The Cassandra Query Language (CQL)

The Cassandra Query Language (CQL)

This document describes the Cassandra Query Language (CQL) version 3. Note that this document describes the last version of the language. However, the **changes** section provides the differences between the versions of CQL since version 3.0.

CQL offers a model similar to SQL. The data is stored in **tables** containing **rows** of **columns**. For that reason, when used in this document, these terms (tables, rows and columns) have the same definition that they have in SQL.

Definitions

Definitions

Conventions

To aid in specifying the CQL syntax, we will use the following conventions in this document:

- Language rules will be given in an informal BNF variant notation. In particular, we'll use square brakets (`[ item ]`) for optional items, `*` and `` ` for repeated items (where ` `` imply at least one).
- The grammar will also use the following convention for convenience: non-terminal term will be lowercase (and link to their definition) while terminal keywords will be provided "all caps". Note however that keywords are `identifiers` and are thus case insensitive in practice. We will also define some early construction using regexp, which we'll indicate with `re(<some regular expression>)`.
- The grammar is provided for documentation purposes and leave some minor details out. For instance, the comma on the last column definition in a `CREATE TABLE` statement is optional but supported if present even though the grammar in this document suggests otherwise. Also, not everything accepted by the grammar is necessarily valid CQL.
- References to keywords or pieces of CQL code in running text will be shown in a `fixed-width font`.

Identifiers and keywords

The CQL language uses *identifiers* (or *names*) to identify tables, columns and other objects. An identifier is a token matching the regular expression `[a-zA-Z][a-zA-Z0-9_]*`.

A number of such identifiers, like `SELECT` or `WITH`, are *keywords*. They have a fixed meaning for the language and most are reserved. The list of those keywords can be found in **Appendix A**.

Identifiers and (unquoted) keywords are case insensitive. Thus `SELECT` is the same than `select` or `sElEcT`, and `myId` is the same than `myid` or `MYID`. A convention often used (in particular by the samples of this documentation) is to use uppercase for keywords and lowercase for other identifiers.

There is a second kind of identifier called a *quoted identifier* defined by enclosing an arbitrary sequence of characters (non-empty) in double-quotes(`"`). Quoted identifiers are never keywords. Thus `"select"` is not a reserved keyword and can be used to refer to a column (note that using this is particularly ill-advised), while `select` would raise a parsing error. Also, unlike unquoted identifiers and keywords, quoted identifiers are case sensitive (`"My Quoted Id"` is *different* from `"my quoted id"`). A fully lowercase quoted identifier that matches `[a-zA-Z][a-zA-Z0-9_]*` is however *equivalent* to the unquoted identifier obtained by removing the double-quote (so `"myid"` is equivalent to `myid` and to `myId` but different from `"myId"`). Inside a quoted identifier, the double-quote character can be repeated to escape it, so `"foo "" bar"` is a valid identifier.

NOTE The *quoted identifier* can declare columns with arbitrary names, and these can

> sometime clash with specific names used by the server. For instance, when using conditional update, the server will respond with a result set containing a special result named `"[applied]"`. If you've declared a column with such a name, this could potentially confuse some tools and should be avoided. In general, unquoted identifiers should be preferred but if you use quoted identifiers, it is strongly advised that you avoid any name enclosed by squared brackets (like `"[applied]"`) and any name that looks like a function call (like `"f(x)"`).

More formally, we have:

```
identifier::= unquoted_identifier | quoted_identifier
unquoted_identifier::= re('[a-zA-Z][link:[a-zA-Z0-9]]*')
quoted_identifier::= '"' (any character where " can appear if doubled)+ '"'
```

Constants

CQL defines the following *constants*:

```
constant::= string | integer | float | boolean | uuid | blob | NULL
string::= ''' (any character where ' can appear if doubled)+ ''' : '$$' (any character
other than '$$') '$$'
integer::= re('-?[0-9]+')
float::= re('-?[0-9]+(.[0-9]*)?([eE][+-]?[0-9+])?') | NAN | INFINITY
boolean::= TRUE | FALSE
uuid::= hex\{8}-hex\{4}-hex\{4}-hex\{4}-hex\{12}
hex::= re("[0-9a-fA-F]")
blob::= '0' ('x' | 'X') hex+
```

In other words:

- A string constant is an arbitrary sequence of characters enclosed by single-quote(`'`). A single-quote can be included by repeating it, e.g. `'It''s raining today'`. Those are not to be confused with quoted `identifiers` that use double-quotes. Alternatively, a string can be defined by enclosing the arbitrary sequence of characters by two dollar characters, in which case single-quote can be used without escaping (`It's raining today`). That latter form is often used when defining **user-defined functions** to avoid having to escape single-quote characters in function body (as they are more likely to occur than `$$`).
- Integer, float and boolean constant are defined as expected. Note however than float allows the special `NaN` and `Infinity` constants.
- CQL supports UUID constants.
- The content for blobs is provided in hexadecimal and prefixed by `0x`.

- The special `NULL` constant denotes the absence of value.

For how these constants are typed, see the **Data types** section.

Terms

CQL has the notion of a *term*, which denotes the kind of values that CQL support. Terms are defined by:

```
term::= constant | literal | function_call | arithmetic_operation | type_hint |
bind_marker
literal::= collection_literal | vector_literal | udt_literal | tuple_literal
function_call::= identifier '(' [ term (',' term)* ] ')'
arithmetic_operation::= '-' term | term ('+' | '-' | '*' | '/' | '%') term
type_hint::= '(' cql_type ')' term
bind_marker::= '?' | ':' identifier
```

A term is thus one of:

- A **constant**
- A literal for either a **collection**, a **vector**, a **user-defined type** or a **tuple**
- A **function** call, either a **native function** or a **user-defined function**
- An **arithmetic operation** between terms
- A type hint
- A bind marker, which denotes a variable to be bound at execution time. See the section on `prepared-statements` for details. A bind marker can be either anonymous (`?`) or named (`:some_name`). The latter form provides a more convenient way to refer to the variable for binding it and should generally be preferred.

Comments

A comment in CQL is a line beginning by either double dashes (`--`) or double slash (`//`).

Multi-line comments are also supported through enclosure within `/ and/` (but nesting is not supported).

```
-- This is a comment
// This is a comment too
/* This is
   a multi-line comment */
```

Statements

CQL consists of statements that can be divided in the following categories:

- `data-definition` statements, to define and change how the data is stored (keyspaces and tables).
- `data-manipulation` statements, for selecting, inserting and deleting data.
- `secondary-indexes` statements.
- `materialized-views` statements.
- `cql-roles` statements.
- `cql-permissions` statements.
- `User-Defined Functions (UDFs)` statements.
- `udts` statements.
- `cql-triggers` statements.

All the statements are listed below and are described in the rest of this documentation (see links above):

```
cql_statement::= statement [ ';' ]
statement:=: ddl_statement :
        | dml_statement
        | secondary_index_statement
        | materialized_view_statement
        | role_or_permission_statement
        | udf_statement
        | udt_statement
        | trigger_statement
ddl_statement::= use_statement
        | create_keyspace_statement
        | alter_keyspace_statement
        | drop_keyspace_statement
        | create_table_statement
        | alter_table_statement
        | drop_table_statement
        | truncate_statement
dml_statement::= select_statement
        | insert_statement
        | update_statement
        | delete_statement
        | batch_statement
secondary_index_statement::= create_index_statement
        | drop_index_statement
materialized_view_statement::= create_materialized_view_statement
        | drop_materialized_view_statement
```

```
role_or_permission_statement::= create_role_statement
        | alter_role_statement
        | drop_role_statement
        | grant_role_statement
        | revoke_role_statement
        | list_roles_statement
        | grant_permission_statement
        | revoke_permission_statement
        | list_permissions_statement
        | create_user_statement
        | alter_user_statement
        | drop_user_statement
        | list_users_statement
udf_statement::= create_function_statement
        | drop_function_statement
        | create_aggregate_statement
        | drop_aggregate_statement
udt_statement::= create_type_statement
        | alter_type_statement
        | drop_type_statement
trigger_statement::= create_trigger_statement
        | drop_trigger_statement
```

Prepared Statements

CQL supports *prepared statements*. Prepared statements are an optimization that allows to parse a query only once but execute it multiple times with different concrete values.

Any statement that uses at least one bind marker (see `bind_marker`) will need to be *prepared*. After which the statement can be *executed* by provided concrete values for each of its marker. The exact details of how a statement is prepared and then executed depends on the CQL driver used and you should refer to your driver documentation.

Data Types

Data Types

CQL is a typed language and supports a rich set of data types, including **native types**, **collection types**, **user-defined types**, **tuple types**, and [custom types]:

```
cql_type::= native_type| collection_type| user_defined_type | tuple_type | custom_type
```

Native types

The native types supported by CQL are:

```
native_type::= ASCII | BIGINT | BLOB | BOOLEAN | COUNTER | DATE
| DECIMAL | DOUBLE | DURATION | FLOAT | INET | INT |
SMALLINT | TEXT | TIME | TIMESTAMP | TIMEUUID | TINYINT |
UUID | VARCHAR | VARINT | VECTOR
```

The following table gives additional information on the native data types, and on which kind of **constants** each type supports:

Type	Constants supported	Description
`ascii`	`string`	ASCII character string
`bigint`	`integer`	64-bit signed long
`blob`	`blob`	Arbitrary bytes (no validation)
`boolean`	`boolean`	Either `true` or `false`
`counter`	`integer`	Counter column (64-bit signed value). See `counters` for details.
`date`	`integer`, `string`	A date (with no corresponding time value). See `dates` below for details.
`decimal`	`integer`, `float`	Variable-precision decimal
`double`	`integer` `float`	64-bit IEEE-754 floating point
`duration`	`duration`,	A duration with nanosecond precision. See `durations` below for details.
`float`	`integer`, `float`	32-bit IEEE-754 floating point

Type	Constants supported	Description
`inet`	`string`	An IP address, either IPv4 (4 bytes long) or IPv6 (16 bytes long). Note that there is no `inet` constant, IP address should be input as strings.
`int`	`integer`	32-bit signed int
`smallint`	`integer`	16-bit signed int
`text`	`string`	UTF8 encoded string
`time`	`integer`, `string`	A time (with no corresponding date value) with nanosecond precision. See `times` below for details.
`timestamp`	`integer`, `string`	A timestamp (date and time) with millisecond precision. See `timestamps` below for details.
`timeuuid`	`uuid`	Version 1 UUID, generally used as a "conflict-free" timestamp. Also see `timeuuid-functions`.
`tinyint`	`integer`	8-bit signed int
`uuid`	`uuid`	A UUID (of any version)
`varchar`	`string`	UTF8 encoded string
`varint`	`integer`	Arbitrary-precision integer
`vector`	`float`	A fixed length non-null, flattened array of float values CASSANDRA-18504 added this data type to **Cassandra 5.0**.

Counters

The `counter` type is used to define *counter columns*. A counter column is a column whose value is a 64-bit signed integer and on which 2 operations are supported: incrementing and decrementing (see the **UPDATE** statement for syntax). Note that the value of a counter cannot be set: a counter does not exist until first incremented/decremented, and that first increment/decrement is made as if the prior value was 0.

Counters have a number of important limitations:

- They cannot be used for columns part of the `PRIMARY KEY` of a table.

- A table that contains a counter can only contain counters. In other words, either all the columns of a table outside the `PRIMARY KEY` have the `counter` type, or none of them have it.
- Counters do not support **expiration**.
- The deletion of counters is supported, but is only guaranteed to work the first time you delete a counter. In other words, you should not re-update a counter that you have deleted (if you do, proper behavior is not guaranteed).
- Counter updates are, by nature, not idemptotent. An important consequence is that if a counter update fails unexpectedly (timeout or loss of connection to the coordinator node), the client has no way to know if the update has been applied or not. In particular, replaying the update may or may not lead to an over count.

