Práctica – Tema 2: Cassandra

Marta Zorrilla – Diego García-Saiz

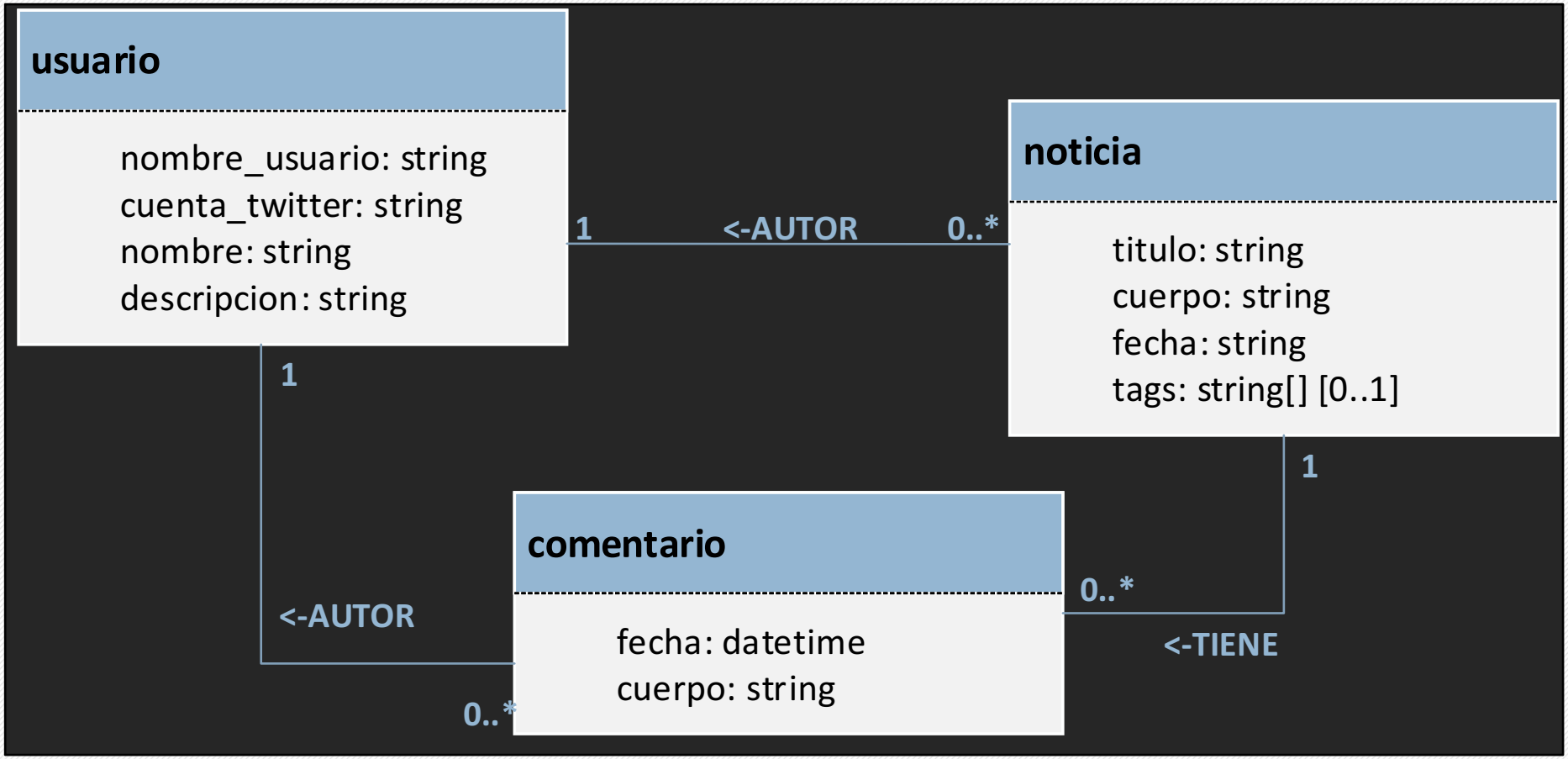
Enero 2017

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

Enunciado

- Se precisa diseñar un blog de noticias donde los usuarios registrados pueda publicar sus comentarios:
  - Cada autor tiene un nombre, un nombre de usuario, una cuenta de *Twitter* y una descripción.
  - Las noticias tienen un título, un cuerpo y una fecha de publicación. Son publicadas por un autor y pueden contener o no, una lista de *tags*.
  - Las noticias reciben comentarios, quedando registrado la persona que lo escribió, el comentario escrito y el momento en el que lo hizo.

