Cassandra Query Language (CQL) : Advanced

Database Roles

Database Roles

CQL uses database roles to represent users and group of users. Syntactically, a role is defined by:

```
role_name ::= identifier | string
```

CREATE ROLE

Creating a role uses the `CREATE ROLE` statement:

```
create_role_statement ::= CREATE ROLE [ IF NOT EXISTS ] role_name
                          [ WITH role_options# ]
role_options ::= role_option ( AND role_option)*
role_option ::= PASSWORD '=' string
                | HASHED PASSWORD '=' string
                | LOGIN '=' boolean
                | SUPERUSER '=' boolean
                | OPTIONS '=' map_literal
                | ACCESS TO DATACENTERS set_literal
                | ACCESS TO ALL DATACENTERS
                | ACCESS FROM CIDRS set_literal
                | ACCESS FROM ALL CIDRS
```

For instance:

```
CREATE ROLE new_role;
CREATE ROLE alice WITH PASSWORD = 'password_a' AND LOGIN = true;
CREATE ROLE alice WITH HASHED PASSWORD =
'$2a$10$JSJEMFm6GeaW9XxT5JIheuEtPvat6i7uKbnTcxX3c1wshIIsGyUtG' AND LOGIN = true;
CREATE ROLE bob WITH PASSWORD = 'password_b' AND LOGIN = true AND SUPERUSER = true;
CREATE ROLE carlos WITH OPTIONS = { 'custom_option1' : 'option1_value', 'custom_option2'
: 99 };
CREATE ROLE alice WITH PASSWORD = 'password_a' AND LOGIN = true AND ACCESS TO DATACENTERS
{'DC1', 'DC3'};
CREATE ROLE alice WITH PASSWORD = 'password_a' AND LOGIN = true AND ACCESS TO ALL
DATACENTERS;
CREATE ROLE bob WITH LOGIN = true and PASSWORD = 'password_d' AND ACCESS FROM CIDRS {
'region1', 'region2' };
CREATE ROLE hob WITH LOGIN = true and PASSWORD = 'password_c' AND ACCESS FROM ALL CIDRS;
```

By default roles do not possess `LOGIN` privileges or `SUPERUSER` status.

Permissions on database resources are granted to roles; types of resources include keyspaces, tables, functions and roles themselves. Roles may be granted to other roles to create hierarchical permissions structures; in these hierarchies, permissions and `SUPERUSER` status are inherited, but the `LOGIN` privilege is not.

If a role has the `LOGIN` privilege, clients may identify as that role when connecting. For the duration of that connection, the client will acquire any roles and privileges granted to that role.

Only a client with with the `CREATE` permission on the database roles resource may issue `CREATE ROLE` requests (see the **relevant section**), unless the client is a `SUPERUSER`. Role management in Cassandra is pluggable and custom implementations may support only a subset of the listed options.

Role names should be quoted if they contain non-alphanumeric characters.

Setting credentials for internal authentication

Use the `WITH PASSWORD` clause to set a password for internal authentication, enclosing the password in single quotation marks.

If internal authentication has not been set up or the role does not have `LOGIN` privileges, the `WITH PASSWORD` clause is not necessary.

USE `WITH HASHED PASSWORD` to provide the jBcrypt hashed password directly. See the `hash_password` tool.

Restricting connections to specific datacenters

If a `network_authorizer` has been configured, you can restrict login roles to specific datacenters with the `ACCESS TO DATACENTERS` clause followed by a set literal of datacenters the user can access. Not specifiying datacenters implicitly grants access to all datacenters. The clause `ACCESS TO ALL DATACENTERS` can be used for explicitness, but there’s no functional difference.

Restricting connections from specific CIDR groups

If a `cidr_authorizer` has been configured, you can restrict roles to login only from specific regions, aka CIDR groups, with the `ACCESS FROM CIDRS` clause followed by a set literal of CIDR groups the user can access from. Not specifying CIDR groups implicitly grants access from all CIDR groups. The clause `ACCESS FROM ALL CIDRS` can be used for explicitness, but there’s no functional difference. This clause can be also be used to remove any CIDR groups restrictions. Valid CIDR groups should be used with `ACCESS FROM CIDRS` clause. `nodetool list-cidrgroups` command can be used to see available CIDR groups in the Cluster.

Creating a role conditionally

Attempting to create an existing role results in an invalid query condition unless the `IF NOT EXISTS` option is used. If the option is used and the role exists, the statement is a no-op:

```
CREATE ROLE other_role;
CREATE ROLE IF NOT EXISTS other_role;
```

ALTER ROLE

Altering a role options uses the `ALTER ROLE` statement:

```
alter_role_statement ::= ALTER ROLE [ IF EXISTS ] role_name WITH role_options
```

For example:

```
ALTER ROLE bob WITH PASSWORD = 'PASSWORD_B' AND SUPERUSER = false;
ALTER ROLE bob WITH HASHED PASSWORD =
'$2a$10$JSJEMFm6GeaW9XxT5JIheuEtPvat6i7uKbnTcxX3c1wshIIsGyUtG' AND SUPERUSER = false;
ALTER ROLE rob WITH LOGIN = true and PASSWORD = 'password_c' AND ACCESS FROM ALL CIDRS;
ALTER ROLE hob WITH LOGIN = true and PASSWORD = 'password_d' AND ACCESS FROM CIDRS {
'region1' };
```

If the role does not exist, the statement will return an error, unless `IF EXISTS` is used in which case the operation is a no-op.

USE `WITH HASHED PASSWORD` to provide the jBcrypt hashed password directly. See the `hash_password` tool.

Restricting connections to specific datacenters

If a `network_authorizer` has been configured, you can restrict login roles to specific datacenters with the `ACCESS TO DATACENTERS` clause followed by a set literal of datacenters the user can access. To remove any data center restrictions, use the `ACCESS TO ALL DATACENTERS` clause.

Restricting connections from specific CIDR groups

If a `cidr_authorizer` has been configured, you can restrict roles to login only from specific regions, aka CIDR groups, with the `ACCESS FROM CIDRS` clause followed by a set literal of CIDR groups the user can access from. Not specifying CIDR groups implicitly grants access from all CIDR groups. The clause `ACCESS FROM ALL CIDRS` can be used for explicitness, but there's no functional difference. This clause can be also be used to remove any CIDR groups restrictions. Valid CIDR groups should be used with `ACCESS`

FROM CIDRS clause. nodetool list-cidrgroups command can be used to see available CIDR groups in the Cluster.

Conditions on executing ALTER ROLE statements:

- a client must have SUPERUSER status to alter the SUPERUSER status of another role
- a client cannot alter the SUPERUSER status of any role it currently holds
- a client can only modify certain properties of the role with which it identified at login (e.g. PASSWORD)
- to modify properties of a role, the client must be granted ALTER permission <cql-permissions> on that role

DROP ROLE

Dropping a role uses the DROP ROLE statement:

```
drop_role_statement ::= DROP ROLE [ IF EXISTS ] role_name
```

DROP ROLE requires the client to have DROP permission <cql-permissions> on the role in question. In addition, client may not DROP the role with which it identified at login. Finally, only a client with SUPERUSER status may DROP another SUPERUSER role.

Attempting to drop a role which does not exist results in an invalid query condition unless the IF EXISTS option is used. If the option is used and the role does not exist the statement is a no-op.

NOTE DROP ROLE intentionally does not terminate any open user sessions. Currently connected sessions will remain connected and will retain the ability to perform any database actions which do not require **authorization**. However, if authorization is enabled, **permissions** of the dropped role are also revoked, subject to the **caching options** configured in **cassandra-yaml** file. Should a dropped role be subsequently recreated and have new **permissions** or **roles**` granted to it, any client sessions still connected will acquire the newly granted permissions and roles.

GRANT ROLE

Granting a role to another uses the GRANT ROLE statement:

```
grant_role_statement ::= GRANT role_name TO role_name
```

For example:

```
GRANT report_writer TO alice;
```

This statement grants the `report_writer` role to `alice`. Any permissions granted to `report_writer` are also acquired by `alice`.

Roles are modelled as a directed acyclic graph, so circular grants are not permitted. The following examples result in error conditions:

```
GRANT role_a TO role_b;
GRANT role_b TO role_a;

GRANT role_a TO role_b;
GRANT role_b TO role_c;
GRANT role_c TO role_a;
```

REVOKE ROLE

Revoking a role uses the `REVOKE ROLE` statement:

```
revoke_role_statement ::= REVOKE role_name FROM role_name
```

For example:

```
REVOKE report_writer FROM alice;
```

This statement revokes the `report_writer` role from `alice`. Any permissions that `alice` has acquired via the `report_writer` role are also revoked.

LIST ROLES

All the known roles (in the system or granted to specific role) can be listed using the `LIST ROLES` statement:

```
list_roles_statement ::= LIST ROLES [ OF role_name] [ NORECURSIVE ]
```

For instance:

```
LIST ROLES;
```

returns all known roles in the system, this requires `DESCRIBE` permission on the database roles resource.

This example enumerates all roles granted to `alice`, including those transitively acquired:

```
LIST ROLES OF alice;
```

This example lists all roles directly granted to `bob` without including any of the transitively acquired ones:

```
LIST ROLES OF bob NORECURSIVE;
```

Users

Prior to the introduction of roles in Cassandra 2.2, authentication and authorization were based around the concept of a `USER`. For backward compatibility, the legacy syntax has been preserved with `USER` centric statements becoming synonyms for the `ROLE` based equivalents. In other words, creating/updating a user is just a different syntax for creating/updating a role.

CREATE USER

Creating a user uses the `CREATE USER` statement:

```
create_user_statement ::= CREATE USER [ IF NOT EXISTS ] role_name
                          [ WITH [ HASHED ] PASSWORD string ]
                          [ user_option ]
user_option: SUPERUSER | NOSUPERUSER
```

For example:

```
CREATE USER alice WITH PASSWORD 'password_a' SUPERUSER;
CREATE USER bob WITH PASSWORD 'password_b' NOSUPERUSER;
CREATE USER bob WITH HASHED PASSWORD
'$2a$10$JSJEMFm6GeaW9XxT5JIheuEtPvat6i7uKbnTcxX3c1wshIIsGyUtG' NOSUPERUSER;
```

The `CREATE USER` command is equivalent to `CREATE ROLE` where the `LOGIN` option is `true`. So, the following pairs of statements are equivalent:

```
CREATE USER alice WITH PASSWORD 'password_a' SUPERUSER;
CREATE ROLE alice WITH PASSWORD = 'password_a' AND LOGIN = true AND SUPERUSER = true;
CREATE USER IF NOT EXISTS alice WITH PASSWORD 'password_a' SUPERUSER;
CREATE ROLE IF NOT EXISTS alice WITH PASSWORD = 'password_a' AND LOGIN = true AND
SUPERUSER = true;

CREATE USER alice WITH PASSWORD 'password_a' NOSUPERUSER;
CREATE ROLE alice WITH PASSWORD = 'password_a' AND LOGIN = true AND SUPERUSER = false;

CREATE USER alice WITH PASSWORD 'password_a' NOSUPERUSER;
CREATE ROLE alice WITH PASSWORD = 'password_a' AND LOGIN = true;

CREATE USER alice WITH PASSWORD 'password_a';
CREATE ROLE alice WITH PASSWORD = 'password_a' AND LOGIN = true;

CREATE ROLE rob WITH LOGIN = true and PASSWORD = 'password_c' AND ACCESS FROM ALL CIDRS;
CREATE ROLE hob WITH LOGIN = true and PASSWORD = 'password_d' AND ACCESS FROM CIDRS {
'region1' };
```

ALTER USER

Altering the options of a user uses the `ALTER USER` statement:

```
alter_user_statement ::= ALTER USER [ IF EXISTS ] role_name [ WITH [ HASHED ] PASSWORD
string] [ user_option]
```

If the role does not exist, the statement will return an error, unless `IF EXISTS` is used in which case the operation is a no-op. For example:

```
ALTER USER alice WITH PASSWORD 'PASSWORD_A';
ALTER USER alice WITH HASHED PASSWORD
'$2a$10$JSJEMFm6GeaW9XxT5JIheuEtPvat6i7uKbnTcxX3c1wshIIsGyUtG';
ALTER USER bob SUPERUSER;
```

DROP USER

Dropping a user uses the `DROP USER` statement:

```
drop_user_statement ::= DROP USER [ IF EXISTS ] role_name
```

LIST USERS

Existing users can be listed using the `LIST USERS` statement:

```
list_users_statement::= LIST USERS
```

Note that this statement is equivalent to `LIST ROLES`**, but only roles with the `LOGIN** privilege are included in the output.

