

UNIVERSIDAD NACIONAL DE INGENIERÍA

Arquitectura de Software

Sharding, cache y escalabilidad

La clave de sharding · Rebalanceo y hot spots · Cache-aside · Invalidación · Estampida · Servicios sin estado

Gerson Garrido

Semana 11

Agenda

Sesión A: sharding

1. Replicar y hacer sharding
2. La clave de sharding
3. Por rango y por hash
4. Hot spots y rebalanceo
5. Lo que se pierde

Sesión B: cache

6. Qué es una cache y qué abre
7. Cache-aside e invalidación
8. Estampida
9. Depender de la cache
10. Escalar hacia los lados

Lectura

A: Kleppmann y Riccomini, *Designing Data-Intensive Applications* (2.ª ed.), cap. 7 *Sharding*; Azure Architecture Center, *Sharding pattern*.
B: Azure, *Cache-Aside pattern*; *Site Reliability Engineering*, cap. 22 (arranque lento y caches frías); *Software Architecture in Practice*, cap. 9.

SECCIÓN 01

Replicar y hacer sharding

01



Lo que la réplica no resolvió

La semana pasada **copiamos** los datos. Sirvió para:

- seguir funcionando si una máquina falla;
- atender más lecturas.

Y no sirve cuando:

- no caben las escrituras: cada réplica las recibe todas;
- no cabe el volumen: cada réplica lo guarda todo;
- una lectura es cara: la réplica solo la manda a otra máquina.

Esta semana

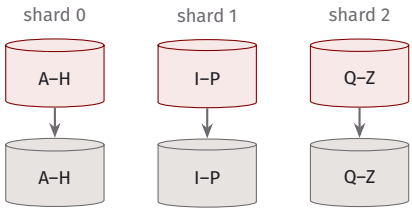
Sharding: repartir los datos. La táctica más cara.

Cache: guardar las respuestas. La más barata de poner, y la que más problemas sutiles trae.

Los mismos datos o datos distintos



Replicación: todos tienen todo



Sharding: cada uno tiene una parte

	Replicación	**Sharding**
Si una máquina se pierde	otra tiene lo mismo	esa parte no está
Más escrituras, más volumen	no	sí

Se combinan: cada shard tiene sus réplicas.

Cuándo hace falta: al final de una lista

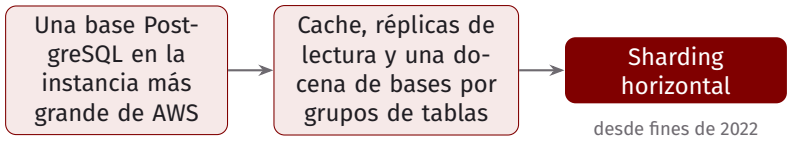
Cuando los datos o las escrituras **ya no caben en una máquina**. Antes:

1. Mejores consultas e índices
2. Una máquina más grande
3. Archivar lo viejo
4. Réplicas de lectura y cache
5. Separar por dominio
6. Sharding

Por qué es la más cara

Es la única que cambia lo que la base de datos garantiza: ya no hay `JOIN`, transacción ni `UNIQUE` entre dos filas cualesquiera. Y casi no tiene vuelta atrás.

En la industria: Figma recorrió la lista



- Tablas de varios terabytes y miles de millones de filas: demasiado para una base de datos.
- Hacer sharding de la **primera** tabla les tomó unos nueve meses.

Sammy Steele, *How Figma's databases team lived to tell the scale* (2024)

SECCIÓN 02

La clave de sharding

02

La clave decide todo

Clave de sharding

El atributo de la fila que la regla mira para decidir a qué shard va.

- Las filas con la misma clave **viven juntas**: entre ellas sigue habiendo `JOIN` y transacciones.
- Una consulta que trae la clave va a **un shard**. Una que no, les pregunta **a todos**.
- La carga de un shard es la suma de la carga de sus claves.

Cómo se elige

Por el patrón de acceso dominante y por qué filas se escriben juntas. Un sistema con muchas organizaciones: por *tenant*. Tinkuy: por `event_id`.

Tinkuy con `event_id`: ¿un shard o todos?

```
make scatter

Clave de sharding: event_id, 8 shards.

endpoint                    la consulta filtra por                          shards
GET /events/:id             WHERE event_id = $1                             1
GET /events/:id/seats       WHERE event_id = $1                             1
POST /orders                WHERE t.id = ANY($1), con el eventId del pedido 1
GET /reports/sales?eventId= WHERE o.event_id = $1                           1
GET /events                 sin filtro: todos los eventos                   8 (scatter-gather)
GET /orders/:id             WHERE o.id = $1                                 8 (scatter-gather)
```

- Ver, elegir asientos y comprar: un shard. La compra sigue siendo una transacción común.
- La clave tiene que viajar con cada pedido: llega hasta el diseño de la API.

