BE SOFTWARE ARCHITECTURE WISE · CAPÍTULO 7

Arquitectura de datos

Los datos sobreviven al código y, en los sistemas distribuidos, son la parte más difícil de hacer bien. Este capítulo trata quién es el propietario de cada dato y por qué una base de datos compartida acopla los servicios, cómo elegir entre bases de datos relacionales y de otros tipos, qué garantizan realmente las transacciones y los niveles de aislamiento, cómo funcionan y fallan la replicación y el particionamiento, CQRS y event sourcing, la caché, cómo hacer evolucionar los esquemas sin tiempo de inactividad, y la analítica, el streaming y el data mesh. Está dirigido a desarrolladores y arquitectos que ya conocen los estilos distribuidos del Capítulo 6 y necesitan tomar decisiones sólidas y defendibles sobre los datos.

bewiseapps.com/es/software-architecture-wise/data-architecture/ · Versión 2026-11-03. Solo información educativa, no asesoramiento profesional.

DE UN VISTAZO

Aprenderás a

- Explicar por qué una base de datos compartida acopla los servicios, y asignar la propiedad de los datos, incluidos los casos en que varios servicios escriben los mismos datos.
- Elegir una base de datos para cada conjunto de datos según la forma de los datos, los patrones de acceso, las necesidades de consistencia, la escala, las operaciones y las habilidades del equipo, y justificar el costo de cada almacén adicional.
- Distinguir los niveles de aislamiento y las anomalías que permiten, incluidas las actualizaciones perdidas y el write skew, y proteger un camino de lectura-modificación-escritura con bloqueos, comprobaciones de versión o transacciones serializables.
- Explicar la replicación líder-seguidor, multilíder y sin líder, las anomalías que causa el retardo de replicación y los riesgos del failover y del split brain.
- Elegir un esquema de particionamiento y una clave de partición (shard key) a partir de volúmenes y consultas medidos, y explicar el hashing consistente, las claves calientes (hot keys) y los índices secundarios.
- Decidir cuándo CQRS, event sourcing y la caché compensan sus costos, y evitar sus modos de fallo habituales.
- Planificar los cambios de esquema con reglas de compatibilidad y el patrón expandir y contraer (expand and contract) para que no requieran tiempo de inactividad.
- Distinguir las arquitecturas de datos operativas y analíticas: almacenes de datos, lagos de datos, lakehouses, procesamiento de flujos y data mesh.

Antes de empezar

Capítulo 6, en especial las secciones sobre microservicios, arquitectura dirigida por eventos, consistencia, CAP y PACELC, y sagas. La sección del Capítulo 0 sobre almacenar datos cubre los fundamentos de bases de datos que aquí se presuponen.

Tiempo

Unas 3½ horas de lectura; 45 minutos para la prueba.

Términos clave

sistema de registro (system of record) · base de datos por servicio · persistencia políglota · nivel de aislamiento · aislamiento de instantánea (snapshot isolation) · write skew (sesgo de escritura) · retardo de replicación (replication lag) · hashing consistente · CQRS · event sourcing · cache-aside · expandir y contraer (expand and contract) · lakehouse · data mesh

Por qué los datos son la parte difícil

El código puede reescribirse en meses; los datos viven durante décadas. Las cuentas de clientes, los libros mayores, los pedidos y los historiales médicos sobreviven a todos los frameworks y a todos los servicios que los tocan, y moverlos es lento, arriesgado y caro. Por eso

las decisiones de este capítulo están entre las más significativas que toma un arquitecto, en el sentido del Capítulo 1: son difíciles de revertir, de amplio alcance y decisivas para atributos de calidad como la disponibilidad, el rendimiento y la modificabilidad.

El Capítulo 6 dividió los sistemas en servicios que se comunican a través de una red. Dividir el código es la mitad fácil. En cuanto hay varios servicios, alguien tiene que decidir quién puede cambiar cada dato, cómo lo ven los demás, qué garantías se mantienen cuando muchos usuarios actúan a la vez y qué ocurre cuando las copias de los datos no coinciden. El capítulo sigue dos ejemplos recurrentes de principio a fin: una plataforma de reparto de comida, con su hora punta de la cena, las ubicaciones de los repartidores y las cartas, y un banco en línea, donde perder o contar dos veces un solo pago es inaceptable.

Ser propietario de los datos, almacenarlos y protegerlos

Las cuatro primeras secciones tratan los datos operativos con los que funcionan los sistemas. Propiedad de los datos establece la regla más importante: todo hecho importante debe tener un único sistema de registro y un único escritor. Explica por qué una base de datos de integración compartida acopla los servicios por muy limpiamente que esté separado su código, cómo resolver los casos en que varios servicios quieren escribir los mismos datos, cómo leer datos de los que no eres propietario y cómo los contratos de datos hacen fiables los datos compartidos.

Elegir una base de datos repasa los almacenes relacionales, documentales, clave-valor, de columnas anchas, de grafos, de series temporales, de búsqueda, vectoriales y de SQL distribuido, y sostiene que la base de datos relacional sigue siendo la opción por defecto adecuada para los sistemas de registro, y que los almacenes especializados se ganan su lugar cumpliendo un requisito que la opción por defecto no puede cumplir. Transacciones y aislamiento va más allá del conocido acrónimo ACID, hasta las anomalías que causa la concurrencia, como las actualizaciones perdidas y el write skew, lo que garantizan realmente los niveles de aislamiento en PostgreSQL, MySQL, Oracle y SQL Server, y cuándo usar bloqueos, comprobaciones de versión o transacciones serializables. Replicación y particionamiento explica cómo las bases de datos copian y dividen los datos entre máquinas, qué efecto tienen el retardo de replicación y el failover sobre la corrección, y cómo elegir una clave de partición a partir de las cifras.

Derivar, almacenar en caché y hacer evolucionar los datos

La segunda mitad trata los datos que se derivan de los sistemas de registro y el cambio a lo largo del tiempo. CQRS y event sourcing separa el modelo que acepta los cambios de los modelos que responden a las consultas, y muestra cómo almacenar un historial de eventos en lugar del estado actual permite la auditoría y las consultas temporales, a un costo real en

complejidad. Caché cubre los patrones, los niveles y los modos de fallo de mantener copias cerca de donde se necesitan, desde el navegador hasta la base de datos, incluidas las estampidas que pueden tumbar un sistema cuando caduca una entrada popular.

Evolución de esquemas aborda el hecho de que todo esquema, ya sean tablas de base de datos, cargas útiles (payloads) de API o eventos, cambiará mientras el sistema está en funcionamiento, y muestra cómo las reglas de compatibilidad y el patrón expandir y contraer permiten el cambio sin tiempo de inactividad. Por último, analítica y streaming pasa de hacer funcionar el negocio a entenderlo: almacenes de datos, lagos de datos y lakehouses, procesamiento de flujos, y el data mesh como forma de organizar los datos analíticos en torno a los dominios.

Cómo encajan las secciones

Las secciones forman una cadena. La propiedad decide dónde se escribe cada hecho. La elección de la base de datos y sus garantías transaccionales deciden qué reglas pueden aplicarse a bajo costo dentro de ese propietario. La replicación y el particionamiento deciden cómo sobreviven a los fallos los datos del propietario y cómo crecen más allá de una máquina. Todo lo demás en el capítulo es una copia deliberada: modelos de lectura, cachés, índices de búsqueda, almacenes de datos y flujos. Cada copia es más rápida o más cómoda para algún propósito, y cada una está desactualizada en cierta medida. La evolución de esquemas permite que el propietario y todas sus copias cambien sin romperse entre sí.

El capítulo presupone el vocabulario de sistemas distribuidos del Capítulo 6, en especial los modelos de consistencia, la arquitectura dirigida por eventos y las sagas y el patrón outbox. Da paso al Capítulo 8, que protege estos datos, y al Capítulo 9, que mantiene rápidos y disponibles los sistemas que los contienen.

La idea que debes llevarte

Para cada dato importante, deberías poder responder tres preguntas: ¿quién es su único escritor?, ¿dónde se aplican sus reglas? y ¿cuán desactualizada puede estar cada copia? La mayoría de los problemas de datos de los sistemas reales, como pedidos que cambian de estado hacia atrás, informes que no coinciden con el libro mayor, reservas duplicadas y clientes que no pueden ver sus propios cambios, son fallos a la hora de responder deliberadamente a una de esas preguntas. Las técnicas de este capítulo son formas de responderlas bien, y cada una de ellas es un compromiso entre consistencia, disponibilidad, rendimiento, costo y el esfuerzo de operar el sistema.

SECCIÓN 7.1 DE 7.8 · INTERMEDIO · 26 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Propiedad de los datos y base de datos por servicio

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.5 Almacenar datos
- 6.3 Microservicios
- 6.4 Dirigida por eventos

Términos anteriores que utiliza: Saga, Consistencia eventual, Esquema, Transacción

LA IDEA EN BREVE

Piensa en un tablón de anuncios compartido en una cocina, donde cualquiera puede borrar y reescribir los pedidos del día. Funciona con dos cocineros. Con diez, los pedidos se sobrescriben, nadie sabe quién cambió qué y rediseñar el tablón implica pedir permiso a todos. Una cocina mejor da a cada puesto su propio tablón y su propia responsabilidad: la parrilla escribe los pedidos de la parrilla, y los demás puestos los leen o le piden a la parrilla que los cambie. La propiedad de los datos consiste en decidir, para cada información, qué parte de un sistema es la única autorizada a cambiarla.

LO QUE NECESITAS SABER

- Un sistema de registro es la fuente autorizada de un dato; la propiedad de los datos significa que un componente lo escribe, aplica sus reglas, controla su esquema y lo publica para los demás.
- Una base de datos compartida (de integración) acopla los servicios a través del esquema, el despliegue, la semántica y la carga en tiempo de ejecución; Fowler (2004) y la mayoría de los arquitectos desaconsejan usarla entre aplicaciones desarrolladas de forma independiente.
- La base de datos por servicio mantiene privados los datos de cada servicio (tablas, esquema o servidor privados) a costa de los joins, las transacciones y la integridad referencial entre servicios, que deben sustituirse por API, eventos, sagas y pipelines de informes.
- La propiedad conjunta puede resolverse dividiendo la tabla, con un pequeño dominio de datos compartido, delegando en un único propietario o consolidando servicios, cada opción con un compromiso distinto (The Hard Parts, 2021).

- Los datos de los que no eres propietario pueden leerse mediante llamadas síncronas, replicación de esquemas de columnas, cachés replicadas o eventos; toda copia es de solo lectura y posiblemente está desactualizada.
- Los contratos de datos establecen el esquema, el significado, la calidad, los niveles de servicio y la propiedad de los datos compartidos entre equipos, y son más útiles cuando se comprueban automáticamente.
- Para cada hecho importante, un buen diseño sabe nombrar su único escritor, dónde se aplican sus reglas y cuán desactualizada puede estar cada copia.

Por qué la propiedad es la primera pregunta sobre los datos

El Capítulo 6 dividió los sistemas en servicios que se despliegan, se escalan y se cambian de forma independiente. El código se divide con relativa facilidad; los datos, no. Un pedido hace referencia a un cliente, un plato de la carta, un restaurante y un repartidor, y cada equipo quiere leer, y a menudo cambiar, alguna parte de él. Por eso, la pregunta más importante en una arquitectura de datos distribuida no es qué base de datos comprar, sino **quién es el propietario de cada dato**: qué componente puede cambiarlo, cuál aplica sus reglas y cómo llegan a verlo todos los demás.

La propiedad importa porque los datos sobreviven al código. Los servicios se reescriben cada pocos años; las cuentas de clientes, los libros mayores y los historiales médicos se conservan durante décadas. Equivocarse con la propiedad produce el tipo caro de deuda arquitectónica: dos servicios que no pueden lanzarse por separado porque comparten tablas, un campo de estado que tres equipos sobrescriben por turnos, o un informe mensual que nadie puede reproducir porque las cifras salen de cinco copias de los mismos datos que no coinciden.

Esta sección usa dos ejemplos recurrentes que vuelven a aparecer a lo largo del capítulo: una plataforma de reparto de comida (clientes, restaurantes, repartidores, pedidos) y un banco en línea (clientes, cuentas, pagos, un libro mayor).

Sistema de registro y propiedad de los datos

Un sistema de registro es la fuente autorizada de un dato concreto: el lugar cuyo valor prevalece cuando las copias no coinciden. En el banco, el libro mayor central es el sistema de registro de los saldos de las cuentas; el saldo que se muestra en la app para dispositivos móviles, el que está en la caché del motor de fraude y el de la carga de anoche en el almacén de datos son todos copias. En la plataforma de reparto de comida, el propio servicio de cartas del restaurante es el sistema de registro de los precios, aunque el índice de búsqueda, la caché sin conexión de la app para dispositivos móviles y las líneas de pedido contengan copias de ellos.

La propiedad de los datos convierte esto en una responsabilidad. El componente propietario (y el equipo que hay detrás) es el único que escribe los datos, aplica sus *invariantes* (reglas que deben cumplirse siempre, como "un asiento del libro mayor nunca se edita, solo se revierte"), decide su esquema y los publica para los demás a través de una interfaz que controla. La propiedad es una decisión de diseño, no una propiedad de la base de datos: dos servicios pueden estar en el mismo servidor de base de datos y tener aun así una propiedad clara, y dos servicios con bases de datos separadas pueden seguir peleándose por los mismos hechos si el límite entre ellos está mal trazado.

Una regla útil de *Software Architecture: The Hard Parts* (Neal Ford, Mark Richards, Pramod Sadalage y Zhamak Dehghani, 2021) es que el servicio que escribe en una tabla es su propietario. Compartir la lectura es barato; compartir la escritura no lo es, porque quien escribe debe aplicar las reglas.

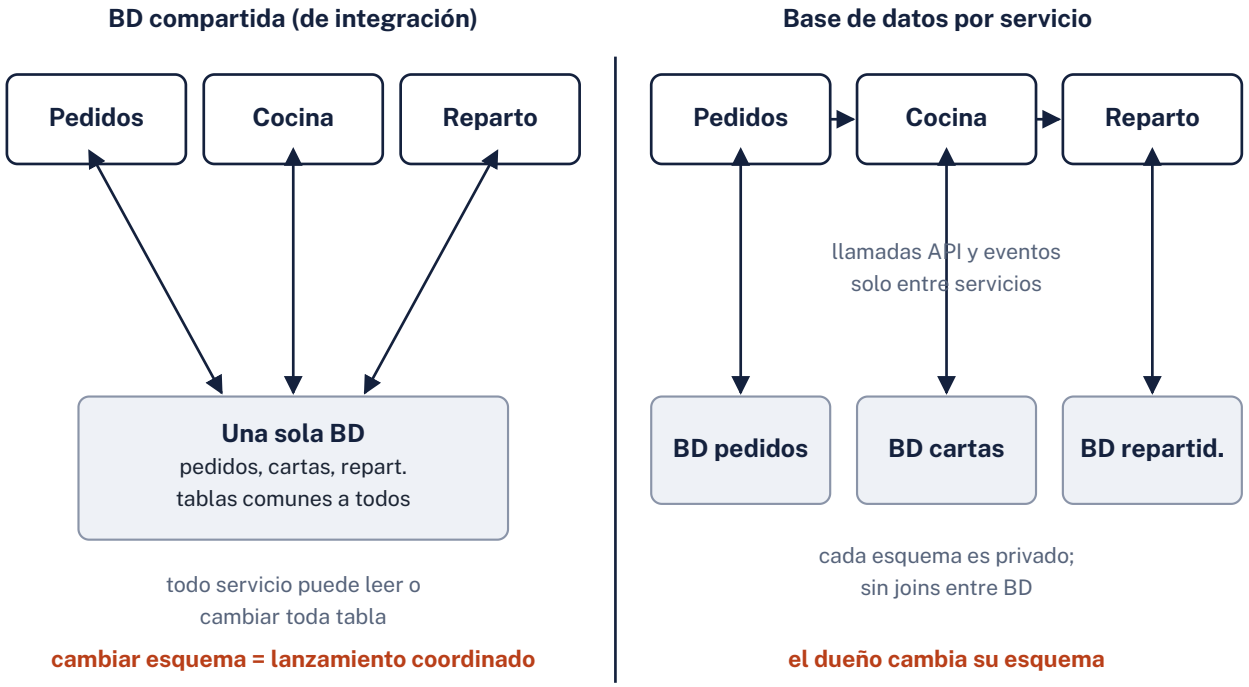
La base de datos compartida y por qué acopla los servicios

La forma más sencilla de permitir que varias aplicaciones trabajen con los mismos datos es darles a todas acceso a una única base de datos. Martin Fowler la llamó base de datos de integración (2004): una base de datos que es el punto de integración entre aplicaciones, a diferencia de una *base de datos de aplicación*, que sirve a una sola aplicación. Parece atractiva: no hay que construir API, los joins funcionan sobre todo y una sola transacción puede actualizar a la vez pedidos y pagos. El veredicto de Fowler, que comparte la mayoría de los arquitectos, es que las bases de datos de integración deben evitarse, porque el acoplamiento que crean es profundo y en gran parte invisible.

Piensa en las formas de acoplamiento que introduce. **Acoplamiento de esquema**: cada tabla y cada columna es, en la práctica, una API pública. Renombrar `orders.state` como `orders.status` rompe cualquier servicio que la lea, y nadie puede estar seguro de cuáles lo hacen, porque las consultas SQL están dispersas por muchas bases de código, informes y hojas de cálculo. **Acoplamiento de despliegue**: un cambio de esquema tiene que coordinarse entre todos los equipos que tocan la tabla, así que servicios que debían lanzarse de forma independiente se lanzan juntos. **Acoplamiento semántico**: el esquema debe servir a todos los clientes, así que se vuelve más general y más confuso (Fowler señala que acaba intentando unificar conceptos que pertenecen a contextos delimitados distintos), y las reglas de negocio de una tabla acaban duplicadas, de forma incoherente, en cada servicio que escribe en ella. **Acoplamiento en tiempo de ejecución**: la consulta lenta de un informe o el proceso por lotes con muchos bloqueos de un servicio degrada a todos los demás, y la base de datos se convierte en un punto único de fallo compartido y en un techo para las conexiones.

Nada de esto es fatal para una sola aplicación o para un monolito modular cuyos módulos respetan las tablas de los demás, ni para la arquitectura basada en servicios de grano grueso, en la que un puñado de servicios de dominio suele compartir una base de datos de forma deliberada. Se convierte en un problema cuando los equipos esperan desplegabilidad independiente pero mantienen las tablas compartidas, que es una vía clásica hacia un *monolito distribuido*.

Base de datos compartida frente a base de datos por servicio



Con una base de datos compartida, cada tabla es una interfaz pública; con una base de datos por servicio, cada esquema es privado y los datos cruzan los límites solo mediante API y eventos.

Base de datos por servicio

La alternativa, asociada a los microservicios, es el patrón de base de datos por servicio: los datos persistentes de cada servicio son privados y los demás servicios solo pueden acceder a ellos a través de la API del servicio o de los eventos que publica. El catálogo de patrones de Chris Richardson (microservices.io) señala que esto no tiene por qué significar un servidor distinto por servicio. Hay tres niveles de separación, y la línea importante es la primera.

Tres formas de mantener privados los datos de un servicio

Nivel	Qué está separado	Solidez del límite	Uso típico
Tablas privadas por servicio	Cada servicio tiene su propio conjunto de tablas en una base de datos compartida, con credenciales separadas que solo pueden acceder a esas tablas.	La más débil: depende de los permisos y de la disciplina; sigue compartiendo capacidad y fallos.	Al principio de una migración; sistemas pequeños; monolitos modulares.
Esquema por servicio	Cada servicio tiene su propio esquema (espacio de nombres) en un servidor compartido.	Propiedad y control de acceso claros; sigue compartiendo el rendimiento y la disponibilidad del servidor.	La mayoría de los equipos que empiezan con varios servicios; más barato de operar.
Servidor de base de datos por servicio	Cada servicio tiene su propia instancia de base de datos, posiblemente de un tipo distinto.	La más sólida: escalado, fallos y elección tecnológica separados.	Servicios con necesidades muy distintas de carga, forma de los datos o disponibilidad.

Los beneficios son el reflejo de los costos de la base de datos compartida: los propietarios cambian sus esquemas libremente, escalan y ajustan sus almacenes de forma independiente, eligen el tipo de base de datos adecuado y contienen los fallos. Los costos son reales y deben constar en cualquier registro de decisión. Los joins entre servicios desaparecen y deben sustituirse por composición de API o datos replicados. Una sola transacción ACID ya no puede abarcar pedidos y pagos; los cambios entre servicios se convierten en sagas con pasos compensatorios. La integridad referencial entre servicios (todo pedido apunta a un cliente real) ya no la garantizan las claves foráneas. Los informes necesitan un pipeline aparte. Y hay más bases de datos que respaldar, parchear y monitorizar.

The Hard Parts plantea la decisión como un equilibrio de fuerzas. Los **desintegradores** empujan hacia la división de los datos: el control de cambios (cuántos servicios se rompen cuando cambia una tabla), los límites de conexiones, la escalabilidad, la tolerancia a fallos, el deseo de tener unidades desplegables de forma independiente y la necesidad de distintos tipos de base de datos. Los **integradores** tiran de los datos para mantenerlos juntos: las relaciones que necesitan integridad referencial y las operaciones que de verdad necesitan una única transacción atómica. Si dos tablas siempre se cambian juntas en una transacción y no tienen vida independiente, eso es una prueba sólida de que pertenecen al mismo propietario.

Cuando varios servicios escriben los mismos datos

Las preguntas sobre la propiedad se vuelven difíciles cuando más de un servicio cambia legítimamente la misma entidad. *The Hard Parts* distingue tres situaciones. Con la **propiedad única**, solo un servicio escribe en la tabla y es, sin más, su propietario. Con la **propiedad común**, la mayoría de los servicios escriben en la misma tabla (un registro de auditoría o un conjunto de parámetros de configuración, por ejemplo); la respuesta habitual es crear un servicio dedicado que sea el propietario de los datos y dar a todos los demás una API, a menudo asíncrona para no ralentizar a quienes escriben. Con la propiedad conjunta, un número pequeño de servicios, normalmente dos, escribe en la misma tabla, por lo general dentro del mismo dominio. El libro ofrece cuatro técnicas para la propiedad conjunta.

Resolver la propiedad conjunta

Técnica	Cómo funciona	Compromiso principal
División de la tabla	Dividir la tabla para que cada servicio sea propietario de las columnas que escribe, en su propia tabla. La fila de `orders` de la plataforma de reparto de comida se convierte en un pedido cuyo propietario es Pedidos y en una comanda de cocina cuyo propietario es Cocina.	Propiedad limpia, pero crear o eliminar la entidad padre tiene que sincronizarse ahora entre servicios, y cualquier vista del conjunto necesita ambas partes.
Dominio de datos	Mantener la tabla compartida entre los pocos servicios implicados, dentro de un pequeño esquema compartido que controlan conjuntamente.	Sencillo y rápido, y las transacciones siguen siendo posibles, pero es una base de datos compartida deliberadamente limitada: los servicios quedan acoplados en los cambios de esquema y el límite entre ellos se debilita.
Delegar	Elegir un único propietario; el otro servicio le envía solicitudes (llamadas síncronas o comandos asíncronos) en lugar de escribir. El propietario se elige por dominio principal (¿de quién es el concepto?) o por características operativas (¿qué servicio necesita las escrituras más rápidas y con mayor disponibilidad?).	Propiedad clara y reglas en un solo lugar, a costa de un salto de red adicional, un nuevo modo de fallo y posiblemente consistencia eventual para el servicio que delega.
Consolidación de servicios	Fusionar en un solo servicio los servicios que comparten los datos.	Elimina el problema y recupera las transacciones locales, pero crea un servicio más grande, con despliegue, escalado y aislamiento de fallos de grano más grueso.

Ninguna técnica es gratuita, y la adecuada depende de con qué frecuencia cambian los datos, de cuán estrictas son sus reglas y de cuán independientes necesitan ser los equipos. En la práctica, delegar es la respuesta más habitual para las entidades de negocio con un ciclo de vida (pedidos, pagos, reservas), la división de la tabla funciona bien cuando las columnas escritas conjuntamente son en realidad hechos distintos que casualmente se almacenaban juntos, y la consolidación es la respuesta honesta cuando los dos servicios se dividieron con un grano demasiado fino desde el principio.

Leer datos de los que no eres propietario

Una vez resuelta la propiedad, todos los demás servicios siguen necesitando *leer* los datos. La pantalla de cocina necesita las notas de entrega del cliente; el motor de fraude necesita los pagos recientes de la cuenta. Hay cuatro grandes opciones, y cada una intercambia frescura por acoplamiento. Ten en cuenta que los diseños dirigidos por eventos, tratados en el Capítulo 6, proporcionan la tercera y la cuarta.

Formas de leer los datos de otro servicio

Opción	Cómo funciona	Frescura	Acoplamiento y costo
Llamada síncrona	Consultar la API del propietario en el momento en que se necesitan los datos.	Actuales, en el momento de la llamada.	Acoplamiento en tiempo de ejecución: si el propietario está lento o caído, también lo está quien llama; las latencias se suman a lo largo de una cadena.
Replicación de esquemas de columnas	Copiar las pocas columnas necesarias (por ejemplo, el nombre y la dirección del restaurante) en la base de datos propia del lector, mantenidas al día mediante eventos o captura de datos de cambios.	Consistencia eventual: desactualizados según el retardo de replicación.	Lecturas y joins locales rápidos; el lector debe gestionar las actualizaciones tardías o perdidas y no debe escribir nunca en la copia.
Caché replicada	Un grid de datos en memoria (como Hazelcast o Apache Ignite) mantiene una copia de los datos del propietario en cada servicio consumidor, sincronizada por el grid.	Casi en tiempo real.	Lecturas muy rápidas; costo de memoria, carga al arrancar y complejidad operativa del grid.
Eventos (transferencia de estado transportada por eventos)	El propietario publica eventos que transportan los datos modificados; los lectores guardan lo que necesiten.	Consistencia eventual.	Acoplamiento temporal débil; los lectores deben hacer frente a los duplicados, al orden y a la evolución de esquemas.

Fowler llama a la cuarta opción transferencia de estado transportada por eventos: eventos como `MenuItemPriceChanged` transportan los nuevos valores, así que los consumidores no necesitan volver a llamar. Un mecanismo relacionado, la captura de datos de cambios (CDC), lee el log de la base de datos del propietario y convierte cada cambio confirmado en un evento, lo que evita el *problema de la escritura dual* (escribir en la base de datos y publicar un evento por separado, de modo que una operación puede tener éxito mientras la otra falla). La bandeja de salida transaccional (transactional outbox), que resuelve el mismo problema desde dentro de la aplicación, se trata junto con las sagas.

ATENCIÓN

Una copia es una caché, no un segundo maestro

Los datos replicados solo son seguros si se tratan como de solo lectura y posiblemente desactualizados. En el momento en que un consumidor empieza a corregir su copia (arreglando localmente la dirección de un restaurante porque la actualización del propietario se retrasó), hay dos sistemas de registro y se irán distanciando. Decide para cada copia cuán desactualizada puede estar, qué ocurre cuando es incorrecta y cómo se reconstruye a partir del propietario cuando se pierde.

SI ESTO ES NUEVO PARA TI

Paso a paso: preguntar al propietario o guardar una copia

La idea más difícil de esta sección es cómo debe usar un servicio los datos de los que no es propietario. Puede preguntar al propietario cada vez, o guardar su propia copia, y cada opción intercambia frescura por dependencia del propietario.

- Para enviar a los repartidores a la puerta correcta, Reparto debe saber dónde está cada restaurante y cómo se llama. Esos datos pertenecen al servicio Restaurante, que es su sistema de registro.
- Opción uno: Reparto llama a la API de Restaurante cada vez que asigna un repartidor. La respuesta siempre está al día, pero si Restaurante está lento o caído, las asignaciones de repartidores también se ralentizan o se atascan, aunque a Reparto en sí no le pase nada.
- Opción dos: Reparto copia solo esas dos columnas en su propia base de datos, lo que la tabla anterior llama replicación de esquemas de columnas. Cada vez que cambian los datos de un restaurante, Restaurante publica un evento que transporta los nuevos valores y Reparto actualiza su copia: la transferencia de estado transportada por eventos del Capítulo 6. Ahora las lecturas son locales y rápidas, y siguen funcionando mientras Restaurante está caído.
- El precio es la desactualización. Cuando un restaurante corrige su dirección, Reparto conserva la antigua hasta que llega el evento. Si el evento se pierde, la copia sigue siendo incorrecta hasta que se actualiza a partir del propietario.
- Por eso la copia es de solo lectura. Si Reparto detecta una dirección incorrecta, se lo comunica a Restaurante, que la corrige y publica la corrección; editar la copia crearía un segundo maestro, como explica la advertencia anterior.
- **La lección:** preguntar al propietario da datos frescos, pero te ata a él mientras funcionas; una copia te libera, pero es de solo lectura y posiblemente está desactualizada. Para cada copia, decide cuán desactualizada puede estar y cómo se reconstruye a partir del propietario.

Informes que abarcan varios servicios

La base de datos por servicio complica una tarea antigua: el informe transversal. “Ingresos por restaurante y por semana, descontando los reembolsos, desglosados por tipo de repartidor” necesita pedidos, pagos, restaurantes y entregas. Generarlo llamando a cuatro API página a página es lento y carga a los servicios operativos con el trabajo de los informes. Consultar directamente sus bases de datos reintroduce por la puerta de atrás el acoplamiento de la base de datos compartida.

La respuesta habitual es separar los informes de las operaciones. Fowler describió una *base de datos de informes*: un almacén separado, con una forma pensada para las consultas y alimentado desde los sistemas operativos, que los usuarios de informes pueden leer libremente. Hoy suele ser un almacén de datos o un lakehouse alimentado por CDC o por eventos, que se trata en analítica y streaming. Para vistas pequeñas y casi en tiempo real dentro del producto (el panel de los pedidos del día de un restaurante), un servicio puede construir su propio modelo de lectura a partir de eventos, la idea que hay detrás de CQRS. En cualquier caso, los pipelines que alimentan los informes dependen de las estructuras de datos de los propietarios, y por eso necesitan contratos.

