Gestores NoSQL – Apache Cassandra

Marta Zorrilla – Diego García-Saiz

Enero 2017

Este material se ofrece con licencia: Creative Commons BY-NC-SA 4.0

Tabla de contenidos

Bibliografía y documentación complementaria

- Bibliografía:
  - J. Carpenter y Ebe Hewitt: **Cassandra: The Definitive Book.** O'Reilly Media (2016)
  - DBA guide to NoSQL: http://www.datastax.com/dbas-guide-to-nosql (ebook gratuito)
  - Getting started and user information (web page): http://docs.datastax.com/en/landing_page/doc/landing_page/current.html

- Tutoriales:
  - https://academy.datastax.com/tutorials

- CQL 3.1:
  - http://docs.datastax.com/en/cql/3.1/cql/cql_intro_c.html
  - https://www.datastax.com/wp-content/uploads/2013/03/cql_3_ref_card.pdf

Introducción: Apache Cassandra

- Es un almacén altamente escalable, eventualmente consistente y distribuido de estructuras clave-valor.
  - Iniciado por Facebook
  - Código abierto en 2008
  - Proyecto apache en 2009
    - Licencia: Apache License 2.0
  - Escrito en Java
  - Multiplataforma
  - Versión actual: 3.9, publicada el 29 de septiembre de 2016.
  - Web: http://cassandra.apache.org/

Introducción: ¿Quién usa Apache Cassandra y para qué?

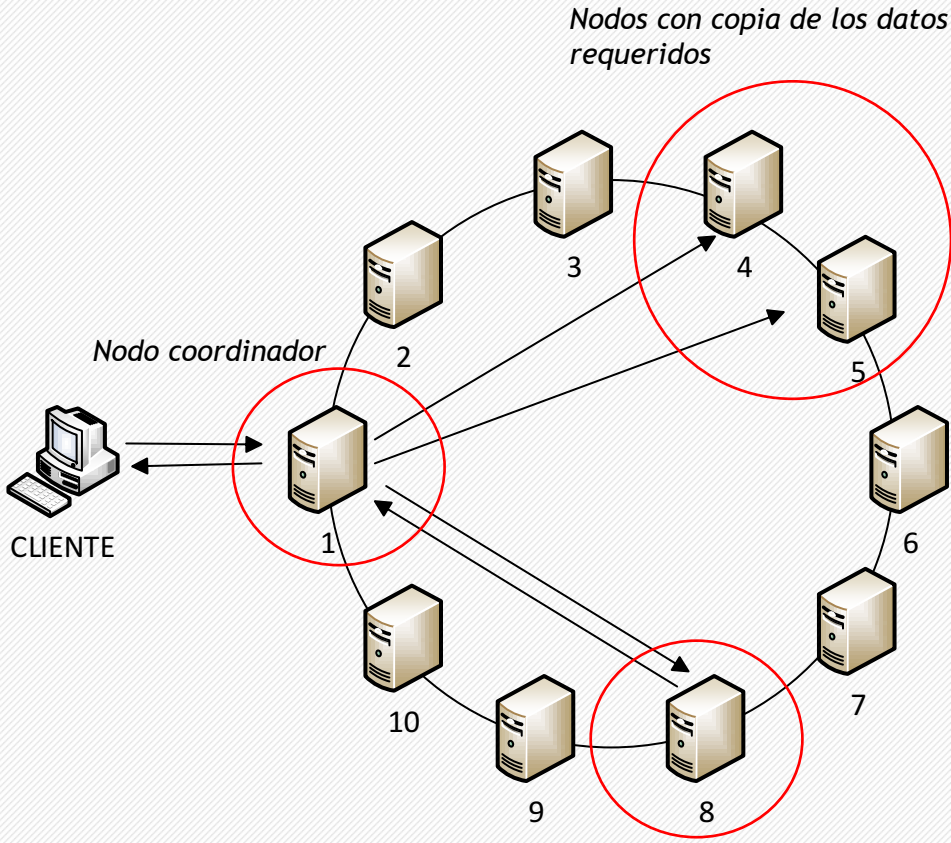
- Algunos usuarios conocidos son:
  - Apple usa 75,000 nodos de Cassandra para *iMessage, iTunes passwords y otros servicios más*
  - CERN usa un prototipo basado en Cassandra para el experimento ATLAS para archivar su sistema de monitorización on-line
  - Facebook usa Cassandra para su sistema de búsquedas en la bandeja de entrada, con una implemetanción de más de 200 nodos
  - Netflix usa Cassandra como base de datos de back-end para su servicio de streaming
  - eBay lo utiliza, entre otros, para anotar los clic del usuario en los botones "Like", "Want" or "Own" en su pág. de favoritos.
  - Etc…

Introducción: Revisión histórica

- Creado por Avanash Lakshaman y Prashany Malik para ser integrado en el motor de búsqueda de Facebook.
  - Concretamente para buscar mensajes en la bandeja de entrada.
- Influido por *Google BigTable* (modelo de datos orientado a columnas) y *Amazon Dynamo* (arquitectura- distribuida *peer-to-peer*).
- Proceso de liberación:
  - Julio 2008: publicado como proyecto *open source* en Google Code.
  - Marzo 2009: pasa a formar parte de los proyectos *Apache Incubator*.
  - Febrero 2010: se convierte en proyecto *top-level* de *Apache Software Foundation*.

Arquitectura

- Varios nodos independientes comunicados mediante un protocolo ***P2P (peer-to-peer)***.
  - Todos los nodos intercambian información con el resto de forma continua.
- No hay nodos "principales". Los **clientes** pueden conectarse a **cualquiera de los nodos** para realizar las **operaciones de lectura y escritura**.
  - El nodo al que se conecta el **cliente** actúa como **coordinador** entre éste y el resto de nodos en dónde se encuentran los datos afectados por la consulta.
  - El **coordinador** determina qué nodos deben responder a la consulta.



Arquitectura: componentes principales en Cassandra

- **Clúster:** contiene uno o varios *datacenter*.
- **Centro de datos (*datacenter*):** colección de nodos, que puede ser virtual o físico.
- **Nodo:** componente básico de la infraestructura de Cassandra en donde se almacenan los datos.
  - ***Commit log*:** fichero en donde se almacena la información sobre cambios en los datos. Sirve para recuperar los datos en caso de un fallo en el sistema.
  - ***MemTable*:** estructura de almacenamiento en memoria. Contiene los datos que aún no han sido escritos en un *SSTable*.
  - ***SSTable*:** fichero que almacenan los datos escritos en disco. Cada fichero *SSTable* es inmutable una vez creado.

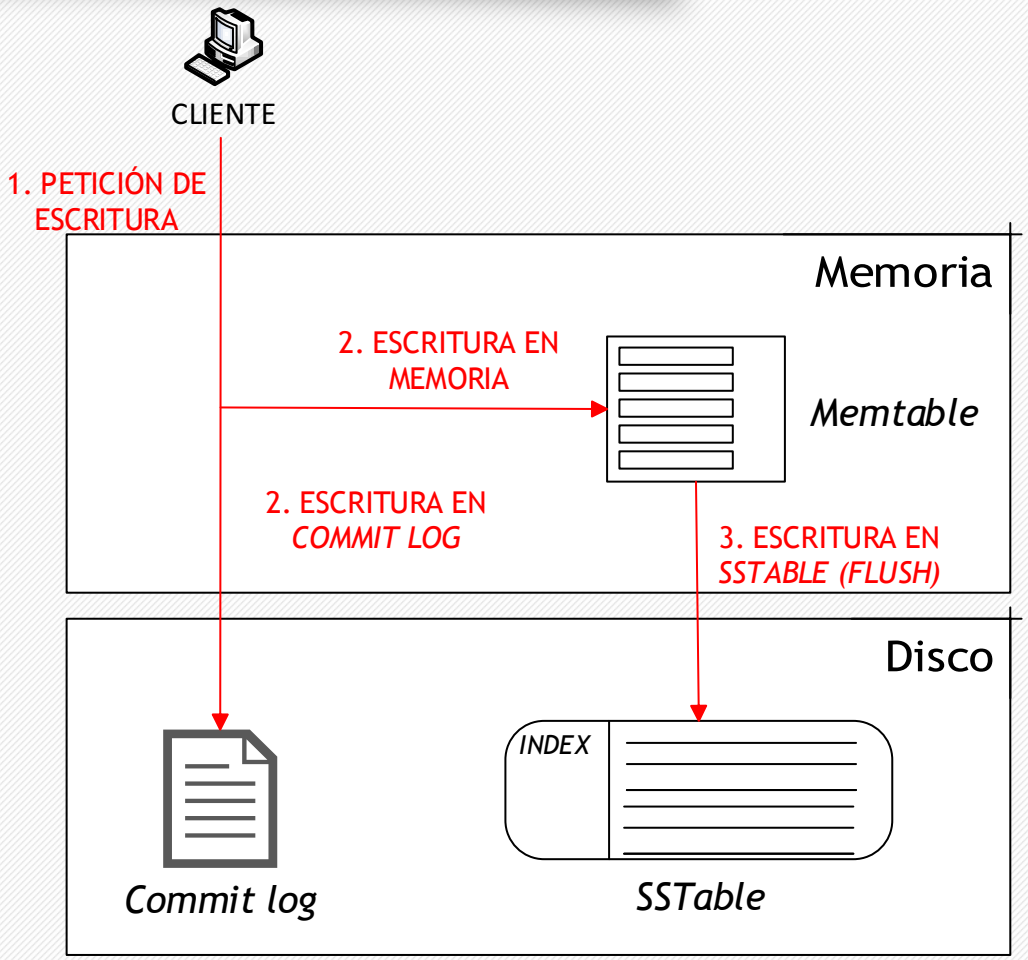
Arquitectura: componentes principales para la configuración de Cassandra

- ***About internode communications (gossip):*** Protocolo de comunicación *peer-to-peer* para descubrir y compartir información sobre la localización y estado de los nodos en un clúster de Cassandra.
- ***Partitioner***: Determina cómo se distribuyen los datos entre los nodos (primera copia, resto de copias).
- ***Replica placement strategy***: Define la estrategia a seguir para almacenar copias de los mismos datos en diferentes nodos, de forma que se aseguren la accesibilidad y la tolerancia a fallos. Se pueden definir diferentes estrategias. No hay copia principal ni copias secundarias de los datos, todas las copias, incluida la primera, son réplicas.
- ***Snitch***: Define la topología que utilizan las estrategias de replicación para colocar las réplicas y dirigir las consultas de forma eficiente.

Arquitectura: peticiones de escritura de datos

M.Zorrilla - D.García

- **Proceso de escritura:**
  - Escritura en el *Commit log*.
  - Escritura en la *MemTable*.
  - Borrado (*Flush*) de la *MemTable*.
  - Escritura en la *SSTable*.
  - Compactación (*Compaction*).



Arquitectura: peticiones de escritura de datos

- **Escritura** en ***MemTable***: almacena las escrituras de cada familia de columnas (símil de tablas en relacional) de forma temporal en memoria.
  - Por ello, en Cassandra, las escritura son “baratas”.

