Cassandra

Introducción a Cassandra

Apache Cassandra es un sistema de gestión de bases de datos NoSQL distribuido, altamente escalable y de alto rendimiento. Fue diseñado para gestionar grandes cantidades de datos en múltiples servidores, proporcionando alta disponibilidad (C**A**P) sin un único punto de fallo (CA**P**).

Cassandra es un proyecto de **código abierto** desarrollado por Apache Software Foundation y escrito en Java. Es una base de datos gratuita con un modelo de negocio basado en servicios y soporte.

Se trata de una base de datos NoSQL **columnar** lo que significa que los datos se almacenan en columnas en lugar de filas. Esto se traduce en un acceso más eficiente a los datos cuando lo que se consulta es un subconjunto de las columnas de una tabla. Estos tipos de base de datos son especialmente apropiados para consultas analíticas.

Esta base de datos, como suele suceder con las bases de datos NoSQL, no garantiza los principios ACID (Atomicity Consistency Isolation Durability. En su lugar garantiza los principios BASE:

- ***Basic Availability***: la base de datos siempre está disponible.
- ***Soft state***: los datos pueden cambiar con el tiempo, incluso sin una entrada de datos.
- ***Eventual consistency***: la base de datos llegará a un estado consistente en algún momento.

¿Cómo usaremos Cassandra?

La forma más sencilla de usar Cassandra es mediante el uso de contenedores Docker. Para ello podemos usar el siguiente comando:

```
docker run --name cassandra -p 9042:9042 -d cassandra:latest
```

Este comando lo usaremos la primera vez que queramos usar Cassandra. En las siguientes ocasiones podremos usar el siguiente comando:

```
docker start cassandra
```

y para parar el contenedor:

```
docker stop cassandra
```

Una vez que el contenedor esté en ejecución podremos conectarnos a él usando el siguiente comando:

```
docker exec -it cassandra cqlsh
```

de esta forma obtendremos una consola de Cassandra ejecutado una *shell* de CQL (cqlsh) desde la que podremos ejecutar comandos CQL.

¿Qué es CQL?

Cassandra Query Language (CQL) es un lenguaje de consulta similar a SQL que nos permitirá interactuar con Cassandra. Cassandra dispone de *drivers* para Java (JDBC), Python (DBAPI2), Node.JS (Datastax), Go (gocql) y C++.

¿Qué es una base de datos columnar?

En una base de datos columnar los datos se almacenan en columnas en lugar de filas. Esto permite que las consultas sean más rápidas.

Estas bases de datos están concebidas para recuperar datos en forma de columnas. Esto es útil cuando se desea recuperar un subconjunto de columnas de una tabla que contiene un gran número de columnas. Estas consultas son típicas en aplicaciones analíticas.

Todos los valores de la misma columna se encuentran juntos en el disco. Esto permite que las consultas, cuando sólo nos interesa un subconjunto de columnas, sean más rápidas.

Características de Cassandra

Cassandra tiene las siguientes características que la diferencian de otras bases de datos:

- Base de datos distribuida.
- Tolerante a fallos.
- Escalable linealmente y de forma horizontal.
- No sigue el patrón **maestro-esclavo**.
- **Permite especificar el nivel de consistencia de las operaciones**.

En primer lugar Cassandra es una **base de datos distribuida**. Esto quiere decir que los datos se almacenan en múltiples nodos de un cluster. Esto permite que **los datos estén replicados** y que la base de datos sea **tolerante a fallos**.

Cassandra es **escalable linealmente**. Esto quiere decir que si añadimos más nodos a la base de datos el rendimiento de la misma aumentará de forma lineal. De esta forma podemos escalar la base de datos de forma horizontal.

No sigue el patrón maestro-esclavo. Todos los nodos son iguales y no hay un nodo maestro, es decir, es una base de datos **peer-to-peer**.

Desventajas de Cassandra Para escalar Cassandra de forma horizontal es necesario añadir nuevos nodos a la base de datos. Esto puede ser complicado en algunos casos.

Para maximizar el rendimiento tendremos que conocer cuales serán las consultas que se realizarán a la base de datos con antelación a su creación. Esto puede ser complicado en algunos casos.

Anillo de Cassandra

Figure 1: Anillo de Cassandra

Arquitectura de Cassandra

A diferencia de un *cluster* maestro-esclavo, en el que el nodo maestro es el responsable de encargarle a los nodos esclavos la ejecución de las consultas, en Cassandra todos los nodos son iguales y no hay un nodo maestro. En un cluster maestro-esclavo el nodo maestro supone un punto único de fallo, mientras que en Cassandra, al no existir nodos maestros, no hay un punto único de fallo. Cuando realizamos una operación en Cassandra habrá un nodo encargado de gestionarla pero ese nodo va cambiando con cada operación.

Un *cluster* de Cassandra se denomina *ring* o anillo. Este anillo está formado por varios nodos interconectados y configurados para propósitos de replicación. Los nodos serán *conscientes* de los otros nodos del anillo y de su estado y se comunicarán entre ellos para replicar los datos cumpliendo las condiciones de consistencia que se hayan establecido.

Cada nodo del anillo tiene la misma importancia que los demás, **no hay un nodo maestro** ya que se trata de un sistema P2P. Del mismo modo, en cada nodo habrá una instancia de Cassandra. Los nodos deberán de encontrarse, idealmente, en ubicaciones diferentes para evitar que un desastre natural pueda afectar a todos los nodos simultáneamente.

Todas estas característica hacen que no exista un **SPOF** (*Single Point Of Failure*), es decir, que no haya un punto único de fallo.

Un anillo de Cassandra también se denomina ***datacenter***.

