

UNIVERSIDAD NACIONAL DE INGENIERÍA

Arquitectura de Software

Los datos como decisión de arquitectura

Transaccional y analítico · Un nodo o muchos · Modelos de datos · Evolución de esquemas y contratos

Gerson Garrido

Semana 9

Agenda

Sesión A

1. Por qué los datos cuestan más
2. Transaccional y analítico
3. Sistema de registro y datos derivados
4. Un nodo o muchos; administrado o propio
5. La escalabilidad es una pregunta

Sesión B

6. Modelos de datos
7. Compatibilidad: hacia atrás y hacia adelante
8. Formatos y evolución de una API
9. Expandir, migrar y contraer

Lectura

A: Kleppmann y Riccomini, *Designing Data-Intensive Applications* (2.ª ed.), cap. 1 y 2; Kleppmann, apuntes de *Concurrent and Distributed Systems* (Cambridge), sección 1.
B: *Designing Data-Intensive Applications*, cap. 3 y 5; Sato, *ParallelChange*; Fowler, *TolerantReader*.

SECCIÓN 01

Por qué los datos cuestan más

01

El código se reemplaza; los datos se quedan

	Un proceso sin estado	Los datos
Reemplazar	Se levanta otro y se apaga el anterior	Tienen que ser los mismos de ayer
Copiar	Dos copias son idénticas	Dos copias divergen con la siguiente escritura
Perder	Alguien reintenta	Se perdió el negocio
Revertir	Se vuelve al código anterior	Las filas ya transformadas no vuelven solas

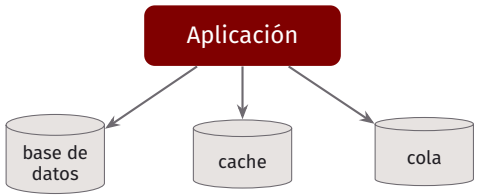
Idea clave

Los datos viven más que el código que los escribió. Dónde viven, cuántas copias hay y quién los escribe son las decisiones que más cuesta deshacer.

Sistemas intensivos en datos

La dificultad está en **los datos**: cuántos son, lo complejos que son, lo rápido que cambian. No en el cómputo.

- Se arman con piezas estándar.
- Cada pieza promete algo distinto.
- La arquitectura es cómo se combinan y qué garantías sobreviven a la combinación.



semanas 10 a 13:
una combinación por semana

SECCIÓN 02

Transaccional y analítico

02

Dos formas de usar los mismos datos

	Transaccional *(OLTP)*	**Analítico** *(OLAP)*
Quién pregunta	Un usuario que espera	Un analista, un proceso por lotes
Qué lee	Pocas filas, por clave	Millones de filas
Cuántas consultas	Muchas por segundo	Pocas por hora
Qué se mide	Latencia (p95, p99)	Throughput
Qué datos	El estado de ahora	La historia

En la tabla de órdenes

«¿De quién es la orden 150 000?» es transaccional. «¿Quiénes gastaron más en marzo?» es analítica.

La diferencia, en el plan de ejecución

psql · una orden por su id

```
Index Scan using orders_pkey on orders (actual rows=1.00 loops=1)
  Index Cond: (id = 150000)
  Buffers: shared hit=4
Execution Time: 0.147 ms
```

psql · los cinco compradores que más gastaron

```
Limit (actual rows=5.00 loops=1)
  Buffers: shared hit=3478
             ->  Gather (actual rows=15000.00 loops=1)
                   Workers Launched: 2
                         ->  Parallel Seq Scan on orders (actual rows=95652.33 loops=3)
Execution Time: 79.445 ms
```

4 páginas contra 3 478: la tabla entera, leída por tres procesos a la vez.