- **Escritura** en ***Commit log*:** en cada nodo, el *Commit log* almacena **TODA** la actividad de escritura para garantizar la durabilidad de los datos.
  - Si el sistema se apaga, se pueden recuperar las escrituras desde el *Commit log*

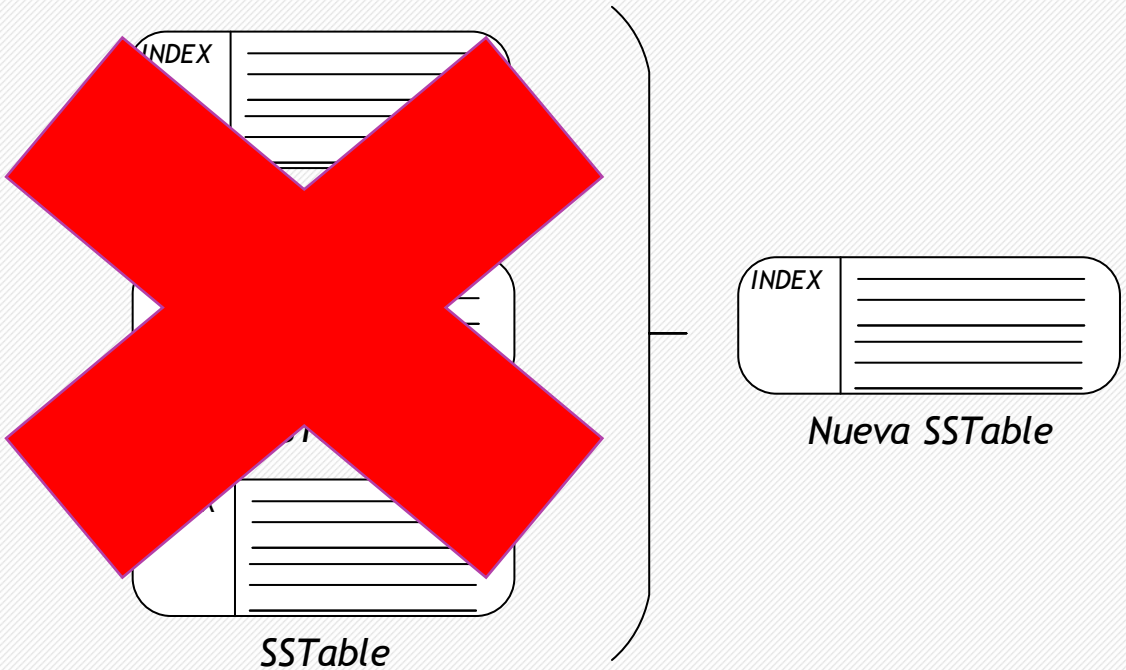
Arquitectura: peticiones de escritura de datos

- ***Flush*** de la ***MemTable***: cuando se llena, se libera la memoria, realizando un borrado de los datos.
- **Escritura** en ***SSTable***: los datos borrados en la fase de *Flush* de la *MemTable* se escriben en disco, en ficheros llamados *SSTable*.
  - Los ficheros *SSTable* son inmutables, no se vuelve a escribir en ellos después de volcar los datos de la *MemTable*.
  - Por ello, los datos de una misma partición pueden estar repartidos en varias *SSTable*.
  - Este es el motivo por el que las lecturas son más “caras” que las escrituras en Cassandra, ya que una lectura suele requerir el acceso y búsqueda en varias *SSTable*.

Arquitectura: peticiones de escritura de datos

M.Zorrilla - D.García

- **Compactación (*Compaction*):** en esta fase, ejecutada periódicamente por Cassandra, se trata de reducir el número de ficheros *SSTable*, eliminando aquellos obsoletos (datos antiguos).
  - El objetivo es hacer más eficientes las lecturas.

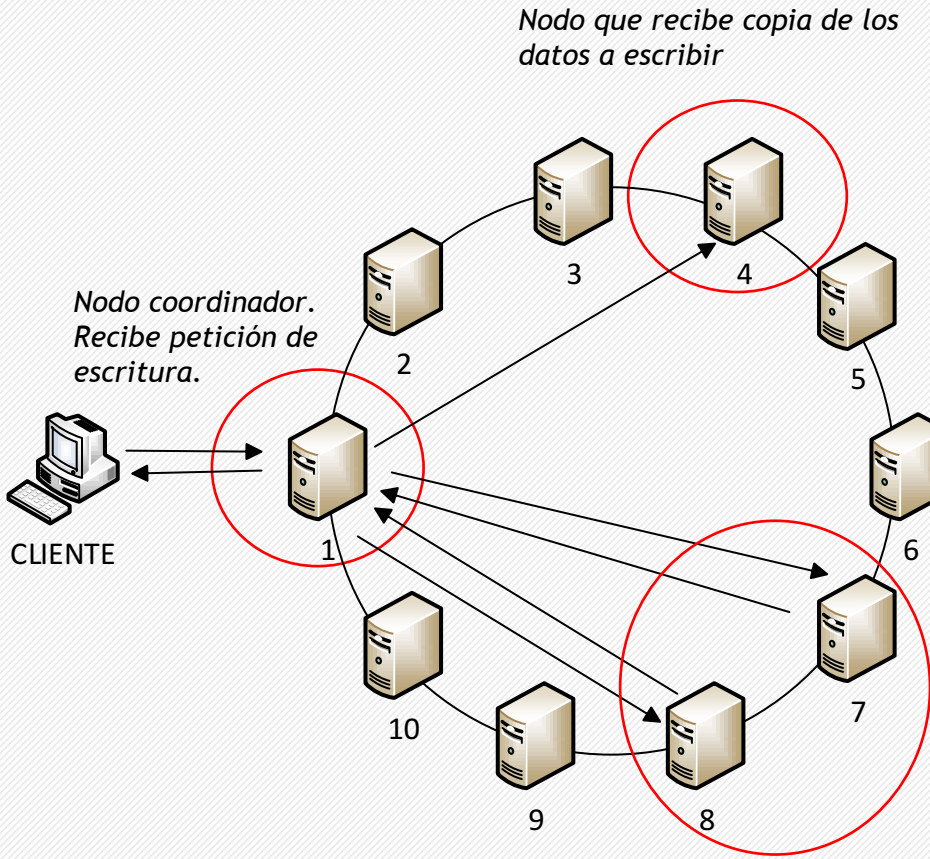


Arquitectura: peticiones de escritura de datos

- A tener en cuenta:
  - Las escrituras son “baratas” porque inicialmente Cassandra no escribe los datos en disco, sino en memoria.
  - Siempre se puede escribir, aunque haya algún nodo caído.
    - ¿Siempre, siempre?...
      - Podría pasar que todos los nodos en donde hay que escribir las réplicas estuviesen caídos, pero es algo muy improbable.
  - No hay búsquedas en la escritura, cuando se realiza una escritura de una fila ya existente (*Primary Key* existente en la BD), la fila anterior es sobrescrita (*upsert*):
    - En *MemTable*, se sobrescribe, siendo el valor de *Primary Key* único para cada fila.
    - En el *commit log*, se almacenan los *logs* de todos los insertados de datos, por lo que habrá tantos *logs* como escrituras se hagan (no se sobrescriben, si tienen la misma *Primary Key*).

Arquitectura: peticiones de escritura de datos

- **Peticiones de escritura de datos a nivel de clúster de nodos:**
  - El nodo coordinador envía una petición de escritura a todas las réplicas que contengan la fila a ser escrita.
    - Los datos se escriben en todos los nodos disponibles que reciben la petición.
    - El nivel definido de consistencia de la escritura determina cuántos nodos han de responder a la petición para determinar que la escritura se ha realizado con éxito.
    - El éxito de la escritura significa que el dato ha sido escrito en el *commit log* y el *MemTable*.
    - Todos los nodos pueden ser coordinadores en una petición.
      - La selección del nodo coordinador en cada petición se realiza en base a una política pre-definida en la Base de Datos.



Arquitectura: peticiones de borrado de datos

- **Eliminando datos: *tombstone***
  - Cuando una fila es eliminada, no se borra del disco (*SSTables* inmutables), sino que se marca como eliminada (***tombstone***).
  - Durante la compactación, las filas marcadas son eliminadas definitivamente del disco.
  - **Tiempo de gracia**: si un nodo que contiene la fila a ser eliminada se encuentra caído en ese momento, aún tiene la posibilidad de recibir la orden de eliminación si se recupera en un intervalo de tiempo predefinido.
    - Si no se recupera en ese tiempo, la fila no será marcada como eliminada, por lo que pueden existir inconsistencias en las réplicas. Por ello, se recomienda que los administradores realicen cada cierto tiempo tareas de mantenimiento en Cassandra.
  - ***LLTS***: los datos pueden tener una "**fecha de caducidad**", a partir de la cual son marcados por Cassandra como eliminados.

Aquitectura: peticiones de lectura de datos

- En cada nodo que recibe la lectura, se accede mediante *key* a la *MemTable* para recuperar los datos requeridos.
  - Si los datos (o parte de ellos) no se encuentran en la *MemTable*, se accede a las *SSTable*.
  - **Problema de lentitud**: si no se ha realizado la tarea de compactación recientemente, es probable que haya que leer en varias *SSTables* para recuperar todas las filas requeridas.

Arquitectura: peticiones de lectura de datos

- En Cassandra, existen 3 tipos diferentes de lectura.
  - ***Direct read request***: el nodo coordinador contacta con un nodo que contenga la réplica requerida.
  - ***Digest request***: contacta con tantos nodos con réplicas como se determine en el nivel de consistencia. Se comprueba la consistencia de los datos retornados por el nodo contactado en el *Direct read request*.
    - Se envía la consulta aquellos nodos que estén respondiendo "más rápido".
    - Si existen inconsistencias, se consideran como válidos las réplicas con un *timestamp* más reciente. Estos son los retornados al cliente.
    - Una vez contactados los nodos determinados por el *consistency level*, se envía un *Digest request* al resto de réplicas, para determinar si son también consistentes.
  - ***Background read repair request***: en caso de existir inconsistencias en las réplicas, las réplicas con un *timestamp* más antiguo son sobrescritas con los datos de la réplica "más actual".

Tareas de administración

- **Seguridad**
  - Seguridad a nivel de usuarios: *logins* con *passwords* y permisos de gestión y administración vía *GRANT/REVOKE*.
  - Opciones de encriptación tanto entre clientes y clústeres como entre nodos.
  - *DataStax Enterprise* ofrece opciones avanzadas de seguridad, como autentificación externa, encriptación de tablas, y auditoría de datos.
- **Copias de respaldo**
  - Ofrece varias opciones de *backup*. Recomendado hacerlos de forma regular ante errores como los borrados accidentales.
  - *DataStax OpsCenter* provee de una interfaz de visualización para el mantenimiento de *backups* y para su restauración (https://www.datastax.com/products/datastax-opscenter)

Tareas de administración

- **Asegurar la consistencia de los datos.**
  - Como existe el riesgo de encontrar datos inconsistentes en los nodos, a pesar de los mecanismos suministrados por Cassandra (p.e. *tombstones*), se recomienda realizar tareas rutinarias de mantenimiento.
    - Un nodo caído recibe una petición de borrado que no puede ejecutar. El nodo se recupera tiempo después de que el "tiempo de gracia" de esa orden haya caducado, por lo que tiene datos inconsistentes (que deberían ser borrados). No se realiza ninguna consulta sobre esos datos, por lo que no hay posibilidad de recibir un *Read Repair Request*.
    - Cassandra ofrece una operación, llamada *repair*, que puede utilizarse por parte del usuario para asegurar que todos los nodos son consistentes (no hay réplicas con diferentes valores).

