column
CREATE TABLE IF NOT EXISTS cycling.cyclist_id (
  lastname text,
  firstname text,
  age int,
  id uuid,
  PRIMARY KEY ((lastname, firstname))
);
// insert data
INSERT INTO cycling.cyclist_id (
  lastname, firstname, age, id
) VALUES (
  'EENKHOORN', 'Pascal', 18, ffdfa2a7-5fc6-49a7-bfdc-3fcdcfdd7156
);
INSERT INTO cycling.cyclist_id (
  lastname, firstname, age, id
) VALUES (
  'WELTEN', 'Bram', 18, 18f471bf-f631-4bc4-a9a2-d6f6cf5ea503
);
INSERT INTO cycling.cyclist_id (
  lastname, firstname, age, id
) VALUES (
  'COSTA', 'Adrien', 17, 15a116fc-b833-4da6-ab9a-4a7775752836
);
// attempt to delete data using IF EXISTS, applied false because Bram Jones does not exist
CAPTURE 'cyclist_id_delete_if_exists.results';
// START-deleteifexists
DELETE id FROM cycling.cyclist_id
WHERE lastname = 'JONES'
  AND firstname = 'Bram'
IF EXISTS;
// END-deleteifexists
CAPTURE OFF;
// delete data using an IF, age does not match so applied is false
CAPTURE 'cyclist_id_delete_if.results';
// START-deletedataif
DELETE id FROM cycling.cyclist_id
WHERE lastname = 'WELTEN'
  AND firstname = 'Bram'
IF age = 20;
// END-deletedataif
```
Examples

**cyclist_name**
Simple single column partition key. Demonstrates inserting, updating, and deleting data.

```
SOURCE '0_create_keyspace.cql';

// START-droptable
DROP TABLE IF EXISTS cycling.cyclist_name;
// END-droptable

// START-pk_at_start
USE cycling;
CREATE TABLE cyclist_name {
  id UUID PRIMARY KEY, lastname text, firstname text
};
// END-pk_at_start

DROP TABLE IF EXISTS cycling.cyclist_name;
// START-pk_at_end
USE cycling;
CREATE TABLE cyclist_name {
  id UUID, lastname text, firstname text, PRIMARY KEY (id)
};
// END-pk_at_end

DROP TABLE IF EXISTS cycling.cyclist_name;
// START-keyspace_included
CREATE TABLE cycling.cyclist_name {
  id UUID, lastname text, firstname text, PRIMARY KEY (id)
};
// END-keyspace_included

DROP TABLE IF EXISTS cycling.cyclist_name;
// Create a table with a simple partition key
// START-simple
CREATE TABLE IF NOT EXISTS cycling.cyclist_name {
  id UUID PRIMARY KEY,
  lastname text,
  firstname text
};
// END-simple

// START-uuid
INSERT INTO cycling.cyclist_name {
  id
} VALUES (
  uuid()
);
// END-uuid

TRUNCATE cycling.cyclist_name;
// Insert data with TTL and timestamp
// START-inserttimestamp
INSERT INTO cycling.cyclist_name {
  id, lastname, firstname
} VALUES {
  6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47, 'KRUIKSWIJK', 'Steven'
}
USING TTL 86400
```
AND TIMESTAMPT 123456789;
// END-insertstampttl

// START-insertifnotexists
INSERT INTO cycling.cyclist_name (id, lastname, firstname)
VALUES (c4b65263-fe58-484e-83e8-f0e1c13d518f, 'RATTO', 'Rissella')
IF NOT EXISTS;
// END-insertifnotexists

DROP INDEX IF EXISTS fn_prefix;
// START-fn_prefix
CREATE CUSTOM INDEX IF NOT EXISTS fn_prefix
ON cycling.cyclist_name (firstname)
USING 'org.apache.cassandra.index.sasi.SASIIndex';
// END-fn_prefix

TRUNCATE cycling.cyclist_name;
// Insert 6 rows of cyclists
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-752ad88b0, 'VAN DER BREGGEN','Anna');
INSERT INTO cycling.cyclist_name (id, lastname, firstname) VALUES
e7ae5cf3-d358-4d99-b900-85902fda49b0, 'FRAME','Alex');
INSERT INTO cycling.cyclist_name (id, lastname, firstname) VALUES
(22084bf-4860-4d6-9a4b-6b5d3a9cbf, 'MARIANNE','Steven');
INSERT INTO cycling.cyclist_name (id, lastname, firstname) VALUES
(fb372533-eb95-4bb4-8685-6ef61e99c4a, 'MATTHEWS', 'Michael');

// View output
CAPTURE 'select_all_from_cycling_initial.results';
// START-select_initial
SELECT *
FROM cycling.cyclist_name;
// END-select_initial
CAPTURE OFF;

CAPTURE 'select_marianne_from_cyclist_name.results';
// START-select_Marianne
SELECT *
FROM cycling.cyclist_name
WHERE firstname = 'Marianne';
// END-select_Marianne
CAPTURE OFF;

CAPTURE 'select_like_M_from_cyclist_name.results';
// START-select_like_M_start
SELECT *
FROM cycling.cyclist_name
WHERE firstname LIKE 'M%';
// END-select_like_M_start
CAPTURE OFF;

// START-no_match_upper
SELECT * FROM cycling.cyclist_name WHERE firstname = 'MARIANNE';
// END-no_match_upper

// START-no_match_lower
Examples

SELECT * FROM cycling.cyclist_name WHERE firstname LIKE 'm%';
// END-no_match_lower

// START-no_match_equality
SELECT * FROM cycling.cyclist_name WHERE firstname = 'M%';
SELECT * FROM cycling.cyclist_name WHERE firstname = '%M';
SELECT * FROM cycling.cyclist_name WHERE firstname = '%M%';
SELECT * FROM cycling.cyclist_name WHERE firstname = 'm%';
// END-no_match_equality

DROP INDEX IF EXISTS fn_contains;

// START-fn_contains
CREATE CUSTOM INDEX IF NOT EXISTS fn_contains
ON cycling.cyclist_name (firstname)
USING 'org.apache.cassandra.index.sasi.SASIIndex'
WITH OPTIONS = { 'mode': 'CONTAINS' };
// END-fn_contains

// These examples generate an error if they are run before the fn_contains index
// is created
// START-like
SELECT * FROM cycling.cyclist_name WHERE firstname LIKE '%m%';
SELECT * FROM cycling.cyclist_name WHERE firstname LIKE '%m%' ALLOW FILTERING;
SELECT * FROM cycling.cyclist_name WHERE firstname LIKE '%M%';
// END-like

CAPTURE 'select_marianne_allow_filtering_from_cyclist_name.results';
// START-select_marianne_allow_filtering
SELECT *
FROM cycling.cyclist_name
WHERE firstname = 'Marianne'
ALLOW FILTERING;
// END-select_marianne_allow_filtering
CAPTURE OFF;

CAPTURE 'select_like_M_partial_from_cyclist_name.results';
// START-select_like_M_partial
SELECT *
FROM cycling.cyclist_name
WHERE firstname LIKE '%M%';
// END-select_like_M_partial
CAPTURE OFF;

CAPTURE 'select_arianne_from_cyclist_name.results';
// START-select_arianne
SELECT *
FROM cycling.cyclist_name
WHERE firstname LIKE '%arian%';
// END-select_arianne
CAPTURE OFF;

CAPTURE 'select_arian_from_cyclist_name.results';
// START-select_arian_name
SELECT *
FROM cycling.cyclist_name
WHERE firstname LIKE '%arian%';
// END-select_arian_name
CAPTURE OFF;

CAPTURE 'select_mar_from_cyclist_name.results';
// START-select_greater_Mar
SELECT *
FROM cycling.cyclist_name
WHERE firstname > 'Mar'
ALLOW FILTERING;
// END-select_greater_Mar
CAPTURE OFF;

// These examples do not return a match when they are run before the
// fn_suffix_allcase index is created
// START-no_match_partial
SELECT * FROM cycling.cyclist_name WHERE firstname = 'MariAnne' ALLOW FILTERING;
SELECT * FROM cycling.cyclist_name WHERE firstname LIKE '%m%';
SELECT * FROM cycling.cyclist_name WHERE firstname LIKE '%M';
SELECT * FROM cycling.cyclist_name WHERE firstname LIKE 'm%';
// END-no_match_partial
DROP INDEX IF EXISTS fn_suffix_allcase;

// START-fn_suffix_allcase
CREATE CUSTOM INDEX IF NOT EXISTS fn_suffix_allcase
ON cycling.cyclist_name (firstname)
USING 'org.apache.cassandra.index.sasi.SASIIndex'
WITH OPTIONS = {
    'mode': 'CONTAINS',
    'analyzer_class': 'org.apache.cassandra.index.sasi.analyzer.NonTokenizingAnalyzer',
    'case_sensitive': 'false'
};
// END-fn_suffix_allcase

CAPTURE 'select_like_m_lower_from_cyclist_name.results';
// START-select_like_m_lower
SELECT *
FROM cycling.cyclist_name
WHERE firstname LIKE '%m%';
// END-select_like_m_lower
CAPTURE OFF;

// START-alter
ALTER TABLE cycling.cyclist_name
ADD age int;

UPDATE cycling.cyclist_name
SET age = 23
WHERE id = 5b6962dd-3f90-4c93-8f61-eabfa4a803e2;

INSERT INTO cycling.cyclist_name (id, age, firstname, lastname)
VALUES (8566eb59-07df-43b1-a21b-666a3c08c08a, 18, 'Marianne', 'DAAE');
// END-alter

CAPTURE 'select_name_and_age_filter_from_cyclist_name.results';
// START-select_name_and_age_filter
SELECT *
FROM cycling.cyclist_name
WHERE firstname = 'Marianne'
AND age > 20
ALLOW FILTERING;
// END-select_name_and_age_filter
CAPTURE OFF;

// Undo the changes in the alter section
DELETE FROM cycling.cyclist_name
WHERE id = 8566eb59-07df-43b1-a21b-666a3c08c08a;
ALTER TABLE cycling.cyclist_name
DROP age;

// Delete data in specified columns from a row, changes cyclist Alex Frame to null in
name columns
Examples

// START-deletecolumndata
DELETE firstname, lastname
FROM cycling.cyclist_name
WHERE id = e7ae5cf3-d358-4d99-b900-85902fda9bb0;
// END-deletecolumndata

// View output
CAPTURE 'cyclist_name_delete_firstname_and_lastname.results';
SELECT *
FROM cycling.cyclist_name;
CAPTURE OFF;

// Delete an entire row, deletes cyclist Alex Frame altogether
CAPTURE 'cyclist_name_delete_applied_true.results';
// START-deleteentirerow
DELETE FROM cycling.cyclist_name
WHERE id = e7ae5cf3-d358-4d99-b900-85902fda9bb0
IF EXISTS;
// END-deleteentirerow
CAPTURE OFF;

// View output
CAPTURE 'cyclist_name_delete_entire_row.results';
SELECT *
FROM cycling.cyclist_name;
CAPTURE OFF;

// Delete a row based on a static column condition, attempts to delete cyclist Michael Matthews but lastname does not match
CAPTURE 'cyclist_name_delete_condition.results';
// START-deletecondition
DELETE FROM cycling.cyclist_name
WHERE id = fb372533-eb95-4bb4-8685-6ef61e994caa
IF firstname = 'Michael'
   AND lastname = 'Smith';
// END-deletecondition
CAPTURE OFF;

// View output
SELECT *
FROM cycling.cyclist_name;

// Delete a row with a WHERE clause, deletes cyclist Steven Kruikswijk
// START-deleterowwhere
DELETE FROM cycling.cyclist_name
WHERE id = 6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47;
// END-deleterowwhere

// View output
CAPTURE 'cyclist_name_delete_row_where.results';
SELECT *
FROM cycling.cyclist_name;
CAPTURE OFF;

// Delete rows using an IN clause on a primary key, deletes cyclists Marianne and Paolo
// START-deletein
DELETE FROM cycling.cyclist_name
WHERE id IN (5b6962dd-3f90-4c93-8f61-eabfa4a803e2, 220844bf-4860-49d6-9a4b-6b5d3a79cbfb);
// END-deletein

// View output
CAPTURE 'cyclist_name_delete_in.results';
SELECT *
FROM cycling.cyclist_name;
CAPTURE OFF;
CAPTURE OFF;

// Reinsert data
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-7523add8d6bc, '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 (e7ae5cf3-d358-4d99-b900-85902fda9bb0, 'FRAME', 'Alex');
INSERT INTO cycling.cyclist_name (id, lastname, firstname) VALUES (220844bf-4860-496d-9a4b-6b5d3a79cbfb, 'TIRALONGO', 'Paolo');
INSERT INTO cycling.cyclist_name (id, lastname, firstname) VALUES (220844bf-4860-496d-9a4b-6b5d3a79cbfb, 'TIRALONGO', 'Paolo');
INSERT INTO cycling.cyclist_name (id, lastname, firstname) VALUES (fb372533-eb95-4bb4-8685-6ef61e994caa, 'MATTHEWS', 'Michael');

// START-comment
ALTER TABLE cycling.cyclist_name
ADD comment text;
// END-comment

// START-update_set_comment
UPDATE cycling.cyclist_name
SET comment = 'Rides hard, gets along with others, a real winner'
WHERE id = fb372533-eb95-4bb4-8685-6ef61e994caa;

UPDATE cycling.cyclist_name
SET comment = 'Rides fast, does not get along with others, a real dude'
WHERE id = 5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
// END-update_set_comment

DROP INDEX IF EXISTS cycling.stdanalyzer_idx;
// START-stdanalyzer_idx
CREATE CUSTOM INDEX IF NOT EXISTS stdanalyzer_idx
ON cycling.cyclist_name (comment) USING 'org.apache.cassandra.index.sasi.SASIIndex'
WITH OPTIONS = {
'mode': 'CONTAINS',
'analyzers_class': 'org.apache.cassandra.index.sasi.index.standard.StandardAnalyzer',
'analyzed': 'true',
'tokenization_skip_stop_words': 'and, the, or',
'tokenization_enable_stemming': 'true',
'tokenization_normalize_lowercase': 'true',
'tokenization_locale': 'en'};
// END-stdanalyzer_idx

SOURCE 'cyclist_name-queries.cql';

// Undo UPDATE statements
UPDATE cycling.cyclist_name
SET comment = NULL
WHERE id = fb372533-eb95-4bb4-8685-6ef61e994caa;

UPDATE cycling.cyclist_name
SET comment = NULL
WHERE id = 5b6962dd-3f90-4c93-8f61-eabfa4a803e2;

// START-update
UPDATE cycling.cyclist_name
SET comment = 'Rides hard, gets along with others, a real winner'
WHERE id = fb372533-eb95-4bb4-8685-6ef61e994caa
IF EXISTS;
// END-update

// START-multicolumns
Examples

