Data Manipulation in Cassandra

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

SELECT

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

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

For example:

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

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

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

The `SELECT` statements reads one or more columns for one or more rows in a table. It returns a result-set of the rows matching the request, where each row contains the values for the selection corresponding to the query. Additionally, functions including aggregations can be applied to the result. A `SELECT` statement contains at least a selection clause and the name of the table on which the selection is executed. CQL does **not** execute joins or sub-queries and a select statement only apply to a single table. A select statement can also have a where clause that can further narrow the query results. Additional clauses can order or limit the results. Lastly, queries that require full cluster filtering can append `ALLOW FILTERING` to any query.

Selection clause

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

Selectors

A `selector` can be one of:

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

Aliases

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

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

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

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

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

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

`WRITETIME` and `TTL` function

Selection supports two special functions that aren't allowed anywhere else: `WRITETIME` and `TTL`. Both functions take only one argument, a column name. These functions retrieve meta-information that is stored internally for each column:

- `WRITETIME` stores the timestamp of the value of the column
- `TTL` stores the remaining time to live (in seconds) for the value of the column if it is set to expire; otherwise the value is `null`.

The `WRITETIME` and `TTL` functions can't be used on multi-cell columns such as non-frozen collections or non-frozen user-defined types.

The `WHERE` clause

The `WHERE` clause specifies which rows are queried. It specifies a relationship for `PRIMARY KEY` columns or a column that has a secondary index defined, along with a set value.
Not all relationships are allowed in a query. For instance, only an equality is allowed on a partition key. The `IN` clause is considered an equality for one or more values. The `TOKEN` clause can be used to query for partition key non-equalities. A partition key must be specified before clustering columns in the `WHERE` clause. The relationship for clustering columns must specify a **contiguous** set of rows to order.
For instance, given:

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

The following query is allowed:

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

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

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

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

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

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

For example:

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

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

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

For example:

```
SELECT * FROM posts
 WHERE userid = 'john doe'
```

```
   AND (blog_title, posted_at) > ('John''s Blog', '2012-01-01');
```

This query will return all rows that sort after the one having “John’s Blog” as `blog_tile` and '2012-01-01' for `posted_at` in the clustering order. In particular, rows having a `post_at <= '2012-01-01'` will be returned, as long as their `blog_title > 'John''s Blog'`.
That would not be the case for this example:

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

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

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

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

Grouping results

The `GROUP BY` option can condense all selected rows that share the same values for a set of columns into a single row.
Using the `GROUP BY` option, rows can be grouped at the partition key or clustering column level. Consequently, the `GROUP BY` option only accepts primary key columns in defined order as arguments. If a primary key column is restricted by an equality restriction, it is not included in the `GROUP BY` clause.
Aggregate functions will produce a separate value for each group. If no `GROUP BY` clause is specified, aggregates functions will produce a single value for all the rows.
If a column is selected without an aggregate function, in a statement with a `GROUP BY`, the first value encounter in each group will be returned.

Ordering results

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

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

Limiting results

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

Allowing filtering

By default, CQL only allows select queries that don't involve a full scan of all partitions. If all partitions are scanned, then returning the results may experience a significant latency proportional to the amount of data in the table. The `ALLOW FILTERING` option explicitly executes a full scan. Thus, the performance of the query can be unpredictable.
For example, consider the following table of user profiles with birth year and country of residence. The birth year has a secondary index defined.

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

CREATE INDEX ON users(birth_year);
```

The following queries are valid:

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

// All users with a particular birth year are returned
```

```
SELECT * FROM users WHERE birth_year = 1981;
```

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

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

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

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

INSERT

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

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

For example:

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

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

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

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

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

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

UPDATE

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

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

For instance:

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

UPDATE UserActions
   SET total = total + 2
   WHERE user = B70DE1D0-9908-4AE3-BE34-5573E5B09F14
     AND action = 'click';
```

The `UPDATE` statement writes one or more columns for a given row in a table. The `WHERE`clause is used to select the row to update and must include all columns of the `PRIMARY KEY`. Non-primary key columns are set using the `SET` keyword. In an `UPDATE` statement, all updates within the same partition key are applied atomically and in isolation.
Unlike in SQL, `UPDATE` does not check the prior existence of the row by default. The row is created if none existed before, and updated otherwise. Furthermore, there is no means of knowing which action occurred.
The `IF` condition can be used to choose whether the row is updated or not if a particular condition is met. However, like the `IF NOT EXISTS` condition, a non-negligible performance cost can be incurred.
Regarding the `SET` assignment:

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

Update parameters

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

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

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

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

DELETE

Deleting rows or parts of rows uses the `DELETE` statement:

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

For example:

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

DELETE phone FROM Users
 WHERE userid IN (C73DE1D3-AF08-40F3-B124-3FF3E5109F22, B70DE1D0-9908-4AE3-BE34-557
3E5B09F14);
```

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

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

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

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

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

BATCH

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

```
batch_statement ::=     BEGIN [ UNLOGGED | COUNTER ] BATCH
                        [ USING update_parameter( AND update_parameter)* ]
                        modification_statement ( ';' modification_statement )*
                        APPLY BATCH
modification_statement ::= insert_statement | update_statement | delete_statement
```

For instance:

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

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

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

Note that:

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

force a particular operation ordering, you must specify per-operation timestamps.

- A LOGGED batch to a single partition will be converted to an UNLOGGED batch as an optimization.

`UNLOGGED` batches

By default, Cassandra uses a batch log to ensure all operations in a batch eventually complete or none will (note however that operations are only isolated within a single partition).
There is a performance penalty for batch atomicity when a batch spans multiple partitions. If you do not want to incur this penalty, you can tell Cassandra to skip the batchlog with the `UNLOGGED` option. If the `UNLOGGED` option is used, a failed batch might leave the patch only partly applied.

`COUNTER` batches

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