Nodos virtuales

Cada nodo del anillo se puede dividir a su vez en *nodos virtuales* (vnodes) concepto similar al de varias máquinas virtuales en una única máquina física. Cada nodo virtual se encarga de una parte del anillo del nodo *real*. De esta forma, si añadimos un nuevo nodo al anillo, este se dividirá en nodos virtuales y cada nodo virtual se encargará de una parte del anillo. Es decir, de una parte de los datos que le corresponde gestionar al nuevo nodo del anillo. Esto permite que el anillo se reequilibre de forma automática. Un nodo *real* puede distribuir sus datos entre varios nodos virtuales. Esto permite mejorar la disponibilidad de los datos al aumentar la replicación. Esto es similar a lo que sucede cuando establecemos varios servicios de una máquina de modo que se ejecuten en diferentes máquinas virtuales o contenedores.

Jerarquía de Cassandra

En primer lugar tendremos el *cluster*, que estará formado por uno o varios *anillos* o *datacenters*. Cada *datacenter* estará formado a su vez por uno o más *racks*, que serán la agrupación lógica de varios servidores o nodos. Finalmente, los nodos estarán constituidos por uno o varios nodos virtuales.–

Escritura y lectura en Cassandra

Figure 2: Escritura y lectura en Cassandra

Escritura y lectura en Cassandra

Proceso de escritura en Cassandra

El dato que entra en Cassandra se divide para ir a dos destinos diferentes:

- ***memtable***: es una **tabla en memoria** que se utiliza para almacenar los datos que se van a escribir en disco. Cuando la *memtable* se llena, se vuelca en disco en forma de *SSTable*.
- ***Commit log***: es un **registro de todas las operaciones** de escritura que se han realizado en Cassandra. Se utiliza para **recuperar los datos en caso de** que se produzca un **fallo** en el sistema.

Podríamos decir que la *SSTable* es la base de datos en sí, mientras que la *memtable* y el *commit log* son mecanismos de Cassandra para mejorar el rendimiento y la disponibilidad de los datos.

(La **SS** en *SSTable* significa *Sorted String* y se refiere a que los datos se almacenan en disco de forma ordenada. Las *SSTable* se almacenan en disco en forma de archivos).

Según se van volcando los datos de la *memtable* a la *SSTable* se irán borrando entradas en el *commit log*.

memtable Consiste en una serie de particiones en memoria. Su utilidad es la de ofrecer gran velocidad de escritura y lectura. Funciona como una caché.

Cuando alcanza un tamaño determinado (indicado en la configuración), se vuelca en disco en forma de *SSTable*.

commit log Los *commitlogs* son logs de sólo escritura (*append only*) de todos los cambios locales a un nodo de Cassandra. Cualquier dato escrito en Cassandra se escribirá primero en un *commit log* antes de escribirse en una *memtable*. Esto proporciona seguridad frente a un apagado inesperado. Al iniciarse un nodo, cualquier cambio registrado en el *commit log* se aplicará a las *memtables*.

SSTable Consiste en una serie de ficheros en disco que contienen los datos de las particiones. Estos ficheros serán **inmutables**. Los datos, una vez se escriben en la *SSTable* **no se podrán modificar**. Las operaciones de modificación se realizarán creando nuevos ficheros, con las modificaciones, con un nuevo *timestamp*.

Cada *SSTable* tendrá asociadas las siguientes estructuras de datos (se crean cuando se crea una *SSTable*):

- Data (`Data.db`): contiene los datos de la *SSTable*.
- Primary index (`Index.db`): contiene los índices de las claves de las columnas con punteros a sus posiciones en el fichero de datos.

- Bloom filter (`Filter.db`): estructura almacenada en memoria que sirve para comprobar si una clave existe en la *memtable* antes de acceder a la *SSTable*.
- Compression information (`CompressionInfo.db`): contiene información sobre la compresión de los datos.
- Statistics (`Statistics.db`): contiene información estadística sobre los datos de la *SSTable*.
- Secondary index (`SI_.*.db`): contiene los índices secundarios. Pueden existir varios en cada *SSTable*.
- ...

Compactaciones y lecturas en Cassandra

Compactaciones

Consiste en combinar varias *SSTable* en una sola para eliminar datos obsoletos y reducir el número de ficheros en disco.

Cassandra tomará todas las versiones de una fila y las combinará en una sola versión con las versiones más recientes de cada columna.

Ese dato lo almacenará en una nueva *SSTable* y borrará las antiguas.

Hay que tener en cuenta que este proceso es muy costoso en términos de CPU y E/S, por lo que se debe de realizar de forma controlada. Dará lugar a picos de uso de CPU y disco. Si el espacio en disco es limitado, se puede llegar a llenar por completo antes de que se haya completado la compactación.

Borrado de datos en Cassandra

Un borrado se realiza como una escritura. Se realiza un borrado lógico creando *tombstones* (lapidas) que indican que una columna ha sido borrada. Los datos de estas *tombstones* se irán borrando *de verdad* en las compactaciones.

Lectura

Una lectura intentará en primer lugar obtener el dato de la *memtable*. Si no lo encuentra ahí, aún tiene la posibilidad de encontrarlo en otra caché llamada *row cache*.

La *row cache* es simplemente una memoria donde se almacenan las filas a las que se ha accedido más recientemente.