Contratos de datos

Un contrato de datos es un acuerdo explícito y versionado entre el propietario de un conjunto de datos y sus consumidores. Establece el esquema (campos, tipos, cuáles pueden ser nulos), el significado de cada campo (¿`amount` está en peniques o en libras, con IVA o sin él?), las expectativas de calidad (ningún identificador de pedido duplicado, como máximo un 0.1% de filas sin restaurante), los niveles de servicio (cuán frescos son los datos, cuánto tiempo se conservan), quién es su propietario y cómo se anuncian los cambios. Aplica la idea de un contrato de API a los datos que circulan entre equipos mediante eventos, flujos de CDC y tablas del almacén de datos.

Los contratos son más útiles cuando son legibles por máquina y se comprueban automáticamente: un pipeline que rechaza un cambio de esquema que rompería a los consumidores registrados, o una prueba que falla cuando se incumple una regla de calidad. El Open Data Contract Standard (ODCS), mantenido por el proyecto Bitol bajo la Linux Foundation AI & Data, define un formato YAML que abarca el esquema, las reglas de calidad, los niveles de servicio, los roles y los canales de soporte. Los contratos de datos no son gratuitos: alguien tiene que escribirlos y mantenerlos, y los contratos demasiado estrictos frenan el cambio. *The Hard Parts* contrapone los contratos estrictos (precisos, verificables, con un acoplamiento fuerte) a los flexibles (más fáciles de hacer evolucionar, pero que necesitan más pruebas), y propone como término medio los contratos dirigidos por el consumidor, en los que los consumidores indican lo que realmente usan. Cómo cambiar un esquema sin romper a los consumidores es el tema de evolución de esquemas.

EJEMPLO PRÁCTICO

Ejemplo práctico: ¿quién es el propietario del estado del pedido en la plataforma de reparto de comida?

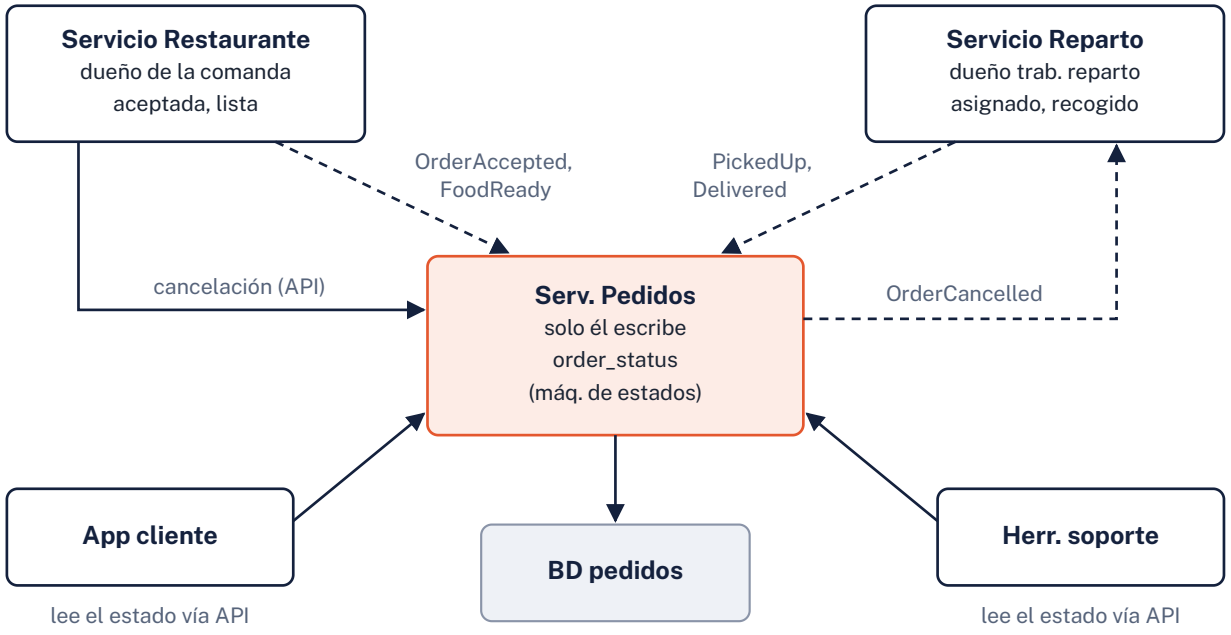
La plataforma tiene tres servicios a los que les importa el progreso de un pedido: Pedidos (recibe los pedidos y gestiona su pago), Restaurante (la tableta de la cocina: acepta pedidos, marca la comida como lista) y Reparto (asigna repartidores, sigue la recogida y la entrega). Hoy los tres escriben en la columna `status` de una tabla `orders` compartida. Los clientes han visto pedidos que pasan de “Recogido” a “En preparación”, y nadie sabe decir dónde están las reglas de cancelación.

Diagnosticar	Propiedad conjunta de una columna por parte de tres escritores. Cada uno escribe lo que sabe, gana la última escritura y los mensajes que llegan tarde hacen retroceder el estado. La regla “un pedido solo puede cancelarse antes de que la cocina empiece” se comprueba en dos servicios, de forma distinta.
Separar los hechos	En realidad hay tres hechos: el estado de la comanda de cocina (cuyo propietario es Restaurante), el estado del trabajo de reparto (cuyo propietario es Reparto) y el estado del pedido que ve el cliente, que combina ambos y contiene las reglas de negocio (cancelación, reembolsos, compensación).
Opción: división de la tabla	Dar a cada servicio su propia columna de estado y dejar que los lectores los combinen. Descartada: cada lector (la app, la herramienta de soporte, las notificaciones) volvería a implementar la combinación, y la regla de cancelación seguiría sin tener un lugar propio.
Opción: consolidación de servicios	Fusionar los tres en un solo servicio. Descartada: tres equipos, cargas muy distintas (las actualizaciones de ubicación de los repartidores llegan cada pocos segundos) y ciclos de lanzamiento diferentes.
Decisión: delegar	Pedidos es el único escritor de `order_status`. Restaurante y Reparto son propietarios de sus propios hechos y publican eventos (`OrderAccepted` , `FoodReady` , `PickedUp` , `Delivered`). Pedidos los consume y aplica una máquina de estados. Una cancelación desde la tableta del restaurante es una solicitud a la API de Pedidos, que la acepta o la rechaza.
Máquina de estados	PLACED, ACCEPTED, PREPARING, READY, PICKED_UP, DELIVERED, con CANCELLED permitido solo desde PLACED o ACCEPTED. Las transiciones solo avanzan, así que un `FoodReady` tardío que llega después de `PickedUp` se registra pero no cambia el estado; los duplicados se ignoran.
Consecuencias	Un solo lugar para las reglas; ningún salto hacia atrás; cada equipo lanza de forma independiente. Costos: el estado va por detrás de la cocina según el retardo de los eventos (normalmente muy por debajo de un segundo), Pedidos debe estar disponible para aceptar cancelaciones y los eventos se convierten en un contrato que debe versionarse.

La jugada clave no fue elegir una técnica de la tabla, sino darse cuenta de que una columna ocultaba tres hechos distintos con tres propietarios naturales.

Deja constancia de la decisión en un registro de decisiones de arquitectura, incluidas las opciones descartadas; el próximo equipo que pregunte “¿por qué no podemos actualizar el estado sin más desde la app del repartidor?” lo necesitará.

El estado del pedido después de la decisión



Cada hecho tiene un solo escritor; los demás leen o solicitan

Cada hecho tiene un solo escritor: Restaurante y Reparto publican aquello de lo que son propietarios, y solo el servicio Pedidos convierte esos eventos en el estado que ve el cliente.

Errores comunes

Dividir los servicios pero mantener las tablas compartidas. Los equipos cargan con el costo operativo de la distribución sin la independencia que se suponía que iba a comprar. Si los servicios no pueden desplegarse sin coordinar un cambio de esquema, no son independientes. **Servicios de entidad.** Un `CustomerService` que no es más que crear, leer, actualizar y eliminar sobre una tabla, y al que llaman todos los demás servicios, es una base de datos compartida con latencia adicional; la propiedad debe seguir a las capacidades de negocio y sus reglas, no a los sustantivos. **Copias que se convierten discretamente en maestros.** Consulta la advertencia anterior. **Dejar los informes para el final.** El primer informe financiero tras la división suele revelar que nadie planificó cómo volver a reunir los datos. **Tratar la propiedad como algo**

puramente técnico. La propiedad también es organizativa: alguien debe responder de la calidad de los datos y, según la ley de Conway, los límites de datos que atraviesan los límites entre equipos tienden a no durar.

Una buena prueba para cualquier diseño de datos es hacerse, para cada hecho importante, tres preguntas: ¿quién es el único escritor?, ¿dónde se aplican sus reglas? y ¿cuán desactualizada puede estar cada copia? Si alguna respuesta es “depende” o “varios servicios”, el diseño no está terminado.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **Dos servicios actualizan cada uno la columna `status` de la misma tabla. ¿Qué tipo de propiedad es esta? Nombra dos formas de resolverla.**

Propiedad conjunta. Puede resolverse dividiendo la tabla, delegando las escrituras en un único propietario, manteniendo un pequeño dominio de datos compartido o consolidando los servicios.

▼ **¿Por qué una base de datos compartida socava el despliegue independiente?**

Cada tabla es, en la práctica, una interfaz pública, así que un cambio de esquema debe coordinarse con todos los servicios que la leen o escriben, lo que obliga a lanzamientos conjuntos.

▼ **¿Cuál es la diferencia entre la replicación de esquemas de columnas y una llamada síncrona a la hora de leer los datos de otro servicio?**

La replicación mantiene una copia local con consistencia eventual que es rápida y está disponible aunque el propietario esté caído; una llamada síncrona devuelve datos actuales, pero acopla la disponibilidad y la latencia de quien llama a las del propietario.

▼ **¿Qué añade un contrato de datos además de un esquema?**

El significado de los campos, las expectativas de calidad, los niveles de frescura y de conservación, la propiedad y cómo se anuncian y versionan los cambios.

▼ **Inténtalo: Reservas, Pagos y Taquilla actualizan la columna status de una tabla compartida tickets, y a veces las entradas pasan de Reembolsado a Pagado. ¿Quién debería ser dueño de status, y cómo deberían los demás cambiarlo o leerlo?**

Nombra a un único escritor. Reservas debe ser dueño del estado de la entrada y hacer cumplir los cambios permitidos en un solo lugar, incluida la regla de que Refunded no puede pasar a Paid. Pagos y la Taquilla no lo escriben; envían un comando o publican un evento, como RefundCompleted, y Reservas decide. Los demás leen a través de la API de Reservas o mantienen copias de solo lectura, aceptando que estén algo desactualizadas. El detalle propio de cada equipo, como el estado del pago, se queda en su propio almacén. Salir de la tabla compartida viene al final.

FUENTES

- Martin Fowler: IntegrationDatabase (2004): https://martinfowler.com/bliki/IntegrationDatabase.html
- Martin Fowler: ReportingDatabase: https://martinfowler.com/bliki/ReportingDatabase.html
- Chris Richardson, microservices.io: Pattern: Database per service: https://microservices.io/patterns/data/database-per-service.html
- Chris Richardson, microservices.io: Pattern: Shared database: https://microservices.io/patterns/data/shared-database.html
- Bitol (Linux Foundation AI & Data): Open Data Contract Standard: https://bitol-io.github.io/open-data-contract-standard/latest/

SECCIÓN 7.2 DE 7.8 · INTERMEDIO · 26 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Elegir una base de datos: relacional, documental, clave-valor, de columnas, de grafos y de series temporales

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.5 Almacenar datos
- 6.6 CAP y consistencia
- 7.1 Propiedad de los datos

Términos anteriores que utiliza: Sistema de registro (system of record), Transacción, SQL (Structured Query Language), Captura de datos de cambios (CDC), Registro de decisiones de arquitectura (ADR)

LA IDEA EN BREVE

Elegir una base de datos es como elegir dónde guardar las cosas en casa. Un archivador con carpetas etiquetadas (una base de datos relacional) conviene para los papeles que necesitas cotejar entre sí. Un armario con conjuntos completos de ropa (un almacén documental) conviene para las cosas que siempre sacas juntas. Una fila de casilleros numerados (un almacén clave-valor) es lo más rápido cuando sabes el número. Un tablero de corcho con hilos entre chinchetas (un grafo) muestra quién conoce a quién. La mayoría de las casas necesitan un archivador y uno o dos muebles especializados, no uno de cada, porque cada mueble adicional necesita espacio, limpieza y alguien que sepa cómo funciona.

LO QUE NECESITAS SABER

- Las bases de datos relacionales siguen siendo la opción por defecto para los datos de negocio porque ofrecen transacciones, restricciones, consultas flexibles, herramientas maduras y conocimientos ampliamente disponibles, y ahora también manejan datos JSON, de texto, geoespaciales y vectoriales a escala moderada.
- Los almacenes documentales, clave-valor, de columnas anchas, de grafos, de series temporales, de búsqueda y vectoriales hacen baratos ciertos patrones de acceso y caros otros; destacan como almacenes especializados o derivados.
- El SQL distribuido (Spanner, CockroachDB, YugabyteDB, TiDB) mantiene SQL y las transacciones al tiempo que escala entre nodos y regiones, a costa de latencia de escritura, dinero y complejidad operativa.

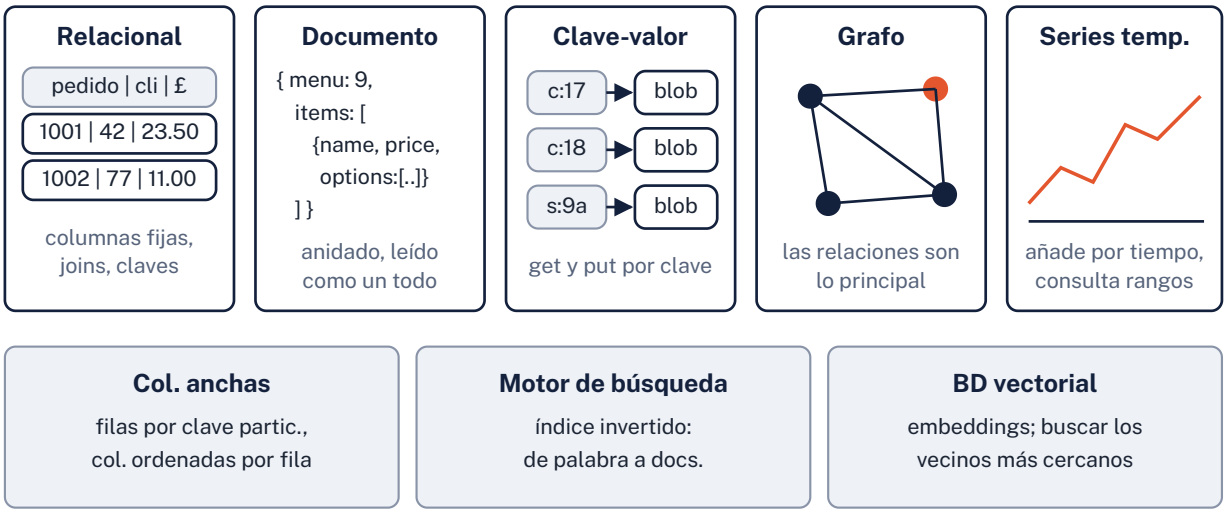
- Elige para cada conjunto de datos a partir de pruebas: forma de los datos, patrones de acceso, necesidades de consistencia, escala medida, carga operativa y conocimientos del equipo; anota primero las consultas y los volúmenes clave.
- Separa los sistemas de registro, que necesitan la tecnología más conservadora, de los almacenes derivados, como los índices de búsqueda y las cachés, que pueden ser especializados porque pueden reconstruirse.
- La persistencia políglota permite que cada conjunto de datos use un almacén adecuado, pero cada tecnología adicional añade un costo operativo y cognitivo duradero; el ensayo de McKinley 'Choose Boring Technology' (2015) aboga por gastar la innovación con moderación.

Por qué la elección merece atención

Una base de datos es uno de los componentes más difíciles de sustituir. Las aplicaciones se reescriben en torno a sus datos, no al revés; migrar terabytes de datos vivos mientras el negocio sigue funcionando puede llevar un año. Por eso, la elección del almacén es una puerta de un solo sentido que merece una decisión explícita, mientras que muchos equipos la toman por costumbre o por moda.

El Capítulo 0 presentó las bases de datos relacionales y las cuatro familias NoSQL. Esta sección va más allá: en qué es realmente buena cada familia, las categorías más recientes (series temporales, búsqueda, vectoriales y SQL distribuido), cómo elegir a partir de pruebas y no del entusiasmo, y cuánto cuesta operar varios tipos a la vez. La respuesta rara vez es una sola base de datos para todo, y casi nunca una base de datos distinta para cada servicio.

Los modelos de datos de un vistazo



Cada familia organiza los datos de forma distinta, y esa forma decide qué consultas son baratas y cuáles son imposibles.

Relacional: sigue siendo la opción por defecto, por buenas razones

Una base de datos relacional almacena los datos en tablas con tipos, claves y restricciones declarados, y responde a consultas SQL declarativas, una idea del artículo de Edgar Codd de 1970. Cincuenta años después sigue siendo la opción por defecto sensata para los datos de negocio, por cuatro razones. Primera, **las transacciones y las restricciones**: la propia base de datos garantiza que un pago y sus asientos en el libro mayor se confirman juntos y que ningún pedido apunta a un cliente inexistente, así que estas reglas no tienen que reimplementarse, de forma imperfecta, en cada servicio. Segunda, **la flexibilidad de las consultas**: no necesitas conocer todas las preguntas futuras cuando diseñas el esquema; un nuevo informe es una nueva consulta, no una nueva tabla. Tercera, **la madurez**: décadas de herramientas para copias de seguridad, replicación, monitorización, migración y diagnóstico de rendimiento, e ingenieros que ya la conocen. Cuarta, **la amplitud**: los sistemas relacionales modernos almacenan e indexan JSON (el `jsonb` de PostgreSQL) y admiten búsqueda de texto completo, datos geoespaciales (PostGIS), particionado por tiempo y similitud vectorial (la extensión pgvector), así que una sola base de datos bien gestionada puede cubrir muchas necesidades a escala moderada.

Los límites reales son concretos. Las escrituras en un único primario están limitadas por una sola máquina; las tablas muy grandes necesitan un particionamiento cuidadoso; los cambios de esquema en tablas enormes requieren técnica (evolución de esquemas); y algunos patrones de acceso, como los recorridos profundos de grafos o la búsqueda de texto ordenada por

relevancia, los resuelven mucho mejor motores especializados. Un solo servidor moderno con unidades de estado sólido rápidas puede atender miles de transacciones por segundo y terabytes de datos, lo que cubre las necesidades de la mayoría de las organizaciones.

Elegir identificadores: secuencias, UUIDv7 y ULID

Las claves son difíciles de cambiar más adelante. Una **secuencia** es compacta y ordenada, pero necesita un único asignador, y el número de pedido 1042 revela tu volumen a los de fuera. Un **UUIDv4** aleatorio puede crearse en cualquier sitio, pero el RFC 9562 advierte de su mala localidad en los índices: las inserciones caen en lugares aleatorios de un árbol B, y el efecto "puede ser drástico". **UUIDv7** pone primero una marca de tiempo de 48 bits en milisegundos y después bits aleatorios, de modo que los valores nuevos quedan cerca unos de otros; PostgreSQL 18 incluye `uuidv7()`, y el RFC aconseja almacenar los UUID en binario. **ULID** es similar: 48 bits de tiempo, 80 de aleatoriedad y 26 caracteres de texto. Los ID ordenados por tiempo revelan cuándo se creó una fila y pueden crear un rango caliente con el particionamiento por rango de claves. El RFC también advierte contra suponer que los UUID son difíciles de adivinar, así que la autorización no debe depender de ellos.

Las familias no relacionales

Una base de datos documental almacena cada registro como un documento autocontenido, normalmente JSON o una forma binaria de JSON. Encaja con los *agregados* en el sentido del diseño guiado por el dominio: cosas que se leen y se escriben como un todo, como la carta de un restaurante con sus secciones, platos y grupos de opciones. Leer un documento es una sola búsqueda, sin joins. El compromiso aparece cuando hay que consultar los mismos datos con otra forma (todos los platos que contienen maní (cacahuete) en todas las cartas) o cambiarlos en muchos documentos a la vez; los productos modernos tienen joins y transacciones entre varios documentos, pero son más costosos y menos naturales que en SQL.

Un almacén clave-valor es un diccionario distribuido: guardas un valor bajo una clave y lo recuperas. Como el almacén nunca mira dentro de los valores, puede ser extremadamente rápido y fácil de particionar. Redis (con licencias de código fuente disponible desde 2024 y con una opción de código abierto AGPL desde Redis 8) y Valkey, su bifurcación con licencia BSD, mantienen los datos en memoria con persistencia opcional y añaden estructuras de datos (conjuntos ordenados, flujos, índices geoespaciales) y caducidad; Amazon DynamoDB es un almacén gestionado y particionado clave-valor y documental. Encajan para sesiones, carritos, contadores de limitación de tasa, feature flags, cachés y la última posición de cada repartidor.

Un almacén de columnas anchas (Bigtable, Apache Cassandra, ScyllaDB, HBase) agrupa las filas por una clave de partición y las almacena ordenadas por una clave de agrupación (clustering key) dentro de cada partición. Está diseñado para volúmenes masivos de escritura

repartidos entre muchos nodos y centros de datos. La disciplina que exige es el *modelado a partir de las consultas*: diseñas una tabla por consulta, duplicando datos libremente, porque las consultas que no parten de la clave de partición son caras o imposibles. Cassandra desciende en parte del artículo de Amazon sobre Dynamo, de 2007, cuyas ideas de replicación aparecen en dos secciones más adelante.

Una base de datos de grafos almacena los nodos y las relaciones como registros de primera clase y responde a consultas de recorrido como "clientes a tres saltos o menos de esta red de fraude" sin los joins en cascada que hacen lentas esas consultas en SQL. Cypher, de Neo4j, es el lenguaje de consulta más conocido, e ISO publicó en 2024 un lenguaje estándar de consulta de grafos, GQL (ISO/IEC 39075). Los grafos son excelentes para la detección de fraude, las recomendaciones y el análisis de redes, y suelen ser un mal almacén principal para los datos transaccionales de negocio.

Almacenes de series temporales, de búsqueda y vectoriales

Una base de datos de series temporales está optimizada para mediciones con marca de tiempo que llegan en orden y rara vez se actualizan: métricas, lecturas de sensores, precios, historial de ubicaciones. Comprime mucho, particiona por tiempo, reduce la resolución de los datos antiguos (conservar cada punto durante una semana y promedios por minuto durante un año) y elimina a bajo costo los datos caducados descartando particiones de tiempo enteras. Algunos ejemplos son InfluxDB, Prometheus (para la monitorización) y TimescaleDB, que es una extensión de PostgreSQL y muestra con qué frecuencia una base de datos relacional puede estirarse para encajar.

Un motor de búsqueda como Elasticsearch, OpenSearch o Apache Solr (todos construidos sobre la biblioteca Apache Lucene) mantiene un *índice invertido* de las palabras a los documentos que las contienen, con análisis de texto (lematización, sinónimos, tolerancia a erratas) y ordenación por relevancia. Responde a "ramen vegano cerca de mí" en milisegundos. Casi nunca debe ser un sistema de registro: los índices se actualizan casi en tiempo real en lugar de forma transaccional, y el diseño habitual alimenta el motor de búsqueda desde la base de datos propietaria y reconstruye el índice a partir de ella cuando hace falta.

Una base de datos vectorial almacena *embeddings*, listas de números que representan el significado de textos, imágenes o productos, y encuentra los elementos más cercanos a un vector de consulta. La búsqueda exacta compara la consulta con cada vector, así que estos almacenes usan índices de búsqueda aproximada de vecinos más cercanos, como los grafos HNSW, que sacrifican un poco de exhaustividad (recall) a cambio de velocidad. Impulsan la búsqueda semántica, las recomendaciones y la recuperación para modelos de lenguaje

grandes, que se tratan en generación aumentada por recuperación. Existen productos dedicados (Milvus, Qdrant, Pinecone, Weaviate), pero para millones de vectores, y no miles de millones, suele bastar con una extensión de una base de datos existente, como pgvector.

SQL distribuido

Las bases de datos de SQL distribuido, a veces llamadas *NewSQL* (un término acuñado por el analista Matthew Aslett en 2011), mantienen el modelo relacional, SQL y las transacciones ACID, pero particionan y replican los datos automáticamente entre muchos nodos, a menudo entre regiones. Google Spanner, descrito en un artículo de OSDI de 2012, usa relojes estrechamente sincronizados (su API TrueTime, respaldada por GPS y relojes atómicos) para ofrecer transacciones globalmente consistentes. CockroachDB, YugabyteDB y TiDB ofrecen modelos similares sobre hardware corriente; CockroachDB usa por defecto el aislamiento serializable.

El precio lo ponen la física y la operación. Una transacción que deben acordar réplicas en varias regiones espera un viaje de ida y vuelta de consenso, así que las escrituras de una sola fila tardan de milisegundos a decenas o cientos de milisegundos, en lugar de muy por debajo de un milisegundo. La planificación de consultas entre particiones es más sutil, los costos son más altos y menos ingenieros conocen sus modos de fallo. El SQL distribuido es la herramienta adecuada cuando de verdad necesitas transacciones relacionales más allá de la capacidad de un servidor o entre regiones (un libro mayor de pagos global, un banco multirregión), y una herramienta cara cuando bastaría con un único primario PostgreSQL con réplicas.

Comparación de las familias de bases de datos

Familia	Modelo de datos y consultas	Ventajas	Puntos débiles	Ejemplos
Relacional	Tablas, claves, SQL con joins y agregaciones	Transacciones y restricciones, consultas ad hoc flexibles, herramientas maduras, ampliamente conocida	Límite del escalado vertical para las escrituras en un nodo; el esquema rígido necesita migraciones	PostgreSQL, MySQL, SQL Server, Oracle
Documento	Documentos de tipo JSON, consultas sobre campos y valores anidados	Encaje natural para agregados que se leen como un todo; estructura flexible	Joins y transacciones entre documentos más débiles o costosos; esquema aplicado en el código	MongoDB, Couchbase, Firestore
Clave-valor	Valor opaco por clave; get, put, delete	Muy rápido, sencillo, fácil de particionar	Sin consultas entre valores; la carga del modelado recae en la aplicación	Redis, Valkey, DynamoDB, etcd
Col. anchas	Filas agrupadas por clave de partición, ordenadas por clave de agrupación	Rendimiento de escritura enorme, escalado horizontal lineal, varios centros de datos	Tablas construidas por consulta; joins y consultas ad hoc limitados	Apache Cassandra, ScyllaDB, HBase, Bigtable
Grafo	Nodos y aristas con propiedades; consultas de recorrido	Consultas de relaciones de varios saltos	Más difícil de particionar; ecosistema más pequeño	Neo4j, Amazon Neptune
Series temp.	Puntos con marca de tiempo, etiquetas; consultas por rango y de agregación	Compresión, conservación, reducción de resolución	Malo para actualizaciones generales y relaciones	InfluxDB, TimescaleDB, Prometheus

Familia	Modelo de datos y consultas	Ventajas	Puntos débiles	Ejemplos
Motor de búsqueda	Índice invertido sobre documentos; consultas de texto ordenadas por relevancia	Búsqueda de texto completo, coincidencia aproximada, facetas	No es un sistema de registro; actualización casi en tiempo real	Elasticsearch, OpenSearch, Apache Solr
Vectorial	Embeddings; búsqueda aproximada de vecinos más cercanos	Similitud semántica para recuperación y recomendaciones	Resultados aproximados; costo de construir el índice; herramientas jóvenes	pgvector, Milvus, Qdrant, Pinecone
SQL distribuido	Modelo relacional, SQL, particionado y replicado	Transacciones y SQL con escalado horizontal y supervivencia multirregión	Mayor latencia de escritura, costo y complejidad operativa	Google Spanner, CockroachDB, YugabyteDB, TiDB

Elegir: los criterios que importan

Elige por conjunto de datos, no por sistema. Para cada conjunto de datos importante, reúne pruebas sobre seis criterios y deja constancia de la decisión en un ADR.

Forma de los datos. ¿Son tabulares por naturaleza, con muchas relaciones, un agregado autocontenido, una red, una serie temporal, texto libre o vectores? **Patrones de acceso.** ¿Qué consultas se ejecutan, con qué frecuencia, con qué objetivos de latencia y cuántas son ad hoc? ¿Predominan las lecturas o las escrituras? ¿Siempre por clave o por muchos atributos? **Necesidades de consistencia.** ¿Necesitan los datos transacciones de varios elementos y restricciones (saldos, existencias, reservas), o unos segundos de desactualización son inofensivos (me gusta, recomendaciones)? Consulta los modelos de consistencia. **Escala.** Tamaño de los datos, escrituras por segundo y crecimiento, actuales y previstos, medidos y no adivinados; un almacén que escala a petabytes no es ninguna ventaja para 50 gigabytes. **Operación.** ¿Quién lo va a operar: copias de seguridad, actualizaciones, failover, monitorización, parches de seguridad? ¿Hay un servicio gestionado en tu nube y cuánto cuesta con tus volúmenes? **Conocimientos del equipo.** ¿Qué conoce ya el equipo lo bastante bien como para depurarlo a las 3 de la madrugada? Una tecnología desconocida trae modos de fallo desconocidos.

Dos hábitos mejoran la elección. Anota las cinco o diez consultas más importantes con sus latencias objetivo *antes* de elegir, y prueba el candidato con volúmenes de datos realistas; muchas decepciones vienen de un almacén que se evaluó en un portátil con mil filas. Y separa el *sistema de registro* de los *almacenes derivados*: el sistema de registro necesita las garantías más sólidas y la tecnología más conservadora, mientras que los almacenes derivados (índices de búsqueda, cachés, modelos de lectura) pueden ser especializados precisamente porque pueden reconstruirse.

