

CQL (*Cassandra Query Language*)

Apache Cassandra - Bases de datos II

Alberto Díaz Álvarez (<alberto.díaz@upm.es>)

Departamento de Sistemas Informáticos

Escuela Técnica superior de Ingeniería de Sistemas Informáticos

License CC BY-NC-SA 4.0

Cassandra Query Language (CQL)

Es el lenguaje principal de comunicación con clústers de Cassandra

- Su sintaxis es muy intuitiva, similar a SQL
- Pero no tiene gramática para características relacionales, como los `join`

```
CREATE KEYSPACE nombre_del_keyspace ...  
DROP KEYSPACE nombre_del_keyspace;  
CREATE TABLE nombre_de_la_tabla ...  
TRUNCATE TABLE nombre_de_la_tabla;  
DROP TABLE nombre_de_la_tabla;  
INSERT INTO nombre_de_la_tabla ...  
UPDATE nombre_de_la_tabla SET ...  
DELETE FROM nombre_de_la_tabla WHERE ...  
// Y más ...
```

Características de CQL

Es case-insensitive; por ejemplo, `SELECT` es lo mismo que `select`

- Lo mismo con los identificadores
- De hecho da igual cómo los creemos, CQL los almacena en minúsculas
- **Excepto** en los identificadores que se enmarcan entre dobles comillas
 - `tabla` es igual que `TABLA`, pero `"tabla"` es diferente de `"TABLA"`

Los comentarios se preceden de `//`, `--` o `/* */`

¿Cómo ejecutamos queries?

Podemos ejecutarlas programáticamente desde un *driver* cliente

- Por ejemplo, desde Python o Java (Datastax Java Driver, el de por defecto)

También desde su *shell*, denominada *cqlsh* (*Cassandra Query Language Shell*)

- Viene de serie con cualquier paquete que instalemos de Cassandra
- Está desarrollado en Python
- Se conecta a un nodo del clúster y ejecuta las queries que le pasemos
 - Si no se especifica, se conecta al nodo local (el de por defecto)

Usar editores propietarios o de terceros

- Por ejemplo, [DataStax Studio](#)

cqlsh

```
$ cqlsh [options] [host [port]]
```

- `options` son opciones que incluyen, entre otras:
 - `--help`: La ayuda, que incluye todas las opciones disponibles
 - `--version`: La versión de `cqlsh`
 - `-u` y `-p`: El usuario y la contraseña
 - `-k`: El `keyspace` a usar
 - `-f`: El fichero de queries a ejecutar
 - `--request-timeout`: El timeout de las queries

Comandos especiales

- **CAPTURE**: Captura la salida de la query en un fichero (no sobrescribe, añade)
- **CONSISTENCY**: Muestra el nivel de consistencia y permite cambiarlo
- **COPY**: Importación y exportación de datos
- **DESCRIBE**: Muestra información sobre el clúster, **keyspace**, tablas, etc.
- **EXIT**: Sale de **cqlsh**
- **PAGING**: Activa o desactiva la paginación en los resultados de las consultas
- **TRACING**: Activa o desactiva las trazas de las consultas

Consistencia "ajustable"

CONSISTENCY permite "ajustar" el nivel de consistencia por operación

- **¿Cuántas réplicas** deben responder para aceptar una operación?
 - ONE, TWO y THREE: Al menos una, dos y tres réplicas, respectivamente
 - QUORUM: La mayoría de las réplicas del clúster
 - ALL: Todas las réplicas
 - LOCAL_*: Limita la respuesta a las réplicas del datacenter local

¿Y si existen inconsistencias entre réplicas? Se resuelven durante las lecturas

- **CUIDADO:** Una mayor consistencia afectará significativamente al rendimiento

Un proceso similar se realiza durante las **operaciones de escritura**

- **CUIDADO:** Si no se puede escribir en todas las réplicas, la operación fallará

Comando **COPY**

Permite importar datos desde ficheros CSV

```
COPY tabla (columna1, columna2, ...) FROM 'fichero.csv' WITH DELIMITER = ',' AND HEADER = TRUE;
```

También permite la exportación de datos a ficheros CSV

```
COPY tabla (columna1, columna2, ...) TO 'fichero.csv' WITH DELIMITER = ',' AND HEADER = TRUE;
```

CUIDADO: No es lo más recomendable para la carga o volcado de datos masivos

Identificadores, constantes y palabras clave

Identificadores

Se usan para nombrar objetos de la base de datos (tablas, columnas, etc.)