Si el dato no se encontró ni en la *memtable* ni en la *row cache*, se acudirá al *Bloom filter* de la *SSTable* que nos podrá decir si el dato existe en la *SSTable* o no. Si el *Bloom filter* nos dice que el dato existe, se acudirá al índice de la *SSTable* para encontrar la posición del dato en el fichero de datos. Si el *Bloom filter* nos dice que el dato no existe, se devolverá un error.

```
flowchart TB
l([lectura]) --> memcache
memcache --> f1{encontrado?}
f1 --encontrado --> fin_ok(fin: dato devuelto)
f1 --no encontrado --> rc[row cache]
```

```
rc --> f2{¿econtrado?}
f2 -- encontrado --> fin_ok
f2 -- no encontrado --> bf([Bloom filter])
bf --> st{¿Está en la
SSTable?}
st -- está --> da[Acceso en disco
a la SSTable]
da --> fin_ok
st -- no está --> fallo(Dato no econtrado)
fallo --> fin_nok(fin: fallo de lectura)
```

Distribución y replicación

Distribución y replicación son los conceptos básicos que emplea Cassandra para garantizar la disponibilidad y tolerancia a fallos. En Cassandra ambas tareas se realizan do forma simultánea. Cuando un dato se distribuye, también se replica. Además de esto los datos se van a organizar en función de su clave primaria (PK). La PK (***Partition Key* no confundir con *primary key***) determina en qué nodo se van a escribir lo datos.

Elementos involucrados en la distribución y replicación

- **Nodos virtuales (*Vnodes*)**: aumentan el grado de granularidad de los datos.
- **Particionador**: determina en qué nodo se va a almacenar un dato en función de su PK.
- **Estrategia de replicación**: determina el número de copias que se van a almacenar de cada dato.
- ***Snitch***: determina la topología de la red.

Nodos virtuales (Vnodes) Aumentan el grado de granularidad de los datos. Al comportarse como nodos *reales* permiten que un nodo almacene más datos que los que le corresponderían en una distribución sin *Vnodes*. Esto hace que haya datos replicados en más nodos y reduce el riesgo de que la caída de un nodo provoque la pérdida de datos.

El intervalo de PKs que le correspondería a un nodo se calcula a partir del valor de dos tokens. El primer token se calcula a partir del hash de la dirección IP del nodo. El segundo token se calcula a partir del hash del nombre del cluster.

Cuando añadimos un nuevo nodo, éste asume la responsabilidad sobre un conjunto de datos que le correspondía a otros nodos. Esto hace que se tenga que realizar un proceso de redistribución de datos. Este proceso se realiza de forma automática y transparente para el usuario.

La proporción de *Vnodes* por nodo es configurable.

Particionador El particionador es el encargado de determinar en qué nodo se va a almacenar un dato en función de su PK. El algoritmo de distribución de datos utiliza una **función hash** para calcular el token de un dato a partir de su **PK**. El token es un número de 64 bits que se utiliza para determinar en

qué nodo se va a almacenar el dato (en relación a los tokens del nodo). Como dijimos cada nodo tendrá dos tokens y el dado ha de almacenarse en el nodo *entre cuyos tokens* se encuentre el token del dato. Es decir, si un nodo tiene los tokens 1 y 100 y el hash del dato fuese 50 el dato se almacenará en dicho nodo.

Obviamente los tokens se generan de forma que todo el rango de posibles valores de hash de un dato esté cubierto por los rangos de tokens de los nodos.

Cassandra proporciona tres particionadores:

- Murmur 3Partitioner (por defecto).
- Random Partitioner.
- ByteOrdered Partitioner.

Estrategia de replicación Cassandra utiliza la replicación para asegurar la disponibilidad y tolerancia a fallos. El **factor de replicación** es el valor que indica el número de copias que se van a almacenar de cada dato y **no debe sobrepasar el número de nodos del *datacenter***. El valor de esta propiedad se puede modificar en tiempo de ejecución.

La replicación se realiza de forma automática y transparente para el usuario. Cassandra proporciona dos estrategias de replicación:

- SimpleStrategy (por defecto): Usado para clusters con un único *datacenter*. Las réplicas se distribuyen en los nodos de forma secuencial.
- NetworkTopologyStrategy: Usado para clusters con varios datacenters. Las réplicas se distribuyen en los nodos de forma secuencial en función de los datacenters. Se puede definir el factor de replicación por datacenter.

Snitch El snitch es el encargado de determinar la topología de la red. Es decir, determina a qué *datacenter* y a qué rack pertenece cada nodo. Cassandra proporciona varios snitches:

- Dynamic.
- GoogleCloudSnitch.
- Simple.
- Ec2Snitch.
- Rackinferring.
- …ññadslafkñjbbccd

Primary key, partition key y clustering key

Una *primary key* en Cassandra estará compuesta de **una *partition key*** (simple o compuesta) **y cero o más *culstering keys***. Al definir una *primary key* la columna o columnas que conforman la ***partition key* siempre aparecerán antes** que los campos que conforman las *clustering keys*.

La *primary key* de una tabla de Cassandra puede tener la siguiente estructura:

- **Un único campo**: en cuyo caso ese campo tiene que ser único para todos los registros de la tabla. Es decir, una columna con valores únicos.
- **Dos o más campos**: en este caso lo que tiene que ser único es la combinación de los dos campos. Pueden definirse de dos formas:

- `PRIMARY KEY (<campo1>, <campo2>, …)`: aquí el `<campo1>` será la ***partition key*** y el resto de campos serán *clustering keys*.
- `PRIMARY KEY ((<campo1, campo2), <campo3>, …)`: en este ejemplo `<campo1>` y `<campo2>` constituyen la *partition key* y el resto serán *clustering keys* (ver **partition key compuesta**).