Modelo de datos

- **Clúster**: las máquinas (nodos) de una instancia de Cassandra. Los clústers pueden contener múltiples *keyspaces*.
- ***Keyspace***: es la agrupación de *ColumnFamilies*. Normalmente hay un *keyspace* por aplicación.
- **Familia de columnas (*ColumnFamilies*)**: contenedor de múltiples columnas (*columns*). Es equivalente a la tabla en el modelo relacional. Cada entrada (*row*) se identifica y accede a ella mediante una row-KEY.
- **Columnas (*Columns*)**: unidad básica de almacenamiento. Consistente en una estructura de tres valores: *name*, *value* y *timestamp*.
- ***SuperColumns***: se consideran *columns* que almacenan *subcolumns*. Son opcionales y en la versión actual se desaconseja su uso.
- **Filas o entradas (*Rows*):** conjunto de columnas de una familia con valor asignado.

Modelo de datos: Columnas

M.Zorrilla - D.García

- **Columnas (*columns*)**
  - Unidad básica de almacenamiento interno. Es una tripleta.

Estructura de la columna:

```
struct Column {
        1: binary   name,
        2: binary   value,
        3: i64  timestamp
}
```

1. *Name*: nombre de la columna
2. *Value*: valor almacenado en la columna
3. *Timestamp*: fecha y hora de escritura de la columna

Modelo de datos: Columnas

- **Columnas (*columns*):**
  - Unidad básica de almacenamiento interno. Es una tripleta.

Estructura de la columna:

```
struct Column {
        1: binary   name,
        2: binary   value,
        3: i64  timestamp
}
```

Ejemplo de columna:

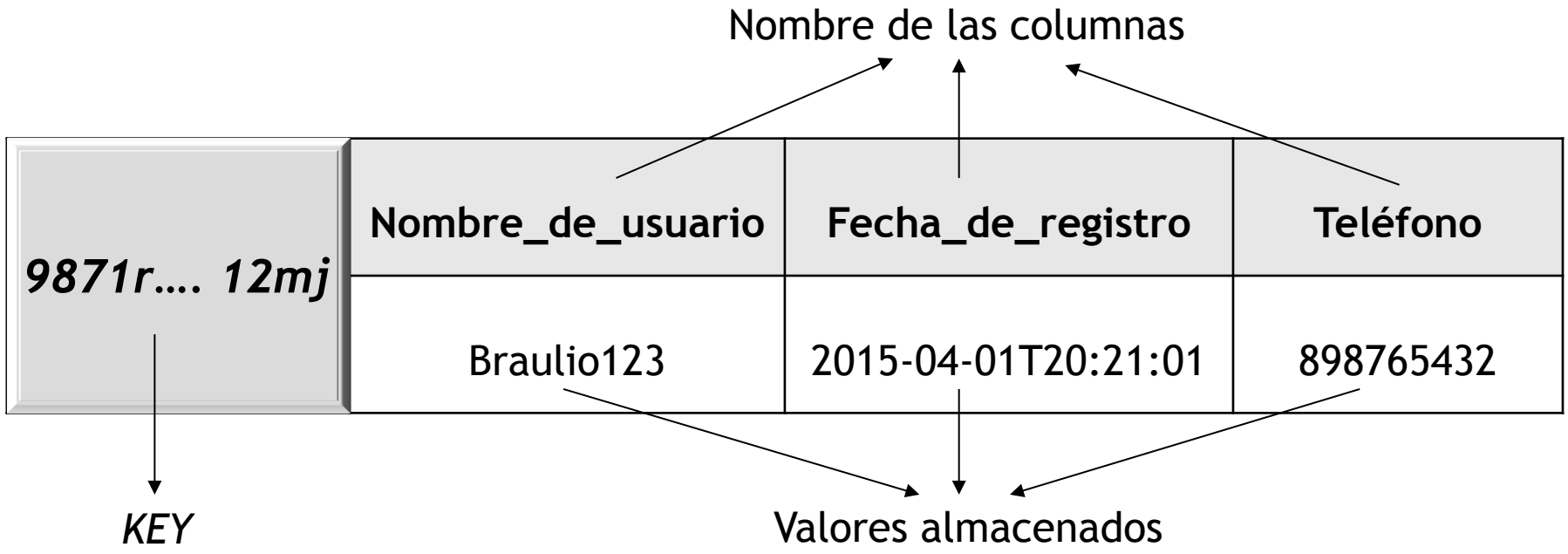
```
{
  "name": "Nombre_de_usuario",
  "value": "Braulio123",
  "timestamp":  987654321
}
```

Modelo de datos: Columnas

- **Columnas (*columns*):**
  - Los tres valores anteriores son proporcionados por el cliente de alto nivel (aplicación), incluido el *timestamp*. Por ello, es necesario tener sincronizado el reloj del sistema con el reloj del clúster.
  - El *timestamp* es utilizado para la resolución de conflictos (*Eventual Consistency*).
  - El *timestamp* puede ser de cualquier tipo pero la convención marca que sea el valor en microsegundos desde 1970.

Modelo de datos: Filas

- **Filas(rows):**
  - Agregación de columnas con un nombre para referenciarlo (*row-KEY*).
  - Equivalente a la fila en el modelo relacional.



Modelo de datos: Filas

- ***Row-KEY*** o **clave primaria** de las filas:
  - La *row-KEY* es el equivalente a la clave primaria del modelo relacional.
  - Tienen dos partes (claves):
    - **De particionado (*partition key*):** las filas con el mismo valor de clave de particionado se almacenan en la misma partición del disco (físicamente juntas).
    - **De agrupamiento (*clustering key*):** determina el orden físico en el que se almacenan las filas.

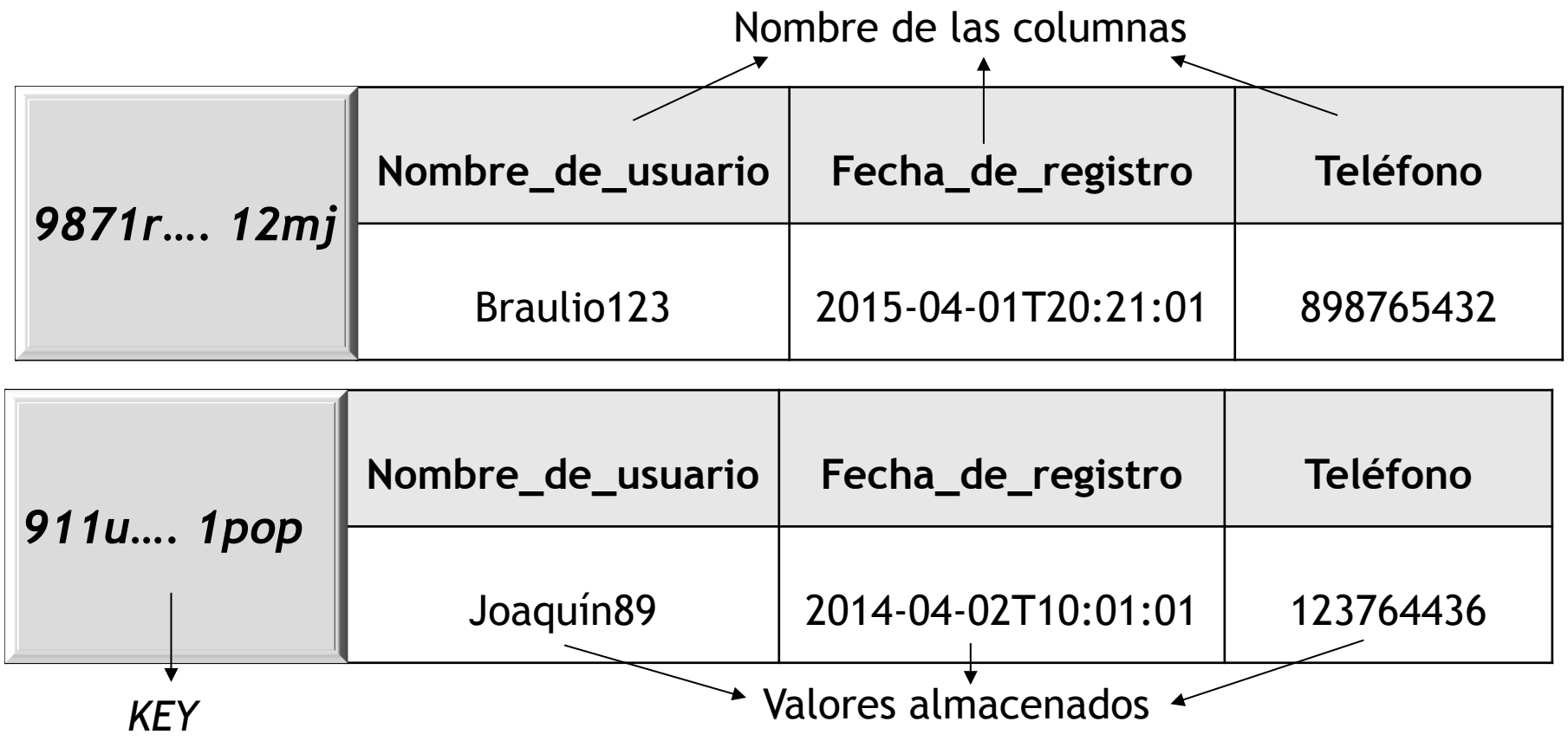
Modelo de datos: Familia de columnas

- **Familia de Columnas (*ColumnFamily*)**
  - Agregación de filas.
  - Cada *ColumnFamily* es guardada en un fichero separado y es ordenado por la *KEY*.
  - Equivalente a las tablas en el modelo relacional.
  - Tipos de columnas:
    - *Standard Columns*
    - *Counter Columns*
    - *Collection Columns: SET, LIST, MAP*
    - *User-defined type*
    - *Tuple-type*
    - *Timestamp type*

Ver más en http://docs.datastax.com/en/cql/3.3/cql/cql_reference/cql_data_types_c.html

Modelo de datos: Familia de columnas

- **Ejemplo de *ColumnFamily*:**



Modelo de datos: *Keyspace*

- Es equiparable al *schema* del modelo relacional. Generalmente uno por aplicación.

- Los atributos básicos de un *keyspace* son:

  - ***Replication factor***: cuánto quieres pagar en rendimiento a favor de consistencia.

  - ***Replica placement strategy***: indica cómo se colocan las réplicas en el anillo:
    - *SimpleStrategy* (solo un centro de datos)
    - *NetworkTopologyStrategy* (varios centros de datos)

Modelo de datos: *Keyspace*

- ***Keyspace***
  - Cassandra ofrece soporte para particionado distribuido de datos
    - *RandomPartitioner*, da buen balanceo de carga
    - *OrderPreservingPartitionioner*, permite ejecutar consultas de rangos, pero exige más trabajo eligiendo *node tokens*
      - Más info en: http://ria101.wordpress.com/2010/02/22/cassandra-randompartitioner-vs-orderpreservingpartitioner/
  - Cassandra tiene consistencia reconfigurable.
    - Al escribir, *consistency level* determina en cuántas réplicas se debe escribir para confirmar a la aplicación cliente.
    - Al leer, se especifica cuántas réplicas deben responder para retornar los datos a la aplicación cliente.

http://www.datastax.com/docs/0.8/dml/data_consistency

Modelo de datos: Cluster

- **Clúster**
  - Los datos en Cassandra se guardan en un Clúster o *Ring* donde se asignan datos a los nodos dentro de un anillo.
  - Un nodo tiene réplicas para diferentes rangos de datos.
  - Si un nodo se cae, su réplica puede responder.
  - Un protocolo *P2P* hace que los datos se repliquen entre nodos acorde a un factor de replicación definido.