Working with timestamps

Values of the `timestamp` type are encoded as 64-bit signed integers representing a number of milliseconds since the standard base time known as the epoch: January 1 1970 at 00:00:00 GMT.

Timestamps can be input in CQL either using their value as an `integer`, or using a `string` that represents an ISO 8601 date. For instance, all of the values below are valid `timestamp` values for Mar 2, 2011, at 04:05:00 AM, GMT:

- `1299038700000`
- `'2011-02-03 04:05+0000'`
- `'2011-02-03 04:05:00+0000'`
- `'2011-02-03 04:05:00.000+0000'`
- `'2011-02-03T04:05+0000'`
- `'2011-02-03T04:05:00+0000'`
- `'2011-02-03T04:05:00.000+0000'`

The `+0000` above is an RFC 822 4-digit time zone specification; `+0000` refers to GMT. US Pacific Standard Time is `-0800`. The time zone may be omitted if desired (`'2011-02-03 04:05:00'`), and if so, the date will be interpreted as being in the time zone under which the coordinating Cassandra node is configured. There are however difficulties inherent in relying on the time zone configuration being as expected, so it is recommended that the time zone always be specified for timestamps when feasible.

The time of day may also be omitted (`'2011-02-03'` or `'2011-02-03+0000'`), in which case the time of day will default to 00:00:00 in the specified or default time zone. However, if only the date part is relevant, consider using the **date** type.

Date type

Values of the `date` type are encoded as 32-bit unsigned integers representing a number of days with "the epoch" at the center of the range (2^31). Epoch is January 1st, 1970

For **timestamps**, a date can be input either as an `integer` or using a date `string`. In the later case, the format should be `yyyy-mm-dd` (so `'2011-02-03'` for instance).

Time type

Values of the `time` type are encoded as 64-bit signed integers representing the number of nanoseconds since midnight.

For **timestamps**, a time can be input either as an `integer` or using a `string` representing the time. In the later case, the format should be `hh:mm:ss[.fffffffff]` (where the sub-second precision is optional and if provided, can be less than the nanosecond). So for instance, the following are valid inputs for a time:

- `'08:12:54'`
- `'08:12:54.123'`
- `'08:12:54.123456'`
- `'08:12:54.123456789'`

Duration type

Values of the `duration` type are encoded as 3 signed integer of variable lengths. The first integer represents the number of months, the second the number of days and the third the number of nanoseconds. This is due to the fact that the number of days in a month can change, and a day can have 23 or 25 hours depending on the daylight saving. Internally, the number of months and days are decoded as 32 bits integers whereas the number of nanoseconds is decoded as a 64 bits integer.

A duration can be input as:

- `(quantity unit)+` like `12h30m` where the unit can be:
  - `y`: years (12 months)
  - `mo`: months (1 month)
  - `w`: weeks (7 days)
  - `d`: days (1 day)
  - `h`: hours (3,600,000,000,000 nanoseconds)
  - `m`: minutes (60,000,000,000 nanoseconds)

    - `s`: seconds (1,000,000,000 nanoseconds)
    - `ms`: milliseconds (1,000,000 nanoseconds)
    - `us` or `µs` : microseconds (1000 nanoseconds)
    - `ns`: nanoseconds (1 nanosecond)
- ISO 8601 format: `P[n]Y[n]M[n]DT[n]H[n]M[n]S or P[n]W`
- ISO 8601 alternative format: `P[YYYY]-[MM]-[DD]T[hh]:[mm]:[ss]`

For example:

```
INSERT INTO RiderResults (rider, race, result)
   VALUES ('Christopher Froome', 'Tour de France', 89h4m48s);
INSERT INTO RiderResults (rider, race, result)
   VALUES ('BARDET Romain', 'Tour de France', PT89H8M53S);
INSERT INTO RiderResults (rider, race, result)
   VALUES ('QUINTANA Nairo', 'Tour de France', P0000-00-00T89:09:09);
```

Duration columns cannot be used in a table's `PRIMARY KEY`. This limitation is due to the fact that durations cannot be ordered. It is effectively not possible to know if `1mo` is greater than `29d` without a date context.

A `1d` duration is not equal to a `24h` one as the duration type has been created to be able to support daylight saving.

Collections

CQL supports three kinds of collections: `maps`, `sets` and `lists`. The types of those collections is defined by:

```
collection_type::= MAP '<' cql_type',' cql_type'>'
    | SET '<' cql_type '>'
    | LIST '<' cql_type'>'
```

and their values can be inputd using collection literals:

```
collection_literal::= map_literal | set_literal | list_literal
map_literal::= '\{' [ term ':' term (',' term : term)* ] '}'
set_literal::= '\{' [ term (',' term)* ] '}'
list_literal::= '[' [ term (',' term)* ] ']'
```

Note however that neither `bind_marker` nor `NULL` are supported inside collection literals.

Noteworthy characteristics

Collections are meant for storing/denormalizing relatively small amount of data. They work well for things like “the phone numbers of a given user”, “labels applied to an email”, etc. But when items are expected to grow unbounded (“all messages sent by a user”, “events registered by a sensor”…), then collections are not appropriate and a specific table (with clustering columns) should be used. Concretely, (non-frozen) collections have the following noteworthy characteristics and limitations:

- Individual collections are not indexed internally. Which means that even to access a single element of a collection, the whole collection has to be read (and reading one is not paged internally).
- While insertion operations on sets and maps never incur a read-before-write internally, some operations on lists do. Further, some lists operations are not idempotent by nature (see the section on **lists** below for details), making their retry in case of timeout problematic. It is thus advised to prefer sets over lists when possible.

Please note that while some of those limitations may or may not be removed/improved upon in the future, it is a anti-pattern to use a (single) collection to store large amounts of data.

Maps

A `map` is a (sorted) set of key-value pairs, where keys are unique and the map is sorted by its keys. You can define and insert a map with:

```
CREATE TABLE users (
   id text PRIMARY KEY,
   name text,
   favs map<text, text> // A map of text keys, and text values
);

INSERT INTO users (id, name, favs)
   VALUES ('jsmith', 'John Smith', { 'fruit' : 'Apple', 'band' : 'Beatles' });

// Replace the existing map entirely.
UPDATE users SET favs = { 'fruit' : 'Banana' } WHERE id = 'jsmith';
```

Further, maps support:

- Updating or inserting one or more elements:

  ```
  UPDATE users SET favs['author'] = 'Ed Poe' WHERE id = 'jsmith';
  UPDATE users SET favs = favs + { 'movie' : 'Cassablanca', 'band' : 'ZZ Top' } WHERE id
  = 'jsmith';
  ```

- Removing one or more element (if an element doesn't exist, removing it is a no-op but no error is thrown):

```
DELETE favs['author'] FROM users WHERE id = 'jsmith';
UPDATE users SET favs = favs - { 'movie', 'band'} WHERE id = 'jsmith';
```

  Note that for removing multiple elements in a `map`, you remove from it a `set` of keys.

Lastly, TTLs are allowed for both `INSERT` and `UPDATE`, but in both cases the TTL set only apply to the newly inserted/updated elements. In other words:

```
UPDATE users USING TTL 10 SET favs['color'] = 'green' WHERE id = 'jsmith';
```

will only apply the TTL to the `{ 'color' : 'green' }` record, the rest of the map remaining unaffected.

Sets

A `set` is a (sorted) collection of unique values. You can define and insert a set with:

```
CREATE TABLE images (
   name text PRIMARY KEY,
   owner text,
   tags set<text> // A set of text values
);

INSERT INTO images (name, owner, tags)
   VALUES ('cat.jpg', 'jsmith', { 'pet', 'cute' });

// Replace the existing set entirely
UPDATE images SET tags = { 'kitten', 'cat', 'lol' } WHERE name = 'cat.jpg';
```

Further, sets support:

- Adding one or multiple elements (as this is a set, inserting an already existing element is a no-op):

```
UPDATE images SET tags = tags + { 'gray', 'cuddly' } WHERE name = 'cat.jpg';
```

- Removing one or multiple elements (if an element doesn't exist, removing it is a no-op but no error is thrown):

```
UPDATE images SET tags = tags - { 'cat' } WHERE name = 'cat.jpg';
```

Lastly, for **sets**, TTLs are only applied to newly inserted values.

Lists

NOTE	As mentioned above and further discussed at the end of this section, lists have limitations and specific performance considerations that you should take into account before using them. In general, if you can use a **set** instead of list, always prefer a set.

A `list` is a (sorted) collection of non-unique values where elements are ordered by their position in the list. You can define and insert a list with:

```
CREATE TABLE plays (
    id text PRIMARY KEY,
    game text,
    players int,
    scores list<int> // A list of integers
)

INSERT INTO plays (id, game, players, scores)
           VALUES ('123-afde', 'quake', 3, [17, 4, 2]);

// Replace the existing list entirely
UPDATE plays SET scores = [ 3, 9, 4] WHERE id = '123-afde';
```

Further, lists support:

- Appending and prepending values to a list:

  ```
  UPDATE plays SET players = 5, scores = scores + [ 14, 21 ] WHERE id = '123-afde';
  UPDATE plays SET players = 6, scores = [ 3 ] + scores WHERE id = '123-afde';
  ```

WARNING	*Warning* The append and prepend operations are not idempotent by nature. So in particular, if one of these operations times out, then retrying the operation is not safe and it may (or may not) lead to appending/prepending the value twice.

- Setting the value at a particular position in a list that has a pre-existing element for that position. An error will be thrown if the list does not have the position:

  ```
  UPDATE plays SET scores[1] = 7 WHERE id = '123-afde';
  ```

- Removing an element by its position in the list that has a pre-existing element for that position. An

error will be thrown if the list does not have the position. Further, as the operation removes an element from the list, the list size will decrease by one element, shifting the position of all the following elements one forward:

```
DELETE scores[1] FROM plays WHERE id = '123-afde';
```

- Deleting *all* the occurrences of particular values in the list (if a particular element doesn't occur at all in the list, it is simply ignored and no error is thrown):

```
UPDATE plays SET scores = scores - [ 12, 21 ] WHERE id = '123-afde';
```

WARNING	*Warning* Setting and removing an element by position and removing occurences of particular values incur an internal *read-before-write*. These operations will run slowly and use more resources than usual updates (with the exclusion of conditional write that have their own cost).

Lastly, for **lists**, TTLs only apply to newly inserted values.

Working with vectors

Vectors are fixed-size sequences of non-null values of a certain data type. They use the same literals as lists.

You can define, insert and update a vector with:

```
CREATE TABLE plays (
    id text PRIMARY KEY,
    game text,
    players int,
    scores vector<int, 3> // A vector of 3 integers
)

INSERT INTO plays (id, game, players, scores)
           VALUES ('123-afde', 'quake', 3, [17, 4, 2]);

// Replace the existing vector entirely
UPDATE plays SET scores = [ 3, 9, 4] WHERE id = '123-afde';
```

Note that it isn't possible to change the individual values of a vector, and it isn't possible to select individual elements of a vector.