Cumplen la expresión regular:

```
[a-zA-Z][a-zA-Z0-9_]*
```

Es decir, tienen que empezar por una letra y, a partir del segundo carácter pueden ser letras, números o el guion bajo

Son case-insensitive, como ya se dijo, salvo que vayan entrecomillados (`quoted identifier`)

Constantes

CQL tiene las siguientes constantes:

- `String`: Secuencia de caracteres entre comillas simples.

```
'Cantando bajo la lluvia'
```

CUIDADO: No confundir con los `quoted identifiers` que usan dobles comillas

- `Integer`, `float` y `boolean`
- `UUID` (Universally unique identifier)
- `Blob`: se escriben en hexadecimal empezando por `0x`
- `NULL` o ausencia de valor

Palabras clave o keywords

CQL distingue dos tipos:

Las **palabras reservadas** no se pueden usar como identificadores, pero se pueden crear identificadores con nombre de palabra reservada con dobles comillas

"AUTHORIZE" "SELECT" "TABLE"

Las **palabras no reservadas** sí se pueden usar como identificadores sin comillas.

Palabras clave de CQL

Términos

Un término es cualquier valor soportado por CQL

- Constante
- Literal
- Llamada a una función nativa o de usuario
- Operación aritmética entre términos
- `Type hint`: Tipo de datos esperado en una consulta CQL (opcional)
- Un `bind maker` usado en las prepared statements, que se representa con el símbolo `?`

Tipos de datos CQL

Tipos de datos

CQL es un lenguaje tipado y soporta diversos tipos de datos:

- Nativos: `INT`, `TEXT`, `VARCHAR`, `DATE`, etc.
- De colección: `map`, `set` y `list`
- Definidos por el usuario
- Tuplas

Los veremos a continuación con más detalle

Tipos nativos

Los **tipos nativos** son los tipos más básicos soportados en CQL.

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


Tipos nativos: Cadenas de caracteres

- ASCII
- INET: Direcciones IPv4 o IPv6
- VARCHAR: Máx 65535 caracteres
- TEXT: Máximo 2GB

Tanto VARCHAR como TEXT almacenan cadenas de longitud variable, pero el primero se usa para cadenas más pequeñas

Tipos nativos: Numéricos

- TINYINT, SMALLINT, INT, BIGINT: Enteros de diverso tamaño
- VARINT
- FLOAT, DECIMAL
- COUNTER

Tipos nativos: Booleanos

- BOOLEAN: Puede ser true o false

Tipos nativos: Fechas

- **DATE**: Fecha en formato 'YYYY-MM-DD'
- **TIME**: Horas, minutos, segundos y milisegundos
- **TIMESTAMP**: Milisegundos desde 01/01/1970
- **DURATION**

CUIDADO: No existe el tipo **DATETIME** como en SQL.

Duration

Se utiliza para almacenar una cantidad de tiempo, como la duración de un evento, un concierto, etc.

Formato `PTnHnMnS` donde

- `P` indica que se trata de un período de tiempo
- `T` inicio de la parte del tiempo (hora, minuto, segundo)
- `H` indica las horas
- `M` indica los minutos
- `S` indica los segundos.

`PT1H30M` representa una duración de 1h y 30 minutos

`PT30S` representa una duración de treinta segundos

Otros tipos nativos

- `blob` Bytes
- `timeuuid` Versión 1 de UUID
- `uuid`: Identificador único universal. Cualquier versión.

Habitualmente los `uuid` se usan como clave primaria, valiéndose de la función nativa `uuid()` para generar valores únicos

Definición de datos (DDL)

Keyspaces y tablas

Un Keyspace es la unidad de organización lógica de más alto nivel

- Es análogo a una base de datos en sistemas relacionales
- Se definen opciones para todas sus tablas

Los datos están organizados en tablas cuya definición indica los datos que van a albergar y sus tipos

- La creación es muy similar a un modelo relacional

Creación de Keyspaces

```
CREATE KEYSPACE [ IF NOT EXISTS ] keyspace_name  
    WITH options
```

Options:

- **replication**: Estrategia de replicación. Parámetro obligatorio.
- **durable_writes**: Si se establece a `true` se guardarán las escrituras en el registro de transacciones. Por defecto a `true`. Parámetro opcional.

Creación de **Keyspaces** : Replication

Este campo especifica cómo se replicarán los datos entre los diferentes nodos de un clúster de Cassandra.

Existen varias estrategias:

- **SimpleStrategy**: Para clústeres con un único centro de datos. No se recomienda para producción.
- **NetworkTopologyStrategy**: Más avanzada, donde se tienen más centros de datos en cada nodo.