Diseño UML


usuario
nombre_usuario: string
cuenta_twitter: string
nombre: string
descripcion: string
1
<-AUTOR
0..*
noticia
titulo: string
cuerpo: string
fecha: string
tags: string[] [0..1]
1
1
<-AUTOR
0..*
comentario
fecha: datetime
cuerpo: string
0..*
<-TIENE

Objetivos del diseño en Cassandra

- Recordamos los objetivos:
  - 1. Distribuir los datos por todo el clúster (equilibrio de carga en los nodos).
  - 2. Optimizar las consultas para que sean lo más rápidas posibles:
    - Cuantas menos particiones tengan que ser accedidas, mejor.

Creación y uso del *Keyspace*

- Primer paso: crear un *Keyspace*. Para el ejemplo, indicaremos una estrategia de replicación simple con factor 5 (ver los apuntes para más información):

```
create keyspace ej1t8 with replication =
{'class':'SimpleStrategy','replication_factor':5};
```

Estrategia de replicación de los datos

Factor de replicación de los datos

- Y para utilizarlo, ejecutamos lo siguiente:

```
use ej1t8;
```

Modelando las consultas sobre usuarios

- En Cassandra, son las consultas que los usuarios desean realizar sobre la BD las que definen qué tablas (familias de columnas) han de ser definidas.

- Supongamos que, inicialmente, se han de optimizar las siguientes DOS consultas:

  - Búsquedas de usuarios por **nombre de usuario**.

  - Búsquedas de usuarios por **cuenta de twitter**.

Modelando las consultas sobre usuarios

- La solución pasaría por crear dos tablas para usuarios con los mismos datos. En una de las tablas la *partition key* será la cuenta de *twitter*, y en la otra el nombre de usuario. De esta forma, permitiendo la redundancia, optimizamos los dos objetivos de Cassandra:

```
CREATE TABLE usuarios_twitter(
        cuenta_twitter text,
        nombre text,
        nombre_usuario text,
        descripcion text,
        PRIMARY KEY(cuenta_twitter));
```

```
CREATE TABLE usuarios_nombre(
        nombre_usuario text,
        cuenta_twitter text,
        nombre text,
        descripcion text,
        PRIMARY KEY(nombre_usuario));
```

Objetivo 1:

- Cada usuario tiene su propia partición.

Objetivo 2:

- Para cada consulta por cuenta de twitter o por nombre de usuario, se accede únicamente a una partición.

Insertado de datos

- Tras crear las dos tablas anteriores, probemos a insertar dos usuarios…
  - … ¡OJO!. Los datos de los usuarios se almacenan de forma redundante en dos tablas, por lo que por cada usuario habrá que hacer dos insertados:

Usuario 1

```
INSERT INTO usuarios_twitter (cuenta_twitter,nombre, nombre_usuario, descripcion)
        VALUES ('Agp77','Agapito','Agp77','Me gustan los blogs');

INSERT INTO usuarios_nombre(nombre_usuario, cuenta_twitter,nombre, descripcion)
        VALUES('Agp77','Agp77','Agapito','Me gustan los blogs');
```

Usuario 2

```
INSERT INTO usuarios_twitter(cuenta_twitter,nombre, nombre_usuario, descripcion)
        VALUES('Charles_UK','Charles','Blogger_forever','Yo soy yo');

INSERT INTO usuarios_nombre(nombre_usuario, cuenta_twitter,nombre, descripcion)
        VALUES('Blogger_forever','Charles_UK','Charles','Yo soy yo');
```

