

Bases de Datos a Gran Escala

Sesión 5 · Bases de datos columnares: Cassandra

Diego Sevilla Ruiz

Departamento de Ingeniería y Tecnología de Computadores

Facultad de Informática · Universidad de Murcia

dsevilla@um.es

Bases de datos columnares

Almacenar por columnas para leer sólo lo que se consulta

Un caso paradigmático

BD transaccional (SQL) de una empresa. **Transacciones comerciales**, ~100 columnas:

TrId	fecha	hora	sucursal	cuantía	cliente	precio medio ítem	nítems	pago	...
...	...	...	...	...	...	...	...	...	...

Cada poco tiempo: saldo medio de las compras de cada sucursal

Un caso paradigmático (ii)

```
SELECT sucursal, AVG(cuántía)
FROM Transacciones
GROUP BY sucursal;
```

Ojo: sólo necesitamos `cuántía` y `sucursal`

Un caso paradigmático (iii)

Cuestiones relevantes:

1. El SGBDR almacena los elementos **por filas**
2. $\Rightarrow$ un bloque de disco contiene un conjunto de filas
3. Como sólo nos interesan dos columnas, hay que desechar los otros 99 campos (lectura de bloques de disco innecesarios)

¿Y si...? Almacenamiento de los datos **por columnas** (Vertica, MonetDB):

- **Todos los datos de una columna se almacenan contiguos en disco:** se leen menos bloques
- Favorece proyecciones, agregados y compresión en consultas analíticas

Esto es **columnar analítico** (sesión 3). Cassandra y HBase pertenecen a otra familia: filas o particiones con celdas, que pueden ser dispersas, localizadas por su clave

Introducción a Cassandra

Qué es y quién la usa

¿Qué es Apache Cassandra?



- Sistema gestor de bases de datos NoSQL distribuido, de almacenamiento *wide column*, gratuito y de código abierto
- Diseñado para manejar grandes cantidades de datos en muchos servidores que no tienen que ser de alto rendimiento, con alta disponibilidad y sin un punto único de fallo
- Sólido soporte para clústeres que abarcan múltiples centros de datos, con *replicación sin maestro asíncrona* que permite operaciones de baja latencia para todos los clientes

¿Qué es Apache Cassandra? (ii)

- Desarrollada por dos ingenieros de Facebook para el mecanismo de búsqueda del buzón de entrada
- Más tarde, Facebook lo lanzó como proyecto de código abierto en Google Code, y después lo entregó a la fundación Apache
- Utilizada por nombres importantes como Netflix, Apple, Weather Channel, eBay y muchos más

Uso en Netflix (por ejemplo)

De understanding-system-design-of-netflix

Cassandra:-

Netflix uses Cassandra for its scalability and lack of single points of failure and for cross-regional deployments. In effect, a single global Cassandra cluster can simultaneously service applications and asynchronously replicate data across multiple geographic locations.

Cassandra data model in Netflix:-

1. 50+ Cassandra clusters
2. 500+ nodes
3. 30TB daily backups
4. 250k Writes/s at each node

Uso en Netflix (ii)

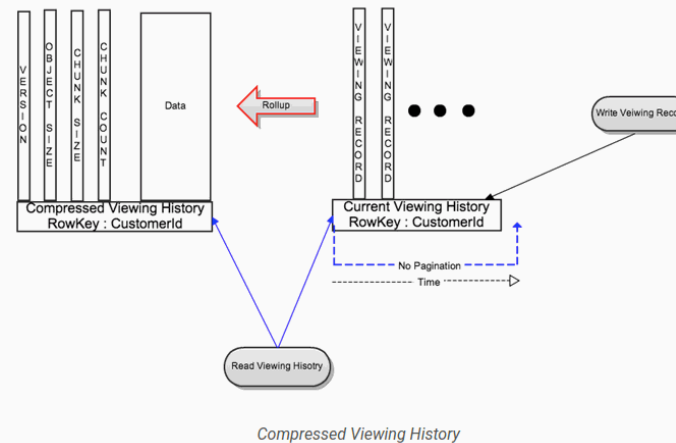
Netflix stores all kinds of data across its Cassandra DB instances, including all user-collected event metrics.

As user data began to increase, there needed to be a more efficient way to manage data storage. Netflix Redesigned data storage architecture with two main goals in mind:

- Smaller Storage Footprint.
- Consistent Read/Write Performance as viewing per member grows.

The solution to the large data problem was to compress the old rows. Data were divided into two types:

- Live Viewing History (LiveVH): Small number of recent viewing records with frequent updates. The data is stored in uncompressed form.
- Compressed Viewing History (CompressedVH): A large number of older viewing records with rare updates. The data is compressed to reduce the storage footprint. Compressed viewing history is stored in a single column per row key.



Modelo de datos

La clave primaria decide en qué nodo vive cada fila

Modelo de datos en Cassandra

- Cassandra es similar a SQL en el sentido de que almacena *tablas* y *filas*
- El lenguaje utilizado para crear tablas y manipular datos es CQL, similar a SQL
- Sin embargo, difiere de SQL en bastantes aspectos: al especificar el esquema de una tabla (`CREATE TABLE`), le estamos diciendo ***cómo va a distribuir los datos entre los hosts de almacenamiento***
- La especificación `PRIMARY KEY` indica en cuál de los nodos se almacenará cada entrada: por eso se llama ***clave de partición***

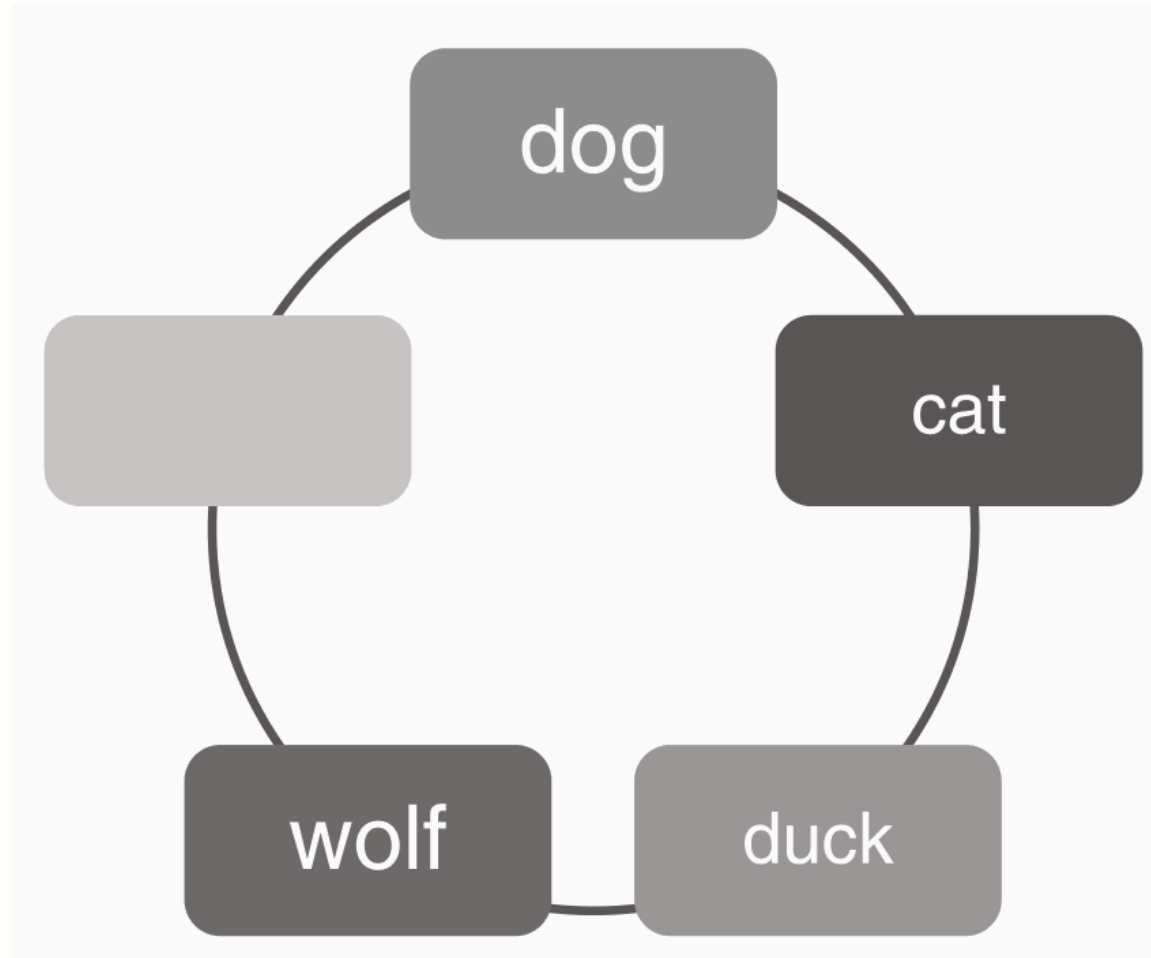
Modelo de datos: ejemplo

```
CREATE TABLE animals (  
  name TEXT PRIMARY KEY,  
  species TEXT,  
  subspecies TEXT,  
  genus TEXT,  
  family TEXT);
```

name especifica en qué ordenador residirá cada fila:

```
select * from animals;  
name | family | genus | species | subspecies  
-----+-----+-----+-----+-----  
cat | Felidae | Felis | F. catus | null  
duck | Anatidae | Anas | A. platyrhynchos | null
```

Modelo de datos: claves de partición



Claves anidadas (compuestas)

- Las claves también pueden ser *anidadas*
- La primera parte especifica la clave de partición, y la segunda, la clave de *clustering* (cómo se agrupan y ordenan las filas dentro de la partición para formar una *wide column*)
- Todas las filas con la misma clave de partición forman, *de forma lógica*, una única fila ancha, ordenada por la clave de *clustering*

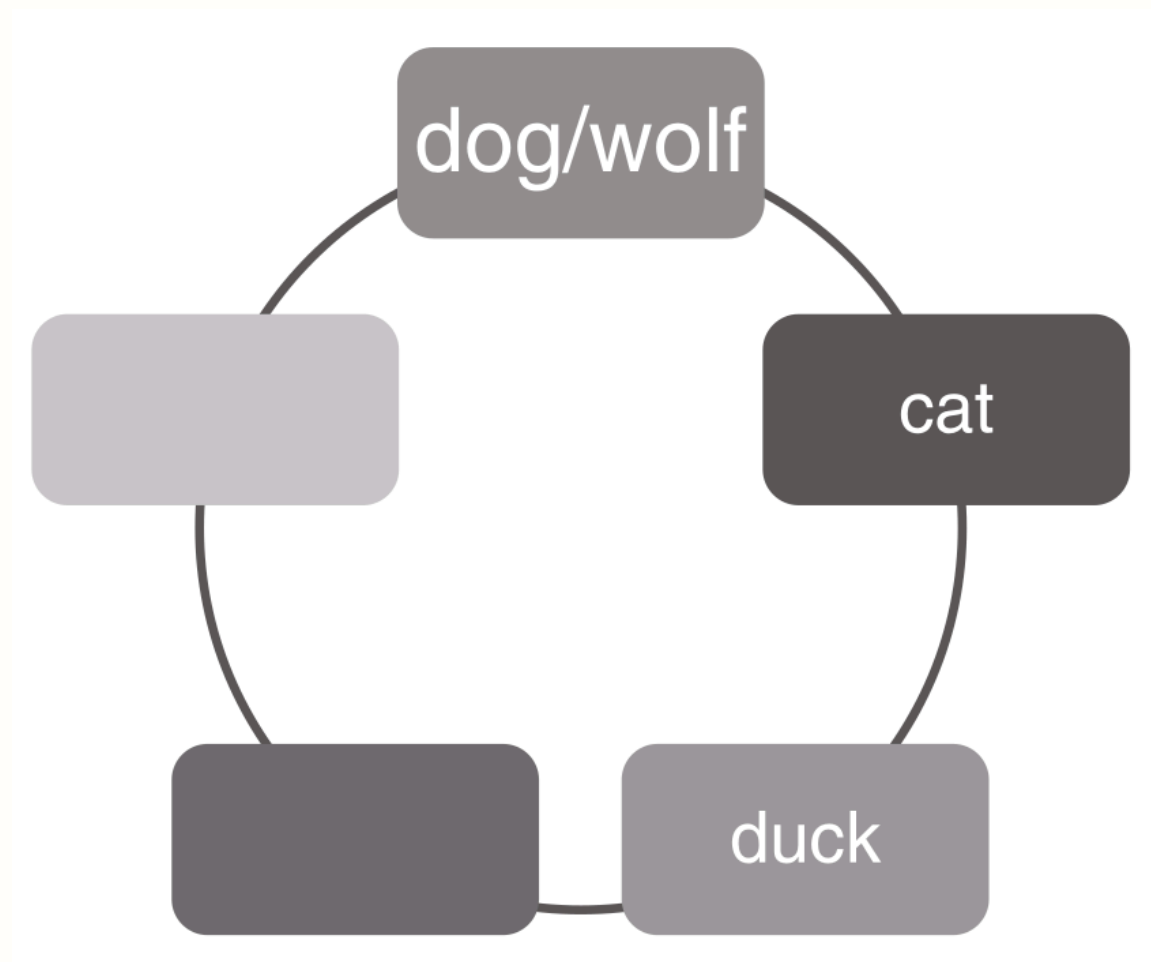
```
CREATE TABLE animals (  
  name TEXT,  
  species TEXT,  
  subspecies TEXT,  
  genus TEXT,  
  family TEXT,  
  PRIMARY KEY ((family, genus), name)  
);
```