Data Control

Permissions

Permissions on resources are granted to roles; there are several different types of resources in Cassandra and each type is modelled hierarchically:

- The hierarchy of Data resources, Keyspaces and Tables has the structure `ALL KEYSPACES` → `KEYSPACE` → `TABLE`.
- Function resources have the structure `ALL FUNCTIONS` → `KEYSPACE` → `FUNCTION`
- Resources representing roles have the structure `ALL ROLES` → `ROLE`
- Resources representing JMX ObjectNames, which map to sets of MBeans/MXBeans, have the structure `ALL MBEANS` → `MBEAN`

Permissions can be granted at any level of these hierarchies and they flow downwards. So granting a permission on a resource higher up the chain automatically grants that same permission on all resources lower down. For example, granting `SELECT` on a `KEYSPACE` automatically grants it on all `TABLES` in that `KEYSPACE`. Likewise, granting a permission on `ALL FUNCTIONS` grants it on every defined function, regardless of which keyspace it is scoped in. It is also possible to grant permissions on all functions scoped to a particular keyspace.

Modifications to permissions are visible to existing client sessions; that is, connections need not be re-established following permissions changes.

The full set of available permissions is:

- `CREATE`
- `ALTER`
- `DROP`
- `SELECT`
- `MODIFY`

- `AUTHORIZE`
- `DESCRIBE`
- `EXECUTE`
- `UNMASK`
- `SELECT_MASKED`

Not all permissions are applicable to every type of resource. For instance, `EXECUTE` is only relevant in the context of functions or mbeans; granting `EXECUTE` on a resource representing a table is nonsensical. Attempting to `GRANT` a permission on resource to which it cannot be applied results in an error response. The following illustrates which permissions can be granted on which types of resource, and which statements are enabled by that permission.

Permission	Resource	Operations
`CREATE`	`ALL KEYSPACES`	`CREATE KEYSPACE` and `CREATE TABLE` in any keyspace
`CREATE`	`KEYSPACE`	`CREATE TABLE` in specified keyspace
`CREATE`	`ALL FUNCTIONS`	`CREATE FUNCTION` in any keyspace and `CREATE AGGREGATE` in any keyspace
`CREATE`	`ALL FUNCTIONS IN KEYSPACE`	`CREATE FUNCTION` and `CREATE AGGREGATE` in specified keyspace
`CREATE`	`ALL ROLES`	`CREATE ROLE`
`ALTER`	`ALL KEYSPACES`	`ALTER KEYSPACE` and `ALTER TABLE` in any keyspace
`ALTER`	`KEYSPACE`	`ALTER KEYSPACE` and `ALTER TABLE` in specified keyspace
`ALTER`	`TABLE`	`ALTER TABLE`
`ALTER`	`ALL FUNCTIONS`	`CREATE FUNCTION` and `CREATE AGGREGATE`: replacing any existing
`ALTER`	`ALL FUNCTIONS IN KEYSPACE`	`CREATE FUNCTION` and `CREATE AGGREGATE`: replacing existing in specified keyspace
`ALTER`	`FUNCTION`	`CREATE FUNCTION` and `CREATE AGGREGATE`: replacing existing
`ALTER`	`ALL ROLES`	`ALTER ROLE` on any role
`ALTER`	`ROLE`	`ALTER ROLE`

Permission	Resource	Operations
`DROP`	`ALL KEYSPACES`	`DROP KEYSPACE` and `DROP TABLE` in any keyspace
`DROP`	`KEYSPACE`	`DROP TABLE` in specified keyspace
`DROP`	`TABLE`	`DROP TABLE`
`DROP`	`ALL FUNCTIONS`	`DROP FUNCTION` and `DROP AGGREGATE` in any keyspace
`DROP`	`ALL FUNCTIONS IN KEYSPACE`	`DROP FUNCTION` and `DROP AGGREGATE` in specified keyspace
`DROP`	`FUNCTION`	`DROP FUNCTION`
`DROP`	`ALL ROLES`	`DROP ROLE` on any role
`DROP`	`ROLE`	`DROP ROLE`
`SELECT`	`ALL KEYSPACES`	`SELECT` on any table
`SELECT`	`KEYSPACE`	`SELECT` on any table in specified keyspace
`SELECT`	`TABLE`	`SELECT` on specified table
`SELECT`	`ALL MBEANS`	Call getter methods on any mbean
`SELECT`	`MBEANS`	Call getter methods on any mbean matching a wildcard pattern
`SELECT`	`MBEAN`	Call getter methods on named mbean
`MODIFY`	`ALL KEYSPACES`	`INSERT`, `UPDATE`, `DELETE` and `TRUNCATE` on any table
`MODIFY`	`KEYSPACE`	`INSERT`, `UPDATE`, `DELETE` and `TRUNCATE` on any table in specified keyspace
`MODIFY`	`TABLE`	`INSERT`, `UPDATE`, `DELETE` and `TRUNCATE` on specified table
`MODIFY`	`ALL MBEANS`	Call setter methods on any mbean
`MODIFY`	`MBEANS`	Call setter methods on any mbean matching a wildcard pattern
`MODIFY`	`MBEAN`	Call setter methods on named mbean

Permission	Resource	Operations
`AUTHORIZE`	`ALL KEYSPACES`	`GRANT PERMISSION` and `REVOKE PERMISSION` on any table
`AUTHORIZE`	`KEYSPACE`	`GRANT PERMISSION` and `REVOKE PERMISSION` on any table in specified keyspace
`AUTHORIZE`	`TABLE`	`GRANT PERMISSION` and `REVOKE PERMISSION` on specified table
`AUTHORIZE`	`ALL FUNCTIONS`	`GRANT PERMISSION` and `REVOKE PERMISSION` on any function
`AUTHORIZE`	`ALL FUNCTIONS IN KEYSPACE`	`GRANT PERMISSION` and `REVOKE PERMISSION` in specified keyspace
`AUTHORIZE`	`FUNCTION`	`GRANT PERMISSION` and `REVOKE PERMISSION` on specified function
`AUTHORIZE`	`ALL MBEANS`	`GRANT PERMISSION` and `REVOKE PERMISSION` on any mbean
`AUTHORIZE`	`MBEANS`	`GRANT PERMISSION` and `REVOKE PERMISSION` on any mbean matching a wildcard pattern
`AUTHORIZE`	`MBEAN`	`GRANT PERMISSION` and `REVOKE PERMISSION` on named mbean
`AUTHORIZE`	`ALL ROLES`	`GRANT ROLE` and `REVOKE ROLE` on any role
`AUTHORIZE`	`ROLES`	`GRANT ROLE` and `REVOKE ROLE` on specified roles
`DESCRIBE`	`ALL ROLES`	`LIST ROLES` on all roles or only roles granted to another, specified role
`DESCRIBE`	`ALL MBEANS`	Retrieve metadata about any mbean from the platform’s MBeanServer
`DESCRIBE`	`MBEANS`	Retrieve metadata about any mbean matching a wildcard patter from the platform’s MBeanServer
`DESCRIBE`	`MBEAN`	Retrieve metadata about a named mbean from the platform’s MBeanServer

Permission	Resource	Operations
`EXECUTE`	`ALL FUNCTIONS`	`SELECT`, `INSERT` and `UPDATE` using any function, and use of any function in `CREATE AGGREGATE`
`EXECUTE`	`ALL FUNCTIONS IN KEYSPACE`	`SELECT`, `INSERT` and `UPDATE` using any function in specified keyspace and use of any function in keyspace in `CREATE AGGREGATE`
`EXECUTE`	`FUNCTION`	`SELECT`, `INSERT` and `UPDATE` using specified function and use of the function in `CREATE AGGREGATE`
`EXECUTE`	`ALL MBEANS`	Execute operations on any mbean
`EXECUTE`	`MBEANS`	Execute operations on any mbean matching a wildcard pattern
`EXECUTE`	`MBEAN`	Execute operations on named mbean
`UNMASK`	`ALL KEYSPACES`	See the clear contents of masked columns on any table
`UNMASK`	`KEYSPACE`	See the clear contents of masked columns on any table in keyspace
`UNMASK`	`TABLE`	See the clear contents of masked columns on the specified table
`SELECT_MASKED`	`ALL KEYSPACES`	`SELECT` restricting masked columns on any table
`SELECT_MASKED`	`KEYSPACE`	`SELECT` restricting masked columns on any table in specified keyspace
`SELECT_MASKED`	`TABLE`	`SELECT` restricting masked columns on the specified table

GRANT PERMISSION

Granting a permission uses the `GRANT PERMISSION` statement:

```
grant_permission_statement ::= GRANT permissions ON resource TO role_name
permissions ::= ALL [ PERMISSIONS ] | permission [ PERMISSION ]
```

```
permission ::= CREATE | ALTER | DROP | SELECT | MODIFY | AUTHORIZE | DESCRIBE | EXECUTE |
UNMASK | SELECT_MASKED
resource ::=    ALL KEYSPACES
                | KEYSPACE keyspace_name
                | [ TABLE ] table_name
                | ALL ROLES
                | ROLE role_name
                | ALL FUNCTIONS [ IN KEYSPACE keyspace_name ]
                | FUNCTION function_name '(' [ cql_type( ',' cql_type )* ] ')'
                | ALL MBEANS
                | ( MBEAN | MBEANS ) string
```

For example:

```
GRANT SELECT ON ALL KEYSPACES TO data_reader;
```

This example gives any user with the role `data_reader` permission to execute `SELECT` statements on any table across all keyspaces:

```
GRANT MODIFY ON KEYSPACE keyspace1 TO data_writer;
```

To give any user with the role `data_writer` permission to perform `UPDATE`, `INSERT`, `UPDATE`, `DELETE` and `TRUNCATE` queries on all tables in the `keyspace1` keyspace:

```
GRANT DROP ON keyspace1.table1 TO schema_owner;
```

To give any user with the `schema_owner` role permissions to `DROP` a specific `keyspace1.table1`:

```
GRANT EXECUTE ON FUNCTION keyspace1.user_function( int ) TO report_writer;
```

This command grants any user with the `report_writer` role permission to execute `SELECT`, `INSERT` and `UPDATE` queries which use the function `keyspace1.user_function( int )`:

```
GRANT DESCRIBE ON ALL ROLES TO role_admin;
```

This grants any user with the `role_admin` role permission to view any and all roles in the system with a `LIST ROLES` statement.

GRANT ALL

When the `GRANT ALL` form is used, the appropriate set of permissions is determined automatically based on the target resource.

Automatic Granting

When a resource is created, via a `CREATE KEYSPACE`, `CREATE TABLE`, `CREATE FUNCTION`, `CREATE AGGREGATE` or `CREATE ROLE` statement, the creator (the role the database user who issues the statement is identified as), is automatically granted all applicable permissions on the new resource.