Otra clave: el id del asiento

```
make shards

Otra clave: shard = hash(seat_id) mod 8
Asientos por shard: 1599, 1621, 1624, 1622, 1617, 1668, 1705, 1624
...
```

El reparto es casi perfecto, y el sistema quedó peor:

- los asientos del evento 1 quedan en los 8 shards: pedir su mapa es preguntarles a todos;
- de las 971 órdenes de ejemplo con más de un asiento, en 912 los asientos caen en shards distintos: serían transacciones entre shards.

Idea clave. Una clave no se juzga por lo parejo que reparte las filas, sino por cuántas operaciones deja dentro de un solo shard.

SECCIÓN 03

Por rango y por hash

03

Dos reglas para ir de la clave al shard

Por rango

Cada shard, un tramo continuo de claves.

- Las vecinas quedan juntas: una consulta por rango lee uno o pocos shards.
- Una clave que crece con el tiempo manda todas las escrituras al último shard.

Por hash

El shard sale de hash(k).

- Claves consecutivas van a shards distintos.
- Se pierde el orden: una consulta por rango les pregunta a todos.

La función tiene que repartir parejo y ser **estable**: la misma respuesta en toda máquina y toda versión. No sirve el hash interno de un lenguaje.

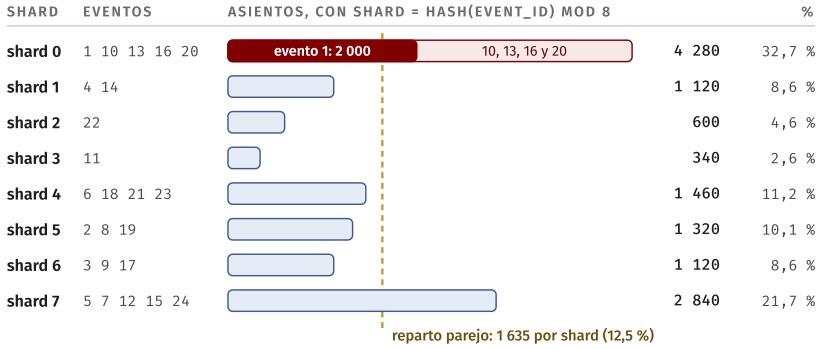
Los datos de Tinkuy en ocho shards, por hash

make shards

```
24 eventos y 13080 asientos en 8 shards. Clave de sharding: event_id.

Por hash: shard = hash(event_id) mod 8
shard  eventos                 asientos  % de los asientos
    0  1 10 13 16 20               4280    32,7 %
    1  4 14                        1120     8,6 %
    2  22                           600     4,6 %
    3  11                           340     2,6 %
    4  6 18 21 23                 1460    11,2 %
    5  2 8 19                     1320    10,1 %
    6  3 9 17                     1120     8,6 %
    7  5 7 12 15 24               2840    21,7 %
El shard más cargado tiene 4280 asientos y el menos cargado 340. El evento 1 está en el shard 0.
```

Los ocho shards, a escala



El shard 0 tiene más de doce veces los asientos del shard 3 (4 280 entre 340).

El evento 1 es el 15,3 % de los asientos y ninguna regla lo divide: todas sus filas tienen la misma clave.

Tres cosas para mirar

1. **El hash no repartió parejo.** Hay 24 claves. Un hash reparte parejo en promedio y con muchas claves: con el id del asiento (13 080 valores) dejó entre 1 599 y 1 705 por shard.
2. **Por rango**, de tres en tres, quedó entre 1 260 y 2 720 asientos por shard. Y los eventos nuevos, que son los que venden, caerían siempre en el último.
3. **El evento 1 está entero en un shard, con las dos reglas.** Tiene 2 000 de los 13 080 asientos, y ninguna regla lo puede dividir: todas sus filas tienen la misma clave.

SECCIÓN 04

Hot spots y rebalanceo

04

Hot spot: una clave que concentra la carga



`make load-sale`: los 200 compradores van a un solo evento.

- Todas las filas de una clave van al mismo shard **por definición**.
- Ninguna función de hash lo arregla.
- En una plataforma de entradas no es un caso raro: el concierto que se agota en un minuto es el negocio.

Error común

«Usamos hash, así que no hay hot spots». El hash reparte claves, no carga.

Qué se hace con una clave caliente

Opción	Qué da	Qué cuesta
Aislarla en su propio shard	El evento grande no degrada a los demás	No divide la carga; el enrutamiento necesita una tabla con excepciones
Partirla con un sufijo: `(event_id, sección)`	Las escrituras se dividen	Leer el evento es preguntarles a varios; comprar en dos secciones cruza shards
Tratarla en la puerta	Los compradores entran al ritmo que un shard aguanta	Una fila de espera: *load shedding* y *backpressure* *(semana 14)*

Rebalanceo: por qué no hash(k) mód n

Agregar un nodo a n nodos parejos exige mover, como mínimo, $\frac{1}{n+1}$ de las claves.