GUÍA DE DECISIÓN

Guía de decisión: ¿qué tipo de almacén para estos datos?

Hazte estas preguntas para cada conjunto de datos (no una sola vez para todo el sistema), en este orden. Detente en el primer *sí* que encaje; si llegas al final, se mantiene la opción relacional por defecto.

Q1 **¿Son estos datos un sistema de registro que necesita transacciones de varias filas, restricciones o consultas ad hoc (pedidos, pagos, cuentas, reservas)?**

- **Sí** Usa una base de datos relacional. Busca más allá solo si puedes demostrar que no puede cumplir un requisito de escala o de latencia medido.
- **No** Continúa: los datos pueden ser derivados, especializados o de un volumen extremadamente alto.

Q2 **¿Es una copia derivada de un sistema de registro para atender una consulta especial (búsqueda de texto completo, recomendaciones, búsqueda por similitud)?**

- **Sí** Usa el almacén especializado para esa consulta (motor de búsqueda, índice de grafos o vectorial), alimentado desde el sistema de registro y reconstruible a partir de él.
- **No** Continúa.

Q3 **¿Se lee y se escribe como un único elemento por clave, sin consultas entre elementos (sesiones, carritos, últimas posiciones, feature flags)?**

- **Sí** Encaja un almacén clave-valor, a menudo en memoria y con caducidad. Decide qué ocurre si se pierde.
- **No** Continúa.

Q4 **¿Es un flujo, casi solo de adición, de mediciones con marca de tiempo que se consultan por rango de tiempo (métricas, lecturas de sensores, historial de ubicaciones)?**

- **Sí** Una base de datos de series temporales o, con volúmenes moderados, tablas particionadas por tiempo en una base de datos relacional.
- **No** Continúa.

Q5 **¿Superarán el volumen de escritura o el tamaño de los datos lo que puede manejar un servidor bien ajustado, con acceso siempre por una clave conocida?**

- **Sí** Un almacén de columnas anchas o clave-valor diseñado para el particionamiento (Cassandra, DynamoDB), con tablas diseñadas en torno a las consultas.
- **No** Continúa.

Q6 **¿Necesitas transacciones relacionales y SQL entre regiones o más allá de la capacidad de un servidor?**

Sí Considera el SQL distribuido (Spanner, CockroachDB, YugabyteDB), aceptando una mayor latencia de escritura y un mayor costo de operación.

No Quédate con la opción relacional por defecto y revísala cuando cambie un requisito medido.

SI ESTO ES NUEVO PARA TI

Paso a paso: la única copia o una copia

La idea que requiere más atención aquí es la diferencia entre un sistema de registro y un almacén derivado, y por qué decide cuán atrevida puede ser la elección de una base de datos. Sigue una edición de una carta a través de los almacenes de la plataforma de reparto de comida.

- Un restaurante cambia la descripción de un plato, y la edición se almacena en la base de datos PostgreSQL de cartas. Esa base de datos es el sistema de registro: siempre que otra copia de una carta no coincide con ella, prevalece su versión.
- La captura de datos de cambios recoge el cambio confirmado y se lo pasa al motor de búsqueda, que actualiza su índice unos segundos después. Durante esos segundos, una búsqueda todavía puede mostrar la descripción antigua, un retardo que el equipo ha aceptado.
- Ahora supón que el clúster de búsqueda falla y su índice se pierde. Los clientes no pueden buscar durante un rato, pero no se ha perdido ninguna información: el equipo reconstruye el índice leyendo de nuevo todas las cartas desde PostgreSQL.
- Imagina en cambio que las ediciones de los restaurantes se guardaran solo en el motor de búsqueda. Su índice se actualiza casi en tiempo real en lugar de dentro de transacciones, y puede que proteja cada escritura con menos cuidado que la base de datos, así que el mismo fallo podría perder ediciones sin nada a partir de lo cual reconstruirlas.
- Por eso la plataforma puede operar un almacén especializado para la búsqueda, elegido para esa única tarea, mientras que los pedidos y las cartas se quedan en PostgreSQL, la base de datos que el equipo ya opera bien.
- **La lección:** antes de elegir un almacén, pregúntate si contendrá la única copia de algo. Si es así, elige la tecnología más conservadora que puedas; si contiene una copia que puede reconstruirse, tienes libertad para especializarte.

La persistencia políglota y su costo

La persistencia políglota consiste en usar distintas tecnologías de almacenamiento para distintos datos dentro de un mismo sistema, elegidas según cómo se usan los datos. Martin Fowler popularizó el término en 2011, y atribuyó su primer uso a Scott Leberknight. Se deriva de forma natural de la base de datos por servicio: cada propietario puede elegir la herramienta adecuada.

El costo es fácil de subestimar. Cada tipo adicional de base de datos necesita sus propios conocimientos especializados, procedimientos de copia de seguridad y restauración, monitorización, planificación de capacidad, refuerzo de la seguridad, ciclo de actualizaciones y conocimientos para las guardias. Ahora los datos se mueven entre almacenes, así que necesitas pipelines (CDC, eventos) y debes razonar sobre el retardo y los fallos de cada uno. La consistencia entre almacenes es, como mucho, eventual. Para un equipo pequeño, cinco bases de datos pueden consumir más esfuerzo que el producto. Una regla razonable es justificar cada nuevo almacén con un requisito que los existentes demostrablemente no pueden cumplir, y preferir servicios gestionados para los especializados.

DE DÓNDE VIENE

Choose Boring Technology

En un ensayo de 2015, Dan McKinley, basándose en sus años como ingeniero en Etsy, sostuvo que cada empresa solo dispone de unos pocos *tokens de innovación*, unos tres, que gastar en tecnologías desconocidas. Las herramientas maduras y “aburridas” tienen modos de fallo conocidos; las nuevas traen incógnitas desconocidas. Cada nueva tecnología añade además un costo operativo y cognitivo permanente a toda la organización. Su consejo: pregúntate primero cómo resolverías el problema con lo que ya operas, y trata la incorporación de una tecnología como una decisión de la organización, no de un solo equipo. El ensayo se cita mucho en la elección de bases de datos, donde el costo de operar algo desconocido dura años.

EJEMPLO PRÁCTICO

Ejemplo práctico: los almacenes de la plataforma de reparto de comida

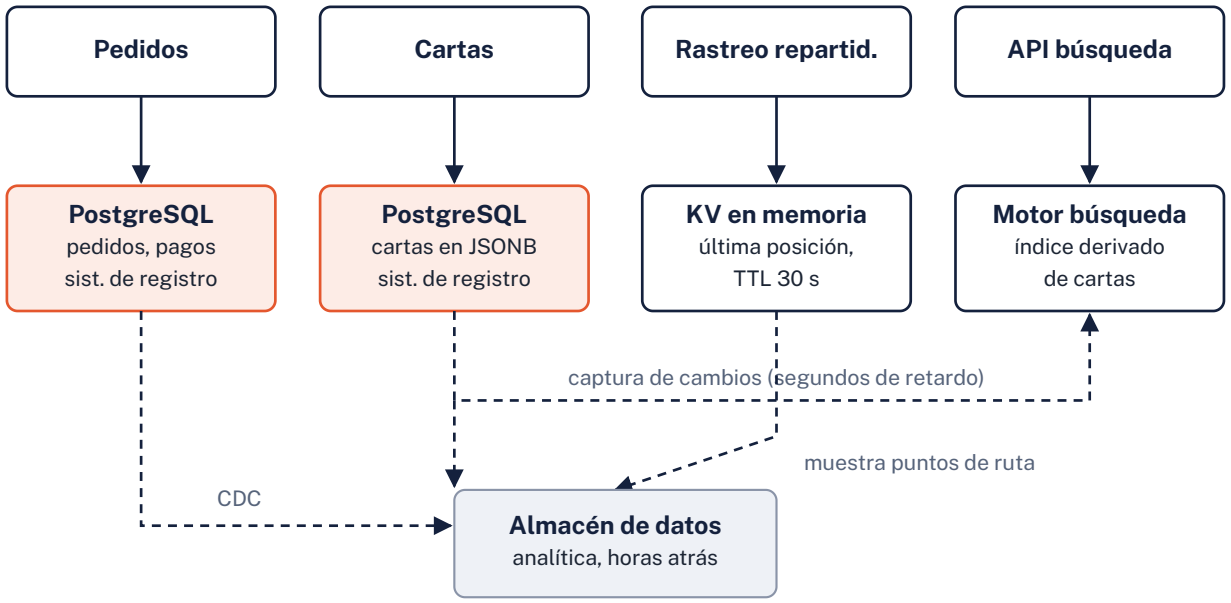
La plataforma opera en 40 ciudades. En el pico de la noche recibe unos 300 pedidos por minuto, tiene 20 000 repartidores conectados, sirve 50 000 cartas de restaurantes y recibe unas 2000 búsquedas por segundo. El equipo de 35 ingenieros ya opera bien PostgreSQL.

Pedidos y pagos	Sistema de registro; transacciones de varias filas (pedido, líneas, autorización del pago); reglas estrictas; consultas ad hoc del equipo de soporte. Unos 5 pedidos nuevos por segundo en el pico (unas pocas decenas de escrituras de filas). **PostgreSQL**, con una réplica para el failover. No hay motivos para nada más.
Cartas	Predominan las lecturas, jerárquicas (secciones, platos, grupos de opciones), editadas por los restaurantes unos pocos miles de veces al día. Los precios deben ser exactamente los que registra el pedido. **PostgreSQL** con un núcleo relacional (restaurante, plato, precio) y `jsonb` para las opciones variables. Un almacén documental encajaría con la forma, pero añadiría una segunda tecnología sin cumplir ningún requisito que PostgreSQL no cumpla.
Ubicaciones de los repartidores	20 000 repartidores que envían cada uno una posición cada 4 segundos suponen 5000 escrituras por segundo, de las que para el despacho solo importa la última de cada repartidor, y solo durante unos 30 segundos. **Almacén clave-valor en memoria** (Redis o Valkey) con consultas geoespaciales y una caducidad de 30 segundos. Perderlo en un reinicio cuesta un ciclo de actualización. Una muestra de puntos (por ejemplo, uno cada 30 segundos) va al almacén de datos para la analítica.
Búsqueda de restaurantes y platos	Texto completo, tolerante a erratas, filtrado por ubicación, horario de apertura y etiquetas dietéticas, ordenado por relevancia. **Motor de búsqueda** (OpenSearch), alimentado desde la base de datos de cartas mediante captura de datos de cambios y reconstruible a partir de ella. Acepta unos segundos de retardo tras una edición de la carta.
Analítica	Ingresos semanales, tiempos de entrega por ciudad, utilización de los repartidores: grandes recorridos y agregaciones sobre meses de historial. **Almacén de datos en la nube** alimentado por CDC y eventos, con horas de antigüedad por diseño. Consulta analítica y streaming.
Descartados	Una base de datos de grafos para las recomendaciones (todavía no hace falta; se revisará con pruebas), una base de datos de series temporales separada para las ubicaciones (la muestra del almacén de datos responde a las preguntas que se plantean) y el SQL distribuido (los pedidos de toda la plataforma, unos 5 por segundo en el pico, caben holgadamente en un solo primario; no hay ningún requisito de transacciones multirregión).

El resultado son dos sistemas de registro sobre una tecnología bien conocida más tres almacenes derivados especializados. Cada tecnología adicional se justifica con un requisito que la opción por defecto no puede cumplir: 5000 escrituras efímeras por segundo y la búsqueda de texto ordenada por relevancia.

Anotar primero las consultas y las cifras hizo que las decisiones fueran casi obvias, y dio al equipo un registro de los supuestos que revisar cuando cambien los volúmenes.

El panorama de almacenes resultante



Dos sistemas de registro; todo lo demás es derivado y puede reconstruirse

Los pedidos y las cartas tienen cada uno un único sistema de registro relacional; el índice de búsqueda, el almacén de datos y el almacén de ubicaciones son derivados o desechables.

Errores comunes

Elecciones para engordar el currículum: elegir un almacén porque es nuevo o porque lo usa una gran empresa, sin el requisito que lo justificaba allí. **Elegir para una escala imaginaria**: diseñar para mil millones de usuarios con un producto que tiene mil, pagando el impuesto de la complejidad desde el primer día. **Usar un almacén derivado como sistema de registro**: un índice de búsqueda o una caché que se convierte en la única copia de los datos, con una durabilidad más débil de lo que nadie suponía. **Ignorar el lado de las consultas**: elegir un almacén clave-valor o de columnas anchas y descubrir después que el negocio necesita informes ad hoc y filtros que no puede atender. **Subestimar la operación**: olvidar que alguien tiene que restaurarlo desde una copia de seguridad, actualizarlo sin tiempo de inactividad y

diagnosticarlo bajo carga. Por último, **tratar la elección como una verdad permanente**: deja constancia de los supuestos (volúmenes, consultas, conocimientos del equipo) para poder revisar la decisión cuando cambien.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **¿Por qué un motor de búsqueda no debe ser, por lo general, el sistema de registro?**

Su índice se actualiza casi en tiempo real en lugar de forma transaccional y sus garantías de durabilidad son más débiles; debe alimentarse desde la base de datos propietaria y reconstruirse a partir de ella cuando haga falta.

▼ **Da dos señales de que un almacén de columnas anchas puede ser una buena opción.**

Un volumen de escritura muy alto, más allá de un servidor, y un acceso que siempre parte de una clave de partición conocida, de modo que las tablas pueden diseñarse por consulta.

▼ **¿Qué representa un 'token de innovación' en el ensayo de McKinley?**

Una de las pocas oportunidades que tiene una empresa de adoptar una tecnología desconocida, cada una de las cuales trae modos de fallo desconocidos y un costo operativo duradero.

▼ **¿Cuándo compensa el costo del SQL distribuido?**

Cuando necesitas transacciones relacionales y SQL más allá de la capacidad de un servidor o entre regiones, como en un libro mayor global; no cuando un único primario con réplicas cumple los requisitos.

▼ **Inténtalo: Un equipo de reparto de comida de 20 personas conoce bien PostgreSQL. Necesita almacenamiento para pedidos y pagos, posiciones en vivo de los repartidores actualizadas cada pocos segundos y búsqueda de texto de restaurantes. Elige un almacén para cada caso y di qué limita cuántos añades.**

Los pedidos y los pagos se quedan en PostgreSQL: un sistema de registro que necesita transacciones, restricciones y consultas ad hoc. Las posiciones de los repartidores se adaptan a un almacén clave-valor con caducidad corta, ya que solo importa el último valor y cambia rápido. La búsqueda de restaurantes se adapta a un motor de búsqueda, o al principio a la búsqueda de texto de PostgreSQL, como un almacén derivado que se puede reconstruir. Cada tecnología adicional añade un costo operativo duradero, así que anota las consultas y los volúmenes y añade una solo con evidencias.

FUENTES

- Dan McKinley: Choose Boring Technology (2015): https://mcfunley.com/choose-boring-technology
- Martin Fowler: PolyglotPersistence (2011): https://martinfowler.com/bliki/PolyglotPersistence.html
- Google Research: Spanner: Google's Globally-Distributed Database (OSDI 2012): https://research.google/pubs/spanner-googles-globally-distributed-database-2/
- Google Cloud: Spanner TrueTime and external consistency: https://cloud.google.com/spanner/docs/true-time-external-consistency
- DeCandia et al.: Dynamo: Amazon's Highly Available Key-value Store (SOSP 2007): https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
- RFC 9562: Universally Unique IDentifiers (UUIDs): https://www.rfc-editor.org/rfc/rfc9562.txt
- ULID specification: https://github.com/ulid/spec
- PostgreSQL documentation: UUID functions: https://www.postgresql.org/docs/current/functions-uuid.html

SECCIÓN 7.3 DE 7.8 · AVANZADO · 28 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Transacciones, ACID y niveles de aislamiento

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.5 Almacenar datos

Términos anteriores que utiliza: SQL (Structured Query Language), Clave foránea, Índice, Linealizabilidad

LA IDEA EN BREVE

Imagina a dos cajeros que comparten un único libro de cuentas en papel para la cuenta de un cliente. Cada uno consulta el saldo, calcula el nuevo en un bloc de notas y lo anota. Si lo hacen en el mismo momento, se pierde el cambio de uno de los cajeros. Una transacción de base de datos es una norma sobre cómo se turnan los cajeros: si pueden mirar las anotaciones a medio escribir del otro, si la página puede cambiar mientras la leen y si uno debe esperar al otro. Normas más estrictas significan menos errores pero más espera, y los nombres de estas normas significan cosas ligeramente distintas en bancos distintos.

LO QUE NECESITAS SABER

- ACID: la atomicidad significa todo o nada ante un fallo; la consistencia significa que se cumplen las invariantes de la aplicación; el aislamiento limita la interferencia entre transacciones concurrentes; la durabilidad significa que los datos confirmados sobreviven a las caídas, normalmente gracias a un registro de escritura anticipada y a réplicas.
- El estándar SQL define read uncommitted, read committed, repeatable read y serializable según las lecturas sucias, las lecturas no repetibles y los fantasmas que prohíben; Berenson et al. (1995) demostraron que estas definiciones son ambiguas e incompletas.
- Las actualizaciones perdidas y el write skew surgen cuando las transacciones leen datos, deciden en la aplicación y vuelven a escribir; el aislamiento de instantánea evita las actualizaciones perdidas, pero permite el write skew.
- MVCC mantiene varias versiones de cada fila para que lectores y escritores no se bloqueen entre sí; las transacciones de larga duración inflan el almacenamiento.
- Los valores por defecto y los nombres difieren: PostgreSQL y Oracle usan por defecto read committed, MySQL InnoDB repeatable read y CockroachDB serializable; el serializable de Oracle es aislamiento de instantánea.

- Protege las rutas de lectura-modificación-escritura con restricciones, actualizaciones condicionales atómicas, SELECT ... FOR UPDATE (pesimista) o columnas de versión (optimista), o usa aislamiento serializable con reintentos de la transacción completa.

Lo que promete una transacción

Una transacción agrupa varias lecturas y escrituras en una unidad que o bien surte efecto por completo (se confirma) o bien no surte ningún efecto (se aborta y se revierte). El Capítulo 0 presentó la idea con una transferencia bancaria. Esta sección trata de la parte que la mayoría de los desarrolladores entiende mal: qué ocurre cuando se ejecutan muchas transacciones a la vez, qué garantías dan realmente las bases de datos y cómo elegir. Los ejemplos usan SQL, pero el razonamiento se aplica a cualquier almacén que ofrezca transacciones.

Las transacciones importan para la arquitectura porque deciden dónde se pueden hacer cumplir las reglas de negocio de forma barata. Dentro de una base de datos, una transacción puede mantener cuadrado un libro de cuentas o impedir que una tabla se reserve dos veces con unas pocas líneas de código. Entre servicios no existe esa herramienta, y por eso la propiedad de los datos y las sagas son tan importantes. Saber exactamente qué garantiza una sola base de datos te dice qué reglas deben quedar dentro de los límites de un único propietario.

ACID en detalle

El término ACID lo acuñaron Theo Härder y Andreas Reuter en 1983, a partir del trabajo anterior de Jim Gray. Cada letra es más restringida de lo que parece. La **atomicidad** tiene que ver con los fallos, no con la concurrencia: si algo sale mal a mitad de camino (una caída, la violación de una restricción, un interbloqueo, la aplicación que decide abortar), se deshace cada cambio hecho hasta ese momento, de modo que la aplicación puede reintentar con seguridad. *Abortabilidad* sería un nombre más preciso.

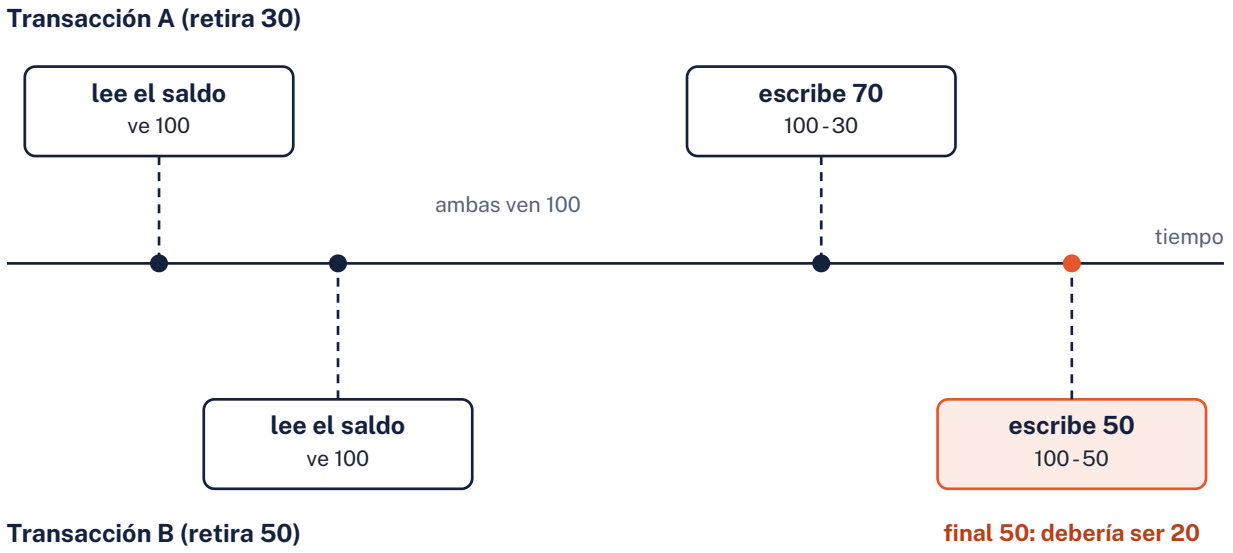
La **consistencia** significa que una transacción lleva la base de datos de un estado válido a otro, donde "válido" significa que se cumplen tus invariantes: los cargos igualan a los abonos, el stock nunca es negativo, todo pedido tiene un cliente. La base de datos puede hacer cumplir algunas invariantes por sí misma (claves, claves foráneas, restricciones `CHECK`, índices únicos), pero la mayoría son reglas de negocio que solo mantienen las transacciones de la aplicación. Por tanto, la consistencia es una propiedad del uso que hace la aplicación de la atomicidad y el aislamiento, no algo que la base de datos proporcione por sí sola. Tampoco tiene relación con la "C" de CAP, que significa linealizabilidad (Capítulo 6).

El **aislamiento** significa que las transacciones que se ejecutan de forma concurrente no interfieren entre sí. En su forma más estricta, la serializabilidad, el resultado de ejecutar transacciones de forma concurrente es el mismo que si se hubieran ejecutado de una en una en algún orden. Los niveles más débiles permiten cierta interferencia a cambio de rendimiento, y el resto de esta sección trata precisamente de qué interferencia. La **durabilidad** significa que, una vez confirmado un commit, los datos sobreviven a las caídas. Las bases de datos de un solo nodo lo consiguen con un registro de escritura anticipada (write-ahead log) que se vuelca al disco antes de que el commit retorne; las bases de datos replicadas añaden copias en otras máquinas. La durabilidad nunca es absoluta: un disco puede mentir sobre un volcado, y la replicación asíncrona puede perder las últimas escrituras confirmadas cuando falla el líder (consulta replicación).

Lo que sale mal sin aislamiento

Los bugs de concurrencia son difíciles de encontrar porque dependen del momento exacto: pasan todas las pruebas en una tranquila máquina de desarrollo y aparecen en el pico de la hora de la cena. La forma más clara de entender los niveles de aislamiento es a través de las anomalías concretas (también llamadas fenómenos) que permiten o evitan. La actualización perdida que se muestra a continuación es la más común en el código de negocio.

Una actualización perdida, paso a paso



Cada transacción es correcta por sí sola; intercaladas, la segunda escritura borra en silencio la primera retirada.

Las anomalías, de la más leve a la más sutil

Anomalía	Qué ocurre	Ejemplo de reparto de comida o de banca
Escritura sucia	Una transacción sobrescribe datos que otra transacción ha escrito pero aún no ha confirmado.	Dos transacciones que asignan un repartidor se intercalan, de modo que la fila del pedido nombra a un repartidor y la fila del repartidor a otro pedido. Todo nivel de aislamiento real lo evita.
Lectura sucia	Una transacción lee datos escritos por otra transacción que no se ha confirmado (y que puede revertirse).	Un informe cuenta un pago de 500 que después se revierte.
Lectura no repetible (difusa)	Una transacción lee la misma fila dos veces y obtiene valores distintos porque otra transacción confirmó un cambio entre ambas lecturas.	El proceso de pago lee un precio de plato de 9.50, una edición del restaurante confirma 11.00 y la segunda lectura del proceso de pago ve 11.00.
Sesgo de lectura (read skew)	Una transacción ve partes de la base de datos en momentos distintos, de modo que los valores relacionados no cuadran.	Un cliente con 500 repartidos en dos cuentas ve 450 en total porque una transferencia de 50 se confirmó entre la lectura de los dos saldos.
Lectura fantasma	Una transacción vuelve a ejecutar una consulta con una condición y encuentra filas añadidas o eliminadas por otra transacción confirmada.	Contar dos veces los repartos abiertos de un repartidor devuelve 2 y después 3.
Actualización perdida	Dos transacciones leen un valor, cada una calcula uno nuevo y lo escribe, y una escritura sobrescribe en silencio a la otra.	Dos retiradas de la misma cuenta (diagrama anterior).
Write skew (sesgo de escritura)	Dos transacciones leen un conjunto de datos que se solapa, toman decisiones basadas en él y actualizan filas distintas, de modo que juntas rompen una regla que cada una comprobó.	Dos médicos de guardia ven cada uno que el otro sigue de guardia y ambos salen de guardia (ejemplo práctico más abajo).

Los niveles de aislamiento del estándar SQL

El estándar SQL (desde SQL-92 en adelante) define cuatro niveles de aislamiento según cuáles de tres fenómenos prohíben: lecturas sucias, lecturas no repetibles y fantasmas. El estándar establece serializable como valor por defecto, pero casi ningún producto convencional lo sigue; la mayoría usa por defecto read committed o repeatable read por rendimiento.

Los niveles del estándar SQL y los fenómenos que descartan

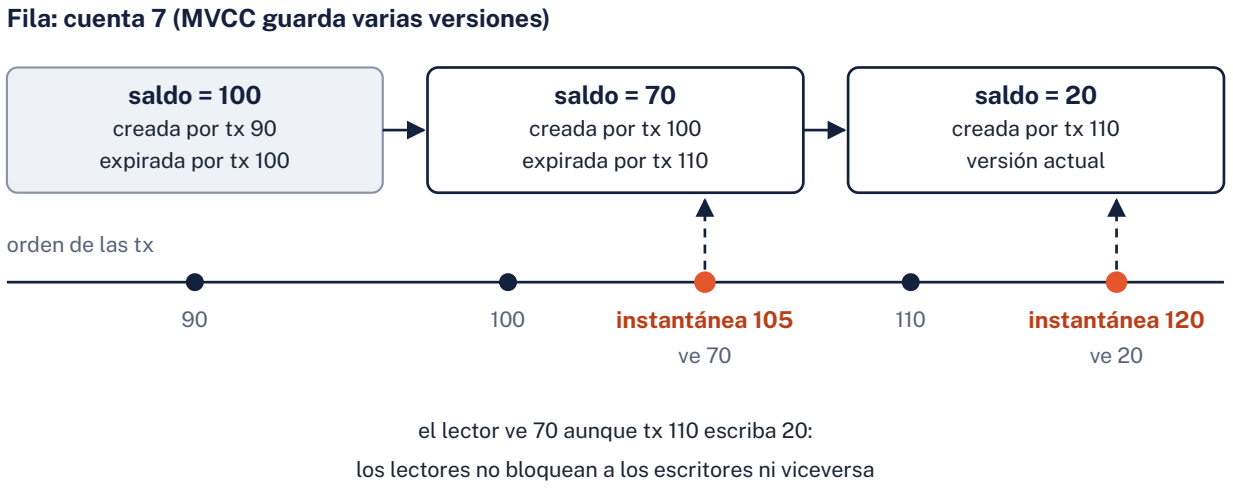
Nivel de aislamiento	Lectura sucia	Lectura no repetible	Lectura fantasma
Read uncommitted	posible	posible	posible
Read committed	evitada	posible	posible
Repeatable read	evitada	evitada	posible
Serializable	evitada	evitada	evitada, y el resultado debe ser igual a algún orden en serie

En 1995, Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil y Patrick O'Neil publicaron *A Critique of ANSI SQL Isolation Levels*. Demostraron que las definiciones en inglés del estándar eran ambiguas, que prohibir los tres fenómenos no proporciona por sí solo serializabilidad y que los niveles del estándar no describían los niveles de aislamiento que usaban los productos reales. Añadieron fenómenos que al estándar se le escapaban, en particular la escritura sucia, la actualización perdida, el read skew y el write skew, y definieron el aislamiento de instantánea (snapshot isolation): cada transacción lee de una instantánea coherente de los datos confirmados en el momento de su inicio, y una transacción no puede confirmar una escritura en una fila que una transacción concurrente ya ha modificado y confirmado (“gana el primero que confirma”). El aislamiento de instantánea evita las lecturas sucias, las lecturas no repetibles, el read skew y las actualizaciones perdidas, así como los fantasmas que afectan a las lecturas, pero permite el write skew, así que no es serializable. Encaja con dificultad entre los niveles del estándar, y por eso los productos lo etiquetan de forma distinta.

MVCC: cómo funcionan las instantáneas

La mayoría de las bases de datos modernas implementan las lecturas de instantánea con control de concurrencia multiversión (MVCC). En lugar de sobrescribir una fila en su sitio, una escritura crea una nueva versión etiquetada con el identificador de la transacción que escribe, y la versión antigua se conserva hasta que ninguna transacción en curso pueda necesitarla todavía. A cada lector se le da una instantánea, que es básicamente una lista de las transacciones que se habían confirmado cuando empezó, y solo ve las versiones que crearon esas transacciones. La gran ventaja es que los lectores nunca esperan a los escritores y los escritores nunca esperan a los lectores; solo entran en conflicto dos escritores sobre la misma fila.

