

SISTEMAS DISTRIBUIDOS

CLASE 12 — CONSISTENCIA EN CACHÉS

Hoja de ruta

1. ¿Cuánta capacidad tiene un sistema?
2. ¿Cómo se aumenta esa capacidad, y qué papel cumple un caché?
3. ¿Cómo funciona un look-aside cache?
4. ¿Cómo se invalidan muchos cachés repartidos en varias regiones?
5. ¿Qué pasa cuando se invalida una clave muy leída?

TEORÍA DE COLAS Y CAPACIDAD

Escalado de una aplicación web

- Primera iteración: servidor HTTP y base relacional en **una sola máquina**
- Siguiente paso: un **load balancer** adelante de varios servidores **stateless**
- Si falla un servidor, se lo reemplaza sin que nadie lo note
- Todos acceden a la **misma base relacional**, que pasa a ser el cuello de botella

El sistema como caja negra

- **λ(t)**: tasa de entradas (requests/s); **δ(t)**: tasa de salidas
- **N(t)**: clientes dentro del sistema, procesándose o esperando

$$\frac{dN(t)}{dt} = \lambda(t) - \delta(t)$$

- Si entra más de lo que sale, N crece y la espera también
- Caso estable: **λ = δ**, y N es constante

Ley de Little

$$L = \lambda W$$

- **L**: clientes dentro del sistema; **λ**: tasa de arribos; **W**: tiempo dentro del sistema
- Las tres son promedios en el tiempo
- Ejemplo, un café: λ = 45 personas/hora y W = 25 minutos
- L = 45 p/h × 25/60 h = **18,75 personas**: la cantidad de sillas

Las dos perillas

$$\lambda = \frac{L}{W}$$

Variable	Nombre en la industria	Qué se puede hacer
λ	Throughput	Nada: es la demanda
L	Paralelismo	Agregar conexiones o threads
W	Tiempo promedio por query	Hacer más cortas las queries

- Para aumentar λ: subir L o bajar W. No hay una tercera opción

Límites del paralelismo en la base

- Ante un pico de demanda, la reacción habitual es **agregar servidores**
- Una base no se multiplica con la misma facilidad
- Las réplicas solo absorben lecturas
- Subir `max_connections` tampoco escala: cada conexión consume memoria y compite por locks y CPU

Look-aside cache

- **Memcached** o **Redis**: un hashmap en memoria con `get(k)` y `set(k, v)`, ~1 ms con la red

1. Buscar la clave en el caché
2. Si está (**hit**), devolverla
3. Si no está (**miss**), consultar la base
4. Guardar el resultado en el caché y devolverlo

Efecto sobre la capacidad

- Hit ratio del 98%; el caché responde en 1 ms y la base en 10 ms

$$W = 0{,}98 \times 1\ \text{ms} + 0{,}02 \times 10\ \text{ms} = 1{,}18\ \text{ms}$$

- W baja de 10 ms a 1,18 ms
- Con el mismo paralelismo en los servidores web, λ = L/W crece **unas 8 veces**

UN CACHÉ AUMENTA LA CAPACIDAD DEL SISTEMA Y PROTEGE LA BASE DE DATOS

La menor latencia es un efecto secundario deseable

LOOK-ASIDE CACHE

Look-aside y look-through

- **Look-aside**: la aplicación habla por separado con el caché y con la base
- **Look-through**: la aplicación solo habla con el caché, y el caché con la base
- Look-through encapsula la consistencia, pero exige **lógica de negocio** en el caché
- Look-aside se arma alrededor de Redis o memcached: es el que usa Facebook

Arquitectura de Facebook

- Regiones con réplicas de todo: frontends **stateless**, memcache **shardeado**, base **particionada**
- Replicación **primary/secondary** entre regiones: las secondary solo leen
- Una escritura que llega a una región secondary viaja a la base del primary
- La carga es de muchas más lecturas que escrituras

Algoritmos de lectura y escritura

```
def read(k):
    v = memcache.get(k)
    if v is None:
        v = db.fetch(k)       # select contra la base
        memcache.set(k, v)
    return v

def write(k, v):
    db.update(k, v)
    memcache.delete(k)        # invalidación
```