REVOKE PERMISSION

Revoking a permission from a role uses the `REVOKE PERMISSION` statement:

```
revoke_permission_statement ::= REVOKE permissions ON resource FROM role_name
```

For example:

```
REVOKE SELECT ON ALL KEYSPACES FROM data_reader;
REVOKE MODIFY ON KEYSPACE keyspace1 FROM data_writer;
REVOKE DROP ON keyspace1.table1 FROM schema_owner;
REVOKE EXECUTE ON FUNCTION keyspace1.user_function( int ) FROM report_writer;
REVOKE DESCRIBE ON ALL ROLES FROM role_admin;
```

Because of their function in normal driver operations, certain tables cannot have their `SELECT` permissions revoked. The following tables will be available to all authorized users regardless of their assigned role:

```
* `system_schema.keyspaces`
* `system_schema.columns`
* `system_schema.tables`
* `system.local`
* `system.peers`
```

LIST PERMISSIONS

Listing granted permissions uses the `LIST PERMISSIONS` statement:

```
list_permissions_statement ::= LIST permissions [ ON resource] [ OF role_name[
NORECURSIVE ] ]
```

For example:

```
LIST ALL PERMISSIONS OF alice;
```

Show all permissions granted to `alice`, including those acquired transitively from any other roles:

```
LIST ALL PERMISSIONS ON keyspace1.table1 OF bob;
```

Show all permissions on `keyspace1.table1` granted to `bob`, including those acquired transitively from any other roles. This also includes any permissions higher up the resource hierarchy which can be applied to `keyspace1.table1`. For example, should `bob` have `ALTER` permission on `keyspace1`, that would be included in the results of this query. Adding the `NORECURSIVE` switch restricts the results to only those permissions which were directly granted to `bob` or one of `bob’s roles:

```
LIST SELECT PERMISSIONS OF carlos;
```

Show any permissions granted to `carlos` or any of ``carlos’s roles, limited to `SELECT`` permissions on any resource.

Dynamic Data Masking (DDM)

Dynamic Data Masking (DDM)

Dynamic data masking (DDM) obscures sensitive information while still allowing access to the masked columns. DDM doesn't alter the stored data. Instead, it just presents the data in its obscured form during `SELECT` queries. This aims to provide some degree of protection against accidental data exposure. However, anyone with direct access to the SSTable files will be able to read the clear data.

Masking functions

DDM is based on a set of CQL native functions that obscure sensitive information. The available functions are:

Function	Description

`mask_null(value)`	Replaces the first argument with a `null` column. The returned value is always a non-existent column, and not a not-null column representing a `null` value.
	
	Examples:
	
	`mask_null('Alice')` → `null`
	
	`mask_null(123)` → `null`

Function	Description

`mask_default(value)`	Replaces its argument by an arbitrary, fixed default value of the same type. This will be `****` for text values, zero for numeric values, `false` for booleans, etc.
	
	Variable-length multi-valued types such as lists, sets and maps are masked as empty collections.
	
	Fixed-length multi-valued types such as tuples, user-defined types (UDTs) and vectors are masked by replacing each of their values by the default masking value of the value type.
	
	Examples:
	
	`mask_default('Alice') → '****'`
	
	`mask_default(123) → 0`
	
	`mask_default((list<int>) [1, 2, 3]) → []`
	
	`mask_default((vector<int, 3>) [1, 2, 3]) → [0, 0, 0]`

`mask_replace(value, replacement])`	Replaces the first argument by the replacement value on the second argument. The replacement value needs to have the same type as the replaced value.
	
	Examples:
	
	`mask_replace('Alice', 'REDACTED') → 'REDACTED'`
	
	`mask_replace(123, -1) → -1`

Function	Description
`mask_inner(value, begin, end, [padding])`	Returns a copy of the first `text`, `varchar` or `ascii` argument, replacing each character except the first and last ones by a padding character. The second and third arguments are the size of the exposed prefix and suffix. The optional fourth argument is the padding character, `\*` by default. Examples: `mask_inner('Alice', 1, 2) → 'Ace'` `mask_inner('Alice', 1, null) → 'A'` `mask_inner('Alice', null, 2) → '*ce'` `mask_inner('Alice', 2, 1, '#') → 'Al##e'`
`mask_outer(value, begin, end, [padding])`	Returns a copy of the first `text`, `varchar` or `ascii` argument, replacing the first and last character by a padding character. The second and third arguments are the size of the exposed prefix and suffix. The optional fourth argument is the padding character, `\*` by default. Examples: `mask_outer('Alice', 1, 2) → '*li'` `mask_outer('Alice', 1, null) → '*lice'` `mask_outer('Alice', null, 2) → 'Ali'` `mask_outer('Alice', 2, 1, '#') → '##ic#'`
`mask_hash(value, [algorithm])`	Returns a `blob` containing the hash of the first argument. The optional second argument is the hashing algorithm to be used, according the available Java security provider. The default hashing algorithm is `SHA-256`. Examples: `mask_hash('Alice')` `mask_hash('Alice', 'SHA-512')`

Those functions can be used on `SELECT` queries to get an obscured view of the data. For example:

```
CREATE TABLE patients (
   id timeuuid PRIMARY KEY,
   name text,
   birth date
);

INSERT INTO patients(id, name, birth) VALUES (now(), 'alice', '1982-01-02');
INSERT INTO patients(id, name, birth) VALUES (now(), 'bob', '1982-01-02');

SELECT mask_inner(name, 1, null), mask_default(birth) FROM patients;

//   system.mask_inner(name, 1, NULL) | system.mask_default(birth)
// -----------------------------------+----------------------------
//                                b** |                 1970-01-01
//                              a**** |                 1970-01-01
```

Attaching masking functions to table columns

A masking function can be permanently attached to any column of a table. If a masking column is defined, `SELECT` queries will always return the column values in their masked form. The masking will be transparent to the users running `SELECT` queries. The only way to know that a column is masked is to consult the table definition.

This is an optional feature that is disabled by default. To use the feature, enable the `dynamic_data_masking_enabled` property in `cassandra.yaml`.

The masks of the columns of a table can be defined in the `CREATE TABLE` to create the table schema. This example uses the `mask_inner` function with two arguments:

```
CREATE TABLE patients (
    id timeuuid PRIMARY KEY,
    name text MASKED WITH mask_inner(1, null),
    birth date MASKED WITH mask_default()
 );
```

When using a `SELECT` query on this data, three arguments are required for the `mask_inner` function, but the first argument is always omitted when attaching the function to the table schema. The value of that first argument is always interpreted as the value of the masked column, in this case a `text` column.

For the same reason, using the masking function `mask_default` doesn't have any argument when creating the table schema, but it requires one argument when used on `SELECT` queries.

Data can be normally inserted into the masked table without alteration. For example:

```
INSERT INTO patients(id, name, birth) VALUES (now(), 'alice', '1984-01-02');
INSERT INTO patients(id, name, birth) VALUES (now(), 'bob', '1982-02-03');
```

The `SELECT` query will return the masked data. The masking function will be automatically applied to the column values.

```
SELECT name, birth FROM patients;

//  name  | birth
// -------+------------
//  a**** | 1970-01-01
//    b** | 1970-01-01
```

An `ALTER TABLE` query can be used to make changes to a masking function on a table column.

```
ALTER TABLE patients ALTER name
  MASKED WITH mask_default();
```

In a similar way, a masking function can be detached from a column with an `ALTER TABLE` query:

```
ALTER TABLE patients ALTER name
  DROP MASKED;
```

Permissions

Ordinary users are created without the `UNMASK` permission and will see masked values. Giving a user the `UNMASK` permission allows them to retrieve the unmasked values of masked columns. Superusers are automatically created with the `UNMASK` permission, and will see the unmasked values in a `SELECT` query results.

For example, suppose that we have a table with masked columns:

```
CREATE TABLE patients (
    id timeuuid PRIMARY KEY,
    name text MASKED WITH mask_inner(1, null),
    birth date MASKED WITH mask_default()
 );
```

And we insert some data into the table:

```
INSERT INTO patients(id, name, birth) VALUES (now(), 'alice', '1984-01-02');
INSERT INTO patients(id, name, birth) VALUES (now(), 'bob', '1982-02-03');
```

```
LOGIN unprivileged
SELECT name, birth FROM patients;

//  name  | birth
// -------+------------
//  a**** | 1970-01-01
//    b** | 1970-01-01
```

Then we create two users with `SELECT` permission for the table, but we only grant the `UNMASK` permission to one of the users:

```
CREATE USER privileged WITH PASSWORD 'xyz';
GRANT SELECT ON TABLE patients TO privileged;
GRANT UNMASK ON TABLE patients TO privileged;

CREATE USER unprivileged WITH PASSWORD 'xyz';
GRANT SELECT ON TABLE patients TO unprivileged;
```

The user with the `UNMASK` permission can see the clear, unmasked data:

```
LOGIN privileged
SELECT name, birth FROM patients;

//  name  | birth
// -------+------------
//  alice | 1984-01-02
//    bob | 1982-02-03
```

The user without the `UNMASK` permission can only see the masked data:

```
LOGIN unprivileged
SELECT name, birth FROM patients;

//  name  | birth
// -------+------------
//  a**** | 1970-01-01
//    b** | 1970-01-01
```

The UNMASK permission works like any other permission, and can be revoked at will:

```
REVOKE UNMASK ON TABLE patients
  FROM privileged;
```

Please note that, when authentication is disabled, the anonymous default user has all the permissions, including the UNMASK permission, and can see the unmasked data. In other words, attaching data masking functions to columns only makes sense if authentication is enabled.

Only users with the UNMASK permission are allowed to use masked columns in the WHERE clause of a SELECT query. Users without the UNMASK permission cannot use this feature. This feature prevents malicious users seeing clear data by running exhaustive, brute force queries. The user without the UNMASK permission will see the following:

```
CREATE USER untrusted_user WITH PASSWORD 'xyz';
GRANT SELECT ON TABLE patients TO untrusted_user;
LOGIN untrusted_user
SELECT name, birth FROM patients WHERE name = 'Alice' ALLOW FILTERING;

// Unauthorized: Error from server: code=2100 [Unauthorized] message="User untrusted_user
has no UNMASK nor SELECT_UNMASK permission on table k.patients"
```

There are some use cases where a trusted database user needs to produce masked data that untrusted external users will query. For instance, a trusted app can connect to the database and with a query extract masked data that will be displayed to its end users. In that case, the trusted user (the app) can be given the SELECT_MASKED permission. This permission lets the user query masked columns in the WHERE clause of a SELECT query, while still only seeing the masked data in the query results:

```
CREATE USER trusted_user WITH PASSWORD 'xyz';
GRANT SELECT, SELECT_MASKED ON TABLE patients TO trusted_user;
LOGIN trusted_user
SELECT name, birth FROM patients WHERE name = 'Alice' ALLOW FILTERING;

//  name  | birth
// -------+------------
//  a**** | 1970-01-01
```

Custom functions

User-defined functions (UDFs) can be attached to a table column. The UDFs used for masking should belong to the same keyspace as the masked table. The column value to mask will be passed as the first argument of the attached UDF. Thus, the UDFs attached to a column should have at least one argument, and that argument should have the same type as the masked column. Also, the attached UDF should return values of the same type as the masked column:

```
CREATE FUNCTION redact(input text)
   CALLED ON NULL INPUT
   RETURNS text
   LANGUAGE java
   AS 'return "redacted";';

CREATE TABLE patients (
   id timeuuid PRIMARY KEY,
   name text MASKED WITH redact(),
   birth date
);
```

This creates a dependency between the table schema and the functions. Any attempt to drop the function will be rejected while this dependency exists. Consequently, you must drop the mask column in the table before dropping the function:

```
ALTER TABLE patients ALTER name
  DROP MASKED;
```

Dropping the column, or its containing table, or its containing keyspace will also remove the dependency.

Aggregate functions cannot be used as masking functions.

Materialized Views

Materialized Views

Materialized views names are defined by:

```
view_name::= re('[a-zA-Z_0-9]+')
```

CREATE MATERIALIZED VIEW

You can create a materialized view on a table using a `CREATE MATERIALIZED VIEW` statement:

```
create_materialized_view_statement::= CREATE MATERIALIZED VIEW [ IF NOT EXISTS ]
view_name
    AS select_statement
    PRIMARY KEY '(' primary_key')'
    WITH table_options
```

For instance:

```
CREATE MATERIALIZED VIEW monkeySpecies_by_population AS
   SELECT * FROM monkeySpecies
   WHERE population IS NOT NULL AND species IS NOT NULL
   PRIMARY KEY (population, species)
   WITH comment='Allow query by population instead of species';
```

The `CREATE MATERIALIZED VIEW` statement creates a new materialized view. Each such view is a set of *rows* which corresponds to rows which are present in the underlying, or base, table specified in the `SELECT` statement. A materialized view cannot be directly updated, but updates to the base table will cause corresponding updates in the view.

Creating a materialized view has 3 main parts:

- The **select statement** that restrict the data included in the view.
- The **primary key** definition for the view.
- The **options** for the view.