Modelo de datos: Índices

- **Índices primarios**
  - Los índices primarios son únicos por cada fila.
  - Las filas son asignadas a cada nodo por el *partitioner* y el *replica placement strategy* en base a su índice primario.
  - Dado que cada nodo conoce qué rango de valores del índice primario almacena, las consultas pueden realizarse de forma eficiente escaneando únicamente los índices de las réplicas de las filas buscadas.
- **Índices secundarios**
  - Se permiten en Cassandra.
  - Son índices sobre columnas.
  - Se implementan como una tabla oculta, separada de la tabla que contiene los valores originales.
    - Por ello, no se recomienda su uso para consultas que requieran comprobar un gran volumen de datos para finalmente retornar un pequeño conjunto de resultados

Lenguaje de consulta y manipulación de datos: CQL 3.x

- CQL3 provee un fina capa de abstracción sobre la estructura interna (CF) con objeto de que *rows* y *columns* signifiquen lo mismo que en SQL
  - Explicación CQL2 vs CQL3 http://www.datastax.com/dev/blog/thrift-to-cql3
- Internamente, CQL3 y *thrift* usan el mismo motor de almacenamiento así que las mejoras se tendrán en ambos.
- No compatible con CQL 2.0 y difiere mucho de él http://cassandra.apache.org/doc/cql3/CQL.html
- Se sugiere trabajar con CQL3 por ser más amigable

http://www.datastax.com/documentation/cql/3.1/cql/cql_intro_c.html

CQL - *Keyspaces*

- Sintaxis para la creación de *Keyspaces*:

```
CREATE KEYSPACE | SCHEMA IF NOT EXISTS keyspace_name
WITH REPLICATION = map AND DURABLE_WRITES = true | false
```

- *Extraído de* *http://docs.datastax.com/en/cql/3.1/cql/cql_reference/create_keyspace_r.html*

- Ejemplo de creación de un *Keyspace*:

```
CREATE KEYSPACE BD_tienda_online
WITH REPLICATION = { 'class' : 'SimpleStrategy',
'replication_factor' : 3 };
```

CQL - *Keyspaces*

M.Zorrilla - D.García

- Ejemplo de creación de un *Keyspace*:

Nombre del *Keyspace*

```
CREATE KEYSPACE BD_tienda_online

WITH REPLICATION = { 'class' : 'SimpleStrategy',

'replication_factor' : 3 };
```

Número de réplicas de los datos en múltiples nodos. Sólo necesario si la clase (*class*) está definida como *SimpleStrategy*.

Define la estrategia de replicación de datos. Dos opciones:

- *SimpleStrategy*: cuando los datos se almacenan en un solo *data center*.
- *NetworkTopologyStrategy*: si se planea desplegar el clúster en múltiples *data centers*.

CQL - Tablas

- En CQL 3.x, las familias de columnas se crean con una sintaxis de creación de tablas (*CREATE TABLE*) similar a la que provee SQL.
- Sintaxis para la creación de familias de columnas (tablas):

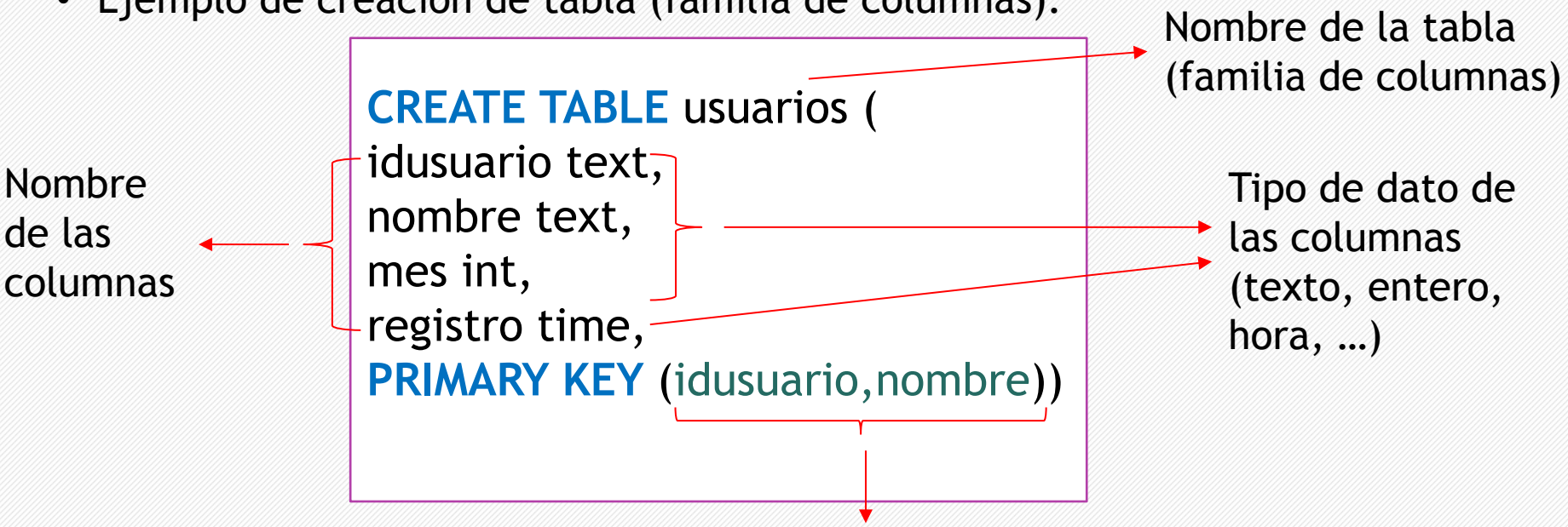
```
CREATE TABLE IF NOT EXISTS keyspace_name.table_name
( column_definition, column_definition, ...)
WITH property AND property ...

column_definition:
column_name cql_type STATIC PRIMARY KEY | column_name
<tuple<tuple_type> tuple<tuple_type>... > PRIMARY KEY | column_name
frozen<user-defined_type> PRIMARY KEY | column_name
frozen<collection_name><collection_type>... PRIMARY KEY | PRIMARY KEY
( partition_key )
```

Extraído de
http://docs.datastax.com/en/cql/3.1/cql/cql_reference/create_table_r.html

CQL - Tablas

M.Zorrilla - D.García

- Ejemplo de creación de tabla (familia de columnas):



Campos que forman parte de la *KEY* de la familia de columnas:

- La primera parte de la *KEY* define la *partition key* o clave de partición. En el ejemplo, “idusuario”.
- La segunda parte de la *KEY* define la *clustering key* o clave de ordenamiento. En el ejemplo, “nombre”.

CQL – Tablas: *Primary Key*

- La *Primary key* contiene las *partition key* y *clustering key* de la tabla:
  - ¡El orden importa!.
    - El primer valor de la *primary key* se toma como *partition key*.
    - El segundo valor, como clustering key.

PRIMARY KEY (mes, idusuario)

Partition key *Clustering key*

  - Tanto la *partition* como la *clustering key* pueden ser compuestas:

PRIMARY KEY ((idusuario,mes),nombre,registro)

Partition key *Clustering key*

CQL – Tablas: *Primary Key*

M.Zorrilla – D.García


```
CREATE TABLE usuarios (
idusuario text,
nombre text,
mes int,
registro time,
PRIMARY KEY (mes,nombre))
```

Partition key *Clustering key*

- Las réplicas de las filas de una partición se encuentran físicamente juntas.
- Las diferentes particiones de las filas de una familia de columnas (tabla) pueden residir físicamente en nodos diferentes

Partition 1 ->

Mes	Nombre	Idusuario	Registro
5	Albert	ap67	21-05-2015
5	John	uj89	02-05-2013

Partition 2 ->

Mes	Nombre	Idusuario	Registro
11	John	jk98	09-11-2013
11	Peter	pk34	08-11-2013

CQL – Tablas: *Primary Key*


```
CREATE TABLE usuarios (
idusuario text,
nombre text,
mes int,
registro time,
PRIMARY KEY (idusuario))
```

Partition key

- En este caso, al ser el idusuario único, todas las particiones de la tabla contienen una sola fila.
- No hay *clustering key*.

Partition 1 ->	**5**	**Albert**	**ap67**	**21-05-2015**
Partition 2 ->	**5**	**John**	**uj89**	**02-05-2013**
Partition 3 ->	**11**	**John**	**jk98**	**09-11-2013**
Partition 4 ->	**11**	**Peter**	**pk34**	**08-11-2013**

CQL – Tablas: Tipos de dato (desde la version 2.2)

M.Zorrilla – D.García

- Numéricos:
  - *Int:* enteros de hasta 32 bits.
  - *Bigint:* enteros de hasta 64 bits.
  - *Smallint:* enteros de hasta 2 bytes
  - *Tinyint:* enteros de hasta 1 byte
  - *Varint:* enteros de precisión arbitraria.
  - *Decimal:* decimales de precisión variable.
  - *Float:* coma-flotante de 32 bits
  - *Double:* coma-flotante de 64 bits.
  - *Counter:* contador de enteros de hasta 64 bits.
- Texto:
  - *Ascii: strings* US-ASCII
  - *Text: strings* UTF-8
  - *Varchar: strings* UTF-8 de longitud variable
  - *Inet: strings* que almacenan direcciones IPv4 o IPv6

CQL – Tablas: Tipos de datos

- De identificación:
  - *UUID:* tipo de dato que sigue el estándar UUID. Es único para cada fila insertada en una tabla. Sirve para identificar a las filas. Ver más en: http://docs.datastax.com/en/archived/cql/3.0/cql/cql_reference/uuid_type_r.html
  - *Timeuuid:* UUID que además sirve como *timestamp* único para cada fila.
- De fecha:
  - *Date:* fechas desde el 1 de Enero de 1970 en formato yyyy-mm-dd, representada como *string*.
  - *Time:* hora, minutos y segundos en formato hh:mm:ss.sss, representado como *string*.
  - *Timestamp:* fecha y hora con precisión al milisegundo en formato yyy-mm-dd hh:mm:ss.sss. Puede ser representada como *string*.

CQL – Tablas: Tipos de datos

- De colección:
  - *List:* colección de uno o más elementos ordenados.
  - *Map:* colección con pares clave-valor.
  - *Set:* colección de uno o más elementos.
- Otros tipos:
  - *Blob:* almacena bytes expresados en hexadecimal.

En el siguiente enlace, se pueden consultar los diferentes tipos de datos mencionados:

https://docs.datastax.com/en/cql/3.3/cql/cql_reference/cql_data_types_c.html

CQL - Consultas

M.Zorrilla - D.García

- En CQL 3.x, las consultas para retornar datos se realizan mediante el comando *SELECT*.
- Sintaxis de *SELECT*:

```
SELECT select_expression FROM keyspace_name.table_name
WHERE relation AND relation ...
ORDER BY ( clustering_column ASC | DESC ...)
LIMIT n
ALLOW FILTERING
```

Extraído de http://docs.datastax.com/en/cql/3.1/cql/cql_reference/select_r.html

- Ejemplos de *SELECT*:

```
SELECT * FROM usuarios

SELECT idusuario FROM usuarios WHERE mes>9 ORDER BY nombre
```

CQL - Consultas

Proyección. Un * indica que se devuelven todas las columnas.

Tabla (familia de columnas) consultada. Solo se puede consultar una tabla (no hay *JOINs*).

```
SELECT idusuario     FROM usuarios

    WHERE idusuario='1p67'     ORDER BY nombre
```

Criterios de búsqueda a cumplir. En el *WHERE*, sólo se pueden utilizar columnas que formen parte de la *particion key* o sobre las que se haya definido un índice secundario. Si la *partition key* es compuesta y se incluye en el *WHERE*, han de incluirse todos sus campos.