Claves compuestas: resultado

```
select * from animals;
```

family	genus	name	species	subspecies
Anatidae	Anas	duck	A. platyrhynchos	null
Felidae	Felis	cat	F. catus	null
Canidae	Canis	dog	C. lupus	C. l. familiaris
Canidae	Canis	wolf	C. lupus	null

Claves compuestas



Modelo de datos vs. SQL

Otro ejemplo: Sistema que recoge eventos, en SQL:

```
CREATE TABLE events (  
  id INT PRIMARY KEY,  
  time TIME,  
  event_type INT REFERENCES event_types(id),  
  data text  
);
```

En Cassandra no hay claves ajenas: referenciar el tipo de evento por su `id` no es posible

Modelo de datos vs. SQL (ii)

Lo que se suele hacer en NoSQL es ***desnormalizar***. incluir la descripción del tipo de evento en la propia tabla:

```
CREATE TABLE events (  
  id UUID PRIMARY KEY,  
  time TIMESTAMP,  
  event_type TEXT,  
  data text  
);
```

Esto funciona y es una copia de cómo lo haríamos en SQL

Modelo de datos vs. SQL (iii)

- Sin embargo, en Cassandra **TAMPOCO** podemos filtrar filas que no sean la clave primaria: Cassandra considera que tardaría demasiado (y de hecho es así)
- Sería difícil encontrar eventos por otras claves que no sean su `id`, como la hora o el tipo de evento
- ¿Cómo solucionarlo?

Modelo de datos vs. SQL (iv)

- **Opción 1:** indexar el campo `time`
 - Puede generar una alta carga, ya que los eventos estarán dispersos en el clúster
- **Opción 2:** utilizar el campo `time` como clave de partición (`PRIMARY KEY`)
 - En una clave primaria con múltiples campos, la primera actúa como clave de partición
 - Los campos adicionales forman parte del nombre de la columna (una *wide column* o *wide row*)

Modelo de datos vs. SQL (v)

Veamos la opción 2:

```
CREATE TABLE events (  
  hour TIMESTAMP,  
  id UUID,  
  time TIMESTAMP, -- El tiempo exacto del evento, para tenerlo de referencia  
  event_type TEXT,  
  data text,  
  PRIMARY KEY(hour, id)  
);
```

Con este modelo podemos consultar fácilmente todos los eventos de una misma hora: todo lo ocurrido en esa hora está en una única fila en el disco

Modelo de datos vs. SQL (vi)

- **SIN EMBARGO:** Ahora durante una hora todos los eventos se escriben al mismo servidor (el resto está inactivo)
- Si se tienen lecturas y/o escrituras intensas, puede ser necesario segmentar la fila en varias, con una granularidad más fina
- Un evento por fila es difícil de consultar; una fila por hora puede generar **cuellos de botella** y filas muy grandes (todas las lecturas/escrituras irían a los mismos archivos en disco)
- Podemos segmentar en varias filas fáciles de leer, siempre que tengamos algo por lo que consultar: aquí podemos elegir `event_type`
- Podemos mejorar el rendimiento asegurando que el orden se guarda por la hora del evento, no por el `id`

Modelo de datos vs. SQL (vii)

```
CREATE TABLE events (  
  hour TIMESTAMP,  
  id UUID,  
  time TIMESTAMP,  
  event_type TEXT,  
  data text,  
  PRIMARY KEY((hour, event_type), time, id)  
) WITH CLUSTERING ORDER BY (time DESC);
```

Modelo de datos vs. SQL (viii)

```
SELECT * FROM events;
```

hour	event_type	time	data	id
2013-06-13 11:00:00	click	2013-06-13 11:00:00	some data	3 ...
2013-06-13 11:00:00	page_view	2013-06-13 11:00:01	some data	2 ...
2013-06-13 11:00:00	error	2013-06-13 11:00:05	some data	0 ...
2013-06-13 11:00:00	redirect	2013-06-13 11:00:09	some data	1 ...

¿Y si queremos consultar **también** por **id**?

Varias opciones:

- Crear un índice secundario sobre **id**
 - No recomendable, porque los valores están dispersos por todo el clúster,
 - Y, ¿el índice dónde vive?
- Vistas materializadas (todavía consideradas experimentales y tienen mucha sobrecarga)
- Crear otra tabla con la misma información, pero con **id** como clave de partición
 - Duplicación de datos, pero muy eficiente
 - Implica dos escrituras (transacción)
- Crear otra tabla **id** -> (**hour** , **event_type** , **time** , **id**), y luego consultar la tabla principal
 - Dos lecturas (aunque todavía muy eficiente) y dos escrituras (transacción), pero menos duplicación de datos
 - **Generalmente la opción preferida**

Una tabla por consulta: Stackoverflow

Dos consultas de la aplicación, dos tablas (las de la práctica):

Consulta	Tabla	Partición	Clustering
Una pregunta a partir de su Id	Questions	Id	CreationDate
Todas las respuestas de una pregunta	Answers	ParentId	CreationDate

- Con una única tabla `Posts` particionada por `Id`, las respuestas de una pregunta quedarían repartidas por todo el clúster
- Con `ParentId` como clave de partición viven juntas y se leen con una sola consulta
- Se modela la consulta antes que la tabla

Colecciones

Conjuntos, listas y mapas dentro de una fila

Colecciones

- Cassandra incluye colecciones como parte de su modelo de datos
- Pueden proporcionar flexibilidad en las consultas: **conjuntos, listas y mapas**

Conjuntos (`SET`): mantienen un conjunto único de elementos sin necesidad de leer antes de escribir

```
CREATE TABLE users (  
    email TEXT PRIMARY KEY,  
    portfolios SET<UUID>,  
    tickers SET<TEXT>  
);
```

Conjuntos: actualización

```
UPDATE users
  SET portfolios = portfolios + {756716f7-2e54-4715-9f00-91dcbea6cf50},
      tickers = tickers + {'AMZN'}
WHERE email = 'foo@bar.com';
UPDATE users
  SET tickers = tickers + {'GOOG'}
WHERE email = 'foo@bar.com';
```

email	portfolios	tickers
foo@bar.com	{756716f7-2e54-4715-9f00-91dcbea6cf50}	{'AMZN', 'GOOG'}

Conjuntos: eliminar elementos y borrar

```
UPDATE users
  SET tickers = tickers - {'AMZN'}
  WHERE email = 'foo@bar.com';
-- tickers ⇒ {'GOOG'}
```

```
DELETE tickers
FROM users
WHERE email = 'foo@bar.com';
-- tickers ⇒ null
```

Listas

Cuando no se requiere unicidad y se necesita mantener el orden.
Ejemplo: permitir que los usuarios especifiquen sus cinco *tickers* principales:

```
ALTER TABLE users ADD top_tickers list<text>;

UPDATE users SET top_tickers = ['GOOG'] WHERE email = 'foo@bar.com';
UPDATE users SET top_tickers = top_tickers + ['AMZN'] WHERE email = 'foo@bar.com';

-- top_tickers ⇒ ['GOOG', 'AMZN']
```

Mapas

Objeto tipo diccionario, útil para datos tabulares dentro de una fila (ayuda a mitigar la falta de `JOIN` s). Ejemplo: fecha de actualización por *ticker*.

```
ALTER TABLE users ADD ticker_updates map<text, timestamp>;

UPDATE users SET ticker_updates = { 'AMZN': '2013-06-13 11:42:12' }
WHERE email = 'foo@bar.com';

UPDATE users SET ticker_updates['GOOG'] = '2013-06-13 12:51:31'
WHERE email = 'foo@bar.com';
```

Mapas: borrado

```
DELETE ticker_updates['AMZN']  
FROM users  
WHERE email = 'foo@bar.com';
```

email	portfolios	ticker_updates
foo@bar.com	{756716f7 ... }	{'GOOG': '2013-06-13 12:51:31-0400'}

Distribución de datos

Tokens, *hash* consistente y filtros de Bloom

Distribución de datos en Apache Cassandra

- Para distribuir los datos, Cassandra usa un modelo *peer-to-peer* basado en claves de partición
- Para determinar qué nodo contiene un dato, Cassandra usa ***tokens***
- Un token es un entero de 64 bits; cada nodo del clúster tiene asignado un rango de tokens
- Al buscar una fila, Cassandra calcula el token asociado (con una función *hash*) y determina de forma única el nodo al que pertenece
- Cassandra usa así un conjunto de tablas *hash* distribuidas
- Al añadir o eliminar nodos, hay que redistribuir los tokens asignados

¿Qué son las tablas *hash* distribuidas (THD)?

Tablas *hash* distribuidas

- Una base de datos distribuida que almacena y recupera información sobre claves en nodos que pueden unirse o abandonar en cualquier momento
- Los nodos se coordinan sólo entre sí, sin ningún nodo central coordinador
- Arquitectura con nodos réplica $\Rightarrow$ tolerante a fallos
- Las claves se mantienen, en general, distribuidas uniformemente entre los nodos
- Para almacenar en múltiples nodos, se toma el *hash* módulo número de servidores
- Si uno de los nodos falla, hay que cambiar el módulo con el nuevo número de servidores, y empezamos a perder información

¿Recordamos qué era el *Consistent Hashing* (CH)?

