Operaciones CRUD en CQL

Antes de empezar a ver las operaciones CRUD en CQL hemos de crear un *keyspace* y una tabla de ejemplo.

Creación de un *keyspace* y una tabla de ejemplo

Para crear el *keyspace* ejecutaremos el siguiente comando:

```
CREATE KEYSPACE sbd
  WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : 2 };
```

a continuación creamos la tabla `miembros`:

```
CREATE TABLE IF NOT EXISTS sbd.miembros (
    id int PRIMARY KEY,
    nombre text,
    apellidos text,
    email text,
    rol text static,
    fecha_alta timestamp,
    fecha_de_nacimiento date
)
WITH comment = 'Tabla con datos de prueba.';
```

Si la *partition key* es compuesta la definiremos de la siguiente forma:

```
CREATE TABLE IF NOT EXISTS sbd.miembros (
    id int,
    nombre text,
    apellidos text,
    email text,
    rol text static,
    fecha_alta timestamp,
    fecha_de_nacimiento date,
    PRIMARY KEY (id, fecha_alta)
) WITH comment = 'Tabla con datos de prueba.',
  AND CLUSTERING ORDER BY (fecha_alta DESC);
```

Como se puede comprobar este código es cada vez más incómodo de escribir en la consola de CQLSH. Lo más cómodo sería escribir un *script* CQL en un fichero de texto plano y ejecutarlo con `cqlsh`.

Para ejecutar un *script* de CQL desde dentro de CQLSH utilizaremos la sentencia `SOURCE`:

```
SOURCE 'path/to/script.cql';
```

Y si lo que necesitamos es ejecutar un *script* de CQL desde la línea de comandos utilizaremos el siguiente comando:

```
cqlsh -f path/to/script.cql
```

Finamente veamos cómo crear un *keyspace* y una tabla de ejemplo utilizando un *script* de CQL.

```
-- Creamos un keyspace
CREATE KEYSPACE sbd
  WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : 2 };

CREATE TABLE IF NOT EXISTS sbd.miembros (
    id uuid,
    nombre text,
    apellidos text,
    email text,
    rol text static,
    fecha_alta timestamp,
    fecha_de_nacimiento date,
    PRIMARY KEY (id, fecha_alta));

-- Creamos una segunda tabla de asignaturas

CREATE TABLE IF NOT EXISTS sbd.asignaturas (
    id uuid PRIMARY KEY,
    nombre text,
    curso int,
    profesor map <text, text>,
    alumnos frozen<list <map<text, text>>>,
    fecha_inicio timestamp,
    fecha_fin timestamp
);

CREATE TABLE IF NOT EXISTS sbd.ciclos (
    id uuid  PRIMARY KEY,
    nombre text,
    horas int,
    modulos frozen<list<text>>
);

-- Comprobamos que se han creado el keyspace y las tablas

DESCRIBE KEYSPACES;

USE sbd;

DESCRIBE TABLES;
```

Guardamos el código anterior en un fichero llamado `script01.cql` y lo copiamos a un volumen al que tenga acceso el contenedor con Cassandra donde vamos a ejecutar `cqlsh`. En la sección donde se explica cómo crear el *cluster* se usó un volumen común a todos los contenedores que se llamón `scripts`. Si guardamos el fichero `.cql` en este directorio podremos escribir el comando:

```
docker exec -it cass1 cqlsh -f /scripts/script01.cql
```

y nuestro código CQL se ejecutará.

Notas:

- Hay que prestar especial atención a las **comas y los puntos y comas**. Si nos olvidamos de poner una coma o un punto y coma en el lugar adecuado obtendremos un error de sintaxis.
- Los códigos de error indican la línea y la columna donde se ha producido el error **con respecto al inicio de la sentencia CQL** que lo produjo. No del script.

Operaciones de escritura

Para escribir datos en una tabla se usará la sentencia `INSERT`. La sintaxis de la sentencia `INSERT` es la siguiente:

```
INSERT INTO <keyspace>.<table_name> (<column_name>, <column_name>, ...)
VALUES (<value>, <value>, ...) | JSON <json_value>
IF NOT EXISTS
USING <option> <value>;
```

No es necesario indicar todos los campos de la tabla. Pero sí se han de indicar todos los campos de la *primary key*.

Si el registro ya existe el registro se sobreescribirá.

Las cláusulas `IF NOT EXISTS` y `USING` son opcionales.

`IF NOT EXISTS`

La cláusula `IF NOT EXISTS` sirve para indicar que no se ha de insertar el registro si ya existe un registro con la misma *primary key*.

El uso de `IF NOT EXISTS` tiene un coste de rendimiento.