```cql
UPDATE cycling.cyclist_name
SET
  firstname = 'Marianne',
  lastname = 'VOS'
WHERE id = 88b8fd18-b1ed-4e96-bf79-4280797c8a80;
// END-multcolumns

// START-records
UPDATE cycling.cyclist_name
SET firstname = NULL
WHERE id IN (
  5b6962dd-3f90-4c93-8f61-eabfa4a803e2,
  fb372533-eb95-4bb4-8685-6ef61e994caa
);
// END-records

// START-upsert
UPDATE cycling.cyclist_name
SET
  firstname = 'Anna',
  lastname = 'VAN DER BREGGEN'
WHERE id = e7cd5752-bc0d-4157-a80f-7523add8dbc;
// END-upsert

// START-if
UPDATE cycling.cyclist_name
SET comment = 'Rides hard, gets along with others, a real winner'
WHERE id = fb372533-eb95-4bb4-8685-6ef61e994caa
IF comment = NULL;
// END-if

// START-fail
UPDATE cycling.cyclist_name
SET comment = 'Rides hard, gets along with others, a real winner'
WHERE id = fb372533-eb95-4bb4-8685-6ef61e994cac
IF comment = 'Rides hard, gets along with others, a real winner';
// END-fail

// START-count_lastname
SELECT COUNT(lastname)
FROM cycling.cyclist_name;
// END-count_lastname

// START-count_rows
SELECT COUNT(*)
FROM cycling.cyclist_name;
// END-count_rows
```

cyclist_points

Select aggregated data using the built-in functions such as `sum`.

```cql
SOURCE '0_create_keyspace.cql';

DROP TABLE IF EXISTS cycling.cyclist_points;
// START-cpoints
CREATE TABLE IF NOT EXISTS cycling.cyclist_points (
  id UUID,
  race_points int,
  firstname text,
  lastname text,
  race_title text,
  PRIMARY KEY (id, race_points)
);
```

CQL for DataStax Enterprise 6.7 Latest DSE version Latest 6.7 patch: 6.7.7
// Insert data
INSERT INTO cycling.cyclist_points (id, race_points, firstname, lastname, race_title)
VALUES (e3b19ec4-774a-4d1c-9e5a-decec1e30aac, 6, 'Giorgia', 'BRONZINI', 'Trofeo Alfredo Binda – Commune di Cittiglio');

INSERT INTO cycling.cyclist_points (id, race_points, firstname, lastname, race_title)
VALUES (e3b19ec4-774a-4d1c-9e5a-decec1e30aac, 75, 'Giorgia', 'BRONZINI', 'Act van Westerveld');

INSERT INTO cycling.cyclist_points (id, race_points, firstname, lastname, race_title)
VALUES (e3b19ec4-774a-4d1c-9e5a-decec1e30aac, 120, 'Giorgia', 'BRONZINI', 'Tour of Chongming Island World Cup');

// Calculate sum of total points for a cyclist
// START-simplesumpoints
SELECT SUM(race_points)
FROM cycling.cyclist_points
WHERE id = e3b19ec4-774a-4d1c-9e5a-decec1e30aac;
// END-simplesumpoints

// Filter and SUM on a clustering column using an AND relation
CAPTURE 'select_sum_race_points_from_cyclist_points.results';
// START-filtersumpoints
SELECT SUM(race_points)
FROM cycling.cyclist_points
WHERE id = e3b19ec4-774a-4d1c-9e5a-decec1e30aac
AND race_points > 7;
// END-filtersumpoints
CAPTURE OFF;

// Filter for MAX points
CAPTURE 'select_max_race_points_from_cyclist_points.results';
// START-filtermaxpts
SELECT MAX(race_points)
FROM cycling.cyclist_points
WHERE id = e3b19ec4-774a-4d1c-9e5a-decec1e30aac;
// END-filtermaxpts
CAPTURE OFF;

// Filter for MIN points
CAPTURE 'select_min_race_points_from_cyclist_points.results';
// START-filterminpts
SELECT MIN(race_points)
FROM cycling.cyclist_points
WHERE id = e3b19ec4-774a-4d1c-9e5a-decec1e30aac;
// END-filterminpts
CAPTURE OFF;

// Filter for AVG points
CAPTURE 'select_avg_race_points_from_cyclist_points.results';
// START-filteravgpts
SELECT AVG(race_points)
FROM cycling.cyclist_points
WHERE id = e3b19ec4-774a-4d1c-9e5a-decec1e30aac;
// END-filteravgpts
CAPTURE OFF;
Examples

// Get WRITETIME for a column
// START-writetime
SELECT WRITETIME (firstname)
FROM cycling.cyclist_points
WHERE id = e3b19ec4-774a-4d1c-9e5a-decec1e30aac;
// END-writetime

cyclist_races
Create user-defined type, insert data into UDT and time fields.

SOURCE '0_create_keyspace.cql';
DROP TABLE IF EXISTS cycling.cyclist_races;

// Find all races for a particular cyclist
// CREATE TYPE - User-Defined Type, race
// CREATE TABLE WITH LIST, SIMPLE PRIMARY KEY
SOURCE 'race-type.cql';

// START-frozenlist
CREATE TABLE IF NOT EXISTS cycling.cyclist_races (
    id UUID PRIMARY KEY,
    lastname text,
    firstname text,
    races list<FROZEN <race>>
);
// END-frozenlist

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

INSERT INTO cycling.cyclist_races (
    id, lastname, firstname, races
) VALUES (
    e7cd5752-bc0d-4157-a80f-7523add8bcd,
    'VAN DER BREGGEN', 'Anna',
    [
        
        {race_title:'Festival Luxembourgise du cyclisme feminin Elsy Jacobs - Prologue - Garnich > Garnich',race_date:'2015-05-01',race_time:'08:13:00'},
        {race_title:'Festival Luxembourgise du cyclisme feminin Elsy Jacobs - Stage 2 - Garnich > Garnich',race_date:'2015-05-02',race_time:'02:41:52'},
        {race_title:'Festival Luxembourgise du cyclisme feminin Elsy Jacobs - Stage 3 - Mamer > Mamer',race_date:'2015-05-03',race_time:'02:31:24'}
    ]
);
### cyclist_sponsors

**Cyclist sponsors set with TIMESTAMP and TTL**

```cql
SOURCE '0_create_keyspace.cql';

DROP TABLE IF EXISTS cycling.cyclist_sponsors_expire;

// START-cyclistsponsorsexpireset
CREATE TABLE IF NOT EXISTS cycling.cyclist_sponsors_expire (cyclist_name text PRIMARY KEY, sponsorship set<text>);
// END-cyclistsponsorsexpireset

// START-insertTTLexpireset
INSERT INTO cycling.cyclist_sponsors_expire (cyclist_name, sponsorship)
VALUES ( 'PRIETO, Marcela', 
'{ "Castrelli", "Alfa Romeo" }' 
) USING TIMESTAMP 100
AND TTL 10000;
// END-insertTTLexpireset

// START-updateTTLexpireset
UPDATE cycling.cyclist_sponsors_expire
USING TIMESTAMP 200
AND TTL 20000
SET sponsorship += { 'Tag Heuer' } WHERE cyclist_name = 'PRIETO, Marcela';
// END-updateTTLexpireset

// START-selectTTLexpireset
SELECT * FROM cycling.cyclist_sponsors_expire;
// END-selectTTLexpireset
```

### cyclist_stats

**Using a UDT, inserting and updating data.**

```cql
// START-droptable
DROP TABLE IF EXISTS cycling.cyclist_stats;
// END-droptable