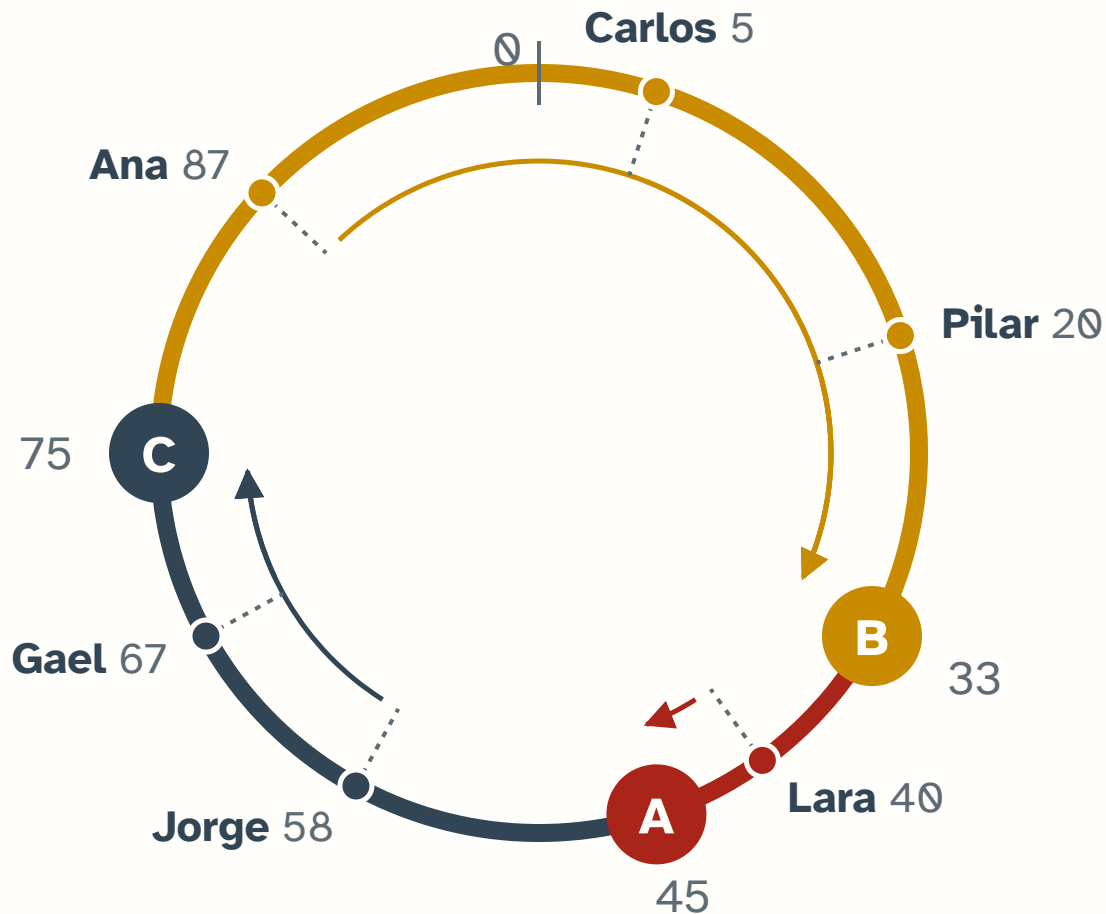
Consistent Hashing

- Similar a lo que pasaba en MySQL con `PARTITION` : las tablas se distribuyen entre distintos *hosts*
- A cada clave se le calcula un *hash*; los posibles valores se disponen en un círculo con aritmética módulo K
- A cada *host* h_i del anillo se le calcula también un valor de *hash*
- Una fila con clave k se almacena en el *host* h_i que cumple:

$$\text{hash}(h_{i-1}) <_{\text{mod } K} \text{hash}(k) <_{\text{mod } K} \text{hash}(h_i)$$

(se almacena en el siguiente *host* siguiendo el círculo)

Consistent Hashing: el anillo



$$\text{hash}(x) = \text{MD5}(x) \bmod 100$$

B hash **33** · guarda (75, 33] pasa por 0
Ana 87 · Carlos 5 · Pilar 20 3 claves

A hash **45** · guarda (33, 45]
Lara 40 1 clave

C hash **75** · guarda (45, 75]
Jorge 58 · Gael 67 2 claves

Cada clave va al primer host que encuentra avanzando en el sentido de las agujas del reloj

Con un punto por host, B guarda 3 claves y A sólo 1

Consistent Hashing (ii)

Para que las claves se distribuyan uniformemente entre servidores:

- Se asignan no una, sino muchas etiquetas (*angles*) a cada servidor: en vez de **A** , **B** y **C** , tener **A0** ... **A9** , **B0** ... **B9** y **C0** ... **C9** , entremezcladas en el círculo
- El factor por el que aumentar el número de etiquetas (*server keys*, o peso) depende de la situación, y puede ser diferente para cada servidor
- Si el servidor **B** fuera el doble de potente, se le asignaría el doble de etiquetas, y terminaría manteniendo el doble de objetos en promedio

Consistent Hashing (iii)

Si eliminamos un servidor (C)...

- Hay que eliminar las etiquetas C0 ... C9 del círculo
- Las claves antes adyacentes a esas etiquetas se reasignan aleatoriamente a A y B ; el resto de claves queda intacto

Si añadimos un servidor (D)...

- Hay que agregar las etiquetas D0 ... D9
- Aproximadamente un tercio de las claves existentes (de A o B) se reasignan a D ; el resto permanece igual

Inciso: *Bloom Filters*

- Otra tecnología habilitadora muy famosa en sistemas distribuidos
- Permiten descartar muy rápido si un elemento está en un conjunto:
 - Si la respuesta es **NO**, el elemento no está en el conjunto
 - En otro caso, no se sabe nada (hay que buscar dentro del conjunto)
- En Cassandra se mantienen por SSTable: permiten descartar un fichero entero sin leerlo (se ve al hablar de las lecturas)

Bloom Filters: funcionamiento

- Se construye un mapa de k bits, donde k es el número de bits de una función *hash*
- Cada elemento e insertado actualiza el mapa (se añaden los bits del *hash*):

$$\text{mapa} = \text{mapa} \mid \text{hash}(e)$$

- Un elemento e' ***puede estar*** en el mapa si están todos los bits de su *hash*:

$$\text{hash}(e') = \text{mapa} \& \text{hash}(e')$$

Bloom Filters: el mapa de bits

Inserción $\text{mapa} = \text{mapa} \mid \text{hash}(e)$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
hash(Ana)	0	0	0	0	0	1	0	1	0	0	1	0	0	0	0	0
hash(Luis)	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0
hash(Jorge)	0	1	0	0	1	0	0	0	1	0	0	0	0	0	0	0

mapa

0	1	0	0	1	1	1	1	1	0	1	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

7 de 16 bits a 1

Comprobación $\text{¿hash}(e') = \text{mapa} \& \text{hash}(e')?$

hash(Ana)	0	0	0	0	0	1	0	1	0	0	1	0	0	0	0	0
hash(Paula)	0	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0
hash(Clara)	0	0	0	0	1	0	0	0	0	0	1	0	1	0	0	0

puede estar
y está

puede estar
falso positivo

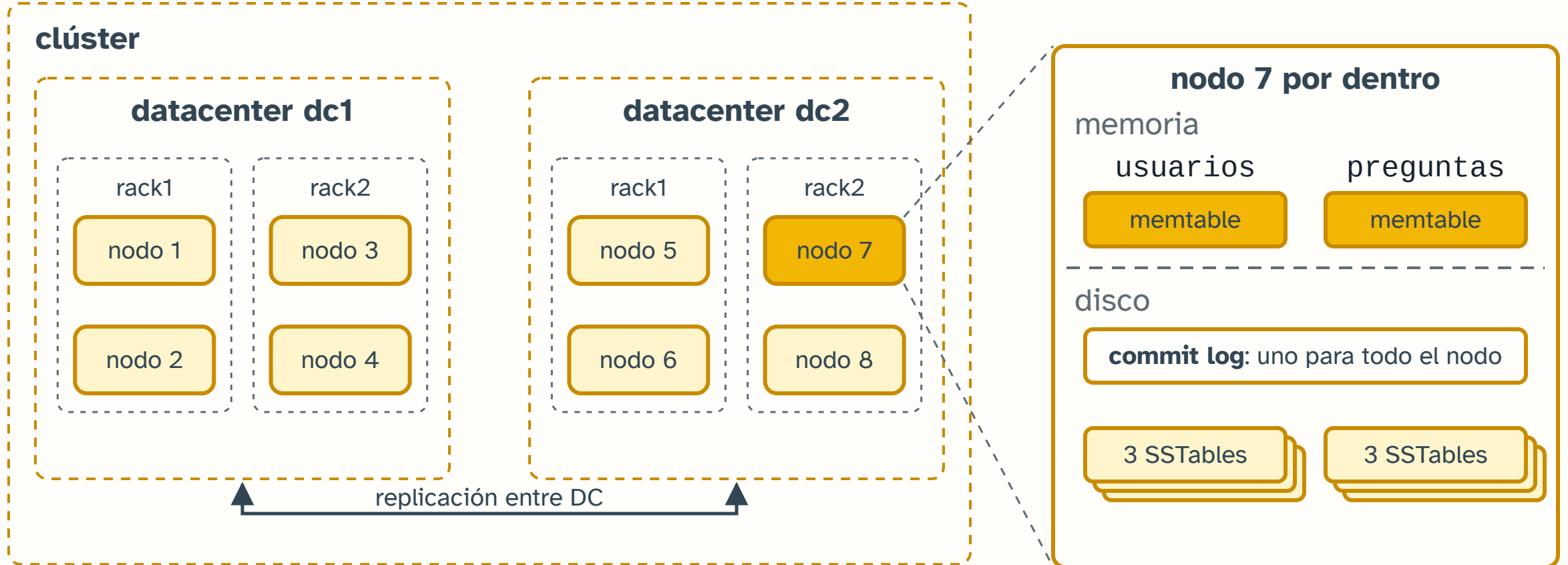
no está
falta el bit 12

hash(e) pone a 1 3 de los 16 bits, elegidos con el MD5 de e · Un solo bit a 0 basta para descartar el elemento

Arquitectura

Nodos, centros de datos y el coordinador de cada petición

Componentes principales



Un **clúster** agrupa uno o más **datacenters**; cada DC, **nodos** repartidos en racks

Todos los nodos son iguales: cualquiera lee, escribe y coordina; no hay maestro

Componentes principales: estructurales

- **Nodo:** componente básico de los datos, máquina donde se almacenan
- **Centro de datos (*DataCenter*):** colección de nodos relacionados; puede ser físico o virtual
- **Clúster:** contiene uno o más centros de datos, y puede extenderse a través de ubicaciones

Componentes principales: internos

- ***Commit Log***: cada escritura se almacena primero aquí; se usa para la recuperación ante fallo
- ***Mem-Table***: tras escribirse en el *commit log*, los datos se almacenan aquí hasta alcanzar un umbral
- ***SSTable***: archivo en disco que almacena los datos de la *MemTable* al alcanzar el umbral; se almacenan secuencialmente y se mantienen por tabla

Commit log, memtable y SSTables



Ana está en la memtable (270) y en las SSTables (250, 260): **gana la más reciente**
La compactación fusionará las SSTables y descartará las versiones viejas

Funcionamiento general

- Cada nodo puede realizar tanto lecturas como escrituras
- No existe jerarquía con nodos maestros o esclavos dentro de un clúster
- Cassandra usa el protocolo *Gossip* para mantener el clúster:
 - Protocolo de comunicación entre pares en el que los nodos intercambian periódicamente información sobre su estado y el de otros nodos
 - Se ejecuta cada segundo, intercambiando mensajes de estado

Funcionamiento general (ii)