`USING`

La cláusula `USING` es opcional y se utiliza para indicar opciones de escritura. Las opciones posibles son dos:

- `TTL`: Tiempo de vida del registro. Una vez transcurrido este tiempo el registro se borrará automáticamente (se marcará para borrado).
- `TIMESTAMP`: *Timestamp* del registro. Si no se especifica se utilizará el *timestamp* actual. Esta opción no es compatible con `IF NOT EXISTS`.

```
INSERT INTO sbd.miembros (id, nombre, apellidos, email, rol, fecha_alta,
fecha_de_nacimiento)
VALUES (UUID(), 'Juan', 'Pérez', 'juan@gmail.com', 'alumno', NOW(), '1990-01-
01');
```

Esta sentencia es idéntica a la sentencia `INSERT` de SQL.

JSON

La cláusula `JSON` sirve para indicar que los valores se van a insertar utilizando JSON. Por ejemplo:

```
INSERT INTO sbd.miembros JSON '{"id": 123, "nombre": "Juan", "apellidos":
"Pérez", "email": "juan@gmail.com", "rol": "alumno", "fecha_alta": "2020-01-01",
"fecha_de_nacimiento": "1990-01-01"}';
```

`UUID()` y `NOW()`

La función `UUID()` es muy importante ya que genera un identificador único. Es importante ya que, al encontrarnos en un entorno distribuido, si no utilizamos un identificador único podríamos tener problemas de colisiones. Podría suceder que se intentasen realizar dos operaciones de inserción desde dos nodos diferentes con el mismo identificador (colisión) y esto podría provocar que se perdiesen datos. El uso de `UUID()` previene este problema.

Por su parte la función `NOW()` sirve para generar *timestamps*. El valor que devuelve `NOW()` el del tipo `timeuuid`. Un `timeuuid` es un identificador **único** que contiene un *timestamp*. El *timestamp* se puede obtener a partir del `timeuuid` utilizando la función `dateOf()`. Los valores generados por `NOW()`, al giual que los generados por `UUID()`, **son únicos**.

Operaciones de lectura

La sentencia `SELECT` de CQL es muy similar a la sentencia `SELECT` de SQL. La diferencia más importante es que en CQL **no se pueden realizar *joins***.

La sintaxis de la sentencia `SELECT` es la siguiente:

```
SELECT <nombre_columna>, <nombre_columna>, ...
FROM <keyspace>.<nombre_tabla>
WHERE <nombre_columna>
  <operador> <valor>
  AND <column_name>
  OPERATOR <value>
  ...
GROUP BY <column_name>
ORDER BY <column_name> ASC | DESC
LIMIT <n>
ALLOW FILTERING;
```

`GROUP BY`

Cuando agrupamos por una columna hay que tener en cuenta que ésta **ha de formar parte de la partition key**

`WHERE`

La cláusula `WHERE` es muy importante ya que nos permite filtrar los datos que queremos obtener.

Columnas válidas para `WHERE`

En la cláusula `WHERE` sólo se pueden utilizar campos que:

- Formen parte de la *partition key*
- Formen parte de de la *clustering key* **SIEMPRE que vayan precedidos por TODOS los campos de la *partition key***. Además hay que tener en cuenta dos salvedades:
    - Las comparaciones respecto a los campos de la *partition key* han de ser siempre de igualdad.
    - Las comparaciones respecto a los campos de la *clustering key* pueden ser de igualdad o de desigualdad.

- Formen parte de un índice creado con la sentencia `CREATE INDEX`.

```
CREATE TABLE cycling.cyclist_points (
   id UUID,
   firstname text,
   lastname text,
   race_title text,
   race_points int,
   PRIMARY KEY (id, race_points ));

SELECT sum(race_points)
FROM cycling.cyclist_points
WHERE id=e3b19ec4-774a-4d1c-9e5a-decec1e30aac
      AND race_points > 7;
```

Operadores

Los operadores pueden ser:

- `=`
- `!=`
- `>`
- `<`
- `>=`
- `<=`
- `IN`: Sirve para comparar un valor con una lista de valores separados por comas: `WHERE id IN (1, 2, 3)`. Y se puede usar con la *partition key*.
- `CONTAINS`: Sirve para filtrar por los datos de una colección. Los tipos `collection` son `set`, `list` y `map`.
- `CONTAINS KEY`: Sirve para filtrar por las claves de un mapa.

```
SELECT * FROM sbd.miembros WHERE id = 123 AND fecha_alta < '2020-01-01';
```

Aunque al hacer la comparación estemos expresando la fecha como una cadena de texto, internamente, Cassandra la convertirá a un tipo `timestamp` y realizará la comparación.