Versiones de una fila con MVCC



Cada actualización crea una nueva versión de la fila; una transacción ve la versión más reciente que se confirmó antes de su instantánea.

El costo es el mantenimiento. Hay que limpiar las versiones antiguas: PostgreSQL las guarda en la tabla y las elimina con *vacuum*, mientras que Oracle y MySQL InnoDB guardan las imágenes previas en registros de deshacer (undo logs). Una transacción que se deja abierta durante horas retiene versiones antiguas, de modo que las tablas y los registros de deshacer se inflan y el rendimiento se degrada. Por eso las transacciones de larga duración son una mala señal operativa en cualquier base de datos MVCC, y un motivo para mantener las consultas analíticas fuera de la base de datos operativa. En **read committed** con MVCC, cada *sentencia* obtiene una instantánea nueva; en el **aislamiento de instantánea**, toda la *transacción* usa una sola.

Los valores por defecto reales, y por qué engañan los nombres

Los nombres de los niveles de aislamiento no son portables. La misma palabra puede significar garantías distintas en productos distintos, y los valores por defecto difieren. El proyecto Hermitage de Martin Kleppmann probó el comportamiento real de muchas bases de datos frente a las anomalías anteriores; la documentación de los fabricantes confirma los valores por defecto. Toma esta tabla como punto de partida y consulta la documentación de la versión que uses.

Mismos nombres, comportamiento distinto: qué hacen las bases de datos reales (resumido a partir de la documentación de los fabricantes y de las pruebas de Hermitage)

Base de datos	Nivel por defecto	Qué son realmente los niveles	Notas
PostgreSQL	Read committed	Read uncommitted se comporta como read committed. Repeatable read es aislamiento de instantánea. Serializable es aislamiento de instantánea serializable (SSI).	Repeatable read aborta una transacción que intenta actualizar una fila modificada desde su instantánea (*could not serialize access due to concurrent update*), así que se evitan las actualizaciones perdidas; el write skew, no.
MySQL (InnoDB)	Repeatable read	Los SELECT simples leen una instantánea tomada en la primera lectura; las lecturas con bloqueo, UPDATE y DELETE actúan sobre las filas confirmadas más recientes con bloqueos de registro y de hueco (record locks y gap locks). Serializable convierte los SELECT simples en lecturas con bloqueo cuando autocommit está desactivado.	En repeatable read, una lectura-modificación-escritura escrita como SELECT y después UPDATE todavía puede perder una actualización o producir write skew, salvo que el SELECT use FOR UPDATE.
Oracle Database	Read committed	Ofrece read committed y serializable; su serializable es aislamiento de instantánea.	Evita las actualizaciones perdidas en serializable, pero permite el write skew.

Base de datos	Nivel por defecto	Qué son realmente los niveles	Notas
Microsoft SQL Server	Read committed (con bloqueos) en SQL Server; Azure SQL Database activa read committed snapshot por defecto	Read committed con bloqueos, read committed snapshot, repeatable read (con bloqueos), snapshot (aislamiento de instantánea) y serializable (con bloqueos).	En SQL Server, el aislamiento de instantánea debe habilitarse por base de datos (ALLOW_SNAPSHOT_ISOLATION) antes de usarlo.
CockroachDB	Serializable	Serializable por defecto; read committed también está disponible.	Los clientes deben reintentar las transacciones que fallan con un error de serialización.

ATENCIÓN

Dos trampas ocultas en los nombres

Primero, “repeatable read” en PostgreSQL y en MySQL son cosas distintas: el de PostgreSQL es un verdadero aislamiento de instantánea que aborta las actualizaciones en conflicto, mientras que el de MySQL mezcla lecturas de instantánea con escrituras con bloqueo sobre los datos más recientes y puede perder actualizaciones escritas como lectura y después escritura. Segundo, “serializable” en Oracle es aislamiento de instantánea y permite el write skew. Un código que es correcto en una base de datos puede ser incorrecto en otra con un ajuste de nombre idéntico.

Cómo lograr la serializabilidad

Las bases de datos que ofrecen verdadera serializabilidad usan una de tres técnicas. La **ejecución realmente en serie** ejecuta las transacciones de una en una en un único hilo, lo cual es rápido si cada transacción es corta y los datos están en memoria (VoltDB, y los scripts y transacciones de Redis, funcionan así). El **bloqueo en dos fases (2PL)**, que usan los niveles serializables de SQL Server y de MySQL, hace que los lectores tomen bloqueos compartidos y los escritores bloqueos exclusivos, incluidos bloqueos sobre rangos para impedir los fantasmas, y los mantiene hasta el final de la transacción; es correcto, pero lectores y escritores se bloquean entre sí, el throughput cae cuando hay contención y los interbloqueos son frecuentes. El **aislamiento de instantánea serializable (SSI)**, descrito por Michael Cahill, Uwe Röhm y Alan Fekete en 2008 y usado en PostgreSQL desde la versión 9.1 (CockroachDB

usa una técnica optimista relacionada), ejecuta las transacciones sobre instantáneas sin bloquear, registra las dependencias de lectura-escritura entre ellas y aborta una transacción cuando aparece un patrón peligroso.

Sea cual sea la técnica, las transacciones serializables pueden fallar con un error de serialización (SQLSTATE `40001` en PostgreSQL y CockroachDB) o con un interbloqueo, y la aplicación debe reintentar la transacción completa. Este es el costo oculto del nivel más fuerte: cada ruta de código transaccional necesita un bucle de reintentos que sea seguro repetir, lo que significa que no puede haber efectos secundarios, como enviar correos electrónicos o llamar a proveedores de pago, dentro de la transacción.

Agrupación de conexiones (connection pooling)

Cada transacción ocupa una conexión a la base de datos, y las conexiones son limitadas (Sección 9.1). Por eso, las aplicaciones mantienen un **pool** y toman prestada una conexión para cada unidad de trabajo. Más grande no es mejor: las notas de HikariCP citan una demostración de Oracle en la que reducir 2048 conexiones a 96 llevó los tiempos de respuesta de unos 100 ms a unos 2 ms, y ofrecen (núcleos × 2) + discos efectivos como punto de partida para probar. El total es el tamaño del pool por el número de instancias, así que escalar horizontalmente puede agotar la base de datos; un agrupador central como PgBouncer ayuda. Su modo de transacción presta una conexión para una transacción cada vez, pero entonces fallan funciones de sesión como `SET` y los bloqueos consultivos de sesión. Nunca mantengas una conexión mientras llamas a otro servicio, o un proveedor de pagos lento agotará el pool.

Control de concurrencia pesimista y optimista

En la práctica, la mayoría de las aplicaciones se ejecutan en read committed o repeatable read y protegen explícitamente rutas concretas de lectura-modificación-escritura. Hay tres herramientas. La mejor, cuando es posible, es una **actualización atómica** que deja a la base de datos hacer la aritmética y la comprobación en una sola sentencia, como `UPDATE accounts SET balance = balance - 30 WHERE id = 7 AND balance ⩾ 30`, y después comprobar que ha cambiado exactamente una fila. Con read committed en PostgreSQL, una actualización concurrente hace que la segunda sentencia espere y después vuelva a evaluar su condición contra la fila nueva, así que no se pierde ninguna actualización.

Cuando la decisión necesita una lógica que la base de datos no puede expresar, usa el bloqueo pesimista: bloquea las filas que lees con `SELECT ... FOR UPDATE`, de modo que las transacciones concurrentes que las quieran esperen hasta que confirmes. Es sencillo y se adapta a operaciones cortas y con mucha contención (asignar un repartidor a un pedido). Los costos son la espera, el riesgo de interbloqueos (bloquea las filas en un orden coherente, por

ejemplo por id ascendente) y un menor throughput. Entre las variantes están `NOWAIT`, que falla de inmediato en lugar de esperar, y `SKIP LOCKED`, que se salta las filas ya bloqueadas y es la forma habitual de construir una cola de trabajos fiable sobre una tabla relacional.

```
BEGIN;
-- Lock the row: other FOR UPDATE readers and writers must wait
SELECT balance FROM accounts WHERE id = 7 FOR UPDATE;
-- application checks: balance (100) ⩾ 30, so allow it
UPDATE accounts SET balance = 70 WHERE id = 7;
INSERT INTO ledger (account_id, amount, kind)
  VALUES (7, -30, 'withdrawal');
COMMIT;  -- lock released; the waiting transaction now sees 70
```

Bloqueo pesimista: la segunda retirada espera en el SELECT hasta que la primera se confirma, y después lee el nuevo saldo.

El control de concurrencia optimista no toma ningún bloqueo mientras el usuario o el código está pensando. Cada fila lleva un número de versión (o una marca de tiempo o un hash). El escritor solo actualiza si la versión sigue siendo la que leyó, y la incrementa. Si no cambia ninguna fila, alguien se le adelantó, y la aplicación recarga y reintenta o se lo comunica al usuario. Esto se adapta a datos con poca contención y a un "tiempo de reflexión" largo: el gerente de un restaurante que edita una carta en un navegador durante cinco minutos no debe mantener un bloqueo de base de datos todo ese tiempo. HTTP incorpora la misma idea, con los ETag y la cabecera `If-Match`.

```
-- Read (no lock); remember version = 12
SELECT id, name, price, version FROM menu_items WHERE id = 501;

-- Later, write only if the version is unchanged
UPDATE menu_items
   SET price = 11.00, version = version + 1
 WHERE id = 501 AND version = 12;

-- Rows affected = 1: success.
-- Rows affected = 0: a concurrent edit won; reload and retry
-- or show the user the other change.
```

Concurrencia optimista con una columna de versión: la actualización solo tiene éxito si nadie ha modificado el elemento de la carta desde que se leyó.

EJEMPLO PRÁCTICO

Ejemplo práctico: write skew con dos médicos de guardia

Un sistema de turnos de un hospital hace cumplir una regla: al menos un médico debe estar de guardia en cada turno. Alice y Bob están de guardia esta noche. Los dos se encuentran mal y pulsan “Salir de guardia” casi en el mismo momento. Cada solicitud ejecuta esta lógica en una transacción: contar los médicos de guardia en el turno; si hay al menos dos, poner mi propia fila como fuera de guardia. La base de datos funciona con aislamiento de instantánea (repeatable read de PostgreSQL o serializable de Oracle).

Inicio	Tabla `doctors`: Alice on_call = true, Bob on_call = true. Regla: count(on_call) >= 1.
Paso 1: T1 (Alice) empieza	T1 toma una instantánea.
Paso 2: T2 (Bob) empieza	T2 toma una instantánea. Ambas instantáneas muestran dos médicos de guardia.
Paso 3: T1 lee	`SELECT count(*) FROM doctors WHERE shift = 1234 AND on_call` devuelve 2. La comprobación se supera.
Paso 4: T2 lee	La misma consulta en la instantánea de T2 también devuelve 2. La comprobación se supera.
Paso 5: T1 escribe	`UPDATE doctors SET on_call = false WHERE name = 'Alice' AND shift = 1234`
Paso 6: T2 escribe	`UPDATE doctors SET on_call = false WHERE name = 'Bob' AND shift = 1234`. Es una fila distinta, así que no se detecta ningún conflicto de escritura-escritura.
Paso 7: ambas confirman	El aislamiento de instantánea no ve ningún conflicto: cada transacción escribió una fila que la otra no escribió. Resultado: nadie está de guardia. Cada transacción era correcta por separado; juntas rompieron la regla.
Con serializable (SSI)	La base de datos detecta que cada transacción leyó datos que la otra modificó después (un ciclo peligroso de dependencias de lectura-escritura). Un commit falla con SQLSTATE 40001; el reintento vuelve a contar, encuentra un médico y se niega.
Solución en un nivel más débil	Bloquea las filas de las que depende la decisión: `SELECT name FROM doctors WHERE shift = 1234 AND on_call FOR UPDATE`, y después cuenta las filas devueltas. Ahora T2 espera a que se liberen los bloqueos de T1. En read committed, T2 vuelve a leer, encuentra solo a Bob de guardia y se niega; en repeatable read de PostgreSQL falla en su lugar con un error de serialización, y el reintento se niega.

El write skew es una generalización de la actualización perdida: las transacciones leen los mismos datos pero escriben filas *distintas*, así que la regla de que gana el primero que confirma, propia del aislamiento de instantánea, nunca se activa.

El mismo patrón aparece en los sistemas de reservas (dos personas que reservan el último asiento o franjas de reunión que se solapan), en los nombres de usuario (dos registros que comprueban que un nombre está libre) y en la banca (dos retiradas de cuentas distintas que juntas superan un límite de descubierto conjunto).

Cuando las filas que demostrarían un conflicto aún no existen, como ocurre con dos reservas que se solapan, `FOR UPDATE` no tiene nada que bloquear. Las opciones son el aislamiento serializable, una restricción de unicidad o de exclusión (PostgreSQL puede prohibir de forma declarativa los rangos de tiempo que se solapan) o *materializar el conflicto*: crear filas, por ejemplo una por sala y franja horaria, con el único fin de poder bloquearlas.

SI ESTO ES NUEVO PARA TI

Paso a paso: cómo hacer segura una lectura seguida de una escritura

El punto en el que más gente tropieza aquí es que poner una lectura y una escritura dentro de una misma transacción no impide que otra transacción se cuele entre ambas. Dos pedidos de un mismo plato muestran por qué, y cómo solucionarlo.

- Al plato del día de un restaurante le quedan 8 raciones, guardadas en una sola fila. Los pedidos X (2 raciones) e Y (3) llegan a la vez, ambos en read committed, el valor por defecto en PostgreSQL y Oracle.
- Cada uno lee 8, resta sus raciones y guarda: X guarda 6 y después Y guarda 5. Las 2 raciones de X han desaparecido del recuento, una actualización perdida, de modo que todavía podrían venderse 2 raciones que no existen. La atomicidad no ayudó: tiene que ver con los fallos, no con turnarse.
- Solución uno, una actualización atómica: la base de datos hace la aritmética y la comprobación en una sola sentencia, como 'resta 3 si quedan al menos 3'. En PostgreSQL, la sentencia de Y espera a que X confirme, vuelve a comprobar contra 6 y deja 3.
- Solución dos, un bloqueo pesimista: cada transacción bloquea la fila al leerla. X llega primero, así que la lectura de Y espera hasta que X confirma, y entonces ve 6, no 8.
- Solución tres, la concurrencia optimista: no hay bloqueo, pero cada escritura indica el número de versión que leyó y solo tiene éxito si la fila sigue teniendo ese número. La escritura de X se aplica e incrementa la versión; la de Y no coincide con ninguna fila, así que Y vuelve a leer y reintenta con 6.
- El aislamiento serializable también lo evita, pero puede hacer que una transacción falle y haya que reintentarla.
- **La lección:** el peligro está en cada lectura-decisión-escritura, se llame como se llame el nivel de aislamiento. Protege cada una con una sentencia atómica, un bloqueo, una comprobación de versión o aislamiento serializable con reintentos.

Cómo elegir una estrategia de aislamiento

No existe un único nivel correcto; existe un nivel correcto para cada fragmento de lógica. Un enfoque práctico parte de las invariantes, no de los ajustes.

Lista de comprobación: cómo hacer segura una ruta de código transaccional

- [ ] **Nombra la invariante**
  Escribe la regla que protege el código (el saldo nunca por debajo de cero, un repartidor por pedido, al menos un médico de guardia). Si no hay ninguna, probablemente el nivel por defecto basta.
- [ ] **Prefiere las restricciones y las sentencias atómicas**
  Los índices únicos, las claves foráneas, las restricciones CHECK y de exclusión, y las actualizaciones condicionales de una sola sentencia las hace cumplir la base de datos sea cual sea el nivel de aislamiento.
- [ ] **Detecta las lecturas-modificaciones-escrituras**
  Todo código que lee un valor, decide en la aplicación y vuelve a escribir es un riesgo de actualización perdida o de write skew. Protégelo con FOR UPDATE, una columna de versión o aislamiento serializable.
- [ ] **Conoce la semántica real de tu base de datos**
  Comprueba qué significan realmente los nombres de los niveles de tu producto (el repeatable read de PostgreSQL y el de MySQL difieren; el serializable de Oracle es aislamiento de instantánea).
- [ ] **Reintenta correctamente**
  Serializable y repeatable read pueden abortar con errores de serialización e interbloqueos. Reintenta la transacción completa con espera progresiva (back-off), y deja los efectos secundarios externos fuera de ella.
- [ ] **Mantén cortas las transacciones**
  Nada de tiempo de reflexión del usuario, llamadas de red ni informes grandes dentro de una transacción; las transacciones largas mantienen bloqueos e inflan el almacenamiento MVCC.
- [ ] **Prueba con concurrencia**
  Ejecuta las rutas críticas de forma concurrente en una prueba de integración contra el motor de base de datos real, no contra un sustituto en memoria con una semántica distinta.

Como regla general, read committed con bloqueo explícito o comprobaciones de versión es un valor por defecto sólido y rápido para la mayoría de las aplicaciones de negocio, siempre que el equipo sea disciplinado a la hora de encontrar las rutas de lectura-modificación-escritura. Serializable es el valor por defecto más seguro cuando las reglas son complejas, el equipo es grande o el costo de una invariante incumplida es alto (dinero, medicina, seguridad), y muchos

equipos lo usan ahora en las rutas críticas, aceptando los reintentos. Los profesionales discrepan de verdad en este punto: unos sostienen que serializable por defecto evita toda una clase de bugs que la revisión de código nunca detectará; otros sostienen que sus reintentos y su contención cuestan demasiado y que el bloqueo explícito hace visible la intención. Ambas posturas son defendibles si la elección es deliberada y está probada.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **¿Por qué el aislamiento de instantánea no es serializable?**

Porque permite el write skew: dos transacciones pueden leer datos que se solapan y escribir filas distintas, rompiendo juntas una regla que cada una comprobó, sin que se detecte ningún conflicto de escritura-escritura.

▼ **¿De qué depende la C de ACID?**

Sobre todo de la aplicación: la base de datos hace cumplir las restricciones declaradas, pero las invariantes de negocio solo se mantienen si las transacciones de la aplicación las preservan.

▼ **¿Cuándo elegirías el control de concurrencia optimista en lugar del pesimista?**

Cuando los conflictos son poco frecuentes o hay un tiempo de reflexión largo entre la lectura y la escritura, como cuando un usuario edita un formulario, de modo que mantener bloqueos sería un desperdicio o imposible.

▼ **¿Por qué el código que usa aislamiento serializable debe poder reintentarse con seguridad?**

Las transacciones serializables pueden abortarse por fallos de serialización o interbloqueos, y la respuesta correcta es volver a ejecutar la transacción completa, así que no debe tener efectos secundarios externos.

▼ **Inténtalo: Dos fans compran la última entrada. Cada transacción lee remaining = 1 y luego escribe remaining = 0 en read committed, así que ambas tienen éxito. Nombra el problema y da la solución más sencilla que usarías, con una alternativa.**

Es una actualización perdida: ambos leyeron, decidieron en el código de la aplicación y volvieron a escribir. La solución más sencilla es una actualización condicional atómica: UPDATE shows SET remaining = remaining - 1 WHERE id = ? AND remaining > 0, y luego comprobar que ha cambiado una fila; si no ha cambiado ninguna, la función está agotada. Como alternativa, usa SELECT … FOR UPDATE, o una columna de versión con reintento. Una restricción CHECK de que remaining sea al menos 0 añade una última red de seguridad.

FUENTES

- Berenson, Bernstein, Gray, Melton, O'Neil and O'Neil: A Critique of ANSI SQL Isolation Levels (1995): https://www.microsoft.com/en-us/research/publication/a-critique-of-ansi-sql-isolation-levels/
- PostgreSQL documentation: Transaction Isolation: https://www.postgresql.org/docs/current/transaction-iso.html
- MySQL 8.4 Reference Manual: Transaction Isolation Levels: https://dev.mysql.com/doc/refman/8.4/en/innodb-transaction-isolation-levels.html
- Microsoft Learn: SET TRANSACTION ISOLATION LEVEL (Transact-SQL): https://learn.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql
- Jepsen: Consistency Models: https://jepsen.io/consistency
- HikariCP wiki: About Pool Sizing: https://github.com/brettwooldridge/HikariCP/wiki/About-Pool-Sizing
- PgBouncer: Features: https://www.pgbouncer.org/features.html

SECCIÓN 7.4 DE 7.8 · AVANZADO · 31 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Replicación, particionamiento y sharding

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.5 Almacenar datos
- 2.5 Disponibilidad y fiabilidad
- 6.6 CAP y consistencia

Términos anteriores que utiliza: Disponibilidad, Latencia, Índice, Registro de escritura anticipada (write-ahead log, WAL), Linealizabilidad

LA IDEA EN BREVE

Imagina una biblioteca muy concurrida. Para seguir atendiendo a los lectores si uno de sus edificios se inunda, guarda copias idénticas de su catálogo en varias sucursales: eso es la replicación. Las copias se actualizan por mensajero, así que una sucursal puede ir un poco atrasada, y un lector que acaba de devolver un libro en una sucursal puede encontrarse con que otra todavía lo tiene como prestado. Cuando la colección se vuelve demasiado grande para un solo edificio, la biblioteca la divide, de la A a la M aquí y de la N a la Z allá: eso es el particionamiento. Dividir bien significa que ninguna sucursal queda desbordada y que los lectores rara vez necesitan visitar todas las sucursales para obtener una respuesta.

LO QUE NECESITAS SABER

- La replicación mantiene copias de los mismos datos en varios nodos para lograr disponibilidad, escalado de lecturas y proximidad; el particionamiento (sharding) reparte los datos entre nodos para lograr throughput de escritura y almacenamiento; los sistemas reales suelen combinar ambos.
- En la replicación líder-seguidor todas las escrituras van a un único líder; los seguidores síncronos protegen las escrituras confirmadas a costa de latencia, mientras que los seguidores asíncronos son rápidos, pero pueden perder escrituras recientes en un failover.
- El retardo de replicación rompe las garantías de leer tus propias escrituras, de lecturas monótonas y de lecturas de prefijo consistente; entre las soluciones están leer del líder las escrituras propias recientes, esperar a una posición del log y fijar a cada usuario a una réplica.

- El failover debe detectar el fallo, elegir el nodo más actualizado y aislar (fencing) al antiguo líder; el split brain, en el que dos nodos aceptan escrituras, puede provocar recuperaciones largas y dolorosas, como mostró el incidente de GitHub de 2018.
- La replicación multilíder debe resolver los conflictos de escritura; la replicación sin líder (al estilo Dynamo) usa quórums con R + W > N, reparación en lectura y antientropía, pero los quórums por sí solos no garantizan la linealizabilidad.
- El particionamiento por rangos admite recorridos por rango, pero crea puntos calientes con claves ordenadas por tiempo; el particionamiento por hash reparte la carga, pero dispersa los rangos; el hashing consistente o las particiones lógicas fijas mantienen barato el reequilibrado.
- Las claves calientes, el desequilibrio (skew) y los índices secundarios (locales implica lecturas de dispersión-recolección; globales implica escrituras de índice remotas, normalmente asíncronas) deben diseñarse a partir de las consultas más importantes.

Dos motivos distintos para repartir los datos

Un único servidor de base de datos acaba quedándose sin algo: disponibilidad (es una sola máquina que puede fallar), capacidad de lectura, capacidad de escritura, almacenamiento o proximidad a usuarios de otros continentes. Hay dos formas complementarias de repartir los datos entre varias máquinas, y es importante no confundirlas. La replicación mantiene una *copia de los mismos datos* en varios nodos, para lograr disponibilidad, escalado de lecturas y menor latencia cerca de los usuarios. El particionamiento, llamado *sharding* en muchos productos, *divide los datos* para que cada nodo guarde un subconjunto, y así lograr throughput de escritura y almacenamiento más allá de una sola máquina. Los sistemas reales suelen hacer ambas cosas: cada partición se replica.

Ambas tienen consecuencias serias para la corrección. La replicación plantea la pregunta de qué ve un lector mientras las copias no coinciden, que es el tema de modelos de consistencia, CAP y PACELC. El particionamiento plantea la pregunta de qué ocurre con las consultas, los índices y las transacciones que abarcan varias particiones. Esta sección explica los mecanismos con la profundidad suficiente para juzgar lo que afirma una base de datos y para diseñar una clave de sharding.

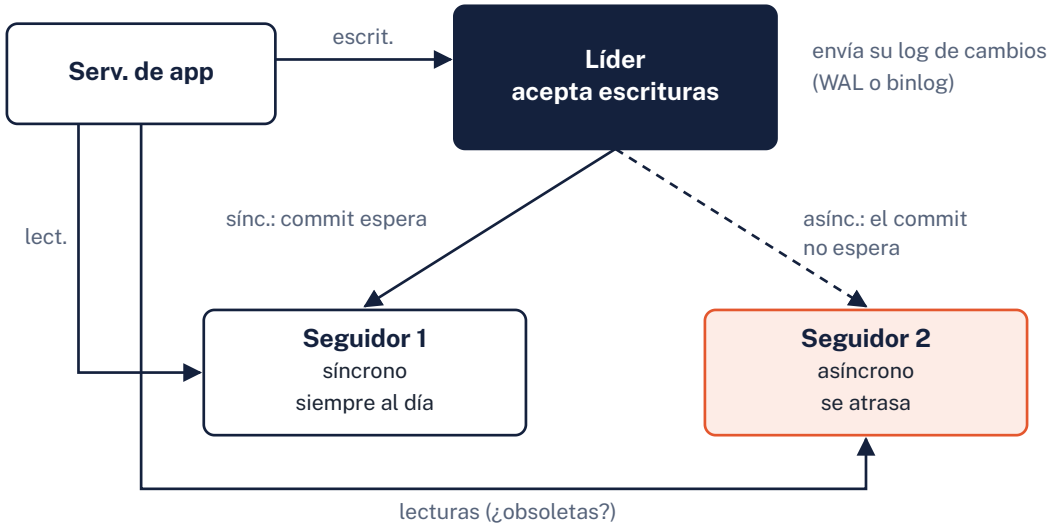
Replicación líder-seguidor

El esquema más común es la replicación líder-seguidor (también llamada primario-réplica; la documentación antigua dice maestro-esclavo). Un nodo, el líder, acepta todas las escrituras. Registra cada cambio en su log (el write-ahead log de PostgreSQL, el binary log de MySQL) y transmite ese log a los seguidores, que aplican los cambios en el mismo orden. Las lecturas

pueden ir al líder o a los seguidores. La replicación en streaming de PostgreSQL, la replicación de MySQL, los grupos de disponibilidad de SQL Server, los conjuntos de réplicas de MongoDB y la mayoría de las bases de datos gestionadas en la nube funcionan así.

La decisión de diseño clave es cuándo comunica el líder que un commit ha tenido éxito. Con la replicación **síncrona**, espera hasta que un seguidor confirma que tiene el cambio, de modo que una escritura confirmada sobrevive a la pérdida del líder; el precio es que cada commit incluye un viaje de ida y vuelta por la red y que, si el seguidor síncrono está caído, las escrituras se detienen. Con la replicación **asíncrona**, el líder confirma de inmediato y envía el cambio después; los commits son rápidos y los seguidores pueden estar lejos, pero si el líder muere, se pierden los últimos cambios que no habían llegado a ningún seguidor, aunque al cliente se le dijo que estaban confirmados. Un compromiso habitual, llamado semisíncrono en MySQL, espera a un seguidor y deja que el resto sea asíncrono. En el banco, perder un pago confirmado es inaceptable, así que lo normal es tener al menos una copia síncrona (o una base de datos basada en consenso); para la base de datos de la carta de la plataforma de reparto de comida, unos segundos de posible pérdida pueden ser un precio aceptable a cambio de velocidad. Este es, en forma concreta, el compromiso del RPO visto en disponibilidad y fiabilidad.

Líder con un seguidor síncrono y otro asíncrono



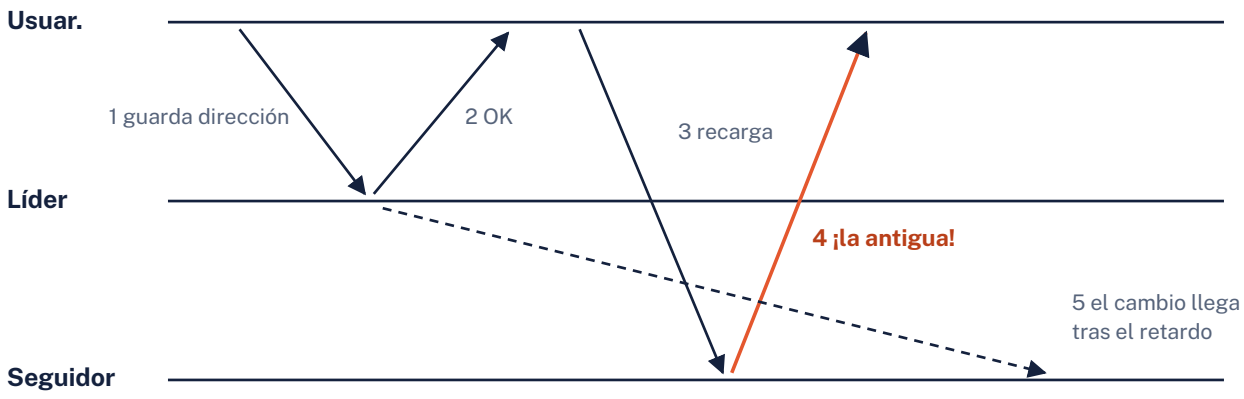
Todas las escrituras van al líder; su log de cambios fluye hacia los seguidores, y el commit solo espera al síncrono.

El retardo de replicación y las anomalías que causa

Los seguidores asíncronos suelen ir milisegundos por detrás, pero con carga, durante una actualización por lotes grande o a través de una red lenta, el retardo de replicación puede crecer hasta segundos o minutos. Leer de los seguidores produce entonces anomalías que los

usuarios notan. Douglas Terry y sus colaboradores definieron las dos primeras garantías que siguen, leer tus propias escrituras y lecturas monótonas, como *garantías de sesión* en 1994; la tercera importa sobre todo cuando los datos también están particionados.