Criterios de ordenamiento. Siempre han de ser sobre columnas que estén incluidas en la *clustering key*.

CQL - Consultas

M.Zorrilla - D.García

Dada la siguiente definición de tabla:

```
CREATE TABLE usuarios (
idusuario text,
nombre text,
mes int,
registro time,
PRIMARY KEY ((idusuario, mes),nombre))
```

Partition key — Clustering key

La siguiente consulta devolvería un error, ya que no se han incluido en el *WHERE* todas las columnas de la *partition key*:

```
SELECT idusuario FROM usuarios WHERE mes=9
```

La siguiente consulta sería correcta, ya que incluye todos los campos de la *partition key* en el *WHERE*:

```
SELECT idusuario FROM usuarios WHERE mes=9 AND
idusuario='agp88'
```

CQL - Consultas

M.Zorrilla - D.García

Dada la siguiente definición de tabla:

```
CREATE TABLE usuarios (
idusuario text,
nombre text,
mes int,
registro time,
PRIMARY KEY (mes,nombre))
```

Partition key Clustering key

La siguiente consulta devolvería un error, ya que por defecto no se pueden hacer búsquedas de rangos sobre la *particion key* (no se puede usar el < o >):

```
SELECT idusuario FROM usuarios WHERE mes>9
```

Solamente se puede utilizar para comparar por la *partition key*, el símbolo de igualdad (=):

```
SELECT idusuario FROM usuarios WHERE mes=9
```

CQL - Consultas

- ***ORDER BY***
  - Sólo puede utilizarse sobre la *clustering key*.
  - Indispensable que las *partition key* aparezcan en la claúsula *where*.
  - Sólo tiene sentido para ordenar particiones.

- ***LIMIT***
  - Número máximo de filas a retornar.

- ***ALLOW FILTERING***
  - Puede utilizarse para realizar consultas en las que en el *WHERE* no haya condiciones en la *partition key*.
  - No recomendado a menos que sea estrictamente necesario: acceso secuencial a los datos

CQL - Consultas

M.Zorrilla - D.García

Dada la siguiente definición de tabla:

```
CREATE TABLE usuarios (
idusuario text,
nombre text,
mes int,
registro time,
PRIMARY KEY ((idusuario, registro),nombre,mes))
```

Partition key *Clustering key*

Si se utiliza el comando *ALLOW FILTERING*, se pueden buscar por rangos en el *WHERE* e incluir otras columnas que no formen parte de la *partition key*, como la *clustering key*:

```
SELECT idusuario FROM usuarios WHERE mes>9 ALLOW FILTERING
```

Estas consultas son **tremendamente ineficientes**, ya que al no buscar por la *partition key*, no es posible determinar en que nodos están almacenados los datos, por lo que la consulta ha de acceder a cada nodo del clúster en busca de los datos.

CQL - Insertados

- En CQL 3.x, para insertar datos nuevos en tablas se utiliza la cláusula INSERT:
- Sintaxis de *INSERT*:

```
INSERT INTO keyspace_name.table_name ( identifier, column_name...)
VALUES ( value, value ... ) IF NOT EXISTS USING option AND option
```

Extraído de http://docs.datastax.com/en/cql/3.1/cql/cql_reference/insert_r.html

- Ejemplo de *INSERT*:

```
INSERT INTO usuarios (idusuario, nombre, mes, nombre)
                              VALUES ('rr66', 'Ruth', 7, '2017-07-
01')
```

CQL – Insertados: UPSERT

- ¡OJO!. Cassandra, por defecto, no comprueba la unicidad de la *Primary Key*. Si existe otro dato con el mismo valor que el que se está insertando, el dato antiguo es sobrescrito (***UPSERT***):

- Ejemplo: el siguiente insertado sobrescribiría la fila del usuario llamado 'Miguel', ya que existe tienen la misma *Primary Key*:

```
CREATE TABLE usuarios (
idusuario text,
nombre text,
mes int,
registro time,
PRIMARY KEY (idusuario)
```

```
INSERT INTO usuarios (idusuario, nombre, mes, nombre)
                VALUES (mg77, 'Manuel', 2, '2018-02-02')
```

mg77	**Manuel**	**2**	**02/02/2018**

Partition 1 ->

apk87	**Ana**	**10**	**21/10/2017**

Partition 2 ->

mg77	**Miguel**	**7**	**07/07/2016**

Partition 3 ->

ebu00	**Esteban**	**1**	**20/01/2016**

Sobrescritura (*UPSERT*)

CQL – Insertados: Algunas opciones

- *USING TTL*: se borra la fila al pasar un tiempo determinado:

```
INSERT INTO usuarios (idusuario, nombre, mes, nombre)
                VALUES (mg77, 'Manuel', 2, '2018-02-02') USING TTL 87900
```

Al pasar 87900 segundos desde el insertado, la fila es borrada.

- *USING TIMESTAMP*: especifica los microsegundos de las columnas del tipo *TIMESTAMP*, si estos no son introducidos en el *INSERT*. Si no se usa, se utiliza el tiempo (en microsegundos) del momento en el que la escritura es ejecutada:

```
INSERT INTO usuarios (idusuario, nombre, mes, nombre)
        VALUES (mg77, 'Manuel', 2, '2018-02-02') USING TIMESTAMP
123456789
```

- *IF NOT EXISTS*: impide que la fila se inserte si ya existe otra fila con la misma *Primary Key* (evita *UPSERTS*). Disminuye el rendimiento del *INSERT*, ya que requiere de una comprobación.

CQL - Actualizados

- En CQL 3.x, para actualizar datos en tablas se utiliza la cláusula *UPDATE*:
- Sintaxis de *UPDATE*:

```
UPDATE keyspace_name.table_name USING option AND option SET
assignment, assignment, ... WHERE row_specification IF column_name =
literal AND column_name = literal . . . IF EXISTS
```

Extraído de http://docs.datastax.com/en/cql/3.1/cql/cql_reference/update_r.html

- Ejemplo de *UPDATE*:

Indispensable que en el *WHERE* aparezca la *Partition Key* al completo. Se permite utilizar la cláusula *IN* en el último campo de la *Partition Key*.

```
UPDATE usuarios SET nombre='Juan' WHERE idusuario = 1;
```

CQL - Borrados

M.Zorrilla - D.García

- En CQL 3.x, para borrar columnas o filas completas en tablas se utiliza la cláusula *DELETE*:
- Sintaxis de *DELETE*:

```
DELETE [ column_name [ , column_name ] [ … ] | column_name [ term ] ]
FROM [ keyspace_name. ] table_name [ USING TIMESTAMP timestamp_value
] WHERE row_specification [ IF [ EXISTS | condition [ AND condition ]
[ … ] ] ]
```

Extraído de http://docs.datastax.com/en/cql/3.1/cql/cql_reference/delete_r.html

- Ejemplos de *DELETE*:

Borra el contenido de la columna “nombre” de la fila con “idusuario=1”

```
DELETE nombre FROM usuarios WHERE idusuario = 1;
```

Borra la fila con “idusuario=1”

```
DELETE FROM usuarios WHERE idusuario = 1;
```

CQL - Índices

- En CQL 3.x, para crear índices secundarios sobre columnas o en tablas se utiliza la cláusula *CREATE INDEX*:
- Sintaxis de *CREATE INDEX*:

```
CREATE CUSTOM INDEX IF NOT EXISTS index_name ON
keyspace_name.table_name ( KEYS ( column_name ) ) USING class_name
WITH OPTIONS = map
```

Extraído de http://docs.datastax.com/en/cql/3.1/cql/cql_reference/create_index_r.html

- Ejemplo de creación de índice:

```
CREATE INDEX idx_nombreUsuario ON usuarios(nombreUsuario);
```

CQL – Otros elementos que pueden incorporarse a la BD

- Crear disparadores:

```
CREATE TRIGGER IF NOT EXISTS trigger_name ON table_name USING
'java_class'
```

Extraído de
http://docs.datastax.com/en/cql/3.1/cql/cql_reference/trigger_r.html

- Crear usuarios:

```
CREATE USER IF NOT EXISTS user_name WITH PASSWORD 'password'
NOSUPERUSER | SUPERUSER
```

Extraído de
http://docs.datastax.com/en/cql/3.1/cql/cql_reference/create_user_r.html

- Crear tipos definidos por el usuario:

```
CREATE TYPE IF NOT EXISTS keyspace.type_name ( field, field, ...)
```

Extraído de
http://docs.datastax.com/en/cql/3.1/cql/cql_reference/cqlRefcreateType.html

CQL – Borrado de elementos

- Para borrar cualquier elemento de la base de datos (tabla, índice, usuario…) se utiliza la cláusula *DROP*. Ejemplos:

```
DROP TABLE usuario;

DROP INDEX indiceNombreUsuario;

DROP TRIGGER disparador1X;

DROP TYPE tipo2Y;

DROP KEYSPACE BD_tienda_online;

…
```

CQL – Modificación de elementos

- Para modificar tablas o *keyspaces* se utiliza la cláusula *ALTER*.
- Modificación de *keyspace*:

```
ALTER KEYSPACE BD_tienda_online
WITH REPLICATION = { 'class' : 'SimpleStrategy',
'replication_factor' : 2 };
```

- Modificación de tablas:

```
ALTER TABLE usuarios ADD apellido1 int;
```
→ Añade una nueva columna

```
ALTER TABLE usuarios ALTER appelido1 TYPE text;
```
→ Cambia el tipo de la columna

```
ALTER TABLE usuarios DROP appelido1;
```
→ Elimina una columna

Más sobre ALTER TABLE en:
http://docs.datastax.com/en/cql/3.1/cql/cql_reference/alter_table_r.html

- El *ALTER* también se puede utilizar para modificar usuarios o tipos definidos.

CQL – Modelado de los datos

- Objetivos:
  - Regla 1: Distribuir los datos por todo el clúster
    - Es deseable que cada nodo del clúster tenga un volumen de datos similar (equilibrio).
    - Como las filas se distribuyen en base a la *partition key* es conveniente escoger una clave primaria adecuada para la aplicación que se trate.
  - Regla 2: Minimizar el número de particiones a leer
    - Las particiones son grupos de filas que comparten la misma *partition key*.
    - Cuantas menos particiones tengan que ser leídas, más rápida será la lectura.

Más información en: http://www.datastax.com/dev/blog/basic-rules-of-cassandra-data-modeling

CQL – Modelado de los datos

- Comenzar por conocer las consultas:
  - Pensar qué acciones nuestra aplicación necesita realizar y cómo va a acceder a los datos.
    - Con esto, posteriormente se diseñan las tablas que den soporte a esos patrones de acceso.
  - Una norma simple para cumplir con este requisito es crear una tabla para cada consulta.

- Desnormalizar para optimizar:
  - Cassandra no tiene soporte para *FOREIGN KEYs* ni para *JOINs*. Su rendimiento es mayor cuando las consultas solo han de acceder a una tabla para obtener los datos requeridos.

CQL – Modelado de los datos

- Planificar para escrituras concurrentes:
  - En una tabla, las filas son identificadas por su clave primaria, que es única.
  - Cassandra no fuerza la unicidad, por lo que insertar con claves primarias filas duplicadas provocará que se sobrescriban (*upsert*).

