Cassandra Query Language (CQL) v3.4.3

Cassandra Query Language (CQL) v3.4.3

\{toc:maxLevel=3}

CQL Syntax

Preamble

This document describes the Cassandra Query Language (CQL) version 3. CQL v3 is not backward compatible with CQL v2 and differs from it in numerous ways. Note that this document describes the last version of the languages. However, the **changes** section provides the diff between the different versions of CQL v3.

CQL v3 offers a model very close to SQL in the sense that data is put in *tables* containing *rows* of *columns*. For that reason, when used in this document, these terms (tables, rows and columns) have the same definition than they have in SQL. But please note that as such, they do **not** refer to the concept of rows and columns found in the internal implementation of Cassandra and in the thrift and CQL v2 API.

Conventions

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

- Language rules will be given in a BNF -like notation:

bc(syntax). ::= TERMINAL

- Nonterminal symbols will have `<angle brackets>`.
- As additional shortcut notations to BNF, we'll use traditional regular expression's symbols (`?`, `+` and `*`) to signify that a given symbol is optional and/or can be repeated. We'll also allow parentheses to group symbols and the `[<characters>]` notation to represent any one of `<characters>`.
- The grammar is provided for documentation purposes and leave some minor details out. For instance, the last column definition in a `CREATE TABLE` statement is optional but supported if present even though the provided grammar in this document suggest it is not supported.
- Sample code will be provided in a code block:

bc(sample). SELECT sample_usage FROM 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-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` for instance. A convention often used (in particular by the samples of this documentation) is to use upper case for keywords and lower case for other identifiers.

There is a second kind of identifiers called *quoted identifiers* defined by enclosing an arbitrary sequence of characters in double-quotes("). Quoted identifiers are never keywords. Thus `"select"` is not a reserved keyword and can be used to refer to a column, while `select` would raise a parse error. Also, contrarily to 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-Z0-9_]*` is 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.

Warning: *quoted identifiers* allows to declare columns with arbitrary names, and those 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 to avoid any name enclosed by squared brackets (like `"[applied]"`) and any name that looks like a function call (like `"f(x)"`).

Constants

CQL defines the following kind of *constants*: strings, integers, floats, booleans, uuids and blobs:

- A string constant is an arbitrary sequence of characters characters enclosed by single-quote('). One can include a single-quote in a string by repeating it, e.g. `'It''s raining today'`. Those are not to be confused with quoted identifiers that use double-quotes.
- An integer constant is defined by `'-'?[0-9]+`.
- A float constant is defined by `'-'?('.'[0-9]*)?([eE][-]?[0-9+])?`. On top of that, `NaN` and `Infinity` are also float constants.
- A boolean constant is either `true` or `false` up to case-insensitivity (i.e. `True` is a valid boolean constant).
- A UUID constant is defined by `hex{8}-hex{4}-hex{4}-hex{4}-hex{12}` where `hex` is an hexadecimal character, e.g. `[0-9a-fA-F]` and `{4}` is the number of such characters.

- A blob constant is an hexadecimal number defined by `0[xX](hex)+` where `hex` is an hexadecimal character, e.g. `[0-9a-fA-F]`.

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).

bc(sample).
— This is a comment
/* This is
a multi-line comment */

Statements

CQL consists of statements. As in SQL, these statements can be divided in 3 categories:

- Data definition statements, that allow to set and change the way data is stored.
- Data manipulation statements, that allow to change data
- Queries, to look up data

All statements end with a semicolon (`;`) but that semicolon can be omitted when dealing with a single statement. The supported statements are described in the following sections. When describing the grammar of said statements, we will reuse the non-terminal symbols defined below:

bc(syntax)..
::= any quoted or unquoted identifier, excluding reserved keywords
::= (`.')?

::= a string constant
::= an integer constant
::= a float constant
::= |
::= a uuid constant
::= a boolean constant
::= a blob constant

::=
|
|
|
|
::= ?'
| ':'
::=
|
|
| '(' ((',')*)? ')'

::=
|
|
::= '\{' (':' (',' ':')*)? '}'
::= '\{' ((',')*)? '}'
::= '[' ((',')*)? ']'

::=

::= (AND)*

::= =' (| |)
p.
Please note that not every possible productions of the grammar above will be valid in practice. Most notably, `<variable> and nested <collection-literal> are currently not allowed inside <collection-literal>.

A <variable> can be either anonymous (a question mark (?)) or named (an identifier preceded by :). Both declare a bind variables for **prepared statements**. The only difference between an anymous and a named variable is that a named one will be easier to refer to (how exactly depends on the client driver used).

The <properties> production is use by statement that create and alter keyspaces and tables. Each <property> is either a *simple* one, in which case it just has a value, or a *map* one, in which case it's value is a map grouping sub-options. The following will refer to one or the other as the *kind* (*simple* or *map*) of the property.

A <tablename> will be used to identify a table. This is an identifier representing the table name that can be preceded by a keyspace name. The keyspace name, if provided, allow to identify a table in another keyspace than the currently active one (the currently active keyspace is set through the USE statement).

For supported <function>, see the section on **functions**.

Strings can be either enclosed with single quotes or two dollar characters. The second syntax has been introduced to allow strings that contain single quotes. Typical candidates for such strings are source code fragments for user-defined functions.

Sample:

bc(sample)..
`some string value'

double-dollar string can contain single ' quotes
p.

Prepared Statement

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

In a statement, each time a column value is expected (in the data manipulation and query statements), a <variable> (see above) can be used instead. A statement with bind variables must then be *prepared*. Once it has been prepared, it can executed by providing concrete values for the bind variables. The exact procedure to prepare a statement and execute a prepared statement depends on the CQL driver used and is beyond the scope of this document.

In addition to providing column values, bind markers may be used to provide values for LIMIT, TIMESTAMP, and TTL clauses. If anonymous bind markers are used, the names for the query parameters

will be [limit], [timestamp], and [ttl], respectively.

Data Definition

CREATE KEYSPACE

Syntax:

bc(syntax)..
::= CREATE KEYSPACE (IF NOT EXISTS)? WITH
p.
Sample:

bc(sample)..
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;
p.
The `CREATE KEYSPACE statement creates a new top-level *keyspace*. A keyspace is a namespace that defines a replication strategy and some options for a set of tables. Valid keyspaces names are identifiers composed exclusively of alphanumerical characters and whose length is lesser or equal to 32. Note that as identifiers, keyspace names are case insensitive: use a quoted identifier for case sensitive keyspace names.

The supported <properties> for CREATE KEYSPACE are:

name	kind	mandatory	default	description
replication	*map*	yes		The replication strategy and options to use for the keyspace.
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. It must at least contains the 'class' sub-option which defines the replication strategy class to use. The rest of the sub-options depends on that replication strategy class. By default, Cassandra support the following 'class':

- `'SimpleStrategy'`: A simple strategy that defines a simple replication factor for the whole cluster. The only sub-options supported is `'replication_factor'` to define that replication factor and is mandatory.
- `'NetworkTopologyStrategy'`: A replication strategy that allows to set the replication factor independently for each data-center. The rest of the sub-options are key-value pairs where each time the key is the name of a datacenter and the value the replication factor for that data-center.

Attempting to create an already existing keyspace 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.

USE

Syntax:

bc(syntax). ::= USE

Sample:

bc(sample). USE myApp;

The `USE` statement takes an existing keyspace name as argument and set it as the per-connection current working keyspace. All subsequent keyspace-specific actions will be performed in the context of the selected keyspace, unless **otherwise specified**, until another USE statement is issued or the connection terminates.

ALTER KEYSPACE

Syntax:

bc(syntax)..
::= ALTER KEYSPACE (IF EXISTS)? WITH
p.
Sample:

bc(sample)..
ALTER KEYSPACE Excelsior
WITH replication = \{’class’: `SimpleStrategy', `replication_factor' : 4};

The `ALTER KEYSPACE` statement alters the properties of an existing keyspace. The supported `<properties>` are the same as for the **`CREATE KEYSPACE`** statement.

DROP KEYSPACE

Syntax:

bc(syntax). ::= DROP KEYSPACE (IF EXISTS)?

Sample:

bc(sample). DROP KEYSPACE myApp;

A `DROP KEYSPACE` statement results in the immediate, irreversible removal of an existing keyspace, including all column families in it, and all data contained in those column families.

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

Syntax:

bc(syntax)..
::= CREATE (TABLE | COLUMNFAMILY) (IF NOT EXISTS)?
`(' ( `,')* `)'
(WITH (AND)*)?

::= (STATIC)? (PRIMARY KEY)?
| PRIMARY KEY `(' ( `,')* `)'

::=
| `(' (,' )* `)'

::=
| COMPACT STORAGE
| CLUSTERING ORDER
p.
Sample:

bc(sample)..
CREATE TABLE monkeySpecies (
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' };
p.
The `CREATE TABLE` statement creates a new table. Each such table is a set of *rows* (usually representing related entities) for which it defines a number of properties. A table is defined by a **name**, it defines the columns composing rows of the table and have a number of **options**. Note that the `CREATE COLUMNFAMILY` syntax is supported as an alias for `CREATE TABLE` (for historical reasons).

Attempting to create an already existing table 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 table already exists.

<tablename>

Valid table names are the same as valid **keyspace names** (up to 32 characters long alphanumerical identifiers). If the table name is provided alone, the table is created within the current keyspace (see `USE`), but if it is prefixed by an existing keyspace name (see `<tablename>` grammar), it is created in the specified keyspace (but does **not** change the current keyspace).

<column-definition>

A `CREATE TABLE` statement defines the columns that rows of the table can have. A *column* is defined by its name (an identifier) and its type (see the [data types] section for more details on allowed types and their properties).

Within a table, a row is uniquely identified by its `PRIMARY KEY` (or more simply the key), and hence all table definitions **must** define a PRIMARY KEY (and only one). A `PRIMARY KEY` is composed of one or more of the columns defined in the table. If the `PRIMARY KEY` is only one column, this can be specified directly after the column definition. Otherwise, it must be specified by following `PRIMARY KEY` by the comma-separated list of column names composing the key within parenthesis. Note that:

bc(sample).
CREATE TABLE t (
k int PRIMARY KEY,
other text
)

is equivalent to

bc(sample).
CREATE TABLE t (
k int,
other text,
PRIMARY KEY (k)
)

Partition key and clustering columns

In CQL, the order in which columns are defined for the `PRIMARY KEY` matters. The first column of the key is called the *partition key*. It has the property that all the rows sharing the same partition key (even across table in fact) are stored on the same physical node. Also, insertion/update/deletion on rows sharing the same partition key for a given table are performed *atomically* and in *isolation*. Note that it is possible to have a composite partition key, i.e. a partition key formed of multiple columns, using an extra set of parentheses to define which columns forms the partition key.