User-Defined Types (UDTs)

CQL support the definition of user-defined types (UDTs). Such a type can be created, modified and removed using the `create_type_statement`, `alter_type_statement` and `drop_type_statement` described below. But once created, a UDT is simply referred to by its name:

```
user_defined_type::= udt_name
udt_name::= [ keyspace_name '.' ] identifier
```

Creating a UDT

Creating a new user-defined type is done using a `CREATE TYPE` statement defined by:

```
create_type_statement::= CREATE TYPE [ IF NOT EXISTS ] udt_name
        '(' field_definition ( ',' field_definition)* ')'
field_definition::= identifier cql_type
```

A UDT has a name (used to declared columns of that type) and is a set of named and typed fields. Fields name can be any type, including collections or other UDT. For instance:

```
CREATE TYPE phone (
    country_code int,
    number text,
);

CREATE TYPE address (
    street text,
    city text,
    zip text,
    phones map<text, phone>
);

CREATE TABLE user (
    name text PRIMARY KEY,
    addresses map<text, frozen<address>>
);
```

Things to keep in mind about UDTs:

- Attempting to create an already existing type will result in an error unless the `IF NOT EXISTS` option is used. If it is used, the statement will be a no-op if the type already exists.
- A type is intrinsically bound to the keyspace in which it is created, and can only be used in that

keyspace. At creation, if the type name is prefixed by a keyspace name, it is created in that keyspace. Otherwise, it is created in the current keyspace.

- As of Cassandra , UDT have to be frozen in most cases, hence the `frozen<address>` in the table definition above.

UDT literals

Once a user-defined type has been created, value can be input using a UDT literal:

```
udt_literal::= '{' identifier ':' term ( ',' identifier ':' term)* '}'
```

In other words, a UDT literal is like a **map`** literal but its keys are the names of the fields of the type. For instance, one could insert into the table define in the previous section using:

```
INSERT INTO user (name, addresses)
   VALUES ('z3 Pr3z1den7', {
     'home' : {
        street: '1600 Pennsylvania Ave NW',
        city: 'Washington',
        zip: '20500',
        phones: { 'cell' : { country_code: 1, number: '202 456-1111' },
                  'landline' : { country_code: 1, number: '...' } }
     },
     'work' : {
        street: '1600 Pennsylvania Ave NW',
        city: 'Washington',
        zip: '20500',
        phones: { 'fax' : { country_code: 1, number: '...' } }
     }
  }
);
```

To be valid, a UDT literal can only include fields defined by the type it is a literal of, but it can omit some fields (these will be set to `NULL`).

Altering a UDT

An existing user-defined type can be modified using an `ALTER TYPE` statement:

```
alter_type_statement::= ALTER TYPE [ IF EXISTS ] udt_name alter_type_modification
alter_type_modification::= ADD [ IF NOT EXISTS ] field_definition
        | RENAME [ IF EXISTS ] identifier TO identifier (AND identifier TO identifier )*
```

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

- Add a new field to the type (`ALTER TYPE address ADD country text`). That new field will be `NULL` for any values of the type created before the addition. If the new field exists, the statement will return an error, unless `IF NOT EXISTS` is used in which case the operation is a no-op.
- Rename the fields of the type. If the field(s) does not exist, the statement will return an error, unless `IF EXISTS` is used in which case the operation is a no-op.

```
ALTER TYPE address RENAME zip TO zipcode;
```

Dropping a UDT

You can drop an existing user-defined type using a `DROP TYPE` statement:

```
drop_type_statement::= DROP TYPE [ IF EXISTS ] udt_name
```

Dropping a type results in the immediate, irreversible removal of that type. However, attempting to drop a type that is still in use by another type, table or function will result in an error.

If the type dropped does not exist, an error will be returned unless `IF EXISTS` is used, in which case the operation is a no-op.

Tuples

CQL also support tuples and tuple types (where the elements can be of different types). Functionally, tuples can be though as anonymous UDT with anonymous fields. Tuple types and tuple literals are defined by:

```
tuple_type::= TUPLE '<' cql_type( ',' cql_type)* '>'
tuple_literal::= '(' term( ',' term )* ')'
```

and can be created:

```
CREATE TABLE durations (
  event text,
  duration tuple<int, text>,
);

INSERT INTO durations (event, duration) VALUES ('ev1', (3, 'hours'));
```

Unlike other composed types, like collections and UDTs, a tuple is always `frozen <frozen>` (without the need of the `frozen` keyword) and it is not possible to update only some elements of a tuple (without updating the whole tuple). Also, a tuple literal should always have the same number of value than declared in the type it is a tuple of (some of those values can be null but they need to be explicitly declared as so).

Custom Types

NOTE	Custom types exists mostly for backward compatibility purposes and their usage is discouraged. Their usage is complex, not user friendly and the other provided types, particularly **user-defined types**, should almost always be enough.

A custom type is defined by:

```
custom_type::= string
```

A custom type is a `string` that contains the name of Java class that extends the server side `AbstractType` class and that can be loaded by Cassandra (it should thus be in the `CLASSPATH` of every node running Cassandra). That class will define what values are valid for the type and how the time sorts when used for a clustering column. For any other purpose, a value of a custom type is the same than that of a `blob`, and can in particular be input using the `blob` literal syntax.

Data Definition

Data Definition

CQL stores data in *tables*, whose schema defines the layout of the data in the table. Tables are located in *keyspaces*. A keyspace defines options that apply to all the keyspace's tables. The **replication strategy** is an important keyspace option, as is the replication factor. A good general rule is one keyspace per application. It is common for a cluster to define only one keyspace for an active application.

This section describes the statements used to create, modify, and remove those keyspace and tables.

Common definitions

The names of the keyspaces and tables are defined by the following grammar:

```
keyspace_name::= name
table_name::= [keyspace_name '.' ] name
name::= unquoted_name | quoted_name
unquoted_name::= re('[a-zA-Z_0-9]\{1, 48}')
quoted_name::= '"' unquoted_name '"'
```

Both keyspace and table name should be comprised of only alphanumeric characters, cannot be empty and are limited in size to 48 characters (that limit exists mostly to avoid filenames (which may include the keyspace and table name) to go over the limits of certain file systems). By default, keyspace and table names are case-insensitive (`myTable` is equivalent to `mytable`) but case sensitivity can be forced by using double-quotes (`"myTable"` is different from `mytable`).

Further, a table is always part of a keyspace and a table name can be provided fully-qualified by the keyspace it is part of. If is is not fully-qualified, the table is assumed to be in the *current* keyspace (see **USE** statement).

Further, the valid names for columns are defined as:

```
column_name::= identifier
```

We also define the notion of statement options for use in the following section:

```
options::= option ( AND option )*
option::= identifier '=' ( identifier
    | constant
    | map_literal )
```

CREATE KEYSPACE

A keyspace is created with a `CREATE KEYSPACE` statement:

```
create_keyspace_statement::= CREATE KEYSPACE [ IF NOT EXISTS ] keyspace_name
    WITH options
```

For example:

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

CREATE KEYSPACE excalibur
   WITH replication = {'class': 'NetworkTopologyStrategy', 'DC1' : 1, 'DC2' : 3}
   AND durable_writes = false;
```

Attempting to create a keyspace that already exists 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 keyspace already exists.

The supported `options` are:

name	kind	mandatory	default	description
`replication`	*map*	yes	n/a	The replication strategy and options to use for the keyspace (see details below).
`durable_writes`	*simple*	no	true	Whether to use the commit log for updates on this keyspace (disable this option at your own risk!).

The `replication` property is mandatory and must contain the `'class'` sub-option that defines the desired **replication strategy** class. The rest of the sub-options depend on which replication strategy is used. By default, Cassandra supports the following `'class'` values:

SimpleStrategy

A simple strategy that defines a replication factor for data to be spread across the entire cluster. This is generally not a wise choice for production, as it does not respect datacenter layouts and can lead to

wildly varying query latency. For production, use `NetworkTopologyStrategy`. `SimpleStrategy` supports a single mandatory argument:

sub-option	type	since	description
`'replication_factor'`	int	all	The number of replicas to store per range

NetworkTopologyStrategy

A production-ready replication strategy that sets the replication factor independently for each data-center. The rest of the sub-options are key-value pairs, with a key set to a data-center name and its value set to the associated replication factor. Options:

sub-option	type	description	`'<datacenter>'`
int	The number of replicas to store per range in the provided datacenter.	`'replication_factor'`	int

When later altering keyspaces and changing the `replication_factor`, auto-expansion will only *add* new datacenters for safety, it will not alter existing datacenters or remove any, even if they are no longer in the cluster. If you want to remove datacenters while setting the `replication_factor`, explicitly zero out the datacenter you want to have zero replicas.

An example of auto-expanding datacenters with two datacenters: `DC1` and `DC2`:

```
CREATE KEYSPACE excalibur
    WITH replication = {'class': 'NetworkTopologyStrategy', 'replication_factor' : 3};

DESCRIBE KEYSPACE excalibur;
```

will result in:

```
CREATE KEYSPACE excalibur WITH replication = {'class': 'NetworkTopologyStrategy', 'DC1':
'3', 'DC2': '3'} AND durable_writes = true;
```

An example of auto-expanding and overriding a datacenter:

```
CREATE KEYSPACE excalibur
   WITH replication = {'class': 'NetworkTopologyStrategy', 'replication_factor' : 3,
'DC2': 2};

DESCRIBE KEYSPACE excalibur;
```

will result in:

```
CREATE KEYSPACE excalibur WITH replication = {'class': 'NetworkTopologyStrategy', 'DC1':
'3', 'DC2': '2'} AND durable_writes = true;
```

An example that excludes a datacenter while using `replication_factor`:

```
CREATE KEYSPACE excalibur
   WITH replication = {'class': 'NetworkTopologyStrategy', 'replication_factor' : 3,
'DC2': 0};

DESCRIBE KEYSPACE excalibur;
```

will result in:

```
CREATE KEYSPACE excalibur WITH replication = {'class': 'NetworkTopologyStrategy', 'DC1':
'3'} AND durable_writes = true;
```

If **transient replication** has been enabled, transient replicas can be configured for both `SimpleStrategy` and `NetworkTopologyStrategy` by defining replication factors in the format `'<total_replicas>/<transient_replicas>'`

For instance, this keyspace will have 3 replicas in DC1, 1 of which is transient, and 5 replicas in DC2, 2 of which are transient:

```
CREATE KEYSPACE some_keyspace
   WITH replication = {'class': 'NetworkTopologyStrategy', 'DC1' : '3/1'', 'DC2' :
'5/2'};
```

USE

The `USE` statement changes the *current* keyspace to the specified keyspace. A number of objects in CQL are bound to a keyspace (tables, user-defined types, functions, etc.) and the current keyspace is the default keyspace used when those objects are referred to in a query without a fully-qualified name (without a prefixed keyspace name). A `USE` statement specifies the keyspace to use as an argument:

```
use_statement::= USE keyspace_name
```

Using CQL:

```
USE excelsior;
```

ALTER KEYSPACE

An `ALTER KEYSPACE` statement modifies the options of a keyspace:

```
alter_keyspace_statement::= ALTER KEYSPACE [ IF EXISTS ] keyspace_name
    WITH options
```

For example:

```
ALTER KEYSPACE excelsior
    WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 4};
```

If the keyspace does not exist, the statement will return an error, unless `IF EXISTS` is used in which case the operation is a no-op. The supported options are the same as for **creating a keyspace**.

DROP KEYSPACE

Dropping a keyspace is done with the `DROP KEYSPACE` statement:

```
drop_keyspace_statement::= DROP KEYSPACE [ IF EXISTS ] keyspace_name
```

For example:

```
DROP KEYSPACE excelsior;
```

Dropping a keyspace results in the immediate, irreversible removal of that keyspace, including all the tables, user-defined types, user-defined functions, and all the data contained in those tables.

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

CREATE TABLE

Creating a new table uses the `CREATE TABLE` statement:

```
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) )*
```

For example, here are some CQL statements to create tables:

```
CREATE TABLE monkey_species (
    species text PRIMARY KEY,
    common_name text,
    population varint,
    average_size int
) WITH comment='Important biological records';