Partition key La principal función de una *partition key* es la distribuir los datos de una manera uniforme entre los nodos del *cluster* y permitir consultas de una manera eficiente. Esto se llevará a cabo aplicando una **función de *hash*** a la *partition key*. Con el *hash* resultante se determinará qué nodo del *cluster* le corresponde y la **partición** dentro de dicho nodo.

***Partition key* simple** Si la *partition key* está formada a partir de una única columna, los valores de este campo serán los que se usen para calcular el *hash*. Una vez calculado el *hash* este determinará en que **partición** se va a guardar el registro. Al mismo tiempo **también** estamos determinando **el nodo** donde se va a almacenar ya que **todos los elementos de una partición han de estar almacenados en el mismo nodo**.

***Partition key* compuesta** Una *partition key* compuesta estará formada por dos o más columnas. Esto implica que se utilizarán varias columnas para determinar dónde se va a almacenar el dato. Esta técncia se utilizará **cuando la cantidad de datos a almacenar es demasiado grande para guardarse en una única partición**. Cuando usamos más de una columna en la *partition key* los datos se dividirán en *pedazos* o *buckets*. Los datos serguirán estando agrupados pero en fragmentos más pequeños. Este método puede ser efectivo en la reducción de *puntos calientes* o **congestión en escrituras**, cuando la partición de un nodo recibe gran cantidad de escrituras.

Clustering key *Clustering* es el proceso de ordenar los datos de una partición y se basa en las columnas definidas como las ***clustering keys***. La definición de las columnas que serán *clustering keys* ha de hacerse con antelación. Esto ha de hacerse así ya que la elección de las columnas que serán *clustering keys* depende de cómo vamos a usar los datos en la aplicación. Es decir, la elección de las *clustering keys* tendrá un gran impacto en el rendimiento de la aplicación que consuma esos datos.

Todos los datos de una partición se almacenan de forma contigua en el dispositivo de almacenamiento ordenados según las columnas definidas como *clustering keys*. Esto hace que la recuperación de los datos sea muy eficiente (siempre que se haga respecto a estas columnas).

Consistencia

Teniendo en cuenta el teorema CAP (Consistency Availability Partition tolerance), Cassandra sacrifica la consistencia para garantizar la disponibilidad (availability) y tolerancia a fallos (partitioning). Esto significa que los datos no se replican de forma síncrona en todos los nodos. En su lugar, se replican de forma asíncrona en los nodos que se le indiquen en la estrategia de replicación. Esto hace que los datos no estén disponibles en todos los nodos al mismo tiempo. Por

lo tanto, si se realiza una lectura en un nodo, es posible que no se obtenga el dato más actualizado.

Se puede definir el grado de consistencia que deseamos a la hora de realizar una lectura o escritura. Pero hay que tener en cuenta que cuanto mayor sea el nivel de consistencia exigido peor serán las otras propiedades de Cassandra (disponibilidad y tolerancia a fallos). Es decir, cuanto mayor sea la consistencia requerida menor será el rendimiento. Si exigimos máxima consistencia en escritura el dato habrá que **escribirlo a todos los nodos** lo que será más lento y costoso. Del mismo modo, si exigimos máxima consistencia en la lectura habrá que **consultar a todos los nodos** para asegurarnos de que vamos a obtener el valor más reciente del dato.

Los niveles de consistencia que se pueden especificar son:

- **Any**: Sólo para escrituras. Se garantiza que la escritura se ha realizado en al menos un nodo.
- **One/Two/Three**: En escrituras se garantiza que el datos ha sido replicado en uno/dos/tres nodos. En lecturas se garantiza que se ha leído el dato más actual de uno/dos/tres nodos.
- **Quorum**: En escrituras se garantiza que el dato ha sido replicado en un *quorum* de nodos. En lecturas se garantiza que se ha leído el dato más actual de un *quorum* de nodos.
- **Local quorum**: En escrituras se garantiza que el dato ha sido replicado en el *quorum* del datacenter local. En lecturas se garantiza que se ha leído el dato más actual de un *quorum* del datacenter local.
- **Each quorum**: En escrituras se garantiza que el dato ha sido replicado en un *quorum* de veces de todos los datacenters. En lecturas se garantiza que se ha leído el dato más actual de un *quorum de nodos de todos los datacenters.
- **All**: En escrituras se garantiza que el dato ha sido replicado en todos los nodos. En lecturas se garantiza que se ha leído el dato más actual de todos los nodos.

El quorum se define según la siguiente fórmula:

$quorum = (factor\ de\ replicacin/2) + 1$

Conceptos generales en modelado de datos

Normalización de datos

La normalización es el proceso de diseñar las tablas y relaciones entre ellas de acuerdo a unas reglas diseñadas con el objetivo de **eliminar la redundancia y la inconsistencia**. Estas técnicas se aplican en los modelos relacionales y surgen de la necesidad de **optimizar el almacenamiento de datos en disco**. En la época en que surgieron estas técnicas, el almacenamiento en disco era muy caro y por tanto se buscaba optimizarlo. En la actualidad la situación ha cambiado y el almacenamiento en disco es muy barato, por lo que la normalización no es tan necesaria como antes. Esta es una de las razones por las que las bases de datos NoSQL como Cassandra no utiliza el modelo relacional.

Ventajas de la normalización

- **Menor espacio de almacenamiento**: Al eliminar la redundancia se reduce el espacio de almacenamiento.
- **Menor tiempo de escritura**: Al eliminar la redundancia se reduce el tiempo de escritura.
- **Mayor integridad de los datos**: Al eliminar la redundancia se evita la posibilidad de que los datos se corrompan (dos copias del mismo dato con distintos valores).