SOURCE 'basic_info-type.cql';
```
Examples

// START-createtable
CREATE TABLE IF NOT EXISTS cycling.cyclist_stats (
    id UUID PRIMARY KEY,
    lastname text,
    basics basic_info
);
// END-createtable

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

INSERT INTO cycling.cyclist_stats (
    id, lastname, basics
) VALUES (
    6cbc55e9-1943-47dc-91f2-f8f9e95992eb, 'VIGANO',
    { birthday:'1984-06-12',nationality:'Italy',weight:'67 kg',height:'1.82 m' }
);

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' }
);

// START-update
UPDATE cycling.cyclist_stats
SET basics.birthday = '2000-12-12'
WHERE id = 220844bf-4860-49d6-9a4b-6b5d3a79cbfb;
// END-update

cyclist_teams
Map data type example. Create an index on a map key and query for a value that contains a key.

SOURCE '0_create_keyspace.cql';

DROP TABLE IF EXISTS cycling.cyclist_teams;

// Create a table with a map
// START-mapColumn
CREATE TABLE IF NOT EXISTS cycling.cyclist_teams (
    id uuid PRIMARY KEY,
    firstname text,
    lastname text,
    teams map<int, text>
);
// END-mapColumn

DROP INDEX IF EXISTS cycling.team_year_idx;

// Create an index on a map key to find all cyclist/team combos for a year
// START-keysidx
CREATE INDEX IF NOT EXISTS team_year_idx
ON cycling.cyclist_teams ( KEYS (teams) );
// END-keysidx

// Insert team data into map for cyclist Vos
// START-insertmapdata
INSERT INTO cycling.cyclist_teams (
Examples

```sql
id, firstname, lastname, teams
VALUES (
  5b6962dd-3f90-4c93-8f61-eabfa4a803e2, 'Marianne', 'VOS',
  }
); // END-insertmapdata

// View data
CAPTURE 'select_all_from_cyclist_teams.results';
// START-select
SELECT *
FROM cycling.cyclist_teams;
// END-select
CAPTURE OFF;

// Delete an element from the map
// START-deletemapdata
DELETE teams[2014]
FROM cycling.cyclist_teams
WHERE id = 5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
// END-deletemapdata

// View data again, 2014 team gone
SELECT *
FROM cycling.cyclist_teams;

// Insert more cyclists
INSERT INTO cycling.cyclist_teams (id, firstname, lastname, teams)
VALUES (
  cb07baad-eac8-4f65-b28a-bddc06a0de23, 'Elizabeth', 'ARMITSTEAD',
  }
);

// View data
SELECT *
FROM cycling.cyclist_teams;

// Query for KEY year 2015
CAPTURE 'cyclist_team-queries.results';
// START-queryindexkey
SELECT *
FROM cycling.cyclist_teams
WHERE teams CONTAINS KEY 2015;
// END-queryindexkey
CAPTURE OFF;
```

Query output:

<table>
<thead>
<tr>
<th>id</th>
<th>firstname</th>
<th>lastname</th>
<th>teams</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Examples

+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
| 5b6962dd-3f90-4c93-8f61-eabfa4a803e2 | Marianne |        VOS | {2015: 'Rabobank-Liv Woman Cycling Team'} |

(2 rows)

events

Uses month and date of cycling races to show how to get ranges that span clustering columns. Example of copying data from a CSV into the table.

events table

SOURCE '0_create_keyspace.cql';

DROP TABLE IF EXISTS cycling.events;

// START-events_table
CREATE TABLE IF NOT EXISTS 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));

COPY cycling.events FROM 'events-data.csv' WITH HEADER = true AND DELIMITER = '|';

// START-select_with_in_and_equals_allow_filtering
SELECT * FROM cycling.events WHERE race = 'Superprestige - Hoogstraten -2017' AND start_month IN (1, 2) ALLOW FILTERING;

// END-select_with_in_and_equals_allow_filtering

// START-select_count
SELECT start_month, MAX(start_day), COUNT(*) FROM cycling.events WHERE year = 2017 AND discipline = 'Cyclo-cross';

// END-select_count

CAPTURE 'select_all_from_events_with_order_by.results';

// START-select_with_order_by
SELECT * FROM cycling.events WHERE year = 2017 AND discipline = 'Cyclo-cross';

// END-select_with_order_by
AND start_month = 1
AND start_day = 1
ORDER BY race;
// END-select_with_order_by
CAPTURE OFF;

// START-select_with_order_multiple_columns
SELECT *
FROM cycling.events
WHERE year = 2017
    AND discipline = 'Cyclo-cross'
    AND start_month = 1
ORDER BY start_day, race;
// END-select_with_order_multiple_columns

// START-select_with_where
SELECT *
FROM cycling.events
WHERE year = 2017
    AND discipline = 'Cyclo-cross'
    AND start_month = 1
    AND start_day = 1;
// END-select_with_where

CAPTURE 'select_from_events_non_existing_year.results';
// START-select_with_non_existing_year
SELECT start_month, MAX(start_day)
FROM cycling.events
WHERE year = 2022
ALLOW FILTERING;
// END-select_with_non_existing_year
CAPTURE OFF;

// Data from CSV
Insert the following comma separated data:

<table>
<thead>
<tr>
<th>Year</th>
<th>Discipline</th>
<th>Start_Month</th>
<th>Start_Day</th>
<th>Race</th>
<th>End_Month</th>
<th>End_Day</th>
<th>Location</th>
<th>UCI_code</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017</td>
<td>Cyclo-cross</td>
<td>1</td>
<td>1</td>
<td>DVV verzekering trofee - GP Sven Nys</td>
<td></td>
<td></td>
<td>Baal</td>
<td>C1</td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>1</td>
<td>1</td>
<td>Mitchellton Bay Cycling Classic</td>
<td>3</td>
<td>Geelong V</td>
<td>Victoria</td>
<td>CRT</td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>1</td>
<td>8</td>
<td>Cycling Australia Road National Championships</td>
<td>8</td>
<td>Ballarat</td>
<td>CN</td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Cyclo-cross</td>
<td>1</td>
<td>8</td>
<td>Belgian Cyclo-cross National Championships</td>
<td></td>
<td>Oostende</td>
<td>CN</td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Cyclo-cross</td>
<td>1</td>
<td>8</td>
<td>British Cyclo-cross National Championships</td>
<td></td>
<td>Bradford</td>
<td>CN</td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Cyclo-cross</td>
<td>1</td>
<td>8</td>
<td>Dutch Cyclo-cross National Championships</td>
<td></td>
<td>Holland</td>
<td>CN</td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Cyclo-cross</td>
<td>1</td>
<td>8</td>
<td>USA Cycling Cyclo-Cross National Championships</td>
<td></td>
<td>Hartford C</td>
<td>CN</td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>1</td>
<td>14</td>
<td>Santos Womens Tour</td>
<td>17</td>
<td>South Australia</td>
<td>WE</td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>1</td>
<td>22</td>
<td>Tour Down Under</td>
<td>22</td>
<td>South Australia</td>
<td>WT</td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Cyclo-cross</td>
<td>1</td>
<td>15</td>
<td>UCI Cyclo-cross World Cup Flugli Regione Lazio</td>
<td></td>
<td>Fiuggi</td>
<td>CDM</td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Cyclo-cross</td>
<td>1</td>
<td>22</td>
<td>UCI Cyclo-cross World Cup Hoogerheide</td>
<td></td>
<td>Hoogerheide</td>
<td>CDM</td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>1</td>
<td>23</td>
<td>Vuelta Ciclista a la Provincia de San Juan</td>
<td>29</td>
<td>San Juan</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>1</td>
<td>26</td>
<td>Cadel Evans Great Ocean Road Race - Towards Zero Race Melbourne</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Melbourne</td>
<td></td>
<td></td>
<td></td>
<td>CRT</td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>1</td>
<td>26</td>
<td>Challenge Mallorca: Trofeo Porreres-Felanitx-Ses Salines-Campos</td>
<td></td>
<td></td>
<td></td>
<td>Mallorca</td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>1</td>
<td>28</td>
<td>Challenge Mallorca: Trofeo Serra de Tramuntana -2017</td>
<td></td>
<td></td>
<td>Mallorca</td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>1</td>
<td>28</td>
<td>Cadel Evans Great Ocean Road Race</td>
<td></td>
<td></td>
<td>Geelong</td>
<td>WE</td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>1</td>
<td>28</td>
<td>Challenge Mallorca: Trofeo Andratx-Mirador des Colomer</td>
<td></td>
<td></td>
<td>Mallorca</td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Cyclo-cross</td>
<td>1</td>
<td>28</td>
<td>UCI Cyclo-cross World Championships</td>
<td>29</td>
<td>Bieles</td>
<td>CN</td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>1</td>
<td>29</td>
<td>Cadel Evans Great Ocean Road Race</td>
<td></td>
<td></td>
<td>Geelong</td>
<td>WT</td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>1</td>
<td>29</td>
<td>Grand Prix Cycliste la Marseillaise</td>
<td></td>
<td></td>
<td>France</td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>1</td>
<td>29</td>
<td>Mailorca Challenge: Trofeo Palma</td>
<td></td>
<td></td>
<td>Mallorca</td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>1</td>
<td>29</td>
<td>Dubai Tour</td>
<td>4</td>
<td>Dubai</td>
<td>2.1 HC</td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>1</td>
<td>31</td>
<td>Ladies Tour of Qatar</td>
<td>3</td>
<td>Qatar</td>
<td>WE</td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>2</td>
<td>1</td>
<td>Jayco Herald Sun Tour</td>
<td>5</td>
<td>Victoria</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>2</td>
<td>1</td>
<td>Volta a la Comunitat Valenciana</td>
<td>5</td>
<td>Valencia</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Year</td>
<td>Discipline</td>
<td>Race Name</td>
<td>Location</td>
<td>Category</td>
<td></td>
<td></td>
<td></td>
<td></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>Road</td>
<td>Etoile de Besseges</td>
<td>Languedoc-Roussillon</td>
<td>2.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Cyclo-cross</td>
<td>DVV verzekeringen trofee - Krawatencross</td>
<td>Lille</td>
<td>C1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Cyclo-cross</td>
<td>DVV verzekeringen trofee - Krawatencross</td>
<td>Lille</td>
<td>C1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>G.P. Costa di Etruschi</td>
<td>Italy</td>
<td>2.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Cyclo-cross</td>
<td>Superprestige - Hoogstraten</td>
<td>Belgium</td>
<td>C1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Tour of Qatar</td>
<td>Qatar</td>
<td>WT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>South African Road Championships</td>
<td>Western Cape</td>
<td>CN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Cyclo-cross</td>
<td>Superprestige - Middelkerke</td>
<td>Middelkerke</td>
<td>C1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Vuelta Ciclista a la Región de Murcia</td>
<td>Murcia</td>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Clasica de Almeria</td>
<td>Almeria</td>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Trofeo Laigueglia</td>
<td>Liguria</td>
<td>HC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Tour of Oman</td>
<td>Oman</td>
<td>HC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Cyclo-cross</td>
<td>Volta ao Algarve em Bicicleta</td>
<td>Algarve</td>
<td>HC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Vuelta a Andalucia Ruta Ciclista Del Sol</td>
<td>Andalucia</td>
<td>HC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Track</td>
<td>UCI Track World Cup III</td>
<td>Cali</td>
<td>CDM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Tour Cycliste International du Haut Var-matin</td>
<td>Provence</td>
<td>2.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Tour Cycliste International La Providence</td>
<td>Provence</td>
<td>2.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Le Tour de Langkawi</td>
<td>Langkawi</td>
<td>HC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Abu Dhabi Tour</td>
<td>Abu Dhabi</td>
<td>WT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Faun Environnement - Classic de l’Ardeche Rhone Crussol</td>
<td>Ardeche</td>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Omloop Het Nieuwsblad Elite</td>
<td>Belgium</td>
<td>WT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Omloop Het Nieuwsblad Elite Women</td>
<td>Gent</td>
<td>WT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Track</td>
<td>UCI Track World Cup IV</td>
<td>Los Angeles</td>
<td>CDM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Kuurne-Brussel-Kuurne</td>
<td>Kuurne</td>
<td>HC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Omloop van het Hageland - Tielt-Winge</td>
<td>Belgium</td>
<td>WE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Le Samyn</td>
<td>Hainaut</td>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Le Samyn des Dames</td>
<td>Hainaut</td>
<td>WE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Strade Bianche</td>
<td>Tuscany</td>
<td>WT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Strade Bianche Women</td>
<td>Tuscany</td>
<td>WT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Dwars door West-Vlaanderen Johan Museeuw Classic</td>
<td>Flanders</td>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>GP Industria and Artigianato</td>
<td>Larciano</td>
<td>HC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Paris - Nice</td>
<td>France</td>
<td>WT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Semana Ciclista Valenciana</td>
<td>Valencia</td>
<td>WE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Tirreno-Adriatico</td>
<td>Italy</td>
<td>WT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Oceania Championships</td>
<td>Canberra</td>
<td>CC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Ronde van Drenthe</td>
<td>Netherlands</td>
<td>WT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Dwars Achter de Westerveld</td>
<td>Dwingeloo</td>
<td>WE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Tour de Koerse - Danilith Classic</td>
<td>Flanders</td>
<td>HC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Handzame Classic</td>
<td>Flanders</td>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Classic Loire Atlantique</td>
<td>Loire</td>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Milan-San Remo</td>
<td>Italy</td>
<td>WT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Tucson Bicycle Classic</td>
<td>Tucson Arizona</td>
<td>NE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Mountain Bike</td>
<td>Cape Epic</td>
<td>CAPE TOWN</td>
<td>SHC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Trofeo Alfredo Binda Comune di Cittiglio</td>
<td>Italy</td>
<td>WT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Volta Ciclista a Catalunya</td>
<td>Catalunya</td>
<td>WT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Dwars Door Vlaanderen</td>
<td>Flanders</td>
<td>WT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Dwars door Vlaanderen women</td>
<td>Waregem</td>
<td>WE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Settimana Internazionale Copp e Bartali</td>
<td>Emilia-Romagna</td>
<td>2.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>E3 Harelbeke</td>
<td>Flanders</td>
<td>WT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Gent Wevelgem</td>
<td>Flanders</td>
<td>WT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Gent Wevelgem Women</td>
<td>Belgium</td>
<td>WT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Driedaagse De Panne-Koksijde</td>
<td>West Flanders</td>
<td>HC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2017</td>
<td>Road</td>
<td>Pajot Hills Classic</td>
<td>Flemish Brabant</td>
<td>WE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Examples

race_sponsors

Race sponsors set, list, and map collection

SOURCE '0_create_keyspace.cql';

// Create a simple table of races with sponsors in a set collection

DROP TABLE IF EXISTS cycling.race_sponsors;

// START-racesponsorsset
CREATE TABLE IF NOT EXISTS cycling.race_sponsors (
    race_name text PRIMARY KEY,
    sponsorship set{text>
); // END-racesponsorsset

// Insert data into table and set
INSERT INTO cycling.race_sponsors (race_name, sponsorship)
VALUES ('Tour de France',
    ['LCL', 'Carrefour', 'Skoda', 'Vittel', 'Krys']
);

INSERT INTO cycling.race_sponsors (race_name, sponsorship)
VALUES ('Tour de Suisse',...
Examples

