Other CQL commands

CQL has some additional commands and constructs that provide additional functionality. A few of them bear a resemblance to their similarly-named SQL counterparts, but may behave differently.

COUNT

CQL allows you to return a count of the number of rows in the result set. Its syntax is quite similar to that of the `COUNT` aggregate function in SQL. This query will return the number of rows in the customer table with the last name `Washburne`:

```
SELECT COUNT(*) FROM fenago_ch3.customer WHERE last_name='Washburne';

 count
-------
     2
(1 rows)
```

The most common usage of this function in SQL was to count the number of rows in a table with an unbound query. Apache Cassandra allows you to attempt this, but it does warn you:

```
SELECT COUNT(*) FROM fenago_ch3.customer;

 count
-------
     3
(1 rows)
Warnings :
Aggregation query used without partition key
```

This warning is Cassandra's way of informing you that the query was not very efficient. As described earlier, unbound queries (queries without `WHERE` clauses) must communicate with every node in the cluster. `COUNT` queries are no different, and so these queries should be avoided.

Note

cqlsh has a hard limit of 10,000 rows per query, so `COUNT` queries run from cqlsh on large tables will max out at that number.

DISTINCT

CQL has a construct that intrinsically removes duplicate partition key entries from a result set, using the `DISTINCT` keyword. It works in much the same way as its SQL counterpart:

```
SELECT DISTINCT last_name FROM customer;

 last_name
-----------
       Tam
 Washburne
(2 rows)
```

The main difference of `DISTINCT` in CQL is that it only operates on partition keys and static columns.

Note

The only time in which `DISTINCT` is useful is when running an unbound query. This can appear to run efficiently in small numbers (fewer than 100). Do remember that it still has to reach out to all the nodes in the cluster.

LIMIT

CQL allows the use of the `LIMIT` construct, which enforces a maximum number of rows for the query to return. This is done by adding the `LIMIT` keyword on the end of a query, followed by an integer representing the number of rows to be returned:

```
SELECT * FROM security_logs_by_location LIMIT 1;

 location_id | day      | time_in | employee_id | mailstop
-------------+----------+---------------------+-------------+----------
       MPLS2 | 20180723 | 2018-07-23 11:49:11 |        samb | M266
(1 rows)
```

STATIC

Static columns are data that is more dependent on the partition keys than on the clustering keys. Specifying a column as `STATIC` ensures that its values are only stored once (with its partition keys) and not needlessly repeated in storage with the row data.

A new table can be created with a `STATIC` column like this:

```
CREATE TABLE fenago_ch3.fighter_jets (
 type TEXT PRIMARY KEY,
 nickname TEXT STATIC,
 serial_number BIGINT);
```

Likewise, an existing table can be altered to contain a `STATIC` column:

```
ALTER TABLE fenago_ch3.users_by_dept ADD department_head TEXT STATIC;
```

Now, we can update data in that column:

```
 INSERT INTO fenago_ch3.users_by_dept (department,department_head) VALUES
('Engineering','Richard');
INSERT INTO fenago_ch3.users_by_dept (department,department_head) VALUES ('Marketing','Erlich');
INSERT INTO fenago_ch3.users_by_dept (department,department_head) VALUES ('Finance/HR','Jared');

SELECT department,username,department_head,title FROM fenago_ch3.users_by_dept ;

 department  | username | department_head | title
-------------+----------+-----------------+---------------
 Engineering |   Dinesh |         Richard |      Dev Lead
 Engineering | Gilfoyle |         Richard | Sys Admin/DBA
 Engineering |  Richard |         Richard |           CEO
   Marketing |   Erlich |          Erlich |           CMO
  Finance/HR |    Jared |           Jared |           COO
(5 rows)
```

As shown, `department_head` only changes as per `department`. This is because `department_head` is now stored with the partition key.

User-defined functions

As of version 3.0, Apache Cassandra allows users to create **user-defined functions** (**UDFs**). As CQL does not supply much in the way of extra tools and string utilities found in SQL, some of that function can be recreated with a UDF. Let's say that we want to query the current year from a `date` column. The `date` column will return the complete year, month, and day:

```
SELECT todate(now()) FROM system.local;

 system.todate(system.now())
----------------------------
                 2018-08-03
(1 rows)
```

To just get the year back, we could handle that in the application code, or, after enabling user-defined functions in the `cassandra.yaml` file, we could write a small UDF using the Java language:

```
CREATE OR REPLACE FUNCTION year (input DATE)
 RETURNS NULL ON NULL INPUT RETURNS TEXT
 LANGUAGE java AS 'return input.toString().substring(0,4);';
```

Now, re-running the preceding query with `todate(now())` nested inside my new `year()` UDF returns this result:

```
SELECT fenago_ch3.year(todate(now())) FROM system.local;

 fenago_ch3.year(system.todate(system.now()))
----------------------------------------------
                                         2018
(1 rows)
```

Note

To prevent injection of malicious code, UDFs are disabled by default. To enable their use, set `enable_user_defined_functions: true` in `cassandra.yaml`. Remember that changes to that file require the node to be restarted before they take effect.

cqlsh commands

It's important to note that the commands described in this section are part of cqlsh only. They are not part of CQL. Attempts to run these commands from within the application code will not succeed.