Desventajas de la normalización

- **Mayor tiempo de lectura**: Al eliminar la redundancia se aumenta el tiempo de lectura, ya que hay que realizar **más consultas** para obtener los datos.
- **Mayor complejidad en las consultas**: Al eliminar la redundancia se aumenta la **complejidad de las consultas** (sentencias *join*).

Desnormalización de datos

La desnormalización es el proceso de añadir redundancia a los datos con el objetivo de **mejorar el rendimiento**. En Cassandra se desnormalizan los datos para evitar las consultas *join*, que serían muy costosas en Cassandra (Cassandra no admite sentencias *join*). Hay que tener en cuenta que el precio del espacio de almacenamiento en disco es muy barato en la actualidad y las velocidades de escritura y lectura son muy altas, por lo que la desnormalización no es un problema en este aspecto.

Ventajas de la desnormalización Son a la inversa que las desventajas de la normalización:

- **Menor tiempo de lectura**: Al añadir redundancia se reduce el tiempo de lectura, ya que hay que realizar **menos consultas** para obtener los datos.
- **Menor complejidad en las consultas**: Al añadir redundancia se reduce la **complejidad de las consultas** (se pueden evitar las sentencias *join*, y los *join* anidados).

Desventajas de la desnormalización

- **Múltiples escrituras**: Al añadir redundancia se aumenta el tiempo de escritura, ya que hay que escribir los datos en varios sitios.
- **Integridad manual de los datos**: Al añadir redundancia se aumenta la posibilidad de que los datos se corrompan (dos copias del mismo dato con distintos valores).

Modelado relacional vs Cassandra

Cassandra es una base de datos diseñada con el objetivo de optimizar el rendimiento de las lecturas a escala. Para ello se sacrifica el rendimiento de las escrituras y la integridad de los datos. Por tanto, el modelado de datos en Cassandra es muy diferente al modelado de datos en bases de datos relacionales.

En Cassandra se emplea la desnormalización de datos en el proceso de modelado.

En un sistema de bases de datos relacional el modelado de datos sigue los siguientes pasos:

```
graph TB
ad([Analizar los datos]) --> id([Identificar las
entidades y
relaciones])
id --> md([Diseñar las tablas
aplicando normalización y
claves foráneas])
md --> app([La aplicación accede
al modelo de datos])
```

El proceso se inicia con los datos *crudos*. Se analizan sus características y se identifican las entidades y relaciones entre ellas. A continuación se diseñan las tablas aplicando la normalización para minimizar la redundancia y finalmente la aplicación accede al modelo de datos diseñando las consultas y demás operaciones **condicionada por el diseño previo**.

En cambio en Cassandra (y otras bases de datos NoSQL) el proceso es el siguiente:

```
graph TB
app([Analizar las
necesidades del
usuario]) --> q([Identificar las
consultas necesarias])
q --> data([Diseñar las tablas
en función de
las consultas])
```

Partimos de analizar cómo necesitamos consumir los datos, qué operaciones necesitará realizar la aplicación sobre ellos. En función de estas necesidades se diseñan las tablas y finalmente se organizan los datos en función del diseño previo.

Se parte de la aplicación y se *sube* hasta los datos, en vez de partir de los datos y *bajar* hasta la aplicación.

Flujo de trabajo en el modelado de datos en Cassandra

El flujo de trabajo en el modelado de datos en Cassandra es el siguiente:

```
graph LR
mc[Modelo conceptual] --> mcl[Mapeado del concepto
a la lógica]
awf[Flujo de trabajo
de la aplicación] --> mcl
mcl --> mld[Modelo lógico
de los datos]
mld --> mlf[Optimizaciones
físicas]
```

```
mlf --> mfd[Modelo físico
de los datos]
```

En primer lugar se analizará el flujo de trabajo de la aplicación (¿Qué hacen los usuarios en la aplicación?) y se determinan también cual es el modelo conceptual de los datos: qué entidades podemos encontrar y relaciones que existen entre las mismas. Con esta información empezaremos a enlazar la lógica de la aplicación al modelo de datos. Seguirán las optimizaciones físicas que se puedan detectar y finalmente el modelo físico de los datos, que es cómo se van a almacenar los datos en disco.

Creación de un cluster

Para realizar las pruebas con Cassandra crearemos un cluster de 3 nodos en local. Para ello utilizaremos Docker y Docker Compose. Los nodos del cluster serán los siguientes: cass1, cass2 y cass3. Los nodos cass1 y cass2 serán los nodos semilla del cluster. El nodo cass3 se unirá al cluster posteriormente.

Obteniendo la imagen de Cassandra

Usaremos la imagen oficial de Cassandra que se encuentra en Docker Hub: https://hub.docker.com/_/cassandra.

Para obtener la imagen ejecutaremos el siguiente comando:

```
docker pull cassandra:latest
```