Attempting to create an already existing materialized view will return an error unless the `IF NOT EXISTS` option is used. If it is used, the statement will be a no-op if the materialized view already exists.

NOTE By default, materialized views are built in a single thread. The initial build can be parallelized by increasing the number of threads specified by the property

> `concurrent_materialized_view_builders` in `cassandra.yaml`. This property can also be manipulated at runtime through both JMX and the `setconcurrentviewbuilders` and `getconcurrentviewbuilders` nodetool commands.

MV select statement

The select statement of a materialized view creation defines which of the base table is included in the view. That statement is limited in a number of ways:

- the **selection** is limited to those that only select columns of the base table. In other words, you can't use any function (aggregate or not), casting, term, etc. Aliases are also not supported. You can however use * as a shortcut of selecting all columns. Further, **static columns** cannot be included in a materialized view. Thus, a `SELECT *` command isn't allowed if the base table has static columns. The `WHERE` clause has the following restrictions:
    - cannot include any `bind_marker`
    - cannot have columns that are not part of the *base table* primary key that are not restricted by an `IS NOT NULL` restriction
    - no other restriction is allowed
    - cannot have columns that are part of the *view* primary key be null, they must always be at least restricted by a `IS NOT NULL` restriction (or any other restriction, but they must have one).
- cannot have an **ordering clause**, a **limit**, or **ALLOW FILTERING**]]]

MV primary key

A view must have a primary key and that primary key must conform to the following restrictions:

- it must contain all the primary key columns of the base table. This ensures that every row of the view correspond to exactly one row of the base table.
- it can only contain a single column that is not a primary key column in the base table.

So for instance, give the following base table definition:

```
CREATE TABLE t (
    k int,
    c1 int,
    c2 int,
    v1 int,
    v2 int,
    PRIMARY KEY (k, c1, c2)
);
```

then the following view definitions are allowed:

```
CREATE MATERIALIZED VIEW mv1 AS
   SELECT * FROM t
   WHERE k IS NOT NULL AND c1 IS NOT NULL AND c2 IS NOT NULL
   PRIMARY KEY (c1, k, c2);

CREATE MATERIALIZED VIEW mv1 AS
  SELECT * FROM t
  WHERE k IS NOT NULL AND c1 IS NOT NULL AND c2 IS NOT NULL
  PRIMARY KEY (v1, k, c1, c2);
```

but the following ones are **not** allowed:

```
// Error: cannot include both v1 and v2 in the primary key as both are not in the base
table primary key

CREATE MATERIALIZED VIEW mv1 AS
   SELECT * FROM t
   WHERE k IS NOT NULL AND c1 IS NOT NULL AND c2 IS NOT NULL AND v1 IS NOT NULL
   PRIMARY KEY (v1, v2, k, c1, c2);

// Error: must include k in the primary as it's a base table primary key column

CREATE MATERIALIZED VIEW mv1 AS
   SELECT * FROM t
   WHERE c1 IS NOT NULL AND c2 IS NOT NULL
   PRIMARY KEY (c1, c2);
```

MV options

A materialized view is internally implemented by a table and as such, creating a MV allows the `same options than creating a table <create-table-options>`.

ALTER MATERIALIZED VIEW

After creation, you can alter the options of a materialized view using the `ALTER MATERIALIZED VIEW` statement:

```
alter_materialized_view_statement::= ALTER MATERIALIZED VIEW [ IF EXISTS ] view_name WITH
table_options
```

The options that can be updated are the same than at creation time and thus the `same than for tables <create-table-options>`. If the view does not exist, the statement will return an error, unless `IF EXISTS` is used in which case the operation is a no-op.

DROP MATERIALIZED VIEW

Dropping a materialized view using the `DROP MATERIALIZED VIEW` statement:

```
drop_materialized_view_statement::= DROP MATERIALIZED VIEW [ IF EXISTS ] view_name;
```

If the materialized view does not exists, the statement will return an error, unless `IF EXISTS` is used in which case the operation is a no-op.

MV Limitations

NOTE Removal of columns not selected in the Materialized View (via `UPDATE base SET unselected_column = null` or `DELETE unselected_column FROM base`) may shadow missed updates to other columns received by hints or repair. For this reason, we advise against doing deletions on base columns not selected in views until this is fixed on CASSANDRA-13826.

Functions

Functions

CQL supports 2 main categories of functions:

- **scalar functions** that take a number of values and produce an output
- **aggregate functions** that aggregate multiple rows resulting from a `SELECT` statement

In both cases, CQL provides a number of native "hard-coded" functions as well as the ability to create new user-defined functions.

NOTE By default, the use of user-defined functions is disabled by default for security concerns (even when enabled, the execution of user-defined functions is sandboxed and a "rogue" function should not be allowed to do evil, but no sandbox is perfect so using user-defined functions is opt-in). See the `user_defined_functions_enabled` in `cassandra.yaml` to enable them.

A function is identifier by its name:

```
function_name ::= [ keyspace_name'.' ] name
```

Scalar functions

Native functions

Cast

The `cast` function can be used to converts one native datatype to another.

The following table describes the conversions supported by the `cast` function. Cassandra will silently ignore any cast converting a datatype into its own datatype.

From	To
`ascii`	`text`, `varchar`
`bigint`	`tinyint`, `smallint`, `int`, `float`, `double`, `decimal`, `varint`, `text`, `varchar`
`boolean`	`text`, `varchar`
`counter`	`tinyint`, `smallint`, `int`, `bigint`, `float`, `double`, `decimal`, `varint`, `text`, `varchar`
`date`	`timestamp`

From	To
`decimal`	`tinyint`, `smallint`, `int`, `bigint`, `float`, `double`, `varint`, `text`, `varchar`
`double`	`tinyint`, `smallint`, `int`, `bigint`, `float`, `decimal`, `varint`, `text`, `varchar`
`float`	`tinyint`, `smallint`, `int`, `bigint`, `double`, `decimal`, `varint`, `text`, `varchar`
`inet`	`text`, `varchar`
`int`	`tinyint`, `smallint`, `bigint`, `float`, `double`, `decimal`, `varint`, `text`, `varchar`
`smallint`	`tinyint`, `int`, `bigint`, `float`, `double`, `decimal`, `varint`, `text`, `varchar`
`time`	`text`, `varchar`
`timestamp`	`date`, `text`, `varchar`
`timeuuid`	`timestamp`, `date`, `text`, `varchar`
`tinyint`	`tinyint`, `smallint`, `int`, `bigint`, `float`, `double`, `decimal`, `varint`, `text`, `varchar`
`uuid`	`text`, `varchar`
`varint`	`tinyint`, `smallint`, `int`, `bigint`, `float`, `double`, `decimal`, `text`, `varchar`

The conversions rely strictly on Java's semantics. For example, the double value 1 will be converted to the text value '1.0'. For instance:

```
SELECT avg(cast(count as double)) FROM myTable
```

Token

The `token` function computes the token for a given partition key. The exact signature of the token function depends on the table concerned and the partitioner used by the cluster.

The type of the arguments of the `token` depend on the partition key column type. The returned type depends on the defined partitioner:

Partitioner	Returned type
Murmur3Partitioner	`bigint`
RandomPartitioner	`varint`
ByteOrderedPartitioner	`blob`

For example, consider the following table:

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

The table uses the default Murmur3Partitioner. The `token` function uses the single argument `text`, because the partition key is `userid` of text type. The returned type will be `bigint`.

Uuid

The `uuid` function takes no parameters and generates a random type 4 uuid suitable for use in `INSERT` or `UPDATE` statements.

Timeuuid functions

`now`

The `now` function takes no arguments and generates, on the coordinator node, a new unique timeuuid at the time the function is invoked. Note that this method is useful for insertion but is largely non-sensical in `WHERE` clauses.

For example, a query of the form:

```
SELECT * FROM myTable WHERE t = now();
```

will not return a result, by design, since the value returned by `now()` is guaranteed to be unique.

`current_timeuuid` is an alias of `now`.

`min_timeuuid` and `max_timeuuid`

The `min_timeuuid` function takes a `timestamp` value `t`, either a timestamp or a date string. It returns a *fake* `timeuuid` corresponding to the *smallest* possible `timeuuid` for timestamp `t`. The `max_timeuuid` works similarly, but returns the *largest* possible `timeuuid`.

For example:

```
SELECT * FROM myTable
 WHERE t > max_timeuuid('2013-01-01 00:05+0000')
   AND t < min_timeuuid('2013-02-02 10:00+0000');
```

will select all rows where the `timeuuid` column `t` is later than `'2013-01-01 00:05+0000'` and earlier than

'2013-02-02 10:00+0000'. The clause t >= maxTimeuuid('2013-01-01 00:05+0000') would still *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').

NOTE The values generated by min_timeuuid and max_timeuuid are called *fake* UUID because they do no respect the time-based UUID generation process specified by the IETF RFC 4122. In particular, the value returned by these two methods will not be unique. Thus, only use these methods for **querying**, not for **insertion**, to prevent possible data overwriting.

Datetime functions

Retrieving the current date/time

The following functions can be used to retrieve the date/time at the time where the function is invoked:

Function name	Output type
current_timestamp	timestamp
current_date	date
current_time	time
current_timeuuid	timeUUID

For example the last two days of data can be retrieved using:

```
SELECT * FROM myTable WHERE date >= current_date() - 2d;
```

Time conversion functions

A number of functions are provided to convert a timeuuid, a timestamp or a date into another native type.

Function name	Input type	Description
to_date	timeuuid	Converts the timeuuid argument into a date type
to_date	timestamp	Converts the timestamp argument into a date type
to_timestamp	timeuuid	Converts the timeuuid argument into a timestamp type
to_timestamp	date	Converts the date argument into a timestamp type

Function name	Input type	Description
`to_unix_timestamp`	`timeuuid`	Converts the `timeuuid` argument into a `bigInt` raw value
`to_unix_timestamp`	`timestamp`	Converts the `timestamp` argument into a `bigInt` raw value
`to_unix_timestamp`	`date`	Converts the `date` argument into a `bigInt` raw value

Blob conversion functions

A number of functions are provided to convert the native types into binary data, or a `blob`. For every **type** supported by CQL, the function `type_as_blob` takes a argument of type `type` and returns it as a `blob`. Conversely, the function `blob_as_type` takes a 64-bit `blob` argument and converts it to a `bigint` value. For example, `bigint_as_blob(3)` returns `0x0000000000000003` and `blob_as_bigint(0x0000000000000003)` returns `3`.

Math Functions

Cql provides the following math functions: `abs`, `exp`, `log`, `log10`, and `round`. The return type for these functions is always the same as the input type.

Function name	Description
`abs`	Returns the absolute value of the input.
`exp`	Returns the number e to the power of the input.
`log`	Returns the natural log of the input.
`log10`	Returns the log base 10 of the input.
`round`	Rounds the input to the nearest whole number using rounding mode `HALF_UP`.

Collection functions

A number of functions are provided to operate on collection columns.

Function name	Input type	Description
`map_keys`	`map`	Gets the keys of the `map` argument, returned as a `set`.
`map_values`	`map`	Gets the values of the `map` argument, returned as a `list`.
`collection_count`	`map`, `set` or `list`	Gets the number of elements in the collection argument.

Function name	Input type	Description
`collection_min`	`set` or `list`	Gets the minimum element in the collection argument.
`collection_max`	`set` or `list`	Gets the maximum element in the collection argument.
`collection_sum`	numeric `set` or `list`	Computes the sum of the elements in the collection argument. The returned value is of the same type as the input collection elements, so there is a risk of overflowing the data type if the sum of the values exceeds the maximum value that the type can represent.
`collection_avg`	numeric `set` or `list`	Computes the average of the elements in the collection argument. The average of an empty collection returns zero. The returned value is of the same type as the input collection elements, which might include rounding and truncations. For example `collection_avg([1, 2])` returns `1` instead of `1.5`.

Data masking functions

A number of functions allow to obscure the real contents of a column containing sensitive data.

Function	Description

`mask_null(value)`	Replaces the first argument with a `null` column. The returned value is always a non-existent column, and not a not-null column representing a `null` value.
	