```
{ 'BMC', 'Tag Heuer', 'Assos' }
);

INSERT INTO cycling.race_sponsors (race_name, sponsorship)
VALUES ('Criterium du Dauphine',
{ 'LCL', 'Skoda' })
);

INSERT INTO cycling.race_sponsors (race_name, sponsorship)
VALUES ('Giro d'Italia',
{ 'Castelli', 'Tag Heuer', 'Alfa Romeo', 'Unibet', 'Pinarello', 'Selle Italia',
'Rovagnati', 'Scrigno' })
);

// View populated table
SELECT *
FROM cycling.race_sponsors;

// Delete all elements from a set for a race, delete Giro sponsors
// START-deleteset
DELETE sponsorship
FROM cycling.race_sponsors
WHERE race_name = 'Giro d''Italia';
// END-deleteset

// View table again, Giro sponsor column is now null
SELECT *
FROM cycling.race_sponsors;

// Drop table for next example
DROP TABLE IF EXISTS cycling.race_sponsors;

// Create table with partition keys for a list collection
// START-racesponsorlisttbl
CREATE TABLE IF NOT EXISTS cycling.race_sponsors (race_year int,
race_name text,
sponsorship list<text>,
PRIMARY KEY (race_year, race_name))
);
// END-racesponsorlisttbl

// Insert data into the table and list
// START-insertsponsorlist
INSERT INTO cycling.race_sponsors (race_year, race_name, sponsorship)
VALUES (2018, 'Tour de France',
[ 'LCL', 'Carrefour', 'Skoda', 'Vittel', 'Krys' ]) 
);
// END-insertsponsorlist

// View populated table
SELECT *
FROM cycling.race_sponsors;

// Delete data from index position 2, which is sponsor Skoda because 0-based numbering
// START-deletelistelement
DELETE sponsorship[2]
FROM cycling.race_sponsors
WHERE race_year = 2018
AND race_name = 'Tour de France';
```
// View table again, Skoda sponsor column is gone
SELECT *
FROM cycling.race_sponsors;

### race_starts

Frozen list full index example.

```
SOURCE '0_create_keyspace.cql';

DROP TABLE IF EXISTS cycling.race_starts;

// create a table with a frozen list of ints
// START-frozenlist
CREATE TABLE IF NOT EXISTS cycling.race_starts (
    cyclist_name text PRIMARY KEY,
    rnumbers FROZEN<LIST<int>>
);  // END-frozenlist

// View created table
DESCRIBE TABLE cycling.race_starts;

DROP INDEX IF EXISTS cycling.rnumbers_idx;

// Create index
// START-fullindex
CREATE INDEX IF NOT EXISTS rnumbers_idx
ON cycling.race_starts ( FULL(rnumbers) );
// END-fullindex

// Insert data
INSERT INTO cycling.race_starts (cyclist_name, rnumbers)
VALUES ('John DEGENKOLB', [39, 7, 14]);

// Select data with WHERE clause
CAPTURE 'race_starts-queries.results';
// START-selectrnumbers
SELECT * FROM cycling.race_starts
WHERE rnumbers = [39, 7, 14];
// END-selectrnumbers
CAPTURE OFF;
```

<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>

(1 rows)

The **SELECT** statement finds any cyclist who has 39 Pro race wins, 7 Grand Tour starts, and 14 Classic starts.

### race_times

```
SOURCE '0_create_keyspace.cql';
```
Examples

DROP TABLE IF EXISTS cycling.race_times;

// START-TABLE
CREATE TABLE IF NOT EXISTS 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-rename
ALTER TABLE cycling.race_times
RENAME race_date TO date;
// END-rename

ALTER TABLE cycling.race_times
RENAME date TO race_date;

race_winners
Using a user-defined type in a table and queries.

SOURCE '0_create_keyspace.cql';

DROP TABLE IF EXISTS cycling.race_winners;

SOURCE 'fullname-type.cql';

// START-usetype
CREATE TABLE IF NOT EXISTS cycling.race_winners (
    cyclist_name FROZEN<fullname>,
    race_name text,
    race_position int,
    PRIMARY KEY (race_name, race_position)
);
// END-usetype
// END-use-type

// START-insert
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' }
};
// END-insert

INSERT INTO cycling.race_winners {
    race_name, race_position,
cyclist_name
} VALUES {
    'National Championships South Africa WJ-ITT (CN)', 2,
    { firstname:'Lynette', lastname:'BENSON' }
};

INSERT INTO cycling.race_winners {
    race_name, race_position,
cyclist_name
} VALUES {
    'National Championships South Africa WJ-ITT (CN)', 3,
    { firstname:'Anja', lastname:'GERBER' }
};

INSERT INTO cycling.race_winners {
    race_name, race_position,
cyclist_name
} VALUES {
    'National Championships South Africa WJ-ITT (CN)', 4,
    { firstname:'Ame', lastname:'VENTER' }
};

INSERT INTO cycling.race_winners {
    race_name, race_position,
cyclist_name
} VALUES {
    'National Championships South Africa WJ-ITT (CN)', 5,
    { firstname:'Danielle', lastname:'VAN NIEKERK' }
};

// START-comment_hyphens
SELECT * FROM cycling.race_winners; -- End of line comment
// END-comment_hyphens

// START-comment_slashes
SELECT * FROM cycling.race_winners; // End of line comment
// END-comment_slashes

// START-comment_span
/* This is a
   comment that spans multiple
lines */
SELECT * FROM cycling.race_winners;
// END-comment_span

**rank_by_year_and_name**

SOURCE '0_create_keyspace.cql';

DROP TABLE IF EXISTS cycling.rank_by_year_and_name;
// Store race information by year and race name using a COMPOSITE PARTITION KEY

USE cycling;
CREATE TABLE IF NOT EXISTS rank_by_year_and_name (    race_year int,    race_name text,    cyclist_name text,    rank int,    PRIMARY KEY ((race_year, race_name), rank)));

DROP TABLE IF EXISTS cycling.rank_by_year_and_name;

CREATE TABLE IF NOT EXISTS cycling.rank_by_year_and_name (    race_year int,    race_name text,    cyclist_name text,    rank int,    PRIMARY KEY ((race_year, race_name), rank)));

DROP INDEX IF EXISTS cycling.rank_idx;

CREATE INDEX IF NOT EXISTS rank_idx ON cycling.rank_by_year_and_name (rank);

DROP INDEX IF EXISTS cycling.race_year_idx;

CREATE INDEX IF NOT EXISTS race_year_idx ON cycling.rank_by_year_and_name (race_year);

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) VALUES (2014, '4th Tour of Beijing', 'Johan Esteban CHAVES', 3);
Examples

```
INSERT INTO cycling.rank_by_year_and_name (race_year, race_name, cyclist_name, rank)
VALUES (2014, '4th Tour of Beijing', 'Phillippe GILBERT', 1);
INSERT INTO cycling.rank_by_year_and_name (race_year, race_name, cyclist_name, rank)
VALUES (2014, '4th Tour of Beijing', 'Daniel MARTIN', 2);
INSERT INTO cycling.rank_by_year_and_name (race_year, race_name, cyclist_name, rank)
VALUES (2014, '4th Tour of Beijing', 'Johan Esteban CHAVES', 3);

// Show all inserted data
CAPTURE 'select_all_from_rank_by_year_and_name.results';
SELECT *
FROM cycling.rank_by_year_and_name;
CAPTURE OFF;

CAPTURE 'select_all_from_rank_by_year_and_name_Japan_2014.results';
// START-select_with_name_and_year
SELECT *
FROM cycling.rank_by_year_and_name
WHERE race_year = 2014
  AND race_name  = 'Tour of Japan - Stage 4 - Minami > Shinshu';
// END-select_with_name_and_year
CAPTURE OFF;

// START-sepstatementswithand
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;
// END-sepstatementswithand

// START-columnalias
CAPTURE 'select_best_rank_from_rank_by_year_and_name.results';
SELECT
  MIN(rank) AS best_rank,
  cyclist_name
FROM cycling.rank_by_year_and_name
WHERE "race_name" = 'Tour of Japan - Stage 4 - Minami > Shinshu'
  AND race_year = 2014;
// END-columnalias
CAPTURE OFF;

// The following query generates a warning, which is normal
// START-countrows
SELECT COUNT(*)
FROM cycling.rank_by_year_and_name;
// END-countrows

// START-limitrows
SELECT cyclist_name
FROM cycling.rank_by_year_and_name
LIMIT 50000;
// END-limitrows

// Query by partition
CAPTURE 'select_all_from_rank_by_year_and_name_partition_limit.results';
// START-partlimit
SELECT *
FROM cycling.rank_by_year_and_name
PER PARTITION LIMIT 2;
// END-partlimit
CAPTURE OFF;

// Select rank - filter on a clustering column
CAPTURE 'select_all_from_rank_by_year_and_name_rank_1.results';
// START-selectrank
```
Examples

```
SELECT *
FROM cycling.rank_by_year_and_name
WHERE rank = 1;
// END-selectrank
CAPTURE OFF;

CAPTURE 'select_all_from_rank_by_year_and_name_2014.results';
// START-select_with_year
SELECT *
FROM cycling.rank_by_year_and_name
WHERE race_year = 2014;
// END-select_with_year
CAPTURE OFF;
```

team_average
Partition key with two clustering columns. Shows use of aggregate and functions.