CREATE TABLE timeline (
    userid uuid,
    posted_month int,
    posted_time uuid,
    body text,
    posted_by text,
    PRIMARY KEY (userid, posted_month, posted_time)
) WITH compaction = { 'class' : 'LeveledCompactionStrategy' };

CREATE TABLE loads (
    machine inet,
    cpu int,
    mtime timeuuid,
    load float,
    PRIMARY KEY ((machine, cpu), mtime)
) WITH CLUSTERING ORDER BY (mtime DESC);
```

A CQL table has a name and is composed of a set of *rows*. Creating a table amounts to defining which **columns** each rows will have, which of those columns comprise the **primary key**, as well as defined **options** for the table.

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

Column definitions

Every row in a CQL table will have the predefined columns defined at table creation. Columns can be added later using an **alter statement**.

A `column_definition` is comprised of the name of the column and its **type**, restricting the values that are accepted for that column. Additionally, a column definition can have the following modifiers:

- `STATIC`: declares the column as a **static column**
- `PRIMARY KEY`: declares the column as the sole component of the **primary key** of the table

Static columns

Some columns can be declared as `STATIC` in a table definition. A column that is static will be "shared" by all the rows belonging to the same partition (having the same **partition key**.

For example:

Code

```
CREATE TABLE t (
    pk int,
    t int,
    v text,
    s text static,
    PRIMARY KEY (pk, t)
);
INSERT INTO t (pk, t, v, s) VALUES (0, 0, 'val0', 'static0');
INSERT INTO t (pk, t, v, s) VALUES (0, 1, 'val1', 'static1');
SELECT * FROM t;
```

Results

```
pk | t | v      | s
  ----+---+--------+-----------
   0  | 0 | 'val0' | 'static1'
   0  | 1 | 'val1' | 'static1'
```

As can be seen, the `s` value is the same (`static1`) for both of the rows in the partition (the partition key being `pk`, and both rows are in the same partition): the second insertion overrides the value for `s`.

The use of static columns has the following restrictions:

- A table without clustering columns cannot have static columns. In a table without clustering

columns, every partition has only one row, and so every column is inherently static)

- Only non-primary key columns can be static.

The Primary key

Within a table, a row is uniquely identified by its `PRIMARY KEY`, and hence all tables **must** define a single PRIMARY KEY. A `PRIMARY KEY` is composed of one or more of the defined columns in the table. Syntactically, the primary key is defined with the phrase `PRIMARY KEY` followed by a comma-separated list of the column names within parenthesis. If the primary key has only one column, you can alternatively add the `PRIMARY KEY` phrase to that column in the table definition. The order of the columns in the primary key definition defines the partition key and clustering columns.

A CQL primary key is composed of two parts:

partition key

- It is the first component of the primary key definition. It can be a single column or, using an additional set of parenthesis, can be multiple columns. A table must have at least one partition key, the smallest possible table definition is:

```
CREATE TABLE t (k text PRIMARY KEY);
```

clustering columns

- The columns are the columns that follow the partition key in the primary key definition. The order of those columns define the *clustering order*.

Some examples of primary key definition are:

- `PRIMARY KEY (a)`: `a` is the single partition key and there are no clustering columns
- `PRIMARY KEY (a, b, c)` : `a` is the single partition key and `b` and `c` are the clustering columns
- `PRIMARY KEY ((a, b), c)` : `a` and `b` compose the *composite* partition key and `c` is the clustering column

IMPORTANT	The primary key uniquely identifies a row in the table, as described above. A consequence of this uniqueness is that if another row is inserted using the same primary key, then an `UPSERT` occurs and an existing row with the same primary key is replaced. Columns that are not part of the primary key cannot define uniqueness.

Partition key

Within a table, CQL defines the notion of a *partition* that defines the location of data within a Cassandra cluster. A partition is the set of rows that share the same value for their partition key.

Note that if the partition key is composed of multiple columns, then rows belong to the same partition when they have the same values for all those partition key columns. A hash is computed from the partition key columns and that hash value defines the partition location. So, for instance, given the following table definition and content:

```
CREATE TABLE t (
    a int,
    b int,
    c int,
    d int,
    PRIMARY KEY ((a, b), c, d)
);
INSERT INTO t (a, b, c, d) VALUES (0,0,0,0);
INSERT INTO t (a, b, c, d) VALUES (0,0,1,1);
INSERT INTO t (a, b, c, d) VALUES (0,1,2,2);
INSERT INTO t (a, b, c, d) VALUES (0,1,3,3);
INSERT INTO t (a, b, c, d) VALUES (1,1,4,4);
SELECT * FROM t;
```

will result in

```
a | b | c | d
---+---+---+---
 0 | 0 | 0 | 0  ①
 0 | 0 | 1 | 1
 0 | 1 | 2 | 2  ②
 0 | 1 | 3 | 3
 1 | 1 | 4 | 4  ③

(5 rows)
```

① Rows 1 and 2 are in the same partition, because both columns `a` and `b` are zero.

② Rows 3 and 4 are in the same partition, but a different one, because column `a` is zero and column `b` is 1 in both rows.

③ Row 5 is in a third partition by itself, because both columns `a` and `b` are 1.

Note that a table always has a partition key, and that if the table has no `clustering columns`, then every partition of that table has a single row. because the partition key, compound or otherwise, identifies a single location.

The most important property of partition is that all the rows belonging to the same partition are guaranteed to be stored on the same set of replica nodes. In other words, the partition key of a table defines which rows will be localized on the same node in the cluster. The localization of data is important to the efficient retrieval of data, requiring the Cassandra coordinator to contact as few

nodes as possible. However, there is a flip-side to this guarantee, and all rows sharing a partition key will be stored on the same node, creating a hotspot for both reading and writing. While selecting a primary key that groups table rows assists batch updates and can ensure that the updates are *atomic* and done in *isolation*, the partitions must be sized "just right, not too big nor too small".

Data modeling that considers the querying patterns and assigns primary keys based on the queries will have the lowest latency in fetching data.

Clustering columns

The clustering columns of a table define the clustering order for the partition of that table. For a given `partition`, all rows are ordered by that clustering order. Clustering columns also add uniqueness to a row in a table.

For instance, given:

```
CREATE TABLE t2 (
    a int,
    b int,
    c int,
    d int,
    PRIMARY KEY (a, b, c)
);
INSERT INTO t2 (a, b, c, d) VALUES (0,0,0,0);
INSERT INTO t2 (a, b, c, d) VALUES (0,0,1,1);
INSERT INTO t2 (a, b, c, d) VALUES (0,1,2,2);
INSERT INTO t2 (a, b, c, d) VALUES (0,1,3,3);
INSERT INTO t2 (a, b, c, d) VALUES (1,1,4,4);
SELECT * FROM t2;
```

will result in

```
a | b | c | d
---+---+---+---
 1 | 1 | 4 | 4  ①
 0 | 0 | 0 | 0
 0 | 0 | 1 | 1
 0 | 1 | 2 | 2
 0 | 1 | 3 | 3

(5 rows)
```

① Row 1 is in one partition, and Rows 2-5 are in a different one. The display order is also different.

Looking more closely at the four rows in the same partition, the `b` clustering column defines the order

in which those rows are displayed. Whereas the partition key of the table groups rows on the same node, the clustering columns control how those rows are stored on the node.

That sorting allows the very efficient retrieval of a range of rows within a partition:

```
SELECT * FROM t2 WHERE a = 0 AND b > 0 and b <= 3;
```

will result in

```
a | b | c | d
---+---+---+---
 0 | 1 | 2 | 2
 0 | 1 | 3 | 3

(2 rows)
```

Table options

A CQL table has a number of options that can be set at creation (and, for most of them, altered later). These options are specified after the `WITH` keyword.

One important option that cannot be changed after creation, `CLUSTERING ORDER BY`, influences how queries can be done against the table. It is worth discussing in more detail here.

Clustering order

The clustering order of a table is defined by the clustering columns. By default, the clustering order is ascending for the clustering column's data types. For example, integers order from 1, 2, … n, while text orders from A to Z.

The `CLUSTERING ORDER BY` table option uses a comma-separated list of the clustering columns, each set for either `ASC` (for *ascending* order) or `DESC` (for _descending order). The default is ascending for all clustering columns if the `CLUSTERING ORDER BY` option is not set.

This option is basically a hint for the storage engine that changes the order in which it stores the row. Beware of the consequences of setting this option:

- It changes the default ascending order of results when queried with a `SELECT` statement with no `ORDER BY` clause.
- It limits how the `ORDER BY` clause is used in `SELECT` statements on that table. Results can only be ordered with either the original clustering order or the reverse clustering order. Suppose you create a table with two clustering columns `a` and `b`, defined `WITH CLUSTERING ORDER BY (a DESC, b ASC)`. Queries on the table can use `ORDER BY (a DESC, b ASC)` or `ORDER BY (a ASC, b DESC)`. Mixed

order, such as `ORDER BY (a ASC, b ASC)` or `ORDER BY (a DESC, b DESC)` will not return expected order.

- It has a performance impact on queries. Queries in reverse clustering order are slower than the default ascending order. If you plan to query mostly in descending order, declare the clustering order in the table schema using `WITH CLUSTERING ORDER BY ()`. This optimization is common for time series, to retrieve the data from newest to oldest.

Other table options

A table supports the following options:

option	kind	default	description
`comment`	*simple*	none	A free-form, human-readable comment
`speculative_retry`	*simple*	99PERCENTILE	Speculative retry options
`cdc`	*boolean*	false	Create a Change Data Capture (CDC) log on the table
`additional_write_policy`	*simple*	99PERCENTILE	Same as `speculative_retry`
`gc_grace_seconds`	*simple*	864000	Time to wait before garbage collecting tombstones (deletion markers)
`bloom_filter_fp_chance`	*simple*	0.00075	The target probability of false positive of the sstable bloom filters. Said bloom filters will be sized to provide the provided probability, thus lowering this value impacts the size of bloom filters in-memory and on-disk.
`default_time_to_live`	*simple*	0	Default expiration time (“TTL”) in seconds for a table
`compaction`	*map*	*see below*	**Compaction options**
`compression`	*map*	*see below*	**Compression options**
`caching`	*map*	*see below*	Caching options
`memtable_flush_period_in_ms`	*simple*	0	Time (in ms) before Cassandra flushes memtables to disk
`read_repair`	*simple*	BLOCKING	Sets read repair behavior (see below)

Speculative retry options

By default, Cassandra read coordinators only query as many replicas as necessary to satisfy consistency levels: one for consistency level `ONE`, a quorum for `QUORUM`, and so on. `speculative_retry`

determines when coordinators may query additional replicas, a useful action when replicas are slow or unresponsive. Speculative retries reduce the latency. The speculative_retry option configures rapid read protection, where a coordinator sends more requests than needed to satisfy the consistency level.

IMPORTANT	Frequently reading from additional replicas can hurt cluster performance. When in doubt, keep the default `99PERCENTILE`.

Pre-Cassandra 4.0 speculative retry policy takes a single string as a parameter:

- `NONE`
- `ALWAYS`
- `99PERCENTILE` (PERCENTILE)
- `50MS` (CUSTOM)

An example of setting speculative retry sets a custom value:

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

This example uses a percentile for the setting:

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

A percentile setting can backfire. If a single host becomes unavailable, it can force up the percentiles. A value of `p99` will not speculate as intended because the value at the specified percentile has increased too much. If the consistency level is set to `ALL`, all replicas are queried regardless of the speculative retry setting.

Cassandra 4.0 supports case-insensitivity for speculative retry values (CASSANDRA-14293). For example, assigning the value as `none`, `None`, or `NONE` has the same effect.

Additionally, the following values are added:

Format	Example	Description
`XPERCENTILE`	90.5PERCENTILE	Coordinators record average per-table response times for all replicas. If a replica takes longer than `X` percent of this table's average response time, the coordinator queries an additional replica. `X` must be between 0 and 100.
`XP`	90.5P	Same as `XPERCENTILE`

Format	Example	Description
`Yms`	25ms	If a replica takes more than `Y` milliseconds to respond, the coordinator queries an additional replica.
`MIN(XPERCENTILE,YMS)`	MIN(99PERCENTILE,35MS)	A hybrid policy that uses either the specified percentile or fixed milliseconds depending on which value is lower at the time of calculation. Parameters are `XPERCENTILE`, `XP`, or `Yms`. This setting helps protect against a single slow instance.
`MAX(XPERCENTILE,YMS) ALWAYS NEVER`	MAX(90.5P,25ms)	A hybrid policy that uses either the specified percentile or fixed milliseconds depending on which value is higher at the time of calculation.

Cassandra 4.0 adds support for hybrid `MIN()` and `MAX()` speculative retry policies, with a mix and match of either `MIN(), MAX()`, `MIN(), MIN()`, or `MAX(), MAX()` (CASSANDRA-14293). The hybrid mode will still speculate if the normal `p99` for the table is < 50ms, the minimum value. But if the `p99` level goes higher than the maximum value, then that value can be used. In a hybrid value, one value must be a fixed time (ms) value and the other a percentile value.

To illustrate variations, the following examples are all valid:

```
min(99percentile,50ms)
max(99p,50MS)
MAX(99P,50ms)
MIN(99.9PERCENTILE,50ms)
max(90percentile,100MS)
MAX(100.0PERCENTILE,60ms)
```

The `additional_write_policy` setting specifies the threshold at which a cheap quorum write will be upgraded to include transient replicas.

Compaction options

The `compaction` options must minimally define the `'class'` sub-option, to specify the compaction strategy class to use. The supported classes are:

- `'SizeTieredCompactionStrategy'`, **STCS** (Default)

- `'LeveledCompactionStrategy'`, **LCS**
- `'TimeWindowCompactionStrategy'`, **TWCS**

If a custom strategies is required, specify the full class name as a **string constant**.

All default strategies support a number of **common options**, as well as options specific to the strategy chosen. See the section corresponding to your strategy for details: **STCS**, **LCS**, **TWCS**.

Compression options

The `compression` options define if and how the SSTables of the table are compressed. Compression is configured on a per-table basis as an optional argument to `CREATE TABLE` or `ALTER TABLE`. The following sub-options are available:

Option	Default	Description
`class`	LZ4Compressor	The compression algorithm to use. Default compressor are: LZ4Compressor, SnappyCompressor, DeflateCompressor and ZstdCompressor. Use `'enabled' : false` to disable compression. Custom compressor can be provided by specifying the full class name as a **string constant**.
`enabled`	true	Enable/disable sstable compression. If the `enabled` option is set to `false`, no other options must be specified.
`chunk_length_in_kb`	64	On disk SSTables are compressed by block (to allow random reads). This option defines the size (in KB) of said block. See **note** for further information.
`compression_level`	3	Compression level. Only applicable for `ZstdCompressor`. Accepts values between `-131072` and `22`.

NOTE Bigger values may improve the compression rate, but will increase the minimum size of data to be read from disk for a read. The default value is an optimal value for compressing tables. Chunk length must be a power of 2 when computing the chunk number from an uncompressed file offset. Block size may be adjusted based on

read/write access patterns such as:

- How much data is typically requested at once
- Average size of rows in the table

For instance, to create a table with LZ4Compressor and a `chunk_length_in_kb` of 4 KB:

```
CREATE TABLE simple (
   id int,
   key text,
   value text,
   PRIMARY KEY (key, value)
) WITH compression = {'class': 'LZ4Compressor', 'chunk_length_in_kb': 4};
```

Caching options

Caching optimizes the use of cache memory of a table. The cached data is weighed by size and access frequency. The `caching` options can configure both the `key cache` and the `row cache` for the table. The following sub-options are available:

Option	Default	Description
`keys`	ALL	Whether to cache keys (key cache) for this table. Valid values are: `ALL` and `NONE`.
`rows_per_partition`	NONE	The amount of rows to cache per partition (row cache). If an integer `n` is specified, the first `n` queried rows of a partition will be cached. Valid values are: `ALL`, to cache all rows of a queried partition, or `NONE` to disable row caching.

For instance, to create a table with both a key cache and 10 rows cached per partition:

```
CREATE TABLE simple (
id int,
key text,
value text,
PRIMARY KEY (key, value)
) WITH caching = {'keys': 'ALL', 'rows_per_partition': 10};
```

Read Repair options

The `read_repair` options configure the read repair behavior, tuning for various performance and consistency behaviors.

The values are:

Option	Default	Description
`BLOCKING`	yes	If a read repair is triggered, the read blocks writes sent to other replicas until the consistency level is reached by the writes.
`NONE`	no	If set, the coordinator reconciles any differences between replicas, but doesn't attempt to repair them.

Two consistency properties are affected by read repair behavior.

- Monotonic quorum reads: Monotonic quorum reads prevents reads from appearing to go back in time in some circumstances. When monotonic quorum reads are not provided and a write fails to reach a quorum of replicas, the read values may be visible in one read, and then disappear in a subsequent read. `BLOCKING` provides this behavior.
- Write atomicity: Write atomicity prevents reads from returning partially-applied writes. Cassandra attempts to provide partition-level write atomicity, but since only the data covered by a SELECT statement is repaired by a read repair, read repair can break write atomicity when data is read at a more granular level than it is written. For example, read repair can break write atomicity if you write multiple rows to a clustered partition in a batch, but then select a single row by specifying the clustering column in a SELECT statement. `NONE` provides this behavior.

Other considerations:

- Adding new columns (see `ALTER TABLE` below) is a constant time operation. Thus, there is no need to anticipate future usage while initially creating a table.

ALTER TABLE

Altering an existing table uses the `ALTER TABLE` statement:

```
alter_table_statement::= ALTER TABLE [ IF EXISTS ] table_name alter_table_instruction
alter_table_instruction::= ADD [ IF NOT EXISTS ] column_definition ( ','
column_definition)*
    | DROP [ IF EXISTS ] column_name ( ',' column_name )*
    | RENAME [ IF EXISTS ] column_name to column_name (AND column_name to column_name)*
```

```
    | ALTER [ IF EXISTS ] column_name ( column_mask | DROP MASKED )
    | WITH options
column_definition::= column_name cql_type [ column_mask]
column_mask::= MASKED WITH ( DEFAULT | function_name '(' term ( ',' term )* ')' )
```

If the table 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 TABLE addamsFamily ADD gravesite varchar;
ALTER TABLE addamsFamily
   WITH comment = 'A most excellent and useful table';
```

The `ALTER TABLE` statement can:

- `ADD` a new column to a table. The primary key of a table cannot ever be altered. A new column, thus, cannot be part of the primary key. Adding a column is a constant-time operation based on the amount of data in the table. If the new column already exists, the statement will return an error, unless `IF NOT EXISTS` is used in which case the operation is a no-op.
- `DROP` a column from a table. This command drops both the column and all its content. Be aware that, while the column becomes immediately unavailable, its content are removed lazily during compaction. Because of this lazy removal, the command is a constant-time operation based on the amount of data in the table. Also, it is important to know that once a column is dropped, a column with the same name can be re-added, unless the dropped column was a non-frozen column like a collection. If the dropped column does not already exist, the statement will return an error, unless `IF EXISTS` is used in which case the operation is a no-op.

WARNING

Warning

Dropping a column assumes that the timestamps used for the value of this column are "real" timestamp in microseconds. Using "real" timestamps in microseconds is the default is and is **strongly** recommended but as Cassandra allows the client to provide any timestamp on any table, it is theoretically possible to use another convention. Please be aware that if you do so, dropping a column will not correctly execute.

- `RENAME` a primary key column of a table. Non primary key columns cannot be renamed. Furthermore, renaming a column to another name which already exists isn't allowed. It's important to keep in mind that renamed columns shouldn't have dependent seconday indexes. If the renamed column does not already exist, the statement will return an error, unless `IF EXISTS` is used in which case the operation is a no-op.
- Use `WITH` to change a table option. The **supported options** are the same as those used when creating a table, with the exception of `CLUSTERING ORDER`. However, setting any `compaction` sub-

options will erase **ALL** previous `compaction` options, so you need to re-specify all the sub-options you wish to keep. The same is true for `compression` sub-options.

DROP TABLE

Dropping a table uses the `DROP TABLE` statement:

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

Dropping a table results in the immediate, irreversible removal of the table, including all data it contains.

If the table does not exist, the statement will return an error, unless `IF EXISTS` is used, when the operation is a no-op.

TRUNCATE TABLE

A table can be truncated using the `TRUNCATE` statement:

```
truncate_statement::= TRUNCATE [ TABLE ] table_name
```

`TRUNCATE TABLE foo` is the preferred syntax for consistency with other DDL statements. However, tables are the only object that can be truncated currently, and the `TABLE` keyword can be omitted.

Truncating a table permanently removes all existing data from the table, but without removing the table itself.

Data Manipulation

Data Manipulation

This section describes the statements supported by CQL to insert, update, delete and query data.

SELECT

Querying data from data is done using a `SELECT` statement:

```
select_statement::= SELECT [ JSON | DISTINCT ] ( select_clause | '*' )
    FROM `table_name`
    [ WHERE `where_clause` ]
    [ GROUP BY `group_by_clause` ]
    [ ORDER BY `ordering_clause` ]
    [ PER PARTITION LIMIT (`integer` | `bind_marker`) ]
    [ LIMIT (`integer` | `bind_marker`) ]
    [ ALLOW FILTERING ]
select_clause::= `selector` [ AS `identifier` ] ( ',' `selector` [ AS `identifier` ] )
selector::== `column_name`
    | `term`
    | CAST '(' `selector` AS `cql_type` ')'
    | `function_name` '(' [ `selector` ( ',' `selector` )_ ] ')'
    | COUNT '(' '_' ')'
where_clause::= `relation` ( AND `relation` )*
relation::= column_name operator term
    '(' column_name ( ',' column_name )* ')' operator tuple_literal
    TOKEN '(' column_name# ( ',' column_name )* ')' operator term
operator::= '=' | '<' | '>' | '<=' | '>=' | '!=' | IN | CONTAINS | CONTAINS KEY
group_by_clause::= column_name ( ',' column_name )*
ordering_clause::= column_name [ ASC | DESC ] ( ',' column_name [ ASC | DESC ] )*
```

For example:

```
SELECT name, occupation FROM users WHERE userid IN (199, 200, 207);
SELECT JSON name, occupation FROM users WHERE userid = 199;
SELECT name AS user_name, occupation AS user_occupation FROM users;

SELECT time, value
FROM events
WHERE event_type = 'myEvent'
  AND time > '2011-02-03'
  AND time <= '2012-01-01'
```

```
SELECT COUNT (*) AS user_count FROM users;
```

The `SELECT` statements reads one or more columns for one or more rows in a table. It returns a result-set of the rows matching the request, where each row contains the values for the selection corresponding to the query. Additionally, **functions** including **aggregations** can be applied to the result.

A `SELECT` statement contains at least a **selection clause** and the name of the table on which the selection is executed. CQL does **not** execute joins or sub-queries and a select statement only apply to a single table. A select statement can also have a **where clause** that can further narrow the query results. Additional clauses can **order** or **limit** the results. Lastly, **queries that require full cluster filtering** can append `ALLOW FILTERING` to any query. For virtual tables, from CASSANDRA-18238, it is not necessary to specify `ALLOW FILTERING` when a query would normally require that. Please consult the documentation for virtual tables to know more.