	Examples:
	
	`mask_null('Alice')` → `null`
	
	`mask_null(123)` → `null`

Function	Description

`mask_default(value)`	Replaces its argument by an arbitrary, fixed default value of the same type. This will be `****` for text values, zero for numeric values, `false` for booleans, etc.
	
	Variable-length multi-valued types such as lists, sets and maps are masked as empty collections.
	
	Fixed-length multi-valued types such as tuples, user-defined types (UDTs) and vectors are masked by replacing each of their values by the default masking value of the value type.
	
	Examples:
	
	`mask_default('Alice') → '****'`
	
	`mask_default(123) → 0`
	
	`mask_default((list<int>) [1, 2, 3]) → []`
	
	`mask_default((vector<int, 3>) [1, 2, 3]) → [0, 0, 0]`

`mask_replace(value, replacement])`	Replaces the first argument by the replacement value on the second argument. The replacement value needs to have the same type as the replaced value.
	
	Examples:
	
	`mask_replace('Alice', 'REDACTED') → 'REDACTED'`
	
	`mask_replace(123, -1) → -1`

Function	Description

`mask_inner(value, begin, end, [padding])`	Returns a copy of the first `text`, `varchar` or `ascii` argument, replacing each character except the first and last ones by a padding character. The second and third arguments are the size of the exposed prefix and suffix. The optional fourth argument is the padding character, `\*` by default.
	
	Examples:
	
	`mask_inner('Alice', 1, 2) → 'Ace'`
	
	`mask_inner('Alice', 1, null) → 'A'`
	
	`mask_inner('Alice', null, 2) → '*ce'`
	
	`mask_inner('Alice', 2, 1, '#') → 'Al##e'`

`mask_outer(value, begin, end, [padding])`	Returns a copy of the first `text`, `varchar` or `ascii` argument, replacing the first and last character by a padding character. The second and third arguments are the size of the exposed prefix and suffix. The optional fourth argument is the padding character, `\*` by default.
	
	Examples:
	
	`mask_outer('Alice', 1, 2) → '*li'`
	
	`mask_outer('Alice', 1, null) → '*lice'`
	
	`mask_outer('Alice', null, 2) → 'Ali'`
	
	`mask_outer('Alice', 2, 1, '#') → '##ic#'`

`mask_hash(value, [algorithm])`	Returns a `blob` containing the hash of the first argument. The optional second argument is the hashing algorithm to be used, according the available Java security provider. The default hashing algorithm is `SHA-256`.
	
	Examples:
	
	`mask_hash('Alice')`
	
	`mask_hash('Alice', 'SHA-512')`

Vector similarity functions

A number of functions allow to obtain the similarity score between vectors of floats.

Function	Description

`similarity_cosine(vector, vector)`	Calculates the cosine similarity score between two float vectors of the same dimension.
	
	Examples:
	
	`similarity_cosine([0.1, 0.2], null) → null`
	
	`similarity_cosine([0.1, 0.2], [0.1, 0.2]) → 1`
	
	`similarity_cosine([0.1, 0.2], [-0.1, -0.2]) → 0`
	
	`similarity_cosine([0.1, 0.2], [0.9, 0.8]) → 0.964238`

`similarity_euclidean(vector, vector)`	Calculates the euclidian distance between two float vectors of the same dimension.
	
	Examples:
	
	`similarity_euclidean([0.1, 0.2], null) → null`
	
	`similarity_euclidean([0.1, 0.2], [0.1, 0.2]) → 1`
	
	`similarity_euclidean([0.1, 0.2], [-0.1, -0.2]) → 0.833333`
	
	`similarity_euclidean([0.1, 0.2], [0.9, 0.8]) → 0.5`

Function	Description

`similarity_dot_product(vector, vector)`	Calculates the dot product between two float vectors of the same dimension.
	
	Examples:
	
	`similarity_dot_product([0.1, 0.2], null) → null`
	
	`similarity_dot_product([0.1, 0.2], [0.1, 0.2]) → 0.525`
	
	`similarity_dot_product([0.1, 0.2], [-0.1, -0.2]) → 0.475`
	
	`similarity_dot_product([0.1, 0.2], [0.9, 0.8]) → 0.625`

User-defined functions

User-defined functions (UDFs) execute user-provided code in Cassandra. By default, Cassandra supports defining functions in *Java*.

UDFs are part of the Cassandra schema, and are automatically propagated to all nodes in the cluster. UDFs can be *overloaded*, so that multiple UDFs with different argument types can have the same function name.

NOTE *JavaScript* user-defined functions have been deprecated in Cassandra 4.1. In preparation for Cassandra 5.0, their removal is already in progress. For more information - CASSANDRA-17281, CASSANDRA-18252.

For example:

```
CREATE FUNCTION sample ( arg int ) ...;
CREATE FUNCTION sample ( arg text ) ...;
```

UDFs are susceptible to all of the normal problems with the chosen programming language. Accordingly, implementations should be safe against null pointer exceptions, illegal arguments, or any other potential source of exceptions. An exception during function execution will result in the entire statement failing. Valid queries for UDF use are `SELECT`, `INSERT` and `UPDATE` statements.

Complex types like collections, tuple types and user-defined types are valid argument and return types in UDFs. Tuple types and user-defined types use the DataStax Java Driver conversion functions. Please see the Java Driver documentation for details on handling tuple types and user-defined types.

Arguments for functions can be literals or terms. Prepared statement placeholders can be used, too.

Note the use the double dollar-sign syntax to enclose the UDF source code.

For example:

```
CREATE FUNCTION some_function ( arg int )
    RETURNS NULL ON NULL INPUT
    RETURNS int
    LANGUAGE java
    AS $$ return arg; $$;

SELECT some_function(column) FROM atable ...;
UPDATE atable SET col = some_function(?) ...;

CREATE TYPE custom_type (txt text, i int);
CREATE FUNCTION fct_using_udt ( udtarg frozen )
    RETURNS NULL ON NULL INPUT
    RETURNS text
    LANGUAGE java
    AS $$ return udtarg.getString("txt"); $$;
```

The implicitly available `udfContext` field (or binding for script UDFs) provides the necessary functionality to create new UDT and tuple values:

```
CREATE TYPE custom_type (txt text, i int);
CREATE FUNCTION fct\_using\_udt ( somearg int )
    RETURNS NULL ON NULL INPUT
    RETURNS custom_type
    LANGUAGE java
    AS $$
        UDTValue udt = udfContext.newReturnUDTValue();
        udt.setString("txt", "some string");
        udt.setInt("i", 42);
        return udt;
    $$;
```

The definition of the `UDFContext` interface can be found in the Apache Cassandra source code for `org.apache.cassandra.cql3.functions.UDFContext`.

```
public interface UDFContext
{
    UDTValue newArgUDTValue(String argName);
    UDTValue newArgUDTValue(int argNum);
    UDTValue newReturnUDTValue();
    UDTValue newUDTValue(String udtName);
    TupleValue newArgTupleValue(String argName);
    TupleValue newArgTupleValue(int argNum);
    TupleValue newReturnTupleValue();
    TupleValue newTupleValue(String cqlDefinition);
}
```

Java UDFs already have some imports for common interfaces and classes defined. These imports are:

```
import java.nio.ByteBuffer;
import java.util.List;
import java.util.Map;
import java.util.Set;
import org.apache.cassandra.cql3.functions.UDFContext;
import com.datastax.driver.core.TypeCodec;
import com.datastax.driver.core.TupleValue;
import com.datastax.driver.core.UDTValue;
```

Please note, that these convenience imports are not available for script UDFs.

CREATE FUNCTION statement

Creating a new user-defined function uses the `CREATE FUNCTION` statement:

```
create_function_statement::= CREATE [ OR REPLACE ] FUNCTION [ IF NOT EXISTS]
    function_name '(' arguments_declaration ')'
    [ CALLED | RETURNS NULL ] ON NULL INPUT
    RETURNS cql_type
    LANGUAGE identifier
    AS string arguments_declaration: identifier cql_type ( ',' identifier cql_type )*
```

For example:

```
CREATE OR REPLACE FUNCTION somefunction(somearg int, anotherarg text, complexarg
frozen<someUDT>, listarg list)
    RETURNS NULL ON NULL INPUT
    RETURNS text
    LANGUAGE java
    AS $$
        // some Java code
    $$;

CREATE FUNCTION IF NOT EXISTS akeyspace.fname(someArg int)
```

```
    CALLED ON NULL INPUT
    RETURNS text
    LANGUAGE java
    AS $$
        // some Java code
    $$;
```

`CREATE FUNCTION` with the optional `OR REPLACE` keywords creates either a function or replaces an existing one with the same signature. A `CREATE FUNCTION` without `OR REPLACE` fails if a function with the same signature already exists. If the optional `IF NOT EXISTS` keywords are used, the function will only be created only if another function with the same signature does not exist. `OR REPLACE` and `IF NOT EXISTS` cannot be used together.

Behavior for `null` input values must be defined for each function:

- `RETURNS NULL ON NULL INPUT` declares that the function will always return `null` if any of the input arguments is `null`.
- `CALLED ON NULL INPUT` declares that the function will always be executed.

Function Signature

Signatures are used to distinguish individual functions. The signature consists of a fully-qualified function name of the <keyspace>.<function_name> and a concatenated list of all the argument types.

Note that keyspace names, function names and argument types are subject to the default naming conventions and case-sensitivity rules.

Functions belong to a keyspace; if no keyspace is specified, the current keyspace is used. User-defined functions are not allowed in the system keyspaces.

DROP FUNCTION statement

Dropping a function uses the `DROP FUNCTION` statement:

```
drop_function_statement::= DROP FUNCTION [ IF EXISTS ] function_name [ '('
arguments_signature ')' ]
arguments_signature::= cql_type ( ',' cql_type )*
```

For example:

```
DROP FUNCTION myfunction;
DROP FUNCTION mykeyspace.afunction;
DROP FUNCTION afunction ( int );
DROP FUNCTION afunction ( text );
```

You must specify the argument types of the function, the arguments_signature, in the drop command if there are multiple overloaded functions with the same name but different signatures. `DROP FUNCTION` with the optional `IF EXISTS` keywords drops a function if it exists, but does not throw an error if it doesn't.

Aggregate functions

Aggregate functions work on a set of rows. Values for each row are input, to return a single value for the set of rows aggregated.

If `normal` columns, `scalar functions`, `UDT` fields, `writetime`, or `ttl` are selected together with aggregate functions, the values returned for them will be the ones of the first row matching the query.

Native aggregates

Count

The `count` function can be used to count the rows returned by a query.

For example:

```
SELECT COUNT (*) FROM plays;
SELECT COUNT (1) FROM plays;
```

It also can count the non-null values of a given column:

```
SELECT COUNT (scores) FROM plays;
```

Max and Min

The `max` and `min` functions compute the maximum and the minimum value returned by a query for a given column.

For example:

```
SELECT MIN (players), MAX (players) FROM plays WHERE game = 'quake';
```

Sum

The `sum` function sums up all the values returned by a query for a given column.

The returned value is of the same type as the input collection elements, so there is a risk of overflowing if the sum of the values exceeds the maximum value that the type can represent.

For example:

```
SELECT SUM (players) FROM plays;
```

The returned value is of the same type as the input values, so there is a risk of overflowing the type if the sum of the values exceeds the maximum value that the type can represent. You can use type casting to cast the input values as a type large enough to contain the type. For example:

```
SELECT SUM (CAST (players AS VARINT)) FROM plays;
```

Avg

The `avg` function computes the average of all the values returned by a query for a given column.

For example:

```
SELECT AVG (players) FROM plays;
```

The average of an empty collection returns zero.

The returned value is of the same type as the input values, which might include rounding and truncations. For example `collection_avg([1, 2])` returns `1` instead of `1.5`. You can use type casting to cast to a type with the desired decimal precision. For example:

```
SELECT AVG (CAST (players AS FLOAT)) FROM plays;
```

User-Defined Aggregates (UDAs)

User-defined aggregates allow the creation of custom aggregate functions. User-defined aggregates can be used in `SELECT` statement.