El reporte compite con la venta

```
make oltp-olap

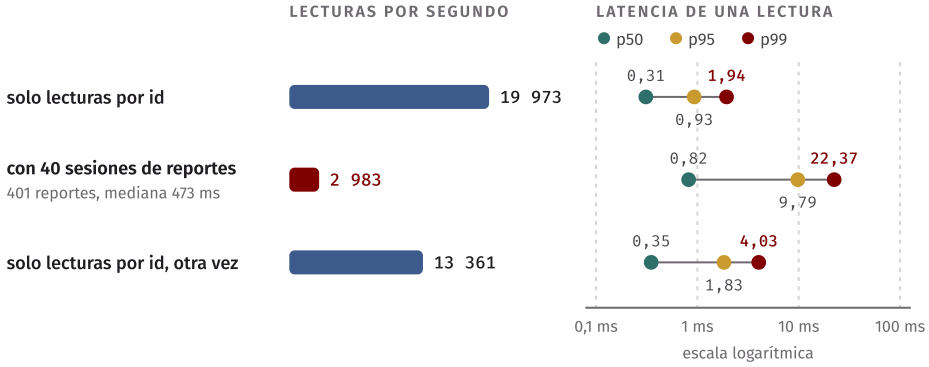
arq_w09: 300000 órdenes; 8 sesiones leen una orden por id durante 5 s.

solo lecturas por id                19973 lecturas/s   p50    0.31 ms   p95    0.93 ms   p99    1.94 ms
con 40 sesiones de reportes          2983 lecturas/s   p50    0.82 ms   p95    9.79 ms   p99   22.37 ms
solo lecturas por id, otra vez      13361 lecturas/s   p50    0.35 ms   p95    1.83 ms   p99    4.03 ms
```

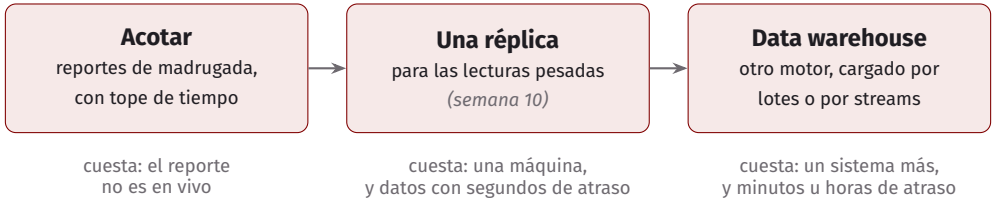
- Las mismas lecturas, los mismos datos, el mismo índice. Cambió quién más usaba la base de datos.
- Throughput: a menos de la sexta parte. p99: más de diez veces. El p50 casi no se movió.
- Condición: el procesador está saturado (40 sesiones de reportes, 14 núcleos).

Laptop con Apple Silicon, dentro de Docker. Tres corridas: con reportes, entre 1 500 y 3 000 lecturas/s; sin ellos, entre 12 000 y 20 000.

La medición, dibujada



Separarlos: de menos a más costo



Trade-off

Cada paso compra aislamiento entre las dos cargas y lo paga con frescura y con operación. La pregunta que decide: ¿qué tan viejo puede ser el dato del reporte?

SECCIÓN 03

Sistema de registro y datos derivados

03

Una sola fuente de verdad para cada dato

Sistema de registro: donde el dato se escribe primero. Si dos lugares no coinciden, este tiene la razón.

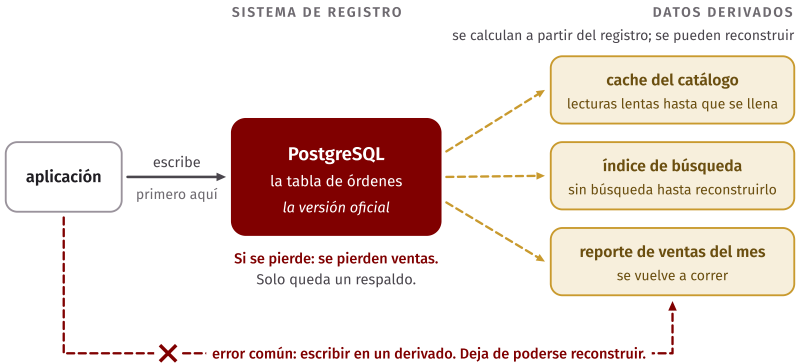
Dato derivado: se obtiene transformando otros (cache, índice, vista materializada, reporte). Se puede reconstruir.

Dato	Si se pierde
Tabla de órdenes	Se pierden ventas
Cache del catálogo	Lecturas lentas un rato
Índice de búsqueda	Sin búsqueda mientras se reconstruye
Reporte del mes	Se vuelve a correr

Error común: tratar un dato derivado como el oficial

Alguien corrige un total en la tabla del reporte, y desde ese día ya no se puede reconstruir.