- ¿Claves naturales o subrogadas?
  - Una clave subrogada es generada con tipos únicos, como un UUID. Garantiza unicidad y evita *upserts*. No sirve si queremos varias filas en una misma partición.
  - Las claves naturales hacen que los datos sean más interpretables y elimina la necesidad de índices o desnormalizaciones adicionales. Sin embargo, a menos que el cliente garantice la unicidad, pueden producirse *upserts*.

CQL – Modelado de los datos: Ejemplo 1

M.Zorrilla – D.García

- **Ejemplo 1:** Se quiere almacenar los datos de los estudiantes de una plataforma de cursos online masivos (MOOC), sabiendo que usualmente se buscan los datos de usuario por correo electrónico o por nombre de usuario. Desde un punto de vista relacional, crearíamos una tabla con todos los datos y un identificador único, que en el caso de Cassandra puede ser de tipo UUID:

PROPUESTA 1.1:

```
CREATE TABLE estudiantes (
    idestudiante uuid,
    nombreUsuario text,
    correoElectronico text,
    nombre text,
    apellido1 text,
    …
    PRIMARY KEY(idestudiante)
)
```

Partition Key

- Al poner el "idestudiante" como *Partition Key*, de tipo UUID (y, por tanto, único para cada estudiante que se inserte), el reparto de los datos en los nodos del clúster será aleatorio y se garantizará un balanceo de carga entre los diferentes nodos;
- … sin embargo, no da solución al problema enunciado, ya que no se pueden realizar búsquedas por nombre de usuario o por correo electrónico, al no haber sido estas columnas elegidas para formar la *Partition Key*.

CQL – Modelado de los datos: Ejemplo 1

- Continuando con el **ejemplo 1** y la anterior propuesta 1.1, podríamos extenderla añadiendo dos nuevas tablas, “estudiantes_por_nombre” y “estudiantes_por_correo”, que permitiesen realizar búsquedas por los campos de nombre de usuario y correo electrónico:

PROPUESTA 1.2:

```
CREATE TABLE estudiantes (
      idestudiante uuid,
      nombreUsuario text,
      correoElectronico text,
      nombre text,
      apellido1 text,
      …
      PRIMARY KEY(idestudiante)
  )
```

```
CREATE TABLE estudiantes (
      idestudiante uuid,
      nombreUsuario text,
      PRIMARY KEY(nombreUsuario)
  )

CREATE TABLE estudiantes (
      idestudiante uuid,
      correoElectronico text,
      PRIMARY KEY(correoElectronico)
  )
```

Partition Key

- Con esta propuesta, se podrían realizar búsquedas por nombre de usuario en la tabla “estudiantes_por_nombre”, o bien por correo electrónico en la tabla “estudiantes_por_correo”, y luego realizar un *JOIN* con la tabla estudiantes utilizando el campo “idestudiante”…
- … ¡pero esto va en contra de la filosofía de Cassandra!. Recordemos: no hay implementación de *JOINS* debido a que sería una operación muy costosa que requeriría de una búsqueda secuencial a lo largo de los nodos del clúster.

CQL – Modelado de los datos: Ejemplo 1

M.Zorrilla - D.García

- Continuando con el **ejemplo 1,** podríamos solucionar el problema de la propuesta 1.2 y evitar los *JOINS* creando dos tablas redundantes, ambas almacenando todos los datos de los estudiantes, y diferenciándose en la *Partition Key*:

PROPUESTA 1.3:

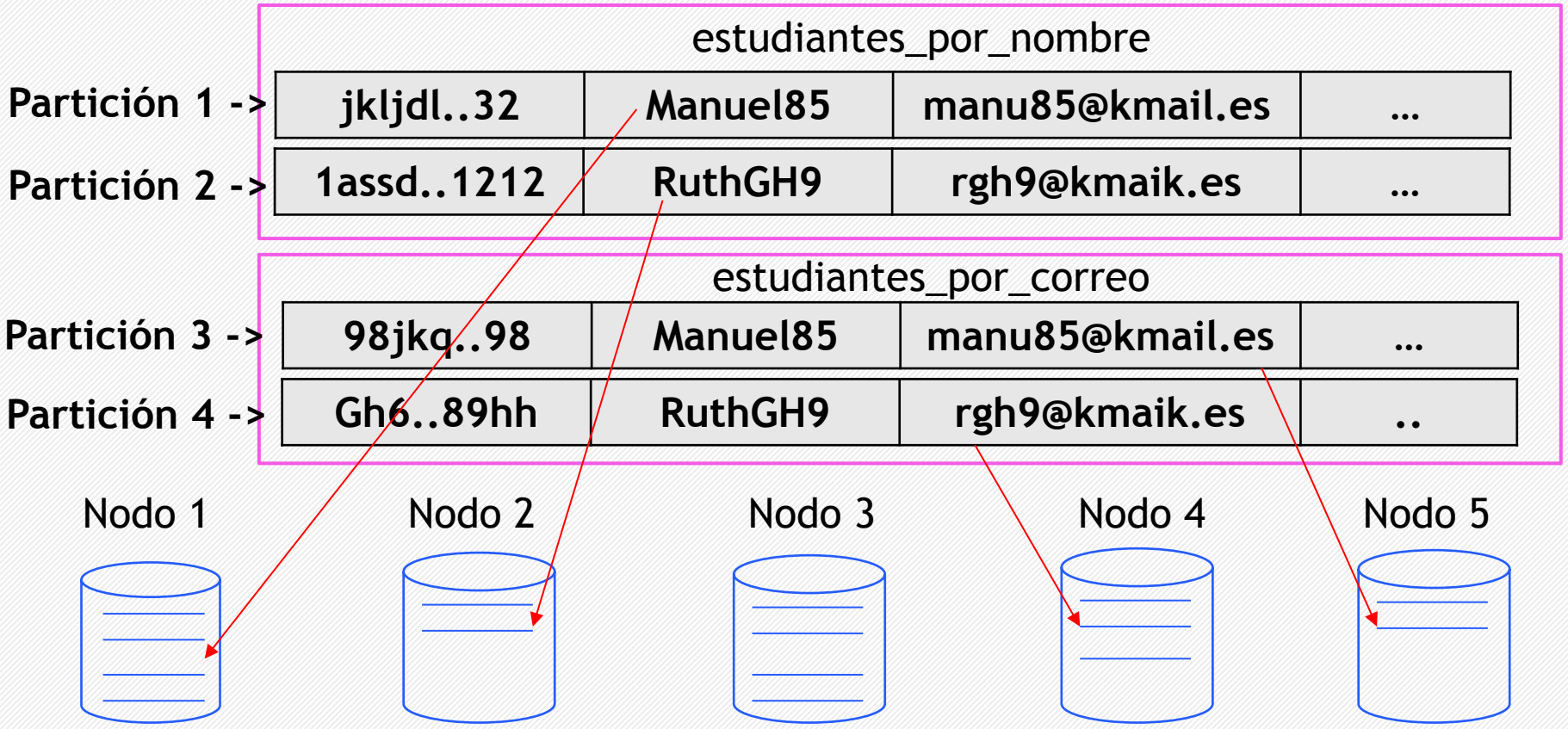
```
CREATE TABLE estudiantes_por_nombre (
      idestudiante uuid,
      nombreUsuario text,
      correoElectronico text,
      nombre text,
      apellido1 text,
      …
      PRIMARY KEY(nombreUsuario)
  )
```

```
CREATE TABLE estudiantes_por_correo (
      idestudiante uuid,
      nombreUsuario text,
      correoElectronico text,
      nombre text,
      apellido1 text,
      …
      PRIMARY KEY(correoElectronico)
  )
```

- Con esta propuesta, se podrían realizar búsquedas por nombre de usuario en la tabla "estudiantes_por_nombre", o bien por correo electrónico en la tabla "estudiantes_por_correo", obteniendo de cada una de ellas todos los datos de los estudiantes.
- Se almacenan los datos de cada estudiante dos veces, por lo que hay redundancia en la Base de datos… ¡pero en eso consiste modelar con Cassandra!. Se admite y se recomienda la redundancia en beneficio del rendimiento en las consultas.

CQL – Modelado de los datos: Ejemplo 1

- Con la propuesta 1.3 del ejemplo 1, además, los datos de los estudiantes se repartirían de forma aleatoria en los diferentes nodos del clúster, en base al valor de la *Partition Key* de cada tabla. Se asume que tanto el correo electrónico como el nombre de usuario son únicos para cada estudiante:



CQL – Modelado de los datos: Ejemplo 2

M.Zorrilla – D.García

- **Ejemplo 2:** supongamos que ahora, además, queremos asignar grupos de tutorías a los diferentes estudiantes, sabiendo que se realizarán sobre la base de datos consultas frecuentes acerca de los datos de los estudiantes en base al grupo al que pertenecen.

  En el modelo relacional, lo solucionaríamos implementando tres tablas: una para los datos de los estudiantes, y otra para los datos de los grupos, y otra más para almacenar la relación de pertenencia de los estudiantes a estos grupos:

PROPUESTA 2.1:

```
CREATE TABLE estudiantes (
    idestudiante uuid,
    nombreUsuario text,
    correoElectronico text,
    nombre text,
    apellido1 text,
    …
    PRIMARY KEY(idestudiante)
)
```

```
CREATE TABLE grupos(
      idgrupo uuid,
      nombreGrupo text,
      …
      PRIMARY KEY(idgrupo)
  )

CREATE TABLE gruposEstudiantes(
      idgrupo uuid,
      idestudiante uuid,
      …
  )
```

- Esto requeriría de *JOINS* para extraer los datos de los estudiantes junto con los de los grupos a los que pertenecen. Por tanto, no es una solución aceptable.

CQL – Modelado de los datos: Ejemplo 2

M.Zorrilla – D.García

- Siguiendo con el **ejemplo 2**, una posible solución pasaría por crear una tabla llamada "estudiantes_en_grupos" que almacenase juntos los datos de los estudiantes y de los grupos, creando redundancia en la base de datos. Dado que se quiere buscar por el grupo, el campo "nombreGrupo" podría ser *la Partition Key*:

PROPUESTA 2.2:

```
CREATE TABLE estudiantes_en grupos(
      nombreUsuario text,
      correoElectronico text,
      nombre text,
      apellido1 text,
      …
      nombreGrupo text,
      …
      PRIMARY KEY(nombreGrupo)
  )
```

- Esta propuesta permitiría buscar por el nombre del grupo, y retornar los datos de todos los estudiantes que se encuentren en dicho grupo…
- … ¡pero no garantiza unicidad!. Al ser el nombre del grupo el único campo que pertenece a la *Primary Key*, se producirían *upserts* cada vez que se insertase un nuevo usuario en un grupo.

CQL – Modelado de los datos: Ejemplo 2

M.Zorrilla – D.García

- Para solucionar el problema anterior de u*pserts*, se debería añadir algún campo más a la *Primary Key* que garantice la unicidad y evite este problema. Podríamos, por tanto, añadir el nombre del usuario como *Clustering Key*:

PROPUESTA 2.3:

```
CREATE TABLE estudiantes_en grupos(
      nombreUsuario text,
      correoElectronico text,
      nombre text,
      apellido1 text,
      …
      nombreGrupo text,
      …
      PRIMARY KEY(nombreGrupo, nombreUsuario)
  )
```

Partition Key *Clustering Key*

- De esta forma, el nombre del usuario (único por estudiante) garantiza la unicidad y elimina el riesgo de *upserts*. Además, al ser parte de la *Clustering Key* y no de la *Partition Key*, se pueden seguir realizando consultas sobre la tabla en base únicamente al nombre del grupo, único campo que pertenece a la *Partition Key*.