Con $h = \text{hash}(k)$, la clave se queda solo si $h \bmod n = h \bmod (n+1)$. Sea $r = h \bmod n(n+1)$: se cumple solo cuando $r < n$.

$$P(\text{se queda}) = \frac{n}{n(n+1)} = \frac{1}{n+1} \qquad P(\text{se mueve}) = \frac{n}{n+1} = \frac{8}{9} \approx 88{,}9\ \% \ \text{ con } n = 8$$

De 8 a 9 nodos

Se mueven ocho de cada nueve claves, cuando bastaba mover una. Y casi todas van de un nodo viejo a otro nodo viejo.

Medido con 100 000 claves

```
make rebalance

...
esquema                se mueven   van al nodo nuevo   claves por nodo después (mín. – máx.)
hash(k) mod n           88,8 %     11,2 %             10897 – 11228
1024 shards lógicos     11,0 %     11,0 %             10964 – 11319
consistent hashing      12,3 %     12,3 %             9710 – 12472

Y con hash(k) mod n, duplicar de 8 a 16 nodos mueve 50,2 % de las claves.
```

- hash(k) mód n solo crece bien **duplicando**: se queda la mitad, y el resto va a nodos nuevos.
- Las otras dos mueven cerca del mínimo, y **solo hacia el nodo nuevo**.

De 8 a 9 nodos, clave por clave

1. hash(k) mod n | **2. 1 024 shards lógicos**

`shard = hash(k) mod 1 024, fijo` — el mapa de shards cambia: 113 de los 1 024 pasan al nodo 8

se queda · cambia entre nodos viejos · va al nodo nuevo

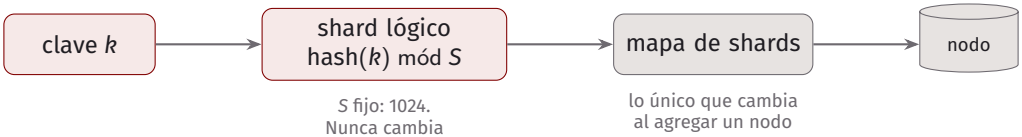


Solo se mueve lo que recibe el nodo nuevo: es el mínimo.

MAKE REBALANCE: 100 000 CLAVES, DE 8 A 9 NODOS — mínimo: 1/9 = 11,1 %

esquema	se mueven	van al nodo nuevo
`hash(k) mod n`	88,8 %	11,2 %
`1024 shards lógicos`	11,0 %	11,0 %

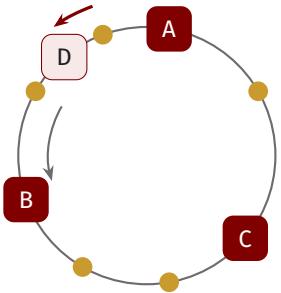
Shards lógicos: separar el shard del nodo



- Una clave nunca cambia de shard. Lo que se mueve son **shards enteros**.
- Se paga con el mapa: un dato en el que todos los que enrutan tienen que estar de acuerdo.
- Notion: 480 shards lógicos en 32 bases de datos. 480 tiene muchos divisores: pueden pasar a 40 y a 48.

Garrett Fidalgo, *Herding elephants: lessons learned from sharding Postgres at Notion* (2021)

Consistent hashing: nodos y claves en un anillo



Cada clave es del primer nodo que encuentra al avanzar.

- No hay mapa escrito: basta la lista de nodos.
- Un nodo nuevo se queda con un tramo. Los demás tramos no se enteran.
- Cada nodo ocupa muchos puntos, para que su parte sea la suma de muchos tramos pequeños.

Trade-off

Shards lógicos: reparto exacto y control, a cambio de un mapa. Consistent hashing: sin mapa, con un reparto aproximado (12,3 % movido, no 11,1 %).

Quién aprieta el botón y quién enruta

Rebalanceo automático o manual

- Mover datos es carga adicional.
- Un rebalanceo automático que se dispara por un nodo lento le agrega carga al nodo que menos la aguanta: falla en cascada.

Quién sabe dónde está cada shard

- El cliente *(Notion: en la aplicación)*
- Una capa de enrutamiento *(Figma: DBProxy)*
- Cualquier nodo, que reenvía

En los tres casos

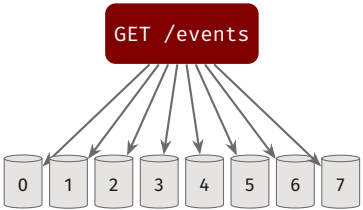
Hace falta un registro de la asignación en el que todos estén de acuerdo.

SECCIÓN 05

Lo que se pierde

05

Consultas entre shards: scatter-gather



Sin la clave, a todos; y después, juntar.

- La latencia es la del shard **más lento**.
- La disponibilidad es la de **todos juntos**.
- No escala: agregar shards le agrega trabajo.