El registro y lo que se deriva de él



SECCIÓN 04

Un nodo o muchos

04

Tres razones para distribuir los datos, y lo que cuesta

Razones (cada una, un escenario)

- Tolerar la falla de una máquina.
- Escalar más allá de una.
- Acercar los datos a los usuarios.

Lo que cuesta

- Fallas parciales.
- Retraso de replicación: leer el pasado.
- Relojes que no coinciden.
- Lo que era una transacción pasa a ser un protocolo.

Una sola máquina aguanta mucho: en la medición anterior, entre 12 000 y 20 000 lecturas por clave por segundo en una laptop. Y es más simple de razonar: un reloj, transacciones, y cuando falla, falla entera.

Servicio administrado o propio

	Administrado	Propio
Quién opera	El proveedor	El equipo
Qué se paga	Dinero, cada mes	Tiempo de gente que sepa
Control	El que el menú permita	Todo
Garantías	Las del contrato	Las que el equipo construya
Salir	Fácil si el motor es estándar	—

Por qué es una decisión de arquitectura

Fija **qué garantías se pueden pedir** (el failover del menú es el piso del escenario) y **cuánto cuesta salir**.

SECCIÓN 05

La escalabilidad es una pregunta

05



Primero, describir la carga

«El sistema es escalable» no dice nada.

La pregunta: si la carga crece *de esta manera*, ¿qué opciones hay, y cuánto cuesta cada una?

- Lecturas y escrituras por segundo.
- La proporción entre ellas.
- El tamaño de los datos.
- La forma: pareja o en picos.
- El caso extremo.

Tinkuy

- Miles de lecturas del catálogo por cada compra.
- Escrituras: casi nada, y un pico el minuto en que abre una venta.
- Caso extremo: un solo evento concentra casi todas las escrituras del pico.

Las líneas de tiempo: calcular al leer o al escribir

calcular al leer

```sql
SELECT p.id, p.created_at
  FROM tl_follows f JOIN tl_posts p
    ON p.author_id = f.followee_id
 WHERE f.follower_id = $1
 ORDER BY p.created_at DESC LIMIT 20
```

Publicar: 1 fila. Leer: del orden de $s \cdot p$ publicaciones.

calcular al escribir (fan-out)

```sql
INSERT INTO tl_home
       (user_id, created_at, post_id)
SELECT follower_id, $2, $3
  FROM tl_follows
 WHERE followee_id = $1
```

Publicar: $1 + f$ filas. Leer: 20 entradas de un índice.

s: cuentas seguidas; p: publicaciones recientes de cada una; f: seguidores del autor. **El trabajo no desaparece: se mueve de una operación a la otra.**

La cuenta famosa rompe la opción que servía para los demás

```
make timeline

10000 usuarios, 50000 publicaciones, 1549845 filas en las líneas de tiempo ya calculadas.
La cuenta 4242 tiene 30 seguidores; la cuenta 1, 9999.

Leer una línea de tiempo (mediana de 500 lectores):
  calculada al leer       0.58 ms
  calculada al escribir   0.32 ms

Publicar (mediana de 20 publicaciones):
  calculada al leer,     cuenta común    0.96 ms   (1 fila)
  calculada al leer,     cuenta famosa   0.93 ms   (1 fila)
  calculada al escribir, cuenta común    1.54 ms   (31 filas)
  calculada al escribir, cuenta famosa   96.33 ms   (10000 filas)
```

El conteo de filas no depende de la máquina: es $1 + f$. Salida habitual: repartir al escribir para las cuentas comunes y buscar al leer las de las pocas cuentas enormes.

La misma decisión, en Tinkuy

tinkuy/src/catalog/seats.ts (hoy: contar en cada lectura)

```ts
export async function countAvailableSeats(eventId: number): Promise<number> {
  const rows = await query<{ available: number }>(
    'SELECT count(*)::int AS available FROM seats WHERE event_id = $1 AND status = 'available'',
    [eventId],
  );
  return rows[0]?.available ?? 0;
}
```

	Contar al leer	Mantener el número al vender
Leer el catálogo	Un `COUNT` por evento	Un número ya calculado
Vender	Nada extra	Una escritura más
Caso extremo	Muchos eventos grandes	Todo el pico cambia la misma fila

Error común: diseñar para una escala que no existe

«Por si crece»

Cada componente distribuido que se agrega por si crece es operación y modos de falla **desde hoy**. El crecimiento es una hipótesis; el costo es un hecho.