The remaining columns of the `PRIMARY KEY` definition, if any, are called __clustering columns. On a given physical node, rows for a given partition key are stored in the order induced by the clustering columns, making the retrieval of rows in that clustering order particularly efficient (see `SELECT`).

`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 instance, in:

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

the last query will return `'static1'` as value for `s`, since `s` is static and thus the 2nd insertion modified this `` `shared'' value. Note however that static columns are only static within a given partition, and if in the example above both rows where from different partitions (i.e. if they had different value for `pk`), then the 2nd insertion would not have modified the value of `s` for the first row.

A few restrictions applies to when static columns are allowed:

- tables with the `COMPACT STORAGE` option (see below) cannot have them
- 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

`<option>`

The `CREATE TABLE` statement supports a number of options that controls the configuration of a new table. These options can be specified after the `WITH` keyword.

The first of these option is `COMPACT STORAGE`. This option is mainly targeted towards backward compatibility for definitions created before CQL3 (see www.datastax.com/dev/blog/thrift-to-cql3 for more details). The option also provides a slightly more compact layout of data on disk but at the price of diminished flexibility and extensibility for the table. Most notably, `COMPACT STORAGE` tables cannot have collections nor static columns and a `COMPACT STORAGE` table with at least one clustering column supports exactly one (as in not 0 nor more than 1) column not part of the `PRIMARY KEY` definition (which imply in particular that you cannot add nor remove columns after creation). For those reasons, `COMPACT STORAGE` is not recommended outside of the backward compatibility reason evoked above.

Another option is `CLUSTERING ORDER`. It allows to define the ordering of rows on disk. It takes the list of the clustering column names with, for each of them, the on-disk order (Ascending or descending). Note that this option affects [what `ORDER BY` are allowed during `SELECT`].

Table creation supports the following other `<property>`:

option	kind	default	description
`comment`	*simple*	none	A free-form, human-readable comment.
`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 impact the size of bloom filters in-memory and on-disk)
`default_time_to_live`	*simple*	0	The default expiration time (``TTL'') in seconds for a table.
`compaction`	*map*	*see below*	Compaction options, see **below**.
`compression`	*map*	*see below*	Compression options, see **below**.
`caching`	*map*	*see below*	Caching options, see **below**.

option	kind	default	description
`crc_check_chance`	simple	1.0	This option defines the probability with which checksums should be checked during reads to detect bit rot and prevent the propagation of corruption to other replicas. The default value is 1 to apply a checksum every time a data chunk is read. Set to 0 to disable checksum checking and to 0.5 for instance to check every other read. Due to technical limitations we only currently apply this for compressed files. If compression is not enabled on the table, no checksums will be verified.

Compaction options

The `compaction` property must at least define the `'class'` sub-option, that defines the compaction strategy class to use. The default supported class are `'SizeTieredCompactionStrategy'`, `'LeveledCompactionStrategy'` and `'TimeWindowCompactionStrategy'`. Custom strategy can be provided by specifying the full class name as a [string constant]. The rest of the sub-options depends on the chosen class. The sub-options supported by the default classes are:

option	supported compaction strategy	default	description
`enabled`	*all*	true	A boolean denoting whether compaction should be enabled or not.

option	supported compaction strategy	default	description
`tombstone_threshold`	*all*	0.2	A ratio such that if a sstable has more than this ratio of gcable tombstones over all contained columns, the sstable will be compacted (with no other sstables) for the purpose of purging those tombstones.
`tombstone_compaction_interval`	*all*	1 day	The minimum time to wait after an sstable creation time before considering it for `tombstone compaction'', where` tombstone compaction'' is the compaction triggered if the sstable has more gcable tombstones than `tombstone_threshold`.
`unchecked_tombstone_compaction`	*all*	false	Setting this to true enables more aggressive tombstone compactions - single sstable tombstone compactions will run without checking how likely it is that they will be successful.

option	supported compaction strategy	default	description
`min_sstable_size`	SizeTieredCompactionStrategy	50MB	The size tiered strategy groups SSTables to compact in buckets. A bucket groups SSTables that differs from less than 50% in size. However, for small sizes, this would result in a bucketing that is too fine grained. `min_sstable_size` defines a size threshold (in bytes) below which all SSTables belong to one unique bucket
`min_threshold`	SizeTieredCompactionStrategy	4	Minimum number of SSTables needed to start a minor compaction.
`max_threshold`	SizeTieredCompactionStrategy	32	Maximum number of SSTables processed by one minor compaction.
`bucket_low`	SizeTieredCompactionStrategy	0.5	Size tiered consider sstables to be within the same bucket if their size is within [average_size * `bucket_low`, average_size * `bucket_high`] (i.e the default groups sstable whose sizes diverges by at most 50%)
`bucket_high`	SizeTieredCompactionStrategy	1.5	Size tiered consider sstables to be within the same bucket if their size is within [average_size * `bucket_low`, average_size * `bucket_high`] (i.e the default groups sstable whose sizes diverges by at most 50%).

option	supported compaction strategy	default	description
`sstable_size_in_mb`	LeveledCompactionStrategy	5MB	The target size (in MB) for sstables in the leveled strategy. Note that while sstable sizes should stay less or equal to `sstable_size_in_mb`, it is possible to exceptionally have a larger sstable as during compaction, data for a given partition key are never split into 2 sstables
`timestamp_resolution`	TimeWindowCompactionStrategy	MICROSECONDS	The timestamp resolution used when inserting data, could be MILLISECONDS, MICROSECONDS etc (should be understandable by Java TimeUnit) - don't change this unless you do mutations with USING TIMESTAMP (or equivalent directly in the client)
`compaction_window_unit`	TimeWindowCompactionStrategy	DAYS	The Java TimeUnit used for the window size, set in conjunction with `compaction_window_size`. Must be one of DAYS, HOURS, MINUTES
`compaction_window_size`	TimeWindowCompactionStrategy	1	The number of `compaction_window_unit` units that make up a time window.

option	supported compaction strategy	default	description
`unsafe_aggressive_sstable_expiration`	TimeWindowCompactionStrategy	false	Expired sstables will be dropped without checking its data is shadowing other sstables. This is a potentially risky option that can lead to data loss or deleted data re-appearing, going beyond what `unchecked_tombstone_compaction` does for single sstable compaction. Due to the risk the jvm must also be started with `-Dcassandra.unsafe_aggressive_sstable_expiration=true`.

Compression options

For the `compression` property, the following sub-options are available:

option	default	description			
`class`	LZ4Compressor	The compression algorithm to use. Default compressor are: LZ4Compressor , SnappyCompressor and DeflateCompressor. Use `'enabled' : false` to disable compression. Custom compressor can be provided by specifying the full class name as a [string constant].			
`enabled`	true	By default compression is enabled. To disable it, set `enabled` to `false`	`chunk_length_in_kb`	64KB	On disk SSTables are compressed by block (to allow random reads). This defines the size (in KB) of said block. Bigger values may improve the compression rate, but increases the minimum size of data to be read from disk for a read

Caching options

For the `caching` property, 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. Other possible options are `ALL`, to cache all rows of a queried partition, or `NONE` to disable row caching.

Other considerations:

- When **inserting** / **updating** a given row, not all columns needs to be defined (except for those part of the key), and missing columns occupy no space on disk. Furthermore, adding new columns (see `ALTER TABLE`) is a constant time operation. There is thus no need to try to anticipate future usage (or to cry when you haven't) when creating a table.

ALTER TABLE

Syntax:

bc(syntax)..
::= ALTER (TABLE | COLUMNFAMILY) (IF EXISTS)?

::= ADD (IF NOT EXISTS)?
| ADD (IF NOT EXISTS)? ((,)*)
| DROP (IF EXISTS)?
| DROP (IF EXISTS)? ((,)*)
| RENAME (IF EXISTS)? TO (AND TO)*
| WITH (AND)*
p.
Sample:

bc(sample)..
ALTER TABLE addamsFamily

ALTER TABLE addamsFamily
ADD gravesite varchar;

ALTER TABLE addamsFamily
WITH comment = `A most excellent and useful column family';
p.

The `ALTER` statement is used to manipulate table definitions. It allows for adding new columns, dropping existing ones, or updating the table options. As with table creation, `ALTER COLUMNFAMILY` is allowed as an alias for `ALTER TABLE`. 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.

The `<tablename>` is the table name optionally preceded by the keyspace name. The `<instruction>` defines the alteration to perform:

- `ADD`: Adds a new column to the table. The `<identifier>` for the new column must not conflict with an existing column. Moreover, columns cannot be added to tables defined with the `COMPACT STORAGE` option. 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`: Removes a column from the table. Dropped columns will immediately become unavailable in the queries and will not be included in compacted sstables in the future. If a column is readded, queries won't return values written before the column was last dropped. It is assumed that timestamps represent actual time, so if this is not your case, you should NOT read previously dropped columns. Columns can't be dropped from tables defined with the `COMPACT STORAGE` option. 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.
- `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.
- `WITH`: Allows to update the options of the table. The **supported** `<option>` (and syntax) are the same as for the `CREATE TABLE` statement except that `COMPACT STORAGE` is not supported. Note that setting any `compaction` sub-options has the effect of erasing all previous `compaction` options, so you need to re-specify all the sub-options if you want to keep them. The same note applies to the set of `compression` sub-options.

CQL type compatibility:

CQL data types may be converted only as the following table.

Data type may be altered to:	Data type
timestamp	bigint
ascii, bigint, boolean, date, decimal, double, float, inet, int, smallint, text, time, timestamp, timeuuid, tinyint, uuid, varchar, varint	blob
int	date
ascii, varchar	text
bigint	time

Data type may be altered to:	Data type
bigint	timestamp
timeuuid	uuid
ascii, text	varchar
bigint, int, timestamp	varint

Clustering columns have stricter requirements, only the below conversions are allowed.