En la industria

Figma: cada scatter-gather aporta la misma carga que si la base de datos no tuviera sharding. Por eso limitaron el SQL que su capa de enrutamiento acepta.

Transacciones, unicidad e identificadores

- **Transacciones entre shards**: dos bases de datos, y ninguna promete nada sobre la otra *(semana 12)*. La clave se elige para que lo que se escribe junto viva junto.
- En Tinkuy, una orden es de un solo evento: con `event_id`, la compra entera vive en un shard.
- `tickets.code UNIQUE`: cada shard lo garantiza *en ese shard*. Entre dos, nadie.
- Ids `GENERATED AS IDENTITY`: ocho shards, ocho contadores que empiezan en 1.

Lo que cambia

Garantías que la base de datos daba sola pasan a ser trabajo de la aplicación.

Índices secundarios: locales o globales

Índice local

Cada shard indexa sus filas.

- Escribir: barato, en la misma transacción.
- Leer por ese atributo: scatter-gather.

Índice global

Una estructura aparte que cubre todos los shards.

- Leer: barato.
- Escribir: toca dos lugares. Casi siempre se actualiza después, de forma asíncrona.

«Las órdenes de `rosa.quispe@correo.pe`»: con uno, les pregunta a los ocho shards; con el otro, puede no ver la orden de hace un segundo.

Error común: sharding por adelantado

«Para cuando crezcamos»

La clave se elige por el patrón de acceso, y un sistema nuevo todavía no conoce el suyo. Una clave equivocada cuesta más que no tener sharding.

El artículo de Notion dice las dos cosas:

- el sharding prematuro puede restringir el producto antes de que el negocio lo defina;
- y entre sus lecciones: *shard earlier*. Esperaron tanto que migrar se volvió difícil.

Cuándo: no antes de conocer los patrones de acceso, y no después de quedarse sin margen para migrar.

SECCIÓN 06

Qué es una cache y qué abre

06

Una cache es un dato derivado

Cache

Guarda el resultado de algo caro para no repetirlo. Todo lo que tiene se puede recalcular desde el **origen**.

Lo que da

- Menos latencia.
- Menos carga sobre el origen.

Lo que abre

- Una copia que puede estar vieja.
- Un componente más que falla.

Dos reglas

Se puede perder: sin ella, el sistema responde bien, más lento. **Puede estar equivocada**: la pregunta es por cuánto tiempo y quién lo nota.

Dónde puede haber una cache



- Cerca del usuario: lo más rápido, y lo más difícil de invalidar.
- En el proceso: sin red, una copia por instancia, se pierde al reiniciar.
- Compartida: es la de esta clase.

Qué conviene guardar

Lo que cumple las tres a la vez:

- se lee mucho más de lo que cambia;
- es caro de calcular;
- tolera estar un poco viejo *(una pregunta de negocio)*.

En Tinkuy		
La lista de eventos y sus precios	Sí	Todos la piden, cambia poco, es la lectura más lenta
Asientos libres en plena venta	Con cuidado	Cambia con cada compra; la compra vuelve a comprobar
El estado de un pago	No	Nadie decide con un dato que puede estar viejo

Para saber si es cara, se mide

Tinkuy inicial: make load-browse

```
duration_event_detail..........: avg=44.37ms  min=12.22ms med=31.31ms p(90)=79.51ms  p(95)=124.86ms p(99)=227.07ms max=272.86ms
duration_events_list...........: avg=384.38ms min=91.79ms med=310.5ms p(90)=780.7ms  p(95)=889.33ms p(99)=944.32ms max=969.19ms
http_reqs......................: 1773   56.617729/s
```

En una laptop con Apple Silicon, dentro de Docker. Los números cambian entre corridas; la relación entre los dos endpoints, no.

La lista hace una consulta y una más por cada evento: un N+1 *(semana 6)*.

Error común

Poner una cache para esconder una consulta mal escrita. La consulta sigue ahí, y aparece entera cada vez que la entrada vence.

SECCIÓN 07

Cache-aside e invalidación

07



Cache-aside: la cache está al lado

La aplicación habla con los dos: con la cache y con el origen.

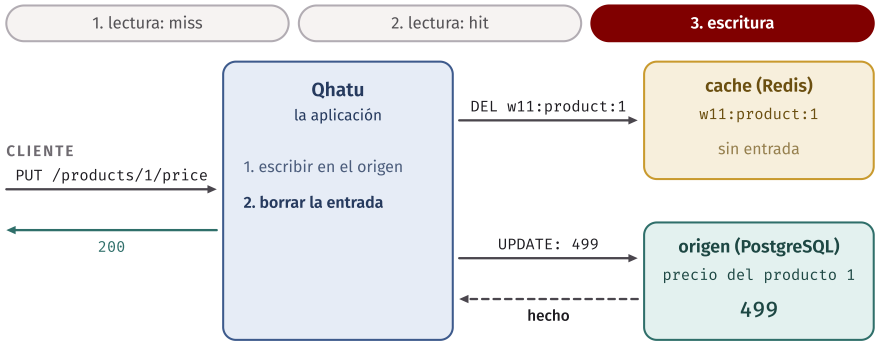
Leer

1. Buscar en la cache: si está, es un *hit*.
2. Si no, *miss*: leer del origen.
3. Guardar con un tiempo de vida y responder.