Creación de Keyspaces : Replication (2)

En ambas estrategias existe el campo `replication_factor` que indica cuántas copias se deben hacer en cada clúster. Se comporta de diferente manera en ambas estrategias:

- SimpleStrategy, cuántas réplicas de cada fragmento de datos se deben crear en todo el clúster.

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

Cada réplica se almacenará en tres nodos diferentes.

Creación de **Keyspaces** : Replication (3)

- NetworkTopologyStrategy

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

Si tuviera dos centros de datos: data1 y data2, se crearían 3 réplicas en cada uno.

```
CREATE KEYSPACE ventas  
  WITH replication = {'class': 'NetworkTopologyStrategy', 'replication_factor' : 3, 'data2': 2};
```

Si se ejecuta **DESCRIBE KEYSPACE ventas;** se puede observar que data1 tendrá un valor de 3 y data2 un valor de 2

CUIDADO: Si se intenta crear un factor de replicación mayor al número de nodos, dará un aviso

USE `keyspace`

Tras haber creado un `keyspace` se puede usar la sentencia `USE` para cambiar el espacio de trabajo.

```
USE ventas;
```

Esto hará que se pueda trabajar con los objetos pertenecientes a ese `keyspace`:
tablas, funciones, etc.

Es similar al comando `USE` en sistemas relacionales

ALTER keyspace

Se pueden modificar las opciones de un keyspace ya existente

```
ALTER KEYSPACE IF EXISTS ventas  
  WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 5};
```

IF EXISTS evita que se devuelva un error si no existe el keyspace

OJO: No se puede cambiar el nombre de un keyspace con esta sentencia

DROP keyspace

Por supuesto podremos borrar un `keyspace` que no queramos utilizar más

```
DROP KEYSPACE IF EXISTS ventas;
```

CUIDADO: Esto hará que se borren también todas las tablas, funciones y datos introducidos de manera irreversible.

CREATE table

Para poder guardar datos en un **keyspace** es necesario una nueva estructura llamada **tabla**.

Su creación es muy similar a como lo haríamos con SQL en un sistema relacional

```
CREATE TABLE [ IF NOT EXISTS ] nombre_tabla (  
    column_definition (, column_definition )*  
    [ PRIMARY KEY (primary_key) ]  
)  
[ WITH table_options ]
```

CREATE table

Un ejemplo de una tabla de personajes para un MMORPG

```
CREATE TABLE personajes (  
  uuid UUID PRIMARY KEY,  
  nombre text,  
  nivel int,  
  creado timestamp,  
  tiempo_jugado duration,  
  clase text,  
  raza text,  
  ultima_conexion timestamp,  
  logros set<text>,  
  equipo list<text>  
);
```


CREATE table

En el ejemplo anterior hemos visto algunos tipos de datos simples ya conocidos y algunos como `set` y `list` que se verán más adelante.

- `PRIMARY KEY`. Identifica una fila. Se puede poner en la misma columna o después de los campos
- Si se pone `STATIC` en una columna, todas las filas de una partición tendrán el mismo valor para ese campo.
 - Una columna `STATIC` no puede ser `PRIMARY KEY`

```
CREATE TABLE clientes (  
    nombre TEXT,  
    apellidos TEXT,  
    email TEXT STATIC,  
    PRIMARY KEY (nombre, apellidos)  
);
```

Partition key y clustering key

Una clave primaria tiene dos partes

- **Partition key** que es obligatoria. Determina el nodo que almacenará la fila.
- **Clustering key** que es opcional y determina la ordenación de los datos en la partición.

```
PRIMARY KEY(partition_key, clustering_key, ck2, ck3)
```

La `partition key` puede tener varias columnas. Lo vemos a continuación

Particiones en tablas y **STATIC**

```
CREATE TABLE particiones (  
    k1 INT,  
    k2 INT,  
    c1 INT,  
    s1 INT STATIC,  
    PRIMARY KEY ((k1, k2), c1)  
);
```

Si insertamos los siguientes valores

```
INSERT INTO particiones (k1, k2, c1, s1) VALUES (1, 2, 3, 4);  
INSERT INTO particiones (k1, k2, c1, s1) VALUES (1, 2, 7, 99);  
INSERT INTO particiones (k1, k2, c1, s1) VALUES (3, 4, 5, 8);
```

Particiones en tablas y **STATIC**

```
SELECT * FROM particiones;
```

k1	k2	c1	s1
3	4	5	8
1	2	3	99
1	2	7	99

Se puede observar como (k1, k2) forman la **partition key** y, por lo tanto, cada valor distinto de esa tupla estará en una partición distinta.