Selection clause

The `select_clause` determines which columns will be queried and returned in the result set. This clause can also apply transformations to apply to the result before returning. The selection clause consists of a comma-separated list of specific *selectors* or, alternatively, the wildcard character (`*`) to select all the columns defined in the table.

Selectors

A `selector` can be one of:

- A column name of the table selected, to retrieve the values for that column.
- A term, which is usually used nested inside other selectors like functions (if a term is selected directly, then the corresponding column of the result-set will simply have the value of this term for every row returned).
- A casting, which allows to convert a nested selector to a (compatible) type.
- A function call, where the arguments are selector themselves. See the section on **functions** for more details.
- The special call `COUNT() to the *COUNT function`, which counts all non-null results.

Aliases

Every *top-level* selector can also be aliased (using AS). If so, the name of the corresponding column in the result set will be that of the alias. For instance:

```
// Without alias
SELECT int_as_blob(4) FROM t;
```

```
//  int_as_blob(4)
// ----------------
//  0x00000004

// With alias
SELECT int_as_blob(4) AS four FROM t;

//  four
// ------------
//  0x00000004
```

NOTE	Currently, aliases aren't recognized in the `WHERE` or `ORDER BY` clauses in the statement. You must use the orignal column name instead.

`WRITETIME`, `MAXWRITETIME` and `TTL` function

Selection supports three special functions that aren't allowed anywhere else: `WRITETIME`, `MAXWRITETIME` and `TTL`. All functions take only one argument, a column name. If the column is a collection or UDT, it's possible to add element selectors, such as `WRITETTIME(phones[2..4])` or `WRITETTIME(user.name)`. These functions retrieve meta-information that is stored internally for each column:

- `WRITETIME` stores the timestamp of the value of the column.
- `MAXWRITETIME` stores the largest timestamp of the value of the column. For non-collection and non-UDT columns, `MAXWRITETIME` is equivalent to `WRITETIME`. In the other cases, it returns the largest timestamp of the values in the column.
- `TTL` stores the remaining time to live (in seconds) for the value of the column if it is set to expire; otherwise the value is `null`.

The `WRITETIME` and `TTL` functions can be used on multi-cell columns such as non-frozen collections or non-frozen user-defined types. In that case, the functions will return the list of timestamps or TTLs for each selected cell.

The `WHERE` clause

The `WHERE` clause specifies which rows are queried. It specifies a relationship for `PRIMARY KEY` columns or a column that has a **secondary index** defined, along with a set value.

Not all relationships are allowed in a query. For instance, only an equality is allowed on a partition key. The `IN` clause is considered an equality for one or more values. The `TOKEN` clause can be used to query for partition key non-equalities. A partition key must be specified before clustering columns in the `WHERE` clause. The relationship for clustering columns must specify a **contiguous** set of rows to order.

For instance, given:

```
CREATE TABLE posts (
    userid text,
    blog_title text,
    posted_at timestamp,
    entry_title text,
    content text,
    category int,
    PRIMARY KEY (userid, blog_title, posted_at)
);
```

The following query is allowed:

```
SELECT entry_title, content FROM posts
 WHERE userid = 'john doe'
   AND blog_title='John''s Blog'
   AND posted_at >= '2012-01-01' AND posted_at < '2012-01-31';
```

But the following one is not, as it does not select a contiguous set of rows (and we suppose no secondary indexes are set):

```
// Needs a blog_title to be set to select ranges of posted_at

SELECT entry_title, content FROM posts
 WHERE userid = 'john doe'
   AND posted_at >= '2012-01-01' AND posted_at < '2012-01-31';
```

When specifying relationships, the `TOKEN` function can be applied to the `PARTITION KEY` column to query. Rows will be selected based on the token of the `PARTITION_KEY` rather than on the value.

IMPORTANT

The token of a key depends on the partitioner in use, and that in particular the `RandomPartitioner` won't yield a meaningful order. Also note that ordering partitioners always order token values by bytes (so even if the partition key is of type int, `token(-1) > token(0)` in particular).

For example:

```
SELECT * FROM posts
 WHERE token(userid) > token('tom') AND token(userid) < token('bob');
```

The `IN` relationship is only allowed on the last column of the partition key or on the last column of the full primary key.

It is also possible to “group” `CLUSTERING COLUMNS` together in a relation using the tuple notation.

For example:

```
SELECT * FROM posts
 WHERE userid = 'john doe'
   AND (blog_title, posted_at) > ('John''s Blog', '2012-01-01');
```

This query will return all rows that sort after the one having “John’s Blog” as `blog_tile` and '2012-01-01' for `posted_at` in the clustering order. In particular, rows having a `post_at <= '2012-01-01'` will be returned, as long as their `blog_title > 'John''s Blog'`.

That would not be the case for this example:

```
SELECT * FROM posts
 WHERE userid = 'john doe'
   AND blog_title > 'John''s Blog'
   AND posted_at > '2012-01-01';
```

The tuple notation may also be used for `IN` clauses on clustering columns:

```
SELECT * FROM posts
 WHERE userid = 'john doe'
   AND (blog_title, posted_at) IN (('John''s Blog', '2012-01-01'), ('Extreme Chess',
'2014-06-01'));
```

The `CONTAINS` operator may only be used for collection columns (lists, sets, and maps). In the case of maps, `CONTAINS` applies to the map values. The `CONTAINS KEY` operator may only be used on map columns and applies to the map keys.

Grouping results

The `GROUP BY` option can condense all selected rows that share the same values for a set of columns into a single row.

Using the `GROUP BY` option, rows can be grouped at the partition key or clustering column level. Consequently, the `GROUP BY` option only accepts primary key columns in defined order as arguments. If a primary key column is restricted by an equality restriction, it is not included in the `GROUP BY` clause.

Aggregate functions will produce a separate value for each group. If no `GROUP BY` clause is specified, aggregates functions will produce a single value for all the rows.

If a column is selected without an aggregate function, in a statement with a `GROUP BY`, the first value

encounter in each group will be returned.

Ordering results

The `ORDER BY` clause selects the order of the returned results. The argument is a list of column names and each column's order (`ASC` for ascendant and `DESC` for descendant, The possible orderings are limited by the **clustering order** defined on the table:

- if the table has been defined without any specific `CLUSTERING ORDER`, then the order is as defined by the clustering columns or the reverse
- otherwise, the order is defined by the `CLUSTERING ORDER` option and the reversed one.

Limiting results

The `LIMIT` option to a `SELECT` statement limits the number of rows returned by a query. The `PER PARTITION LIMIT` option limits the number of rows returned for a given partition by the query. Both types of limits can used in the same statement.

Allowing filtering

By default, CQL only allows select queries that don't involve a full scan of all partitions. If all partitions are scanned, then returning the results may experience a significant latency proportional to the amount of data in the table. The `ALLOW FILTERING` option explicitly executes a full scan. Thus, the performance of the query can be unpredictable.

For example, consider the following table of user profiles with birth year and country of residence. The birth year has a secondary index defined.

```
CREATE TABLE users (
    username text PRIMARY KEY,
    firstname text,
    lastname text,
    birth_year int,
    country text
);

CREATE INDEX ON users(birth_year);
```

The following queries are valid:

```
// All users are returned
SELECT * FROM users;
```

```
// All users with a particular birth year are returned
SELECT * FROM users WHERE birth_year = 1981;
```

In both cases, the query performance is proportional to the amount of data returned. The first query returns all rows, because all users are selected. The second query returns only the rows defined by the secondary index, a per-node implementation; the results will depend on the number of nodes in the cluster, and is indirectly proportional to the amount of data stored. The number of nodes will always be multiple number of magnitude lower than the number of user profiles stored. Both queries may return very large result sets, but the addition of a `LIMIT` clause can reduced the latency.

The following query will be rejected:

```
SELECT * FROM users WHERE birth_year = 1981 AND country = 'FR';
```

Cassandra cannot guarantee that large amounts of data won't have to scanned amount of data, even if the result is small. If you know that the dataset is small, and the performance will be reasonable, add `ALLOW FILTERING` to allow the query to execute:

```
SELECT * FROM users WHERE birth_year = 1981 AND country = 'FR' ALLOW FILTERING;
```

INSERT

Inserting data for a row is done using an `INSERT` statement:

```
insert_statement::= INSERT INTO table_name ( names_values | json_clause )
    [ IF NOT EXISTS ]
    [ USING update_parameter ( AND update_parameter )* ]
names_values::= names VALUES tuple_literal
json_clause::= JSON string [ DEFAULT ( NULL | UNSET ) ]
names::= '(' column_name ( ',' column_name )* ')'
```

For example:

```
INSERT INTO NerdMovies (movie, director, main_actor, year)
   VALUES ('Serenity', 'Joss Whedon', 'Nathan Fillion', 2005)
   USING TTL 86400;

INSERT INTO NerdMovies JSON '{"movie": "Serenity", "director": "Joss Whedon", "year":
2005}';
```

The `INSERT` statement writes one or more columns for a given row in a table. Since a row is identified

by its `PRIMARY KEY`, at least one columns must be specified. The list of columns to insert must be supplied with the `VALUES` syntax. When using the `JSON` syntax, `VALUES` are optional. See the section on **JSON support** for more detail. All updates for an `INSERT` are applied atomically and in isolation.

Unlike in SQL, `INSERT` does not check the prior existence of the row by default. The row is created if none existed before, and updated otherwise. Furthermore, there is no means of knowing which action occurred.

The `IF NOT EXISTS` condition can restrict the insertion if the row does not exist. However, note that using `IF NOT EXISTS` will incur a non-negligible performance cost, because Paxos is used, so this should be used sparingly.

Please refer to the **UPDATE** section for informations on the `update_parameter`. Also note that `INSERT` does not support counters, while `UPDATE` does.

UPDATE

Updating a row is done using an `UPDATE` statement:

```
update_statement ::=    UPDATE table_name
                        [ USING update_parameter ( AND update_parameter )* ]
                        SET assignment( ',' assignment )*
                        WHERE where_clause
                        [ IF ( EXISTS | condition ( AND condition)*) ]
update_parameter ::= ( TIMESTAMP | TTL ) ( integer | bind_marker )
assignment: simple_selection'=' term
                `| column_name'=' column_name ( '+' | '-' ) term
                | column_name'=' list_literal'+' column_name
simple_selection ::= column_name
                        | column_name '[' term']'
                        | column_name'.' field_name
condition ::= `simple_selection operator term
```

For instance:

```
UPDATE NerdMovies USING TTL 400
   SET director   = 'Joss Whedon',
       main_actor = 'Nathan Fillion',
       year       = 2005
 WHERE movie = 'Serenity';

UPDATE UserActions
   SET total = total + 2
   WHERE user = B70DE1D0-9908-4AE3-BE34-5573E5B09F14
```

```
    AND action = 'click';
```

The `UPDATE` statement writes one or more columns for a given row in a table. The `WHERE` clause is used to select the row to update and must include all columns of the `PRIMARY KEY`. Non-primary key columns are set using the `SET` keyword. In an `UPDATE` statement, all updates within the same partition key are applied atomically and in isolation.

Unlike in SQL, `UPDATE` does not check the prior existence of the row by default. The row is created if none existed before, and updated otherwise. Furthermore, there is no means of knowing which action occurred.

The `IF` condition can be used to choose whether the row is updated or not if a particular condition is met. However, like the `IF NOT EXISTS` condition, a non-negligible performance cost can be incurred.