Data type may be altered to:	Data type
ascii, text, varchar	blob
ascii, varchar	text
ascii, text	varchar

DROP TABLE

Syntax:

bc(syntax). ::= DROP TABLE (IF EXISTS)?

Sample:

bc(sample). DROP TABLE worldSeriesAttendees;

The `DROP TABLE` statement results in the immediate, irreversible removal of a table, including all data contained in it. As for table creation, `DROP COLUMNFAMILY` is allowed as an alias for `DROP TABLE`.

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.

TRUNCATE

Syntax:

bc(syntax). ::= TRUNCATE (TABLE | COLUMNFAMILY)?

Sample:

bc(sample). TRUNCATE superImportantData;

The `TRUNCATE` statement permanently removes all data from a table.

CREATE INDEX

The `CREATE INDEX` statement is used to create a new secondary index for a given (existing) column in a

given table. A name for the index itself can be specified before the `ON` keyword, if desired.

Syntax:

bc(syntax)..
::= CREATE (CUSTOM)? INDEX (IF NOT EXISTS)? ( )?
ON `(' `)'
(USING (WITH OPTIONS =)?)?

::=
| keys( )
p.
Sample:

bc(sample).
CREATE INDEX userIndex ON NerdMovies (user);
CREATE INDEX ON Mutants (abilityId);
CREATE INDEX ON users (keys(favs));
CREATE INDEX ON users (age) USING 'sai';
CREATE CUSTOM INDEX ON users (email) USING `path.to.the.IndexClass';
CREATE CUSTOM INDEX ON users (email) USING `path.to.the.IndexClass' WITH OPTIONS = \{'storage': `/mnt/ssd/indexes/'};

If data already exists for the column, it will be indexed asynchronously. After the index is created, new data for the column is indexed automatically at insertion time. Attempting to create an already existing index 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 index already exists.

Index Types

The `USING` keyword optionally specifies an index type. There are two built-in types:

- legacy_local_table - (default) legacy secondary index, implemented as a hidden local table
- sai - "storage-attched" index, implemented via optimized SSTable/Memtable-attached indexes

To create a custom index, a fully qualified class name must be specified.

Indexes on Map Keys

When creating an index on a **map column**, you may index either the keys or the values. If the column identifier is placed within the `keys()` function, the index will be on the map keys, allowing you to use `CONTAINS KEY` in `WHERE` clauses. Otherwise, the index will be on the map values.

DROP INDEX

Syntax:

bc(syntax). ::= DROP INDEX (IF EXISTS)? (`.')?

Sample:

bc(sample)..
DROP INDEX userIndex;

DROP INDEX userkeyspace.address_index;
p.
The `DROP INDEX` statement is used to drop an existing secondary index. The argument of the statement is the index name, which may optionally specify the keyspace of the index.

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

CREATE MATERIALIZED VIEW

Syntax:

bc(syntax)..
::= CREATE MATERIALIZED VIEW (IF NOT EXISTS)? AS
SELECT (`(' ( `,') * `)' | `')
FROM
(WHERE)?
PRIMARY KEY `(' ( `,') `)'
(WITH (AND)*)?
p.
Sample:

bc(sample)..
CREATE MATERIALIZED VIEW monkeySpecies_by_population AS
SELECT *
FROM monkeySpecies
WHERE population IS NOT NULL AND species IS NOT NULL
PRIMARY KEY (population, species)
WITH comment=`Allow query by population instead of species';
p.
`The `CREATE MATERIALIZED VIEW` statement creates a new materialized view. Each such view is a set of *rows* which corresponds to rows which are present in the underlying, or base, table specified in the `SELECT` statement. A materialized view cannot be directly updated, but updates to the base table will cause corresponding updates in the view.

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

`WHERE` Clause

The `<where-clause>` is similar to the [where clause of a `SELECT` statement], with a few differences. First, the where clause must contain an expression that disallows `NULL` values in columns in the view's primary key. If no other restriction is desired, this can be accomplished with an `IS NOT NULL` expression. Second, only columns which are in the base table's primary key may be restricted with expressions other than `IS NOT NULL`. (Note that this second restriction may be lifted in the future.)

ALTER MATERIALIZED VIEW

Syntax:

bc(syntax). ::= ALTER MATERIALIZED VIEW
WITH (AND)*

The `ALTER MATERIALIZED VIEW` statement allows options to be update; these options are the same as `CREATE TABLE`'s options.

DROP MATERIALIZED VIEW

Syntax:

bc(syntax). ::= DROP MATERIALIZED VIEW (IF EXISTS)?

Sample:

bc(sample). DROP MATERIALIZED VIEW monkeySpecies_by_population;

The `DROP MATERIALIZED VIEW` statement is used to drop an existing materialized view.

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

CREATE TYPE

Syntax:

bc(syntax)..
::= CREATE TYPE (IF NOT EXISTS)?
`(' ( `,')* `)'

::= (`.')?

::=

Sample:

bc(sample)..

CREATE TYPE address (
street_name text,
street_number int,
city text,
state text,
zip int
)

CREATE TYPE work_and_home_addresses (
home_address address,
work_address address
)
p.
The `CREATE TYPE` statement creates a new user-defined type. Each type is a set of named, typed fields. Field types may be any valid type, including collections and other existing user-defined types.

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.

`<typename>`

Valid type names are identifiers. The names of existing CQL types and **reserved type names** may not be used.

If the type name is provided alone, the type is created with the current keyspace (see `USE`). If it is prefixed by an existing keyspace name, the type is created within the specified keyspace instead of the current keyspace.

ALTER TYPE

Syntax:

bc(syntax)..
::= ALTER TYPE (IF EXISTS)?

::= ADD (IF NOT EXISTS)?
| RENAME (IF EXISTS)? TO (AND TO)*
p.
Sample:

bc(sample)..
ALTER TYPE address ADD country text

ALTER TYPE address RENAME zip TO zipcode AND street_name TO street
p.
The `ALTER TYPE` statement is used to manipulate type definitions. It allows for adding new fields, renaming existing fields, or changing the type of existing fields. 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.

DROP TYPE

Syntax:

bc(syntax)..
::= DROP TYPE (IF EXISTS)?
p.
The `DROP TYPE` statement results in the immediate, irreversible removal of a type. Attempting to drop a type that is still in use by another type or a table will result in an error.

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

CREATE TRIGGER

Syntax:

bc(syntax)..
::= CREATE TRIGGER (IF NOT EXISTS)? ( )?
ON
USING

Sample:

bc(sample).
CREATE TRIGGER myTrigger ON myTable USING `org.apache.cassandra.triggers.InvertedIndex';

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

DROP TRIGGER

Syntax:

bc(syntax)..
::= DROP TRIGGER (IF EXISTS)? ( )?
ON
p.
Sample:

bc(sample).
DROP TRIGGER myTrigger ON myTable;

`DROP TRIGGER` statement removes the registration of a trigger created using `CREATE TRIGGER`.

CREATE FUNCTION

Syntax:

bc(syntax)..
::= CREATE (OR REPLACE)?
FUNCTION (IF NOT EXISTS)?
(`.')?
`(' ( `,')* `)'
(CALLED | RETURNS NULL) ON NULL INPUT
RETURNS
LANGUAGE
AS

Sample:

bc(sample).
CREATE OR REPLACE FUNCTION somefunction
(somearg int, anotherarg text, complexarg frozen, listarg list)
RETURNS NULL ON NULL INPUT
RETURNS text
LANGUAGE java
AS + ;
CREATE FUNCTION akeyspace.fname IF NOT EXISTS
(someArg int)
CALLED ON NULL INPUT
RETURNS text
LANGUAGE java
AS + ;

`CREATE FUNCTION` creates or replaces a user-defined function.

Function Signature

Signatures are used to distinguish individual functions. The signature consists of:

1. The fully qualified function name - i.e *keyspace* plus *function-name*
2. The concatenated list of all argument types

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

`CREATE FUNCTION` with the optional `OR REPLACE` keywords either creates a function or replaces an existing one with the same signature. A `CREATE FUNCTION` without `OR REPLACE` fails if a function with the same

signature already exists.

Behavior on invocation with `null` values must be defined for each function. There are two options:

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

If the optional `IF NOT EXISTS` keywords are used, the function will only be created if another function with the same signature does not exist.

`OR REPLACE` and `IF NOT EXIST` cannot be used together.

Functions belong to a keyspace. If no keyspace is specified in `<function-name>`, the current keyspace is used (i.e. the keyspace specified using the `USE` statement). It is not possible to create a user-defined function in one of the system keyspaces.

See the section on **user-defined functions** for more information.

DROP FUNCTION

Syntax:

bc(syntax)..
::= DROP FUNCTION (IF EXISTS)?
(`.')?
(`(' ( `,')* `)')?

Sample:

bc(sample).
DROP FUNCTION myfunction;
DROP FUNCTION mykeyspace.afunction;
DROP FUNCTION afunction (int);
DROP FUNCTION afunction (text);

`DROP FUNCTION` statement removes a function created using `CREATE FUNCTION`.
You must specify the argument types (**signature**) of the function to drop if there are multiple functions with the same name but a different signature (overloaded functions).

`DROP FUNCTION` with the optional `IF EXISTS` keywords drops a function if it exists.

CREATE AGGREGATE

Syntax:

bc(syntax)..

::= CREATE (OR REPLACE)?
AGGREGATE (IF NOT EXISTS)?
(`.')?
`(' ( `,')* `)'
SFUNC
STYPE
(FINALFUNC)?
(INITCOND)?
p.
Sample:

bc(sample).
CREATE AGGREGATE myaggregate (val text)
SFUNC myaggregate_state
STYPE text
FINALFUNC myaggregate_final
INITCOND `foo';

See the section on **user-defined aggregates** for a complete example.

`CREATE AGGREGATE` creates or replaces a user-defined aggregate.

`CREATE AGGREGATE` with the optional `OR REPLACE` keywords either creates an aggregate or replaces an existing one with the same signature. A `CREATE AGGREGATE` without `OR REPLACE` fails if an aggregate with the same signature already exists.

`CREATE AGGREGATE` with the optional `IF NOT EXISTS` keywords either creates an aggregate if it does not already exist.

`OR REPLACE` and `IF NOT EXIST` cannot be used together.

Aggregates belong to a keyspace. If no keyspace is specified in `<aggregate-name>`, the current keyspace is used (i.e. the keyspace specified using the `USE` statement). It is not possible to create a user-defined aggregate in one of the system keyspaces.