Leer tus propias escrituras, incumplido



Solución: leer del líder los cambios recientes del usuario, o esperar a que el seguidor se ponga al día

El líder confirma el cambio de la propia usuaria, pero aún no se ha aplicado en el seguidor que atiende su siguiente lectura.

Anomalías del retardo y sus soluciones

Garantía	Qué sale mal sin ella	Soluciones habituales
Leer tus propias escrituras (read-your-writes, read-after-write)	Una clienta cambia su dirección de entrega, recarga y ve la antigua: parece que el guardado falló, así que lo vuelve a intentar.	Leer del líder los datos que el usuario puede haber acabado de cambiar (por ejemplo, durante un minuto después de su última escritura); o registrar la posición del log del líder en el momento de escribir y leer de un seguidor solo cuando haya aplicado esa posición.
Lecturas monótonas	La app de un repartidor se actualiza dos veces; la primera lectura llega a un seguidor al día y muestra un pedido como asignado, la segunda llega a uno retrasado y el pedido vuelve a parecer sin asignar. El tiempo parece ir hacia atrás.	Dirigir las lecturas de cada usuario al mismo seguidor (por ejemplo, aplicando un hash al id del usuario) y cambiar a otro solo cuando no esté disponible.
Lecturas de prefijo consistente	Los lectores ven una respuesta antes que la pregunta, o el evento “entregado” de un pedido antes que “recogido”, porque los dos se escribieron en particiones distintas que se retrasan de forma distinta.	Escribir los datos relacionados causalmente en la misma partición, o seguir explícitamente las dependencias causales.

Ninguna de estas anomalías existe si todas las lecturas van al líder, así que la solución más sencilla suele ser enviar al líder las lecturas que deben estar actualizadas y usar los seguidores solo para las lecturas que toleran datos obsoletos: listados de productos, informes, feeds de búsqueda. Diseña esto de forma deliberada para cada consulta, en lugar de dirigir “todas las lecturas a las réplicas” por defecto.

Failover y split brain

El failover (conmutación por error) sustituye a un líder que ha fallado por un seguidor. Tiene tres pasos: detectar que el líder ha fallado (normalmente mediante un timeout), elegir el nuevo líder (idealmente el seguidor más actualizado, a menudo mediante un protocolo de consenso como Raft entre los nodos o un coordinador externo como etcd o ZooKeeper) y redirigir las escrituras hacia él. Cada paso puede salir mal. Un timeout demasiado corto provoca un failover durante un breve corte de red o un pico de carga, lo que empeora las cosas; uno demasiado largo alarga la interrupción. Con la replicación asíncrona, se pierden las escrituras que el antiguo líder aún no había enviado y, si el antiguo líder vuelve y todavía las conserva, entran en conflicto con las escrituras nuevas.

El fallo más peligroso es el split brain (cerebro dividido): dos nodos creen ser el líder y ambos aceptan escrituras, de modo que los datos divergen de maneras que puede ser imposible fusionar automáticamente. Suele producirse tras una partición de red en la que el antiguo líder queda aislado pero sigue vivo. La protección necesita un quórum (un nodo solo puede liderar mientras una mayoría esté de acuerdo), *fencing* (la autoridad del nuevo líder se impone mediante un token siempre creciente que el almacenamiento comprueba, o se apaga a la fuerza el antiguo líder, una técnica apodada STONITH: shoot the other node in the head, es decir, dispara al otro nodo en la cabeza) y clientes que sigan de forma fiable al líder actual.

El incidente de GitHub del 21 de octubre de 2018 muestra lo difícil que es esto en la práctica. Un fallo de red cortó la conexión entre su nodo central de la Costa Este de EE. UU. y su centro de datos principal de la Costa Este durante 43 segundos. Sus herramientas de failover, funcionando según lo previsto, promovieron a primarios las bases de datos de la Costa Oeste. Cuando volvió la conectividad, ambas costas tenían escrituras de las que carecía la otra, así que GitHub no pudo simplemente volver atrás; para proteger la integridad de los datos decidió restaurar y conciliar, y el servicio estuvo degradado durante 24 horas y 11 minutos. El failover automático entre regiones había convertido un corte de 43 segundos en una recuperación de un día entero.

Copias de seguridad, y cómo demostrar que se pueden restaurar

La replicación no es una copia de seguridad: una réplica copia fielmente un `DELETE` equivocado. La copia base de PostgreSQL más el archivado continuo del WAL permite recuperar hasta un momento elegido justo antes del error, y las copias deben estar donde un mismo incidente no pueda alcanzar a ambas. Una copia de seguridad solo cuenta una vez que se ha restaurado. GitLab lo aprendió en enero de 2017: después de que se borraran datos de producción por error, resultó que sus copias periódicas con `pg_dump` habían estado fallando por una discrepancia de versiones, los correos de aviso de los fallos habían sido rechazados y el bucket de almacenamiento estaba vacío. Se recuperó a partir de una instantánea manual de unas seis horas de antigüedad. Por eso, automatiza restauraciones periódicas en un entorno aislado, comprueba los datos, mide el tiempo empleado frente al RTO y el RPO, y genera una alerta cuando una copia o una prueba de restauración falle o no se ejecute.

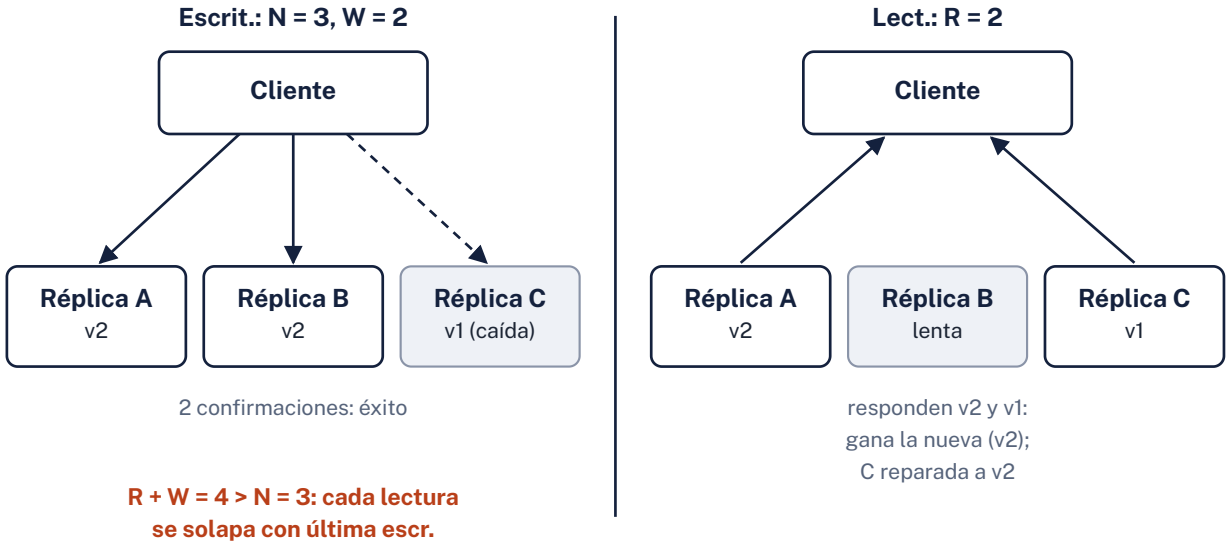
Replicación multilíder y sin líder

En la replicación multilíder, varios nodos aceptan escrituras y se replican entre sí. Se usa para varios centros de datos (cada región escribe localmente, así que los usuarios evitan la latencia de cruzar un océano), para clientes que funcionan sin conexión (una app de calendario o de notas es en la práctica un líder en cada dispositivo) y para la edición colaborativa. El problema inevitable son los **conflictos de escritura**: dos líderes modifican el mismo registro de forma

concurrente. Entre las soluciones están last write wins (sencillo, pero descarta datos en silencio y depende de los relojes), fusionar los valores, conservar ambos y preguntar a una persona, o usar tipos de datos diseñados para fusionarse automáticamente (CRDT, tipos de datos replicados libres de conflictos). El enfoque más fiable es evitar los conflictos: asignar a cada registro una región de origen que sea la única que acepta sus escrituras, como el país de residencia de un cliente.

La replicación sin líder, popularizada por el artículo Dynamo de Amazon (DeCandia y colaboradores, 2007) y usada en Apache Cassandra, ScyllaDB y Riak, no tiene ningún líder. El cliente, o un nodo coordinador que actúa en su nombre, envía cada escritura a las N réplicas de una clave y la considera correcta cuando W de ellas la confirman; envía cada lectura a las réplicas y espera R respuestas, quedándose con la versión más reciente. Si R + W > N, cada conjunto de lectura se solapa con cada conjunto de escritura correcta en al menos una réplica, así que una lectura normalmente ve la última escritura confirmada. Las réplicas que se perdieron escrituras se reparan mediante la *reparación en lectura* (read repair: el coordinador vuelve a escribir la versión más reciente cuando ve una obsoleta) y una *antientropía* en segundo plano que compara las réplicas. Cassandra expone estas opciones en cada solicitud como niveles de consistencia, como ONE, QUORUM y ALL.

Una escritura y una lectura de quórum con N = 3



Con W = 2 y R = 2, al menos una réplica de cada lectura ha visto la última escritura confirmada.

A menudo se confía demasiado en los quórums. R + W > N no hace por sí solo que el sistema sea linealizable: las escrituras concurrentes, las escrituras que fallaron en algunas réplicas pero tuvieron éxito en otras, los *quórums laxos* (sloppy quorums: escribir en nodos sustitutos durante los fallos) y last-write-wins con relojes desfasados permiten datos obsoletos o perdidos. El Capítulo 6 recorre la aritmética de los quórums y estos casos límite en CAP y

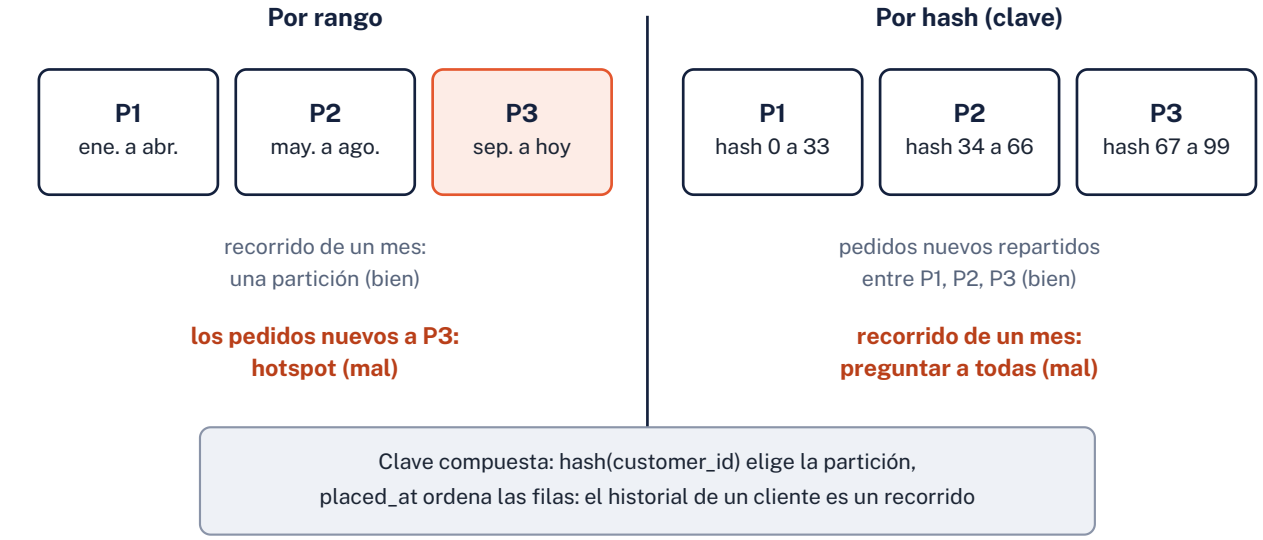
PACELC. Los almacenes sin líder son excelentes para lograr una alta disponibilidad de escritura frente a fallos y entre centros de datos; encajan mal con invariantes como “no vender nunca dos veces el mismo asiento”, salvo que ofrezcan una transacción ligera o una función de consenso para esas operaciones.

Particionamiento: por rango de claves o por hash

El particionamiento asigna cada registro a una partición según su **clave de partición** (o clave de sharding), y el objetivo es repartir los datos y la carga de forma uniforme. Una partición que recibe una carga desproporcionada es un punto caliente, y un reparto desigual se llama *desequilibrio (skew)*. Hay dos esquemas básicos.

El **particionamiento por rango de claves** da a cada partición un rango contiguo de claves, como los tomos de una enciclopedia. Los recorridos por rango son eficientes porque las claves adyacentes se guardan juntas (todos los pedidos de octubre en una o dos particiones), y las particiones pueden dividirse cuando crecen, como en Bigtable, HBase, Spanner y el sharding por rangos de MongoDB. El peligro es una clave que aumenta con el tiempo: si la clave es una marca de tiempo o un número de pedido secuencial, cada escritura nueva cae en la última partición, que se convierte en un punto caliente mientras las demás están ociosas. El **particionamiento por hash** aplica una función hash a la clave y asigna rangos de valores hash a las particiones, de modo que reparte uniformemente incluso las claves secuenciales. El precio es que las consultas por rango sobre la clave tienen que preguntar ahora a todas las particiones. Muchos almacenes combinan ambos con una *clave compuesta*: Cassandra aplica un hash a la clave de partición para elegir el nodo y ordena las filas dentro de la partición por columnas de agrupamiento (clustering columns), de modo que `(customer_id, placed_at)` pone juntos los pedidos de un cliente, en orden temporal, en un solo nodo.

Particionamiento por rango frente a por hash para los pedidos

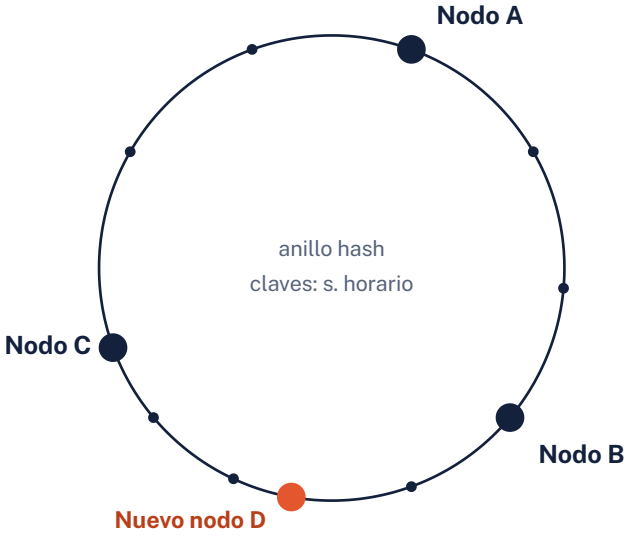


El particionamiento por rango mantiene juntos los rangos, pero concentra las escrituras ordenadas por tiempo; el hash reparte las escrituras, pero dispersa los rangos.

Hashing consistente y reequilibrado

La forma obvia de asignar hashes a nodos, `hash(key) mod N`, tiene un defecto fatal: cuando N cambia, casi todas las claves se mueven. Pasar de 4 a 5 nodos reubica alrededor del 80% de las claves (cualquier clave cuyo resto difiera para 4 y para 5), y pasar de 10 a 11, alrededor del 91%, lo que significa copiar casi todo el conjunto de datos solo para añadir una máquina. El hashing consistente, introducido por David Karger y sus colaboradores en 1997 para las cachés web, coloca tanto los nodos como las claves en un espacio de hash circular; cada clave pertenece al siguiente nodo en el sentido de las agujas del reloj. Un nodo nuevo asume solo las claves situadas entre él y su predecesor, alrededor de 1/N del total, y al retirar uno, este entrega sus claves a su vecino. Con pocos nodos los arcos son desiguales, así que a cada nodo físico se le asignan muchos *nodos virtuales* (tokens) en distintos puntos del anillo, lo que equilibra la carga y permite que un nodo nuevo tome un poco de cada nodo existente. Dynamo y Cassandra usan este esquema.

Hashing consistente en un anillo



Añadir nodo D

hash mod N, de 4 a 5 nodos:
se mueve ~80% de claves

hashing consistente:
solo se mueven las claves
entre B y D (C → D),
aprox. 1/N del total

nodos virtuales: un servidor
ocupa muchos puntos del anillo
y equilibra la carga

Las claves pertenecen al siguiente nodo en sentido horario, así que un nodo nuevo asume un arco en lugar de redistribuirlo todo.

Hay otras dos estrategias de reequilibrado habituales. Con un **número fijo de particiones**, se crean al principio muchas más particiones que nodos (por ejemplo, 1024 para 10 nodos) y se asignan particiones enteras a los nodos; añadir un nodo mueve unas pocas particiones enteras y nunca vuelve a aplicar el hash a las claves. Los shards de Elasticsearch, los vBuckets de Couchbase y muchos esquemas de sharding hechos a mano usan este enfoque, y la decisión principal es elegir un número de particiones lo bastante grande para el crecimiento futuro. Con el **particionamiento dinámico**, como en HBase y MongoDB, una partición se divide cuando supera un umbral de tamaño y las mitades pueden moverse a nodos distintos. Sea cual sea el esquema, el reequilibrado mueve grandes cantidades de datos por la red mientras el sistema atiende tráfico, así que muchos equipos prefieren que una persona apruebe cada movimiento, y lo limitan para proteger la latencia.

Las solicitudes también deben dirigirse a la partición correcta: o bien cualquier nodo acepta una solicitud y la reenvía (los nodos de Cassandra conocen el anillo mediante gossip), o bien una capa de enrutamiento conoce el mapa de particiones (el `mongos` de MongoDB), o bien la biblioteca cliente guarda el mapa. El propio mapa suele guardarse en un servicio de coordinación como etcd o ZooKeeper, o en el grupo de consenso de la propia base de datos.

SI ESTO ES NUEVO PARA TI

Paso a paso: añadir un nodo sin moverlo todo

La idea más difícil de esta sección es por qué la forma obvia de repartir las claves entre los nodos se desmorona cuando se añade un nodo, y cómo lo evita el hashing consistente. Sigue los pedidos de la plataforma de reparto de comida a medida que crece un clúster.

- Una función hash convierte la clave de cada pedido en un número. Supón, para simplificar, que los números van de 0 a 11 y que un pedido va al nodo 'hash mod 3', el resto de dividir entre 3. El pedido 7 está en el nodo 1.
- Se añade un cuarto nodo y la regla pasa a ser 'hash mod 4'. Ahora el pedido 7 corresponde al nodo 3.
- De los doce números, solo 0, 1 y 2 dan el mismo resto para 3 y para 4. Los otros nueve pedidos, el 75% de ellos, deben copiarse a otro nodo solo para añadir una máquina.
- En cambio, el hashing consistente sitúa los nodos y las claves en un círculo de valores hash, y cada clave pertenece al siguiente nodo en el sentido de las agujas del reloj. En el círculo dibujado arriba, el nuevo nodo D se sitúa entre B y C.
- Solo cambian de propietario las claves del arco entre B y D, que pasan de C a D. Todas las demás claves se quedan donde están, así que se mueve aproximadamente la parte de los datos de un nodo, no la mayoría.
- Con pocos puntos, los arcos son desiguales y D solo alivia a C, así que cada servidor recibe muchos puntos alrededor del círculo, llamados nodos virtuales, y un servidor nuevo toma una pequeña porción de cada uno de los existentes.
- **La lección:** juzga una regla de ubicación por lo que ocurre cuando cambia el clúster, no solo por lo uniformemente que reparte hoy las claves. El hashing consistente, o muchas particiones creadas de antemano y movidas enteras, mantiene pequeño el movimiento cuando se une un nodo.

Claves calientes y desequilibrio

El hash reparte uniformemente las *claves*, no la *carga*. Si una clave es mucho más activa que el resto, su partición está caliente sea cual sea el esquema: la cuenta de una celebridad en una red social, el restaurante insignia de una cadena nacional durante una promoción o la propia cuenta de liquidación de un banco, que interviene en cada pago. Los almacenes gestionados hacen explícitos los límites; Amazon DynamoDB, por ejemplo, documenta un máximo por partición de 3000 unidades de lectura y 1000 unidades de escritura por segundo. Entre las mitigaciones están almacenar en caché las lecturas calientes, dividir una clave caliente añadiéndole un pequeño sufijo aleatorio (escrituras de `restaurant:88#0` a `restaurant:88#9`,

a costa de leer las diez y combinarlas), mover un cliente (tenant) muy grande a particiones dedicadas y agregar contadores en memoria antes de escribirlos. Todas añaden complejidad, así que primero mide qué claves están realmente calientes.

SaaS multiinquilino: silo, pool y puente

Supón que la plataforma de reparto de comida también vende su software de pedidos a cadenas de restaurantes, y que cada cadena es un **inquilino (tenant)**. La pregunta clave es qué comparten los inquilinos. AWS SaaS Lens nombra tres modelos. En el modelo **silo**, cada inquilino tiene recursos dedicados, quizá su propia base de datos, lo que da el aislamiento más fuerte y el costo más alto, con muchas bases de datos que aprovisionar y migrar. En el modelo **pool**, los inquilinos lo comparten todo y cada fila lleva un ID de inquilino, lo que es lo más barato y lo más sencillo de operar. El modelo **puente** (bridge) mezcla ambos por componente: un servicio puede estar en silo por motivos regulatorios o de vecinos ruidosos, mientras otro está en pool. Muchos productos también mezclan por cliente: mantienen en pool a las cadenas pequeñas y dan a una cadena nacional su propia base de datos.

En un pool, un solo filtro olvidado muestra a una cadena los pedidos de su rival. La **seguridad a nivel de fila** traslada la comprobación a la base de datos. En PostgreSQL, una política como `USING (tenant_id = current_setting('app.tenant_id')::uuid)` limita todas las consultas sobre la tabla. Las trampas importan. Los superusuarios y los roles `BYPASSRLS` se saltan las políticas, y también los propietarios de las tablas, salvo que se establezca `FORCE ROW LEVEL SECURITY`, así que la aplicación debe conectarse con un rol normal. Con el pooling de transacciones de PgBouncer no se admite un `SET` de sesión, así que establece el inquilino con `SET LOCAL` dentro de cada transacción. Las comprobaciones de unicidad y de claves foráneas omiten las políticas, lo que puede revelar que un valor existe. Trata la política como una última barrera, con pruebas que intenten cruzar de un inquilino a otro, y recuerda que las claves de caché, las colas y los índices de búsqueda también necesitan el inquilino.

Un **vecino ruidoso** (noisy neighbour) es un inquilino cuya actividad perjudica a otros, como la promoción nacional de una cadena que agota la capacidad compartida. La guía de Microsoft lo trata como gobernanza de recursos: cuotas por inquilino y límites de tasa, topes en las operaciones costosas, tareas pesadas trasladadas a horas de poca actividad y telemetría etiquetada con el ID del inquilino para poder ver la causa. Después, los inquilinos pueden reequilibrarse o trasladarse a capacidad dedicada.

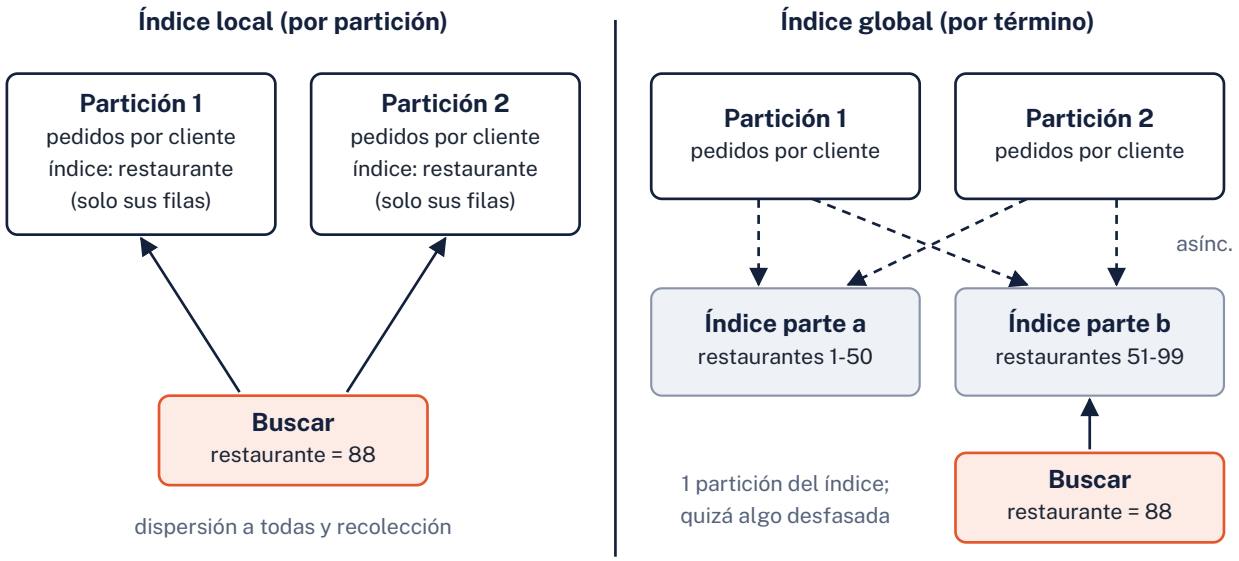
Cuando una base de datos no basta, el **sharding consciente del inquilino** usa el ID del inquilino como clave de partición. En Citus, por ejemplo, todas las tablas llevan la columna del inquilino, que también forma parte de las claves primarias y foráneas, de modo que las filas de un inquilino permanecen juntas y las consultas y transacciones de un solo inquilino se ejecutan en un nodo como PostgreSQL normal. El hashing no resuelve la asimetría, así que una cadena muy

grande puede necesitar su propio shard; el reequilibrador de Citus no tiene en cuenta qué inquilinos comparten un shard, y esos traslados requieren herramientas. Los informes que abarcan todos los inquilinos corresponden a un almacén de datos. Decide pronto también sobre las restauraciones: Microsoft señala que restaurar un inquilino desde una base de datos compartida puede significar restaurar toda la base de datos en otro lugar y recuperar selectivamente los datos de ese inquilino. Las celdas llevan la misma idea más lejos, a toda la pila.

Índices secundarios entre particiones

El particionamiento funciona sin problemas para las búsquedas por la clave de partición. Las consultas por cualquier otro atributo necesitan un índice secundario, y hay dos formas de particionarlo. Un **índice local** (particionado por documento) vive con cada partición y cubre solo las filas de esa partición. Las escrituras actualizan una sola partición, pero una consulta por el atributo indexado debe enviarse a todas las particiones y fusionar los resultados (*dispersión-recolección*, scatter-gather), así que su latencia la marca la partición más lenta y su costo crece con el número de particiones. Un **índice global** (particionado por término) está a su vez particionado por el valor indexado, así que una consulta lee una sola partición del índice; pero una sola escritura puede necesitar ahora actualizar particiones del índice en otros nodos, lo que normalmente se hace de forma asíncrona, así que el índice puede ir brevemente por detrás de los datos. DynamoDB muestra ambos: un índice secundario local comparte la clave de partición de la tabla, mientras que un índice secundario global tiene la suya propia y es eventualmente consistente.

Índices secundarios locales y globales



Un índice local abarata las escrituras y dispersa las consultas; un índice global hace que las consultas sean dirigidas y que las actualizaciones del índice sean remotas y normalmente asíncronas.

EJEMPLO PRÁCTICO

Ejemplo práctico: dimensionar y elegir una clave de sharding para los pedidos

La plataforma de reparto de comida ha crecido hasta 30 países y 10 millones de pedidos al día, con un pico vespertino del triple de la tasa media. Cada pedido genera unas 10 escrituras a lo largo de su vida (creación, líneas, cambios de estado, referencias de pago) y ocupa unos 4 KB, índices incluidos. Los clientes pueden ver todo su historial de pedidos; la normativa exige que los pedidos puedan recuperarse durante seis años, pero el personal de soporte solo necesita acceso en línea a los dos últimos.

Tasa de escritura	10 000 000 / 86 400 s = unos 116 pedidos por segundo de media; en el pico, unos 350 por segundo; con 10 escrituras cada uno, unas 3500 escrituras por segundo en el pico. Un único primario bien ajustado puede con esto; el throughput por sí solo no obliga a hacer sharding.
Crecimiento del almacenamiento	10 000 000 x 4 KB = 40 GB al día, unos 14.6 TB al año, unos 29 TB para dos años en línea.
¿Necesitamos hacer sharding?	Si solo tuvieran que estar en línea 90 días (3.6 TB), un primario más réplicas, con particionamiento de tablas por tiempo y archivado en el almacén de datos, sería la respuesta aburrida y correcta. Con dos años en línea y 29 TB, los tiempos de restauración y de reconstrucción de réplicas en un solo nodo se vuelven inaceptables, así que el equipo hace sharding.
Dimensionar los shards	Objetivo de unos 2 TB por shard para que una réplica pueda reconstruirse en horas: 29 TB / 2 TB = unos 15, así que 16 shards, cada uno con un líder y dos seguidores. Se crean 1024 particiones lógicas asignadas a los 16 shards, para que crecer signifique mover particiones enteras, no volver a aplicar el hash. Escrituras en el pico por shard: 3500 / 16 = unas 220 por segundo.
Candidata: order_id (hash)	Escrituras perfectamente uniformes, pero tanto el historial de un cliente como la cola en vivo de un restaurante pasan a ser de dispersión-recolección entre 16 shards. Descartada como clave principal de ubicación.
Candidata: placed_at (rango)	Muy buena para informes por rango de tiempo, pero cada pedido nuevo escribe en la partición más reciente: un punto caliente garantizado. Descartada.
Candidata: restaurant_id	La cola de un restaurante es local, pero las grandes cadenas y las promociones crean claves calientes, y el historial del cliente se dispersa. Descartada.
Candidata: país o ciudad	Facilita la residencia de datos y las consultas locales, pero una sola ciudad grande puede concentrar el 15% de todos los pedidos: desequilibrio. Se conserva solo como primer nivel grueso para la residencia (clústeres de la UE y de fuera de la UE), no como clave de sharding.