Creando el `docker-compose.yml`

```yaml
version: "3.8"
networks:
  cassandra:
services:
  cass1:
    image: cassandra:latest
    container_name: cass1
    hostname: cass1
    mem_limit: 2g
    healthcheck:
      test: ["CMD", "cqlsh", "-e", "describe keyspaces"]
      interval: 5s
      timeout: 5s
      retries: 60
    networks:
      - cassandra
    ports:
      - "9042:9042"
    volumes:
      - ./data/cass1:/var/lib/cassandra # Para almacenar los datos de la base de datos.
      - ./etc/cass1:/etc/cassandra # Para poder editar los archivos de configuración.
      - ./scripts:/scripts # Para escribir y ejecutar scripts de CQL.
```

```
    environment: &environment
      CASSANDRA_SEEDS: "cass1,cass2"
      CASSANDRA_CLUSTER_NAME: SolarSystem
      CASSANDRA_DC: Mars
      CASSANDRA_RACK: West
      CASSANDRA_ENDPOINT_SNITCH: GossipingPropertyFileSnitch
      CASSANDRA_NUM_TOKENS: 128

  cass2:
    image: cassandra:latest
    container_name: cass2
    hostname: cass2
    mem_limit: 2g
    healthcheck:
      test: ["CMD", "cqlsh", "-e", "describe keyspaces"]
      interval: 5s
      timeout: 5s
      retries: 60
    networks:
      - cassandra
    ports:
      - "9043:9042"
    volumes:
      - ./data/cass2:/var/lib/cassandra
      - ./etc/cass2:/etc/cassandra
      - ./scripts:/scripts
    environment:
      <<: *environment
    depends_on:
      cass1:
        condition: service_healthy
  cass3:
    image: cassandra:latest
    container_name: cass3
    hostname: cass3
    mem_limit: 2g
    healthcheck:
      test: ["CMD", "cqlsh", "-e", "describe keyspaces"]
      interval: 5s
      timeout: 5s
      retries: 60
    networks:
      - cassandra
    ports:
      - "9045:9042"
    volumes:
      - ./data/cass3:/var/lib/cassandra
      - ./etc/cass3:/etc/cassandra
      - .scripts:/scripts
    environment:
```

```
    <<: *environment
  depends_on:
    cass2:
      condition: service_healthy
```

Este archivo de configuración crea un *cluster* de 3 nodos de Cassandra en local.

Cada nodo tiene tres volúmenes asociados:

- `/data/cass1:/var/lib/cassandra`: Contiene los ficheros de la base de datos del nodo.
- `/etc/cass1:/etc/cassandra`: Contiene los archivos de configuración del nodo.
- `./scripts:/scripts`: Contendrá los scripts de CQL que vayamos escribiendo.

Antes de arrancar el *cluster* hemos de crear los directorios `data/cass1`, `data/cass2` y `data/cass3` y los directorios `etc/cass1`, `etc/cass2` y `etc/cass3`.

A continuación copiaremos los archivos de configuración de Cassandra en los directorios `etc/cass1`, `etc/cass2` y `etc/cass3`.

Para ello hemos primero de obtener los archivos de configuración de Cassandra. Para ello los copiaremos de un contenedor de Cassandra que se ejecutará temporalmente.

```
docker run --rm -d --name temp cassandra:latest
docker cp temp:/etc/cassandra .
docker stop temp
```

las opciones `-rm` y `-d` indican que el contenedor se elimine automáticamente al pararlo y que se ejecute en segundo plano.

Ahora tendremos un directorio `cassandra` con los archivos de configuración que hemos copiado de la máquina temporal.

Copiar los archivos de configuración de Cassandra

Copiaremos los archivos de configuración de Cassandra en los directorios `etc/cass1`, `etc/cass2` y `etc/cass3`:

```
cp -a ./cassandra ./etc/cass1/
cp -a ./cassandra ./etc/cass2/
cp -a ./cassandra ./etc/cass3/
```

La opción `-a` de `cp` indica que se copien los archivos de forma recursiva y que se conserven los permisos, propietarios y fechas de los archivos.

Si estamos en Windows hemos de hacer lo mismo copiando directamente el contenido del directorio `cassandra` a `etc/cass1` y los demás.

Todo esto lo hacemos para que podamos **modificar los archivos de configuración de Cassandra** de cada nodo de forma independiente editando los archivos de los directorios locales `etc/cass1`, `etc/cass2` y `etc/cass3`.

Nota respecto al volumen `scripts`: El volumen `scripts` se ha creado para poder escribir y ejecutar scripts de CQL desde dentro de los contenedores. Para ello hemos de copiar los scripts de CQL en el directorio `scripts` del directorio raíz del proyecto. Los scripts de CQL se ejecutarán desde dentro de los contenedores con el siguiente comando:

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

No es necesario crear el directorio `/scripts` en los contenedores ya que se crea automáticamente al crear el volumen `scripts`.

Sí es necesario crear el directorio `scripts` en el host… *creo*.

Iniciar el *cluster*

Los nodos `cass1` y `cass2` serán los nodos designados como semilla del *cluster*.

Para iniciar el *cluster* ejecutaremos el siguiente comando:

```
docker-compose up -d
```

y el resultado debería ser el siguiente:

```
> docker-compose up -d
[+] Running 3/3
   Container cass1 Healthy                                    0.0s
   Container cass2 Healthy                                    0.5s
   Container cass3 Started                                    0.7s
>
```

Para comprobar que los contenedores se han iniciado correctamente ejecutaremos el siguiente comando:

Revisar esto.

```
docker-compose ps
```

Abrir una consola de CQLSH

Para empezar a trabajar con Cassandra hemos de abrir una consola de CQLSH que nos permitirá ejecutar comandos CQL.