Signatures for user-defined aggregates follow the **same rules** as for user-defined functions.

`STYPE` defines the type of the state value and must be specified.

The optional `INITCOND` defines the initial state value for the aggregate. It defaults to `null`. A non-`null` `INITCOND` must be specified for state functions that are declared with `RETURNS NULL ON NULL INPUT`.

`SFUNC` references an existing function to be used as the state modifying function. The type of first argument of the state function must match `STYPE`. The remaining argument types of the state function must match the argument types of the aggregate function. State is not updated for state functions declared with `RETURNS NULL ON NULL INPUT` and called with `null`.

The optional `FINALFUNC` is called just before the aggregate result is returned. It must take only one argument with type `STYPE`. The return type of the `FINALFUNC` may be a different type. A final function declared with `RETURNS NULL ON NULL INPUT` means that the aggregate's return value will be `null`, if the last state is `null`.

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

See the section on **user-defined aggregates** for more information.

DROP AGGREGATE

Syntax:

bc(syntax)..
::= DROP AGGREGATE (IF EXISTS)?
(`.')?
(`(' ( `,')* `)')?
p.

Sample:

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

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

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

Signatures for user-defined aggregates follow the **same rules** as for user-defined functions.

Data Manipulation

INSERT

Syntax:

bc(syntax)..
::= INSERT INTO
((VALUES)

| (JSON))
(IF NOT EXISTS)?
(USING (AND)*)?

::= `(' ( `,')* `)'

::= `(' ( `,')* `)'

::= TIMESTAMP
| TTL
p.
Sample:

bc(sample)..
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}'
p.
The `INSERT` statement writes one or more columns for a given row in a table. Note that since a row is identified by its `PRIMARY KEY`, at least the columns composing it must be specified. The list of columns to insert to must be supplied when using the `VALUES` syntax. When using the `JSON` syntax, they are optional. See the section on **`INSERT JSON`** for more details.

Note that 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 mean to know which of creation or update happened.

It is however possible to use the `IF NOT EXISTS` condition to only insert if the row does not exist prior to the insertion. But please note that using `IF NOT EXISTS` will incur a non negligible performance cost (internally, Paxos will be used) so this should be used sparingly.

All updates for an `INSERT` are applied atomically and in isolation.

Please refer to the **`UPDATE`** section for information on the `<option>` available and to the **collections** section for use of `<collection-literal>`. Also note that `INSERT` does not support counters, while `UPDATE` does.

UPDATE

Syntax:

bc(syntax)..
::= UPDATE
(USING (AND)*)?
SET (`,')*

WHERE
(IF (AND condition)*)?

::= ='
| `=' (' | `-') (| |) + | `=' `'
| `[' `]' `='
| `.' `='

::=
| CONTAINS (KEY)?
| IN
| `[' `]'
| `[' `]' IN
| `.'
| `.' IN

::= `<' | `⇐' | `=' | `!=' | `>=' | `>'
::= (| `(' ( ( `,')*)? `)')

::= (AND)*

::= ='
| `(' (,' )* )' `='
| IN `(' ( ( `,')*)? `)'
| IN
| `(' (,' )* )' IN `(' ((`,' )* )? `)'
| `(' (,' )* `)' IN

::= TIMESTAMP
| TTL
p.
Sample:

bc(sample)..
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';
p.
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 composing the PRIMARY KEY. Other columns values are specified through <assignment> after the SET keyword.

Note that unlike in SQL, `UPDATE` does not check the prior existence of the row by default (except through the use of `<condition>`, see below): the row is created if none existed before, and updated otherwise. Furthermore, there are no means to know whether a creation or update occurred.

It is however possible to use the conditions on some columns through `IF`, in which case the row will not be updated unless the conditions are met. But, please note that using `IF` conditions will incur a non-negligible performance cost (internally, Paxos will be used) so this should be used sparingly.

In an `UPDATE` statement, all updates within the same partition key are applied atomically and in isolation.

The `c = c + 3` form of `<assignment>` is used to increment/decrement counters. The identifier after the `=' sign **must** be the same than the one before the `=' sign (Only increment/decrement is supported on counters, not the assignment of a specific value).

The `id = id + <collection-literal>` and `id[value1] = value2` forms of `<assignment>` are for collections. Please refer to the **relevant section** for more details.

The `id.field = <term>` form of `<assignemt>` is for setting the value of a single field on a non-frozen user-defined types.

`<options>`

The `UPDATE` and `INSERT` statements support the following options:

- `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.
- `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.

DELETE

Syntax:

bc(syntax)..
::= DELETE ((`,')*)?
FROM
(USING TIMESTAMP)?
WHERE
(IF (EXISTS | ((AND)*)))?

::=

| `[' `]'
| `.'

::= (AND)*

::=
| (' (,')*)'
| IN `(' ( ( `,')*)? `)'
| IN
| `(' (,' )* )' IN `(' ((`,' )* )? `)'
| `(' (,' )* `)' IN

::= `=' | `<' | `>' | `⇐' | `>='
::= (| `(' ( ( `,')*)? `)')

::= (| `!=')
| CONTAINS (KEY)?
| IN
| `[' `]' (| `!=')
| `[' `]' IN
| `.' ( | `!=')
| `.' IN

Sample:

bc(sample)..
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);
p.
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>. The id[value] syntax in <selection> is for non-frozen collections (please refer to the **collection section** for more details). The id.field syntax is for the deletion of non-frozen user-defined types. 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 clause. A range of rows may be deleted using an inequality operator (such as >=).

DELETE supports the TIMESTAMP option with the same semantics as the **UPDATE** statement.

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 (internally, Paxos will be used) and so should be used sparingly.

BATCH

Syntax:

bc(syntax)..
::= BEGIN (UNLOGGED | COUNTER) BATCH
(USING (AND)*)?
(`;')*
APPLY BATCH

::=
|
|

::= TIMESTAMP
p.
Sample:

bc(sample).
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:

1. It saves network round-trips between the client and the server (and sometimes between the server coordinator and the replicas) when batching multiple updates.
2. All updates in a `BATCH` belonging to a given partition key are performed in isolation.
3. 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`** for more details.

Note that:

- `BATCH` statements may only contain `UPDATE`, `INSERT` and `DELETE` statements.
- 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. 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.

`UNLOGGED`

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`

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

`<option>`

`BATCH` supports both the `TIMESTAMP` option, with similar semantic to the one described in the **`UPDATE`** statement (the timestamp applies to all the statement inside the batch). However, if used, `TIMESTAMP` **must not** be used in the statements within the batch.

Queries

SELECT

Syntax:

bc(syntax)..
::= SELECT (JSON)?
FROM
(WHERE)?
(GROUP BY)?
(ORDER BY)?
(PER PARTITION LIMIT)?
(LIMIT)?
(ALLOW FILTERING)?

::= DISTINCT?

::= (AS)? (`,' (AS)?)*
| `*'

::=
|
| WRITETIME `(' `)'
| MAXWRITETIME `(' `)'
| COUNT `(' `' `)'
| TTL `(' `)'
| CAST `(' AS `)'

| '(' ((,')? `)'
| `.'
| `[' `]'
| `[' ? .. ? `]'

::= (AND)*

::=
| (' (,')*)'
| IN '(' ((',')*)? ')'
| '(' (,')* `)' IN `(' ((`,' )* )? `)'
| TOKEN `(' ( `,')* `)'

::= =' | '<' | '>' | '⇐' | '>=' | CONTAINS | CONTAINS KEY
::= (,')*
::= (,')*
::= (ASC | DESC)?
::= '(' (,')* `)'
p.
Sample:

bc(sample)..
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 (*) FROM users;

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 rows, where each row contains the collection of columns corresponding to the query. If the JSON keyword is used, the results for each row will contain only a single column named 'json''. See the section on 'SELECT JSON for more details.

<select-clause>

The <select-clause> determines which columns needs to be queried and returned in the result-set. It consists of either the comma-separated list of or the wildcard character (*) to select all the columns defined for the table. Please note that for wildcard SELECT queries the order of columns returned is not

specified and is not guaranteed to be stable between Cassandra versions.

A `<selector>` is either a column name to retrieve or a `<function>` of one or more `<term>`s. The function allowed are the same as for `<term>` and are described in the **function section**. In addition to these generic functions, the `WRITETIME` and `MAXWRITETIME` (resp. `TTL`) function allows to select the timestamp of when the column was inserted (resp. the time to live (in seconds) for the column (or null if the column has no expiration set)) and the `CAST` function can be used to convert one data type to another. The `WRITETIME` and `TTL` functions can't be used on multi-cell columns such as non-frozen collections or non-frozen user-defined types.

Additionally, individual values of maps and sets can be selected using `[ <term> ]`. For maps, this will return the value corresponding to the key, if such entry exists. For sets, this will return the key that is selected if it exists and is thus mainly a way to check element existence. It is also possible to select a slice of a set or map with `[ <term> ... <term> `], where both bound can be omitted.

Any `<selector>` can be aliased using `AS` keyword (see examples). Please note that `<where-clause>` and `<order-by>` clause should refer to the columns by their original names and not by their aliases.

The `COUNT` keyword can be used with parenthesis enclosing `*`. If so, the query will return a single result: the number of rows matching the query. Note that `COUNT(1)` is supported as an alias.

`<where-clause>`

The `<where-clause>` specifies which rows must be queried. It is composed of relations on the columns that are part of the `PRIMARY KEY` and/or have a [secondary index] defined on them.

Not all relations are allowed in a query. For instance, non-equal relations (where `IN` is considered as an equal relation) on a partition key are not supported (but see the use of the `TOKEN` method below to do non-equal queries on the partition key). Moreover, for a given partition key, the clustering columns induce an ordering of rows and relations on them is restricted to the relations that allow to select a **contiguous** (for the ordering) set of rows. For instance, given

bc(sample).
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:

bc(sample).
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):

bc(sample).
SELECT entry_title, content FROM posts WHERE userid=`john doe' AND posted_at >= `2012-01-01' AND posted_at < `2012-01-31'

When specifying relations, the `TOKEN` function can be used on the `PARTITION KEY` column to query. In that case, rows will be selected based on the token of their `PARTITION_KEY` rather than on the value. Note that 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). Example:

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

Moreover, the `IN` relation is only allowed on the last column of the partition key and 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 instance:

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

will request all rows that sorts 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'`, which wouldn't be the case for:

bc(sample).
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`:

bc(sample).
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 on 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.