CQL – Modelado de los datos: Ejemplo 2

M.Zorrilla – D.García

- Utilizando la propuesta 2.3, los estudiantes del mismo grupo se almacenarían en la misma partición.

```
CREATE TABLE estudiantes_en grupos(
      nombreUsuario text,
      correoElectronico text,
      nombre text,
      apellido1 text,
      …
      nombreGrupo text,
      …
      PRIMARY KEY(nombreGrupo, nombreUsuario)
  )
```

	nombreUsu.	…	nombreGrupo	…
Partición 1 ->	Manuel85	…	A	…
	Carol90	…	A	…
Partición 2 ->	Ruth66	…	B	…
	IvanOtl	…	B	…

CQL – Modelado de los datos: Ejemplo 2

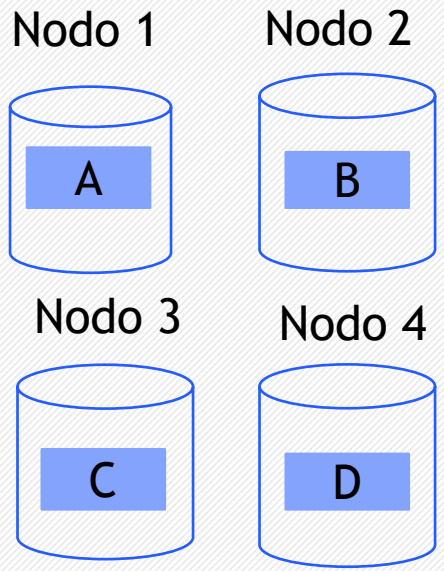
- Con la propuesta 2.3, se cumple el primer objetivo de Cassandra al respecto de optimizar consultas, ya que podemos consultar los datos de los estudiantes de un grupo concreto tan sólo accediendo a una partición.
- Sin embargo… ¿se cumple el segundo objetivo, relativo al equilibrio en el reparto de los datos a lo largo de los nodos?.

```
CREATE TABLE estudiantes_en grupos(
      nombreUsuario text,
      correoElectronico text,
      nombre text,
      apellido1 text,
      …
      nombreGrupo text,
      …
      PRIMARY KEY(nombreGrupo, nombreUsuario)
  )
```

Imaginemos que tenemos 4 nodos en el clúster, y que hay 4 grupos de estudiantes diferentes: A, B, C y D. Los datos del grupo A se almacenan en el nodo 1, los del B en el 2, los del C en el 3 y los del D en el nodo 4.

Si los grupos tienen un número similar de estudiantes, entonces el reparto será homogéneo entre los nodos, cumpliendo con el objetivo del balanceo de carga. Por ejemplo:

Grupo A: 100 estudiantes.
Grupo B: 89 estudiantes.
Grupo C: 94 estudiantes.
Grupo D: 85 estudiantes.



CQL – Modelado de los datos: Ejemplo 2

M.Zorrilla – D.García

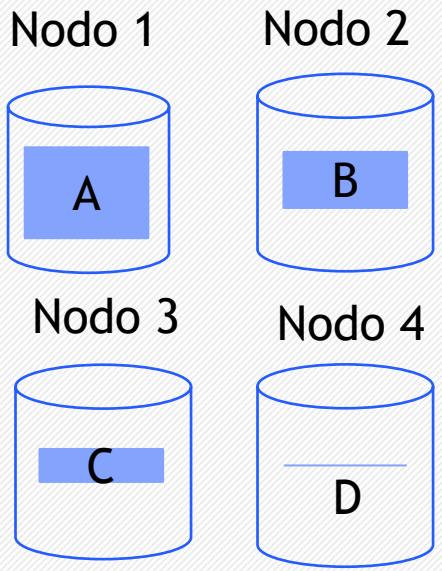
- Con la propuesta 2.3, se cumple el primer objetivo de Cassandra al respecto de optimizar consultas, ya que podemos consultar los datos de los estudiantes de un grupo concreto tan sólo accediendo a una partición.
- Sin embargo… ¿se cumple el segundo objetivo, relativo al equilibrio en el reparto de los datos a lo largo de los nodos?.

```
CREATE TABLE estudiantes_en grupos(
      nombreUsuario text,
      correoElectronico text,
      nombre text,
      apellido1 text,
      …
      nombreGrupo text,
      …
      PRIMARY KEY(nombreGrupo, nombreUsuario)
  )
```

Imaginemos que tenemos 4 nodos en el clúster, y que hay 4 grupos de estudiantes diferentes: A, B, C y D. Los datos del grupo A se almacenan en el nodo 1, los del B en el 2, los del C en el 3 y los del D en el nodo 4.

Si los grupos no tienen un número similar de estudiantes, entonces ¡el reparto no será homogéneo entre los nodos, y no se cumplirá con el objetivo del balanceo de carga! Por ejemplo:

Grupo A: 350 estudiantes.
Grupo B: 89 estudiantes.
Grupo C: 25 estudiantes.
Grupo D: 1 estudiantes.



CQL – Modelado de los datos: Ejemplo 2

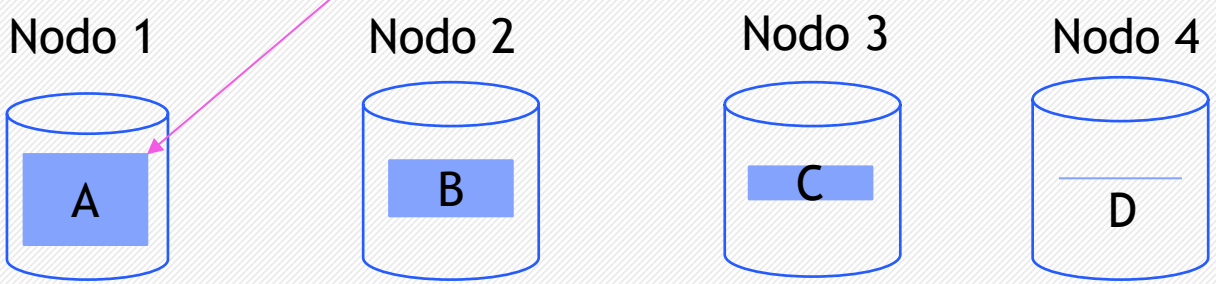
M.Zorrilla – D.García

- Con la propuesta 2.3, aunque la distribución de los datos no fuese homogénea entre los nodos, dado que los estudiantes de cada grupo se almacenan en la misma partición, solamente sería necesaria una consulta a una partición para devolver los datos de los estudiantes de un grupo concreto:

```
CREATE TABLE estudiantes_en grupos(
      nombreUsuario text,
      correoElectronico text,
      nombre text,
      apellido1 text,
      …
      nombreGrupo text,
      …
      PRIMARY KEY(nombreGrupo, nombreUsuario)
  )
```

SELECT * FROM estudiantes_en_grupos WHERE nombreGrupo = 'A'

Esta consulta devuelve los datos de todos los estudiantes del grupo A, accediendo a una sola partición y nodo.



CQL – Modelado de los datos: Ejemplo 2

M.Zorrilla – D.García

- Para solucionar el problema del reparto de los datos entre los nodos del clúster, podríamos pensar en añadir un nuevo campo a la tabla que forme parte de la *Partition Key* junto al nombre del grupo. Este campo representaría un valor hash calculado en base a la inicial del nombre de usuario de los estudiantes.

PROPUESTA 2.4:

```
CREATE TABLE estudiantes_en grupos(
      nombreUsuario text,
      correoElectronico text,
      nombre text,
      apellido1 text,
      …
      nombreGrupo text,
      prefijo_hash int,
      …
      PRIMARY KEY((nombreGrupo, prefijo_hash), nombreUsuario)
  )
```

Partition Key

Clustering Key

- Con esta propuesta, los datos de los estudiantes se repartirían en las particiones no sólo en base al grupo al que pertenecen, sino también según la inicial de su nombre de usuario. Por ejemplo, los usuarios del grupo A con iniciales de la A a la M se podrían almacenar en una partición, y los estudiantes del grupo A con iniciales de la N a la Z en otro. Esto ayudaría a repartir más los datos de estudiantes de aquellos grupos que pudiesen ser excesivamente grandes.

CQL – Modelado de los datos: Ejemplo 2

M.Zorrilla - D.García

- Utilizando la propuesta 2.4, los estudiantes del mismo grupo se almacenarían en la misma partición.

```
CREATE TABLE estudiantes_en grupos(
      nombreUsuario text,
      correoElectronico text,
      nombre text,
      apellido1 text,
      …
      nombreGrupo text,
      prefijo_hash int,
      …
      PRIMARY KEY((nombreGrupo, prefijo_hash),
                  nombreUsuario)
  )
```

	nombreUsu.	…	nombreGrupo	prefijo
Partición 1 ->	Manuel85	…	A	1
	Carol90	…	A	1
Partición 2 ->	Ruth66	…	B	2
Partición 3 ->	IvanOtl	…	B	1

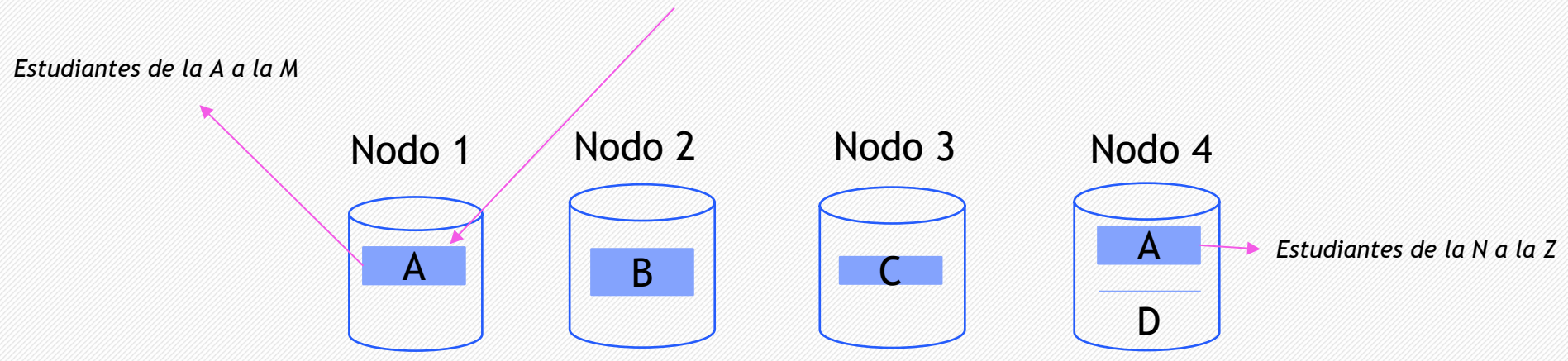
CQL – Modelado de los datos: Ejemplo 2

M.Zorrilla - D.García

- Con la propuesta 2.4, si por ejemplo el grupo A tuviese un número mucho más alto de estudiantes que el resto, sus datos podrían estar más repartidos entre los nodos al separarse los estudiantes con iniciales de la A a la M de los que tienen iniciales de la N a la Z:

```
CREATE TABLE estudiantes_en grupos(
        nombreUsuario text,
        correoElectronico text,
        nombre text,
        apellido1 text,
        …
        nombreGrupo text,
        prefijo_hash int,
        …
        PRIMARY KEY((nombreGrupo, prefijo_hash),
                        nombreUsuario)
    )
```