Each aggregate requires an *initial state* of type `STYPE` defined with the ``INITCOND`value (default value: `null``). The first argument of the state function must have type `STYPE`. The remaining arguments of the state function must match the types of the user-defined aggregate arguments. 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 FINALFUNC is executed with last state value as its argument.

The STYPE value is mandatory in order to distinguish possibly overloaded versions of the state and/or final function, since the overload can appear after creation of the aggregate.

A complete working example for user-defined aggregates (assuming that a keyspace has been selected using the USE statement):

```
CREATE OR REPLACE FUNCTION test.averageState(state tuple<int,bigint>, val int)
    CALLED ON NULL INPUT
    RETURNS tuple
    LANGUAGE java
    AS $$
        if (val != null) {
            state.setInt(0, state.getInt(0)+1);
            state.setLong(1, state.getLong(1)+val.intValue());
        }
        return state;
    $$;

CREATE OR REPLACE FUNCTION test.averageFinal (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 OR REPLACE AGGREGATE test.average(int)
    SFUNC averageState
    STYPE tuple
    FINALFUNC averageFinal
    INITCOND (0, 0);

CREATE TABLE test.atable (
    pk int PRIMARY KEY,
    val int
);

INSERT INTO test.atable (pk, val) VALUES (1,1);
INSERT INTO test.atable (pk, val) VALUES (2,2);
INSERT INTO test.atable (pk, val) VALUES (3,3);
INSERT INTO test.atable (pk, val) VALUES (4,4);
```

```
SELECT test.average(val) FROM atable;
```

CREATE AGGREGATE statement

Creating (or replacing) a user-defined aggregate function uses the `CREATE AGGREGATE` statement:

```
create_aggregate_statement ::= CREATE [ OR REPLACE ] AGGREGATE [ IF NOT EXISTS ]
                                function_name '(' arguments_signature')'
                                SFUNC function_name
                                STYPE cql_type:
                                [ FINALFUNC function_name]
                                [ INITCOND term ]
```

See above for a complete example.

The `CREATE AGGREGATE` command with the optional `OR REPLACE` keywords creates either an aggregate or replaces an existing one with the same signature. A `CREATE AGGREGATE` without `OR REPLACE` fails if an aggregate with the same signature already exists. The `CREATE AGGREGATE` command with the optional `IF NOT EXISTS` keywords creates an aggregate if it does not already exist. The `OR REPLACE` and `IF NOT EXISTS` phrases cannot be used together.

The `STYPE` value defines the type of the state value and must be specified. The optional `INITCOND` defines the initial state value for the aggregate; the default value is `null`. A non-null `INITCOND` must be specified for state functions that are declared with `RETURNS NULL ON NULL INPUT`.

The `SFUNC` value references an existing function to use as the state-modifying function. The first argument of the state function must have type `STYPE`. The remaining arguments of the state function must match the types of the user-defined aggregate arguments. The state function is called once for each row, and the value returned by the state function becomes the new state. State is not updated for state functions declared with `RETURNS NULL ON NULL INPUT` and called with `null`. After all rows are processed, the optional `FINALFUNC` is executed with last state value as its argument. It must take only one argument with type `STYPE`, but the return type of the `FINALFUNC` may be a different type. A final function declared with `RETURNS NULL ON NULL INPUT` means that the aggregate's return value will be `null`, if the last state is `null`.

If no `FINALFUNC` is defined, the overall return type of the aggregate function is `STYPE`. If a `FINALFUNC` is defined, it is the return type of that function.

DROP AGGREGATE statement

Dropping an user-defined aggregate function uses the `DROP AGGREGATE` statement:

```
drop_aggregate_statement::= DROP AGGREGATE [ IF EXISTS ] function_name[ '('
arguments_signature ')'
]
```

For instance:

```
DROP AGGREGATE myAggregate;
DROP AGGREGATE myKeyspace.anAggregate;
DROP AGGREGATE someAggregate ( int );
DROP AGGREGATE someAggregate ( text );
```

The `DROP AGGREGATE` statement removes an aggregate created using `CREATE AGGREGATE`. You must specify the argument types of the aggregate to drop if there are multiple overloaded aggregates with the same name but a different signature.

The `DROP AGGREGATE` command with the optional `IF EXISTS` keywords drops an aggregate if it exists, and does nothing if a function with the signature does not exist.

Triggers

Triggers

Triggers are identified with a name defined by:

```
trigger_name ::= identifier
```

CREATE TRIGGER

Creating a new trigger uses the `CREATE TRIGGER` statement:

```
create_trigger_statement ::= CREATE TRIGGER [ IF NOT EXISTS ] trigger_name
    ON table_name
    USING string
```

For instance:

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

The actual logic that makes up the trigger can be written in any Java (JVM) language and exists outside the database. You place the trigger code in a `lib/triggers` subdirectory of the Cassandra installation directory, it loads during cluster startup, and exists on every node that participates in a cluster. The trigger defined on a table fires before a requested DML statement occurs, which ensures the atomicity of the transaction.

DROP TRIGGER

Dropping a trigger uses the `DROP TRIGGER` statement:

```
drop_trigger_statement ::= DROP TRIGGER [ IF EXISTS ] trigger_nameON table_name
```

For instance:

```
DROP TRIGGER myTrigger ON myTable;
```

Creating columns with a single value (static column)

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:

```
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
);
```

CQL

```
SELECT *
FROM cycling.country_flag;
```

Result

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

(4 rows)
```

IMPORTANT

The following restrictions apply:

- 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 do **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.

Using a counter

Using a counter

A counter is a special column for storing a number that is updated by increments or decrements.

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

Procedure

1. Create a table for the counter column.

```
CREATE TABLE IF NOT EXISTS cycling.popular_count (
  id UUID PRIMARY KEY,
  popularity counter
);
```

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

```
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;
```

3. The popularity column has a value of 64.

CQL

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

Result

```
select_all_from_popular_count.result : file is corrupt
```

Additional increments or decrements changes the value of the counter column.

Good use of BATCH statement

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 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)
);
```

Note that `balance` is `STATIC`.

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

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.

```
BEGIN BATCH

  INSERT INTO cycling.cyclist_expenses (
    cyclist_name, balance
  ) VALUES (
    'Vera ADRIAN', 0
  ) IF NOT EXISTS;

  INSERT INTO cycling.cyclist_expenses (
    cyclist_name, expense_id, amount, description, paid
  ) VALUES (
    'Vera ADRIAN', 1, 7.95, 'Breakfast', false
  );

APPLY BATCH;
```

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

NOTE

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

+

```
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.

```
cyclist_name | expense_id | balance | amount | description | paid
--------------+------------+---------+--------+-------------+-------
  Vera ADRIAN |          1 |       0 |   7.95 |   Breakfast | False

(1 rows)
```

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

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

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

(1 rows)
```

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

```
BEGIN BATCH

  INSERT INTO cycling.cyclist_expenses (
    cyclist_name, expense_id, amount, description, paid
  ) VALUES (
    'Vera ADRIAN', 2, 13.44, 'Lunch', true
  );

  INSERT INTO cycling.cyclist_expenses (
    cyclist_name, expense_id, amount, description, paid
  ) VALUES (
    'Vera ADRIAN', 3, 25.00, 'Dinner', false
  );

  UPDATE cycling.cyclist_expenses
  SET balance = -32.95
  WHERE cyclist_name = 'Vera ADRIAN'
  IF balance = -7.95;

APPLY BATCH;
```

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

```
(3 rows)
```

- 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

(3 rows)
```

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

Multiple partition logged batch

- Another example is using `BATCH` 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:

```
BEGIN BATCH

  INSERT INTO cycling.cyclist_expenses (
    cyclist_name, expense_id, amount, description, paid
  ) VALUES (
    'John SMITH', 3, 15.00, 'Lunch', false
  );

  INSERT INTO cycling.cyclist_name (
    id, lastname, firstname
  ) VALUES (
    6ab09bec-e68e-48d9-a5f8-97e6fb4c9b12, 'SMITH', 'John'
  );

APPLY BATCH;
```

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

Collection data types overview

Collection data types overview

Apache Cassandra provides collection data 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.

Apache Cassandra avoids joins between two tables by storing the user's email addresses in a collection column in the user table. Each collection specifies the data type of the data held.

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

Creating collections

Creating collections

CQL defines the following collection data types:

- **set: store unordered items**
- **list: store ordered items**
- **map: store key-value pairs**

Collections are intended for insertion and retrieval as a collection. Thus, **Apache Cassandra** reads collections in their entirety, and can affect retrieval performance. A good rule of thumb is that sets are more performant than lists, so use a set if you can. Collections should be smaller than the following maximum size to prevent querying delays.

Guard rails for non-frozen collections:

- No more than 2 billion items in a collection.
- Maximum size of an item in a `set` is 65,535 bytes.
- Maximum size of an item in a `list` or `map` is 2 GB.
- Maximum number of keys in a `map` is 65,535.
- Lists can incur a read-before-write operation for some insertions.

IMPORTANT Collections are not paged internally.

Collections cannot be *sliced*;

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

Using list type

Using list type

A list is similar to a set; it 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.

Prerequisite

- **Keyspace** must exist

In the following example, a list called events stores all the race events on an upcoming calendar. The table is called upcoming_calendar. Each event listed in the list will have a text data type. Events can have several events occurring in a particular month and year, so duplicates can occur. The list can be ordered so that the races appear in the order that they will take place, rather than alphabetical order.

CQL

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

Result

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

(2 rows)
```

Using map type

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.

Prerequisite

- **Keyspace** must exist

In the following example, each team listed in the `map` called `teams` will have a `year` of integer type and a `team name` of text type. The table is named `cyclist_teams`. The map collection is specified with a map column name and the pair of data types enclosed in angle brackets.

CQL

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

Result

```
id                                   | firstname | lastname | teams
--------------------------------------+-----------+----------+
------------------------------------------------------------------------
---------
 5b6962dd-3f90-4c93-8f61-eabfa4a803e2 |  Marianne |      VOS | {2014: 'Rabobank-
Liv Woman Cycling Team', 2015: 'Rabobank-Liv Woman Cyclin
g Team'}

(1 rows)
```

set column

set column

A set consists of a unordered 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.

Prerequisite

- **Keyspace** must exist

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

CQL

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

Result

```
id                                   | lastname        | teams
--------------------------------------+-----------------+
--------------------------------------------------------------------------------
----------------------
 cb07baad-eac8-4f65-b28a-bddc06a0de23 |      ARMITSTEAD |                    {'AA
Drink - Leontien.nl', 'Boels-Dolmans Cycling Team', 'Te
am Garmin - Cervelo'}
 5b6962dd-3f90-4c93-8f61-eabfa4a803e2 |             VOS | {'Nederland bloeit',
'Rabobank Women Team', 'Rabobank-Liv Giant', 'Rabobank-Liv
 Woman Cycling Team'}
 1c9ebc13-1eab-4ad5-be87-dce433216d40 |           BRAND |   {'AA Drink -
Leontien.nl', 'Leontien.nl', 'Rabobank-Liv Giant', 'Rabobank-Liv
 Woman Cycling Team'}
 e7cd5752-bc0d-4157-a80f-7523add8dbcd | VAN DER BREGGEN |
{'Rabobank-Liv Woman Cycling Team', 'Sengers Ladies Cycling Tea
m', 'Team Flexpoint'}
```

```
(4 rows)
```

CREATE TABLE

CREATE TABLE

Defines the columns of a new table.

Apache Cassandra supports creating 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**.

See also: ALTER TABLE, **DROP TABLE**, **CREATE CUSTOM INDEX** for Storage-Attached Indexes (SAI), **CREATE INDEX** for secondary indexes (2i)

Syntax

BNF definition:

```
create_table_statement::= CREATE TABLE [ IF NOT EXISTS ] table_name '('
    column_definition  ( ',' column_definition )*
    [ ',' PRIMARY KEY '(' primary_key ')' ]
     ')' [ WITH table_options ]
column_definition::= column_name cql_type [ STATIC ] [ column_mask ] [ PRIMARY KEY]
column_mask::= MASKED WITH ( DEFAULT | function_name '(' term ( ',' term )* ')' )
primary_key::= partition_key [ ',' clustering_columns ]
partition_key::= column_name  | '(' column_name ( ',' column_name )* ')'
clustering_columns::= column_name ( ',' column_name )*
table_options::= COMPACT STORAGE [ AND table_options ]
    | CLUSTERING ORDER BY '(' clustering_order ')'
    [ AND table_options ]  | options
clustering_order::= column_name (ASC | DESC) ( ',' column_name (ASC | DESC) )*
```

```
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> (ASC | DESC) ] ]
  [ [ AND ] ID = '<table_hash_tag>' ] ] ;
```

▼ *Syntax legend*

Table 1. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
`< >`	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.
`<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.