```
SOURCE '0_create_keyspace.cql';

DROP TABLE IF EXISTS cycling.team_average;

CREATE TABLE IF NOT EXISTS 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');
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');
```

upcoming_calendar
Find all calendar events for a particular year and month.

```
SOURCE '0_create_keyspace.cql';

DROP TABLE IF EXISTS cycling.upcoming_calendar;

// Find all calendar events for a particular year and month
CREATE TABLE IF NOT EXISTS cycling.upcoming_calendar (year int, month int, events list<text>,
  PRIMARY KEY (year, month)
);
INSERT INTO cycling.upcoming_calendar (year, month, events) VALUES (2015, 06, [ 'Criterium du Dauphine', 'Tour de Suisse']);

INSERT INTO cycling.upcoming_calendar (year, month, events) VALUES (2015, 07, [ 'Tour de France']);

CAPTURE 'select_all_from_upcoming_calendar.results';

// START-select
SELECT * FROM cycling.upcoming_calendar;
// END-select
CAPTURE OFF;

// START-insert
UPDATE cycling.upcoming_calendar SET events = [ 'Criterium du Dauphine', 'Tour de Suisse'] WHERE year = 2015 AND month = 06;
// END-insert

// START-prepend
UPDATE cycling.upcoming_calendar SET events = [ 'Tour de France'] + events WHERE year = 2015 AND month = 06;
// END-prepend

// START-append
UPDATE cycling.upcoming_calendar SET events = events + [ 'Tour de France'] WHERE year = 2017 AND month = 05;
// END-append

// START-position
UPDATE cycling.upcoming_calendar SET events[2] = 'Tour de France' WHERE year = 2015 AND month = 06;
// END-position

// START-set
UPDATE cycling.upcoming_calendar SET events[2] = 'Vuelta Ciclista a Venezuela' WHERE year = 2015 AND month = 06;
// END-set

// START-ttl
UPDATE cycling.upcoming_calendar USING TTL 10000000 SET events[2] = 'Vuelta Ciclista a Venezuela' WHERE year = 2015 AND month = 06;
// END-ttl

// START-addcolumn

Examples

ALTER TABLE cycling.upcoming_calendar
ADD description map<text,text>;
// END-addcolumn

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

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

CAPTURE 'select_description_from_upcoming_calendar.results';
// START-select_description
SELECT description
FROM cycling.upcoming_calendar
WHERE year = 2015 AND month = 6;
// END-select_description

CAPTURE OFF;

// START-value
UPDATE cycling.upcoming_calendar
SET events = events - [ 'Tour de France' ] WHERE year = 2015 AND month = 06;
// END-value

// START-map
UPDATE cycling.upcoming_calendar
SET description = description + {
    'Criterium du Dauphine' : 'Easy race'
} WHERE year = 2015 AND month = 06;
// END-map

Cycling Materialized Views
Create materialized views from cycling tables.

cyclist_by_age-mv
Supports queries by age without an index or allow filtering.

// Source keyspace, tables, and insert data
SOURCE '0_create_keyspace.cql';
// SOURCE 'cyclist_base-table.cql';
// SOURCE 'cyclist_base_ext-table.cql';

// START-dropmv
DROP MATERIALIZED VIEW IF EXISTS cycling.cyclist_by_age;  // END-dropmv

// START-dropOtherMVs
DROP MATERIALIZED VIEW IF EXISTS cycling.cyclist_by_birthday;
DROP MATERIALIZED VIEW IF EXISTS cycling.cyclist_by_age;
DROP MATERIALIZED VIEW IF EXISTS cycling.cyclist_by_birthday_and_age;
DROP MATERIALIZED VIEW IF EXISTS cycling.cyclist_by_country_and_birthday;
DROP MATERIALIZED VIEW IF EXISTS cycling.cyclist_by_birthday_and_age19;
DROP MATERIALIZED VIEW IF EXISTS cycling.cyclist_by_age_birthday_cid;
DROP MATERIALIZED VIEW IF EXISTS cycling.cyclist_by_birthday_Netherlands;
// END-dropOtherMVs

// ORIGINAL MV
// START-age
CREATE MATERIALIZED VIEW IF NOT EXISTS 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';
// END-age

CAPTURE 'select_from_cyclist_by_age_18.results';
// START-select_from_cyclist_by_age_18
SELECT age, name, birthday
FROM cycling.cyclist_by_age
WHERE age = 18;
// END-select_from_cyclist_by_age_18
CAPTURE OFF;

// START-cyclist_by_birthday
CREATE MATERIALIZED VIEW IF NOT EXISTS cycling.cyclist_by_birthday
AS SELECT age, birthday, name, country
FROM cycling.cyclist_base
WHERE birthday IS NOT NULL
AND cid IS NOT NULL
PRIMARY KEY (birthday, cid);
// END-cyclist_by_birthday

// START-cyclist_by_country
CREATE MATERIALIZED VIEW IF NOT EXISTS cycling.cyclist_by_country
AS SELECT age, birthday, name, country
FROM cycling.cyclist_base
WHERE country IS NOT NULL
AND cid IS NOT NULL
PRIMARY KEY (country, cid);
// END-cyclist_by_country

// ADD 2 non-PK columns
// START-plainMVtwoNonPKcols
CREATE MATERIALIZED VIEW IF NOT EXISTS cycling.cyclist_by_birthday_and_age
AS SELECT age, cid, birthday, country, name
FROM cycling.cyclist_base
WHERE age IS NOT NULL
AND birthday IS NOT NULL
AND cid IS NOT NULL
PRIMARY KEY (cid, birthday, age);
// END-plainMVtwoNonPKcols

// ADD 2 non-PK columns and specify one in the WHERE phrase
// START-WhereCountryMVtwoNonPKcols
CREATE MATERIALIZED VIEW IF NOT EXISTS cycling.cyclist_by_country_and_birthday
AS SELECT age, cid, birthday, country, name
FROM cycling.cyclist_base
WHERE country = 'Netherlands'
AND cid IS NOT NULL
PRIMARY KEY (country, cid);
// END-WhereCountryMVtwoNonPKcols
Examples

```sql
FROM cycling.cyclist_base
WHERE birthday IS NOT NULL
  AND cid IS NOT NULL
  AND country = 'Australia'
PRIMARY KEY (cid, country, birthday);
// END-WhereCountryMVtwoNonPKcols

// ADD 2 non-PK columns and specify one in the WHERE phrase
// START-WhereAgeMVtwoNonPKcols
CREATE MATERIALIZED VIEW IF NOT EXISTS cycling.cyclist_by_birthday_and_age19 AS
  SELECT age, cid, birthday, country, name
  FROM cycling.cyclist_base
  WHERE birthday IS NOT NULL
  AND cid IS NOT NULL
  AND age = 19
PRIMARY KEY (cid, birthday, age);
// END-WhereAgeMVtwoNonPKcols

// ADD multiple PK columns, but MV has different order for values
// START-MVmultPKcols
CREATE MATERIALIZED VIEW IF NOT EXISTS cycling.cyclist_by_age_birthday_cid AS
  SELECT age, cid, birthday, country, name
  FROM cycling.cyclist_base_ext
  WHERE age IS NOT NULL
  AND birthday IS NOT NULL
  AND cid IS NOT NULL
PRIMARY KEY (age, birthday, cid);
// END-MVmultPKcols

// START-MVNetherlands
CREATE MATERIALIZED VIEW IF NOT EXISTS cycling.cyclist_by_birthday_Netherlands
  AS SELECT age, birthday, name, country
  FROM cycling.cyclist_base
  WHERE birthday IS NOT NULL
  AND cid IS NOT NULL
  AND country = 'Netherlands'
PRIMARY KEY (birthday, cid);
// END-MVNetherlands

// add a comment to describe the table
// START-comment
ALTER MATERIALIZED VIEW cycling.cyclist_by_age
  WITH comment = 'A most excellent and useful view'
  AND bloom_filter_fp_chance = 0.02;
// END-comment

// alter example
// START-compression
ALTER MATERIALIZED VIEW cycling.cyclist_by_age
  WITH compression = {
    'sstable_compression' : 'DeflateCompressor',
    'chunk_length_kb' : 64
  }
  AND compaction = {
    'class' : 'SizeTieredCompactionStrategy',
    'max_threshold' : 64
};
// END-compression

// alter example
// START-cache
ALTER MATERIALIZED VIEW cycling.cyclist_by_age
  WITH caching = {
    'keys' : 'NONE',
    'rows_per_partition' : '15'
};
```
INSERT INTO cycling.cyclist_base {
    cid, name, age, birthday, country
} VALUES (
    d1aad83b-be60-47a4-bd6e-069b8da0d97b, 'Johannes HEIDER', 27, '1987-09-04', 'Germany');

INSERT INTO cycling.cyclist_base {
    cid, name, age, birthday, country
} VALUES (1c526849-d3a2-42a3-bcf9-7903c80b3d16, 'Kanden GROVES', 19, '1998-12-23', 'Australia');

INSERT INTO cycling.cyclist_base {
    cid, name, age, birthday, country
} VALUES (96c4c40d-58c8-4710-b73f-681e9b1f70ae, 'Benjamin DYBALL', 29, '1989-04-20', 'Australia');

INSERT INTO cycling.cyclist_base {
    cid, name, age, birthday, country
} VALUES (410919ef-bd1b-4efa-8256-b0fd8ab67029, 'Iskandarbek SHODIEV', 19, '1999-01-04', 'Uzbekistan');

INSERT INTO cycling.cyclist_base {
    cid, name, age, birthday, country
} VALUES (d1aad83b-be60-47a4-bd6e-069b8da0d97b, 'Johannes HEIDER', 27, '1987-09-04', 'Germany');
Examples

'Germany'

);  

INSERT INTO cycling.cyclist_base_ext (cid, name, age, birthday, country)
VALUES ('1c526849-d3a2-42a3-bcf9-7903c80b3d16', 'Kanden GROVES', 19, '1998-12-23', 'Australia');

INSERT INTO cycling.cyclist_base_ext (cid, name, age, birthday, country)
VALUES ('96c4c40d-58c8-4710-b73f-681e9b1f70ae', 'Benjamin DYBALL', 29, '1989-04-20', 'Australia');

INSERT INTO cycling.cyclist_base_ext (cid, name, age, birthday, country)
VALUES ('410919ef-bd1b-4efa-8256-b0fd8ab67029', 'Iskandarbek SHODIEV', 19, '1999-01-04', 'Uzbekistan');

// END-cyclist_base_extData

// SELECTS

CAPTURE 'select_cyclist_by_age.results';
// START-selectMVage
SELECT * FROM cycling.cyclist_by_age;
// END-selectMVage
CAPTURE OFF;

CAPTURE 'select_cyclist_by_birthday_and_age.results';
// START-selectMVBdayAge
SELECT * FROM cycling.cyclist_by_birthday_and_age;
// END-selectMVBdayAge
CAPTURE OFF;

CAPTURE 'select_cyclist_by_country_and_birthday.results';
// START-selectMVCountryBday
SELECT * FROM cycling.cyclist_by_country_and_birthday;
// END-selectMVCountryBday
CAPTURE OFF;

CAPTURE 'select_cyclist_by_birthday_and_age19.results';
// START-selectMV19BdayAge
SELECT * FROM cycling.cyclist_by_birthday_and_age19;
// END-selectMV19BdayAge
CAPTURE OFF;

CAPTURE OFF;

CAPTURE OFF;

CAPTURE OFF;

// START-selectMV19BdayCID
SELECT * FROM cycling.cyclist_by_age_birthday_cid
WHERE age = 19;
Cycling queries
Querying the cycling keyspace tables.