`<order-by>`

The `ORDER BY` option allows to select the order of the returned results. It takes as argument a list of column names along with the order for the column (`ASC` for ascendant and `DESC` for descendant, omitting the order being equivalent to `ASC`). Currently the possible orderings are limited (which depends on the table `CLUSTERING ORDER`):

- if the table has been defined without any specific `CLUSTERING ORDER`, then then allowed orderings are the order induced by the clustering columns and the reverse of that one.
- otherwise, the orderings allowed are the order of the `CLUSTERING ORDER` option and the reversed one.

`<group-by>`

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

Using the `GROUP BY` option, it is only possible to group rows at the partition key level or at a clustering column level. By consequence, the `GROUP BY` option only accept as arguments primary key column names in the primary key order. If a primary key column is restricted by an equality restriction it is not required to be present 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.

`LIMIT` and `PER PARTITION LIMIT`

The `LIMIT` option to a `SELECT` statement limits the number of rows returned by a query, while the `PER PARTITION LIMIT` option limits the number of rows returned for a given partition by the query. Note that both type of limit can used in the same statement.

`ALLOW FILTERING`

By default, CQL only allows select queries that don't involve `filtering'' server side, i.e. queries where we know that all (live) record read will be returned (maybe partly) in the result set. The reasoning is that those` non filtering" queries have predictable performance in the sense that they will execute in a time that is proportional to the amount of data **returned** by the query (which can be controlled through `LIMIT`).

The `ALLOW FILTERING` option allows to explicitly allow (some) queries that require filtering. Please note that a query using `ALLOW FILTERING` may thus have unpredictable performance (for the definition above), i.e. even a query that selects a handful of records **may** exhibit performance that depends on the total amount of data stored in the cluster.

For instance, considering the following table holding user profiles with their year of birth (with a secondary index on it) and country of residence:

bc(sample)..
CREATE TABLE users (
username text PRIMARY KEY,
firstname text,
lastname text,
birth_year int,
country text
)

CREATE INDEX ON users(birth_year);
p.

Then the following queries are valid:

bc(sample).
SELECT * FROM users;
SELECT firstname, lastname FROM users WHERE birth_year = 1981;

because in both case, Cassandra guarantees that these queries performance will be proportional to the amount of data returned. In particular, if no users are born in 1981, then the second query performance will not depend of the number of user profile stored in the database (not directly at least: due to secondary index implementation consideration, this query may still depend on the number of node in the cluster, which indirectly depends on the amount of data stored. Nevertheless, the number of nodes will always be multiple number of magnitude lower than the number of user profile stored). Of course, both query may return very large result set in practice, but the amount of data returned can always be controlled by adding a `LIMIT`.

However, the following query will be rejected:

bc(sample).
SELECT firstname, lastname FROM users WHERE birth_year = 1981 AND country = `FR';

because Cassandra cannot guarantee that it won't have to scan large amount of data even if the result to those query is small. Typically, it will scan all the index entries for users born in 1981 even if only a handful are actually from France. However, if you `know what you are doing'', you can force the execution of this query by using `ALLOW FILTERING` and so the following query is valid:

bc(sample).
SELECT firstname, lastname FROM users WHERE birth_year = 1981 AND country = `FR' ALLOW FILTERING;

Database Roles

CREATE ROLE

Syntax:

bc(syntax)..
::= CREATE ROLE (IF NOT EXISTS)? (WITH (AND)*)?

::= PASSWORD =
| LOGIN =
| SUPERUSER =
| OPTIONS =
p.

Sample:

bc(sample).
CREATE ROLE new_role;
CREATE ROLE alice WITH PASSWORD = `password_a' AND LOGIN = true;
CREATE ROLE bob WITH PASSWORD = `password_b' AND LOGIN = true AND SUPERUSER = true;
CREATE ROLE carlos WITH OPTIONS = \{ `custom_option1' : `option1_value', `custom_option2' : 99 };

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

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

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

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

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

Setting credentials for internal authentication

Use the `WITH PASSWORD` clause to set a password for internal authentication, enclosing the password in single quotation marks.
If internal authentication has not been set up or the role does not have `LOGIN` privileges, the `WITH PASSWORD` clause is not necessary.

Creating a role conditionally

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

bc(sample).
CREATE ROLE other_role;

CREATE ROLE IF NOT EXISTS other_role;

ALTER ROLE

Syntax:

bc(syntax)..
::= ALTER ROLE (IF EXISTS)? (WITH (AND)*)?

::= PASSWORD =
| LOGIN =
| SUPERUSER =
| OPTIONS =
p.

Sample:

bc(sample).
ALTER ROLE bob WITH PASSWORD = `PASSWORD_B' AND SUPERUSER = false;

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

Conditions on executing `ALTER ROLE` statements:

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

DROP ROLE

Syntax:

bc(syntax)..
::= DROP ROLE (IF EXISTS)?
p.

Sample:

bc(sample).
DROP ROLE alice;
DROP ROLE IF EXISTS bob;

`DROP ROLE` requires the client to have `DROP` **permission** on the role in question. In addition, client may

not DROP the role with which it identified at login. Finaly, only a client with SUPERUSER status may DROP another SUPERUSER role.
Attempting to drop a role which does not exist results in an invalid query condition unless the IF EXISTS option is used. If the option is used and the role does not exist the statement is a no-op.

GRANT ROLE

Syntax:

bc(syntax).
::= GRANT TO

Sample:

bc(sample).
GRANT report_writer TO alice;

This statement grants the report_writer role to alice. Any permissions granted to report_writer are also acquired by alice.
Roles are modelled as a directed acyclic graph, so circular grants are not permitted. The following examples result in error conditions:

bc(sample).
GRANT role_a TO role_b;
GRANT role_b TO role_a;

bc(sample).
GRANT role_a TO role_b;
GRANT role_b TO role_c;
GRANT role_c TO role_a;

REVOKE ROLE

Syntax:

bc(syntax).
::= REVOKE FROM

Sample:

bc(sample).
REVOKE report_writer FROM alice;

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

LIST ROLES

Syntax:

bc(syntax).
::= LIST ROLES (OF)? (NORECURSIVE)?

Sample:

bc(sample).
LIST ROLES;

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

bc(sample).
LIST ROLES OF `alice`;

Enumerate all roles granted to `alice`, including those transitively aquired.

bc(sample).
LIST ROLES OF `bob` NORECURSIVE

List all roles directly granted to `bob`.

CREATE USER

Prior to the introduction of roles in Cassandra 2.2, authentication and authorization were based around the concept of a `USER`. For backward compatibility, the legacy syntax has been preserved with `USER` centric statments becoming synonyms for the `ROLE` based equivalents.

Syntax:

bc(syntax)..
::= CREATE USER (IF NOT EXISTS)? (WITH PASSWORD)? ()?

::= SUPERUSER
| NOSUPERUSER
p.

Sample:

bc(sample).
CREATE USER alice WITH PASSWORD `password_a' SUPERUSER;
CREATE USER bob WITH PASSWORD `password_b' NOSUPERUSER;

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

bc(sample)..

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

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

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

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

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

ALTER USER

Syntax:

bc(syntax)..
::= ALTER USER (IF EXISTS)? (WITH PASSWORD)? ( )?

::= SUPERUSER
| NOSUPERUSER
p.

bc(sample).
ALTER USER alice WITH PASSWORD `PASSWORD_A';
ALTER USER bob SUPERUSER;

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

DROP USER

Syntax:

bc(syntax)..
::= DROP USER (IF EXISTS)?
p.

Sample:

bc(sample).
DROP USER alice;

DROP USER IF EXISTS bob;

LIST USERS

Syntax:

bc(syntax).
::= LIST USERS;

Sample:

bc(sample).
LIST USERS;

This statement is equivalent to

bc(sample).
LIST ROLES;

but only roles with the `LOGIN` privilege are included in the output.

Database Identities

ADD IDENTITY

Syntax:

bc(syntax)..
::= ADD IDENTITY (IF NOT EXISTS)? TO ROLE ?

Sample:

bc(sample).
ADD IDENTITY 'id1' TO ROLE 'role1';

Only a user with privileges to add roles can add identities

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

Adding an identity conditionally

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

bc(sample).
ADD IDENTITY IF NOT EXISTS 'id1' TO ROLE 'role1';

DROP IDENTITY

Syntax:

bc(syntax)..
::= DROP IDENTITY (IF EXISTS)?
p.

Sample:

bc(sample).
DROP IDENTITY 'testIdentity';
DROP IDENTITY IF EXISTS 'testIdentity';

Only a user with privileges to drop roles can remove identities

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

Data Control

Permissions

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

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

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

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

The full set of available permissions is:

- `CREATE`
- `ALTER`
- `DROP`
- `SELECT`
- `MODIFY`
- `AUTHORIZE`
- `DESCRIBE`
- `EXECUTE`
- `UNMASK`
- `SELECT_MASKED`

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

permission	resource	operations			
`CREATE`	`ALL KEYSPACES`	`CREATE KEYSPACE` `CREATE TABLE` in any keyspace			
`CREATE`	`KEYSPACE`	`CREATE TABLE` in specified keyspace			
`CREATE`	`ALL FUNCTIONS`	`CREATE FUNCTION` in any keyspace `CREATE AGGREGATE` in any keyspace			
`CREATE`	`ALL FUNCTIONS IN KEYSPACE`	`CREATE FUNCTION` in keyspace `CREATE AGGREGATE` in keyspace			
`CREATE`	`ALL ROLES`	`CREATE ROLE`			

permission	resource	operations			
ALTER	ALL KEYSPACES	ALTER KEYSPACE ALTER TABLE in any keyspace			
ALTER	KEYSPACE	ALTER KEYSPACE ALTER TABLE in keyspace			
ALTER	TABLE	ALTER TABLE			
ALTER	ALL FUNCTIONS	CREATE FUNCTION replacing any existing CREATE AGGREGATE replacing any existing			
ALTER	ALL FUNCTIONS IN KEYSPACE	CREATE FUNCTION replacing existing in keyspace CREATE AGGREGATE replacing any existing in keyspace			
ALTER	FUNCTION	CREATE FUNCTION replacing existing CREATE AGGREGATE replacing existing			
ALTER	ALL ROLES	ALTER ROLE on any role			
ALTER	ROLE	ALTER ROLE			
DROP	ALL KEYSPACES	DROP KEYSPACE DROP TABLE in any keyspace			