Required parameters

table_name

Name of the table to index.

column_name

Name of the column.

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] [, ...]`

Restriction:

- A table must have at least one `PRIMARY KEY`.
- When `PRIMARY KEY` is at the end of a column definition, that column is the only primary key for the table, and is defined as the partition-key[partition 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.

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] [, ...]`

Restriction:

- A table must have at least one `PRIMARY KEY`.
- When `PRIMARY KEY` is at the end of a column definition, that column is the only primary key for the table, and is defined as the partition-key[partition 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.

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.

NOTE 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`.

```
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 use to enable the largest possible bloom filter (uses the most memory) and `1.0` is the maximum value disabling the bloom filter.

TIP 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

comment = **'some text that describes the table'**

Provide documentation on the table.

TIP Enter a description of the types of queries the table was designed to satisfy.

default_time_to_live

TTL (Time To Live) in seconds, where zero is disabled. The maximum configurable value is `630720000` (20 years). Beginning in 2018, the expiration timestamp can exceed the maximum value supported by the storage engine; see the warning below. If the value is greater than zero, TTL is enabled for the entire table and an expiration timestamp is added to each column. A new TTL timestamp is calculated each time the data is updated and the row is removed after all the data expires.

Default value: `0` (disabled).

WARNING	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.

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.

NOTE	Tombstoned records within the grace period are excluded from **hints** or **batched mutations**.

In a single-node cluster, this property can safely be set to zero. You can also reduce this value for tables whose data is not explicitly deleted — for example, tables containing only data with TTL set, or tables with `default_time_to_live` set. However, if you lower the `gc_grace_seconds` value, consider its interaction with these operations:

- **hint replays**: When a node goes down and then comes back up, other nodes replay the write operations (called **hints**) that are queued for that node while it was unresponsive. The database does not replay hints older than gc_grace_seconds after creation. The **max_hint_window** 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 configuration/cass_yaml_file.html#batchlog_replay_throttle[batchlog_replay_throttle] 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, **Apache Cassandra** decreases the index summaries for the coldest SSTables to the maximum set by max_index_interval. The max_index_interval is the sparsest possible sampling in relation to memory pressure.

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.

Unresolved directive in create-table.adoc - include::partial$compaction-strategies.adoc[]

compression = { compression_map }

Configure the `compression_map` by specifying the compression algorithm `class` followed by the subproperties in simple JSON format.

TIP Implement custom compression classes using the `org.apache.cassandra.io.compress.ICompressor` interface.

```
compression = {
   ['class' : '<compression_algorithm_name>',
     'chunk_length_in_kb' : '<value>',
     'crc_check_chance' : '<value>',]
   | 'sstable_compression' : '']
}
```

class

Sets the compressor name. **Apache Cassandra** provides the following built-in classes:

Compression Algorithm	Cassandra Class	Compression	Decompression	Ratio	C* Version
LZ4	`LZ4Compressor`	A+	A+	C+	`>=1.2.2`
LZ4HC	`LZ4Compressor`	C+	A+	B+	`>= 3.6`
Zstd	`ZstdCompressor`	A-	A-	A+	`>= 4.0`
Snappy	`SnappyCompressor`	A-	A	C	`>= 1.0`
Deflate (zlib)	`DeflateCompressor`	C	C	A	`>= 1.0`

IMPORTANT Use only compression implementations bundled with **Apache Cassandra**.

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:

```
compaction = {
     'class' : '<compaction_strategy_name>',
     '<property_name>' : <value> [, ...] }
```

where the <compaction_strategy_name> is **SizeTieredCompactionStrategy**, **TimeWindowCompactionStrategy**, or **LeveledCompactionStrategy**.

IMPORTANT Use only compaction implementations bundled with **Apache Cassandra**. See **Compaction** for more details.

Common properties

The following properties apply to all compaction strategies.

```
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>,
     'unchecked_tombstone_compaction' : (true | false),
     'min_threshold' : <num_sstables>,
     'max_threshold' : <num_sstables> }
```

enabled

Enable background compaction.

- `true` runs minor compactions.
- `false` disables minor compactions.

TIP 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 its 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 starts 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 base on an estimated tombstone ratio, this setting makes the minimum interval between two single SSTable compactions tunable to prevent an SSTable from being constantly re-compacted.

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, **Apache Cassandra** 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.

Restriction: Not used in `LeveledCompactionStrategy`.

Default: `4`

max_threshold

The maximum number of SSTables before a minor compaction is triggered.

Restriction: Not used in `LeveledCompactionStrategy`.

Default: `32`

SizeTieredCompactionStrategy

The compaction class `SizeTieredCompactionStrategy` (STCS) triggers a minor compaction when table meets the `min_threshold`. Minor compactions do not involve all the tables in a keyspace. See **SizeTieredCompactionStrategy (STCS)**.

NOTE Default compaction strategy.

The following properties only apply to SizeTieredCompactionStrategy:

```
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 [average-size * bucket_low] and [average-size * 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 [average-size * bucket_low] and [average-size * 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)

NOTE 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)**.

NOTE All of the properties for STCS are also valid for TWCS.

The following properties apply only to TimeWindowCompactionStrategy:

```
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`

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).

NOTE 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 **Apache Cassandra** database does not split the data into two SSTables.

Default: `160`

CAUTION The default value, 160 MB, may be inefficient and negatively impact database indexing and the queries that rely on indexes. For example, consider the benefit of using higher values for sstable_size_in_mb in tables that use (SAI) indexes. For related information, see **Compaction strategies**.

DateTieredCompactionStrategy (deprecated)

IMPORTANT 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)

Apache Cassandra 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`

Optional parameters

Table keywords

CLUSTERING ORDER BY (column_name ASC | DESC)

Order rows storage to make use of the on-disk sorting of columns. Specifying order can make query results more efficient. Options are:

`ASC`: ascending (default order)

`DESC`: descending, reverse order

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.

index_name

Name of the index. Enclose in quotes to use special characters or preserve capitalization. If no name is specified, **Apache Cassandra** names the index: `<table_name>_<column_name>_idx`.

keyspace_name

Name of the keyspace that contains the table to index. If no name is specified, the current keyspace is used.

Usage notes

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.

Indexing with the `CREATE INDEX` command can impact performance. Before creating an index, be aware of when and **when not to create an index**.

Restriction: Indexing counter columns is not supported.

Examples

Create a table with UUID as the primary key

Create the `cyclist_name` table with UUID as the primary key:

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

Create a compound primary key

Create the `cyclist_category` table and store the data in reverse order:

```
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);
```

Create a composite partition key

Create a table that is optimized for query by cyclist rank by year:

```
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)
);
```

Create a table with a vector column

Create a table with a vector column

```
CREATE TABLE IF NOT EXISTS cycling.comments_vs (
  record_id timeuuid,
  id uuid,
  commenter text,
  comment text,
  comment_vector VECTOR <FLOAT, 5>,
  created_at timestamp,
  PRIMARY KEY (id, created_at)
)
WITH CLUSTERING ORDER BY (created_at DESC);
```

Create a table with a frozen UDT

Create the `race_winners` table that has a frozen user-defined type (UDT):

```
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 **Create a user-defined type** for information on Create 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**.

Create a table with a CDC log

Create a change data capture log for the `cyclist_id` table:

```
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.

CAUTION 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.

```
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:

```
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:

```
SELECT id
FROM system_schema.tables
WHERE keyspace_name = 'cycling'
```

```
   AND table_name = 'cyclist_emails';
```

To perform a restoration of the table, see **Backups** for more information.

DROP TABLE

DROP TABLE

Immediate, irreversible removal of a table, including all data contained in the table.

Restriction: 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})"`

See also: CREATE TABLE, **ALTER TABLE**, **CREATE CUSTOM INDEX** for Storage-Attached Indexes (SAI), **CREATE INDEX** for secondary indexes (2i)

Syntax

BNF definition:

```
drop_table_statement::= DROP TABLE [ IF EXISTS ] table_name
```

```
DROP TABLE [ IF EXISTS ] [<keyspace_name>.]<table_name> ;
```

▼ *Syntax legend*

Table 1. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
`< >`	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.

Syntax conventions	Description
`...`	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.
`<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.

Required parameters

table_name

Name of the table to drop.

Optional parameters

keyspace_name

Name of the keyspace that contains the table to drop. If no name is specified, the current keyspace is used.

Examples

Drop the `cyclist_name` table:

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

CREATE INDEX

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. **Apache Cassandra** supports creating an index on most columns, including the partition and cluster columns of a PRIMARY KEY, collections, and static columns. Indexing via this `CREATE INDEX` command can impact performance. Before creating an index, be aware of when and **when not to create an index**.

Use **CREATE CUSTOM INDEX** for Storage-Attached Indexes (SAI).

Restriction: 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>) ;
```

▼ *Syntax legend*

include:cassandra:partial$cql-syntax-legend.adoc[]

index_name

Optional identifier for index. If no name is specified, DataStax Enterprise names the index: `<table_name>_<column_name>_idx`. Enclose in quotes to use special characters or preserve capitalization.

Examples

Creating an index on a clustering column

Define a table having a **composite partition key**, and then create an index on a clustering column.

```
CREATE TABLE 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)
);
```

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

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_keys_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
--------------------------------------+-----------+------------+
------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------
 cb07baad-eac8-4f65-b28a-bddc06a0de23 | Elizabeth | ARMITSTEAD | {2011: 'Team Garmin -
Cervelo', 2012: 'AA Drink - Leontien.nl', 2013: 'Boels:Dolmans Cycling Team', 2014:
'Boels:Dolmans Cycling Team', 2015: 'Boels:Dolmans Cycling Team'}
 5b6962dd-3f90-4c93-8f61-eabfa4a803e2 |  Marianne |        VOS |
{2015: 'Rabobank-Liv Woman Cycling Team'}

(2 rows)
```

Creating an index on map entries

You can create an index on map entries. An `ENTRIES` index can be created only on a map column of a table that doesn't have an existing index.

To index collection entries, use the `ENTRIES` keyword and map name in nested parentheses:

```
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.

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

The example finds cyclists who are the same age.

```
cyclist_name     | blist
------------------+----------------------------------------------------------
   Claudio HEINEN | {'age': '23', 'bday': '27/07/1992', 'nation': 'GERMANY'}
 Laurence BOURQUE |  {'age': '23', 'bday': '27/07/1992', 'nation': 'CANADA'}
```

```
(2 rows)
```

Use the same index to find cyclists from the same country.

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

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

(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:

```
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.

```
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)
```

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]

(1 rows)
```

CREATE CUSTOM INDEX

CREATE CUSTOM INDEX

Cassandra 5.0 is the only supported database currently.

Creates a Storage-Attached Indexing (SAI) index. You can create multiple secondary indexes on the same database table, with each SAI index based on any column in the table. All column date types except the following are supported for SAI indexes:

- `counter`
- non-frozen user-defined type (UDT)

One exception

You cannot define an SAI index based on the partition key when it's comprised of only one column. If you attempt to create an SAI index in this case, SAI issues an error message.

However,you can define an SAI index on one of the columns in a table's composite partition key, i.e., a partition key comprised of multiple columns. If you need to query based on one of those columns, an SAI index is a helpful option. In fact, you can define an SAI index on each column in a composite partition key, if needed.

Defining one or more SAI indexes based on any column in a database table (with the rules noted above) subsequently gives you the ability to run performant queries that use the indexed column to filter results.

See the **SAI section**.