Para abrir una consola de CQLSH ejecutaremos el siguiente comando:

```
docker exec -it cass1 cqlsh
```

Con este último paso estaremos listos para empezar a trabajar con Cassandra.

version: “3.8” networks: cassandra: services: cass1: image: cassandra:latest container_name: cass1 hostname: cass1 mem_limit: 2g healthcheck: test: [“CMD”, “cqlsh”, “-e”, “describe keyspaces”] interval: 5s timeout: 5s retries: 60 networks: - cassandra ports: - “9042:9042” volumes: - ./data/cass1:/var/lib/cassandra - ./etc/cass1:/etc/cassandra - ./scripts:/scripts

```
environment: &environment
  CASSANDRA_SEEDS: "cass1,cass2"
  CASSANDRA_CLUSTER_NAME: SolarSystem
  CASSANDRA_DC: Mars
```

Estructura de un sistema Cassandra

Figure 3: Estructura de un sistema Cassandra

```
CASSANDRA_RACK: West
CASSANDRA_ENDPOINT_SNITCH: GossipingPropertyFileSnitch
CASSANDRA_NUM_TOKENS: 128
```

cass2: image: cassandra:latest container_name: cass2 hostname: cass2 mem_limit: 2g healthcheck: test: [“CMD”, “cqlsh”, “-e”, “describe keyspaces”] interval: 5s timeout: 5s retries: 60 networks: - cassandra ports: - “9043:9042” volumes: - ./data/cass2:/var/lib/cassandra - ./etc/cass2:/etc/cassandra - ./scripts:/scripts environment: «: *environment depends_on: cass1: condition: service_healthy cass3: image: cassandra:latest container_name: cass3 hostname: cass3 mem_limit: 2g healthcheck: test: [“CMD”, “cqlsh”, “-e”, “describe keyspaces”] interval: 5s timeout: 5s retries: 60 networks: - cassandra ports: - “9045:9042” volumes: - ./data/cass3:/var/lib/cassandra - ./etc/cass3:/etc/cassandra - ./scripts:/scripts environment: «:* environment depends_on: cass2: condition: service_healthy

Lenguaje CQL (*Cassandra Query Language*)

Este lenguaje tiene muchas similitudes con SQL, pero también algunas diferencias. En esta sección vamos a ver las principales diferencias entre ambos lenguajes.

Cassandra **no permite realizar operaciones de *join*** entre tablas. Esto es debido a que las tablas en Cassandra están diseñadas para ser consultadas de forma independiente. Por lo tanto, si necesitamos realizar una consulta que implique datos de varias tablas, tendremos que realizar varias consultas y combinar los resultados en nuestra aplicación.

Si queremos hacer una agrupación de datos sólo podremos hacerlo con respecto a las columnas de la clave primaria. Por ejemplo, si tenemos una tabla con las columnas `id`, `name`, `age` y `city` y queremos agrupar por `city` y `age` tendremos que crear una tabla con una clave primaria compuesta por `city` y `age`. No podremos agrupar por `city` y `name` porque `name` no forma parte de la clave primaria.

Los elementos de una base de datos SQL tienen una correspondencia directa con los elementos de una base de datos Cassandra:

SQL	Cassandra
Base de datos	Keyspace
Tabla	*Column family* - CF
Primary key	*Primary key / Row key*
Column name	*Column name / key*
Column value	*Column value*

Tipos de datos en Cassandra

Categoría	Tipo de dato CQL	Descripción
String	`ascii`	Cadena de caracteres ASCII
”	`text`	Cadena de caracteres UTF-8
”	`varchar`	Cadena de caracteres UTF-8
”	`inet`	Dirección IP
Numeric	`int`	Número entero de 32 bits
”	`bigint`	Número entero de 64 bits
”	`float`	Número de coma flotante de 32 bits
”	`double`	Número de coma flotante de 64 bits
”	`decimal`	Número decimal de precisión variable
”	`varint`	Número entero de precisión variable
”	`counter`	Contador de 64 bits (no se admite como clave)
UUID	`uuid`	Identificador único universal
”	`timeuuid`	Identificador único universal con información de tiempo
Collections	`list`	Lista de elementos ordenada
”	`set`	Conjunto de elementos no ordenado
”	`map`	Mapa de pares clave-valor
Misc	`boolean`	Valor booleano
”	`blob`	Secuencia de bytes
”	`timestamp`	Marca de tiempo

Comentarios en CQL

Cassandra admite tres tipos de comentarios:

- Comentarios de una línea: `--<`
- Comentarios de una línea: `//`
- Comentarios de varias líneas: `/* */`

Creación del modelo de datos

Ejecución de scripts de CQL

Para ejecutar instrucciones CQL podemos simplemente lanzar una shell de CQLSH en un contenedor de Cassandra como vimos en la sección de creación del cluster:

```
docker exec -it cass1 cqlsh
```

Pero pronto nos daremos cuenta de que hay instrucciones que tienen una sintaxis demasiado compleja para ser ejecutadas desde la shell. Por ello, lo más cómodo es escribir las instrucciones CQL en un fichero de texto y ejecutarlas desde la shell de CQLSH.

Los *scripts* de CQL son ficheros de texto con extensión `.cql` que contienen instrucciones CQL. Para ejecutar un *script* de CQL desde la shell de CQLSH utilizamos la sentencia `SOURCE`:

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

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

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

Como en nuestro ejemplo vamos a utilizar contenedores Docker, vamos a hacer lo siguiente.

Cuando creamos el fichero `docker-compose.yml` indicamos que el directorio `./scripts` del host se montase en el directorio `/scripts` de los contenedores. Por lo tanto, si queremos ejecutar un *script* de CQL desde la línea de comandos, lo que tenemos que hacer en primer lugar es copiar el *script* en el directorio `scripts` del host para luego ejecutar el siguiente comando:

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

Creación de un *keyspace*

Para crear un *keyspace* utilizamos la sentencia `CREATE KEYSPACE`:

```
CREATE KEYSPACE <keyspace_name>
  WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : <n> };
```

Existe también la sentencia opcional `AND DURABLE_WRITES = <verdadero o falso>` que indica que si los datos se han de escribir o no en el disco. Esta opción está activada por defecto.