permission	resource	operations			
DROP	KEYSPACE	DROP TABLE in specified keyspace			
DROP	TABLE	DROP TABLE			
DROP	ALL FUNCTIONS	DROP FUNCTION in any keyspace DROP AGGREGATE in any existing			
DROP	ALL FUNCTIONS IN KEYSPACE	DROP FUNCTION in keyspace DROP AGGREGATE in existing			
DROP	FUNCTION	DROP FUNCTION			
DROP	ALL ROLES	DROP ROLE on any role			
DROP	ROLE	DROP ROLE			
SELECT	ALL KEYSPACES	SELECT on any table			
SELECT	KEYSPACE	SELECT on any table in keyspace			
SELECT	TABLE	SELECT on specified table			
SELECT	ALL MBEANS	Call getter methods on any mbean			
SELECT	MBEANS	Call getter methods on any mbean matching a wildcard pattern			
SELECT	MBEAN	Call getter methods on named mbean			

permission	resource	operations			
MODIFY	ALL KEYSPACES	INSERT on any table UPDATE on any table DELETE on any table TRUNCATE on any table			
MODIFY	KEYSPACE	INSERT on any table in keyspace UPDATE on any table in keyspace DELETE on any table in keyspace TRUNCATE on any table in keyspace	MODIFY	TABLE	INSERT UPDATE DELETE TRUNCATE
MODIFY	ALL MBEANS	Call setter methods on any mbean			
MODIFY	MBEANS	Call setter methods on any mbean matching a wildcard pattern			
MODIFY	MBEAN	Call setter methods on named mbean			
AUTHORIZE	ALL KEYSPACES	GRANT PERMISSION on any table REVOKE PERMISSION on any table			

permission	resource	operations			
AUTHORIZE	KEYSPACE	GRANT PERMISSION on table in keyspace REVOKE PERMISSION on table in keyspace			
AUTHORIZE	TABLE	GRANT PERMISSION REVOKE PERMISSION			
AUTHORIZE	ALL FUNCTIONS	GRANT PERMISSION on any function REVOKE PERMISSION on any function			
AUTHORIZE	ALL FUNCTIONS IN KEYSPACE	GRANT PERMISSION in keyspace REVOKE PERMISSION in keyspace			
AUTHORIZE	ALL FUNCTIONS IN KEYSPACE	GRANT PERMISSION in keyspace REVOKE PERMISSION in keyspace			
AUTHORIZE	FUNCTION	GRANT PERMISSION REVOKE PERMISSION			
AUTHORIZE	ALL MBEANS	GRANT PERMISSION on any mbean REVOKE PERMISSION on any mbean			

permission	resource	operations			
AUTHORIZE	MBEANS	GRANT PERMISSION on any mbean matching a wildcard pattern REVOKE PERMISSION on any mbean matching a wildcard pattern			
AUTHORIZE	MBEAN	GRANT PERMISSION on named mbean REVOKE PERMISSION on named mbean			
AUTHORIZE	ALL ROLES	GRANT ROLE grant any role REVOKE ROLE revoke any role			
AUTHORIZE	ROLES	GRANT ROLE grant role REVOKE ROLE revoke role			
DESCRIBE	ALL ROLES	LIST ROLES all roles or only roles granted to another, specified role			
DESCRIBE	@ALL MBEANS	Retrieve metadata about any mbean from the platform's MBeanServer			

permission	resource	operations			
DESCRIBE	@MBEANS	Retrieve metadata about any mbean matching a wildcard patter from the platform's MBeanServer			
DESCRIBE	@MBEAN	Retrieve metadata about a named mbean from the platform's MBeanServer			
EXECUTE	ALL FUNCTIONS	SELECT, INSERT, UPDATE using any function use of any function in CREATE AGGREGATE			
EXECUTE	ALL FUNCTIONS IN KEYSPACE	SELECT, INSERT, UPDATE using any function in keyspace use of any function in keyspace in CREATE AGGREGATE			
EXECUTE	FUNCTION	SELECT, INSERT, UPDATE using function use of function in CREATE AGGREGATE			
EXECUTE	ALL MBEANS	Execute operations on any mbean			

permission	resource	operations			
EXECUTE	MBEANS	Execute operations on any mbean matching a wildcard pattern			
EXECUTE	MBEAN	Execute operations on named mbean			
UNMASK	ALL KEYSPACES	See the clear contents of masked columns on any table			
UNMASK	KEYSPACE	See the clear contents of masked columns on any table in keyspace			
UNMASK	TABLE	See the clear contents of masked columns on the specified table			
SELECT_MASKED	ALL KEYSPACES	SELECT restricting masked columns on any table			
SELECT_MASKED	KEYSPACE	SELECT restricting masked columns on any table in specified keyspace			

permission	resource	operations			
SELECT_MASKED	TABLE	SELECT restricting masked columns on the specified table			

GRANT PERMISSION

Syntax:

bc(syntax)..
::= GRANT (ALL (PERMISSIONS)? | (PERMISSION)?) ON TO

::= CREATE | ALTER | DROP | SELECT | MODIFY | AUTHORIZE | DESCRIBE | UNMASK | SELECT_MASKED EXECUTE

::= ALL KEYSPACES
| KEYSPACE
| (TABLE)?
| ALL ROLES
| ROLE
| ALL FUNCTIONS (IN KEYSPACE)?
| FUNCTION
| ALL MBEANS
| (MBEAN | MBEANS)
p.

Sample:

bc(sample).
GRANT SELECT ON ALL KEYSPACES TO data_reader;

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

bc(sample).
GRANT MODIFY ON KEYSPACE keyspace1 TO data_writer;

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

bc(sample).
GRANT DROP ON keyspace1.table1 TO schema_owner;

This gives any user with the `schema_owner` role permissions to `DROP keyspace1.table1`.

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

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

bc(sample).
GRANT DESCRIBE ON ALL ROLES TO role_admin;

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

GRANT ALL

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

Automatic Granting

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

REVOKE PERMISSION

Syntax:

bc(syntax)..
::= REVOKE (ALL (PERMISSIONS)? | (PERMISSION)?) ON FROM

::= CREATE | ALTER | DROP | SELECT | MODIFY | AUTHORIZE | DESCRIBE | UNMASK | SELECT_MASKED EXECUTE

::= ALL KEYSPACES
| KEYSPACE
| (TABLE)?
| ALL ROLES
| ROLE
| ALL FUNCTIONS (IN KEYSPACE)?
| FUNCTION
| ALL MBEANS
| (MBEAN | MBEANS)
p.

Sample:

bc(sample)..

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

LIST PERMISSIONS

Syntax:

bc(syntax)..
::= LIST (ALL (PERMISSIONS)? |)
(ON)?
(OF (NORECURSIVE)?)?

::= ALL KEYSPACES
| KEYSPACE
| (TABLE)?
| ALL ROLES
| ROLE
| ALL FUNCTIONS (IN KEYSPACE)?
| FUNCTION
| ALL MBEANS
| (MBEAN | MBEANS)
p.

Sample:

bc(sample).
LIST ALL PERMISSIONS OF alice;

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

bc(sample).
LIST ALL PERMISSIONS ON keyspace1.table1 OF bob;

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

bc(sample).
LIST SELECT PERMISSIONS OF carlos;

Show any permissions granted to `carlos` or any of `carlos`'s roles, limited to `SELECT` permissions on any

resource.

Data Types

CQL supports a rich set of data types for columns defined in a table, including collection types. On top of those native
and collection types, users can also provide custom types (through a JAVA class extending `AbstractType` loadable by
Cassandra). The syntax of types is thus:

bc(syntax)..
::=
|
|
| // Used for custom types. The fully-qualified name of a JAVA class

::= ascii
| bigint
| blob
| boolean
| counter
| date
| decimal
| double
| float
| inet
| int
| smallint
| text
| time
| timestamp
| timeuuid
| tinyint
| uuid
| varchar
| varint

::= list `<' `>'`
`| set '<' '>'`
`| map '<' ',' '>'`
`::= tuple '<' (',' )* `>'`
p. Note that the native types are keywords and as such are case-insensitive. They are however not reserved ones.

The following table gives additional informations on the native data types, and on which kind of

[constants] each type supports:

type	constants supported	description
`ascii`	strings	ASCII character string
`bigint`	integers	64-bit signed long
`blob`	blobs	Arbitrary bytes (no validation)
`boolean`	booleans	true or false
`counter`	integers	Counter column (64-bit signed value). See **Counters** for details
`date`	integers, strings	A date (with no corresponding time value). See [Working with dates] below for more information.
`decimal`	integers, floats	Variable-precision decimal
`double`	integers	64-bit IEEE-754 floating point
`float`	integers, floats	32-bit IEEE-754 floating point
`inet`	strings	An IP address. It can be either 4 bytes long (IPv4) or 16 bytes long (IPv6). There is no `inet` constant, IP address should be inputed as strings
`int`	integers	32-bit signed int
`smallint`	integers	16-bit signed int
`text`	strings	UTF8 encoded string
`time`	integers, strings	A time with nanosecond precision. See **Working with time** below for more information.
`timestamp`	integers, strings	A timestamp. Strings constant are allow to input timestamps as dates, see [Working with timestamps] below for more information.
`timeuuid`	uuids	Type 1 UUID. This is generally used as a ``conflict-free'' timestamp. Also see the [functions on Timeuuid]
`tinyint`	integers	8-bit signed int

type	constants supported	description
`uuid`	uuids	Type 1 or type 4 UUID
`varchar`	strings	UTF8 encoded string
`varint`	integers	Arbitrary-precision integer

For more information on how to use the collection types, see the **Working with collections** section below.

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.

Timestamp can be input in CQL as simple long integers, giving the number of milliseconds since the epoch, as defined above.

They can also be input as string literals in any of the following ISO 8601 formats, each representing the time and date Mar 2, 2011, at 04:05:00 AM, GMT.:

- `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— the date will be interpreted as being in the time zone under which the coordinating Cassandra node is configured.

- `2011-02-03 04:05`
- `2011-02-03 04:05:00`
- `2011-02-03 04:05:00.000`
- `2011-02-03T04:05`
- `2011-02-03T04:05:00`
- `2011-02-03T04:05:00.000`

There are clear difficulties inherent in relying on the time zone configuration being as expected, though, so it is recommended that the time zone always be specified for timestamps when feasible.