Escribir

1. Escribir en el origen.
2. Borrar la entrada.

Cache-aside, paso a paso



La próxima lectura no la encuentra, va al origen y trae 499.

El patrón, en código

src/cache.ts

```ts
export async function cacheAside<T>(cache, key, ttlMs, load) {
  const fullKey = cache.prefix + key;
  // 1. Buscar en la cache.
  const raw = await safely(cache, () => cache.redis.get(fullKey), null);
  if (raw !== null) {
    cache.stats.hits++;
    return decode<T>(raw).v;
  }
  // 2. No está: leer del origen.
  cache.stats.misses++;
  const value = await load();
  // 3. Guardar y responder. Siempre con TTL.
  await safely(cache, () => cache.redis.set(fullKey, encode(value), { PX: ttlMs }), null);
  return value;
}
```

Resumido: sin los tipos y sin la cache negativa. El archivo completo está en `code/week-11/`.

Qhatu: el ejemplo de la semana

El catálogo de una bodega en línea. El stock es la **suma** de los movimientos: la lista recorre 80 100 filas.

con Qhatu corriendo

```
$ curl -s -o /dev/null -w '%{time_total}\n' localhost:3000/products
0.067759
$ curl -s -o /dev/null -w '%{time_total}\n' localhost:3000/products
0.004028
$ redis-cli --scan --pattern 'w11:*'
w11:products
w11:product:1
$ redis-cli ttl w11:product:1
28
$ curl -s localhost:3000/stats
{"cache":{"hits":2,"misses":2,"negativeHits":0,"staleServed":0,"errors":0},"originReads":2}
```

Medido: sin cache y con cache

make compare: 50 usuarios, 15 segundos

```
== CACHE_MODE=off
duration_list: med=17.58ms  p(95)=39.99ms  p(99)=71.84ms  max=119.35ms
lecturas que llegaron a PostgreSQL: 6708   hits: 0   misses: 0
== CACHE_MODE=aside
duration_list: med=581.45µs p(95)=1.89ms   p(99)=85.82ms  max=207.03ms
lecturas que llegaron a PostgreSQL: 275   hits: 7115   misses: 275
```

- La carga sobre el origen: de 6 708 consultas a 275. Es lo estable entre corridas.
- La mediana: unas treinta veces menor.
- El p99 subió. Los que llegan cuando la entrada venció pagan el origen completo, y más.

Una cache mejora la mediana mucho más que la cola.

Otras formas de combinar cache y origen

Patrón	Quién habla con el origen	Qué se paga
Cache-aside	La aplicación, al leer y al escribir	Hay que acordarse de invalidar
Read-through	La cache, cuando le falta algo	La cache tiene que saber leer del origen
Write-through	La cache, antes de confirmar la escritura	Cada escritura paga dos
Write-behind	La cache, *después* de confirmar	Si la cache se cae antes, **la escritura se pierde**

Write-behind

Por un rato, el único lugar donde existe el dato es la cache: dejó de ser un dato derivado.

Invalidación: dos mecanismos, y se usan los dos

Por tiempo: TTL

La entrada se borra sola pasado *T*.

- Nada tiene más de *T* de viejo.
- *T* sale del negocio: cuánto retraso tolera *ese* dato.

Por evento: borrar al escribir

Cuando el dato cambia, se borra su entrada.

- Exacto, si nadie se olvida en ningún lugar donde se escribe.
- Primero el origen, después la cache.

Error común

Guardar sin decidir cómo se invalida: un TTL al azar y datos viejos sin que nadie sepa por cuánto tiempo.

Sin invalidación: datos viejos

Qhatu con INVALIDATE=off

```
$ curl -s -X POST localhost:3000/products/1/sales -d '{"quantity": 5}'
{"sold":5}
$ curl -s localhost:3000/products/1
{"product":{"id":1,"name":"Arroz extra 250 g","category":"abarrotes","priceCents":455,"stock":1663}}
$ psql -d arq_w11 -c 'SELECT sum(quantity) AS stock FROM stock_movements WHERE product_id = 1'
 stock
-------
  1658
(1 row)
```

- PostgreSQL dice 1 658 y la API dice 1 663, hasta que venza el TTL.
- Una venta deja viejas **dos** entradas: el detalle del producto y la lista.