Regarding the `SET` assignment:

- `c = c + 3` will increment/decrement counters, the only operation allowed. The column name after the '=' sign **must** be the same than the one before the '=' sign. Increment/decrement is only allowed on counters. See the section on **counters** for details.
- `id = id + <some-collection>` and `id[value1] = value2` are for collections. See the **collections** for details.
- `id.field = 3` is for setting the value of a field on a non-frozen user-defined types. See the **UDTs** for details.

Update parameters

`UPDATE` and `INSERT` statements support the following parameters:

- `TTL`: specifies an optional Time To Live (in seconds) for the inserted values. If set, the inserted values are automatically removed from the database after the specified time. Note that the TTL concerns the inserted values, not the columns themselves. This means that any subsequent update of the column will also reset the TTL (to whatever TTL is specified in that update). By default, values never expire. A TTL of 0 is equivalent to no TTL. If the table has a default_time_to_live, a TTL of 0 will remove the TTL for the inserted or updated values. A TTL of `null` is equivalent to inserting with a TTL of 0.

`UPDATE`, `INSERT`, `DELETE` and `BATCH` statements support the following parameters:

- `TIMESTAMP`: sets the timestamp for the operation. If not specified, the coordinator will use the current time (in microseconds) at the start of statement execution as the timestamp. This is usually a suitable default.

DELETE

Deleting rows or parts of rows uses the DELETE statement:

```
delete_statement::= DELETE [ simple_selection ( ',' simple_selection ) ]
    FROM table_name
    [ USING update_parameter ( AND update_parameter# )* ]
    WHERE where_clause
    [ IF ( EXISTS | condition ( AND condition)*) ]
```

For example:

```
DELETE FROM NerdMovies USING TIMESTAMP 1240003134
 WHERE movie = 'Serenity';

DELETE phone FROM Users
 WHERE userid IN (C73DE1D3-AF08-40F3-B124-3FF3E5109F22, B70DE1D0-9908-4AE3-BE34-
5573E5B09F14);
```

The DELETE statement deletes columns and rows. If column names are provided directly after the DELETE keyword, only those columns are deleted from the row indicated by the WHERE clause. Otherwise, whole rows are removed.

The WHERE clause specifies which rows are to be deleted. Multiple rows may be deleted with one statement by using an IN operator. A range of rows may be deleted using an inequality operator (such as >=).

DELETE supports the TIMESTAMP option with the same semantics as in **updates**.

In a DELETE statement, all deletions within the same partition key are applied atomically and in isolation.

A DELETE operation can be conditional through the use of an IF clause, similar to UPDATE and INSERT statements. However, as with INSERT and UPDATE statements, this will incur a non-negligible performance cost because Paxos is used, and should be used sparingly.

BATCH

Multiple INSERT, UPDATE and DELETE can be executed in a single statement by grouping them through a BATCH statement:

```
batch_statement ::=     BEGIN [ UNLOGGED | COUNTER ] BATCH
                        [ USING update_parameter( AND update_parameter)* ]
```

```
                        modification_statement ( ';' modification_statement )*
                        APPLY BATCH
modification_statement ::= insert_statement | update_statement | delete_statement
```

For instance:

```
BEGIN BATCH
   INSERT INTO users (userid, password, name) VALUES ('user2', 'ch@ngem3b', 'second
user');
   UPDATE users SET password = 'ps22dhds' WHERE userid = 'user3';
   INSERT INTO users (userid, password) VALUES ('user4', 'ch@ngem3c');
   DELETE name FROM users WHERE userid = 'user1';
APPLY BATCH;
```

The `BATCH` statement group multiple modification statements (insertions/updates and deletions) into a single statement. It serves several purposes:

- It saves network round-trips between the client and the server (and sometimes between the server coordinator and the replicas) when batching multiple updates.
- All updates in a `BATCH` belonging to a given partition key are performed in isolation.
- By default, all operations in the batch are performed as *logged*, to ensure all mutations eventually complete (or none will). See the notes on **UNLOGGED batches** for more details.

Note that:

- `BATCH` statements may only contain `UPDATE`, `INSERT` and `DELETE` statements (not other batches for instance).
- Batches are *not* a full analogue for SQL transactions.
- If a timestamp is not specified for each operation, then all operations will be applied with the same timestamp (either one generated automatically, or the timestamp provided at the batch level). Due to Cassandra's conflict resolution procedure in the case of timestamp ties, operations may be applied in an order that is different from the order they are listed in the `BATCH` statement. To force a particular operation ordering, you must specify per-operation timestamps.
- A LOGGED batch to a single partition will be converted to an UNLOGGED batch as an optimization.

`UNLOGGED` batches

By default, Cassandra uses a batch log to ensure all operations in a batch eventually complete or none will (note however that operations are only isolated within a single partition).

There is a performance penalty for batch atomicity when a batch spans multiple partitions. If you do not want to incur this penalty, you can tell Cassandra to skip the batchlog with the `UNLOGGED` option. If

the `UNLOGGED` option is used, a failed batch might leave the patch only partly applied.

`COUNTER` batches

Use the `COUNTER` option for batched counter updates. Unlike other updates in Cassandra, counter updates are not idempotent.

Arithmetic Operators

Arithmetic Operators

CQL supports the following operators:

Operator	Description
- (unary)	Negates operand
+	Addition
-	Substraction
*	Multiplication
/	Division
%	Returns the remainder of a division

Number Arithmetic

All arithmetic operations are supported on numeric types or counters.

The return type of the operation will be based on the operand types:

left/right	tinyint	smallint	int	bigint	counter	float	double	varint	decimal
tinyint	tinyint	smallint	int	bigint	bigint	float	double	varint	decimal
smallint	smallint	smallint	int	bigint	bigint	float	double	varint	decimal
int	int	int	int	bigint	bigint	float	double	varint	decimal
bigint	bigint	bigint	bigint	bigint	bigint	double	double	varint	decimal
counter	bigint	bigint	bigint	bigint	bigint	double	double	varint	decimal
float	float	float	float	double	double	float	double	decimal	decimal
double	double	double	double	double	double	double	double	decimal	decimal
varint	varint	varint	varint	decimal	decimal	decimal	decimal	decimal	decimal
decimal	decimal	decimal	decimal	decimal	decimal	decimal	decimal	decimal	decimal

*, / and % operators have a higher precedence level than + and - operator. By consequence, they will be evaluated before. If two operator in an expression have the same precedence level, they will be evaluated left to right based on their position in the expression.

Datetime Arithmetic

A `duration` can be added (+) or substracted (-) from a `timestamp` or a `date` to create a new `timestamp` or `date`. So for instance:

```
SELECT * FROM myTable WHERE t = '2017-01-01' - 2d;
```

will select all the records with a value of `t` which is in the last 2 days of 2016.

JSON Support

JSON Support

Cassandra 2.2 introduced JSON support to `SELECT <select-statement>` and `INSERT <insert-statement>` statements. This support does not fundamentally alter the CQL API (for example, the schema is still enforced). It simply provides a convenient way to work with JSON documents.

SELECT JSON

With `SELECT` statements, the `JSON` keyword is used to return each row as a single `JSON` encoded map. The remainder of the `SELECT` statement behavior is the same.

The result map keys match the column names in a normal result set. For example, a statement like `SELECT JSON a, ttl(b) FROM ...` would result in a map with keys `"a"` and `"ttl(b)"`. However, there is one notable exception: for symmetry with `INSERT JSON` behavior, case-sensitive column names with upper-case letters will be surrounded with double quotes. For example, `SELECT JSON myColumn FROM ...` would result in a map key `"\"myColumn\""` with escaped quotes).

The map values will JSON-encoded representations (as described below) of the result set values.

INSERT JSON

With `INSERT` statements, the new `JSON` keyword can be used to enable inserting a `JSON` encoded map as a single row. The format of the `JSON` map should generally match that returned by a `SELECT JSON` statement on the same table. In particular, case-sensitive column names should be surrounded with double quotes. For example, to insert into a table with two columns named "myKey" and "value", you would do the following:

```
INSERT INTO mytable JSON '{ "\"myKey\"": 0, "value": 0}';
```

By default (or if `DEFAULT NULL` is explicitly used), a column omitted from the `JSON` map will be set to `NULL`, meaning that any pre-existing value for that column will be removed (resulting in a tombstone being created). Alternatively, if the `DEFAULT UNSET` directive is used after the value, omitted column values will be left unset, meaning that pre-existing values for those column will be preserved.

JSON Encoding of Cassandra Data Types

Where possible, Cassandra will represent and accept data types in their native `JSON` representation. Cassandra will also accept string representations matching the CQL literal format for all single-field types. For example, floats, ints, UUIDs, and dates can be represented by CQL literal strings. However,

compound types, such as collections, tuples, and user-defined types must be represented by native `JSON` collections (maps and lists) or a JSON-encoded string representation of the collection.

The following table describes the encodings that Cassandra will accept in `INSERT JSON` values (and `from_json()` arguments) as well as the format Cassandra will use when returning data for `SELECT JSON` statements (and `from_json()`):

Type	Formats accepted	Return format	Notes
`ascii`	string	string	Uses JSON's `\u` character escape
`bigint`	integer, string	integer	String must be valid 64 bit integer
`blob`	string	string	String should be 0x followed by an even number of hex digits
`boolean`	boolean, string	boolean	String must be "true" or "false"
`date`	string	string	Date in format `YYYY-MM-DD`, timezone UTC
`decimal`	integer, float, string	float	May exceed 32 or 64-bit IEEE-754 floating point precision in client-side decoder
`double`	integer, float, string	float	String must be valid integer or float
`float`	integer, float, string	float	String must be valid integer or float
`inet`	string	string	IPv4 or IPv6 address
`int`	integer, string	integer	String must be valid 32 bit integer
`list`	list, string	list	Uses JSON's native list representation
`map`	map, string	map	Uses JSON's native map representation
`smallint`	integer, string	integer	String must be valid 16 bit integer
`set`	list, string	list	Uses JSON's native list representation

Type	Formats accepted	Return format	Notes
`text`	string	string	Uses JSON's `\u` character escape
`time`	string	string	Time of day in format `HH-MM-SS[.fffffffff]`
`timestamp`	integer, string	string	A timestamp. Strings constant allows to input `timestamps as dates <timestamps>`. Datestamps with format `YYYY-MM-DD HH:MM:SS.SSS` are returned.
`timeuuid`	string	string	Type 1 UUID. See `constant` for the UUID format
`tinyint`	integer, string	integer	String must be valid 8 bit integer
`tuple`	list, string	list	Uses JSON's native list representation
`UDT`	map, string	map	Uses JSON's native map representation with field names as keys
`uuid`	string	string	See `constant` for the UUID format
`varchar`	string	string	Uses JSON's `\u` character escape
`varint`	integer, string	integer	Variable length; may overflow 32 or 64 bit integers in client-side decoder

The from_json() Function

The `from_json()` function may be used similarly to `INSERT JSON`, but for a single column value. It may only be used in the `VALUES` clause of an `INSERT` statement or as one of the column values in an `UPDATE`, `DELETE`, or `SELECT` statement. For example, it cannot be used in the selection clause of a `SELECT` statement.

The to_json() Function

The `to_json()` function may be used similarly to `SELECT JSON`, but for a single column value. It may only be used in the selection clause of a `SELECT` statement.

Appendices

Appendices

Appendix A: CQL Keywords

CQL distinguishes between *reserved* and *non-reserved* keywords. Reserved keywords cannot be used as identifier, they are truly reserved for the language (but one can enclose a reserved keyword by double-quotes to use it as an identifier). Non-reserved keywords however only have a specific meaning in certain context but can used as identifier otherwise. The only *raison d'être* of these non-reserved keywords is convenience: some keyword are non-reserved when it was always easy for the parser to decide whether they were used as keywords or not.