The time of day may also be omitted, if the date is the only piece that matters:

- `2011-02-03`
- `2011-02-03+0000`

In that case, the time of day will default to 00:00:00, in the specified or default time zone.

Working with dates

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

A date can be input in CQL as an unsigned integer as defined above.

They can also be input as string literals in the following format:

- `2014-01-01`

Working with time

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

A time can be input in CQL as simple long integers, giving the number of nanoseconds since midnight.

They can also be input as string literals in any of the following formats:

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

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: incrementation and decrementation (see `UPDATE` for syntax). Note the value of a counter cannot be set. A counter doesn't exist until first incremented/decremented, and the first incrementation/decrementation is made as if the previous value was 0. Deletion of counter columns is supported but have some limitations (see the Cassandra Wiki for more information).

The use of the counter type is limited in the following way:

- It cannot be used for column that is 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.

Working with collections

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 the messages sent by a given user'', ` events registered by a sensor'', ...), then collections are not appropriate anymore and a specific table (with clustering columns) should be used. Concretely, collections have the following limitations:

- Collections are always read in their entirety (and reading one is not paged internally).
- Collections cannot have more than 65535 elements. More precisely, while it may be possible to insert more than 65535 elements, it is not possible to read more than the 65535 first elements (see CASSANDRA-5428 for details).
- While insertion operations on sets and maps never incur a read-before-write internally, some operations on lists do (see the section on lists below for details). It is thus advised to prefer sets over lists when possible.

Please note that while some of those limitations may or may not be loosen in the future, the general rule that collections are for denormalizing small amount of data is meant to stay.

Maps

A `map` is a [typed] set of key-value pairs, where keys are unique. Furthermore, note that the map are internally sorted by their keys and will thus always be returned in that order. To create a column of type `map`, use the `map` keyword suffixed with comma-separated key and value types, enclosed in angle brackets. For example:

bc(sample).
CREATE TABLE users (
id text PRIMARY KEY,
given text,
surname text,
favs map<text, text> // A map of text keys, and text values
)

Writing `map` data is accomplished with a JSON-inspired syntax. To write a record using `INSERT`, specify the entire map as a JSON-style associative array. *Note: This form will always replace the entire map.*

bc(sample).
INSERT INTO users (id, given, surname, favs)
VALUES (`jsmith', `John', `Smith', \{ `fruit' : `apple', `band' : `Beatles' })

Adding or updating key-values of a (potentially) existing map can be accomplished either by subscripting the map column in an `UPDATE` statement or by adding a new map literal:

bc(sample).
UPDATE users SET favs[`author'] = `Ed Poe' WHERE id = `jsmith'
UPDATE users SET favs = favs + \{ `movie' : `Cassablanca' } WHERE id = `jsmith'

Note that TTLs are allowed for both `INSERT` and `UPDATE`, but in both case the TTL set only apply to the newly inserted/updated *values*. In other words,

bc(sample).
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.

Deleting a map record is done with:

bc(sample).
DELETE favs[`author'] FROM users WHERE id = `jsmith'

Sets

A `set` is a [typed] collection of unique values. Sets are ordered by their values. To create a column of type `set`, use the `set` keyword suffixed with the value type enclosed in angle brackets. For example:

bc(sample).
CREATE TABLE images (
name text PRIMARY KEY,
owner text,
date timestamp,
tags set
);

Writing a `set` is accomplished by comma separating the set values, and enclosing them in curly braces. *Note: An `INSERT` will always replace the entire set.*

bc(sample).
INSERT INTO images (name, owner, date, tags)
VALUES (`cat.jpg', `jsmith', `now', \{ `kitten', `cat', `pet' });

Adding and removing values of a set can be accomplished with an `UPDATE` by adding/removing new set values to an existing `set` column.

bc(sample).
UPDATE images SET tags = tags + \{ `cute', `cuddly' } WHERE name = `cat.jpg';
UPDATE images SET tags = tags - \{ `lame' } WHERE name = `cat.jpg';

As with **maps**, TTLs if used only apply to the newly inserted/updated *values*.

Lists

A `list` is a [typed] collection of non-unique values where elements are ordered by there position in the list. To create a column of type `list`, use the `list` keyword suffixed with the value type enclosed in angle brackets. For example:

bc(sample).
CREATE TABLE plays (
id text PRIMARY KEY,
game text,
players int,
scores list
)

Do note that as explained below, lists have some limitations and performance considerations to take into account, and it is advised to prefer **sets** over lists when this is possible.

Writing `list` data is accomplished with a JSON-style syntax. To write a record using `INSERT`, specify the entire list as a JSON array. *Note: An `INSERT` will always replace the entire list.*

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

Adding (appending or prepending) values to a list can be accomplished by adding a new JSON-style array to an existing `list` column.

bc(sample).
UPDATE plays SET players = 5, scores = scores + [14, 21] WHERE id = `123-afde';
UPDATE plays SET players = 5, scores = [12] + scores WHERE id = `123-afde';

It should be noted that append and prepend are not idempotent operations. This means that if during an append or a prepend the operation timeout, it is not always safe to retry the operation (as this could result in the record appended or prepended twice).

Lists also provides the following operation: setting an element by its position in the list, removing an element by its position in the list and remove all the occurrence of a given value in the list. *However, and contrarily to all the other collection operations, these three operations induce an internal read before the update, and will thus typically have slower performance characteristics*. Those operations have the following syntax:

bc(sample).
UPDATE plays SET scores[1] = 7 WHERE id = `123-afde'; // sets the 2nd element of scores to 7 (raises an error is scores has less than 2 elements)
DELETE scores[1] FROM plays WHERE id = `123-afde'; // deletes the 2nd element of scores (raises an error is scores has less than 2 elements)
UPDATE plays SET scores = scores - [12, 21] WHERE id = `123-afde'; // removes all occurrences of 12

and 21 from scores

As with **maps**, TTLs if used only apply to the newly inserted/updated *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.

Functions

CQL3 distinguishes between built-in functions (so called `native functions') and **user-defined functions**. CQL3 includes several native functions, described below:

Cast

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

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

from	to
`ascii`	`text`, `varchar`
`bigint`	`tinyint`, `smallint`, `int`, `float`, `double`, `decimal`, `varint`, `text`, `varchar`
`boolean`	`text`, `varchar`

from	to
counter	tinyint, smallint, int, bigint, float, double, decimal, varint, text, varchar
date	timestamp
decimal	tinyint, smallint, int, bigint, float, double, varint, text, varchar
double	tinyint, smallint, int, bigint, float, decimal, varint, text, varchar
float	tinyint, smallint, int, bigint, double, decimal, varint, text, varchar
inet	text, varchar
int	tinyint, smallint, bigint, float, double, decimal, varint, text, varchar
smallint	tinyint, int, bigint, float, double, decimal, varint, text, varchar
time	text, varchar
timestamp	date, text, varchar
timeuuid	timestamp, date, text, varchar
tinyint	tinyint, smallint, int, bigint, float, double, decimal, varint, text, varchar
uuid	text, varchar
varint	tinyint, smallint, int, bigint, float, double, decimal, text, varchar

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

bc(sample).
SELECT avg(cast(count as double)) FROM myTable

Token

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

The type of the arguments of the `token` depend on the type of the partition key columns. The return type depend on the partitioner in use:

- For Murmur3Partitioner, the return type is `bigint`.
- For RandomPartitioner, the return type is `varint`.

- For ByteOrderedPartitioner, the return type is `blob`.

For instance, in a cluster using the default Murmur3Partitioner, if a table is defined by

bc(sample).
CREATE TABLE users (
userid text PRIMARY KEY,
username text,
…
)

then the `token` function will take a single argument of type `text` (in that case, the partition key is `userid` (there is no clustering columns so the partition key is the same than the primary key)), and the return type will be `bigint`.

Uuid

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

Timeuuid functions

`now`

The `now` function takes no arguments and generates, on the coordinator node, a new unique timeuuid (at the time where the statement using it is executed). Note that this method is useful for insertion but is largely non-sensical in `WHERE` clauses. For instance, a query of the form

bc(sample).
SELECT * FROM myTable WHERE t = now()

will never return any result by design, since the value returned by `now()` is guaranteed to be unique.

`min_timeuuid` and `max_timeuuid`

The `min_timeuuid` (resp. `max_timeuuid`) function takes a `timestamp` value `t` (which can be [either a timestamp or a date string]) and return a *fake* `timeuuid` corresponding to the *smallest* (resp. *biggest*) possible `timeuuid` having for timestamp `t`. So for instance:

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

will select all rows where the `timeuuid` column `t` is strictly older than `2013-01-01 00:05+0000' but strictly younger than `2013-02-02 10:00+0000'. Please note that `t >= max_timeuuid('2013-01-01 00:05+0000')` would still *not* select a `timeuuid` generated exactly at `2013-01-01 00:05+0000' and is essentially equivalent to `t > max_timeuuid('2013-01-01 00:05+0000')`.

Warning: We called the values generated by `min_timeuuid` and `max_timeuuid` *fake* UUID because they do no respect the Time-Based UUID generation process specified by the RFC 4122. In particular, the value returned by these 2 methods will not be unique. This means you should only use those methods for querying (as in the example above). Inserting the result of those methods is almost certainly *a bad idea*.

Time conversion functions

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

function name	input type	description
`to_date`	`timeuuid`	Converts the `timeuuid` argument into a `date` type
`to_date`	`timestamp`	Converts the `timestamp` argument into a `date` type
`to_timestamp`	`timeuuid`	Converts the `timeuuid` argument into a `timestamp` type
`to_timestamp`	`date`	Converts the `date` argument into a `timestamp` type
`to_unix_timestamp`	`timeuuid`	Converts the `timeuuid` argument into a `bigInt` raw value
`to_unix_timestamp`	`timestamp`	Converts the `timestamp` argument into a `bigInt` raw value
`to_unix_timestamp`	`date`	Converts the `date` argument into a `bigInt` raw value

Blob conversion functions

A number of functions are provided to `convert'' the native types into binary data (`blob`). For every `<native-type> type` supported by CQL3 (a notable exceptions is `blob`, for obvious reasons), the function `type_as_blob` takes a argument of type `type` and return it as a `blob`. Conversely, the function `blob_as_type` takes a 64-bit `blob` argument and convert it to a `bigint` value. And so for instance, `bigint_as_blob(3)` is `0x0000000000000003` and `blob_as_bigint(0x0000000000000003)` is `3`.