CONSISTENCY

By default, cqlsh is set to a consistency level of `ONE`. But it also allows you to specify a custom consistency level, depending on what you are trying to do. These different levels can be set with the `CONSISTENCY` command:

```
CONSISTENCY LOCAL_QUORUM;

SELECT last_name,first_name FROM customer ;

 last_name | first_name
-----------+------------
       Tam |      Simon
 Washburne |      Hoban
 Washburne |       Zoey

(3 rows)
```

On a related note, queries at `CONSISTENCY ALL` force a read repair to occur. If you find yourself troubleshooting a consistency issue on a small to mid-sized table (fewer than 20,000 rows), you can quickly repair it by setting the consistency level to `ALL` and querying the affected rows.

Note

Querying at consistency `ALL` and forcing a read repair comes in handy when facing replication errors on something such as the `system_auth` tables.

```
CONSISTENCY ALL;
Consistency level set to ALL.

SELECT COUNT(*) FROM system_auth.roles;

 count
-------
    4
(1 rows)

Warnings :
Aggregation query used without partition key
```

COPY

The `COPY` command delivered with cqlsh is a powerful tool that allows you to quickly export and import data. Let's assume that I wanted to duplicate my `customer` table data into another query table. I'll start by creating the new table:

```
CREATE TABLE customer_by_company ( last_name text,
  first_name text,
  addresses list<frozen<customer_address>>,
  company text,
  PRIMARY KEY (company,last_name,first_name));
```

Next, I will export the contents of my `customer` table using the `COPY TO` command:

```
COPY customer (company,last_name,first_name,addresses) TO '/home/aploetz/Documents/Fenago/customer.txt' WITH DELIMITER= '|' AND HEAD
Reading options from the command line: {'header': 'true', 'delimiter': '|'}
Using 3 child processes

Starting copy of fenago_ch3.customer with columns [company, last_name, first_name, addresses].
Processed: 3 rows; Rate: 28 rows/s; Avg. rate: 7 rows/s
3 rows exported to 1 files in 0.410 seconds.
```

And finally, I will import that file into a new table using the `COPY FROM` command:

```
COPY customer_by_company (company,last_name,first_name,addresses) FROM '/home/aploetz/Documents/Fenago/customer.txt' WITH HEADER=tru
Reading options from the command line: {'header': 'true', 'delimiter': '|'}
Using 3 child processes

Starting copy of fenago_ch3.customer_by_company with columns [company, last_name, first_name, addresses].
Processed: 3 rows; Rate: 5 rows/s; Avg. rate: 7 rows/s
3 rows imported from 1 files in 0.413 seconds (0 skipped).
```

Sometimes exporting larger tables will require additional options to be set, in order to eliminate timeouts and errors. For instance, the `COPY TO` options of `PAGESIZE` and `PAGETIMEOUT` control how many rows are read for export at once and how long each read has before it times out. These options can help to effectively break up the export operation into smaller chunks, which may be necessary based on the size and topology of the cluster.

Note

`COPY` has a bit of a bad reputation, as early versions were subject to timeouts when exporting large tables. Remember that the `COPY` tool itself is also subject to the constraints applied by the Apache Cassandra storage model. This means that exporting a large table is an expensive operation. That being said, I have managed to leverage the `PAGESIZE` and `PAGETIMEOUT` options to successfully export 350,000,000 rows from a 40-node cluster without timeouts or errors.

DESCRIBE

`DESCRIBE` is a command that can be used to show the definition(s) for a particular object. Its command structure looks like this:

```
DESC[RIBE] (KEYSPACE|TABLE|TYPE|INDEX) <object_name>;
```

In putting it to use, you can quickly see that it can be used to view things such as full table options, keyspace replication, and index definitions.

Note

The `DESCRIBE` command can be shortened to `DESC`.

Here, we will demonstrate using the `DESC` command on a table:

```
DESC TYPE customer_address;

CREATE TYPE fenago_ch3.customer_address (
 type text,
 street text,
 city text,
 state text,
 postal_code text,
 country text,
 street2 text
);
```

Likewise, the `DESC` command can be used to describe an `INDEX`:

```
DESC INDEX order_status_idx;

CREATE INDEX order_status_idx ON fenago_ch3.order_status_by_week (status);
```