Syntax

BNF definition:

```
index_name::= re('[a-zA-Z_0-9]+')
```

```
CREATE CUSTOM INDEX [ IF NOT EXISTS ] [ <index_name> ]
  ON [ <keyspace_name>.]<table_name> (<column_name>)
    | [ (KEYS(<map_name>)) ]
    | [ (VALUES(<map_name>)) ]
    | [ (ENTRIES(<map_name>)) ]
  USING 'StorageAttachedIndex'
  [ WITH OPTIONS = { <option_map> } ] ;
```

▼ *Syntax legend*

Table 1. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
`< >`	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.
`<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.

index_name

Optional identifier for index. If no name is specified, the default used is `<table_name>\_<column_name>\_idx`. Enclose in quotes to use special characters or to preserve capitalization.

column_name

The name of the table column on which the SAI index is being defined. SAI allows only alphanumeric characters and underscores in names. SAI returns `InvalidRequestException` if you try to define an index on a column name that contains other characters, and does not create the index.

map_name

Used with **collections**, identifier of the `map_name` specified in `CREATE TABLE` … `map(<map_name>)`. The regular column syntax applies for collection types `list` and `set`.

option_map

Define options in JSON simple format.

Option	Description
`case_sensitive`	Ignore case in matching string values. Default: `true`.
`normalize`	When set to `true`, perform Unicode normalization on indexed strings. SAI supports Normalization Form C (NFC) Unicode. When set to `true`, SAI normalizes the different versions of a given Unicode character to a single version, retaining all the marks and symbols in the index. For example, SAI would change the character Å (U+212B) to Å (U+00C5). When implementations keep strings in a normalized form, equivalent strings have a unique binary representation. See Unicode Standard Annex #15, Unicode Normalization Forms. Default: `false`.
`ascii`	When set to `true`, SAI converts alphabetic, numeric, and symbolic characters that are not in the Basic Latin Unicode block (the first 127 ASCII characters) to the ASCII equivalent, if one exists. For example, this option changes à to a. Default: `false`.
similarity_function	Vector search relies on computing the similarity or distance between vectors to identify relevant matches. The similarity function is used to compute the similarity between two vectors. Valid options are: EUCLIDEAN, DOT_PRODUCT, COSINE Default: `COSINE`

Query operators

SAI supports the following query operators for tables with SAI indexes:

- Numerics: `=`, `<`, `>`, `⇐`, `>=`, `AND`, `OR`, `IN`
- Strings: `=`, `CONTAINS`, `CONTAINS KEY`, `AND`, `OR`, `IN`

SAI does not supports the following query operators for tables with SAI indexes:

- Strings or Numerics: `LIKE`

Examples

These examples define SAI indexes for the `cycling.cyclist_semi_pro` table, which is demonstrated in the **SAI quickstart**.

```
CREATE INDEX lastname_sai_idx ON cycling.cyclist_semi_pro (lastname)
USING 'sai'
WITH OPTIONS = {'case_sensitive': 'false', 'normalize': 'true', 'ascii': 'true'};

CREATE INDEX age_sai_idx ON cycling.cyclist_semi_pro (age)
USING 'sai';

CREATE INDEX country_sai_idx ON cycling.cyclist_semi_pro (country)
USING 'sai'
WITH OPTIONS = {'case_sensitive': 'false', 'normalize': 'true', 'ascii': 'true'};

CREATE INDEX registration_sai_idx ON cycling.cyclist_semi_pro (registration)
USING 'sai';
```

For sample queries that find data in `cycling.cyclist_semi_pro` via these sample SAI indexes, see **Submit CQL queries**. Also refer **Querying with SAI**.

SAI collection map examples with keys, values, and entries

The following examples demonstrate using collection maps of multiple types (`keys`, `values`, `entries`) in SAI indexes. For related information, see **Creating collections** and **Using map type**.

Also refer to the SAI collection examples of type **list and set** in this topic.

First, create the keyspace:

```
CREATE KEYSPACE demo3 WITH REPLICATION =
       {'class': 'SimpleStrategy', 'replication_factor': '1'};
```

Next, use the keyspace:

```
USE demo3;
```

Create an `audit` table, with a collection map named `text_map`:

```
CREATE TABLE audit ( id int PRIMARY KEY , text_map map<text, text>);
```

Create multiple SAI indexes on the same `map` column, each using `KEYS`, `VALUES`, and `ENTRIES`.

```
CREATE CUSTOM INDEX ON audit (KEYS(text_map)) USING 'StorageAttachedIndex';
CREATE CUSTOM INDEX ON audit (VALUES(text_map)) USING 'StorageAttachedIndex';
CREATE CUSTOM INDEX ON audit (ENTRIES(text_map)) USING 'StorageAttachedIndex';
```

Insert some data:

```
INSERT INTO audit (id, text_map) values (1, {'Carlos':'Perotti', 'Marcel':'Silva'});
INSERT INTO audit (id, text_map) values (2, {'Giovani':'Pasi', 'Frances':'Giardello'});
INSERT INTO audit (id, text_map) values (3, {'Mark':'Pastore', 'Irene':'Cantona'});
```

Query all data:

Query

```
SELECT * FROM audit;
```

Result

```
 id | text_map
----+---------------------------------------------
  1 | {'Carlos': 'Perotti', 'Marcel': 'Silva'}
  2 | {'Frances': 'Giardello', 'Giovani': 'Pasi'}
  3 | {'Irene': 'Cantona', 'Mark': 'Pastore'}

(3 rows)
```

Query using the SAI index for specific entries in the `map` column:

Query

```
SELECT * FROM audit WHERE text_map['Irene'] = 'Cantona' AND text_map['Mark'] =
'Pastore';
```

Result

```
 id | text_map
----+------------------------------------------
  3 | {'Irene': 'Cantona', 'Mark': 'Pastore'}

(1 rows)
```

Query using the SAI index for specific keys in the `map` column using `CONTAINS KEY`:

Query

```
SELECT * FROM audit WHERE text_map CONTAINS KEY 'Giovani';
```

Result

```
 id | text_map
----+----------------------------------------------
  2 | {'Frances': 'Giardello', 'Giovani': 'Pasi'}

(1 rows)
```

Query using the SAI index for specific values in the `map` column with `CONTAINS`:

Query

```
SELECT * FROM audit WHERE text_map CONTAINS 'Silva';
```

Result

```
 id | text_map
----+-------------------------------------------
  1 | {'Carlos': 'Perotti', 'Marcel': 'Silva'}
```

```
(1 rows)
```

Remember that in CQL queries using SAI indexes, the `CONTAINS` clauses are supported with, and specific to:

- SAI **collection maps** with `keys`, `values`, and `entries`
- SAI **collections** with `list` and `set` types

SAI collection examples with list and set types

These examples demonstrate using collections with the `list` and `set` types in SAI indexes. For related information, see:

- **Creating collections**
- **Using list type**
- **Using set type**

If you have not already, create the keyspace.

```
CREATE KEYSPACE IF NOT EXISTS demo3 WITH REPLICATION =
       {'class': 'SimpleStrategy', 'replication_factor': '1'};
```

```
USE demo3;
```

Using the list type

Create a `calendar` table with a collection of type `list`.

```
CREATE TABLE calendar (key int PRIMARY KEY, years list<int>);
```

Create an SAI index using the collection's `years` column.

```
CREATE CUSTOM INDEX ON calendar(years) USING 'StorageAttachedIndex';
```

Insert some random `int` list data for `years`, just for demo purposes.

TIP Notice the `INSERT` command's square brackets syntax for list values.

```
INSERT INTO calendar (key, years) VALUES (0, [1990,1996]);
INSERT INTO calendar (key, years) VALUES (1, [2000,2010]);
INSERT INTO calendar (key, years) VALUES (2, [2001,1990]);
```

Query with CONTAINS example:

Query

```
SELECT * FROM calendar WHERE years CONTAINS 1990;
```

Result

```
 key | years
-----+--------------
   0 | [1990, 1996]
   2 | [2001, 1990]
(2 rows)
```

This example created the calendar table with years list<int>. Of course, you could have created the table with years list<text>, for example, inserted 'string' values, and queried on the strings.

Using the set type

Now create a calendar2 table with a collection of type set.

```
CREATE TABLE calendar2 (key int PRIMARY KEY, years set<int>);
```

Create an SAI index using the collection's years column — this time for the calendar2 table.

```
CREATE CUSTOM INDEX ON calendar2(years) USING 'StorageAttachedIndex';
```

Insert some random int set data for years, again just for demo purposes.

TIP

Notice the INSERT command's curly braces syntax for set values.

```
INSERT INTO calendar2 (key, years) VALUES (0, {1990,1996});
INSERT INTO calendar2 (key, years) VALUES (1, {2000,2010});
INSERT INTO calendar2 (key, years) VALUES (2, {2001,1990,2020});
```

Query with `CONTAINS` example from the list:

Query

```
SELECT * FROM calendar2  WHERE years CONTAINS 1990;
```

Result

```
 key | years
-----+--------------------
   0 |       {1990, 1996}
   2 | {1990, 2001, 2020}

(2 rows)
```

DROP INDEX

DROP INDEX

Removes an existing index. The default index name is `table_name_column_name_idx`.

Syntax

BNF definition:

```
drop_index_statement::= DROP INDEX [ IF EXISTS ] index_name
```

```
DROP INDEX [ IF EXISTS ] [<keyspace_name>.]<index_name> ;
```

▼ *Syntax legend*

Table 1. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
`< >`	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.

Syntax conventions	Description
`<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.

Example

Drop the index `rank_idx` from the `cycling.rank_by_year_and_name` table.

```
DROP INDEX IF EXISTS cycling.rank_idx;
```

CQL Commands

CQL Commands

This section describes the Cassandra Query Language (CQL) commands supported by the **Apache Cassandra** database.

ALTER KEYSPACE
: Changes keyspace replication strategy and enables or disables commit log.

ALTER MATERIALIZED VIEW
: Changes the table properties of a materialized view.

ALTER ROLE
: Changes password and sets superuser or login options.

ALTER TABLE
: Modifies the columns and properties of a table, or modify

ALTER TYPE
: Modifies an existing user-defined type (UDT).

ALTER USER (Deprecated)
: Deprecated. Alter existing user options.

BATCH
: Applies multiple data modification language (DML) statements with atomicity and/or in isolation.

CREATE AGGREGATE
: Defines a user-defined aggregate.

CREATE CUSTOM INDEX
: Creates a storage-attached index.

CREATE FUNCTION
: Creates custom function to execute user provided code.

CREATE INDEX
: Defines a new index for a single column of a table.

CREATE KEYSPACE

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

CREATE ROLE

Creates a cluster wide database object used for access control.

CREATE TABLE

Creates a new table.

CREATE TYPE

Creates a custom data type in the keyspace that contains one or more fields of related information.

CREATE USER (Deprecated)

Deprecated. Creates a new user.

DELETE

Removes data from one or more columns or removes the entire row

DROP AGGREGATE

Deletes a user-defined aggregate from a keyspace.

DROP FUNCTION

Deletes a user-defined function (UDF) from a keyspace.

DROP INDEX

Removes an index from a table.

DROP KEYSPACE

Removes the keyspace.

DROP MATERIALIZED VIEW

Removes the named materialized view.

DROP ROLE

Removes a role.

DROP TABLE

Removes the table.

DROP TYPE

Drop a user-defined type.

DROP USER (Deprecated)

Removes a user.

GRANT

Allow access to database resources.

INSERT
: Inserts an entire row or upserts data into existing rows.

LIST PERMISSIONS
: Lists permissions on resources.

LIST ROLES
: Lists roles and shows superuser and login status.

LIST USERS (Deprecated)
: Lists existing internal authentication users and their superuser status.

RESTRICT
: Denies the permission on a resource, even if the role is directly granted or inherits permissions.

RESTRICT ROWS
: Configures the column used for row-level access control.

REVOKE
: Removes privileges on database objects from roles.

SELECT
: Returns data from a table.

TRUNCATE
: Removes all data from a table.

UNRESTRICT
: Removes a restriction from a role.

UNRESTRICT ROWS
: Removes the column definition for row-level access control.

UPDATE
: Modifies one or more column values to a row in a table.

USE
: Selects the keyspace for the current client session.