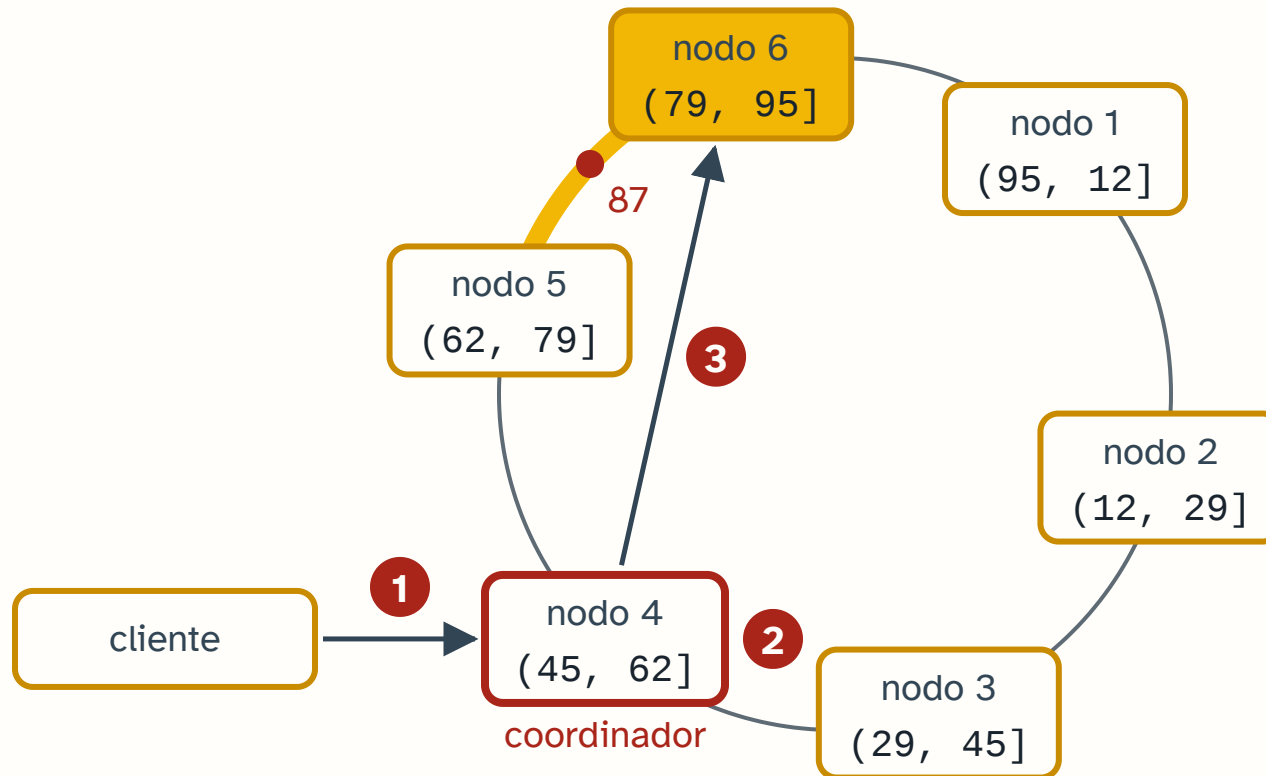
- Cuando un cliente envía una solicitud a un clúster Cassandra, llega primero al nodo ***coordinador***, que actúa como proxy entre el cliente y los nodos involucrados
- El *partitioner* es el componente responsable de determinar cómo distribuir los datos entre los nodos: calcula el *hash* de la clave de partición de una fila
- El *partitioner* no es un nodo: es la misma función en todo el clúster (`Murmur3Partitioner` por omisión), así que cualquier nodo que coordine una petición la aplica y sabe a qué nodos va la fila

Ejemplo de nodo coordinador

Un cliente quiere escribir una nueva fila cuya clave de partición es **Ana** . Envía la fila, con su clave, al nodo 4, que actúa como coordinador: aplica el *partitioner* a la clave y obtiene su token, **87**

Cada nodo guarda el rango de tokens que termina en el suyo, el que aparece en su caja. 87 cae en el rango (79, 95], que guarda el nodo 6, y el coordinador le reenvía la fila aunque él no la guarde

Ejemplo de nodo coordinador: el anillo de tokens



- 1 El cliente envía la fila, con su clave 'Ana', a un nodo cualquiera, que la **coordina**: aquí, nodo 4
- 2 El **coordinador** calcula el token: $\text{token}('Ana') = 87$, que cae en el rango (79, 95]
- 3 La reenvía a **nodo 6**, dueño de ese rango: el coordinador no tiene por qué guardarla

Cada nodo guarda el rango que termina en su token · Tokens de 0 a 99 (MD5 mod 100); Cassandra usa Murmur3, de -2^{63} a $2^{63} - 1$

Replicación y consistencia

Cuántas copias se guardan y cuántas deben responder

Consistencia en Apache Cassandra

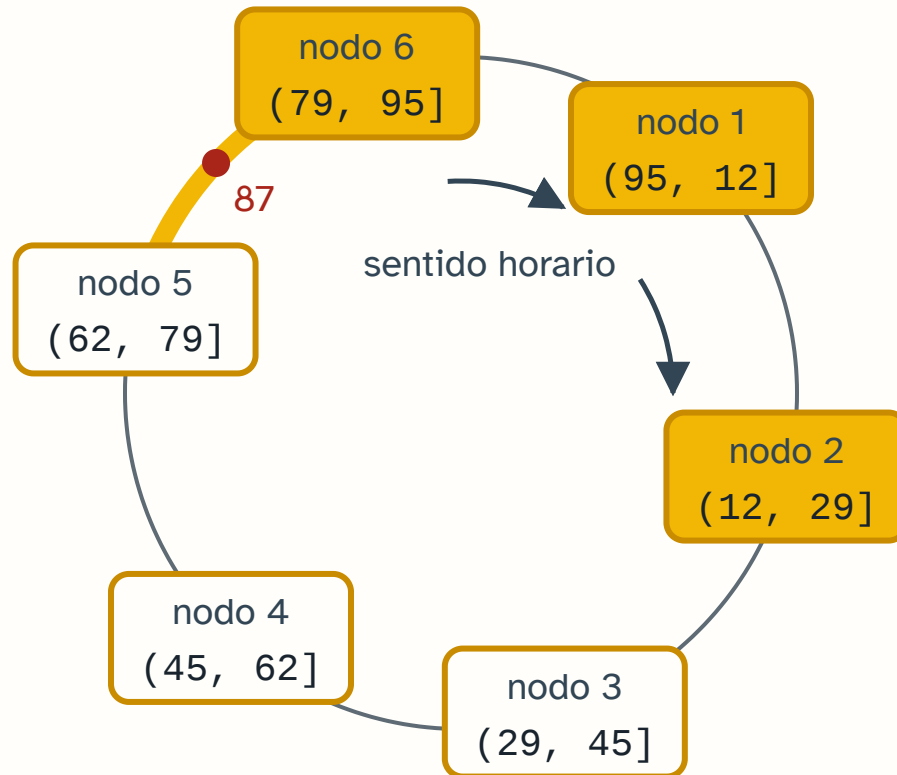
- Cassandra permite ajustar disponibilidad y consistencia configurando **replication factor** y **consistency level**
- El nivel de consistencia se define por consulta, y permite ajustar cuándo se ofrece el resultado a los clientes
- El factor de replicación asegura que las escrituras se envíen a todas las réplicas
- Ejemplo: con consistencia 3 en un clúster con 3 nodos, se necesitaría que los 3 nodos confirmen la escritura antes de darla por buena

Estrategias de replicación

Se eligen al crear el *keyspace* (`WITH replication = { ... }`), y deciden en qué nodos quedan las réplicas de cada fila:

- **SimpleStrategy** : despliegues en un solo centro de datos; no considera la topología de la red. Sólo toma la decisión del *partitioner* y coloca las réplicas restantes en sentido horario
- **NetworkTopologyStrategy** : despliegues en múltiples centros de datos. También parte de la decisión del *partitioner*, pero tiene en cuenta la configuración de *racks* y *datacenters*: cada DC tiene su número de réplicas, repartidas entre *racks* distintos

Réplicas con SimpleStrategy

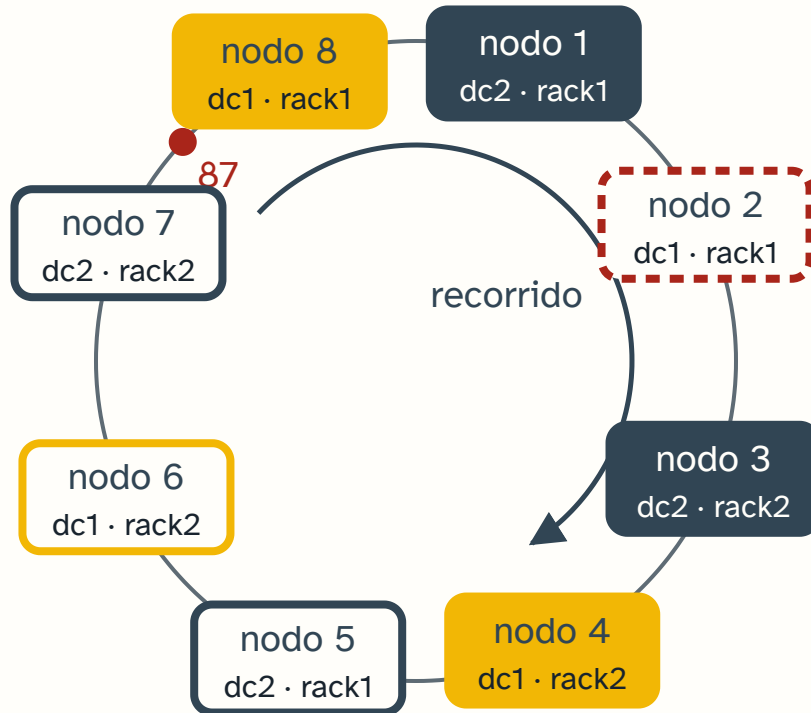


```
{'class': 'SimpleStrategy',  
'replication_factor': 3}
```

rango	réplicas
(95, 12]	nodos 1, 2 y 3
(12, 29]	nodos 2, 3 y 4
(29, 45]	nodos 3, 4 y 5
(45, 62]	nodos 4, 5 y 6
(62, 79]	nodos 5, 6 y 1
(79, 95]	nodos 6, 1 y 2

Cada nodo guarda 3 rangos: el suyo y los de los 2 anteriores · Las réplicas de **Ana**: nodos 6, 1 y 2

Réplicas con NetworkTopologyStrategy



```
{'class': 'NetworkTopologyStrategy',  
'dc1': 2, 'dc2': 2}
```

Recorrido desde token('Ana') = 87:

nodo 8 dc1 · rack1	→ réplica 1 de dc1
nodo 1 dc2 · rack1	→ réplica 1 de dc2
nodo 2 dc1 · rack1	→ se salta: rack1 ya tiene una
nodo 3 dc2 · rack2	→ réplica 2 de dc2
nodo 4 dc1 · rack2	→ réplica 2 de dc1

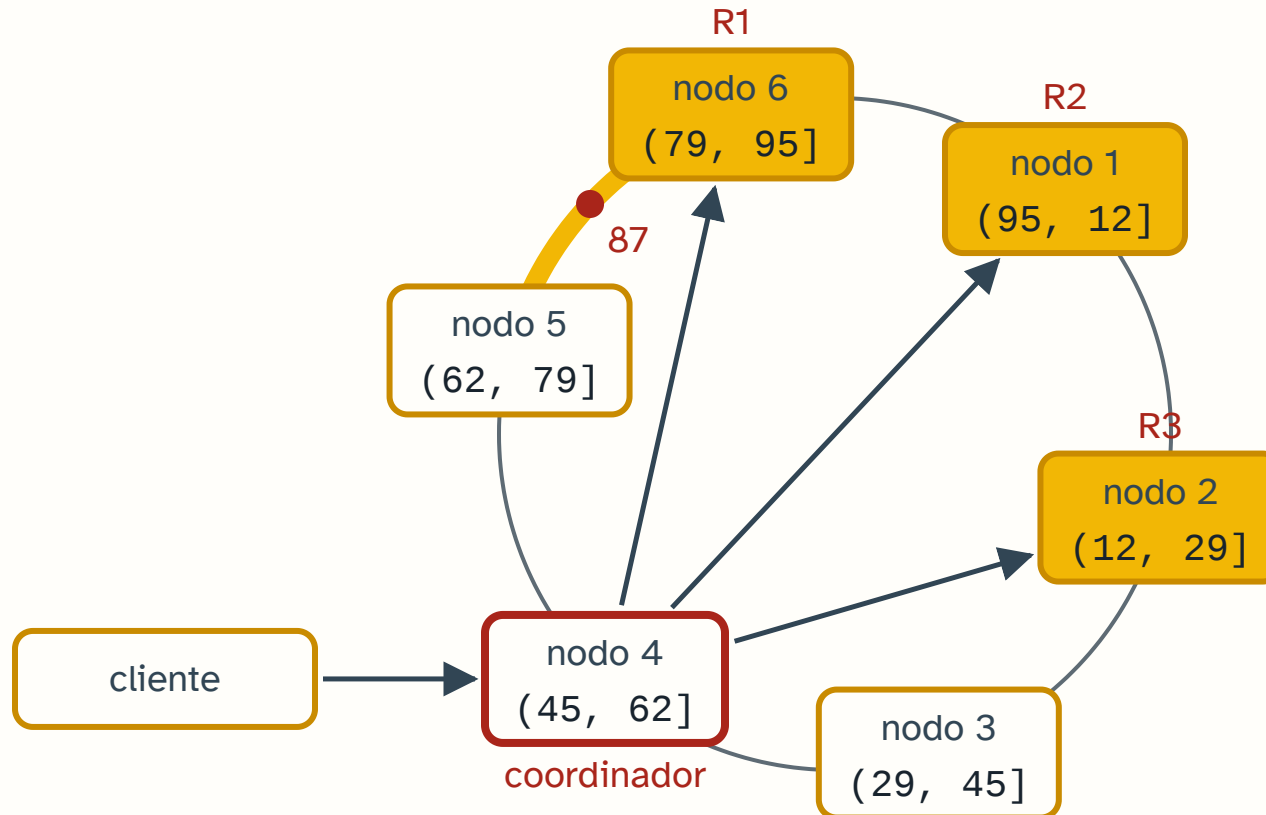
Cada DC elige **sus propias réplicas** en el mismo anillo y las reparte **entre racks distintos**