SELECT * FROM estudiantes_en_grupos WHERE nombreGrupo = 'A' and prefijo_hash = 1

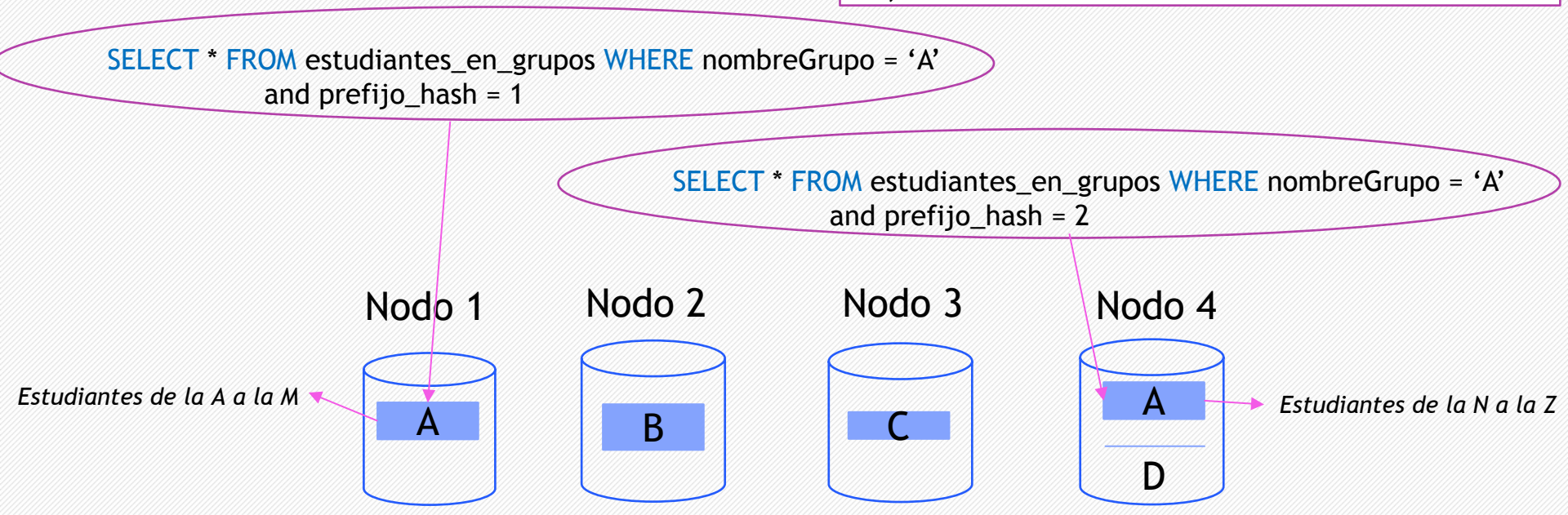


CQL – Modelado de los datos: Ejemplo 2

M.Zorrilla – D.García

- Con la propuesta 2.4, el problema es que si quisiésemos consultar los datos de todos los estudiantes del grupo A, requeriríamos varias consultas (dos en este caso), por lo que se pierde eficiencia a cambio de que los datos se repartan de forma más homogénea… ¡no se puede siempre tener ambos!.

```
CREATE TABLE estudiantes_en grupos(
      nombreUsuario text,
      correoElectronico text,
      nombre text,
      apellido1 text,
      …
      nombreGrupo text,
      prefijo_hash int,
      …
      PRIMARY KEY((nombreGrupo, prefijo_hash),
                  nombreUsuario)
  )
```



CQL – Modelado de los datos: Ejemplo 3

M.Zorrilla - D.García

- **Ejemplo 3:** partiendo de la propuesta 2.3, se quiere además obtener los datos de los estudiantes de un grupo concreto ordenados descendentemente según su fecha de asignación en el grupo. Para ello, habría que añadir un campo de tipo fecha a la tabla, y utilizarlo como *clustering key* para poder ordenar por él, tal como se muestra en el siguiente ejemplo:

PROPUESTA 3.1:

```
CREATE TABLE estudiantes_en grupos(
      nombreUsuario text,
      correoElectronico text,
      …
      nombreGrupo text,
      …
      fechaAsignacion timestamp
      PRIMARY KEY(nombreGrupo, fechaMatricula, nombreUsuario)
  )
```

- El campo “fechaAsignacion” se establece como primer campo de la *clustering key*, por lo que sería posible la ordenación por fecha de asignación al grupo.
- Los datos de tipo *timestamp* no garantizan la unicidad: por ejemplo, podrían existir dos usuarios con la misma fecha de matricula. Por ello, es necesario mantener en la *Primary Key*, como parte de la *clustering key*, al campo “nombreUsuario” para garantizar la unicidad y evitar UPSERTS… a menos que utilizásemos un tipo de dato para definir la fecha de matriculación que garantizase que dos estudiantes no pueden tener nunca el mismo valor en dicho campo.

CQL – Modelado de los datos: Ejemplo 3

M.Zorrilla – D.García

- Si definimos la fecha de asignación como *timeuuid*, podríamos obviar el campo "nombreUsuario" en la *Primary Key*, ya que este tipo de dato garantiza la unicidad:

PROPUESTA 3.2:

```
CREATE TABLE estudiantes_en grupos(
      nombreUsuario text,
      correoElectronico text,
      …
      nombreGrupo text,
      …
      fechaAsignacion timeuuid
      PRIMARY KEY(nombreGrupo, fechaAsignacion)
  )
```

- Con esta propuesta, podríamos ordenar a los estudiantes según su fecha de asignación.

CQL – Modelado de los datos: Ejemplo 3

- ¿Qué pasaría si realizásemos la siguiente consulta utilizando la propuesta 3.2?:

```
SELECT nombreUsuario, fechaAsignacion
FROM estudiantes_en_grupos WHERE
nombreGrupo = 'A'
```

- El resultado sería el siguiente:

nombreUsuario	fechaAsignacion
Manuel85	21/11/2016
Carol90	15/02/2017
DesireRJ	24/05/2017

-
```
CREATE TABLE estudiantes_en grupos(
    nombreUsuario text,
    correoElectronico text,
    …
    nombreGrupo text,
    …
    fechaAsignacion timeuuid
    PRIMARY KEY(nombreGrupo, fechaAsignacion)
)
```

Los usuarios salen ordenados por la fecha en orden ascendente, ya que por defecto en Cassandra se almacenan en este orden según el valor de la *clustering key*.

CQL – Modelado de los datos: Ejemplo 3

M.Zorrilla – D.García

- ¿Qué pasaría si realizásemos la siguiente consulta utilizando la propuesta 3.2?:

```
SELECT nombreUsuario, fechaAsignacion
FROM estudiantes_en_grupos WHERE
nombreGrupo = 'A' ORDER BY
fechaAsignacion DESC
```

```
CREATE TABLE estudiantes_en grupos(
    nombreUsuario text,
    correoElectronico text,
    …
    nombreGrupo text,
    …
    fechaAsignacion timeuuid
    PRIMARY KEY(nombreGrupo, fechaAsignacion)
)
```

- El resultado sería el siguiente:

nombreUsuario	fechaAsignacion
DesireRJ	24/05/2017
Carol90	15/02/2017
Manuel85	21/11/2016

Haciendo uso de la cláusula *ORDER BY*, e indicando explícitamente que se ordenen de forma descendente (*DESC*), conseguimos el resultado deseado… ¿existe una forma más eficiente de dar solución al problema propuesto?.

CQL – Modelado de los datos: Ejemplo 3

M.Zorrilla – D.García

- En Cassandra, podemos indicar de forma explícita al crear las tablas el tipo de ordenamiento físico según la *clustering key* que por defecto es ascendente:

PROPUESTA 3.3:

```
CREATE TABLE estudiantes_en grupos(
      nombreUsuario text,
      correoElectronico text,
      …
      nombreGrupo text,
      …
      fechaAsignacion timeuuid
      PRIMARY KEY(nombreGrupo, fechaAsignacion)
  ) WITH CLUSTERING ORDER BY (fechaAsignacion DESC)
```

- De esta forma, los datos de los estudiantes ya estarían físicamente ordenados descendentemente por la fecha.

CQL – Modelado de los datos: Ejemplo 3

M.Zorrilla – D.García

- ¿Qué pasaría si realizásemos la siguiente consulta utilizando la propuesta 3.3?:

```
SELECT nombreUsuario, fechaAsignacion
FROM estudiantes_en_grupos WHERE
nombreGrupo = 'A'
```

```
CREATE TABLE estudiantes_en grupos(
    nombreUsuario text,
    correoElectronico text,
    …
    nombreGrupo text,
    …
    fechaAsignacion timeuuid
    PRIMARY KEY(nombreGrupo, fechaAsignacion)
) WITH CLUSTERING ORDER BY (fechaAsignacion DESC)
```

- El resultado sería el siguiente:

nombreUsuario	fechaAsignacion
DesireRJ	24/05/2017
Carol90	15/02/2017
Manuel85	21/11/2016

Los usuarios salen ordenados por la fecha en orden descendente sin necesidad de utilizar el *ORDER BY*, haciendo, por tanto, más eficiente la consulta.

CQL – Modelado de los datos: Ejemplo 3

M.Zorrilla – D.García

- ¡OJO!. La *clustering key* sólo sirve para ordenar los datos de cada partición **INDIVIDUALMENTE**. Es decir, si se ejecuta un *SELECT* sin *WHERE*, retornando los datos de todas las particiones, sólo aparecerán ordenador los datos de cada partición por separado. Veámoslo con un ejemplo: ¿Qué pasaría si ejecutamos la siguiente consulta sobre la tabla "estudiantes_en grupos" de la propuesta 3.3?

```
SELECT nombreUsuario, nombreGrupo, fechaAsignacion FROM estudiantes_en_grupos
```

- El resultado sería el siguiente:

nombreUsuario	nombreGrupo	fechaAsignacion
DesireRJ	A	24/05/2017
Carol90	A	15/02/2017
Manuel85	A	21/11/2016
Ruth66	B	12/04/2017
IvanOtl	B	01/01/2017

- ¡Los estudiantes aparecen ordenados por fecha según su grupo!. En definitiva, Cassandra no permite ordenamientos que impliquen datos de varias particiones.

Ventajas de Cassandra

- Cassandra tiene notables ventajas:
  - Escalabilidad horizontal: puede añadirse nuevo hardware si aumentan los requisitos de la base de datos.
  - Respuesta rápida aunque la demanda crezca (si está bien configurado).
  - Elevadas velocidades de escritura para gestionar volúmenes de datos incrementales (siempre que la implementación de la base de datos sea adecuada).
  - Almacenamiento distribuido de los datos.
  - Tolerante a fallos.
  - Dispone de un *API* sencilla para múltiples lenguajes
  - Desde la versión 0.8 ofrece un lenguaje de acceso y manipulación de datos sencillo y basado en SQL: CQL.

Desventajas de Cassandra

- Hay algunas desventajas que un sistema de almacenamiento tan escalable ofrece en contrapartida:
  - No hay *JOINS* (a cambio de más velocidad).
  - No permite ordenar resultados en tiempo de consulta.
  - Lenguaje CQL, con una sintaxis y semántica próxima a SQL, no está estandarizado.
  - No Garantiza ACID.
    - Consistencia eventual: puede haber nodos cuyos datos no estén correctamente actualizados.
  - El desarrollo de aplicaciones cliente con base de datos Cassandra es más complejo (esquemas flexibles, dinámicos y desnormalizados).