1. **Describir la carga** de hoy y la que se espera, con parámetros.
2. **Saber dónde está el límite** del diseño actual: se mide.
3. **No cerrarse la puerta**: el acceso a datos detrás de un módulo, cada módulo owner de sus tablas.

SECCIÓN 06

Modelos de datos

06

El modelo de datos decide qué preguntas son fáciles

Modelo	Fácil	Difícil
Relacional	Muchos a muchos, `JOIN`, restricciones, transacciones	Armar un objeto con muchas partes
De documentos	Un agregado que se lee y se escribe entero	Relaciones entre documentos
De grafos	Recorridos de longitud variable	Menos gente lo sabe operar
Clave-valor	Acceso por clave, a escala y con prisa	Cualquier otra pregunta

No se elige por moda

Se escriben las consultas que el sistema hace todo el tiempo y se mira con qué modelo cada una es una operación natural.

El mismo pedido, de dos maneras

```
make models

== 2. Una pregunta que cruza pedidos: cuántos ejemplares se vendieron de cada libro
Documento: hay que abrir todos los documentos y desarmar sus listas.
== 3. Corregir el título de un libro
Relacional:
UPDATE 1
Documento: el título está copiado en cada pedido que lo incluye.
UPDATE 416
== 4. Un dato mal formado
Relacional (esquema al escribir): la base de datos lo rechaza.
ERROR:  new row for relation "m_items" violates check constraint "m_items_quantity_check"
Documento (esquema al leer): entra, y falla después, en quien lo lea.
ERROR:  invalid input syntax for type integer: "dos"
```

Leer un pedido entero: una fila en el documento, dos `JOIN` en las tablas. El precio *debe* estar copiado en el pedido (es un hecho histórico); el título no.

Esquema al escribir y esquema al leer

Al escribir

La estructura está en la base de datos. Lo que no la cumple, no entra. El error le llega a quien escribe.

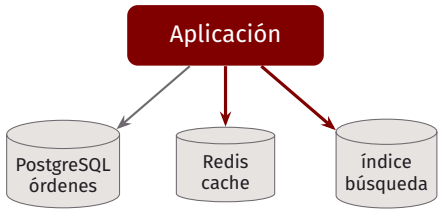
Al leer

La base de datos acepta todo. La estructura está en el código que lee. El error le llega a quien lee, meses después.

«Sin esquema»

Quiere decir que el esquema está en el código que lee, sin nadie que lo haga cumplir.

Persistencia políglota



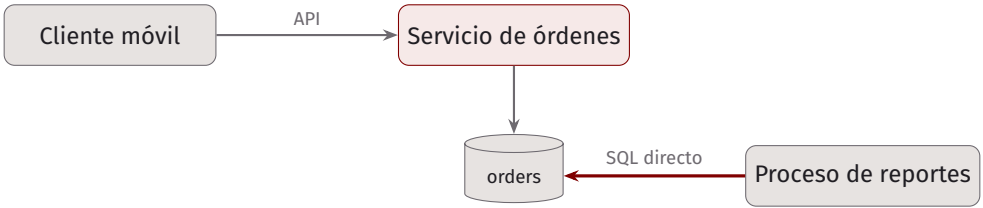
Una base de datos por necesidad.

- **Se gana** ajuste: cada dato donde sus preguntas son fáciles.
- **Se paga** operación: tres sistemas que instalar, vigilar y respaldar.
- **Se paga** consistencia: ya no hay una transacción que cubra las tres escrituras.

La que se sostiene

Un sistema de registro por dato; los demás almacenes son derivados que se alimentan de él.

El modelo de datos como interfaz



- Si solo el servicio toca la tabla, el esquema es un detalle de implementación.
- Si otro la lee, cada columna es una promesa que el servicio hizo sin saberlo.
- De ahí la regla de las semanas 4 y 5: cada módulo o servicio es owner de sus datos.