En Cassandra **nunca se debe de hacer** `SELECT *`. Siempre se ha de seleccionar utilizando, como mínimo, algún campo de la *partition key*. En el caso de realizar un `SELECT *` se producirá un *full table scan*. Esto es, se leerán todos los datos de la tabla en todos los nodos del *cluster*.

`ALLOW FILTERING`

Si no se especifica la *partition key* en la cláusula `WHERE` se producirá un error. Si queremos realizar una consulta que no incluya la *partition key* hemos de indicar que permitimos filtrar los datos utilizando la cláusula `ALLOW FILTERING`. Esta cláusula es muy peligrosa ya que puede provocar que se produzca un *full table scan*.

Full table scan

Un *full table scan* es una operación que lee todos los datos de una tabla. En Cassandra, al ser un sistema distribuido, esto es muy costoso. Por ello, **nunca se debe de hacer un *full table scan***.

Operaciones de actualización

Para actualizar los valores de los datos usaremos el comando `UPDATE`. La sintaxis de este comando es la siguiente:

```
UPDATE <keyspace>.<table_name>
USING TTL <valor> | USING TIMESTAMP <valor>
SET <column_name> = <value>, <column_name> = <value>, ...
WHERE <column_name> OPERATOR <value>
  AND <column_name> OPERATOR <value>
  ...
IF EXISTS | IF <condición>
  AND <condición>
  ...
```

Como en el caso de la sentencia `INSERT`, las cláusulas `USING`, `IF EXISTS` y `IF` son opcionales. Las cláusulas `IF EXISTS` e `IF` tienen un coste de rendimiento.

```
UPDATE sbd.miembros
SET nombre = 'Juanito'
WHERE id = 12345678-1234-1234-1234-123456789012 AND fecha_alta = '2020-01-01';
```

La cláusula `WHERE` sirve para especificar la fila o filas que van a ser modificadas.

- Para especificar una única fila hemos de indicar cual es el valor de la clave primarios que queremos modificar: `primary_key_name = primary_key_value, ...`. Si ésta está formada por varias columnas: `primary_key_c1_name = primary_key_c1_value AND ...` y se han de suministrar los valores de todos los campos que formen parte de la *primary key*
- Si queremos actualizar varias filas hemos de incluir la cláusula `IN` seguida de una lista de valores separados por comas: `primary_key_name IN (primary_key_value, ...)`. Esto sólo se puede aplicar al último campo de la *partition key*.

Upsert

El comportamiento de `UPDATE` es similar al de `INSERT`. Si la fila que vamos a modificar no existe se creará. Si existe se actualizará.

Operaciones de borrado

Para borrar datos de una tabla usaremos el comando `DELETE` que tiene la siguiente sintaxis:

```
DELETE <column_name>, <column_name>, ...
FROM <keyspace>.<table_name>
USING TIMESTAMP <valor>
WHERE <column_name> OPERATOR <value>
  AND <column_name> OPERATOR <value>
  ...
IF EXISTS | IF <condición>
  AND <condición>
  ...
```

Los borrados serán a nivel de columnas. Es decir, si queremos borrar un conjunto de columnas hemos de indicarlas explícitamente. Si queremos borrar una fila completa no hemos de indicar ninguna columna.

`USING TIMESTAMP`

La cláusula `USING TIMESTAMP` sirve para indicar a partir de que *timestamp* se han de borrar los datos. Es decir, Cassandra marcará para borrado aquellas filas que son anteriores al *timestamp* indicado.

Borrado de varias filas

Para seleccionar más de una fila para borrado se han de seguir los mismos pasos que en el caso de la sentencia `UPDATE`.

```
DELETE FROM sbd.miembros
WHERE id = 12345678-1234-1234-1234-123456789012 AND fecha_alta = '2020-01-01';
```

Si en lugar de indicar `id` y `fecha_alta` indicamos únicamente `id` se borrarán todos los registros con ese `id`, es decir, esa partición.

¿Como se borran los datos en Cassandra?

Los borrados en Cassandra están diseñados de forma que se priorice el rendimiento.

Cassandra trata una operación de borrado com si se tratara de una inserción o un *upsert*. Lo que se añade es una marca de borrado o *tombstone*. Los *tombstones* tienen fecha de expiración de manera que, cuando esta se alcanza se realizará el borrado como parte del proceso de compactación de Cassandra.

Se recomienda hacer el menor número de operaciones de borrado posible y, cuando sea posible, hacer borrados de grandes bloques de datos en lugar de hacerlo registro a registro.