Aggregating cycling data
An aggregate function operates on a set of rows. The aggregate function receives values for each row and returns one value for all of the rows.

This section shows queries that use the aggregate functions. Each query is followed by the returned output.

This query uses the AVG function to return the average team racing time:

```cql
SELECT AVG(cyclist_time_sec) AS Average FROM cycling.team_average WHERE team_name = 'UnitedHealthCare Pro Cycling Womens Team';
```

Output:

```
average
-------
11474
```
Examples

This query uses the `COUNT` function to return the number of cyclists in the specified team:

```sql
SELECT COUNT(cyclist_name) AS Row_Count
FROM cycling.team_average
WHERE team_name = 'UnitedHealthCare Pro Cycling Womens Team';
```

Output:

```
row_count
---------
 3
```

This query uses the `MIN` function to return the fastest team racing time:

```sql
SELECT MIN(cyclist_time_sec) AS Fastest
FROM cycling.team_average
WHERE team_name = 'UnitedHealthCare Pro Cycling Womens Team';
```

Output:

```
fastest
-------
11449
```

This query uses the `MAX` function to return the slowest team racing time:

```sql
SELECT MAX(cyclist_time_sec) AS Slowest
FROM cycling.team_average
WHERE team_name = 'UnitedHealthCare Pro Cycling Womens Team';
```

Output:

```
slowest
-------
11490
```

This query uses the `SUM` function to return the total team racing times:

```sql
SELECT SUM(cyclist_time_sec) AS Total_Time
FROM cycling.team_average
WHERE team_name = 'UnitedHealthCare Pro Cycling Womens Team';
```

Output:

```
total_time
----------
34424
```
Cycling user-defined functions

Sample user-defined functions and aggregates.

Requires setting the `enable_user_defined_functions` and `enable_scripted_user_defined_functions` to true in the `cassandra.yaml`.

Cycling user-defined aggregate `team_average`

Creates a Java user-defined aggregate function that calculates the average of a target column.

```
SOURCE '0_create_keyspace.cql';

// START-dropagg
DROP AGGREGATE IF EXISTS cycling.avgState;
// END-dropagg

DROP TABLE IF EXISTS cycling.test_avg;
// START-state
CREATE OR REPLACE FUNCTION cycling.avgState {
  state tuple<int, bigint>,
  val int
}
  CALLED ON NULL INPUT
  RETURNS tuple<int, bigint>
  LANGUAGE java AS
$$
  if (val != null) {
    state.setInt(0, state.getInt(0) + 1);
    state.setLong(1, state.getLong(1) + val.intValue());
  }
  return state;
$$
;
// END-state

// START-test
CREATE TABLE IF NOT EXISTS 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);

INSERT INTO cycling.test_avg {
  id, state, val
} VALUES (2, (79, 10000), 9999);

SELECT state, avgstate(state, val), val
FROM cycling.test_avg;
// END-test

// START-final
CREATE OR REPLACE FUNCTION cycling.avgFinal {
  state tuple<int,bigint>
```
Examples

```sql
flog
User-defined function.

SOURCE '0_create_keyspace.cql';

// START-dropflog
DROP FUNCTION IF EXISTS cycling.fLog;
// END-dropflog

// START-flog
CREATE OR REPLACE FUNCTION cycling.fLog (input double) CALLED ON NULL INPUT RETURNS double LANGUAGE java AS $$
    return Double.valueOf(Math.log(input.doubleValue()));
$$
;
// END-flog

left
Create a user-define function that trims row data from the left. Requires that both enable_user_defined_functions and enable_scripted_user_defined_functions are enabled.

CREATE OR REPLACE FUNCTION cycling.left (column text, num int)
RETURNS NULL ON NULL INPUT
RETURNS text LANGUAGE javascript AS $$
    column.substring(0, num)
$$
;
Cycling search index
Search index examples using the cycling tables.

comments search index
Modifying the configuration and schema on the comments table search index.

Depends on the comments table.

// SOURCE 'comments-table.cql';

// START-create
CREATE SEARCH INDEX IF NOT EXISTS
ON cycling.comments
WITH COLUMNS record_id { excluded:true }, * {excluded:false};
// END-create

// **CHANGING the CONFIG**

// START-directoryFactory
ALTER SEARCH INDEX CONFIG
ON cycling.comments
SET directoryFactory = 'encrypted';
// END-directoryFactory

// START-active
RELOAD SEARCH INDEX
ON cycling.comments;
// END-active

// START-rebuild
RELOAD SEARCH INDEX
ON cycling.comments;
REBUILD SEARCH INDEX
ON cycling.comments;
// END-rebuild

// START-cache
ALTER SEARCH INDEX CONFIG
ON cycling.comments
SET autoCommitTime = 10000;
// END-cache

// START-requesth
ALTER SEARCH INDEX CONFIG
ON cycling.comments
ADD requestHandler[@name='/elevate',@class='solr.SearchHandler', @startup='lazy']
WITH $$$"defaults":[{"echoParams":"explicit"}],"last-components":["elevator"]$$$;
// END-requesth

// Policy to merge large segments due to deletes
// START-delete
ALTER SEARCH INDEX CONFIG
ON cycling.comments
SET
  indexConfig.mergePolicyFactory[@class='org.apache.solr.index.AutoExpungeDeletesTieredMergePolicyFactory'].bool
  = true;

ALTER SEARCH INDEX CONFIG
ON cycling.comments
Examples

SET
indexConfig.mergePolicyFactory[@class='org.apache.solr.index.AutoExpungeDeletesTieredMergePolicyFactory'].int[@name='maxMergedSegmentMB'] = 1005;

ALTER SEARCH INDEX CONFIG
ON cycling.comments
SET
indexConfig.mergePolicyFactory[@class='org.apache.solr.index.AutoExpungeDeletesTieredMergePolicyFactory'].int[@name='forceMergeDeletesPctAllowed'] = 25;

// END-delete

// START-commit
COMMIT SEARCH INDEX
ON cycling.comments;
// END-commit

// **CHANGING the SCHEMA**

// START-add_string_field
ALTER SEARCH INDEX SCHEMA
ON cycling.comments
ADD fields.field[@name='fieldname', @type='StrField', @multiValued = 'false', @indexed='true'];
// END-add_string_field

// START-column_name
ALTER SEARCH INDEX SCHEMA
ON cycling.comments
ADD FIELD record_id;
// END-column_name

// START-rename
ALTER SEARCH INDEX SCHEMA
ON cycling.comments
SET field[@name='fieldname']@name = 'anotherFieldName';
// END-rename

// START-type
ALTER SEARCH INDEX SCHEMA
ON cycling.comments
SET field[@name='fieldname']@type = 'UUIDField';
// END-type

// START-drop_another
ALTER SEARCH INDEX SCHEMA
ON cycling.comments
DROP field anotherFieldName;
// END-drop_another

// **CHANGING the CONFIG**

// START-autocommit
ALTER SEARCH INDEX CONFIG
ON cycling.comments
SET autoCommitTime = 1000;
// END-autocommit

// START-add_text_field
ALTER SEARCH INDEX SCHEMA
ON cycling.comments
SET types.fieldType[@name='TextField']@class='org.apache.solr.schema.TextField';
ALTER SEARCH INDEX SCHEMA
ON cycling.comments
SET fields.field[@name='comment']@type='TextField';
// END-add_text_field
// START-drop_text_field
ALTER SEARCH INDEX SCHEMA
ON cycling.comments
DROP field comment;

ALTER SEARCH INDEX SCHEMA
ON cycling.comments
DROP types.fieldType[@name='TextField'];
// END-drop_text_field

// START-add_dynamic_text_field
ALTER SEARCH INDEX SCHEMA
ON cycling.comments
ADD types.fieldType[@class='org.apache.solr.schema.TextField', @name='TextField']
WITH '{"analyzer":{"class":"org.apache.lucene.analysis.standard.StandardAnalyzer"}}';

ALTER SEARCH INDEX SCHEMA
ON cycling.comments
ADD dynamicField[@name='*fieldname', @type='TextField'];
// END-add_dynamic_text_field

RELOAD SEARCH INDEX
ON cycling.comments;
REBUILD SEARCH INDEX
ON cycling.comments;

Index
Experimental secondary index.

Index on cyclist_name
Create an experimental secondary index.

USE cycling;
DROP INDEX IF EXISTS fn_prefix;
DROP INDEX IF EXISTS fn_contains;
DROP INDEX IF EXISTS fn_sparse;
DROP INDEX IF EXISTS fn_notcasesensitive;
DROP INDEX IF EXISTS stdanalyzer_idx;

// START-prefix
CREATE CUSTOM INDEX IF NOT EXISTS fn_prefix
ON cycling.comments (commenter)
USING 'org.apache.cassandra.index.sasi.SASIIndex';
// END-prefix

// START-contains
CREATE CUSTOM INDEX IF NOT EXISTS fn_contains
ON cycling.comments (comment)
USING 'org.apache.cassandra.index.sasi.SASIIndex'
WITH OPTIONS = {
  'mode' : 'CONTAINS'
};
// END-contains

// START-sparse
CREATE CUSTOM INDEX IF NOT EXISTS fn_sparse
ON cycling.comments (record_id)
USING 'org.apache.cassandra.index.sasi.SASIIndex'
WITH OPTIONS = {
  'mode' : 'SPARSE'
};
//
// END-sparse

// START-case
CREATE CUSTOM INDEX IF NOT EXISTS fn_notcasesensitive
ON cycling.comments (comment)
USING 'org.apache.cassandra.index.sasi.SASIIndex'
WITH OPTIONS = {
    'analyzer_class' : 'org.apache.cassandra.index.sasi.analyzer.NonTokenizingAnalyzer',
    'case_sensitive' : 'false'
};
// END-case

// START-stdanalyzer
CREATE CUSTOM INDEX IF NOT EXISTS stdanalyzer_idx
ON cycling.comments (comment)
USING 'org.apache.cassandra.index.sasi.SASIIndex'
WITH OPTIONS = {
    'mode' : 'CONTAINS',
    'analyzer_class' : 'org.apache.cassandra.index.sasi.analyzer.StandardAnalyzer',
    'analyzed' : 'true',
    'tokenization_skip_stop_words' : 'and, the, or',
    'tokenization_enable_stemming' : 'true',
    'tokenization_normalize_lowercase' : 'true',
    'tokenization_locale' : 'en'
};
// END-stdanalyzer

// START-dprefix
DROP INDEX IF EXISTS fn_prefix;
// END-dprefix

### Cycling access control

Roles and access control for the cycling keyspace with different authentication and authorization models.

### Cycling internal

Internal authentication with internally managed access control.