SECCIÓN 07

Compatibilidad

07

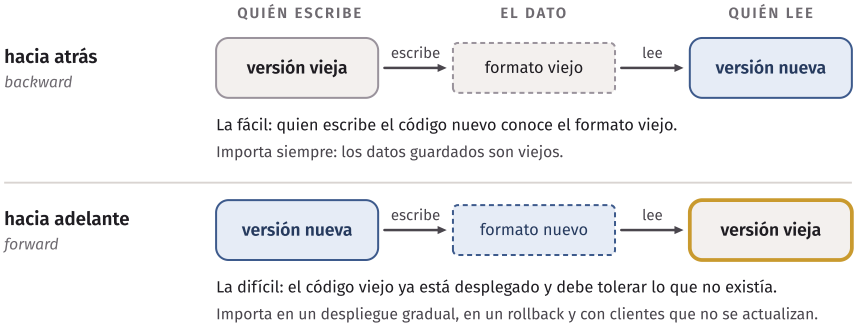
Versiones que conviven

En un sistema que no se detiene, el código viejo y el nuevo funcionan a la vez sobre los mismos datos:

- **Un despliegue gradual**: unas instancias con la versión anterior y otras con la nueva.
- **Un cliente que no se actualiza**: la aplicación móvil instalada hace dos años.
- **Un mensaje que quedó en una cola**.
- **Los datos guardados**: toda fila la escribió alguna versión anterior.

> **Idea clave**
>
> Un cambio de formato o de contrato no se diseña para «antes» y «después»: se diseña para **«durante»**.

Hacia atrás y hacia adelante



- Sin compatibilidad hacia adelante, el rollback no funciona: quedan filas nuevas leídas por código viejo.

El campo que se pierde

formats/document-roundtrip.ts

```ts
export function cancelDroppingUnknown(document: string): string {
  const raw = JSON.parse(document) as OrderV1;
  const order: OrderV1 =
    { id: raw.id, buyerName: raw.buyerName, totalCents: raw.totalCents, status: 'cancelled' };
  return JSON.stringify(order);
}
```

node formats/document-roundtrip.ts

```
escrito por el código nuevo:  {"id":7,"buyerName":"Ana Ccori",...,"totalCents":10000,"currency":"USD","status":"paid"}
anulado por el código viejo:  {"id":7,"buyerName":"Ana Ccori","totalCents":10000,"status":"cancelled"}
anulado conservando el resto: {"id":7,"buyerName":"Ana Ccori",...,"totalCents":10000,"currency":"USD","status":"cancelled"}
```

Tolerar lo desconocido no es solo no fallar: es no destruirlo. Esta función no dio ningún error. La que conserva el resto escribe `{ ...raw, status: 'cancelled' }`.

SECCIÓN 08

Formatos y evolución de una API

08



JSON: legible, sin esquema, ambiguo con los números

```
node formats/json-numbers.ts

texto recibido:   {"orderId": 9007199254740993, "totalCents": 1999, "price": 0.1}
orderId leído:    9007199254740992
Number.MAX_SAFE_INTEGER = 9007199254740991
0.1 + 0.2 = 0.30000000000000004   (por eso los montos van en céntimos, como enteros)
```

- Cada mensaje lleva los nombres de sus campos: se entiende solo, y ocupa más.
- Nada dice qué campos son obligatorios ni de qué tipo: esquema al leer.
- «Número» y nada más. JavaScript solo representa enteros exactos hasta $2^{53} - 1$.

Binarios con esquema: etiquetas en vez de nombres

```
node formats/tagged.ts

JSON (109 bytes): {"id":150,"buyerName":"Ana Ccori","totalCents":32000,"givenName":"Ana","familyName":"Ccori","currency":"USD"}
binario (35 bytes): 08 96 01 12 09 41 6e 61 20 43 63 6f 72 69 18 80 fa 01 22 03 41 6e 61 2a 05 43 6f 72 69 32 03 55 53 44

Un lector con el esquema v1 lee el mensaje escrito con el v2:
  conoce:    {"id":150,"buyerName":"Ana Ccori","totalCents":32000}
  no conoce: etiquetas 4, 5, 6
```

- `08 96 01`: el campo 1 vale 150. El nombre `id` no viaja.
- Agregar un campo es compatible en las dos direcciones: el lector viejo sabe cuántos bytes saltar.
- Cambiar un nombre es gratis. La etiqueta es para siempre: no se reutiliza.
- Avro: sin etiquetas; se emparejan por nombre el esquema de quien escribió y el de quien lee.