Keyword	Reserved?
ADD	yes
AGGREGATE	no
ALL	no
ALLOW	yes
ALTER	yes
AND	yes
APPLY	yes
AS	no
ASC	yes
ASCII	no
AUTHORIZE	yes
BATCH	yes
BEGIN	yes
BIGINT	no
BLOB	no
BOOLEAN	no
BY	yes
CALLED	no
CLUSTERING	no
COLUMNFAMILY	yes
COMPACT	no
CONTAINS	no

Keyword	Reserved?
COUNT	no
COUNTER	no
CREATE	yes
CUSTOM	no
DATE	no
DECIMAL	no
DELETE	yes
DESC	yes
DESCRIBE	yes
DISTINCT	no
DOUBLE	no
DROP	yes
ENTRIES	yes
EXECUTE	yes
EXISTS	no
FILTERING	no
FINALFUNC	no
FLOAT	no
FROM	yes
FROZEN	no
FULL	yes
FUNCTION	no
FUNCTIONS	no
GRANT	yes
IF	yes
IN	yes
INDEX	yes
INET	no
INFINITY	yes
INITCOND	no
INPUT	no

Keyword	Reserved?
INSERT	yes
INT	no
INTO	yes
JSON	no
KEY	no
KEYS	no
KEYSPACE	yes
KEYSPACES	no
LANGUAGE	no
LIMIT	yes
LIST	no
LOGIN	no
MAP	no
MASKED	no
MODIFY	yes
NAN	yes
NOLOGIN	no
NORECURSIVE	yes
NOSUPERUSER	no
NOT	yes
NULL	yes
OF	yes
ON	yes
OPTIONS	no
OR	yes
ORDER	yes
PASSWORD	no
PERMISSION	no
PERMISSIONS	no
PRIMARY	yes
RENAME	yes

Keyword	Reserved?
REPLACE	yes
RETURNS	no
REVOKE	yes
ROLE	no
ROLES	no
SCHEMA	yes
SELECT	yes
SELECT_MASKED	no
SET	yes
SFUNC	no
SMALLINT	no
STATIC	no
STORAGE	no
STYPE	no
SUPERUSER	no
TABLE	yes
TEXT	no
TIME	no
TIMESTAMP	no
TIMEUUID	no
TINYINT	no
TO	yes
TOKEN	yes
TRIGGER	no
TRUNCATE	yes
TTL	no
TUPLE	no
TYPE	no
UNLOGGED	yes
UNMASK	no
UPDATE	yes

Keyword	Reserved?
USE	yes
USER	no
USERS	no
USING	yes
UUID	no
VALUES	no
VARCHAR	no
VARINT	no
WHERE	yes
WITH	yes
WRITETIME	no
MAXWRITETIME	no

Appendix B: CQL Reserved Types

The following type names are not currently used by CQL, but are reserved for potential future use. User-defined types may not use reserved type names as their name.

type
bitstring
byte
complex
enum
interval
macaddr

Appendix C: Dropping Compact Storage

Starting version 4.0, Thrift and COMPACT STORAGE is no longer supported.

`ALTER ⋯ DROP COMPACT STORAGE` statement makes Compact Tables CQL-compatible, exposing internal structure of Thrift/Compact Tables:

- CQL-created Compact Tables that have no clustering columns, will expose an additional clustering column `column1` with `UTF8Type`.
- CQL-created Compact Tables that had no regular columns, will expose a regular column `value` with

`BytesType`.

- For CQL-Created Compact Tables, all columns originally defined as `regular` will be come `static`
- CQL-created Compact Tables that have clustering but have no regular columns will have an empty value column (of `EmptyType`)
- SuperColumn Tables (can only be created through Thrift) will expose a compact value map with an empty name.
- Thrift-created Compact Tables will have types corresponding to their Thrift definition.

Changes

Changes

The following describes the changes in each version of CQL.

3.4.7

- Add vector similarity functions (18640)
- Remove deprecated functions `dateOf` and `unixTimestampOf`, replaced by `toTimestamp` and `toUnixTimestamp` (18328)
- Added support for attaching masking functions to table columns (18068)
- Add UNMASK permission (18069)
- Add SELECT_MASKED permission (18070)
- Add support for using UDFs as masking functions (18071)
- Adopt snake_case function names, deprecating all previous camelCase or alltogetherwithoutspaces function names (18037)
- Add new `vector` data type (18504)

3.4.6

- Add support for IF EXISTS and IF NOT EXISTS in ALTER statements (16916)
- Allow GRANT/REVOKE multiple permissions in a single statement (17030)
- Pre hashed passwords in CQL (17334)
- Add support for type casting in WHERE clause components and in the values of INSERT/UPDATE statements (14337)
- Add support for CONTAINS and CONTAINS KEY in conditional UPDATE and DELETE statement (10537)
- Allow to grant permission for all tables in a keyspace (17027)
- Allow to aggregate by time intervals (11871)

3.4.5

- Adds support for arithmetic operators (11935)
- Adds support for `+` and `-` operations on dates (11936)
- Adds `currentTimestamp`, `currentDate`, `currentTime` and `currentTimeUUID` functions (13132)

3.4.4

- `ALTER TABLE ALTER` has been removed; a column's type may not be changed after creation (12443).
- `ALTER TYPE ALTER` has been removed; a field's type may not be changed after creation (12443).

3.4.3

- Adds a new `duration data types <data-types>` (11873).
- Support for `GROUP BY` (10707).
- Adds a `DEFAULT UNSET` option for `INSERT JSON` to ignore omitted columns (11424).
- Allows `null` as a legal value for TTL on insert and update. It will be treated as equivalent to inserting a 0 (12216).

3.4.2

- If a table has a non zero `default_time_to_live`, then explicitly specifying a TTL of 0 in an `INSERT` or `UPDATE` statement will result in the new writes not having any expiration (that is, an explicit TTL of 0 cancels the `default_time_to_live`). This wasn't the case before and the `default_time_to_live` was applied even though a TTL had been explicitly set.
- `ALTER TABLE ADD` and `DROP` now allow multiple columns to be added/removed.
- New `PER PARTITION LIMIT` option for `SELECT` statements (see CASSANDRA-7017.
- `User-defined functions <cql-functions>` can now instantiate `UDTValue` and `TupleValue` instances via the new `UDFContext` interface (see CASSANDRA-10818.
- `User-defined types <udts>` may now be stored in a non-frozen form, allowing individual fields to be updated and deleted in `UPDATE` statements and `DELETE` statements, respectively. (CASSANDRA-7423).

3.4.1

- Adds `CAST` functions.

3.4.0

- Support for `materialized views <materialized-views>`.
- `DELETE` support for inequality expressions and `IN` restrictions on any primary key columns.
- `UPDATE` support for `IN` restrictions on any primary key columns.

3.3.1

- The syntax `TRUNCATE TABLE X` is now accepted as an alias for `TRUNCATE X`.

3.3.0

- `User-defined functions and aggregates <cql-functions>` are now supported.
- Allows double-dollar enclosed strings literals as an alternative to single-quote enclosed strings.
- Introduces Roles to supersede user based authentication and access control
- New `date`, `time`, `tinyint` and `smallint data types <data-types>` have been added.
- `JSON support <cql-json>` has been added
- Adds new time conversion functions and deprecate `dateOf` and `unixTimestampOf`.

3.2.0

- `User-defined types <udts>` supported.
- `CREATE INDEX` now supports indexing collection columns, including indexing the keys of map collections through the `keys()` function
- Indexes on collections may be queried using the new `CONTAINS` and `CONTAINS KEY` operators
- `Tuple types <tuples>` were added to hold fixed-length sets of typed positional fields.
- `DROP INDEX` now supports optionally specifying a keyspace.

3.1.7

- `SELECT` statements now support selecting multiple rows in a single partition using an `IN` clause on combinations of clustering columns.
- `IF NOT EXISTS` and `IF EXISTS` syntax is now supported by `CREATE USER` and `DROP USER` statements, respectively.

3.1.6

- A new `uuid()` method has been added.
- Support for `DELETE ⋯ IF EXISTS` syntax.

3.1.5

- It is now possible to group clustering columns in a relation, see `WHERE <where-clause>` clauses.
- Added support for `static columns <static-columns>`.

3.1.4

- `CREATE INDEX` now allows specifying options when creating CUSTOM indexes.

3.1.3

- Millisecond precision formats have been added to the `timestamp <timestamps>` parser.

3.1.2

- `NaN` and `Infinity` has been added as valid float constants. They are now reserved keywords. In the unlikely case you we using them as a column identifier (or keyspace/table one), you will now need to double quote them.

3.1.1

- `SELECT` statement now allows listing the partition keys (using the `DISTINCT` modifier). See CASSANDRA-4536.
- The syntax `c IN ?` is now supported in `WHERE` clauses. In that case, the value expected for the bind variable will be a list of whatever type `c` is.
- It is now possible to use named bind variables (using `:name` instead of `?`).

3.1.0

- `ALTER TABLE DROP` option added.
- `SELECT` statement now supports aliases in select clause. Aliases in WHERE and ORDER BY clauses are not supported.
- `CREATE` statements for `KEYSPACE`, `TABLE` and `INDEX` now supports an `IF NOT EXISTS` condition. Similarly, `DROP` statements support a `IF EXISTS` condition.
- `INSERT` statements optionally supports a `IF NOT EXISTS` condition and `UPDATE` supports `IF` conditions.

3.0.5

- `SELECT`, `UPDATE`, and `DELETE` statements now allow empty `IN` relations (see CASSANDRA-5626.

3.0.4

- Updated the syntax for custom `secondary indexes <secondary-indexes>`.
- Non-equal condition on the partition key are now never supported, even for ordering partitioner as this was not correct (the order was **not** the one of the type of the partition key). Instead, the `token` method should always be used for range queries on the partition key (see `WHERE clauses <where-clause>`).

3.0.3

- Support for custom `secondary indexes <secondary-indexes>` has been added.

3.0.2

- Type validation for the `constants <constants>` has been fixed. For instance, the implementation used to allow `'2'` as a valid value for an `int` column (interpreting it has the equivalent of `2`), or `42` as a valid `blob` value (in which case `42` was interpreted as an hexadecimal representation of the blob). This is no longer the case, type validation of constants is now more strict. See the `data types <data-types>` section for details on which constant is allowed for which type.
- The type validation fixed of the previous point has lead to the introduction of blobs constants to allow the input of blobs. Do note that while the input of blobs as strings constant is still supported by this version (to allow smoother transition to blob constant), it is now deprecated and will be removed by a future version. If you were using strings as blobs, you should thus update your client code ASAP to switch blob constants.
- A number of functions to convert native types to blobs have also been introduced. Furthermore the token function is now also allowed in select clauses. See the `section on functions <cql-functions>` for details.

3.0.1

- Date strings (and timestamps) are no longer accepted as valid `timeuuid` values. Doing so was a bug in the sense that date string are not valid `timeuuid`, and it was thus resulting in confusing behaviors. However, the following new methods have been added to help working with `timeuuid`: `now`, `minTimeuuid`, `maxTimeuuid` , `dateOf` and `unixTimestampOf`.
- Float constants now support the exponent notation. In other words, `4.2E10` is now a valid floating point value.

Versioning

Versioning of the CQL language adheres to the Semantic Versioning guidelines. Versions take the form X.Y.Z where X, Y, and Z are integer values representing major, minor, and patch level respectively. There is no correlation between Cassandra release versions and the CQL language version.

version	**description**
Major	The major version *must* be bumped when backward incompatible changes are introduced. This should rarely occur.
Minor	Minor version increments occur when new, but backward compatible, functionality is introduced.
Patch	The patch version is incremented when bugs are fixed.