Así una caída de un rack, o de un DC entero, no se lleva todas las copias

Niveles de consistencia de escritura

- El nivel de consistencia de escritura se elige con el comando `CONSISTENCY` de `cqlsh` o, en el *driver*, dentro de un perfil de ejecución (se ve más adelante): no es una cláusula del `INSERT`
- Es el número de réplicas que deben confirmar al coordinador que su inserción local fue exitosa
- En cuanto recibe tantos *acknowledgements* como el nivel configurado, el coordinador informa al cliente del éxito, sin esperar a las réplicas restantes
- Niveles: `ANY` , `ONE` , `TWO` , `THREE` , `QUORUM` , `LOCAL_QUORUM` , `EACH_QUORUM` , `ALL`
- `ONE` es aceptable si se necesita rendimiento muy alto y no importa perder alguna escritura; `QUORUM` asegura que la mayoría de nodos responda antes de considerar la operación exitosa

Escritura con tres réplicas: el coordinador reparte la fila



- 1 El coordinador calcula $\text{token}('Ana') = 87$
- 2 El **partitioner** da la primera: **nodo 6 (R1)**, dueño de (79, 95]
- 3 SimpleStrategy añade las siguientes, en sentido horario: **nodo 1 (R2) y nodo 2 (R3)**
- 4 Envía la fila a las tres a la vez y responde según el nivel de consistencia

Factor de replicación **RF = 3** · El coordinador no es réplica: sólo reparte la escritura y espera las confirmaciones

Cuándo responde el coordinador

La escritura va a las **3 réplicas a la vez**; el nivel fija cuántas confirmaciones espera el coordinador

ONE

espera 1 de 3

responde a los 2 ms



R2 y R3 también aplican la escritura, pero después

QUORUM

espera 2 de 3

responde a los 5 ms



R3 también aplica la escritura, pero después

ALL

espera 3 de 3

responde a los 9 ms



tiempo (ms)



ANY espera lo mismo que ONE mientras haya réplicas vivas: su diferencia está en qué pasa cuando caen

Niveles de consistencia y sus compromisos

Cada lectura y cada escritura eligen cuántas réplicas deben responder. Con factor de replicación (RF) n :

Nivel	Réplicas que responden	Compromiso
ANY (sólo escritura)	ninguna en concreto: vale una <i>pista</i> (<i>hint</i>) guardada por el coordinador	la escritura más rápida y disponible, pero puede tardar en ser visible
ONE / LOCAL_ONE	1	rápido y muy disponible, pero puede leer una réplica atrasada
QUORUM / LOCAL_QUORUM	la mayoría: $\lfloor n/2 \rfloor + 1$	equilibrio: consistencia sin exigir que estén todas vivas
ALL	las n	la máxima consistencia, pero falla en cuanto cae un solo nodo

LOCAL_ cuenta sólo las réplicas del *datacenter* local; con un único centro de datos equivale al nivel sin prefijo

Réplicas caídas: qué niveles siguen escribiendo

Cada círculo, una réplica:



viva



caída

Nivel • necesita

ANY

una réplica o una pista

ONE

1 réplica

QUORUM

la mayoría: 2

ALL

las 3

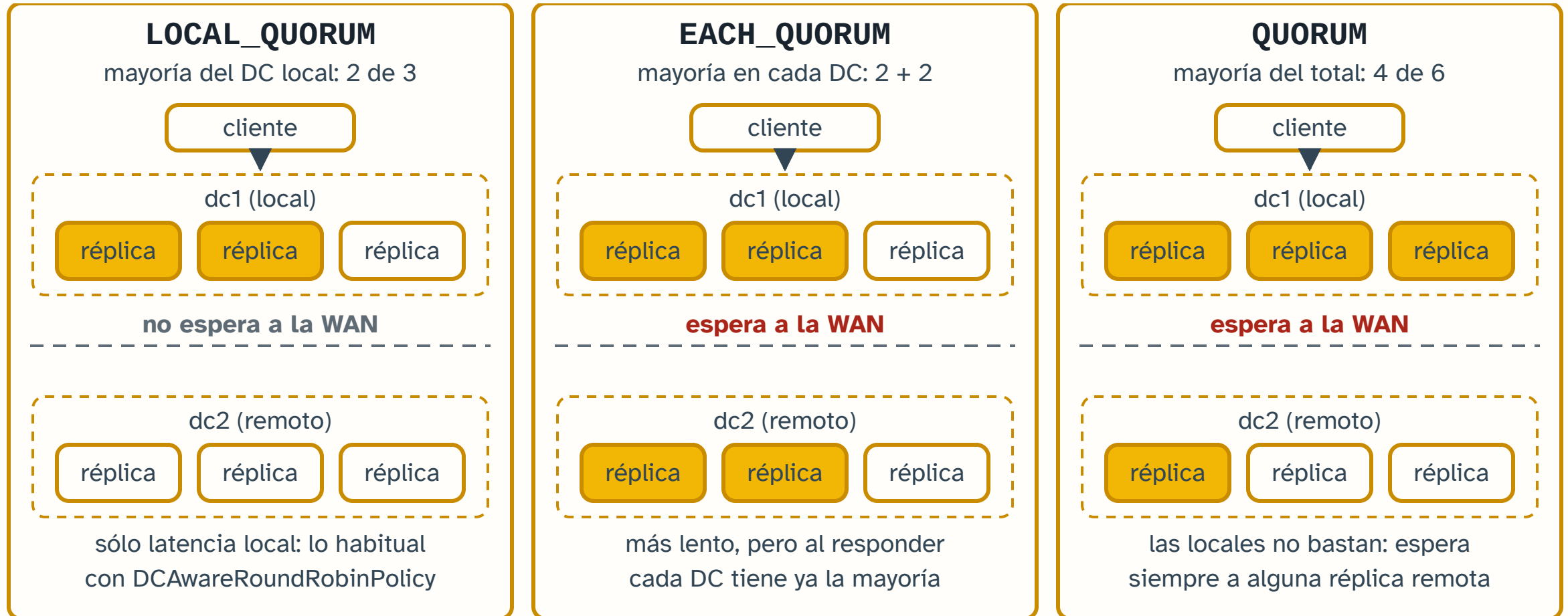
Réplicas caídas (RF = 3)				
	0	1	2	3
ANY	 escribe	 escribe	 escribe	 escribe (pista)
ONE	 escribe	 escribe	 escribe	 Unavailable
QUORUM	 escribe	 escribe	 Unavailable	 Unavailable
ALL	 escribe	 Unavailable	 Unavailable	 Unavailable

Unavailable: el coordinador ve que no quedan vivas las réplicas que exige el nivel y rechaza la escritura sin intentarla

Lectura: el mismo cálculo, salvo ANY, que sólo existe para escribir

Con RF = 2 (las prácticas) la mayoría es 2: QUORUM **no tolera ninguna caída**

Consistencia con varios *datacenters*



NetworkTopologyStrategy con 3 réplicas por DC · Con un solo DC, LOCAL_QUORUM equivale a QUORUM

Escribir a **ONE** no espera a las demás réplicas

- El coordinador envía la escritura a **todas** las réplicas a la vez, pero responde en cuanto contestan las que exige el nivel
- Con **ONE**, las demás la aplican un instante **después**: una lectura inmediata a **ONE** puede caer en una réplica atrasada y devolver el dato antiguo
- Sólo una lectura de **varias** réplicas *reconcilia*: el coordinador pide los datos a una réplica y un resumen (*digest*) a las demás; si difieren, se queda con la versión de marca de tiempo más reciente y repara a las atrasadas (*read repair*)
- La regla que garantiza ver la última escritura: $R + W > RF$, donde R y W son las réplicas leídas y escritas, porque así comparten al menos una

$R + W > RF$: lecturas inmediatas posibles

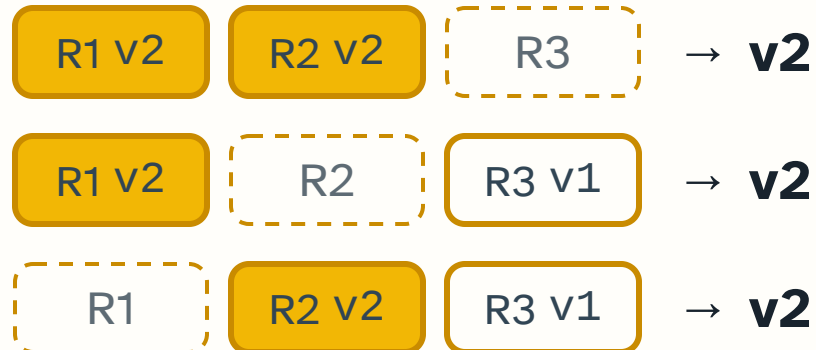
escritura QUORUM · lectura QUORUM

$$R + W = 2 + 2 = 4 > 3 = RF$$

Cuando responde la escritura de v2:



Lecturas inmediatas posibles (2 de 3):



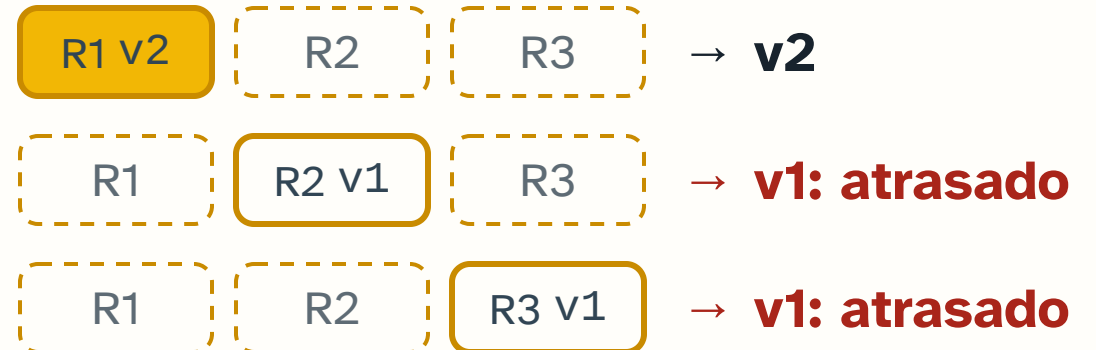
escritura ONE · lectura ONE

$$R + W = 1 + 1 = 2 \leq 3 = RF$$

Cuando responde la escritura de v2:



Lecturas inmediatas posibles (1 de 3):



Si $R + W > RF$, toda lectura comparte **al menos una réplica** con la última escritura
El coordinador devuelve la versión con la marca de tiempo más reciente y repara las atrasadas (read repair)

Por qué **LOCAL_QUORUM** por omisión

- Con dos nodos ($RF = 2$) la mayoría es 2, luego $R + W = 2 + 2 > 2$: cada lectura ve lo último escrito, con independencia de qué réplica responda antes
- El precio es la **disponibilidad**: con $RF = 2$ necesita los dos nodos, y si uno cae falla con **Unavailable**, mientras que **ONE** seguiría respondiendo
- En producción se usa $RF = 3$: la mayoría (2) sigue disponible con un nodo caído
- Con un solo nodo ($RF = 1$) se comporta como **ONE**
- La carga masiva usa **ANY**, porque sólo importa la velocidad; las lecturas posteriores, con **LOCAL_QUORUM**, consultan las réplicas y reconcilian