Por qué borrar y no actualizar

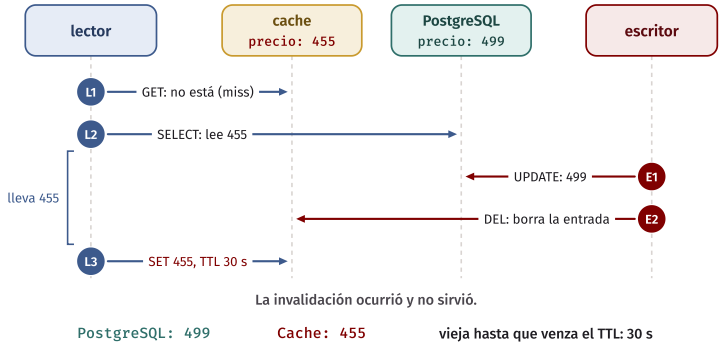
dos escritores: A pone 470 y B pone 499

```
$ node race.ts --writers --update A1 B1 B2 A2
A1  UPDATE en PostgreSQL: el precio ahora es 470
B1  UPDATE en PostgreSQL: el precio ahora es 499
B2  SET en la cache: 499
A2  SET en la cache: 470

PostgreSQL: 499
Cache:      470
VIEJO: la cache va a responder 470 durante 29994 ms más, hasta que venza el TTL.
```

- En PostgreSQL ganó B. En la cache ganó A. Nada obliga a que los dos órdenes coincidan.
- Borrando, cualquier orden deja la cache sin entrada: la próxima lectura trae 499.

La carrera que queda: una lectura y una escritura



- El escritor borra una entrada que todavía no existe.
- El lector guarda, recién traído, un precio que ya no es. La invalidación ocurrió y no sirvió.
- De los diez órdenes posibles de esos cinco pasos, solo este deja la cache vieja.

Qué la acota y qué la elimina

```bash
$ node race.ts L1 L2 E1 E2 L3
...
VIEJO: la cache va a responder 455 durante 29998 ms más, hasta que venza el TTL.
```

La acota: el TTL

Toda entrada se guarda con tiempo de vida. El peor caso dura *T*.

La elimina: un lease

Al tener un miss, el lector recibe un token. Un borrado lo invalida, y el guardado se rechaza.

Idea clave. Con cache-aside, la consistencia no se garantiza: **se acota**. Elegir el TTL es elegir el techo.

La tabla de decisiones, antes de programar

Columna	La pregunta
¿Va a la cache?	¿Cuánto se lee por cada cambio, cuánto cuesta, cuánto retraso tolera?
Clave	¿Qué hace que dos respuestas sean distintas?
TTL	¿Cuánto retraso tolera el negocio para este dato?
Qué la invalida	¿Qué escrituras la dejan vieja, y dónde están en el código?
Cuán vieja puede estar	Si falla una invalidación, ¿qué ve el usuario y por cuánto tiempo?
Cache vacía o caída	¿El origen aguanta?

«No va a la cache» también es una decisión, y se escribe. Archivo: `code/week-11/cache-decisions-template.md`.

SECCIÓN 08

Estampida

08

Una entrada muy pedida vence

Todas las peticiones tienen un miss a la vez, y todas van al origen a calcular lo mismo.

$$\text{consultas por estampida} \approx \lambda \cdot d$$

```
make stampede: la lista vence cada 2 segundos, 50 usuarios

== CACHE_MODE=aside, la lista vence cada 2 segundos
Consultas a arq_w11 que contienen "LEFT JOIN stock_movements": 95, en 5 ráfagas
ráfaga  empieza en  consultas  dura
     1       0,0 s         50    96 ms
     2       2,1 s         14    25 ms
     3       4,2 s          9    16 ms
     4       6,2 s          9     8 ms
     5       8,2 s         13    17 ms
```

Había que calcular la lista 5 veces. Se calculó 95. Y d no es constante: con el origen más lento, la misma prueba dio 140.

Que solo uno recalcule

Tres claves por entrada:

- `w11:products`: el valor. Vive mucho más que el TTL.
- `w11:products:fresh`: una marca que vive el TTL. Mientras existe, el valor está vigente.
- `w11:products:lock`: solo mientras alguien recalcula.

```
SET lock token NX PX 5000
```

`NX`: solo si no existe. De cincuenta que lo mandan, uno recibe `OK`.

`PX`: se borra solo. El lock de un proceso muerto vence.

- El que obtiene el lock va al origen. **Los demás reciben el valor anterior.**
- Es un lock de optimización: si falla, se trabaja de más. Uno del que dependa la corrección es otro problema *(semana 12)*.