Decisión	Sharding por `hash(customer_id)`, con las filas ordenadas por `placed_at` dentro de cada partición. El id del pedido incorpora el número de partición del cliente, de modo que una búsqueda por id de pedido va directamente a un solo shard.
Cubrir las demás consultas	La cola en vivo del restaurante la atiende un modelo de lectura independiente con clave de restaurante, construido a partir de los eventos de pedido (un índice global deliberado y eventualmente consistente). La analítica se ejecuta en el almacén de datos. No se necesitan transacciones entre clientes: cada pedido afecta a un solo cliente.

Las cifras cambiaron la pregunta. El throughput nunca obligó a hacer sharding; lo hicieron los datos conservados en línea. Reducir la ventana en línea habría eliminado por completo la necesidad, algo que vale la pena plantear al negocio antes de construir un sistema con sharding.

La clave de sharding se eligió a partir de las consultas más importantes y de sus necesidades de consistencia, no solo a partir del modelo de datos. A cada una de las demás consultas se le dio una ruta explícita, diseñada por separado.

Elegir una clave de sharding es una de las decisiones más difíciles de revertir: cambiarla más adelante significa mover todas las filas. Regístrala en un ADR con los volúmenes y las consultas que se supusieron.

Errores comunes

Tratar las réplicas como copias de seguridad. Una réplica copia fielmente un `DELETE` erróneo en milisegundos; siguen haciendo falta copias de seguridad y recuperación a un momento dado (Capítulo 0). **Leer de las réplicas por defecto** sin decidir qué consultas toleran el retardo. **Confiar en un failover automático que nunca se ha ensayado**, o que puede promover un nodo mientras el antiguo líder sigue aceptando escrituras. **Hacer sharding demasiado pronto**: una base de datos con sharding complica los joins, las transacciones, los cambios de esquema y las operaciones, y los nodos individuales modernos son muy grandes; agota primero la indexación, la caché, las réplicas de lectura, el archivado y el escalado vertical. **Hacer sharding por una clave que crece de forma monótona**, lo que concentra cada inserción en una sola partición. **Olvidar las rutas de acceso secundarias** y descubrir después de la división que una consulta importante debe preguntar ahora a todos los shards. Como muestra el resto de este capítulo, las buenas alternativas suelen implicar datos derivados: modelos de lectura, cachés y almacenes analíticos, cada uno construido para un tipo de consulta.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **¿Por qué la replicación asíncrona corre el riesgo de perder escrituras que a los clientes se les dijo que estaban confirmadas?**

El líder confirma antes de que ningún seguidor tenga el cambio; si falla antes de enviarlo, el seguidor promovido nunca recibió esas escrituras.

▼ **Un usuario guarda un cambio y ve de inmediato el valor antiguo. ¿Qué garantía se incumple y cuál es una solución?**

Leer tus propias escrituras (read-your-writes). Atiende desde el líder las lecturas que ese usuario haga de datos modificados recientemente, o espera hasta que el seguidor haya aplicado la posición del log de la escritura.

▼ **¿Por qué hash mod N es una mala forma de asignar claves a nodos?**

Cambiar N mueve la mayoría de las claves (alrededor del 80% al pasar de 4 a 5 nodos), mientras que el hashing consistente o las particiones lógicas fijas mueven solo alrededor de 1/N.

▼ **¿Cuál es el compromiso entre los índices secundarios locales y los globales?**

Los índices locales mantienen las escrituras en una sola partición, pero obligan a las consultas a preguntar a todas las particiones; los índices globales permiten que las consultas vayan a una sola partición del índice, pero exigen actualizaciones del índice entre particiones, normalmente asíncronas.

▼ **Inténtalo: Una plataforma recibe 6 millones de pedidos al día, cada uno causa 10 escrituras y el pico es tres veces la media. Cada shard aguanta con seguridad 500 escrituras por segundo. ¿Cuántos shards hacen falta? Compara el ID de cliente y la marca de tiempo del pedido como clave de sharding.**

Media: 60 000 000 / 86 400 = 694 escrituras por segundo. Pico: unas 2083. Shards: 2083 / 500 = 4.2, es decir, 5, más margen para crecer. Un hash del ID de cliente reparte la carga de manera uniforme y mantiene junto el historial de cada cliente, la consulta más importante. Una clave de marca de tiempo envía todas las escrituras nuevas a un solo shard, lo que crea un punto caliente. Los informes por rango de tiempo necesitan entonces scatter-gather (dispersión-recolección) o, mejor, un almacén analítico separado. Añade réplicas a cada shard para la disponibilidad.

FUENTES

- DeCandia et al.: Dynamo: Amazon's Highly Available Key-value Store (SOSP 2007): https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
- GitHub: October 21 post-incident analysis (2018): https://github.blog/2018-10-30-oct21-post-incident-analysis/
- PostgreSQL documentation: High Availability, Load Balancing, and Replication (log-shipping standby servers): https://www.postgresql.org/docs/current/warm-standby.html
- Amazon DynamoDB Developer Guide: Best practices for designing and using partition keys effectively: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-partition-key-design.html
- Microsoft Azure Architecture Center: Sharding pattern: https://learn.microsoft.com/en-us/azure/architecture/patterns/sharding
- AWS SaaS Lens: Silo, pool, and bridge models: https://docs.aws.amazon.com/wellarchitected/latest/saas-lens/silo-pool-and-bridge-models.html
- PostgreSQL documentation: Row Security Policies: https://www.postgresql.org/docs/current/ddl-rowsecurity.html
- PostgreSQL documentation: SET: https://www.postgresql.org/docs/current/sql-set.html
- PgBouncer: Features: https://www.pgbouncer.org/features.html
- Microsoft Learn: Noisy Neighbor antipattern: https://learn.microsoft.com/en-us/azure/architecture/antipatterns/noisy-neighbor/noisy-neighbor
- Citus documentation: Multi-tenant applications: https://docs.citusdata.com/en/stable/use_cases/multi_tenant.html
- Microsoft Learn: Multitenant data approaches: https://learn.microsoft.com/en-us/azure/architecture/guide/multitenant/approaches/storage-data
- GitLab: Postmortem of database outage of January 31 (2017): https://about.gitlab.com/blog/2017/02/10/postmortem-of-database-outage-of-january-31/
- PostgreSQL documentation: Continuous archiving and point-in-time recovery: https://www.postgresql.org/docs/current/continuous-archiving.html

SECCIÓN 7.5 DE 7.8 · AVANZADO · 26 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

CQRS y event sourcing

ANTES DE LEER ESTO

Esta sección se basa en:

- 5.7 Diseño guiado por el dominio
- 6.4 Dirigida por eventos
- 7.3 Transacciones

Términos anteriores que utiliza: Agregado, Contexto delimitado (bounded context), Consistencia eventual, Control de concurrencia optimista

LA IDEA EN BREVE

Un banco lleva un libro mayor en el que se anota cada depósito y cada retiro, una línea tras otra, y nunca se borra nada; los errores se corrigen añadiendo una línea de corrección. Tu saldo es simplemente la suma de todas las líneas. Tu extracto mensual es un resumen ordenado que se prepara a partir de ese libro mayor para que se lea con facilidad. El event sourcing almacena los datos como lo hace el libro mayor: como una lista de cosas que ocurrieron. CQRS mantiene el libro mayor (para hacer cambios) separado de los extractos (para consultar cosas). Ambos son potentes, y ambos añaden una complejidad real.

LO QUE NECESITAS SABER

- La separación comando-consulta (Bertrand Meyer, 1988) dice que un método debe cambiar el estado o devolver una respuesta, pero no ambas cosas; CQRS (Greg Young, hacia 2010) aplica la misma división a modelos completos.
- En CQRS, un modelo de escritura hace cumplir las reglas de negocio sobre los comandos, mientras que las proyecciones mantienen actualizados uno o más modelos de lectura desnormalizados para las consultas; los modelos de lectura son datos derivados que se pueden reconstruir.
- El event sourcing almacena cada cambio como un evento inmutable, con nombre en pasado, en un flujo de solo anexado (append-only) por agregado; el estado actual es un plegado por la izquierda (left fold) determinista de los eventos.
- El event sourcing ofrece una pista de auditoría completa, consultas temporales, modelos de lectura reconstruibles y retroactivos, y la reproducción para depurar; los snapshots (instantáneas) acortan los plegados largos, pero son solo una caché.

- Los costos son reales: complejidad, consistencia eventual de los modelos de lectura, versionado de eventos mediante upcasting o nuevos tipos de evento, y el borrado de datos personales de un log inmutable, normalmente mediante crypto-shredding.
- Martin Fowler advierte que CQRS añade una complejidad arriesgada en la mayoría de los sistemas; aplica cualquiera de los dos patrones a contextos delimitados concretos con necesidades de lectura y escritura realmente distintas, nunca a todo un sistema por defecto.

De la separación comando-consulta a CQRS

La idea empieza siendo pequeña. En *Object-Oriented Software Construction* (primera edición, 1988), Bertrand Meyer propuso la separación comando-consulta: todo método debe ser o bien un comando que cambia el estado y no devuelve nada, o bien una consulta que devuelve una respuesta y no cambia nada. Hacer una pregunta nunca debería cambiar la respuesta. Martin Fowler señala que la regla tiene excepciones razonables (sacar un elemento de una pila cambia la pila y a la vez devuelve el elemento), pero como norma por defecto hace que sea más fácil razonar sobre el código, porque las consultas pueden llamarse libremente, en cualquier orden, sin efectos secundarios.

Hacia 2010, Greg Young elevó el principio de los métodos a los modelos completos y llamó al resultado Command Query Responsibility Segregation (CQRS). En lugar de un único modelo de objetos, y normalmente un único conjunto de tablas, que sirve tanto para las actualizaciones como para las lecturas, el sistema tiene un **modelo de escritura** que gestiona los comandos y hace cumplir las reglas de negocio, y uno o más **modelos de lectura** con la forma de las preguntas que hacen los usuarios. La idea clave es que escribir y leer son trabajos distintos. Escribir necesita reglas estrictas, transacciones pequeñas y una estructura normalizada para que cada hecho se almacene una sola vez. Leer necesita velocidad, formas desnormalizadas que coincidan con las pantallas y los informes, y a menudo mucho más tráfico. Un único modelo que intenta hacer bien ambas cosas tiende a hacer mal las dos en cuanto el dominio se vuelve rico.

Qué cambia realmente CQRS

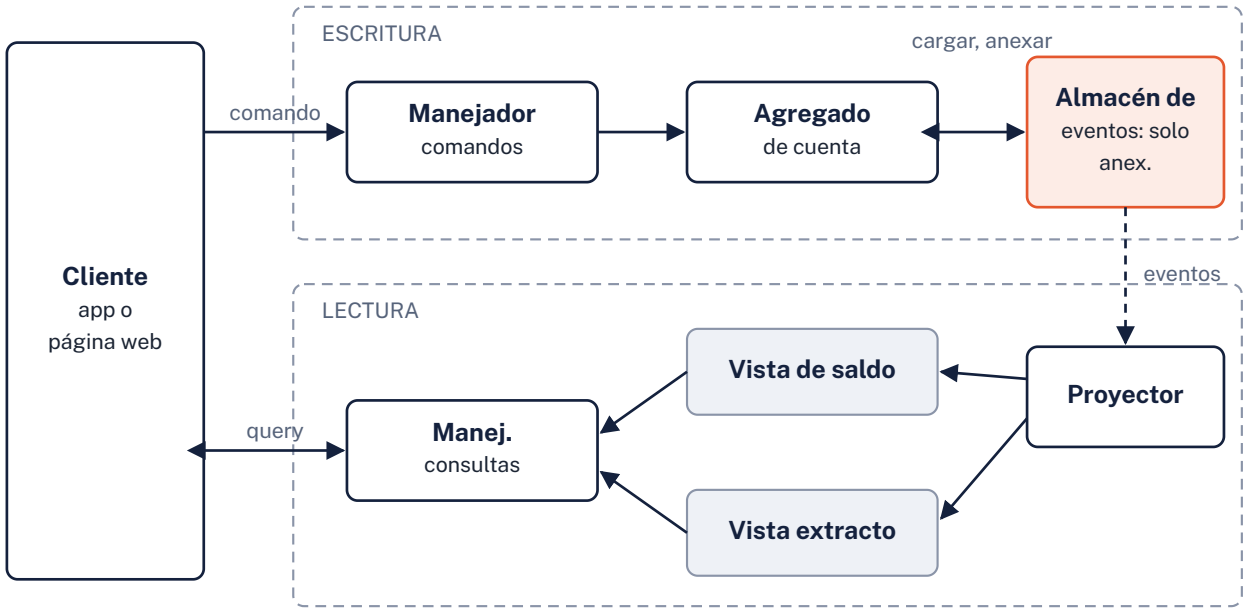
En el lado de escritura, un comando como `WithdrawMoney(account 42, £50)` llega a un manejador de comandos. El manejador carga el agregado correspondiente (el conjunto de datos que debe mantenerse consistente dentro de una transacción, un término del diseño guiado por el dominio que se trata en la Sección 5.7), le pide que aplique las reglas de negocio y guarda el resultado. Los comandos son solicitudes y pueden rechazarse: la cuenta puede estar bloqueada o los fondos pueden ser insuficientes. En el lado de lectura, una consulta como

“muéstrame los últimos 30 movimientos con el saldo acumulado” se responde desde un modelo de lectura construido exactamente para esa pantalla, sin joins, sin lógica de negocio y sin bloqueos sobre las tablas de escritura.

Algo tiene que mantener actualizados los modelos de lectura. Esa tarea corresponde a una proyección: código que escucha los cambios del lado de escritura y actualiza un modelo de lectura. Las proyecciones pueden mantener una tabla de saldos, un índice de búsqueda de movimientos, un documento de extracto mensual y un almacén de características (feature store) para puntuar el fraude, cada uno a partir del mismo flujo de cambios. Por tanto, los modelos de lectura son **datos derivados**: siempre pueden desecharse y reconstruirse a partir del lado de escritura. Esa propiedad es la que hace seguro añadir un nuevo modelo de lectura el año que viene, o cambiar la forma de uno antiguo.

CQRS admite grados, y el más ligero suele ser todo lo que un sistema necesita. En el grado más ligero, los modelos de escritura y de lectura son clases distintas sobre la misma base de datos, y las lecturas se sirven mediante vistas o código de consulta dedicado. En el grado intermedio, los modelos de lectura viven en tablas separadas que se actualizan en la misma transacción que la escritura. En el grado más pesado, los modelos de lectura viven en almacenes distintos (un motor de búsqueda, una base de datos documental, una caché) y se actualizan de forma asíncrona a partir de eventos, lo que aporta escala y flexibilidad, pero también consistencia eventual. CQRS no requiere event sourcing, un intermediario de mensajes (message broker) ni microservicios; son decisiones independientes que a menudo, aunque no necesariamente, se toman juntas.

CQRS con event sourcing



Los comandos y las consultas siguen caminos distintos; el almacén de eventos es la fuente de verdad y cada modelo de lectura se deriva de él mediante un proyector.

SI ESTO ES NUEVO PARA TI

Si esto es nuevo para ti

Piensa en el libro mayor de un banco y en un extracto bancario. El libro mayor es donde se anota cada depósito y cada retiro, línea a línea, siguiendo reglas estrictas. El extracto es un resumen preparado a partir del libro mayor en un formato fácil de leer. Nunca escribes en el extracto para mover dinero, y nunca lees el libro mayor línea a línea para averiguar tu saldo. CQRS aplica al software la misma división: un modelo para cambiar cosas y otros modelos para consultarlas. El event sourcing va un paso más allá y hace que el propio libro mayor sea lo único que se almacena de forma permanente.

Event sourcing: el estado como una secuencia de hechos

La mayoría de los sistemas almacenan el estado actual. Cuando Asha retira £120, la fila de `accounts` pasa de £500 a £380 y el valor antiguo desaparece, a menos que alguien escriba también una fila de auditoría. El event sourcing, descrito por Martin Fowler en 2005, le da la vuelta a esto: el sistema almacena cada cambio como un evento inmutable en un log de solo anexado, y el estado actual es lo que se obtiene al reproducir los eventos en orden. Los eventos son la fuente de verdad; el saldo es un cálculo.

Los eventos se nombran en pasado porque registran hechos que ya han ocurrido y se han aceptado: `AccountOpened`, `MoneyDeposited`, `MoneyWithdrawn`. Un comando (`WithdrawMoney`) es una solicitud que puede rechazarse; un evento (`MoneyWithdrawn`) ya no puede rechazarse, solo compensarse con un evento posterior. Cada agregado tiene su propio flujo de eventos, numerado a partir de la versión 1, y el almacén garantiza que los eventos de un flujo se conservan en orden.

Reconstruir el estado es un **plegado por la izquierda (left fold)**: se empieza sin nada, se aplica el primer evento, se aplica el segundo al resultado, y así sucesivamente. La función que aplica los eventos debe ser pura y determinista: no puede llamar a un reloj, a un generador de números aleatorios ni a otro servicio, porque reproducir los mismos eventos siempre debe dar el mismo estado. Cualquier valor que venga del exterior (un tipo de cambio, una marca de tiempo, una puntuación de fraude) se registra dentro del evento en el momento en que se decide por primera vez.

La escritura sigue un ciclo fijo. El manejador de comandos carga el flujo, lo pliega en el estado actual, ejecuta la regla de negocio y anexa los nuevos eventos con la versión que espera escribir. Si entretanto otro comando ha anexado eventos al mismo flujo, la anexión falla y el manejador vuelve a intentarlo con el estado actualizado. Es la concurrencia optimista de la Sección 7.3, aplicada por flujo, y es la razón por la que dos retiros simultáneos no pueden gastar ambos las mismas £100.

EJEMPLO PRÁCTICO

Ejemplo práctico: una cuenta bancaria como eventos

La cuenta corriente de Asha es el flujo `account-42`. Los importes se almacenan en peniques, como recomienda la Sección 0.2 para el dinero. Se han anexado cinco eventos; luego ella intenta retirar £200.

v1 AccountOpened (límite de descubierto £0)	Estado: saldo £0.00, versión 1
v2 MoneyDeposited £500.00	£0.00 + £500.00 = £500.00, versión 2
v3 MoneyWithdrawn £120.00	£500.00 − £120.00 = £380.00, versión 3
v4 MoneyDeposited £75.50	£380.00 + £75.50 = £455.50, versión 4
v5 MoneyWithdrawn £300.00	£455.50 − £300.00 = £155.50, versión 5
Comando: WithdrawMoney £200.00	Regla: el saldo menos el importe no debe quedar por debajo de menos el límite. £155.50 − £200.00 = −£44.50, por debajo de £0. Comando rechazado; no se anexa ningún evento.
Comando: WithdrawMoney £20.00	£155.50 − £20.00 = £135.50 está permitido. El manejador anexa v6 MoneyWithdrawn £20.00, esperando que la versión 6 esté libre.
Consulta temporal: saldo después de v3	Plegar solo de v1 a v3: £380.00. No hizo falta ninguna tabla de historial.
Pregunta de auditoría: ¿por qué el saldo es de £135.50?	Los seis eventos son la respuesta completa, en orden, con quién, cuándo y por qué registrados en cada uno.

El comando rechazado no dejó rastro en el flujo. Si el banco debe registrar los rechazos (para el análisis de fraude, por ejemplo), anexa un evento `WithdrawalRefused` aparte, que no cambia el saldo.

Un libro mayor de partida doble es el sistema con event sourcing más antiguo que existe: los contables nunca borran una línea, sino que registran un asiento de corrección. Por eso los dominios bancarios encajan de forma natural con el event sourcing, y por eso suele enseñarse con una cuenta.

```
# A bank account as a stream of events, folded into current state.
# Amounts are in pence so that no floating-point rounding creeps in.
from functools import reduce

events = [
    {"v": 1, "type": "AccountOpened",  "overdraft_limit": 0},
    {"v": 2, "type": "MoneyDeposited", "amount": 50000},
    {"v": 3, "type": "MoneyWithdrawn", "amount": 12000},
    {"v": 4, "type": "MoneyDeposited", "amount": 7550},
    {"v": 5, "type": "MoneyWithdrawn", "amount": 30000},
]

def apply(state, e):                      # pure: old state + event → new state
    if e["type"] == "AccountOpened":
        return {"balance": 0, "limit": e["overdraft_limit"], "version": e["v"]}
    sign = 1 if e["type"] == "MoneyDeposited" else -1
    return {**state, "balance": state["balance"] + sign * e["amount"], "version": e["v"]}

def decide_withdraw(state, amount):       # command side: check the rule, emit events
    if state["balance"] - amount < -state["limit"]:
        raise ValueError("insufficient funds")
    return [{"v": state["version"] + 1, "type": "MoneyWithdrawn", "amount": amount}]

now = reduce(apply, events, None)         # {'balance': 15550, 'limit': 0, 'version': 5}
as_at_v3 = reduce(apply, events[:3], None) # temporal query: balance 38000
```

Plegar el flujo en el estado actual, decidir un comando a partir de ese estado y responder a una consulta temporal plegando solo una parte del flujo.

```
-- An event store can be an ordinary table. The primary key enforces
-- optimistic concurrency: only one writer can append version 6.
CREATE TABLE account_events (
  account_id  uuid        NOT NULL,
  version     integer     NOT NULL,
  event_type  text        NOT NULL,
  data        jsonb       NOT NULL,
  recorded_at timestamptz NOT NULL DEFAULT now(),
  PRIMARY KEY (account_id, version)
);

-- The command handler read the stream up to version 5, decided, and now appends:
INSERT INTO account_events (account_id, version, event_type, data)
VALUES ('2f9c0d6e-4a51-4c7e-9a43-6d1f2b7e8a10', 6, 'MoneyWithdrawn', '{"amount": 2000}');
-- If a concurrent command already appended version 6, this fails with a
-- unique violation; the handler reloads the stream and decides again.
```

Un almacén de eventos mínimo en una base de datos relacional: la clave primaria compuesta convierte una anexión concurrente en un conflicto detectable.

Plegado, snapshots y consultas temporales



Consulta temporal: el saldo en v3 es el plegado de v1 a v3 solamente, que da £380.00.

El estado actual es el plegado de todos los eventos; un snapshot acorta el plegado, y plegar un prefijo del flujo responde a preguntas sobre el pasado.

Lo que te da el event sourcing

Una pista de auditoría completa por construcción. En un sistema basado en el estado, el log de auditoría es una segunda escritura que alguien puede olvidar, y puede no coincidir con los datos. En un sistema con event sourcing, el log son los datos, así que no puede desviarse de ellos. Los reguladores de la banca, los seguros y la sanidad lo valoran mucho.

Consultas temporales. "¿Cuál era el saldo a las 23:59 del 31 de marzo?" o "¿Qué pedidos estaban en estado PREPARING cuando falló la pantalla de la cocina?" se convierten en plegados sobre un prefijo de cada flujo, en lugar de en un trabajo forense sobre copias de seguridad.

Reconstrucciones y nuevos modelos de lectura. Como los modelos de lectura son derivados, un equipo puede corregir un bug en una proyección, borrar su salida y reproducir todos los eventos para producir una vista corregida. También puede construir, sobre todo el historial, un modelo de lectura en el que nadie pensó cuando se registraron los eventos, como un informe de la frecuencia con la que los clientes retiran efectivo durante la hora siguiente a cobrar.

Depuración y análisis hipotético (what-if). Copiar un flujo en un entorno de pruebas y reproducirlo recrea el estado exacto que causó un bug. Reproducir los mismos eventos con reglas modificadas muestra lo que habrían hecho las nuevas reglas.

Una fuente natural de eventos de integración. Los eventos pueden alimentar a otros servicios, aunque es mejor tratar los eventos almacenados en un flujo como un detalle privado del servicio propietario. Publicarlos directamente acopla a todos los consumidores al modelo interno; muchos equipos prefieren traducirlos a un conjunto más pequeño de eventos de integración públicos, diseñados como los contratos publicados de larga duración descritos en la Sección 6.4.

Snapshots y flujos largos

Plegar cinco eventos es instantáneo; plegar 400 000 eventos de una cuenta de comercio con mucho movimiento en cada comando no lo es. Un snapshot (instantánea) almacena el estado plegado de un flujo en una versión conocida, de modo que la carga pasa a ser "leer el último snapshot y aplicar solo los eventos posteriores". Los snapshots son una caché, no una fuente de verdad: pueden borrarse y recrearse en cualquier momento, en segundo plano, sin detener el sistema. Tómalos cada N eventos o según un calendario, y guarda junto a ellos la versión del código que aplica los eventos, porque un snapshot producido por código antiguo es incorrecto en cuanto cambia la lógica del plegado.

Antes de recurrir a los snapshots, revisa el diseño del flujo. Los flujos muy largos suelen indicar que el agregado es demasiado grande. Una cuenta bancaria que dura 40 años puede modelarse como una serie de flujos más cortos, uno por periodo de extracto, cada uno abierto con un evento `BalanceCarriedForward` que registra el saldo de cierre del periodo anterior. Esto refleja cómo cierran los libros los contables, y mantiene cada flujo lo bastante corto como para cargarse rápidamente.

Los costos, dichos con franqueza

Complejidad y falta de familiaridad. El event sourcing cambia la forma en que cada desarrollador del equipo piensa en la persistencia. Preguntas sencillas como "¿qué clientes viven en Leeds?" no pueden plantearse en absoluto al almacén de eventos; necesitan una proyección. Las herramientas para inspeccionar, migrar y reparar almacenes de eventos están menos maduras que las de las tablas relacionales, y menos ingenieros han operado un sistema así en producción.

Consistencia eventual de los modelos de lectura. Cuando las proyecciones se ejecutan de forma asíncrona, un usuario puede hacer un retiro y luego ver un saldo que todavía no lo incluye. Las mitigaciones habituales son devolver la nueva versión del flujo desde el comando y hacer que el lado de consulta espere hasta que su proyección haya procesado esa versión (una garantía de leer tus propias escrituras, como en la Sección 7.4); actualizar la interfaz de usuario de forma optimista a partir del resultado del comando; o actualizar los modelos de lectura

críticos de forma síncrona en la misma transacción que la anexión. Las proyecciones también deben ser idempotentes y registrar la posición que han procesado, porque los eventos pueden entregarse más de una vez tras una caída.

Versionado de eventos. Los eventos se almacenan para siempre, así que su esquema nunca puede cambiarse sin más. Cuando `MoneyDeposited` incorpora un campo de moneda, años de eventos antiguos siguen sin tenerlo. Las técnicas habituales, catalogadas por Greg Young en *Versioning in an Event Sourced System*, incluyen los lectores tolerantes, que aportan valores por defecto para los campos que faltan; el upcasting (una función que convierte una versión antigua de un evento a la forma actual en el momento de leerlo, de modo que el código del dominio solo ve una versión); la introducción de un nuevo tipo de evento junto al antiguo; y, como último recurso, la copia de un flujo entero en uno nuevo con los eventos transformados. La Sección 7.7 trata las reglas generales de compatibilidad.

Borrar datos personales. Un log inmutable choca con el derecho de supresión que establece la legislación de protección de datos. La técnica habitual es el crypto-shredding (destrucción criptográfica): los campos personales de los eventos se cifran con una clave guardada por cliente en un almacén de claves aparte, y la supresión consiste en destruir esa clave para que el texto cifrado ya no pueda leerse. La alternativa es mantener los datos personales totalmente fuera de los eventos y almacenar solo una referencia a un registro que pueda borrarse. Si destruir una clave cuenta o no como supresión es una valoración jurídica más que técnica, así que acuerda el enfoque con el delegado de protección de datos; la Sección 8.7 trata las obligaciones más amplias.

Carga operativa. Reconstruir una proyección sobre mil millones de eventos puede llevar horas, así que los equipos necesitan una forma de ejecutar la nueva proyección junto a la antigua y cambiar a ella cuando se ponga al día. Los almacenes de eventos crecen sin límite, y con ellos crecen el almacenamiento y los tiempos de copia de seguridad y de reproducción.

ATENCIÓN

La advertencia de Fowler

El artículo de Martin Fowler de 2011 sobre CQRS es inusualmente directo: escribe que “para la mayoría de los sistemas, CQRS añade una complejidad arriesgada”, y recomienda usarlo solo en partes concretas de un sistema, normalmente un contexto delimitado con necesidades de lectura y escritura realmente distintas, nunca como la arquitectura de todo el sistema. Lo mismo se aplica, con más motivo, al event sourcing. Los equipos que aplicaron ambos en todas partes porque estaban de moda han relatado años de complejidad accidental, sobre todo en áreas sencillas de crear, leer, actualizar y borrar, como los datos de referencia o los perfiles de usuario.

GUÍA DE DECISIÓN

Guía de decisión: ¿debería esta parte del sistema usar CQRS o event sourcing?

Hazte estas preguntas sobre un contexto delimitado, no sobre todo el sistema.

Q1 **¿Esta área consiste sobre todo en operaciones sencillas de crear, leer, actualizar y borrar, con pocas reglas de negocio?**

- **Sí** Usa un modelo convencional con una base de datos. Aquí ninguno de los dos patrones compensará su costo.
- **No** Continúa: el dominio tiene un comportamiento real que merece la pena modelar.

Q2 **¿Las lecturas y las escrituras difieren mucho en forma o en volumen (por ejemplo, muchas pantallas desnormalizadas, búsquedas, o mil lecturas por cada escritura)?**

- **Sí** Considera CQRS. Empieza por la forma más ligera: código de lectura separado o vistas sobre la misma base de datos.
- **No** Mantén un único modelo; añade una réplica de lectura o una caché si el único problema es la carga de lectura.

Q3 **¿Tiene valor el propio historial: auditoría regulatoria, consultas temporales, reproducción con la lógica corregida o muchos modelos de lectura futuros a partir de los mismos hechos?**

- **Sí** Considera el event sourcing para este contexto, idealmente un dominio parecido a un libro mayor, como las cuentas, los pagos o los ciclos de vida de los pedidos.
- **No** Almacena el estado actual; publica eventos con una bandeja de salida (outbox) si otros servicios necesitan enterarse de los cambios.