Dentro de cada nodo

Commit log, memtables, SSTables e índices locales

Procesamiento de las escrituras

Cuando llega una solicitud de escritura a un nodo, ocurren principalmente dos cosas:

- Se agrega al ***commit log*** en disco: garantiza la durabilidad (sobrevive incluso ante un fallo del nodo)
- Se envía a la ***memtable***: cuando se llena, los datos se vuelcan a una ***SSTable*** en disco, y el *commit log* se limpia

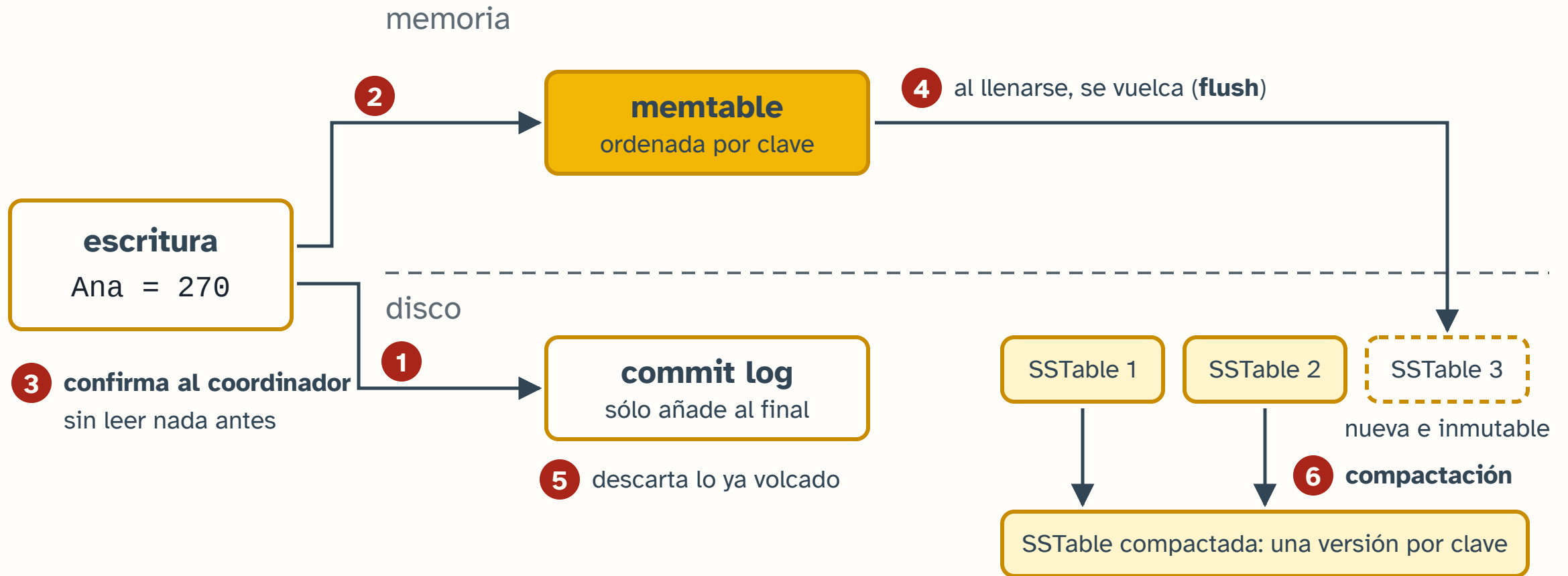
Recordatorio: *commit log*, *memtable* y *SSTables*



Ana está en la memtable (270) y en las SSTables (250, 260): **gana la más reciente**

La compactación fusionará las SSTables y descartará las versiones viejas

Procesamiento de las escrituras: el camino completo



Escribir no lee ni modifica ficheros: **sólo añade** al commit log y crea SSTables nuevas; por eso es tan rápido

Procesamiento de las lecturas

- Los filtros de Bloom identifican aproximadamente si una fila/columna existe en una SSTable
- Se mantienen por SSTable: cada una tiene un filtro de Bloom correspondiente en memoria

Un filtro de Bloom es una estructura de datos genérica para verificar si un elemento está presente en un conjunto o no

- Si la respuesta es «puede estar presente», Cassandra busca en la SSTable; si no está, continúa en otras SSTables
- Así ahorra muchas exploraciones innecesarias, y un gran coste de E/S de disco

Creación de índices

- En Cassandra los índices secundarios son complicados de usar y pueden afectar mucho al rendimiento
- La tabla de índices se almacena en cada nodo del clúster: una consulta con un índice secundario puede convertirse en una pesadilla de rendimiento si accede a varios nodos
- Regla general: indexar una columna con **baja cardinalidad** de pocos valores
- Actualmente este tipo de índices sufre de bajo rendimiento y alta latencia

SAI (*Storage-Attached Index*)

- Nuevo tipo de índice en Apache Cassandra (desde 5.0), con mejor rendimiento que los índices secundarios tradicionales
- Permite búsquedas eficientes en columnas no clave
- Soporta igualdad, rangos y `CONTAINS` en colecciones, pero no `LIKE`

Ventajas:

- Menor consumo de almacenamiento
- Soporte para múltiples tipos de datos
- Consultas más eficientes que con índices tradicionales

Consideraciones:

- Sigue siendo local a cada nodo: una consulta que no restringe la partición pregunta a todos
- Puede afectar el rendimiento en inserciones frecuentes

Índice SAI vs. clave de partición

Característica	Índice SAI	Clave de partición
Función principal	Búsquedas en columnas no clave	Distribución de datos en nodos
Impacto en almacenamiento	Crea estructuras adicionales	Define ubicación en el clúster
Rendimiento en consultas	Optimiza búsquedas secundarias	Muy rápido en clave de partición
Eficiencia en escrituras	Puede ralentizar inserciones	Altamente eficiente
Uso recomendado	Igualdad, rango y texto	Consultas rápidas y directas

Acceso desde Python

CQL con `cassandra-driver` : conexión, perfiles y operaciones básicas

Acceso a la base de datos

- Cassandra usa CQL (*Cassandra Query Language*) para interactuar con la base de datos
- Apache Cassandra cuenta con una *shell* de línea de comandos, `cqlsh`, sobre la cual se pueden invocar las consultas CQL
- Para el acceso desde Python contamos con la librería `cassandra-driver` (`pip install cassandra-driver`)
- Al conectarnos a un clúster, especificamos distintos **perfiles de ejecución** (`ExecutionProfile`), que permiten ejecutar consultas de forma diferente en una misma sesión

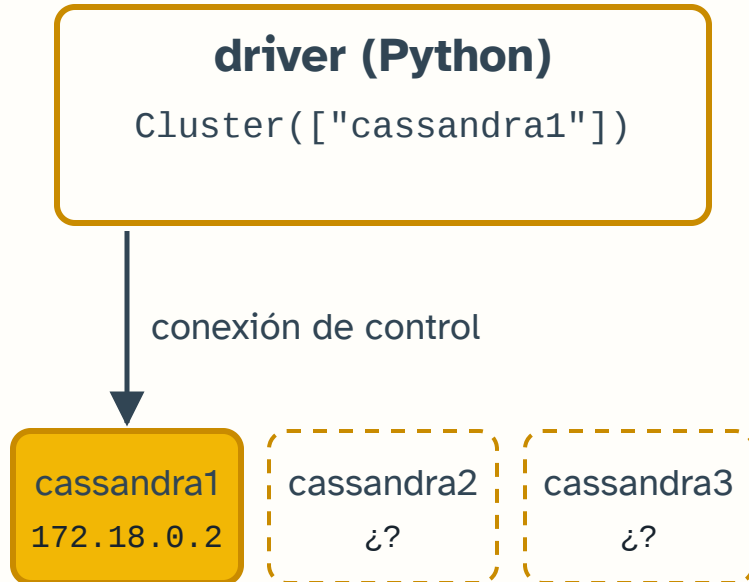
Conexión al clúster: puntos de contacto y perfiles

Conectarse a Cassandra no se parece a abrir una conexión a un servidor SQL: el *driver* mantiene un *pool* de conexiones **con todos los nodos que considera accesibles**, y decide, consulta a consulta, a cuál se la envía

- **Puntos de contacto** (`contact_points`): sólo se usan en el arranque. El *driver* abre la *control connection* contra uno de ellos y descubre el resto por *gossip* (`system.local` / `system.peers`)
- **Hosts** (`cluster.metadata.all_hosts()`): cada nodo descubierto, con dirección, *datacenter*, *rack* y estado. El *driver* se conecta a la dirección que **cada nodo publica** (`rpc_address` / `broadcast_rpc_address`), no a la de `contact_points`
- **Perfiles de ejecución** (`ExecutionProfile`): agrupan política de balanceo, consistencia, reintentos, *timeout* y `row_factory` ; cada consulta elige uno con `execution_profile=`

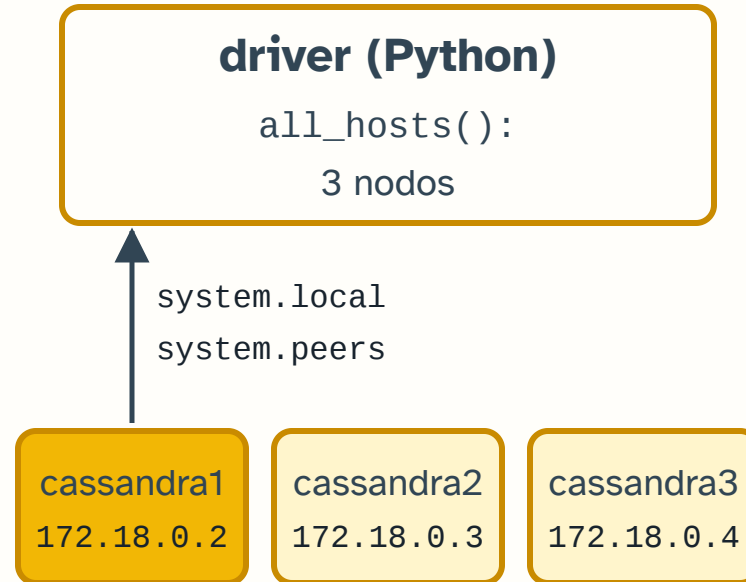
Conectar: arranque, descubrimiento y *pool*

1 Arranque



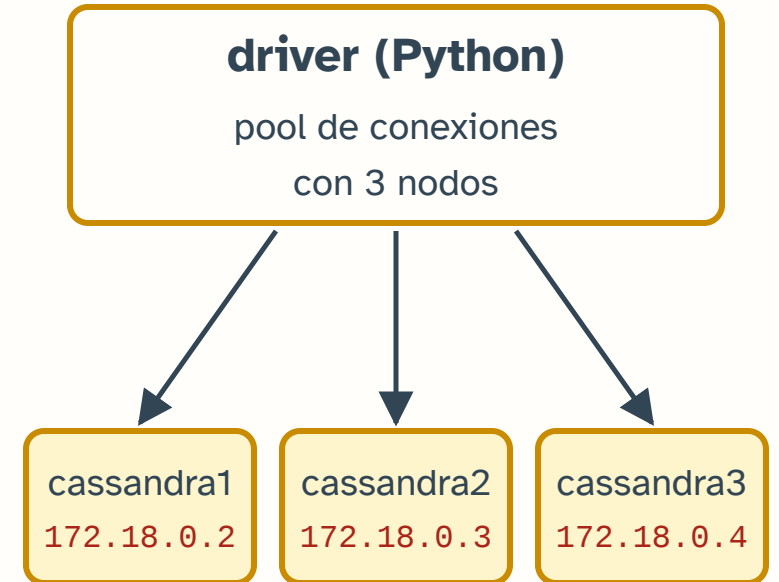
`contact_points` sólo sirve para arrancar
basta uno accesible; varios, por si falla