Aggregates

Aggregate functions work on a set of rows. They receive values for each row and returns one value for the whole set.
If `normal` columns, `scalar functions`, `UDT` fields, `writetime`, `maxwritetime` or `ttl` are selected together with aggregate functions, the values returned for them will be the ones of the first row matching the query.

CQL3 distinguishes between built-in aggregates (so called `native aggregates') and **user-defined**

aggregates. CQL3 includes several native aggregates, described below:

Count

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

bc(sample).
SELECT COUNT (*) FROM plays;
SELECT COUNT (1) FROM plays;

It also can be used to count the non null value of a given column. Example:

bc(sample).
SELECT COUNT (scores) FROM plays;

Max and Min

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

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

Sum

The `sum` function can be used to sum up all the values returned by a query for a given column.

bc(sample).
SELECT SUM (players) FROM plays;

Avg

The `avg` function can be used to compute the average of all the values returned by a query for a given column.

bc(sample).
SELECT AVG (players) FROM plays;

User-Defined Functions

User-defined functions allow execution of user-provided code in Cassandra. By default, Cassandra supports defining functions in *Java* and *JavaScript*. Support for other JSR 223 compliant scripting languages (such as Python, Ruby, and Scala) has been removed in 3.0.11.

UDFs are part of the Cassandra schema. As such, they are automatically propagated to all nodes in the cluster.

UDFs can be *overloaded* - i.e. multiple UDFs with different argument types but the same function name. Example:

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

User-defined functions are susceptible to all of the normal problems with the chosen programming language. Accordingly, implementations should be safe against null pointer exceptions, illegal arguments, or any other potential source of exceptions. An exception during function execution will result in the entire statement failing.

It is valid to use *complex* types like collections, tuple types and user-defined types as argument and return types. Tuple types and user-defined types are handled by the conversion functions of the DataStax Java Driver. Please see the documentation of the Java Driver for details on handling tuple types and user-defined types.

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

Note that you can use the double-quoted string syntax to enclose the UDF source code. For example:

bc(sample)..
CREATE FUNCTION some_function (arg int)
RETURNS NULL ON NULL INPUT
RETURNS int
LANGUAGE java
AS return arg; ;

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

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

User-defined functions can be used in `SELECT`, `INSERT` and `UPDATE` statements.

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

bc(sample).
CREATE TYPE custom_type (txt text, i int);

CREATE FUNCTION fct_using_udt (somearg int)
RETURNS NULL ON NULL INPUT
RETURNS custom_type
LANGUAGE java
AS + UDTValue udt = udfContext.newReturnUDTValue(); + udt.setString(``txt'', ``some string''); + udt.setInt(``i'', 42); + return udt; + ;

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

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

Java UDFs already have some imports for common interfaces and classes defined. These imports are: Please note, that these convenience imports are not available for script UDFs.

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

See `CREATE FUNCTION` and `DROP FUNCTION`.

User-Defined Aggregates

User-defined aggregates allow creation of custom aggregate functions using **UDFs**. Common examples of aggregate functions are *count*, *min*, and *max*.

Each aggregate requires an *initial state* (`INITCOND`, which defaults to `null`) of type `STYPE`. The first argument of the state function must have type `STYPE`. The remaining arguments of the state function must match the types of the user-defined aggregate arguments. The state function is called once for

each row, and the value returned by the state function becomes the new state. After all rows are processed, the optional `FINALFUNC` is executed with last state value as its argument.

`STYPE` is mandatory in order to be able to distinguish possibly overloaded versions of the state and/or final function (since the overload can appear after creation of the aggregate).

User-defined aggregates can be used in `SELECT` statement.

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

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

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

CREATE OR REPLACE AGGREGATE average ( int )
SFUNC averageState
STYPE tuple<int,bigint>
FINALFUNC averageFinal
INITCOND (0, 0);

CREATE TABLE atable (
pk int PRIMARY KEY,
val int);
INSERT INTO atable (pk, val) VALUES (1,1);
```

INSERT INTO atable (pk, val) VALUES (2,2);
INSERT INTO atable (pk, val) VALUES (3,3);
INSERT INTO atable (pk, val) VALUES (4,4);
SELECT average(val) FROM atable;
p.

See `CREATE AGGREGATE` and `DROP AGGREGATE`.

JSON Support

Cassandra 2.2 introduces JSON support to `SELECT` and `INSERT` 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 new `JSON` keyword can be used to return each row as a single `JSON` encoded map. The remainder of the `SELECT` statment behavior is the same.

The result map keys are the same as 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, this 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\""` (note the 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:

bc(sample).
INSERT INTO mytable JSON `\{\''myKey\": 0, ``value": 0}'

Any columns which are ommitted from the `JSON` map will be defaulted to a `NULL` value (which will result in a tombstone being created).

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 are allow to input timestamps as dates, see [Working with dates] below for more information. Datestamps with format `YYYY-MM-DD HH:MM:SS.SSS` are returned.
`timeuuid`	string	string	Type 1 UUID. See [Constants] 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 [Constants] 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.

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 identifer 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
CAST	no
CLUSTERING	no
COLUMNFAMILY	yes

Keyword	Reserved?
COMPACT	no
CONTAINS	no
COUNT	no
COUNTER	no
CREATE	yes
CUSTOM	no
DATE	no
DECIMAL	no
DEFAULT	yes
DELETE	yes
DESC	yes
DESCRIBE	yes
DISTINCT	no
DOUBLE	no
DROP	yes
DURATION	no
ENTRIES	yes
EXECUTE	yes
EXISTS	no
FILTERING	no
FINALFUNC	no
FLOAT	no
FROM	yes
FROZEN	no
FULL	yes
FUNCTION	no
FUNCTIONS	no
GRANT	yes
GROUP	no
IF	yes
IN	yes

Keyword	Reserved?
INDEX	yes
INET	no
INFINITY	yes
INITCOND	no
INPUT	no
INSERT	yes
INT	no
INTO	yes
IS	yes
JSON	no
KEY	no
KEYS	no
KEYSPACE	yes
KEYSPACES	no
LANGUAGE	no
LIKE	no
LIMIT	yes
LIST	no
LOGIN	no
MAP	no
MASKED	no
MATERIALIZED	yes
MBEAN	yes
MBEANS	yes
MODIFY	yes
NAN	yes
NOLOGIN	no
NORECURSIVE	yes
NOSUPERUSER	no
NOT	yes
NULL	yes

Keyword	Reserved?
OF	yes
ON	yes
OPTIONS	no
OR	yes
ORDER	yes
PARTITION	no
PASSWORD	no
PER	no
PERMISSION	no
PERMISSIONS	no
PRIMARY	yes
RENAME	yes
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

Keyword	Reserved?
TIMEUUID	no
TINYINT	no
TO	yes
TOKEN	yes
TRIGGER	no
TRUNCATE	yes
TTL	no
TUPLE	no
TYPE	no
UNLOGGED	yes
UNMASK	no
UNSET	yes
UPDATE	yes
USE	yes
USER	no
USERS	no
USING	yes
UUID	no
VALUES	no
VARCHAR	no
VARINT	no
VIEW	yes
WHERE	yes
WITH	yes
WRITETIME	no

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

type
byte
complex
date
enum
interval
macaddr

Changes

The following describes the changes in each version of CQL.

3.4.3

- Support for GROUP BY. See <group-by> (see CASSANDRA-10707.

3.4.2

- Support for selecting elements and slices of a collection (CASSANDRA-7396.

3.4.2

- INSERT/UPDATE options for tables having a default_time_to_live specifying a TTL of 0 will remove the TTL from the inserted or updated values
- ALTER TABLE ADD and DROP now allow mutiple columns to be added/removed
- New PER PARTITION LIMIT option (see CASSANDRA-7017.
- **User-defined functions** can now instantiate UDTValue and TupleValue instances via the new UDFContext interface (see CASSANDRA-10818.
- `User-defined types''#createTypeStmt 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. See Cast.

3.4.0

- Support for **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

- Adds new **aggregates**
- User-defined functions are now supported through `CREATE FUNCTION` and `DROP FUNCTION`.
- User-defined aggregates are now supported through `CREATE AGGREGATE` and `DROP AGGREGATE`.
- Allows double-dollar enclosed strings literals as an alternative to single-quote enclosed strings.
- Introduces Roles to supercede user based authentication and access control
- [`Date`] and `Time` data types have been added
- `JSON` support has been added
- `Tinyint` and `Smallint` data types have been added
- Adds new time conversion functions and deprecate `dateOf` and `unixTimestampOf`. See `Time conversion functions`

3.2.0

- User-defined types are now supported through `CREATE TYPE`, `ALTER TYPE`, and `DROP TYPE`
- [`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 were added to hold fixed-length sets of typed positional fields (see the section on [types])
- `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. See [SELECT WHERE] clauses.
- `IF NOT EXISTS` and `IF EXISTS` syntax is now supported by `CREATE USER` and `DROP USER` statmenets, 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 relatiion, see **SELECT WHERE** clauses.
- Added support for `STATIC` columns, see [static in CREATE TABLE].

3.1.4

- `CREATE INDEX` now allows specifying options when creating CUSTOM indexes (see [CREATE INDEX reference]).

3.1.3

- Millisecond precision formats have been added to the timestamp parser (see [working with dates]).

3.1.2

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

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 has been reenabled for CQL3 tables and has new semantics now: the space formerly used by dropped columns will now be eventually reclaimed (post-compaction). You should not readd previously dropped columns unless you use timestamps with microsecond precision (see CASSANDRA-3919 for more details).
- `SELECT` statement now supports aliases in select clause. Aliases in WHERE and ORDER BY clauses are not supported. See the **section on select** for details.
- `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].
- 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**).

3.0.3

- Support for custom [secondary indexes] has been added.

3.0.2

- Type validation for the [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] 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 inputing blobs. Do note that while inputing blobs as strings constant is still supported by this version (to allow smoother transition to blob constant), it is now deprecated (in particular the [data types] section does not list strings constants as valid blobs) 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** 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`. See the [section dedicated to these methods] for more detail.
- `` `Float constants''#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.