Medido: de 95 consultas a 5

```
make stampede
```

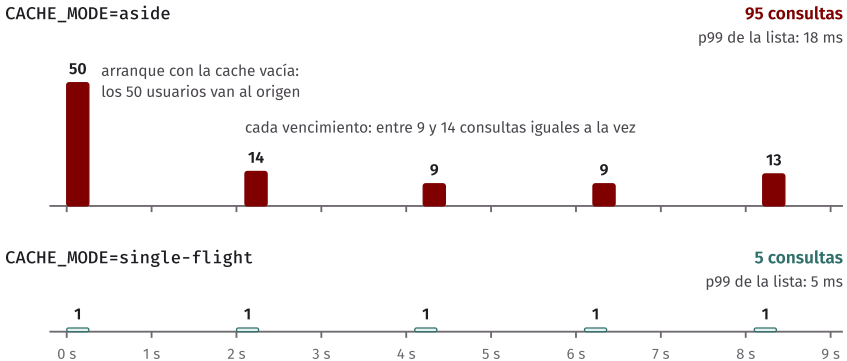
```
== CACHE_MODE=single-flight, la lista vence cada 2 segundos
Consultas a arq_w11 que contienen "LEFT JOIN stock_movements": 5, en 5 ráfagas
ráfaga  empieza en  consultas  dura
     1       0,0 s          1     0 ms
     2       2,0 s          1     0 ms
     3       4,1 s          1     0 ms
     4       6,1 s          1     0 ms
     5       8,1 s          1     0 ms
```

Trade-off

Se cambia carga sobre el origen por frescura: mientras uno recalcula, los demás reciben un dato que **ya venció**. La alternativa es que esperen.

Facebook llama *lease* a un mecanismo así: Nishtala y otros, *Scaling Memcache at Facebook* (NSDI 2013), sección 3.2

Las ráfagas, a escala



La entrada vence cada 2 segundos: en 9 segundos había que calcular la lista 5 veces.

Jitter: que no venzan todas juntas

Mil entradas guardadas en el mismo instante, con el mismo TTL, vencen en el mismo instante.

```
make jitter

TTL de 60 s exactos: vencen entre el segundo 60 y el 60; en el peor segundo vencen 1000 de 1000
TTL de 60 s ± 10 %  : vencen entre el segundo 54 y el 66; en el peor segundo vencen 98 de 1000
```

- Cuesta una línea, y que el TTL deje de ser un número exacto.
- Tercer remedio: recalcular antes de que venza, con un proceso de fondo. Sirve para pocas claves conocidas, y es un proceso más que puede detenerse sin que nadie lo note.

SECCIÓN 09

Depender de la cache

09



Cache de latencia y cache de capacidad

De latencia

Sin ella, el servicio aguanta la carga esperada, más lento.

De capacidad

Sin ella, no la aguanta.

- La diferencia no está en el código: está en cuánta carga soporta el origen solo.
- Cambia con el tiempo sin que nadie toque nada: el tráfico crece y la base de datos queda dimensionada para los misses.
- Se descubre con la cache **caída** y con la cache **vacía**. Mejor que no sea en producción.

Site Reliability Engineering, cap. 22, *Addressing Cascading Failures: Slow Startup and Cold Caching*

Redis caído: el servicio que no lo previó

```
make redis-down, CACHE_FAILURE=closed

t= 4s  pedidos= 460  hits= 440  viejos=   0  origen=  20  errores_cache=   0  5xx=   0  p95=5,6 ms
   (segundo 4: svc kill redis)
t= 5s  pedidos= 500  hits=  49  viejos=   0  origen=   1  errores_cache= 450  5xx= 450  p95=1,0 ms
t= 6s  pedidos= 499  hits=   0  viejos=   0  origen=   0  errores_cache= 499  5xx= 499  p95=0,1 ms
t= 7s  pedidos= 486  hits=   0  viejos=   0  origen=   0  errores_cache= 486  5xx= 486  p95=0,1 ms
t= 8s  pedidos= 465  hits=   0  viejos=   0  origen=   0  errores_cache= 465  5xx= 465  p95=0,1 ms
   (segundo 8: svc start redis)
t= 9s  pedidos= 451  hits=  87  viejos=   0  origen=  64  errores_cache= 300  5xx= 300  p95=90,8 ms
...
http_req_failed: 37.95% 2200 out of 5796
```

Un componente opcional tumbó al sistema

Todos los pedidos fallaron con PostgreSQL sano y sin recibir una sola consulta.

Redis caído: el servicio que degrada

```
make redis-down, CACHE_FAILURE=open

t= 4s  pedidos= 495  hits= 472  viejos=   0  origen=  23  errores_cache=   0  5xx=   0  p95=1,0 ms
   (segundo 4: svc kill redis)
t= 5s  pedidos= 439  hits=  11  viejos=   0  origen= 434  errores_cache= 860  5xx=   0  p95=24,8 ms
t= 6s  pedidos= 447  hits=   0  viejos=   0  origen= 443  errores_cache= 890  5xx=   0  p95=27,7 ms
t= 7s  pedidos= 449  hits=   0  viejos=   0  origen= 452  errores_cache= 901  5xx=   0  p95=23,3 ms
t= 8s  pedidos= 440  hits=   0  viejos=   0  origen= 447  errores_cache= 887  5xx=   0  p95=28,7 ms
   (segundo 8: svc start redis)
t= 9s  pedidos= 470  hits= 188  viejos=   0  origen= 270  errores_cache= 479  5xx=   0  p95=25,5 ms
...
http_req_failed: 0.00%  0 out of 5690
```

- Una cache caída se ve como una cache vacía. Y el cliente de Redis **falla al instante**, sin encolar.
- ¿PostgreSQL aguantó? Aquí sí: es una cache de latencia, *con 50 usuarios*. Hay que medirlo.