Por ejemplo, para crear un *keyspace* llamado `my_keyspace` con una estrategia de replicación `SimpleStrategy` y un factor de replicación de 1 en el *datacenter 1* y 3 en el *datacenter 2* utilizaríamos la siguiente sentencia:

```
CREATE KEYSPACE my_keyspace
  WITH REPLICATION = { 'class' : 'NetworkTopologyStrategy', 'dc1' : 1, 'dc2' : 3 };
```

Para indicar que vamos a utilizar el *keyspace* `my_keyspace` utilizamos la sentencia `USE`:

```
USE my_keyspace;
```

Modificar un *keyspace* Para modificar un *keyspace* utilizamos la sentencia `ALTER KEYSPACE`:

```
ALTER KEYSPACE <keyspace_name>
  WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : <n> };
```

Borrar un *keyspace* Para borrar un *keyspace* utilizamos la sentencia `DROP KEYSPACE`:

```
DROP KEYSPACE <keyspace_name>;
```

Creación de una tabla Para crear una tabla utilizamos la sentencia `CREATE TABLE`:

```
CREATE TABLE <table_name> [     IF NOT EXISTS ] (
  <column_name> <type> PRIMARY KEY,
  <column_name> <type>,
  ...
);
```

Veamos en detalle con un ejemplo:

```
CREATE TABLE monkey_species (
    species text PRIMARY KEY,
    common_name text,
    population varint,
    average_weight float,
    average_height float
) WITH comment = 'Tabla que almacena información sobre especies de monos';
```

En este ejemplo hemos creado una tabla llamada `monkey_species` con las siguientes columnas:

- `species`: clave primaria de tipo `text`.
- `common_name`: columna de tipo `text`.
- `population`: columna de tipo `varint`.
- `average_weight`: columna de tipo `float`.
- `average_height`: columna de tipo `float`.

También hemos añadido un comentario a la tabla.

Otro ejemplo algo más complejo:

```
CREATE TABLE timeline (
    user_id uuid,
    posted_month int,
    posted_time timeuuid,
    body text,
    posted_by text,
    PRIMARY KEY (user_id, posted_month, posted_time)
) WITH COMPACTION = { 'class' : 'LeveledCompactionStrategy' };
```

En este ejemplo hemos creado una tabla llamada `timeline` en la que definimos la clave primaria como una clave primaria compuesta por tres columnas:

- `user_id`: columna de tipo `uuid`.
- `posted_month`: columna de tipo `int`.
- `posted_time`: columna de tipo `timeuuid`.

No hemos especificado cual es la partition key y cual es la clustering key. Si queremos que `user_id` sea la partition key y `posted_month` y `posted_time` sean las clustering keys deberíamos haber definido la clave primaria de la siguiente forma:

```
PRIMARY KEY ((user_id) posted_month, posted_time)
```

al encerrar una o más columnas entre paréntesis indicamos que son la partition key. En este caso `user_id` es la partition key y `posted_month` y `posted_time` son las clustering keys.

Si tenemos una clustering key compuesta por varias columnas podemos indicar también la ordenación de las mismas. Por ejemplo, si queremos que `posted_month` sea descendente y `posted_time` ascendente deberíamos haber definido la clave primaria de la siguiente forma:

```
PRIMARY KEY ((user_id) posted_month, posted_time ) WITH CLUSTERING ORDER BY (posted_month
```

Si hay alguna columnas cuyos valores no van a cambiar podemos indicarlo con el modificador `static`. Por ejemplo, si queremos que `posted_by` sea una columna estática deberíamos haber definido la tabla de la siguiente forma:

```
CREATE TABLE timeline (
    user_id uuid,
    posted_month int,
    posted_time timeuuid,
    body text,
    posted_by text STATIC,
    PRIMARY KEY (user_id, posted_month, posted_time)
) WITH COMPACTION = { 'class' : 'LeveledCompactionStrategy' };
```

Nótese que: Una **primary key ha de ser única** pero ni la PK (partition key) ni la CK (clustering key) han de ser únicas por separado.

Modificar una tabla Para modificar una tabla utilizamos la sentencia `ALTER TABLE`:

```
ALTER TABLE <table_name>
  ADD <column_name> <type>,
  DROP <column_name>,
  ALTER <column_name> TYPE <type>,
  RENAME <column_name> TO <new_column_name>,
  WITH <option> = <value>,
  ...
```

Un ejemplo sería:

```
ALTER TABLE monkey_species
  ADD average_lifespan int,
  DROP average_height,
  ALTER average_weight TYPE float,
  RENAME common_name TO name,
  WITH comment = 'Tabla que almacena información sobre especies de monos';
```

Si quisiésemos eliminar todos los registros de una tabla podemos utilizar la sentencia `TRUNCATE`:

```
TRUNCATE [ TABLE ] <table_name>;
```

Para eliminar una tabla utilizamos la sentencia `DROP TABLE`:

```
DROP TABLE [ IF EXISTS ] <table_name>;
```

Ver la definición de una tabla

Para ver la definición de una tabla utilizamos la sentencia `DESCRIBE` indicando a qué tabla nos referimos:

```
DESCRIBE TABLE <table_name>;
```

Por ejemplo, para ver la estructura de la tabla `sbd.miembros`:

```
DESCRIBE TABLE sbd.miembros;
```

Importar datos ds CSV a una tabla

```
COPY keypsace.tableName (col1,col2,col3.....) FROM 'file/file.csv' WITH DELIMITER=',' AND
```

`IF NOT EXISTS`

La utilidad de esta cláusula es evitar que se produzca un error si intentamos crear un *keyspace* o una tabla que ya existe. Si la cláusula `IF NOT EXISTS` está presente y el *keyspace* o la tabla ya existen, la sentencia no tiene ningún efecto.

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_nacimien
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"
```

`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.