```cql
SOURCE 'flog-function.cql';

// when authentication and authorization is enabled

// internal roles used as user accounts
DROP ROLE IF EXISTS sys_admin;
DROP ROLE IF EXISTS team_manager;
DROP ROLE IF EXISTS sandy;
DROP ROLE IF EXISTS role_admin;
CREATE ROLE IF NOT EXISTS sys_admin WITH SUPERUSER = true; // gives role access to everything
CREATE ROLE IF NOT EXISTS team_manager WITH PASSWORD = 'RockIt4Us!';
CREATE ROLE IF NOT EXISTS sandy WITH PASSWORD = 'password' AND LOGIN = true;
CREATE ROLE IF NOT EXISTS role_admin WITH PASSWORD = 'changeme' AND LOGIN = true;

// data resource examples
GRANT MODIFY ON KEYSSPACE cycling TO team_manager;
GRANT AUTHORIZE ON ALL KEYSACES TO sys_admin;

// internal role permission collects as DB object
GRANT sys_admin TO team_manager;
GRANT team_manager TO sandy;
GRANT SELECT ON ALL KEYSACES TO team_manager;
GRANT EXECUTE ON FUNCTION cycling.fLog(double) TO team_manager;

// removing access
```

CQL for DataStax Enterprise 6.7 Latest DSE version Latest 6.7 patch: 6.7.7
REVOKE SELECT ON ALL KEYSPACES FROM team_manager;
REVOKE EXECUTE ON FUNCTION cycling.fLog(double) FROM team_manager;
REVOKE sys_admin FROM team_manager;
REVOKE team_manager FROM sandy;

// role management examples
GRANT DESCRIBE, ALTER ON ALL ROLES TO sys_admin;

LIST ROLES;
LIST ROLES OF sandy;
LIST ALL PERMISSIONS OF sandy;
LIST ALL PERMISSIONS ON cycling.cyclist_name OF team_manager;

// START-CHANGE_PW
ALTER ROLE sandy WITH PASSWORD = 'bestTeam';
// END-CHANGE_PW

Search index examples

Use these step-by-step tutorials with sample keyspaces, tables, and data to learn how to setup and use DSE Search index functionality.

Useful external resources:

• Tutorial how to index and query geospatial Polygons and MultiPolygons.
• Docker container for running Silk on DSE Search.

Creating a demo keyspace for tutorials

Step-by-step instructions to create a keyspace for tutorials found in this section.

1. Get a list of datacenter names (DC) in the cluster.

   $ dsetool status

   The header line contains the datacenter name (DC: datacenter_name) and the type of workload.

   DC: Cassandra  Workload: Cassandra  Graph: no
   ================================================
   ...
   DC: Solr  Workload: Search  Graph: no
   ================================================
   ...

2. Start a cqlsh session:

   cqlsh

   To connect cqlsh to a remote node use the host switch with the hostname or IP address.

3. Create a demo keyspace with a replication factor of 1 in each datacenter.

   In multi-datacenter environments use NetworkTopologyStrategy and set the replication factor for each datacenter to at least one.

   CREATE KEYSPACE demo
   WITH replication = {
      'class': 'NetworkTopologyStrategy',
      ...
Examples

```javascript
'Cassandra': '1',
'Solr': '1'};
```

Datacenter names are case-sensitive. Exit cqlsh and ensure that the name exactly matches the DC name from the dsetool status output.

**Multi-faceted search using healthcare data**

This quick start example provides an overview of creating and altering search indexes using CQL index management commands.

**Prerequisites:**

1. Create a demo keyspace with a replication factor of at least 1 in the search datacenter, see [Creating a demo keyspace for tutorials](#).

2. Download the `health_data.csv` onto a search node.

1. Launch cqlsh on a search node:

   a. Determine which nodes in the cluster are running a search workload:

   ```
   $ dsetool status
   ```

DSE Search operations are available only on search-enabled nodes. DataStax recommends single workload datacenters.

The following example shows a development environment where all nodes in the cluster are in the same physical location, on the same rack, and the nodes have been separated into datacenters based on their workloads:

<table>
<thead>
<tr>
<th>DC: Main</th>
<th>Workload: Cassandra</th>
<th>Graph: no</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>Health [0,1]</td>
<td>Owns</td>
</tr>
<tr>
<td></td>
<td>10.10.10.111</td>
<td>15.51 MiB</td>
</tr>
<tr>
<td></td>
<td>10.10.10.113</td>
<td>19.51 MiB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DC: Search</th>
<th>Workload: Search</th>
<th>Graph: no</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>Health [0,1]</td>
<td>Owns</td>
</tr>
<tr>
<td></td>
<td>10.10.10.108</td>
<td>18.13 MiB</td>
</tr>
</tbody>
</table>
b. Launch a `cqlsh` session on a search node from the directory that contains the `health_data.csv`:

```
$ cd ~ && cqlsh -k demo
```

A CQL session starts using the **demo keyspace**.

```
Connected to cluster1 at 10.10.10.108:9042.
[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:demo>
```

The active keyspace name appears in the `cqlsh` prompt.

2. Set up the `health_data` table with data:
   a. Create the table:

   ```
   USE healthcare;
   // START-create-table
   CREATE TABLE IF NOT EXISTS healthcare.health_data {
   "id" INT,
   "num_smokers" INT,
   "age" INT,
   "age_unit" VARCHAR,
   "age_months" INT,
   "major_medical_coverage" VARCHAR,
   "dental_coverage" VARCHAR,
   "routine_medical_coverage" VARCHAR,
   "employer_paid_plan" VARCHAR,
   "secondary_smoke" VARCHAR,
   "county" VARCHAR,
   "screening_month" VARCHAR,
   "pets" VARCHAR,
   "asthma" VARCHAR,
   "bronchitis" VARCHAR,
   "goiter" VARCHAR,
   "hay_fever" VARCHAR,
   "thyroid_disease" VARCHAR,
   "chronic_bronchitis" VARCHAR,
   "diagnosed_asthma" VARCHAR,
   "diagnosed_cataracts" VARCHAR,
   "diagnosed_emphysema" VARCHAR,
   "diagnosed_goiter" VARCHAR,
   "diagnosed_gout" VARCHAR,
   "diagnosed_hay_fever" VARCHAR,
   "diagnosed_lupus" VARCHAR,
   "diagnosed_other_cancer" VARCHAR,
   "diagnosed_skin_cancer" VARCHAR,
   "diagnosed_stroke" VARCHAR,
   "diagnosed_thyroid_disease" VARCHAR,
   "diagnosed_congestive_heart_failure" VARCHAR,
   "ethnicity" VARCHAR,
   "exam_status" VARCHAR,
   "family_sequence" INT,
   "family_size" INT,
   "fips" VARCHAR,
   }
   ```
"grade_completed" VARCHAR,
"household_size" INT,
"health_status" VARCHAR,
"marital_status" VARCHAR,
"bird" VARCHAR,
"cat" VARCHAR,
"dog" VARCHAR,
"fish" VARCHAR,
"other_pet" VARCHAR,
"race" VARCHAR,
"race_ethnicity" VARCHAR,
"gender" VARCHAR,
"birthplace" VARCHAR,
"annual_income_20000" VARCHAR,
"income_group" INT,
"monthly_income_total" INT,
PRIMARY KEY ("id", "age")
WITH gc_grace_seconds = 0;
// END-create-table

After loading data that contains null values, temporarily set the grace period to zero to clean up tombstones.

b. Load data from the CSV file:

COPY health_data (  
"id",
"num_smokers",
"age",
"age_unit",
"age_months",
"major_medical_coverage",
"dental_coverage",
"routine_medical_coverage",
"employer_paid_plan",
"secondary_smoke",
"county",
"screening_month",
"pets",
"asthma",
"bronchitis",
"goiter",
"hay_fever",
"thyroid_disease",
"chronic_bronchitis",
"diagnosed_asthma",
"diagnosed_catarracts",
"diagnosed_emphysema",
"diagnosed_goiter",
"diagnosed_gout",
"diagnosed_hay_fever",
"diagnosed_lupus",
"diagnosed_other_cancer",
"diagnosed_skin_cancer",
"diagnosed_stroke",
"diagnosed_thyroid_disease",
"diagnosed_congestive_heart_failure",
"ethnicity",
"cat" VARCHAR,
"family_sequence",
"family_size",
"fips",
"grade_completed",
"household_size",
"health_status",
"marital_status",
"bird",
"cat",
"dog",
"fish",
"other_pet",
"race",
"race_ethnicity",
"gender",
"birthplace",
"annual_income_20000",
"income_group",
"monthly_income_total")
FROM 'health_data.csv';

If `health_data.csv` is not in the directory where you launch cqlsh, specify the full path to the file.

The script loads 20050 with no rows skipped.

Using 1 child processes

Starting copy of demo.health_data with columns [id, num_smokers, age, age_unit, age_months, major_medical_coverage, dental_coverage, routine_medical_coverage, employer_paid_plan, secondary_smoke, county, screening_month, pets, asthma, bronchitis, goiter, hay_fever, thyroid_disease, chronic_bronchitis, diagnosed_asthma, diagnosed_cataracts, diagnosed_emphysema, diagnosed_goiter, diagnosed_gout, diagnosed_hay_fever, diagnosed_lupus, diagnosed_other_cancer, diagnosed_skin_cancer, diagnosed_stroke, diagnosed_thyroid_disease, diagnosed_congestive_heart_failure, ethnicity, exam_status, family_sequence, family_size, fips, grade_completed, household_size, health_status, marital_status, bird, cat, dog, fish, other_pet, race, race_ethnicity, gender, birthplace, annual_income_20000, income_group, monthly_income_total].
Processed: 20050 rows; Rate:  4782 rows/s; Avg. rate:  4560 rows/s
20050 rows imported from 1 files in 4.397 seconds (0 skipped).

c. Verify the number of records:

```
SELECT COUNT(*) FROM demo.health_data;
```

```
count
-------
20050

(1 row)
```

Warnings :
Aggregation query used without partition key

3. Create the search index:

```
CREATE SEARCH INDEX ON demo.health_data
WITH COLUMNS * {excluded:false}, age_months, monthly_income_total {excluded: true};
```

- Only columns identified in the COLUMNS options are included. All columns are included when this option is omitted.
4. Display the schema:

```
DESCRIBE ACTIVE SEARCH INDEX SCHEMA ON demo.health_data;
```

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<schema name="autoSolrSchema" version="1.5">
  <types>
    <fieldType class="org.apache.solr.schema.StrField" name="StrField"/>
    <fieldType class="org.apache.solr.schema.TrieIntField" name="TrieIntField"/>
  </types>
  <fields>
    <field indexed="true" multiValued="false" name="grade_completed" stored="true" type="StrField"/>
    <field indexed="true" multiValued="false" name="diagnosed_thyroid_disease" stored="true" type="StrField"/>
    <field indexed="true" multiValued="false" name="pets" stored="true" type="StrField"/>
    <field indexed="true" multiValued="false" name="secondary_smoke" stored="true" type="StrField"/>
    <field indexed="true" multiValued="false" name="diagnosed_lupus" stored="true" type="StrField"/>
    <field indexed="true" multiValued="false" name="gender" stored="true" type="StrField"/>
    <field indexed="true" multiValued="false" name="birthplace" stored="true" type="StrField"/>
    <field indexed="true" multiValued="false" name="income_group" stored="true" type="TrieIntField"/>
    <field indexed="true" multiValued="false" name="marital_status" stored="true" type="StrField"/>
    <field indexed="true" multiValued="false" name="age_months" stored="true" type="TrieIntField"/>
    <field indexed="true" multiValued="false" name="bird" stored="true" type="StrField"/>
    <field indexed="true" multiValued="false" name="hay_fever" stored="true" type="StrField"/>
    <field indexed="true" multiValued="false" name="diagnosed_hay_fever" stored="true" type="StrField"/>
    <field indexed="true" multiValued="false" name="routine_medical_coverage" stored="true" type="StrField"/>
    <field indexed="true" multiValued="false" name="annual_income_20000" stored="true" type="StrField"/>
    <field indexed="true" multiValued="false" name="exam_status" stored="true" type="StrField"/>
    <field indexed="true" multiValued="false" name="other_pet" stored="true" type="StrField"/>
    <field indexed="true" multiValued="false" name="diagnosed_stroke" stored="true" type="StrField"/>
    <field indexed="true" multiValued="false" name="employer_paid_plan" stored="true" type="StrField"/>
    <field indexed="true" multiValued="false" name="family_sequence" stored="true" type="TrieIntField"/>
    <field indexed="true" multiValued="false" name="diagnosed_cataracts" stored="true" type="StrField"/>
    <field indexed="true" multiValued="false" name="major_medical_coverage" stored="true" type="StrField"/>
    <field indexed="true" multiValued="false" name="diagnosed_gout" stored="true" type="StrField"/>
    <field indexed="true" multiValued="false" name="age_unit" stored="true" type="StrField"/>
    <field indexed="true" multiValued="false" name="goiter" stored="true" type="StrField"/>
    <field indexed="true" multiValued="false" name="chronic_bronchitis" stored="true" type="StrField"/>
  </fields>
</schema>
```
Quick Start for CQL index management

This quick start example provides an overview of creating and altering search indexes using CQL index management commands.
Creating a search index

1. Launch cqlsh and create a tutorial keyspace on a solr node:

   cqlsh

2. Set up the table schema and create a default index:

3. Create the keyspace, create the table, and then create the search index on the users table from the KillrVideo application:

   ```
   CREATE KEYSPACE demo WITH
   replication = {
      'class': 'SimpleStrategy',
      'replication_factor': 1};
   
   CREATE TABLE demo.users {
      userid uuid,
      firstname text,
      lastname text,
      email text,
      created_date timestamp,
      PRIMARY KEY (userid));
   
   CREATE SEARCH INDEX ON demo.users;
   ```

   A new search index is generated on the table. Existing data is reindexed.

4. Use the CQL shell DESCRIBE SEARCH INDEX SCHEMA View the pending search index schema

   The generated schema looks like this:

   ```
   <schema name="autoSolrSchema" version="1.5">
   <types>
   <fieldType class="org.apache.solr.schema.TextField" name="TextField">
   <analyzer>
   <tokenizer class="solr.StandardTokenizerFactory"/>
   <filter class="solr.LowerCaseFilterFactory"/>
   </analyzer>
   </fieldType>
   <fieldType class="org.apache.solr.schema.UUIDField" name="UUIDField"/>
   <fieldType class="org.apache.solr.schema.TrieDateField" name="TrieDateField"/>
   </types>
   <fields>
   <field indexed="true" multiValued="false" name="firstname" stored="true" type="TextField"/>
   <field docValues="true" indexed="true" multiValued="false" name="userid" stored="true" type="UUIDField"/>
   <field indexed="true" multiValued="false" name="email" stored="true" type="TextField"/>
   <field indexed="true" multiValued="false" name="lastname" stored="true" type="TextField"/>
   <field docValues="true" indexed="true" multiValued="false" name="created_date" stored="true" type="TrieDateField"/>
   </fields>
   <uniqueKey>userid</uniqueKey>
   ```
5. To increase tolerance of non-ASCII characters in the name field, add a new fieldType to the schema with this `ALTER SEARCH INDEX SCHEMA` statement:

```
ALTER SEARCH INDEX SCHEMA ON demo.users
  ADD types.fieldType [  
    @name='TextField_intl',  
    @class='org.apache.solr.schema.TextField' ]  
WITH $$
  "analyzer": [ {  
    "type": "index",  
    "tokenizer": { "class": "solr.StandardTokenizerFactory" },  
    "filter": [ { "class": "solr.LowerCaseFilterFactory" },  
      { "class": "solr.ASCIIFoldingFilterFactory" }  
    ]  
  },  
  {  
    "type": "search",  
    "tokenizer": { "class": "solr.StandardTokenizerFactory" },  
    "filter": [  
      { "class": "solr.LowerCaseFilterFactory" },  
      { "class": "solr.ASCIIFoldingFilterFactory" }  
    ]  
  }  
}$$;
```

The dollar signs ($$) syntax in the `ALTER SEARCH INDEX SCHEMA` example are dollar quotes to escape a single quotation mark, see Escaping single quotation marks. This new fieldType has separate index and search analysis phases:

```
<fieldType class="org.apache.solr.schema.TextField" name="TextField_intl">
  <analyzer type="index">
    <filter class="solr.LowerCaseFilterFactory"/>
    <tokenizer class="solr.StandardTokenizerFactory"/>
  </analyzer>
  <analyzer type="search">
    <filter class="solr.LowerCaseFilterFactory"/>
    <tokenizer class="solr.StandardTokenizerFactory"/>
  </analyzer>
</fieldType>
```

6. Change the type on the firstname and lastname fields:

```
ALTER SEARCH INDEX SCHEMA ON demo.users
  SET field[@name='firstname']@type = 'TextField_intl';
```

```
ALTER SEARCH INDEX SCHEMA ON demo.users
  SET field[@name='lastname']@type = 'TextField_intl';
```

7. In contrast to the dsetool search index management commands, all changes made with `ALTER SEARCH INDEX` affect only the pending resources for the search index. To use the changes:
Examples

a. Reload the index.

```
RELOAD SEARCH INDEX ON demo.users;
```

b. If there is existing data in the index, rebuild the index.

```
REBUILD SEARCH INDEX ON demo.users;
```

**Term and phrase searches using the wikipedia demo**

The Wikipedia demo scripts automatically download 3,000+ Wikipedia articles, create a CQL keyspace and table, insert the articles, and create a search index on both the title and body columns.

**Prerequisites:** The demo scripts connect to the localhost on the Solr port. Ensure that the Solr interface and port 127.0.0.1:8983 are accessible.

1. Start DataStax Enterprise as a search node.

2. Go to `installation_directory/demos/wikipedia`.

3. Run the script to add the wikipedia schema:

```
$ ./1-add-schema.sh
```

   This script creates the `wiki` keyspace with a single table `solr`.

4. To use the demo in a cluster that has more than one node, change the keyspace replication from SimpleStrategy to NetworkTopologyStrategy, and set the factor to 1 in each datacenter:

```
$ cqlsh -e 'ALTER KEYSPACE wiki WITH replication = {'class':
'NetworkTopologyStrategy', 'Cassandra': 1, 'Solr': 1};
```

   In this example, the cluster has two datacenters, Cassandra and Solr. Datacenter names are case-sensitive.

5. Load the data and index the table using the second script (`2-index.sh`).

```
$ ./2-index.sh --wikifile wikipedia-sample.bz2
```

   3,000 articles are loaded into the `solr` table and then indexed.

   ```
   Start indexing wikipedia...
   ------------> config properties:
   docs.file = wikipedia-sample.bz2
   keep.image.only.docs = false
   ----------------------
   Indexed 1000
   Indexed 2000
   Indexed 3000
   Finished
   ```
Visit http://localhost:8983/demos/wikipedia/ to see data

6. Verify that the data was successfully loaded into the keyspace/table:

$ cqlsh -e 'DESC KEYSPACE wiki; SELECT count(*) FROM wiki.solr;'

The results show the details of the keyspace, table schema, index settings, and number of articles.

CREATE KEYSPACE wiki WITH replication = {'class': 'SimpleStrategy', 'replication_factor': '1'} AND durable_writes = true;

CREATE TABLE wiki.solr (
    id text PRIMARY KEY,
    body text,
    date text,
    solr_query text,
    title text
) WITH 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 = 86400
    AND max_index_interval = 2048
    AND memtable_flush_period_in_ms = 0
    AND min_index_interval = 128
    AND speculative_retry = '99PERCENTILE';

CREATE CUSTOM INDEX wiki_solr_solr_query_index ON wiki.solr (solr_query) USING
    'com.datastax.bdp.search.solr.Cql3SolrSecondaryIndex';

<table>
<thead>
<tr>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>3579</td>
</tr>
</tbody>
</table>

(1 rows)

Warnings :
Aggregation query used without partition key

7. Start cqlsh using the wiki keyspace.

$ cqlsh -k wiki

CQL shell session starts on the localhost in the wiki keyspace.

Connected to pw-search 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.
8. Disable paging, for faster query results on small data sets:

```
PAGING off
```

Paging is turned off only for the session. Paging is enabled after a restart. Use a cqlshrc file to change the default startup parameters for cqlsh.

Disabling Query paging.

9. Display the solr table search index schema:

```
DESCRIBE ACTIVE SEARCH INDEX SCHEMA ON solr;
```

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<schema name="autoSolrSchema" version="1.5">
  <types>
    <fieldType class="org.apache.solr.schema.TextField" name="TextField">
      <analyzer>
        <tokenizer class="solr.WikipediaTokenizerFactory"/>
      </analyzer>
    </fieldType>
    <fieldType class="org.apache.solr.schema.StrField" name="StrField"/>
  </types>
  <fields>
    <field indexed="true" multiValued="false" name="body" stored="true" type="TextField"/>
    <field indexed="true" multiValued="false" name="title" stored="true" type="TextField"/>
    <field docValues="true" indexed="true" multiValued="false" name="id" stored="true" type="StrField"/>
    <field docValues="true" indexed="true" multiValued="false" name="date" stored="true" type="StrField"/>
  </fields>
  <uniqueKey>id</uniqueKey>
</schema>
```

10. Execute queries against the table using the index:

- Return the titles of articles that contain the word national:

```
SELECT title FROM solr WHERE solr_query='title:national';
```

Seven records are returned.

```
title
Bolivia national football team 1999
Bolivia national football team 2000
Kenya national under-20 football team
Bolivia national football team 2001
Bolivia national football team 2002
Israel men's national inline hockey team
List of French born footballers who have played for other national teams
```
Using secure cluster

You can run the Term and phrase searches using the wikipedia demo on a secure cluster.

Kerberos options

- `-a` enable Kerberos authentication
- `-h hostname` server hostname (not required if server hostname resolution is correctly set up)

HTTP basic authentication

Use with DseAuthenticator:.
- `-u` username
- `-p` password

You can use this option to run the shell scripts:

```
./1-add-schema.sh -u root -p password && ./2-index.sh -u root -p password
```

SSL options

- `-e cert` enable HTTPS for client to node encryption, using `cert` certificate file
- `-k` disable strict hostname checking for SSL certificates

Indexing and querying polygons

Using United States of America state data, this tutorial demonstrates how to index and query geospatial shapes, such as Polygons and MultiPolygons.

Geospatial data is stored in the database in WKT (Well Known Text) format. To support Polygons and MultiPolygons fields, set the field type in the table index schema to `solr.SpatialRecursivePrefixTreeFieldType`.

`SpatialRecursivePrefixTreeFieldType` supports multiValued spatial data. Most states are describable as Polygons, but others such as Hawaii and Alaska are MultiPolygons.

Polygonal searches use the following geospatial predicates:

- **Intersects**: Matches if the indexed value **overlaps** the search criteria.
- **IsWithin**: Matches if the indexed value **completely encapsulates** the search criteria.
- **IsDisjointTo**: Matches if the index value **does not overlap or touch** the search criteria.


1.