Las reglas de evolución de un formato

Cambio	¿Compatible?
Agregar un campo opcional o con valor por omisión	Sí, en las dos direcciones
Agregar un campo obligatorio	No hacia atrás: los datos viejos no lo tienen
Quitar un campo obligatorio	No hacia adelante: el lector viejo lo exige
Cambiar el nombre de un campo	Gratis con etiquetas; incompatible en JSON
Cambiar el tipo de un campo	En general, no

Con JSON valen las mismas reglas

La diferencia es que nadie las comprueba.

Evolución de una API: exigir menos, prometer más

Cambio	En la respuesta	En la petición
Agregar un campo opcional	Compatible*	Compatible
Agregar un campo obligatorio	Compatible	Incompatible
Quitar un campo	Incompatible	Compatible
Cambiar un nombre o un tipo	Incompatible	Incompatible
Cambiar un significado	Incompatible, sin error	Incompatible, sin error

*** Si el cliente es un lector tolerante**

Toma de la respuesta lo que necesita y no opina sobre el resto.

Ley de Hyrum

La observación

Con suficientes usuarios de una API, no importa lo que el contrato prometa: alguien dependerá de cada comportamiento observable del sistema.

- El **orden** de una lista que el contrato no ordena: «tu última compra» es el último elemento.
- El **texto** de un mensaje de error.
- La **latencia**: un timeout de 200 ms contra un servicio que respondía en 150.

«No cambié el contrato» no prueba que el cambio sea seguro: antes de quitar algo, se mide si alguien lo usa.

Hyrum Wright, hyrumslaw.com

Compatible en la forma, incompatible en el significado

```
node errors/currency-meaning.ts

El servicio responde: {"id":300010,...,"buyerName":"Ana Ccori",...,"totalCents":10000,"currency":"USD","status":"paid",...}
```

```
el cliente móvil antiguo muestra

Orden #300010
A nombre de: Ana Ccori
Total: S/ 100.00
Estado: pagada
```

- `currency` es un campo opcional más: el cliente antiguo lo ignora y *sigue funcionando*.
- `totalCents` no cambió de nombre ni de tipo. Cambió de significado.
- Ningún error, ningún log: datos equivocados que se ven correctos.

Error común: una versión nueva para cada cambio

`/v2` para todo

Cada versión viva es código que mantener, probar y operar. Y los clientes de `/v1` siguen ahí.

Cuándo sí

Cuando las dos formas no pueden convivir en un mismo contrato. La mayoría de los cambios se pueden hacer compatibles: agregar, esperar, quitar.

En la industria

Stripe usa versiones con fecha, cada una con pocos cambios incompatibles, y encierra cada cambio en un módulo que transforma la respuesta.

Brandur Leach, stripe.com/blog/api-versioning (2017)

SECCIÓN 09

Expandir, migrar y contraer

09

Error común: renombrar en un solo paso

`ALTER TABLE coupons RENAME COLUMN discount TO percent_off` es atómico. El despliegue del código, no.



Durante los minutos que dura el despliegue

La mitad del sistema lee un nombre que no existe. En desarrollo no se nota: hay una sola instancia.

Tres versiones del código, cuatro estados del esquema

expand-contract/steps/

```sql
-- 1-expand.sql
ALTER TABLE coupons
    ADD COLUMN percent_off integer
    CHECK (percent_off BETWEEN 1 AND 100);

-- 2-migrate.sql
UPDATE coupons SET percent_off = discount
 WHERE percent_off IS NULL;
ALTER TABLE coupons
    ALTER COLUMN percent_off SET NOT NULL,
    ALTER COLUMN discount DROP NOT NULL;

-- 3-contract.sql
ALTER TABLE coupons DROP COLUMN discount;
```

- **antigua**: solo conoce `discount`.
- **puente**: escribe en las dos; lee `coalesce(percent_off, discount)`.
- **nueva**: solo conoce `percent_off`.

Dos detalles

La columna nueva admite `NULL`: la antigua sigue insertando.
En el paso 2 se intercambian las obligaciones.