Memoria finita: desalojo y cache negativa

```
make eviction

maxmemory-policy = noeviction
  escrituras rechazadas: 3524  ("OOM command not allowed when used memory > 'maxmemory'.")
...
maxmemory-policy = allkeys-lru
  escrituras rechazadas: 0
  entradas desalojadas:  3525
```

`noeviction` es el valor por omisión de Redis. LRU: sale la menos usada recientemente; LFU: la menos frecuente. Abajo, la cache negativa: guardar también «no existe», con un TTL corto.

```
make negative

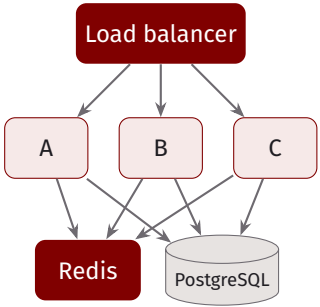
200 pedidos de GET /products/999999, que no existe:
  sin cache negativa:              200 consultas a PostgreSQL
  con cache negativa (TTL de 5 s):  1 consulta a PostgreSQL
```

SECCIÓN 10

Escalar hacia los lados

10

Más copias del servicio: sin estado



Servicio sin estado

No guarda nada propio entre un pedido y el siguiente. El estado vive en un almacén compartido.

- Cualquier instancia atiende cualquier pedido, y cualquiera se puede apagar.
- Lo rompen: sesiones en memoria, archivos en el disco local, una cache local que tiene que ser exacta.

Escalar hacia los lados no elimina el estado: **lo mueve**.

The Twelve-Factor App, VI. *Processes*

Error común: cache en el proceso, con dos instancias

make local-cache

```
GET /products/2 → instancia A responde 490
GET /products/2 → instancia B responde 490
PUT /products/2/price 550 → lo atiende la instancia A
GET /products/2 → instancia B responde 490
GET /products/2 → instancia A responde 550
GET /products/2 → instancia B responde 490
GET /products/2 → instancia A responde 550
```

Peor que un dato viejo

Un dato que cambia hacia atrás: 490, 550, 490, 550. Es la anomalía de las lecturas que retroceden *(semana 10)*, con caches en lugar de réplicas.

Load balancer y health checks

Repartir

- Por turnos.
- A la menos ocupada.

Dejar de enviar a quien no pasa el health check:

- `/health`: el proceso vive.
- `/ready`: además puede trabajar.

La trampa

Si el health check falla cuando Redis no responde, una caída de Redis saca a **todas** las instancias a la vez: la degradación se vuelve caída total.

Regla

Comprobar lo que esa instancia necesita y que otra podría tener bien.

Laboratorio: cache-aside en Tinkuy y una estampida

En parejas, sobre su copia de Tinkuy, con `code/week-11/`.

1. Línea de base: `make load-browse`; mediana y p95 de la lista y del detalle.
2. Llenar la tabla de decisiones **antes** de programar.
3. Cache-aside para el detalle de un evento, con el módulo de `lab/`. Medir. Mirar Redis.
4. Comprar y pedir el detalle: ¿cuántos asientos muestra? Agregar la invalidación.
5. `race.ts`: encontrar el orden que deja la cache vieja a pesar de la invalidación.
6. Lista con TTL de 2 s y carga: contar las consultas con `make pg-report`.
7. Que solo uno recalcule, con `SET NX PX`. Comparar.
8. `svc kill redis` con la carga corriendo. ¿PostgreSQL aguanta?
9. Sharding en papel: `make shards` y `make scatter`.

Para la próxima clase

- Esta semana se entrega **P3 · Lecturas que escalan** y se publica **P4 · Pagos como servicio**.
- Terminar la tabla de decisiones del laboratorio, corregida con lo que midieron.
- Semana 12, transacciones. Leer de *Designing Data-Intensive Applications* (2.ª ed.) el cap. 8 (Sesión A), y de *Software Architecture: The Hard Parts* los cap. 9 y 12 (Sesión B).
- Lectura libre: PostgreSQL 18, *Transaction Isolation* y *Explicit Locking*.

Pregunta

Tinkuy vende un asiento con un `SELECT` y después un `UPDATE`. Si dos compradores leen antes de que cualquiera escriba, ¿qué pasa? ¿Se parece a la carrera de hoy?

Lecturas de hoy: *Designing Data-Intensive Applications* (2.ª ed.), cap. 7; Azure, *Sharding pattern* y *Cache-Aside pattern*; *SRE*, cap. 22; *Software Architecture in Practice*, cap. 9.