Q4 **¿Tiene el equipo la experiencia, las herramientas y el tiempo necesarios para gestionar el versionado de eventos, las reconstrucciones de proyecciones y la consistencia eventual?**

- **Sí** Adelante: mantén pequeña el área con event sourcing y registra la decisión y sus motivos en un ADR.
- **No** Almacena el estado actual más una tabla de auditoría de solo anexado, que aporta la mayor parte del beneficio de auditoría por una fracción del costo.

Cuándo no usar ninguno de los dos, y errores comunes

Ninguno de los dos patrones encaja en un dominio que consiste sobre todo en formularios sobre datos, en un equipo pequeño que construye su primera versión, ni en un sistema cuyo único problema es la carga de lectura (una réplica de lectura o una caché es mucho más

barata). El event sourcing también encaja mal donde los datos deben borrarse de verdad bajo demanda y en gran volumen, o donde la mayoría de las consultas abarcan muchos agregados, ya que cada una de esas consultas necesita su propia proyección.

Los errores comunes son previsibles. Los equipos aplican los patrones a todo el sistema en lugar de a un contexto. Exponen los eventos almacenados como contrato público, lo que congela su modelo interno. Ponen valores derivados o llamadas externas en la función que aplica los eventos, de modo que la reproducción da respuestas distintas. Diseñan los eventos como blobs genéricos `AccountUpdated`, que registran que algo cambió, pero no qué ocurrió ni por qué, y pierden la mayor parte del valor. Y olvidan que un modelo de lectura es una caché: empiezan a escribir en él directamente, y a partir de ese momento ya no puede reconstruirse.

Usada de forma selectiva, la combinación es potente. Una plataforma de reparto de comida podría aplicar event sourcing al ciclo de vida del pedido (realizado, aceptado, en preparación, recogido, entregado, reembolsado), porque las disputas y los reembolsos necesitan el historial exacto, y mantener las cartas, los perfiles de los restaurantes y la configuración de los usuarios como tablas ordinarias. Un banco podría aplicar event sourcing al libro mayor y a los pagos y almacenar de forma convencional los datos de contacto de los clientes, lo que además mantiene la mayor parte de los datos personales fuera del log inmutable.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Por qué la función que aplica los eventos al estado nunca debe llamar al reloj del sistema?

Porque reproducir los mismos eventos siempre debe producir el mismo estado. Un reloj (o un número aleatorio, o una llamada externa) haría que la reproducción diera respuestas distintas; cualquier valor así debe capturarse dentro del evento en el momento en que se decide por primera vez.

▼ ¿Cuál es la diferencia entre el comando WithdrawMoney y el evento MoneyWithdrawn?

El comando es una solicitud que las reglas de negocio pueden rechazar; el evento es un hecho registrado que ya se ha aceptado y que solo puede compensarse con un evento posterior, nunca eliminarse.

▼ Una proyección tiene un bug que ha producido extractos erróneos durante tres meses. ¿Cómo ayuda el event sourcing?

Como el modelo de lectura es derivado, el equipo corrige la proyección, construye un nuevo modelo de lectura reproduciendo todos los eventos (normalmente junto al antiguo) y cambia al nuevo, sin necesidad de reparar los datos a mano.

▼ **¿Forma parte un snapshot de la fuente de verdad?**

No. Es un plegado en caché de un flujo en una versión dada, que puede borrarse y recrearse a partir de los eventos en cualquier momento.

▼ **Inténtalo: El arquitecto quiere event sourcing en todas partes. Piensa en un libro mayor de puntos de fidelización que necesita un registro de auditoría completo, y en la tabla de horarios de apertura de los recintos. ¿Lo aplicarías a cada uno? Nombra dos costos que señalarías para el libro mayor.**

Úsalo para el libro mayor, si acaso: un historial de solo anexar da la pista de auditoría, y los saldos se pueden reconstruir reproduciendo los eventos. Los horarios de los recintos son un CRUD simple, de modo que basta con una tabla sencilla, y event sourcing solo añadiría riesgo. Para el libro mayor, señala el versionado de eventos, ya que los eventos antiguos permanecen para siempre, y el borrado de datos personales de un log inmutable, normalmente mediante crypto-shredding. Los modelos de lectura también serán eventualmente consistentes. Aplica el patrón por contexto, nunca por defecto.

FUENTES

- Martin Fowler: CQRS (2011): https://martinfowler.com/bliki/CQRS.html
- Martin Fowler: Event Sourcing (2005): https://martinfowler.com/eaaDev/EventSourcing.html
- Martin Fowler: Command Query Separation: https://martinfowler.com/bliki/CommandQuerySeparation.html
- Microsoft Azure Architecture Center: Event Sourcing pattern: https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing
- Greg Young: Versioning in an Event Sourced System (Leanpub): https://leanpub.com/esversioning

SECCIÓN 7.6 DE 7.8 · INTERMEDIO · 24 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Estrategias de caché y sus trampas

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.2 Datos y algoritmos
- 0.4 Comunicación en red
- 2.4 Rendimiento y escalabilidad

Términos anteriores que utiliza: Latencia, Rendimiento (throughput), HTTP, Sistema de registro (system of record), Leer tus propias escrituras (read-your-writes)

LA IDEA EN BREVE

Imagina una cafetería con mucho movimiento. En lugar de moler granos para cada pedido, el barista tiene sobre el mostrador un pequeño tarro de café recién molido. La mayoría de los pedidos se sirven desde el tarro en segundos; solo cuando se vacía va alguien al almacén. El tarro es una caché. Hace que el servicio sea rápido y le ahorra trabajo al almacén, pero plantea preguntas: ¿cuánto tiempo puede reposar el café molido antes de perder el sabor?, ¿qué pasa cuando el tarro se vacía en plena hora punta de la mañana?, ¿y quién se acuerda de tirarlo cuando cambian los granos? Las cachés de software plantean exactamente las mismas preguntas.

LO QUE NECESITAS SABER

- Una caché cambia una segunda copia de los datos, posiblemente desactualizada, por menor latencia, mayor rendimiento (throughput) y protección de la fuente; primero decide qué puede estar desactualizado, cuánto y para quién.
- Cache-aside es la opción segura por defecto; read-through traslada la carga a la capa de caché; write-through mantiene la caché al día a costa de escrituras más lentas; write-behind arriesga perder datos y nunca debe contener dinero; refresh-ahead conviene para unas pocas claves muy solicitadas.
- Combina la invalidación al producirse un cambio con un TTL como red de seguridad; en las escrituras, actualiza la base de datos y luego borra la entrada de la caché en lugar de sobrescribirla.
- Evita las estampidas con la agrupación de solicitudes, la caducidad con jitter, stale-while-revalidate, la actualización anticipada probabilística y el precalentamiento de la caché.

- La tasa de aciertos determina la carga de la base de datos mucho más que la latencia media: cada nueve adicional en la tasa de aciertos divide las consultas a la base de datos por unas diez, y una caché fría puede desbordar una base de datos dimensionada para el tráfico con caché.
- Hay cachés en los navegadores, las CDN, los proxies inversos, las aplicaciones y las bases de datos; HTTP las controla con Cache-Control, ETag y las solicitudes condicionales (RFC 9111, RFC 9110).
- LRU y LFU son las políticas de desalojo (eviction) habituales; las claves deben incluir todo lo que cambia la respuesta, y los datos personales o financieros deben quedar fuera de las cachés compartidas.

Por qué usar una caché y qué cuesta

Una caché guarda una copia de los datos en algún lugar más rápido o más cercano que su ubicación original, para que las solicitudes repetidas no paguen el precio completo cada vez. La Sección 0.2 presentó la idea como un intercambio de espacio por tiempo. A escala de arquitectura, el intercambio es más interesante, porque una caché cambia tres cualidades a la vez. Reduce la latencia (una búsqueda en Redis en el mismo centro de datos suele tardar mucho menos de un milisegundo; una consulta a la base de datos que une cinco tablas puede tardar decenas de milisegundos). Aumenta el rendimiento (throughput) y protege la base de datos, porque cada acierto (hit) es una consulta que la base de datos nunca ve. Y puede mejorar la disponibilidad, porque una caché capaz de servir datos algo antiguos sigue funcionando mientras la fuente está lenta o caída.

El precio se paga en corrección y en complejidad. Una caché es una segunda copia de los datos, y dos copias pueden no coincidir. Por tanto, toda caché encarna una decisión sobre cuán desactualizada puede estar una respuesta, y para quién. Una carta de restaurante con cinco minutos de antigüedad es aceptable; un saldo bancario con cinco minutos de antigüedad, mostrado a un cliente que decide si puede permitirse un pago, no lo es. Las cachés también añaden un componente que puede fallar, consumen memoria que cuesta dinero y crean nuevos modos de fallo, como las estampidas y los arranques en frío, que no existen sin ellas. Por eso la primera pregunta no es “¿qué caché?”, sino “¿qué puede estar desactualizado, cuánto y qué pasa si lo está?”

La eficacia de una caché se mide por su tasa de aciertos: los aciertos divididos entre todas las búsquedas. La tasa de aciertos depende de lo sesgado que sea el patrón de acceso. El tráfico de lectura de la mayoría de los sistemas de consumo está muy sesgado (una pequeña fracción de los restaurantes, productos o cuentas recibe la mayoría de las lecturas), así que una caché

que contiene un pequeño porcentaje de los datos a menudo puede responder a la mayoría de las lecturas. El acceso uniformemente aleatorio, en cambio, se aprovecha mal de la caché hagas lo que hagas.

Los cinco patrones clásicos

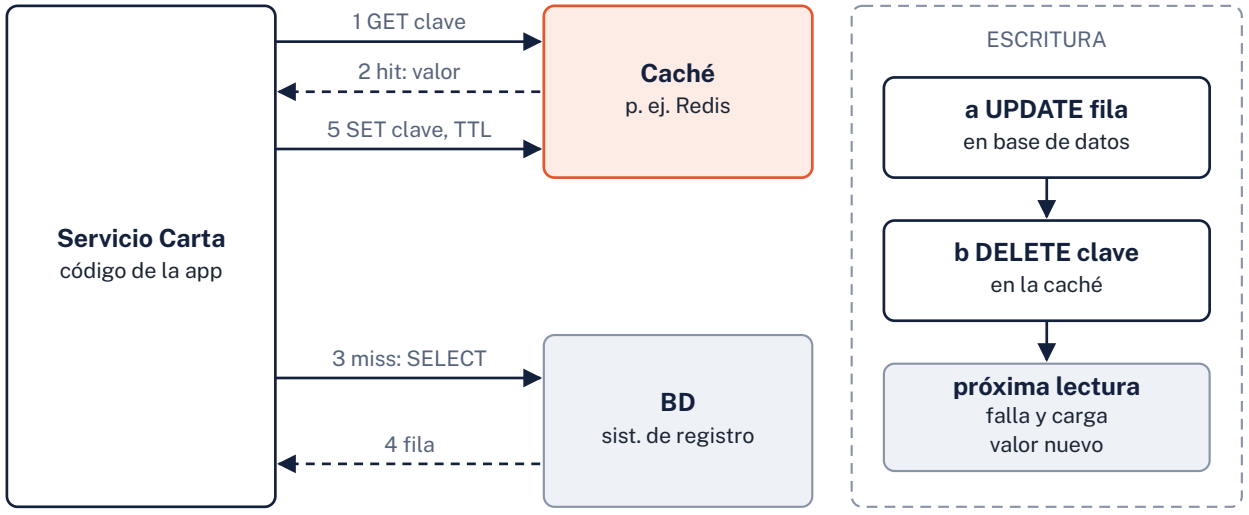
Los patrones de caché se diferencian en quién carga la caché, en cuándo llegan las escrituras a la fuente y, por tanto, en cuán desactualizados y cuán duraderos pueden ser los datos. La tabla los compara; los párrafos que la siguen explican dónde encaja cada uno.

Comparación de los patrones de caché

Patrón	Cómo funcionan las lecturas	Cómo funcionan las escrituras	Ventajas	Riesgos
Cache-aside (carga diferida)	La aplicación consulta la caché; en caso de fallo (miss), lee la base de datos y almacena el resultado	La aplicación escribe en la base de datos y luego borra (o actualiza) la entrada de la caché	Sencillo; solo almacena en caché lo que se usa; si la caché falla, todo se reduce a lecturas más lentas	La primera lectura tras la caducidad es lenta; las condiciones de carrera pueden dejar entradas desactualizadas; estampidas
Read-through	La aplicación pide los datos a la caché; la propia biblioteca de caché los carga desde la base de datos en caso de fallo	Suele combinarse con escrituras write-through o cache-aside	La lógica de carga está en un solo lugar	Los mismos problemas de desactualización y estampidas; la capa de caché debe conocer la fuente de datos
Write-through	Como en read-through o cache-aside	La escritura va a la caché y a la base de datos de forma síncrona antes de confirmarse	La caché nunca está más desactualizada que la base de datos para los datos escritos a través de ella	Cada escritura paga ambas latencias; almacena en caché datos que quizá nunca se lean
Write-behind (write-back)	Se sirven desde la caché	La escritura se confirma en cuanto está en la caché; se vuelca a la base de datos más tarde, a menudo por lotes	Escrituras muy rápidas; absorbe ráfagas; agrupa en lotes el trabajo de la base de datos	Pérdida de datos si la caché falla antes del volcado; la base de datos va con retraso; problemas de orden y conflictos
Refresh-ahead	Se sirven desde la caché	No afecta a las escrituras: las entradas muy solicitadas se recargan poco antes de caducar	Los elementos populares nunca caducan en las narices de los usuarios	Recargas desperdiciadas de elementos que no se habrían leído; necesita estadísticas de acceso

Cache-aside es la opción por defecto por una buena razón: es fácil razonar sobre él y falla de forma segura, porque si la caché desaparece la aplicación simplemente lee la base de datos (más despacio). Read-through traslada la misma lógica a una biblioteca o un producto de caché, lo que resulta ordenado, pero no cambia el comportamiento. Write-through conviene para los datos que se leen poco después de escribirse, como la sesión de un usuario o su carrito de la compra. Write-behind es el arriesgado: convierte la caché en el sistema de registro temporal, así que solo es apropiado cuando perder los últimos segundos de escrituras es aceptable (contadores de visitas, eventos de analítica, puntuaciones de juegos) o cuando la propia caché está replicada y es duradera. Nunca uses write-behind para el dinero. Refresh-ahead es una optimización para un pequeño conjunto de claves muy solicitadas, como la lista de restaurantes de la página de inicio en una ciudad con mucha actividad.

Cache-aside en la práctica



Las lecturas llenan la caché de forma diferida cuando hay un fallo; las escrituras cambian primero la base de datos y después borran la copia en caché en lugar de sobrescribirla.

```
import json, random

TTL = 300  # seconds

def get_menu(restaurant_id):
    key = f"menu:{restaurant_id}:v3"            # version the key with the data shape
    cached = cache.get(key)
    if cached is not None:
        return json.loads(cached)                # hit
    with cache.lock(f"lock:{key}", timeout=5):  # coalesce: one loader per key
        cached = cache.get(key)                  # someone may have filled it meanwhile
        if cached is not None:
            return json.loads(cached)
        menu = db.load_menu(restaurant_id)       # miss: go to the system of record
        jitter = random.randint(0, TTL // 10)    # spread expiry times out
        cache.set(key, json.dumps(menu), ex=TTL + jitter)
        return menu

def update_price(restaurant_id, item_id, price):
    db.update_price(item_id, price)              # 1. change the system of record
    cache.delete(f"menu:{restaurant_id}:v3")     # 2. invalidate, do not overwrite
```

Cache-aside con una clave versionada, un bloqueo por clave para agrupar los fallos concurrentes, una caducidad con jitter y la invalidación por borrado después de la escritura en la base de datos.

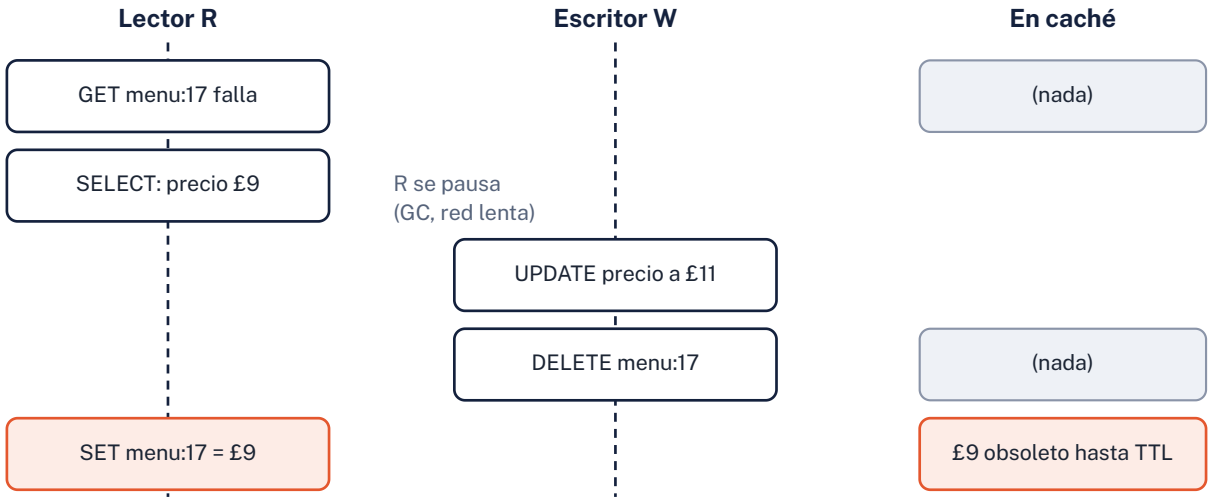
Tiempo de vida e invalidación

Hay dos formas de impedir que una caché sirva datos que han cambiado. La primera es un tiempo de vida (TTL): cada entrada caduca tras un periodo fijo, lo que acota la desactualización sin ninguna coordinación. Un TTL de cinco minutos en las cartas significa que un cambio de precio es visible en todas partes en un plazo de cinco minutos. La segunda es la invalidación explícita: cuando la fuente cambia, algo borra o actualiza la copia en caché. La invalidación da datos más frescos, pero exige que cada ruta del código que modifica los datos conozca todas las cachés que los contienen, que es justo lo que falla a medida que los sistemas crecen. En los sistemas distribuidos, la invalidación suele estar dirigida por eventos: el servicio de carta publica `MenuChanged` y cada servicio que tiene una copia borra su entrada. La mayoría de los diseños profesionales combinan ambas: invalidan al producirse un cambio para tener datos frescos y mantienen un TTL como red de seguridad para la invalidación que se haya escapado.

La dificultad queda recogida en una frase ingeniosa atribuida a Phil Karlton, de Netscape: “Solo hay dos cosas difíciles en informática: la invalidación de caché y ponerles nombre a las cosas.” Es difícil porque la invalidación es un problema de sistemas distribuidos disfrazado. La escritura en la base de datos y el borrado en la caché son dos operaciones en dos sistemas sin una transacción compartida, así que una puede tener éxito mientras la otra falla, y las operaciones de distintas solicitudes pueden intercalarse.

Dos reglas evitan la mayoría de los problemas. Primera: en una escritura, actualiza la base de datos y luego **borra** la entrada de la caché; no escribas el nuevo valor en la caché. Dos escritores concurrentes que actualizan cada uno la caché pueden terminar en el orden inverso al de sus commits en la base de datos y dejar el valor más antiguo en caché indefinidamente, mientras que un borrado solo puede causar un fallo adicional. Segunda: acepta que incluso borrar al escribir deja una pequeña condición de carrera, que se muestra a continuación, y acótala con un TTL. El artículo de Facebook de 2013 sobre el escalado de memcache describe los **leases** (concesiones) para este caso: la caché da un token al cliente que tuvo el fallo y rechaza la escritura posterior de ese cliente si la clave se borró entretanto.

La carrera del SET obsoleto



Borrar al escribir reduce la ventana de datos obsoletos, pero no la cierra; un TTL acota cuánto tiempo sobrevive el valor obsoleto.

SI ESTO ES NUEVO PARA TI

Paso a paso: por qué borrar en lugar de actualizar

La idea que requiere más atención aquí es la invalidación de caché: cambiar la base de datos y cambiar la caché son pasos separados en dos sistemas, así que las solicitudes que se solapan pueden dejar atrás un valor antiguo. Dos cambios rápidos de dirección muestran por qué hay que borrar, y no sobrescribir.

- La plataforma guarda en caché el perfil de cada cliente, con la dirección de entrega incluida, usando cache-aside: una solicitud que falla carga el perfil desde la base de datos y lo almacena. La caché dice Apartamento 1. En menos de un segundo, el cliente guarda Apartamento 2 y luego lo corrige a Apartamento 3: las solicitudes A y B.
- Supón que cada guardado hace commit en la base de datos y después copia su dirección sobre la que hay en caché. A hace commit de Apartamento 2, B hace commit de Apartamento 3: la base de datos es correcta.
- Pero A se detiene entre sus dos pasos. B pone Apartamento 3 en la caché; luego A, con retraso, pone Apartamento 2. Ahora la caché contradice a la base de datos, y todos los lectores ven el apartamento equivocado.
- Nada lo repara: no va a llegar ningún otro guardado, así que la dirección errónea permanece hasta que la entrada caduque, o indefinidamente si nunca caduca.
- Ahora haz que cada guardado elimine el perfil en caché. Lleguen en el orden que lleguen las eliminaciones, la caché acaba vacía; la siguiente solicitud falla y almacena Apartamento 3 desde la base de datos. Una eliminación tardía cuesta como mucho un fallo adicional.
- Eliminar reduce el hueco sin cerrarlo: un lector lento todavía puede almacenar un valor antiguo justo después de una eliminación, como muestra la carrera anterior. Un tiempo de vida en cada entrada limita cuánto tiempo sobrevive.
- **La lección:** ante un cambio, haz primero commit en la base de datos y después elimina la copia en caché en lugar de sustituirla, y da a cada entrada un tiempo de caducidad para atrapar lo que se le escape a la invalidación.

Estampidas y el thundering herd

Una cache stampede (estampida de caché) se produce cuando una entrada popular caduca, o un nodo de caché se reinicia vacío, y cientos de solicitudes concurrentes fallan en el mismo instante. Cada una va a la base de datos a calcular el mismo valor, y la base de datos, dimensionada para la tasa de fallos normal de la caché, se ve desbordada. Las respuestas más lentas hacen que se acumulen más solicitudes, y la sobrecarga puede propagarse en cascada. El fenómeno relacionado del **thundering herd** (manada en estampida) describe a muchos

clientes en espera que se despiertan a la vez para competir por un único recurso. Facebook informó de que, para un conjunto de claves especialmente propensas a esto, los leases redujeron la tasa máxima de consultas a la base de datos de 17 000 a 1300 por segundo.

Las defensas se combinan bien. La agrupación de solicitudes (request coalescing), también llamada single flight, permite que solo una solicitud por clave vaya a buscar el dato a la fuente mientras las demás esperan su resultado; dentro de un proceso es un mapa de futuros en curso, y entre procesos, un bloqueo breve en la caché, como en el código anterior. La **caducidad con jitter** añade una cantidad aleatoria a cada TTL para que las entradas escritas a la vez (tras un despliegue o un vaciado de la caché) no caduquen a la vez. **Stale-while-revalidate** sirve el valor caducado a todos mientras una solicitud en segundo plano obtiene uno nuevo, lo que elimina por completo el pico de latencia cuando una ligera desactualización es aceptable. La **caducidad anticipada probabilística** (Vattani, Chierichetti y Lowenstein, VLDB 2015) hace que cada lector decida al azar, con una probabilidad que aumenta a medida que se acerca la caducidad, si actualiza antes de tiempo, de modo que normalmente una solicitud actualiza la entrada antes de que caduque de verdad. Por último, el **precalentamiento** de una caché nueva a partir de una lista de claves muy solicitadas antes de que reciba tráfico evita la estampida del arranque en frío tras un reinicio.

EJEMPLO PRÁCTICO

Ejemplo práctico: lo que la tasa de aciertos hace con la latencia y la carga de la base de datos

El servicio de carta de la plataforma de reparto de comida recibe 20 000 lecturas de cartas por segundo en el pico de la noche. Una búsqueda en la caché tarda 0.5 ms; una consulta a la base de datos tarda 8 ms. En caso de fallo, el servicio paga ambas (8.5 ms) y almacena el resultado de forma asíncrona. La base de datos aguanta unas 3000 consultas de este tipo por segundo antes de que la latencia se dispare.

Fórmula	Latencia media = 0.5 + (1 − h) × 8 ms. Carga de la base de datos = (1 − h) × 20 000 consultas por segundo.
Sin ninguna caché	8 ms de media; 20 000 consultas por segundo, casi siete veces lo que la base de datos puede aguantar.
Tasa de aciertos del 80%	0.5 + 0.2 × 8 = 2.1 ms; 4000 consultas por segundo: todavía por encima de la capacidad, así que la base de datos encola y la latencia sube.
Tasa de aciertos del 95%	0.5 + 0.05 × 8 = 0.9 ms; 1000 consultas por segundo: holgado.
Tasa de aciertos del 99%	0.5 + 0.01 × 8 = 0.58 ms; 200 consultas por segundo.
Del 95% al 99%	La latencia media mejora solo 0.32 ms, pero la carga de la base de datos se divide por cinco, de 1000 a 200 consultas por segundo.
El clúster de caché se reinicia vacío	La tasa de aciertos cae casi al 0% durante un tiempo: 20 000 consultas por segundo llegan a una base de datos dimensionada para 3000. Sin agrupación de solicitudes, precalentamiento ni limitación de tasa (rate limiting), la base de datos se cae.
Percentiles	Con un 95% de aciertos, el 5% de las solicitudes tarda 8.5 ms, así que el p95 queda justo en el límite y el p99 es de 8.5 ms, no de 0.9 ms.

La tasa de aciertos importa mucho más para la base de datos que para la latencia media: cada nueve adicional de aciertos divide la carga de la base de datos por unas diez.

Una caché sin la que el sistema no puede funcionar ya no es una optimización; es una dependencia crítica, y hay que planificar su fallo como la Sección 2.5 planifica el de cualquier otra.

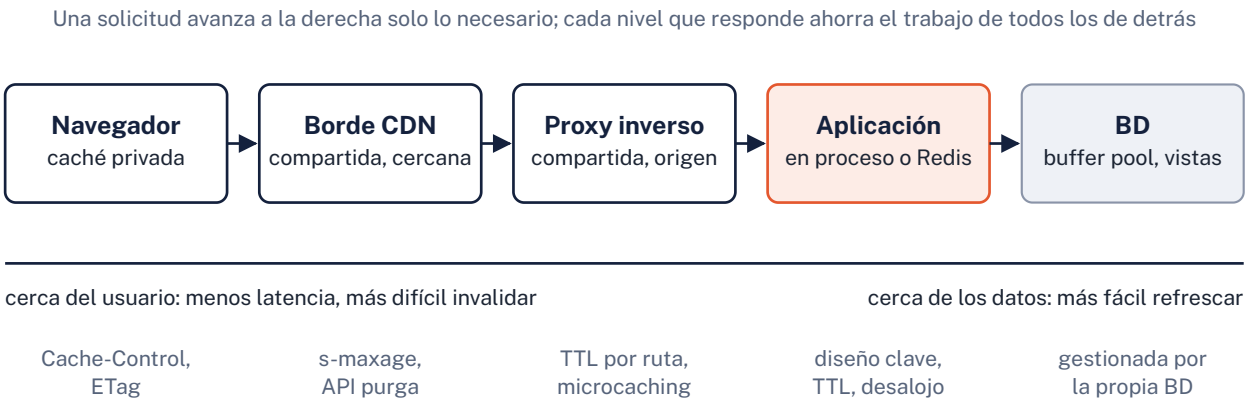
Las medias favorecen a las cachés; como mostró la Sección 2.4, los fallos definen la cola.

Niveles de caché

Hay cachés en todos los niveles entre el usuario y el disco, y una solicitud la responde el primer nivel que tiene una copia utilizable. El **navegador** mantiene una caché privada controlada por cabeceras HTTP. Una red de distribución de contenidos (CDN) mantiene copias compartidas en ubicaciones del borde (edge) cercanas a los usuarios, ideal para imágenes, scripts y respuestas de API públicas como las cartas. Un **proxy inverso** en el origen (Nginx, Varnish) puede almacenar en caché respuestas completas durante unos segundos, lo que se llama microcaching y puede absorber un pico repentino. La **aplicación** almacena en caché objetos calculados, ya sea en el propio proceso (lo más rápido, pero cada instancia tiene su propia copia y no coinciden entre sí) o en una caché compartida como Redis o Memcached. La **base de datos** almacena en caché páginas en memoria (los shared buffers de PostgreSQL, el buffer pool de InnoDB) y puede precalcular resultados costosos en vistas materializadas.

Cuanto más lejos de los datos, mayor es el ahorro por acierto y más difícil la invalidación. Puedes borrar una clave de Redis en un milisegundo; puedes purgar una CDN, más despacio; no puedes llegar en absoluto al navegador de un cliente, así que el `max-age` que envíes es una promesa con la que tendrás que vivir. Por eso las duraciones largas en el navegador se usan para archivos cuya URL cambia cuando cambia su contenido (`app.3f9a1c.js`), y las cortas, o la revalidación, para todo lo demás.

Dónde viven las cachés



Cada nivel que responde a una solicitud ahorra trabajo a todos los niveles que hay detrás; los niveles más cercanos al usuario ahorran más, pero son más difíciles de invalidar.

Cabeceras de caché HTTP

HTTP tiene un modelo de caché completo, definido en el RFC 9111 (junio de 2022), y usarlo bien puede eliminar carga antes de que llegue a tus servidores. `Cache-Control: max-age=60` permite que cualquier caché reutilice una respuesta durante 60 segundos; `s-maxage` lo sustituye para las cachés compartidas, como las CDN; `private` prohíbe que las cachés

compartidas la almacenen (úsalo para todo lo personal); `no-cache`, de forma confusa, significa “almacénala, pero consulta con el servidor antes de cada reutilización”; y `no-store` significa “no guardes nunca una copia”, el ajuste correcto para los saldos y los extractos bancarios. Las extensiones `stale-while-revalidate` y `stale-if-error` (RFC 5861, 2010) aplican a nivel HTTP las ideas sobre estampidas y disponibilidad vistas antes.