2 Descubrimiento



el nodo de contacto lista sus pares:
dirección, datacenter, rack y estado

3 Conexiones



cada conexión va a la dirección que
publica el nodo (`rpc_address`)

Qué nodo atiende cada consulta lo decide la política de balanceo del **perfil de ejecución**

Modelo antiguo frente a perfiles

```
cluster = Cluster(["cassandra1"], load_balancing_policy=...) # obsoleto
session.default_timeout = 40 # obsoleto
session.row_factory = dict_factory # obsoleto
```

Los dos modelos son **excluyentes**: en cuanto se pasa `execution_profiles=` al `Cluster`, cualquier atributo del modelo antiguo lanza un error:

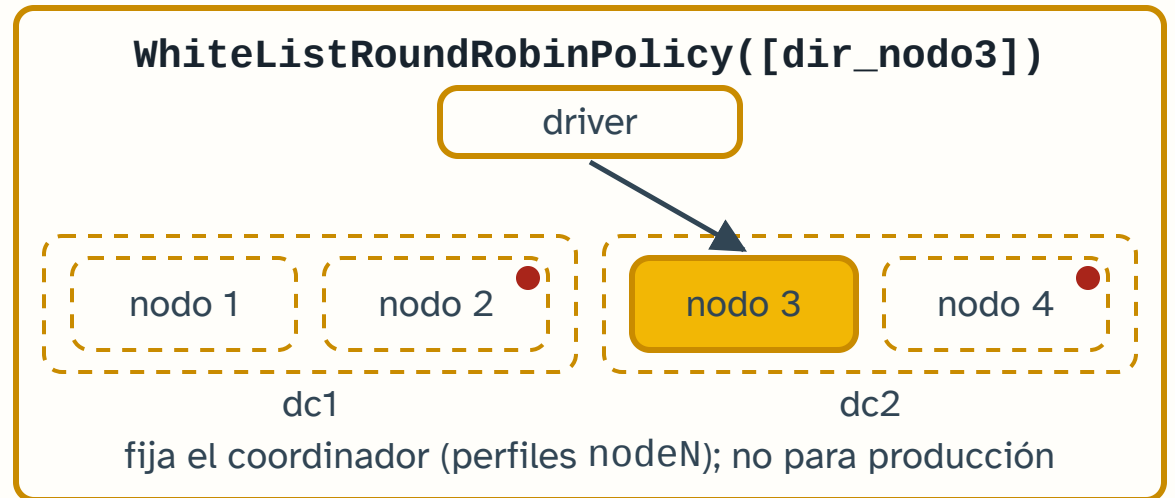
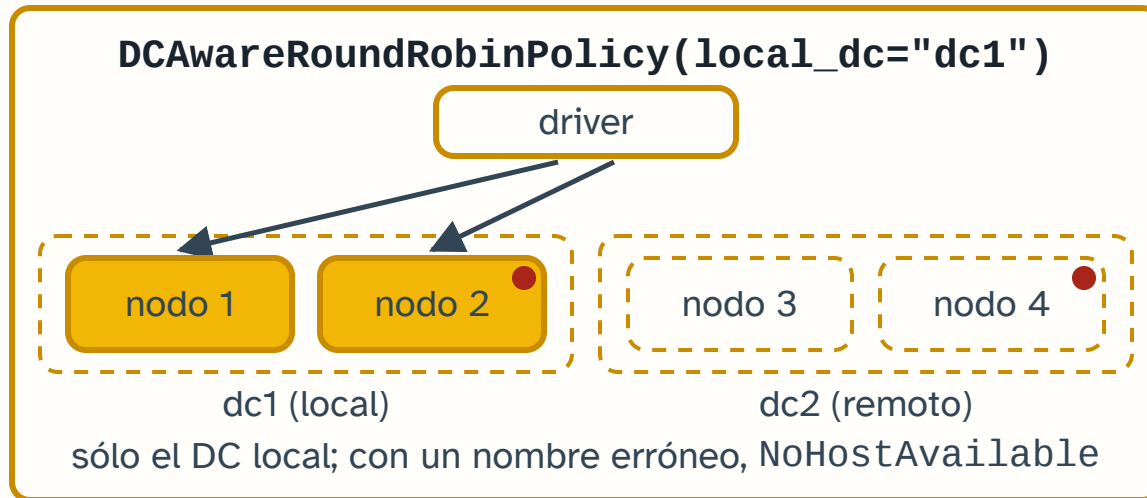
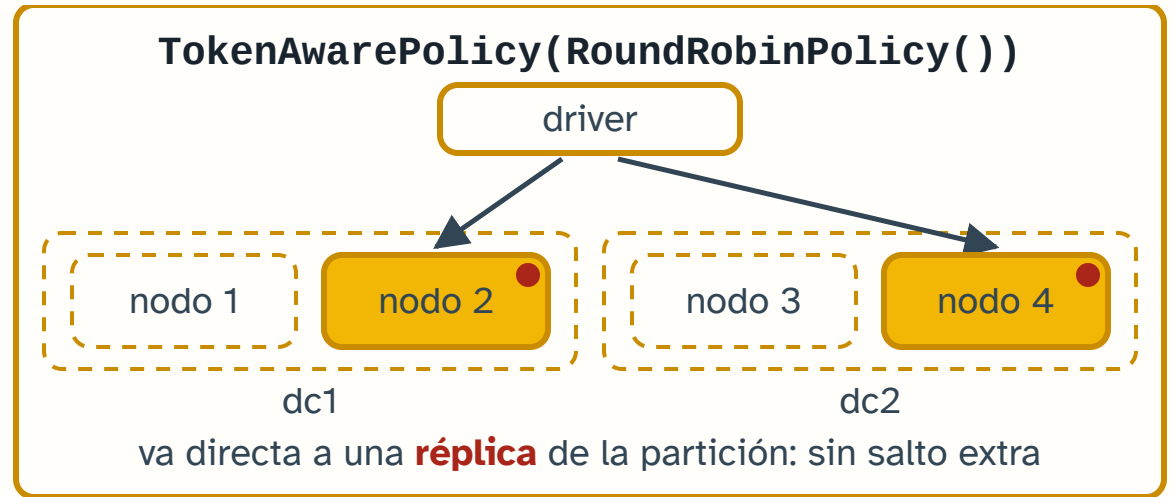
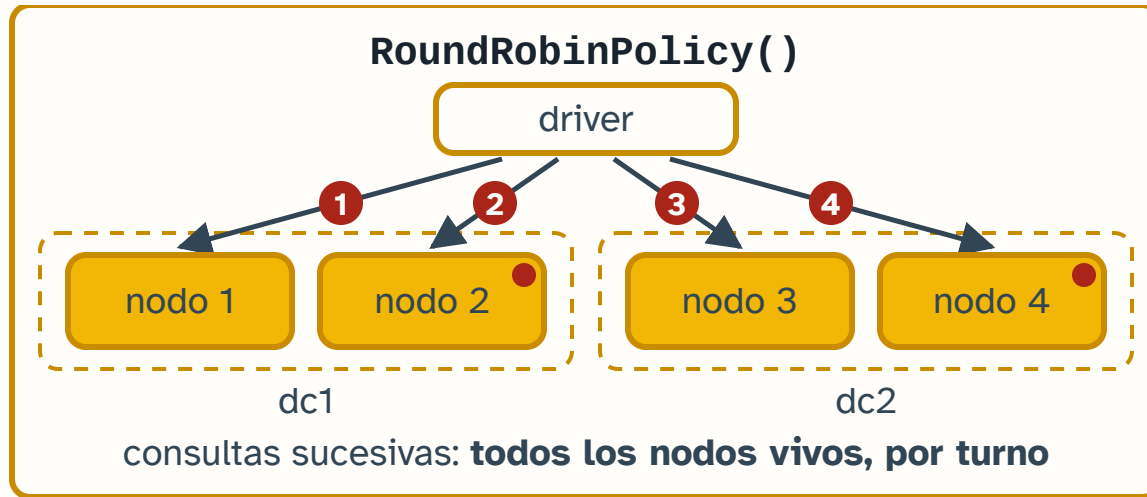
- `Cluster(..., load_balancing_policy=..., execution_profiles=...)` → `ValueError`
- `session.default_timeout = 40` → `ValueError: usar un perfil en su lugar`

El *timeout* pasa a ser `request_timeout` **dentro** del perfil; lo mismo para `row_factory` y el nivel de consistencia por omisión

Políticas de balanceo

- `RoundRobinPolicy` : reparte entre todos los nodos vivos
- `DCAwareRoundRobinPolicy(local_dc= ... )` : sólo usa los nodos del *datacenter* local
- `TokenAwarePolicy(política_hija)` : con sentencias preparadas, envía la petición directamente a una **réplica** de la partición, ahorrando un salto de coordinación
- `WhiteListRoundRobinPolicy` / `HostFilterPolicy` : restringen el conjunto de nodos utilizables — útiles para fijar consultas a un nodo (administración, depuración), **no** para producción: anulan el reparto de carga y el *token awareness*

Políticas de balanceo: quién puede coordinar

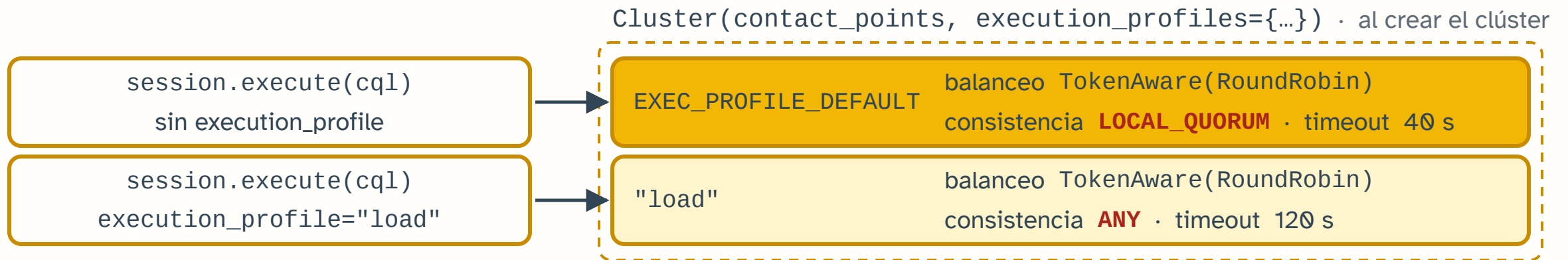


Crear los perfiles y conectar

```
from cassandra import ConsistencyLevel
from cassandra.cluster import EXEC_PROFILE_DEFAULT, Cluster, ExecutionProfile
from cassandra.policies import RoundRobinPolicy, TokenAwarePolicy
from cassandra.query import named_tuple_factory

query_profile = ExecutionProfile(    # perfil por omisión: consultas interactivas
    load_balancing_policy=TokenAwarePolicy(RoundRobinPolicy()),
    consistency_level=ConsistencyLevel.LOCAL_QUORUM,    # mayoría de réplicas; lee y reconcilia
    request_timeout=40.0,
    row_factory=named_tuple_factory,
)
load_profile = ExecutionProfile(    # carga masiva: ANY sólo existe para escritura
    load_balancing_policy=TokenAwarePolicy(RoundRobinPolicy()),
    consistency_level=ConsistencyLevel.ANY,
    request_timeout=120.0,
    row_factory=named_tuple_factory,
)
cluster = Cluster(
    contact_points,
    execution_profiles={EXEC_PROFILE_DEFAULT: query_profile, "load": load_profile},
)
session = cluster.connect()
```

Una sesión, varios perfiles



Modelo antiguo: session.default_timeout = 40 → **ValueError**

Con perfiles, timeout, consistencia y row_factory van **dentro** de cada perfil (request_timeout, consistency_level...)