Invalidación y TTL

- Memcached y Redis admiten un **time to live** por clave
- Con un TTL de media hora, el sistema puede devolver datos viejos durante media hora
- Con un TTL de un segundo, el hit ratio se derrumba
- El TTL sirve como red de seguridad; las invalidaciones son **explícitas**
- Aun así, look-aside implica **consistencia eventual**

Race condition 1: set en lugar de delete

- C1 escribe X = 1 en la base
- C2 escribe X = 2 en la base y hace `set(X, 2)`
- C1 llega tarde con `set(X, 1)`: la base queda con **2** y el caché con **1**
- Con `delete`, el orden no importa: la operación es **idempotente**
- Escribir en la base y en el caché es una **pequeña transacción distribuida**

ESCALADO Y SINCRONIZACIÓN DE CACHÉS

Sharding y clusters

- Son shards, no réplicas: se shardea para sumar **memoria**
- Usan **consistent hashing**
- Cada request necesita muchas claves y termina accediendo a **todos los shards**
- Cada **cluster** tiene frontends y shards propios; hay varios por región

Invalidación cross región

- La escritura borra solo el memcache **de su región**
- La otra región recibe V2 por la replicación, pero su memcache sigue con **V1**, indefinidamente
- **mcsqueal**: un proceso conectado al **write-ahead log** de la base de cada región
- Detecta cada update que llega y hace el `delete` contra el memcache local

Claves embebidas en el SQL

```
UPDATE users SET name = 'foo' WHERE id = 123 -- KEYS ['users:123']
```

- El frontend agrega las claves a invalidar como **comentario**: el motor lo ignora, mcsqueal lo parsea
- La técnica general se llama **change data capture** (CDC); por ejemplo, **Debezium** + Kafka
- Sirve para actualizar cualquier subsistema, siempre con deletes

Doble camino de la invalidación

- Un `delete` directo a la otra región acoplaría el frontend a la topología de otro data center
- mcsqueal también corre en la región local: cubre un `delete` local que se perdió
- El `delete` local existe para que el mismo request lea lo que acaba de escribir
- El diseño no depende de que las claves expiren

Race condition 2: réplica atrasada

1. FE1, en la región secondary, escribe en la base del primary
2. FE1 borra la clave de su memcache
3. FE2 lee: miss, y consulta la réplica local
4. La réplica todavía no recibió el cambio: FE2 guarda **el valor viejo**

Remote marker

- Escritura: `set(rk)`, después `update_master` y `delete(k)`
- Lectura: si `get(k)` da miss y `get(rk)` da hit, se lee del **master**
- mcsqueal borra el marcador cuando la réplica se pone al día
- Mientras dura la transición, el caché queda deshabilitado para esa clave

THUNDERING HERD Y LEASES

Thundering herd

- Una **hot key** es una clave muy leída: casi nunca llega a la base
- Al editarse, el `delete` produce miles de misses al mismo tiempo
- Todos los frontends consultan la base: se satura, y con ella **todos los demás requests**
- Lo que provoca la estampida no es el volumen invalidado, sino **qué clave** se invalida

Leases

- Un solo frontend va a la base; memcache, centralizado, hace de árbitro
- `get(k)` → (miss, **lease** de 64 bits): memcache guarda la clave con el lease y sin valor
- Otro `get(k)` → (miss, **retry**): esperar unos milisegundos y reintentar
- `set(k, v, lease)`: se acepta solo si el lease coincide

Race condition 3: lectura lenta

- C1: miss y lectura de la base → V1
- C2: escribe V2 y hace `delete(k)`
- C1 llega tarde con `set(k, V1)`
- Sin leases: la base tiene V2 y el caché **V1**

El lease como optimistic locking

- El `delete` de C2 borra la entrada entera: clave, valor y lease
- El `set(k, V1, lease)` de C1 falla: ese lease ya no existe
- El próximo lector obtiene un lease nuevo y guarda el valor correcto
- Es el mismo papel que cumple la versión en **CAS** (`gets` / `cas` de memcached)
- El lease tiene un TTL corto: un cliente caído no bloquea la clave

Síntesis

- Un caché multiplica la capacidad, y el precio es mantener dos storages sincronizados
- **mcsqueal**: la invalidación viaja por el log de la base
- **Remote marker**: la lectura evita la réplica atrasada mientras dura la transición
- **Lease**: un solo lector por clave, y ningún `set` con un valor viejo