Insertado de datos

- Probemos a realizar más insertados:
  - ¿Podríamos insertar filas con columnas sin valor?. Sin problema (¿por qué?... repasar los apuntes), probemos a ejecutar lo siguiente:

```
INSERT INTO usuarios_twitter (cuenta_twitter,nombre, nombre_usuario)
    VALUES (‘Pete el magnifico',‘Pedro',‘Pete el blogger')
```

  - ¿Podríamos insertar una fila sin *Primary Key (PK)*?. Obviamente no, ya que la *partition key*, que forma parte de la *PK*, es necesaria para determinar en que nodo se almacenan los datos. La siguiente instrucción retornará por tanto un error:

```
INSERT INTO usuarios_twitter (nombre, nombre_usuario)
    VALUES (‘Pedro',‘Pete el blogger')
```

Consultas sobre usuarios

- Probemos ahora a consultar los datos del usuario ‘Agp77’:
  - Por cuenta de twitter:

```
SELECT * FROM usuarios_twitter WHERE cuenta_twitter = 'Agp77'
```

  - Por nombre de usuario:

```
SELECT * FROM usuarios_nombre  WHERE nombre_usuario = ‘Agp77'
```

Consultas sobre índices

- ¿Y si quisiésemos consultar por nombre?. Ejecutemos la siguiente consulta:

```
SELECT * FROM usuarios_twitter WHERE nombre = 'Agapito‘
```

  - ¡Da error!. La consulta no forma parte de la *partition key*. Podríamos utilizar *ALLOW FILTERING*:

```
SELECT * FROM usuarios_twitter WHERE nombre = 'Agapito‘ ALLOW FILTERING;
```

  - La anterior consulta es tremendamente ineficiente, ya que realizar una búsqueda secuencial en todos los nodos que contienen datos de la tabla en cuestión. Otra opción más eficiente sería crear un índice sobre el campo, lo que permitiría realizar la búsqueda sin el *ALLOW FILTERING*:

```
CREATE INDEX nom_sec ON usuarios_twitter(nombre);

SELECT * FROM usuarios_twitter WHERE nombre = 'Agapito‘;
```

  - No obstante, una búsqueda utilizando índices es más lenta que si el campo fuese parte de la *partition key*. Los índices sólo se recomienda para casos muy concretos en los que no existan alternativas más eficientes.

Modelando otra consulta: noticias

- Vamos a añadir otra consulta, esta vez sobre las noticias:
  - 1. Se producen consultas frecuentes al final del día sobre las 10 últimas noticias publicadas, con título, cuerpo, nombre de usuario que la publica y sus tags. Usualmente, se escriben 10 o más noticias al día, con una media regular de entre 100 y 130 por día.

Modelando otra consulta: noticias

- Supongamos que mantenemos las mismas tablas para las consultas sobre usuarios:

```
CREATE TABLE usuarios_twitter(
        cuenta_twitter text,
        nombre text,
        nombre_usuario text,
        descripción text,
        PRIMARY KEY(cuenta_twitter)
```

```
CREATE TABLE usuarios_nombre(
        nombre_usuario text,
        cuenta_twitter text,
        nombre text,
        descripción text,
        PRIMARY KEY(nombre_usuario)
```

- Y ahora queremos optimizar la nueva consulta sobre noticias. La estructura de la tabla sería la siguiente:

```
CREATE TABLE noticias(
        titulo text,
        cuerpo text,
        fecha timestamp,
        tags Set<text>,
        nombre_usuario text,
        PRIMARY KEY (???))
```

¿PRIMARY KEY(fecha)?

Garantiza el objetivo 1. ✓

Pero no el objetivo 2: queremos consultar las 10 primeras noticias, y hacerlo en cuantas menos particiones, mejor. ✗

Además, si se escriben dos noticias justo a la vez, se produciría un ***upsert***.

Más aún, las noticias no aparecerían en orden

Modelando otra consulta: noticias


```
CREATE TABLE noticias(
        titulo text,
        cuerpo text,
        fecha timestamp,
        tags Set<text>,
        nombre_usuario text,
        PRIMARY KEY (???))
```

¿PRIMARY KEY(id)?