Creación de `keyspaces`

```
factor = min(len(cluster.metadata.all_hosts()), 3)

session.execute(f"""
    CREATE KEYSPACE IF NOT EXISTS mi_keyspace
    WITH replication = {{'class': 'SimpleStrategy', 'replication_factor': {factor}}}
    """)
```

Genera un *keyspace* con `SimpleStrategy`, con un factor de replicación igual al número de nodos (hasta 3): la opción más sencilla para un único *datacenter*, pero **ignora la topología** del clúster

keyspaces en un clúster con varios DC

Con `GossipingPropertyFileSnitch`, cada nodo declara su *datacenter*. la estrategia adecuada es `NetworkTopologyStrategy`, con un factor de replicación **por DC** calculado a partir de los nodos ya descubiertos:

```
from collections import Counter

replicas_por_dc = Counter(host.datacenter for host in cluster.metadata.all_hosts())
factores = ", ".join(f"'{dc}': {min(n, 3)}" for dc, n in sorted(replicas_por_dc.items()))

session.execute(f"""
    CREATE KEYSPACE IF NOT EXISTS stackoverflow
    WITH replication = {{'class': 'NetworkTopologyStrategy', {factores}}}
    """)
```

`SimpleStrategy` queda desaconsejada en cuanto hay más de un *datacenter*

Creación de tablas

```
session.execute("""
    CREATE TABLE IF NOT EXISTS mi_keyspace.usuarios (
        nombre TEXT,
        apellidos TEXT,
        edad INT,
        dni TEXT,
        PRIMARY KEY (dni, apellidos)
    )""")
```

Tabla `usuarios` dentro de `mi_keyspace`, con `dni` como clave de partición y `apellidos` como columna de *clustering*

Inserción de datos

```
session.execute("""
    INSERT INTO mi_keyspace.usuarios (nombre, apellidos, edad, dni)
    VALUES ('Juan', 'Perez', 30, '11111111');
""")
session.execute("""
    INSERT INTO mi_keyspace.usuarios (nombre, apellidos, edad, dni)
    VALUES ('Ana', 'Terroso', 41, '22222222');
""")
```

Consultas de datos

```
select_query = session.prepare("SELECT * FROM mi_keyspace.usuarios")
rows = session.execute(select_query)
for row in rows:
    print(row)
```

Filtrar por la clave de partición (`dni`) es directo y eficiente, y **siempre** debe usarse una sentencia preparada (evita inyección CQL):

```
select_por_dni = session.prepare(
    "SELECT * FROM mi_keyspace.usuarios WHERE dni = ?"
)
for row in session.execute(select_por_dni, ("11111111",)):
    print(f"DNI: {row.dni}, Nombre: {row.nombre}, Apellidos: {row.apellidos}, Edad: {row.edad}")
```

Filtrar por `edad` (que no es la clave primaria) exige un índice, como se ve a continuación, o `ALLOW FILTERING` — que recorre todo el clúster y debe evitarse en producción

Creación de índices SAI

Para crear índices SAI, `CREATE INDEX` junto con `USING 'sai'`:

```
CREATE INDEX apellidos_sai_idx ON mi_keyspace.usuarios (apellidos)
USING 'sai'
WITH OPTIONS = {'case_sensitive': 'false', 'normalize': 'true', 'ascii': 'true'};
```

- `case_sensitive: false` : las búsquedas no distinguen mayúsculas/minúsculas
- `normalize: true` : normalización Unicode durante las búsquedas
- `ascii: true` : convierte caracteres especiales a su equivalente ASCII

Más opciones: `create-custom-index`

Construcción de índices en segundo plano

- `execute()` sobre un `CREATE INDEX` sólo confirma que el cambio de esquema **se propagó** por el clúster, no que el índice ya esté construido
- El *build* —recorrer las SSTables existentes y poblar el índice— es un proceso aparte, en segundo plano
- Con varios índices construyéndose a la vez sobre tablas de tamaños distintos, un `time.sleep(N)` fijo puede terminar antes de que estén disponibles: una consulta que use el índice fallaría con `IndexNotAvailableException`
- Mejor: sondear `system_views.sstable_tasks` (equivalente CQL de `nodetool compactionstats`, desde Cassandra 4.0) hasta que no queden tareas de tipo *Secondary index build*, y confirmar con una consulta real de bajo coste

Construcción de índices: sondeo

```
import time

def build_en_marcha(session, keyspace: str) → bool | None:
    """None si system_views no está disponible (plan B: reintentar la query real)."""
    try:
        filas = session.execute(
            "SELECT kind FROM system_views.sstable_tasks WHERE keyspace_name = %s",
            (keyspace,),
        )
    except Exception:
        return None
    return any("index" in (fila.kind or "").lower() for fila in filas)

while build_en_marcha(session, "mi_keyspace"):
    time.sleep(2)
```

Actualización y eliminación de datos

```
update_query = session.prepare("""
    UPDATE mi_keyspace.usuarios SET edad = ? WHERE dni = ? AND apellidos = ?
""")
session.execute(update_query, (35, '11111111', 'Perez'))

delete_query = session.prepare("""
    DELETE FROM mi_keyspace.usuarios WHERE dni = ? AND apellidos = ?
""")
session.execute(delete_query, ('22222222', 'Terroso'))
```

Al terminar, es importante cerrar la conexión con la base de datos:

```
session.shutdown()
```

Escrituras, borrados y vistas

Lo que cuesta cada escritura y cómo consultar fuera de la clave

Un borrado no borra: escribe una lápida

- Las SSTables son inmutables: no se puede quitar una fila de un fichero ya escrito
- `DELETE` escribe una marca, la **tombstone** (lápida), con su marca de tiempo; cada lectura debe leerla y saltarse lo que tapa
- La **compactación** fusiona SSTables y sólo descarta la lápida pasado `gc_grace_seconds` (10 días por omisión), para que llegue a todas las réplicas
- También dejan lápida un `null` escrito de forma explícita, una asignación completa de una colección y un dato con TTL caducado
- Muchas lápidas degradan las lecturas: Cassandra avisa (`tombstone_warn_threshold`) y acaba abortando la consulta (`tombstone_failure_threshold`)

Por eso el cargador de las prácticas no escribe los nulos (`UNSET_VALUE`) y las colecciones se rellenan con `col = col + ?`

TTL y marca temporal

```
INSERT INTO sesiones (user_id, token) VALUES (24, 'abc') USING TTL 3600;
```

```
SELECT token, TTL(token), WRITETIME(token)  
FROM sesiones WHERE user_id = 24;
```

- `USING TTL n`: la celda caduca a los `n` segundos y pasa a ser una lápida
- El TTL se aplica **por celda**: una fila puede quedar a medio caducar
- `WRITETIME()` devuelve la marca de tiempo de la escritura; `TTL()`, los segundos que le quedan
- Cada celda lleva su marca: ante dos versiones, **gana la última escrita** (*last write wins*), que es lo que usa la reparación de lecturas

Toda escritura es un *upsert*

- `INSERT` y `UPDATE` hacen lo mismo: escriben las celdas indicadas, existiera la fila o no
- Un `INSERT` sobre una clave que ya existe **no da error**: sustituye en silencio (por eso la clave de `Answers` debe ser única)
- `IF NOT EXISTS` convierte la escritura en **condicional** (*lightweight transaction*): sólo se aplica si la fila no existe

```
INSERT INTO actividad (user_id, event_id, payload)
VALUES (24, 7f3a0c1e-5d6b-4c21-9a47-2b8e1d90a3c4, 'primero')
IF NOT EXISTS USING TTL 300;
-- [applied] = true la primera vez, false las siguientes
```

- Usa el protocolo **Paxos**, con varios viajes entre réplicas: es correcta, pero costosa, y no debe ser el camino normal de escritura

BATCH: atomicidad, no rendimiento

Un **BATCH** agrupa escrituras para que se apliquen **todas o ninguna**, no para ir más rápido:

Escribe en	Qué ocurre	Coste
Una sola partición	Una única mutación en sus réplicas: atómica y aislada	Como una escritura normal
Varias particiones (<i>logged</i>)	Se guarda antes en el <i>batchlog</i> , que garantiza que acabarán aplicándose; sin aislamiento	Más que las escrituras sueltas

- Uso típico: mantener sincronizadas una tabla y sus copias desnormalizadas
- Un lote con muchas particiones sobrecarga al coordinador: Cassandra avisa y rechaza lotes grandes (`batch_size_warn_threshold`)

El tipo COUNTER

```
CREATE TABLE contador_visitas (  
  post_id bigint PRIMARY KEY,  
  visitas counter  
);
```

```
UPDATE contador_visitas SET visitas = visitas + 1 WHERE post_id = 1003;
```

- Se modifica sólo con incrementos y decrementos: no hay `INSERT` de un valor
- Una tabla con un `counter` sólo admite otras columnas `counter` además de la clave primaria
- No es idempotente: reintentar un incremento fallido puede contarlo dos veces
- No admite el nivel `ANY`

Vistas materializadas

```
CREATE MATERIALIZED VIEW questions_by_author AS
  SELECT * FROM Questions
  WHERE OwnerUserId IS NOT NULL AND CreationDate IS NOT NULL AND Id IS NOT NULL
  PRIMARY KEY (OwnerUserId, CreationDate, Id);
```

- Otra tabla, con **otra clave de partición**, que mantiene Cassandra cada vez que se escribe en la tabla base
- Su clave debe contener todas las columnas de la clave de la base (`Id` , `CreationDate`) y, como mucho, una más (`OwnerUserId`)
- Se actualiza de forma asíncrona: puede ir por detrás de la tabla base
- Está desactivada por omisión (`materialized_views_enabled`) y se considera experimental

¿Vista materializada o tabla escrita por la aplicación?

	Vista materializada	Tabla manual
Quién la mantiene	Cassandra	La aplicación (con BATCH o escrituras repetidas)
Forma de la clave	Limitada: la de la base más una columna	Libre
Si falla a medias	Poco control sobre errores y reintentos	Control total
Coste de escritura	El mismo: duplica la escritura	El mismo: duplica la escritura

- En ambos casos la duplicación es deliberada: se paga en escritura y en espacio para leer de una sola partición

Consultar fuera de la clave: opciones

Mecanismo	Sirve para	Coste principal
Índice secundario clásico	Igualdad con baja cardinalidad	Local: pregunta a muchos nodos
Índice SAI	Rangos y varias condiciones sobre una tabla	Local; escritura y compactación más lentas
Vista materializada	Otra clave de partición, mantenida por Cassandra	Escritura duplicada y asíncrona
Tabla dirigida por la consulta	La consulta conocida y frecuente	Escritura duplicada y a mano
ALLOW FILTERING	Pruebas y tablas pequeñas	Recorre filas para descartarlas

Un mecanismo general amplía lo que se puede consultar, pero no sustituye al modelado dirigido por consultas

Paginación

- Una consulta grande no llega de golpe: `fetch_size` fija cuántas filas trae cada página
- El *driver* pide la siguiente página al seguir iterando el resultado
- Para un «ver más» entre peticiones distintas, el servidor devuelve un `paging_state` opaco que se pasa con la **misma** consulta para continuar
- A diferencia de `OFFSET` en SQL, reanudar no obliga a leer y descartar las filas anteriores
- El `paging_state` puede contener valores de la clave: no se entrega sin protección a clientes no fiables

Casos de uso y conclusiones

- Entre sus limitaciones: no hay `JOIN`, y las agregaciones (`COUNT`, `AVG`, `GROUP BY`) sólo son razonables dentro de una partición — para agregar sobre todo el clúster hay que precomputar los datos
- La ventaja principal es su velocidad de escritura y la forma de distribuir los datos, que permite alta escalabilidad
- Las lecturas por clave de partición son rápidas y predecibles; las que no la conocen (índices, `ALLOW FILTERING`) no lo son, y suelen exigir otra tabla
- Usos más comunes:
 - **Aplicaciones IoT:** miles de sensores enviando información que debe sincronizarse y correlacionarse en tiempo real
 - **Monitorización de aplicaciones web:** almacenar la actividad de los usuarios en sitios web

Referencias

- Jim R. Wilson, Eric Redmond. **Seven Databases in Seven Weeks**
- Russell Bradberry, Eric Lubow. **Practical Cassandra — A Developer's Approach**