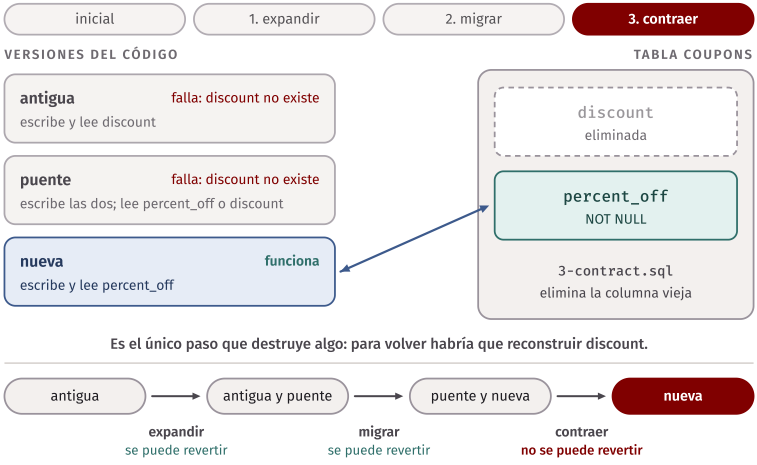
Quién puede convivir con quién

```
make expand-contract
```

```
== 1-expand.sql
escribe \ lee antigua     puente      nueva
antigua       bien        bien        lee NULL
puente        bien        bien        bien
nueva         no escribe  no escribe  no escribe
  nueva: null value in column "discount" of relation "coupons" violates not-null constraint
  pueden convivir: antigua y puente

== 2-migrate.sql
escribe \ lee antigua     puente      nueva
antigua       no escribe  no escribe  no escribe
puente        bien        bien        bien
nueva         lee NULL    bien        bien
  antigua: null value in column "percent_off" of relation "coupons" violates not-null constraint
  pueden convivir: puente y nueva
```

La secuencia



La secuencia: dos observaciones

- En ningún estado conviven la antigua y la nueva: por eso el cambio en un paso es imposible.
- En cada estado conviven dos versiones **consecutivas**: cada paso es un despliegue gradual normal.

Idea clave

Un cambio incompatible se convierte en una secuencia de cambios compatibles, cada uno reversible. La excepción es el último: contraer destruye.

Las tres fases y sus condiciones

Fase	Qué se hace	Condición
Expandir	Lo nuevo junto a lo viejo, opcional. El código escribe en ambos	Termina cuando todo lo que escribe, escribe en los dos
Migrar	Copiar los datos antiguos; mover a quienes leen, uno por uno	Termina cuando nadie lee ni escribe lo viejo
Contraer	Quitar lo viejo: primero del código, después del esquema	Empieza con evidencia de que nadie lo usa

En la industria

Stripe movió sus suscripciones a una tabla propia con cuatro pasos: escribir en las dos tablas, cambiar las lecturas, cambiar las escrituras, eliminar lo anterior.

Jacqueline Xu, *Online migrations at scale*, stripe.com/blog (2017)

Migrar los datos sin detener a nadie

```
make lock-time

300000 órdenes. Durante cada migración, una sesión lee y otra modifica órdenes al azar.

migración                                              duró   lectura más lenta   escritura más lenta
todo en una transacción (errors/rename-in-one-step.sql)  2311 ms             2311 ms              2309 ms
un UPDATE de toda la tabla                               2554 ms               10 ms              2500 ms
lotes de 5000 filas (tools/backfill-batches.ts)          2584 ms               26 ms                58 ms
```

- **Todo en una transacción**: el `ALTER TABLE` toma un lock que no deja ni leer, hasta el `COMMIT`.
- **Un `UPDATE` de toda la tabla**: se puede leer; modificar una fila que ya pasó espera hasta el final.
- **Por lotes**: cada lote es su propia transacción. Ningún lock dura más que un lote.

Laptop con Apple Silicon, dentro de Docker. Cinco corridas: las dos primeras formas hicieron esperar entre 2,3 y 6,8 s; con lotes, entre 43 y 109 ms en cuatro corridas y 612 ms en la peor.

El backfill por lotes

tools/backfill-batches.ts

```ts
for (let from = first; from <= last; from += options.batchSize) {
  const updated = await conn.query(
    'UPDATE orders
        SET given_name = ${GIVEN_NAME_SQL}, family_name = ${FAMILY_NAME_SQL}
      WHERE id >= $1 AND id < $2 AND given_name IS NULL',
    [from, from + options.batchSize],
  );
}
```

- Avanza por rangos de la clave primaria. Solo toca las filas que faltan: se puede interrumpir y retomar.
- Lo que se paga: la migración deja de ser atómica. Todo lector debe funcionar con filas migradas y sin migrar.
- Lo que los lotes no resuelven: «Luz Marina Flores» queda con apellido «Marina Flores». 51 686 filas dudosas.