Garantiza el objetivo 1. Al ser la *partition key* única (tipo UDDI), los datos se repartirán a lo largo de todos los nodos, por lo que habrá un equilibrio en la carga de cada uno de ellos. Además, se garantiza la unicidad (no hay riesgo de *upserts*). ✓

Sigue sin conseguirse el objetivo 2. Los datos no están ordenados en cada partición por fecha. ✗

```
CREATE TABLE noticias(
        titulo text,
        cuerpo text,
        fecha timestamp,
        tags Set<text>,
        nombre_usuario text,
        PRIMARY KEY (???))
```

¿PRIMARY KEY((dia,mes, anio),fecha)?
Partition key por día, mes y año: si las noticias son escritas de forma regular por día, se garantiza el objetivo 1. ✓

Si además se escriben al menos 10 noticias al día, se garantiza el acceso a una sola partición en la consulta. Dado que el enunciado nos decía que se escribían al menos 10 noticias por día, el objetivo 2 también está garantizado. ✓

Al estar ordenadas por fecha, la consulta es aún más eficiente, se muestran las 10 primeras filas de la partición.

Sigue sin garantizarse la unicidad (riesgo de *upserts*) ✗

Modelando otra consulta: noticias

M.Zorrilla - D.García


```
CREATE TABLE noticias(
        titulo text,
        cuerpo text,
        fecha timestamp,
        tags Set<text>,
        nombre_usuario text,
        PRIMARY KEY (???))
```

¿PRIMARY KEY((dia,mes, anio),fecha,nombre_usuario)

La fecha sigue siendo el primer criterio de ordenación. ✓

En la escritura, se ordena también por usuario, lo que puede hacerla un poco más lenta.

No obstante, si asumimos que un usuario no es capaz de escribir dos noticias a la vez (si la aplicación cliente lo impide), se garantiza la unicidad. ✓

Incluso así, si el cliente falla al controlar este tipo de eventos, pueden producirse *upserts*. ✗

```
CREATE TABLE noticias(
        titulo text,
        cuerpo text,
        fecha timeuuid,
        tags Set<text>,
        nombre_usuario text,
        PRIMARY KEY (???))
```

¿PRIMARY KEY((dia,mes, anio),fecha)

La fecha sigue siendo criterio de ordenación. Ahora es única para cada fila (*timeuudi* en vez de *timestamp*), por lo que no se necesitan de más campos en la *PRIMARY KEY* para garantizar unicidad. ✓

En este caso, se garantiza al 100% la unicidad (ningún riesgo de upserts). ✓

¡No olvides añadir WITH CLUSTERING ORDER BY (fecha DESC)!

Insertado de datos en noticias

- Probemos algún INSERT en la última tabla:

```
insert into noticias (fecha,dia,mes,anio,titulo)
values(now(),10,12,2015,'1');
insert into noticias (fecha,dia,mes,anio,titulo,tags)
values(now(),10,12,2015,'2',{'a','b'});
```

Función que devuelve la fecha actual

...

Insertado en 'Set'

- Y consultemos las 10 últimas noticias:

```
select * from noticias where dia =10 and mes =12 and
anio=2015 limit 10;
```

Noticias: ¿y sí cambiasen las condiciones de partida?

- La solución es válida ya que en el enunciado se indica que “se escriben 10 o más noticias al día”.
    - Si esto no fuese así, ¿esta solución seguiría siendo válida? ¿Por qué? ¿Qué cambiaría en el diseño de la tabla si, por ejemplo, se escribiesen más de 10 noticias por mes, pero menos de 10 por día?.

Modelando otra consulta: noticias de usuario

- Vamos a añadir unas consultas más, esta vez sobre las noticias:
  - 1. Consultas frecuentes al final del día sobre las 10 últimas noticias publicadas, con título, cuerpo, nombre de usuario que la publica y sus tags. Usualmente, se escriben 10 o más noticias al día, con una media regular de entre 100 y 130 por día.
  - 2. Consultas frecuentes de las noticias publicadas por un usuario concreto, según su nombre de usuario, y en las que aparezca también su descripción de usuario.

Modelando otra consulta: noticias de usuario

- Seguimos con la misma base de datos: tablas “usuario_nombre” y “usuario_twitter” siguen existiendo, así como la tabla noticias anterior, que llamaremos “noticias_fecha”.