El valor de **s1** se comparte cuando (k1, k2) son iguales.

Opciones de tabla

TBD

ALTER table

Se puede modificar una tabla con el comando ALTER

- Añadir una nueva columna con ADD. No puede formar parte de la clave primaria
- Borrar una columna con DROP, se borrará también todo su contenido
- RENAME permite cambiar el nombre de una clave primaria

Además se puede utilizar WITH al final del comando, igual que en la creación

ALTER table

Si partimos de la tabla `clientes` que creamos anteriormente, podríamos modificarla con las siguientes sentencias:

```
ALTER TABLE clientes ADD telefono TEXT;
```

```
ALTER TABLE clientes DROP email;
```

```
ALTER TABLE clientes RENAME nombre TO name;
```

A partir de Cassandra 3.0 no se permite modificar tipos de datos de columnas

DROP table

```
DROP TABLE [ IF EXISTS ] nombre;
```

Permite borrar una tabla existente y todo su contenido.

Si la tabla no existe dará error, salvo si se pone **IF EXISTS**

TRUNCATE table

```
TRUNCATE [ TABLE ] nombre;
```

El comando **TRUNCATE** vacía una tabla sin borrarla. Es decir, sólo elimina el contenido.

- **TABLE** se puede omitir.

Manipulación de datos (DML)

DML

Todas las operaciones que se pueden hacer dentro del Lenguaje de Manipulación de Datos o DML son similares a las que se pueden hacer en SQL.

- `SELECT` para consultar datos
- `INSERT` para insertar datos
- `UPDATE` para modificaciones
- `DELETE` para borrados

SELECT

Las consultas o queries se hacen con la sentencia **SELECT**

```
SELECT nombre, apellidos  
FROM clientes  
WHERE nombre = 'Pepa';
```

- En CQL no existen subconsultas ni el uso de **JOIN**
- Solo se pueden hacer consultas a una tabla

SELECT

CQL puede devolver los resultados como JSON, en lugar de como una tabla.

```
SELECT JSON nombre, apellidos FROM clientes;
```

Devolvería algo similar a

```
[json]
```

```
-----  
{ "nombre": "Luisa", "apellidos": "Martínez" }  
{ "nombre": "Pepa", "apellidos": "Gómez" }
```

Claúsula **WHERE**

Filtra la filas por las que buscar

- Sólo se permiten columnas que forman parte de la clave primaria o índice secundario
- Si quisiéramos usar otras columnas se puede usar **ALLOW FILTERING** pero es muy ineficiente
- No es tan potente como en una base de datos relacional

ALLOW FILTERING

```
CREATE TABLE personas (  
  uuid UUID,  
  nombre TEXT,  
  PRIMARY KEY(uuid)  
);
```

Si intentamos filtrar por el campo nombre, nos daría el siguiente error:

```
Cannot execute this query as it might involve data filtering may have  
unpredictable performance.  
If you want to execute this query despite the performance unpredictability,  
use ALLOW FILTERING
```

Bastaría con ponerlo al final de la sentencia

```
SELECT * FROM personas WHERE nombre = 'Ana' ALLOW FILTERING;
```

Índice secundario

Para poder filtrar por columnas que no forman parte de la clave con `WHERE` se deben crear índices secundarios.

- Esta manera es más eficiente que con `ALLOW FILTERING`

```
CREATE INDEX nombre_indice ON Tabla (columna);
```


Agrupando resultados

Con el comando `GROUP BY` se pueden agrupar filas cuyas columnas tengan los mismos valores

- Sólo se puede usar con columnas que se hayan definido como `PRIMARY KEY` (partition keys o clustering)

```
SELECT ciudad, count(*)  
FROM clientes  
GROUP BY nombre;
```

Ordenando y limitando resultados

`ORDER BY` permite ordenar el resultado por una o varias columnas

- Esas columnas deben pertenecer a la `clustering key`

`LIMIT` limita los resultado a un número de filas especificado

```
SELECT * FROM clientes LIMIT 10;
```

INSERT

Sirve para insertar filas en una tabla.

UPDATE

Actualiza una fila

DELETE

Borra filas o partes de filas

BATCH

Funciones

Funciones definidas por el usuario

Colecciones

Tipos de datos definidos por el usuario

Comandos shell

Seguridad y roles

Gracias