La cola de locks



- `ADD COLUMN` que admite `NULL` o con valor por omisión constante: solo cambia el catálogo. Cambiar un tipo (`integer` a `bigint`) reescribe la tabla.
- Un reporte de dos minutos más un `ALTER` de un milisegundo: dos minutos de servicio detenido.
- Defensa: `SET lock_timeout = '2s'` antes del `ALTER`. Si no consigue el lock, falla y se reintenta.

Cuándo contraer: saber que nadie usa lo viejo

Un servicio no puede ver qué campos de una respuesta lee un cliente. Lo que sí puede medir:

Qué se mide	Qué no ve
Quién llama (`user-agent`, clave)	Un cliente desconocido que también usa lo viejo
Qué envía el cliente	A quien solo lee
Quién pidió la forma nueva	Cuenta de más, nunca de menos

- El número solo vale con un plazo que cubra el uso más espaciado que se conozca.
- Quien lee la tabla directamente **no aparece en ninguna métrica** del servicio.

Lo que cuesta, y las alternativas

Trade-off

Se compra cero tiempo de inactividad y pasos reversibles. Se paga con tiempo (tres o más despliegues), con código que maneja las dos formas, y con una deuda si nadie hace la contracción.

- **Ventana de mantenimiento**: simple; exige aceptar la caída y cambiar a todos los consumidores a la vez.
- **Versión nueva de la API**: cuando las dos formas no pueden convivir.
- **Un trigger** que sincroniza las columnas: atrapa a todos los que escriben; es lógica fuera del código.

En Tinkuy

- `orders.buyer_name`: el nombre completo, en una columna. En la API, `buyer.name`.
- `orders.total_cents`: en soles, porque nada dice otra cosa.
- El módulo `reports` lee esas tablas con su propio SQL.

El cambio

`buyer_name` pasa a ser nombre y apellido; `total_cents` pasa a llevar moneda.

Ninguno de los dos se puede hacer en un paso

Con el reporte y un cliente antiguo leyendo, hace falta la secuencia completa. El laboratorio la practica sobre un servicio más pequeño.

Laboratorio: cambiar un contrato sin romper a quien lo usa

En parejas, con `code/week-09/`. Se trabaja en `lab/`; `orders-v1/` y `consumers/` no se tocan.

1. `make` y `make test`: el servicio, el cliente móvil antiguo y el reporte que lee la tabla.
2. Clasificar cada parte del cambio: `lab/compatibility-table.md`.
3. Hacerlo mal: `errors/rename-in-one-step.sql` y `make consumers`. Deshacer.
4. Expandir: columnas y campos nuevos; el servicio escribe en ambos. `make consumers S=lab`.
5. Migrar: `make backfill`, `make lock-time`, y mover el reporte.
6. El cliente que no se actualiza: un contador de quién usa todavía lo viejo.
7. Contraer, cuando el contador esté en cero.
8. Hacia adelante: `make forward`. ¿`orders-v1` acepta una orden nueva? ¿Qué conserva?

Para la próxima clase

- Sin entregas esta semana. **P3 · Lecturas que escalan** se entrega en la semana 11.
- Terminar el laboratorio, con las decisiones del paso 6 escritas.
- Semana 10, replicación. Leer de *Designing Data-Intensive Applications* (2.ª ed.), cap. 6 *Replication*: la introducción, *Single-Leader Replication* y los problemas del retraso de replicación.
- Lectura libre: Kleppmann, apuntes de sistemas distribuidos de Cambridge, sección 5, *Replication*.

Pregunta

Si el reporte se va a una réplica, ¿puede no incluir una venta de hace un segundo? ¿Y la pantalla «Mi compra» de quien acaba de pagar?

Lecturas de hoy: *Designing Data-Intensive Applications* (2.ª ed.), cap. 1, 2, 3 y 5; Kleppmann, apuntes de Cambridge, sección 1; Sato, *ParallelChange*; Fowler, *TolerantReader*.