- Queremos además optimizar la consulta 2:
  - Nueva tabla: “noticias_usuario”

Modelando otra consulta: noticias de usuario

M.Zorrilla - D.García


```
CREATE TABLE noticias_usuario(
        titulo text,
        cuerpo text,
        fecha timeuuid,
        descripcion text,
        tags Set<text>,
        nombre_usuario text,
        PRIMARY KEY (???))
```

¿PRIMARY KEY(nombre_usuario) ?

Se garantiza que todas las noticias de un usuario estén en la misma partición… ✓

¡NO SIRVE!. Dado que se espera que un usuario escriba varias noticias, al guardar una nueva noticia de un usuario, la anterior será sobrescrita (*upsert*) ✗

```
CREATE TABLE noticias_usuario(
        titulo text,
        cuerpo text,
        fecha timeuuid,
        descripcion text,
        tags Set<text>,
        nombre_usuario text,
        PRIMARY KEY (???))
```

¿PRIMARY KEY(nombre_usuario, fecha) ?

Se garantiza que todas las noticias de un usuario estén en la misma partición. ✓

Se añade el timeuddi de la noticia a la *PRIMARY KEY* para garantizar la unicidad. ✓

Modelando otra consulta: noticias de usuario

M.Zorrilla - D.García

- ¿Y si en vez de dos tablas para las noticias, tenemos una sola tabla que responda a las dos consultas?:

```
CREATE TABLE noticias (
        titulo text,
        cuerpo text,
        fecha timeuudi,
        descripción text,
        tags Set<text>,
        nombre_usuario text,
        dia int, mes int, int anio,
        PRIMARY KEY (???))
```

¿PRIMARY KEY((nombre_usuario, dia, mes anio), fecha) ?

Se garantiza que todas las noticias de un usuario publicadas en un día concreto estén en la misma partición.

El objetivo 1 se optimiza: más particiones, luego más distribución de los datos ✓

Pero el objetivo 2 se ve perjudicado: Al consultar las 10 primeras noticias, o por usuarios, muy probablemente haya que acceder a varias particiones. ✗

Más aún, las noticias sólo están ordenadas por fecha para cada usuario, por lo que no se podrían retornar TODAS las noticias en orden, sino sólo las de un usuario concreto. ✗

Modelando otra consulta: comentarios

- Pasemos ahora a las consultas sobre los comentarios:
  - 1. Mostrar los comentarios en un rango de fechas, ordenados descendentemente por fecha, junto con el título de la noticia a la que pertenecen y el usuario que los realiza.

Modelando otra consulta: comentarios

- Otra vez, suponemos que el resto de tablas de usuarios y noticias siguen existiendo en la base de datos.

- Tratamos ahora de responde a la consulta 1 sobre los comentarios:

```
CREATE TABLE comentarios (
        nombre_usuario text,
        texto text,
        fecha timeuddi,
        titulo_noticia text,
        PRIMARY KEY (???))
```

¿PRIMARY KEY(fecha) ?

Se garantiza unicidad -> timeuddi.

El objetivo 1 se cumple: 1 partición por fila. ✓

El objetivo 2, no se cumple: hay que consultar demasiadas particiones, ya que la *partition key* es la propia fecha. ✗

Además, cada comentario está en una partición diferente (unicidad del timeuddi), por lo que no se retornarán en orden. ✗

Modelando otra consulta: comentarios

M.Zorrilla - D.García


```
CREATE TABLE comentarios (
        nombre_usuario text,
        texto text,
        fecha timeuddi,
        titulo_noticia text,
        dia int, mes int, int anio,
        PRIMARY KEY (???))
WITH CLUSTERING ORDER BY
        (??? DESC)
```

¿**PRIMARY KEY**((dia, mes, anio), fecha)
WITH CLUSTERING ORDER BY (fecha **DESC**)?

Se garantiza unicidad -> timeuddi.

Particiones por día: si el número de comentarios es uniforme (o aproximadamente) por día, se cumple el objetivo 1. ✓

Mejora en el objetivo 2: menos particiones por consultar. Si el rango de fechas consultado excede del día, hay que realizar varias subconsultas, una por cada día en el rango consultado, para devolverlas en orden ✓

¿**PRIMARY KEY**((mes, anio), fecha)
WITH CLUSTERING ORDER BY (fecha **DESC**)?

Se garantiza unicidad -> timeuddi.

```
CREATE TABLE comentarios (
        nombre_usuario text,
        texto text,
        fecha timeuddi,
        titulo_noticia text,
        mes int, anio int,
        PRIMARY KEY (???)) WITH
        CLUSTERING ORDER BY
                (??? DESC)
```

Menos particiones (más filas por partición): Menos beneficioso para el objetivo 1 (menos distribución), más beneficioso para el objetivo 2 (menos particiones a consultar). ✓
Si el rango de consultas excede el mes, también será necesario hacer varias consultas por cada mes en el rango. ✓

Modelando otra consulta: comentarios


```
CREATE TABLE comentarios (
        nombre_usuario text,
        texto text,
        fecha timeuddi,
        titulo_noticia text,
        dia int, mes int, int anio,
        PRIMARY KEY (???))
WITH CLUSTERING ORDER BY
                (??? DESC)
```


¿PRIMARY KEY((dia, mes, anio), fecha)

WITH CLUSTERING ORDER BY (fecha DESC)?

Se garantiza unicidad -> timeuddi.

Particiones por dia: si el número de comentarios es uniforme (o aproximadamente) por dia, se cumple el objetivo 1.

Mejora en el objetivo 2: menos particiones por consultar. Si el rango de fechas consultado excede del dia, hay que realizar varias subconsultas, una por cada dia en el rango consultado, para devolverlas en orden

NO SIEMPRE SE PUEDEN CONSEGUIR OPTIMIZAR LOS DOS OBJETIVOS. A VECES OPTIMIZAR UNO VA EN DETERIMIENTO DEL OTRO, POR LO QUE DEPENDIENDO DE LA SITUACIÓN HABRÁ QUE DECIDIR QUÉ HACER.

```
CREATE TABLE comentarios (
        nombre_usuario text,
        texto text,
        fecha timeuddi,
        titulo_noticia text,
        mes int, anio int,
        PRIMARY KEY (???)) WITH     CLUSTERING ORDER BY
                (??? DESC)
```

¿PRIMARY KEY((mes, anio), fecha)
&
WITH CLUSTERING ORDER BY (fecha DESC)?

Se garantiza unicidad -> timeuddi.

Menos particiones (más filas por partición): Menos beneficioso para el objetivo 1 (menos distribución), más beneficioso para el objetivo 2 (menos particiones a consultar).

Si el rango de consultas excede el mes, también será necesario hacer varias consultas por cada mes en el rango.

Practicar con otras consultas

- Probemos a realizar varios insertados sobre la tabla “comentarios”.
  - Para facilitar las cosas, sustituiremos en esta práctica el tipo timeuuiD de la fecha por un timestamp.
- Una vez hechas las inserciones, hagamos una consulta con rangos…
  - …¿sobre qué atributos?

Practicar con más consultas

- Probemos la siguientes consultas:

```
SELECT * FROM comentarios WHERE mes = 1;
```


  - Lanza error: todos los campos de la *partition key* han de estar en el *WHERE*.

```
SELECT * FROM comentarios WHERE mes = 1 and anio <  2015
```

  - Lanza error: no se pueden ejecutar operaciones diferentes a las de igualdad (=) sobre la *partition key*.

```
SELECT * FROM comentarios WHERE fecha > ??? And fecha < ???
ALLOW FILTERING;
```

  - Se permite, pero requiere de búsqueda secuencial sobre los valores de la *clustering key*.

Modelando la consulta comentarios de los autores de las noticias

- Pasemos ahora a las consultas sobre los comentarios:
  - 1. Mostrar los comentarios en un rango de fechas, ordenados descendentemente por fecha, junto con el título de la noticia a la que pertenecen y el usuario que los realiza.
  - 2. Mostrar los comentarios hechos por los mismos autores de la noticia, según el nombre de usuario, ordenados por fecha. (PARA REALIZAR POR EL ESTUDIANTE)