Cuando una copia en caché está desactualizada, el cliente no necesita descargarla de nuevo. Un validador le permite preguntar si ha cambiado algo: el servidor envía con la respuesta un `ETag` (un identificador de versión opaco) o una fecha `Last-Modified`, y más tarde el cliente envía `If-None-Match` o `If-Modified-Since`. Si nada ha cambiado, el servidor responde `304 Not Modified` sin cuerpo, lo que ahorra ancho de banda y, a menudo, la mayor parte del trabajo del servidor. Estas solicitudes condicionales se definen en el RFC 9110. La cabecera `Vary` indica a las cachés qué cabeceras de la solicitud (como `Accept-Language`) producen respuestas distintas, para que nunca se sirva una carta en francés a una solicitud en inglés.

```
GET /restaurants/17/menu HTTP/1.1
Host: api.example-food.test

HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: public, max-age=60, s-maxage=300, stale-while-revalidate=30
ETag: "menu-17-v42"

... 18 KB of menu JSON ...

# 100 seconds later the copy is past max-age plus the stale-while-revalidate
# window, so the browser must revalidate before using it:
GET /restaurants/17/menu HTTP/1.1
If-None-Match: "menu-17-v42"

HTTP/1.1 304 Not Modified
Cache-Control: public, max-age=60, s-maxage=300, stale-while-revalidate=30
ETag: "menu-17-v42"

# A bank balance, by contrast, must never sit in any cache:
Cache-Control: no-store
```

Una respuesta pública de carta que los navegadores pueden reutilizar durante 60 segundos y las CDN durante 300, revalidada de forma barata con un ETag; los datos financieros personales usan no-store.

Desalojo, dimensionamiento y riesgos de consistencia

Una caché es más pequeña que los datos que tiene detrás, así que, cuando se llena, debe desalojar algo. Menos recientemente usado (LRU) desaloja la entrada que lleva más tiempo sin tocarse y funciona bien cuando el uso reciente predice el uso futuro. Menos frecuentemente

usado (LFU) desaloja la entrada que se usa con menos frecuencia y resiste los recorridos puntuales (un informe nocturno que toca una vez cada restaurante vaciaría una caché LRU de sus entradas realmente muy solicitadas). Las cachés de producción se aproximan a ambas: Redis, por ejemplo, toma una muestra de unas pocas claves y desaloja la mejor candidata entre ellas, y ofrece políticas como `allkeys-lru`, `allkeys-lfu` y `volatile-ttl`. Su política por defecto, `noeviction`, rechaza las escrituras cuando la memoria está llena, lo que conviene a Redis usado como almacén, pero no como caché, así que configura la política de forma deliberada.

Más allá de la desactualización, las cachés introducen riesgos de consistencia más sutiles. Un usuario que cambia su dirección de entrega y luego ve la antigua, porque la lectura vino de una caché que otra instancia aún no había invalidado, sufre la ruptura de la garantía de leer sus propias escrituras (Sección 7.4); una solución habitual es saltarse la caché para las escrituras recientes del propio usuario. Las cachés en proceso de muchas instancias no coinciden entre sí, así que solicitudes consecutivas enviadas a instancias distintas pueden mostrar un valor que retrocede. Almacenar en caché un error o un resultado vacío sin un TTL corto convierte una interrupción breve en una larga. Almacenar en caché datos que mezclan usuarios (una página con el nombre del cliente, guardada en caché solo bajo la URL) filtra datos personales al siguiente visitante, una clase de incidente de seguridad bien conocida. Y las claves de caché deben incluir todo lo que cambia la respuesta: el inquilino (tenant), el idioma, la moneda, los permisos y la versión del formato de los datos.

ATENCIÓN

Errores comunes

- Añadir una caché antes de medir: a menudo la solución real es un índice o una consulta mejor, que elimina el problema sin una segunda copia de los datos.
- No poner ningún TTL y confiar por completo en una invalidación que una ruta de código olvidada se saltará.
- Actualizar la caché al escribir en lugar de borrar la entrada, lo que crea carreras de orden entre escritores.
- Dimensionar la base de datos para la carga con caché, de modo que un reinicio de la caché tumba toda la plataforma.
- Almacenar datos personales o financieros en cachés compartidas, u olvidar `private` y `Vary` en las respuestas HTTP.
- No vigilar en la monitorización la tasa de aciertos, la tasa de desalojo y la memoria, de modo que la caché deja de ayudar sin que nadie lo note.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Por qué una escritura debería borrar la entrada de la caché en lugar de poner el nuevo valor en la caché?

Porque dos escritores concurrentes pueden terminar sus actualizaciones de la caché en el orden inverso al de sus commits en la base de datos y dejar en caché el valor más antiguo; un borrado, en el peor caso, causa un fallo adicional que recarga el valor actual.

▼ Una búsqueda en la caché cuesta 1 ms y una lectura de la base de datos 10 ms. ¿Cuál es, aproximadamente, la latencia media con una tasa de aciertos del 90% con cache-aside?

1 + 0.1 × 10 = 2 ms (cada solicitud paga la búsqueda; una de cada diez paga además la lectura de la base de datos).

▼ ¿Cuál es la diferencia entre Cache-Control: no-cache y no-store?

no-cache permite que una caché almacene la respuesta, pero le exige revalidarla con el servidor antes de cada reutilización; no-store prohíbe guardar ninguna copia.

▼ ¿Por qué puede importar subir la tasa de aciertos del 99% al 99.9% aunque la latencia media apenas cambie?

Porque la carga de la base de datos es proporcional a la tasa de fallos, que baja del 1% al 0.1%: las consultas que llegan a la base de datos se dividen por diez.

▼ Inténtalo: Un servicio de carta recibe 12 000 lecturas por segundo. Una búsqueda en caché tarda 0.5 ms, una consulta a la base de datos 8 ms, y la base de datos aguanta 1000 consultas por segundo. ¿Qué tasa de aciertos necesitas? ¿Qué ocurre si la caché se reinicia vacía?

Los fallos de caché deben mantenerse en 1000 o menos de 12 000, así que la tasa de aciertos debe ser de al menos el 91.7%. Al 95%, la base de datos recibe 600 consultas por segundo y la latencia media es 0.5 + 0.05 x 8 = 0.9 ms. Una caché fría envía las 12 000 a la base de datos, doce veces su límite, lo que puede hacer que colapse. Calienta la caché antes de aceptar tráfico, agrupa las solicitudes de la misma clave y aplica jitter a los tiempos de caducidad.

FUENTES

- RFC 9111: HTTP Caching (IETF, 2022): https://www.rfc-editor.org/rfc/rfc9111.html
- RFC 5861: HTTP Cache-Control Extensions for Stale Content: https://www.rfc-editor.org/rfc/rfc5861.html

- MDN Web Docs: HTTP caching: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Caching
- Nishtala et al.: Scaling Memcache at Facebook (NSDI 2013): https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala
- Microsoft Azure Architecture Center: Cache-Aside pattern: https://learn.microsoft.com/en-us/azure/architecture/patterns/cache-aside

SECCIÓN 7.7 DE 7.8 · INTERMEDIO · 25 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Evolución de esquemas, migraciones y compatibilidad

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.5 Almacenar datos
- 0.8 Cómo se hace el software
- 6.5 Comunicación y API

Términos anteriores que utiliza: Cambio incompatible (breaking change), Protocol Buffers, Entrega continua, Retardo de replicación (replication lag), Idempotente

LA IDEA EN BREVE

Imagina un hospital que cambia el formulario en papel que se usa para cada ingreso mientras los pacientes siguen llegando. Si simplemente cambia los formularios de un día para otro, las enfermeras del turno de noche tendrán en la mano el formulario antiguo mientras el turno de día espera el nuevo, y los expedientes archivados seguirán usando el diseño antiguo. Un hospital cuidadoso imprime un formulario con las casillas antiguas y las nuevas, lo usa hasta que todos se han pasado al nuevo y los expedientes antiguos se han copiado, y solo entonces elimina las casillas antiguas. Los equipos de software cambian la estructura de sus datos exactamente de esta forma cuidadosa, paso a paso.

LO QUE NECESITAS SABER

- Los esquemas deben cambiar mientras el código antiguo y el nuevo se ejecutan en paralelo y siguen existiendo datos antiguos, por lo que cada estado de esquema desplegado debe funcionar con al menos dos versiones del código.
- La compatibilidad hacia atrás significa que los lectores nuevos pueden leer datos antiguos; la compatibilidad hacia adelante significa que los lectores antiguos pueden leer datos nuevos; la compatibilidad completa permite actualizar lectores y escritores en cualquier orden.
- Expandir y contraer (expand and contract, o cambio en paralelo) divide un cambio incompatible (breaking change) en pasos compatibles: añadir la nueva estructura, escribir en ambas (escritura dual) y hacer backfill, cambiar los lectores, dejar de escribir en la estructura antigua y, por último, eliminarla.

- Las migraciones son scripts versionados que se guardan con el código y que aplican automáticamente herramientas como Flyway, Liquibase y Alembic; nunca edites una migración ya aplicada y ensaya con datos de tamaño de producción.
- Los backfills grandes se ejecutan como trabajos separados, por lotes, con ritmo regulado e idempotentes; técnicas en línea como la creación concurrente de índices y las restricciones NOT VALID evitan bloqueos largos.
- Los registros de esquemas imponen reglas de compatibilidad a los esquemas de eventos; Avro se apoya en los valores por defecto y en la resolución entre el esquema del escritor y el del lector, y Protocol Buffers en no reutilizar nunca los números de campo.

Por qué los esquemas deben evolucionar sin detener el mundo

Todo dato almacenado o transmitido tiene un esquema: las tablas y los tipos de columna de una base de datos, los campos de una respuesta JSON de una API, la definición del registro de un evento en un flujo. Los esquemas cambian constantemente porque el negocio cambia: el banco añade un segundo titular de cuenta, la plataforma de reparto de comida empieza a registrar las propinas, un regulador exige un campo nuevo. La pregunta arquitectónica es cómo cambiar un esquema mientras el sistema sigue funcionando, cuando la forma antigua la siguen usando el código, los datos almacenados y otros equipos.

Tres hechos lo hacen difícil. Primero, en cualquier sistema que se despliega sin tiempo de inactividad, las versiones antigua y nueva del código se ejecutan en paralelo: un despliegue gradual (rolling deployment) sustituye las instancias una a una, un canary release (lanzamiento canario) envía una pequeña parte del tráfico a la nueva versión y un rollback trae de vuelta la versión antigua. Por tanto, cada estado del esquema debe funcionar con al menos dos versiones del código. Segundo, los datos sobreviven al código. Las filas escritas hace cinco años, los eventos de un log que se conservan para siempre, los mensajes que esperan en una cola y las copias de seguridad conservan todos la forma antigua. Tercero, los esquemas se comparten: una columna la leen informes, trabajos de extracción y a veces otros servicios, y un esquema de eventos lo leen consumidores que el productor quizá ni siquiera conozca. La Sección 7.1 defendía que los datos tengan un propietario y estén detrás de una interfaz; la evolución de esquemas es donde esa disciplina da sus frutos, porque un esquema privado puede cambiarse con el acuerdo de un solo equipo.

Compatibilidad hacia atrás, hacia adelante y completa

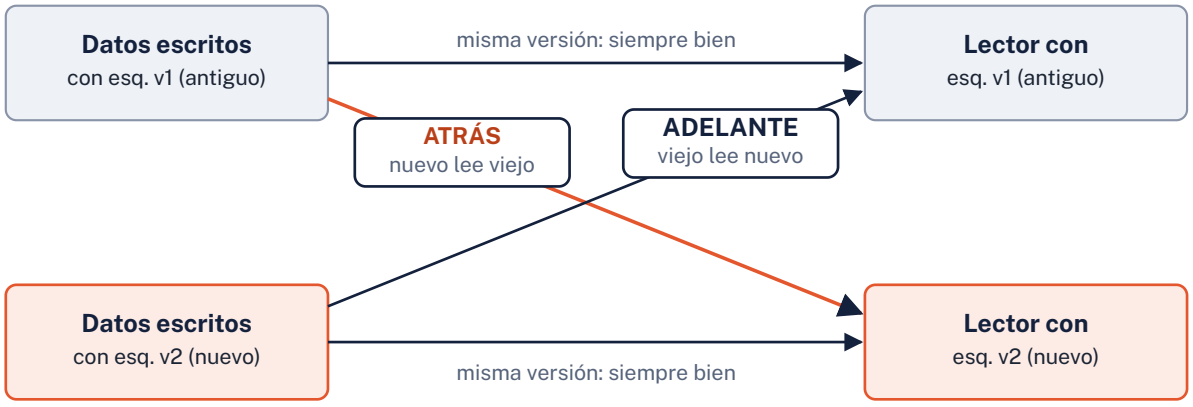
La compatibilidad se define en términos de lectores y escritores. Un cambio es compatible hacia atrás cuando el código que usa el esquema **nuevo** puede leer datos escritos con el **antiguo**. Añadir un campo opcional con un valor por defecto es compatible hacia atrás: el lector nuevo aporta el valor por defecto cuando a los datos antiguos les falta el campo. Un cambio es

compatible hacia adelante cuando el código que usa el esquema **antiguo** puede leer datos escritos con el **nuevo**. Eso exige que los lectores antiguos ignoren los campos que no conocen, que es la disciplina del lector tolerante (tolerant reader). Un cambio es **totalmente compatible** cuando es ambas cosas, lo que significa que lectores y escritores pueden actualizarse en cualquier orden.

La dirección que necesitas depende de quién se actualiza primero. Si los consumidores se actualizan antes que los productores (algo habitual cuando un flujo tiene pocos consumidores y los controlas tú), necesitas compatibilidad hacia atrás. Si los productores se actualizan primero, que es el caso habitual en una base de datos (el código nuevo empieza a escribir antes de que se hayan vuelto a desplegar todos los lectores) y en las API públicas, necesitas compatibilidad hacia adelante. Cuando no puedes controlar el orden, o necesitas revertir cualquiera de los dos lados, apunta a la compatibilidad completa. Confluent Schema Registry, por ejemplo, usa por defecto la compatibilidad hacia atrás comprobada contra la versión anterior y ofrece variantes transitivas que comprueban contra todas las versiones anteriores, lo que importa cuando los consumidores pueden reproducir datos antiguos desde el principio de un log.

Una advertencia sobre el vocabulario. En el diseño de API, “compatible hacia atrás” se usa a menudo de forma laxa con el sentido de “los clientes existentes siguen funcionando”, lo que, en términos de lectores y escritores, es compatibilidad hacia adelante para los clientes que leen tus respuestas y compatibilidad hacia atrás para las solicitudes que tú lees de ellos. Cuando una discusión depende del término, pregunta qué lado cambia y qué lado debe seguir funcionando.

Qué lector puede leer qué datos



COMPLETA = ambas diagonales funcionan: lectores y escritores, en cualquier orden

La compatibilidad hacia atrás protege a los lectores nuevos de los datos antiguos; la compatibilidad hacia adelante protege a los lectores antiguos de los datos nuevos.

Cambios habituales y su compatibilidad

Cambio	Hacia atrás (lo nuevo lee lo antiguo)	Hacia adelante (lo antiguo lee lo nuevo)	Enfoque seguro
Añadir un campo opcional o una columna que admite nulos con un valor por defecto	Sí	Sí, si los lectores ignoran los campos desconocidos	El cambio seguro por excelencia
Añadir un campo obligatorio sin valor por defecto	No: a los datos antiguos les falta	Sí	Añadirlo como opcional, hacer backfill y después imponerlo
Eliminar un campo	Sí	Solo si los lectores antiguos tienen un valor por defecto para él	Dejar de leerlo en todas partes primero y después eliminarlo
Renombrar un campo o una columna	No	No	Expandir y contraer: añadir el nuevo, copiar, cambiar, eliminar el antiguo
Cambiar un tipo (de texto a entero, de int a long)	Depende del formato: ampliar suele ser seguro, reducir no	Depende	Añadir un campo nuevo del tipo nuevo y migrar
Cambiar el significado sin cambiar la forma (de peniques a libras)	Erróneo sin avisar	Erróneo sin avisar	Nunca: crear un campo nuevo con un nombre nuevo

SI ESTO ES NUEVO PARA TI

Paso a paso: lectores antiguos, lectores nuevos

El punto en el que más gente tropieza aquí es la diferencia entre la compatibilidad hacia atrás y hacia adelante, y por qué la que necesitas depende de qué lado se actualiza primero. Sigue un campo nuevo a través de la plataforma de reparto de comida.

- La plataforma empieza a registrar las propinas, así que una nueva versión del evento OrderPlaced añade un campo, tip_pence. Los eventos ya escritos y almacenados no tienen ese campo.
- Supón que primero se actualiza un consumidor. Su código nuevo lee eventos escritos antes de que existieran las propinas, incluidos eventos antiguos reproducidos desde el principio del log, así que debe rellenar el campo que falta con un valor por defecto, aquí null, que significa que no se registró propina. Este caso, en el que la versión nueva se las arregla con datos antiguos, se llama compatibilidad hacia atrás.
- Supón, en cambio, que primero se actualiza el productor. Los consumidores que aún ejecutan código antiguo reciben ahora eventos con un campo del que nunca han oído hablar, y simplemente deben ignorarlo, como hace un lector tolerante. Este caso, en el que la versión antigua se las arregla con datos nuevos, se llama compatibilidad hacia adelante.
- Con lectores tolerantes, un campo opcional con un valor por defecto funciona en ambas direcciones, así que el cambio es totalmente compatible: cualquiera de los dos lados puede actualizarse primero y cualquiera puede revertirse.
- Compáralo con renombrar total_pence a amount_pence, para lectores que buscan los campos por su nombre. Los lectores nuevos buscan amount_pence y no lo encuentran en los eventos antiguos; los lectores antiguos buscan total_pence y no lo encuentran en los nuevos. El cambio se rompe en ambas direcciones a la vez.
- **La lección:** antes de cambiar un esquema, pregunta quién leerá datos nuevos con código antiguo y quién leerá datos antiguos con código nuevo. Los cambios aditivos con valores por defecto son seguros en ambos sentidos; todo lo demás necesita la técnica paso a paso que viene a continuación.

Expandir y contraer: la técnica central

La mayoría de los cambios incompatibles pueden dividirse en una secuencia de cambios compatibles. El patrón se llama expandir y contraer (expand and contract), o cambio en paralelo (parallel change; la descripción de Danilo Sato de 2014 en martinfowler.com llama a las fases expandir, migrar y contraer). Primero, **expandir**: añadir la nueva estructura junto a la antigua, de modo que funcionen tanto el código antiguo como el nuevo. Después, **migrar**: cambiar los escritores para que mantengan ambas, copiar los datos existentes y pasar los

lectores a la nueva estructura uno a uno. Por último, **contraer**: cuando ya nada usa la estructura antigua, eliminarla. Cada paso se despliega por separado, cada uno es compatible con la versión anterior y cada uno puede revertirse sin perder datos.

El costo es tiempo y un desorden temporal: el cambio abarca varios lanzamientos y, durante días o semanas, el esquema lleva las dos formas. La ventaja es que ningún lanzamiento necesita tiempo de inactividad, ningún despliegue necesita coordinarse entre equipos y cualquier paso puede pausarse. *Refactoring Databases* (2006), de Scott Ambler y Pramod Sadalage, cataloga decenas de refactorizaciones de bases de datos basadas en la misma idea, con un **periodo de transición** durante el cual coexisten los esquemas antiguo y nuevo, sincronizados por la aplicación o mediante triggers.

EJEMPLO PRÁCTICO

Ejemplo práctico: renombrar una columna sin tiempo de inactividad

La tabla de restaurantes de la plataforma de reparto de comida tiene una columna `postcode`. La plataforma se está expandiendo a países que usan otros formatos postales, y el equipo quiere que se llame `postal_code`. La tabla tiene 4 millones de filas, el servicio de restaurantes se despliega varias veces al día con actualizaciones graduales y un trabajo de extracción nocturno también lee la columna. Un único `ALTER TABLE ... RENAME COLUMN` rompería todas las instancias en ejecución del código antiguo en el momento en que se confirmara.

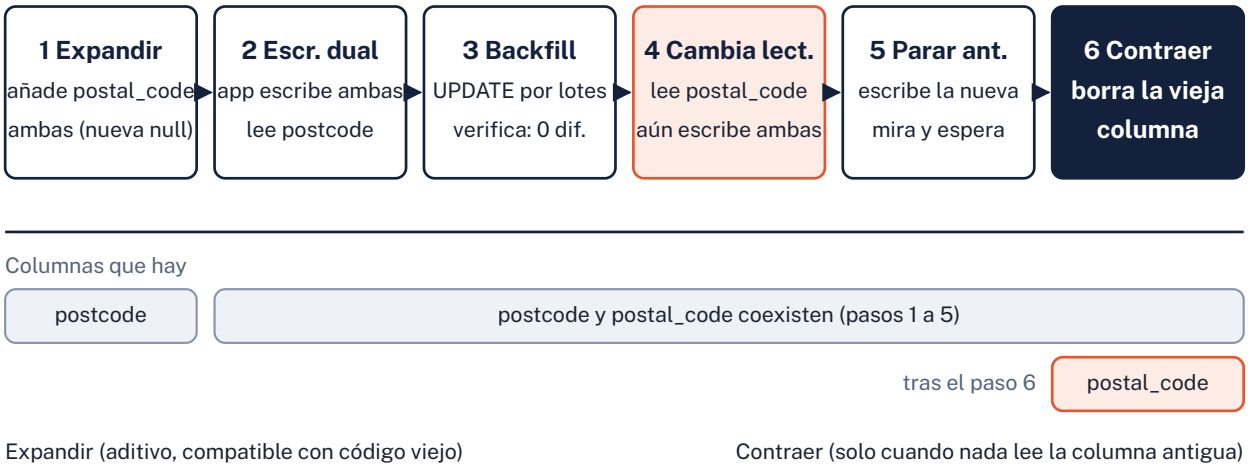
Lanzamiento 1: expandir	Una migración añade `postal_code text NULL`. En PostgreSQL 11 y posteriores, añadir una columna sin valor por defecto o con un valor por defecto constante es un cambio solo de metadatos. El código antiguo ignora la columna nueva; nada se rompe.
Lanzamiento 2: escritura dual	La aplicación escribe el mismo valor en ambas columnas en cada inserción y actualización, y sigue leyendo `postcode`. (Un trigger de base de datos que copie una columna en la otra es una alternativa cuando varias aplicaciones escriben en la tabla).
Paso 3: backfill	Un trabajo en segundo plano copia `postcode` en `postal_code` para las filas existentes en lotes de 5000 por rango de clave primaria, con pausas entre lotes y vigilando el retardo de replicación (replication lag). Después, una consulta de verificación cuenta las filas en las que las dos difieren: debe devolver 0.
Lanzamiento 4: cambiar las lecturas	La aplicación lee `postal_code` pero sigue escribiendo en ambas columnas, así que revertir al lanzamiento 2 o 3 sigue siendo seguro. El trabajo de extracción también se cambia para que lea la columna nueva.
Lanzamiento 5: dejar de escribir en la columna antigua	La aplicación escribe solo en `postal_code`. A estas alturas la columna antigua no debe ser `NOT NULL`, o las inserciones fallarán. Espera lo suficiente para tener la seguridad de que no hará falta revertir al lanzamiento 4.
Lanzamiento 6: contraer	Una migración impone `NOT NULL` en `postal_code` (mediante una restricción de comprobación `NOT VALID`, validada por separado, que PostgreSQL 12 y posteriores pueden usar para establecer `NOT NULL` sin un recorrido largo y bloqueante) y, por último, elimina `postcode`.

Seis despliegues para un solo cambio de nombre parecen excesivos, pero los pasos pueden combinarse cuando el equipo controla todos los lectores y acepta una ventana de rollback corta; el principio que importa es que cada estado desplegado funcione con el código anterior y el posterior.

El paso de contracción es el que los equipos olvidan. Haz un seguimiento explícito (un ticket, un estado de ADR, una regla de lint que señale la columna antigua) o el esquema se llenará de columnas migradas a medias.

El cambio de nombre en seis pasos desplegables

Un renombrado lógico en seis pasos desplegables; cada uno se puede revertir al anterior



Durante los pasos 1 a 5 existen ambas columnas, así que cualquier versión del código del paso anterior puede seguir ejecutándose.

```
-- Migration V41 (expand): metadata-only change, no table rewrite in PostgreSQL 11+
SET lock_timeout = '3s';            -- fail fast rather than queue behind long queries
ALTER TABLE restaurants ADD COLUMN postal_code text;

-- Backfill job (run repeatedly by a script, not inside one big migration)
UPDATE restaurants
SET    postal_code = postcode
WHERE  id ⩾ :from_id AND id < :from_id + 5000   -- one small batch per transaction
AND    postal_code IS DISTINCT FROM postcode;

-- Verify before switching reads
SELECT count(*) FROM restaurants WHERE postal_code IS DISTINCT FROM postcode;  -- expect
0

-- Migration V44: enforce NOT NULL without a long blocking scan
ALTER TABLE restaurants ADD CONSTRAINT postal_code_nn CHECK (postal_code IS NOT NULL) NOT
VALID;
ALTER TABLE restaurants VALIDATE CONSTRAINT postal_code_nn;  -- scans, but allows writes
ALTER TABLE restaurants ALTER COLUMN postal_code SET NOT NULL;  -- PostgreSQL 12+: uses
the check, skips scan
ALTER TABLE restaurants DROP CONSTRAINT postal_code_nn;

-- Migration V45 (contract), weeks later, once nothing reads postcode
ALTER TABLE restaurants DROP COLUMN postcode;
```

Sentencias de PostgreSQL para los pasos de expansión, backfill, verificación y contracción, escritas para evitar bloqueos largos.

Las migraciones como código en el pipeline de entrega

Los cambios en la base de datos deben estar en el control de versiones junto al código de la aplicación, como una serie ordenada de migraciones, cada una un pequeño script que lleva el esquema de una versión a la siguiente. *Evolutionary Database Design*, de Pramod Sadalage y Martin Fowler (publicado por primera vez en 2003 y reescrito en 2016), estableció la práctica, y las herramientas están maduras. **Flyway** ejecuta en orden archivos SQL versionados como `V41__add_postal_code.sql` y registra cada uno en una tabla `flyway_schema_history`. **Liquibase** describe los cambios como changesets en SQL, XML, YAML o JSON, de los que lleva registro en una tabla `DATABASECHANGELOG`. **Alembic**, para Python y SQLAlchemy, genera scripts de revisión con funciones `upgrade()` y `downgrade()` y registra la revisión actual en `alembic_version`. Flyway y Liquibase también guardan una suma de comprobación (checksum) de cada migración aplicada y se detienen con un error si alguna se ha editado después, lo que evita que los entornos diverjan sin que nadie lo note.

En un pipeline de entrega continua, las mismas migraciones se ejecutan contra una base de datos nueva en las pruebas, después contra staging y después contra producción, de modo que producción nunca recibe un cambio que no se haya ensayado. Las buenas prácticas añaden

algunas reglas. Nunca edites una migración que se haya aplicado en algún sitio; escribe una nueva. Mantén los cambios de esquema y el código que depende de ellos en despliegues separados, con el cambio de esquema primero en las expansiones y al final en las contracciones. Prueba las migraciones con una copia de datos de tamaño de producción, porque una sentencia que tarda 50 ms en una base de datos de pruebas puede bloquear una tabla grande durante una hora. Prefiere avanzar con una migración nueva antes que usar migraciones descendentes que eliminan columnas y, con ellas, datos. Y deja que la base de datos te avise de los bloqueos: en PostgreSQL, un `ALTER TABLE` que espera su bloqueo bloquea todas las consultas que llegan después, así que establece un `lock_timeout` corto y reintenta en lugar de quedarte en cola detrás de un informe de larga duración.

Backfills grandes y cambios de esquema en línea

Un backfill (relleno de datos) rellena una columna o tabla nueva para las filas existentes. Hecho como un único `UPDATE` sobre 400 millones de transacciones, mantiene bloqueos durante horas, genera un enorme volumen de registro de escritura anticipada (write-ahead log, WAL) que las réplicas deben reproducir (lo que provoca retardo de replicación) y, en PostgreSQL, deja versiones de fila muertas que el vacuum debe limpiar. El enfoque profesional es un trabajo separado y reiniciable que procesa lotes pequeños por rango de clave primaria, confirma cada lote, registra su progreso, reduce su ritmo cuando aumentan el retardo de replicación o la carga de la base de datos, y es idempotente, de modo que volver a ejecutar un lote no causa ningún daño. Ejecútalo fuera de la herramienta de migraciones, para que un despliegue nunca tenga que esperarlo.

Algunos cambios no pueden hacerse en el sitio sin bloquear. PostgreSQL ofrece `CREATE INDEX CONCURRENTLY` (que no puede ejecutarse dentro de un bloque de transacción, así que hay que indicárselo a las herramientas de migración) y la secuencia `NOT VALID` seguida de `VALIDATE CONSTRAINT` mostrada arriba. Para MySQL, herramientas como gh-ost y pt-online-schema-change construyen una copia en la sombra de la tabla con el esquema nuevo, la mantienen sincronizada con las escrituras en curso y la ponen en su lugar con un breve cambio de nombre. Cambiar el tipo de una columna suele hacerse mejor con expandir y contraer hacia una columna nueva que con un `ALTER` en el sitio, que en la mayoría de las bases de datos reescribe toda la tabla.

Registros de esquemas, Avro y Protocol Buffers

Para los eventos y los mensajes, el equivalente de una herramienta de migraciones es un registro de esquemas. Los productores registran cada versión del esquema; el registro la comprueba contra la regla de compatibilidad configurada y rechaza un cambio incompatible en

el momento de la compilación o del despliegue, en lugar de en producción. Cada mensaje lleva un identificador de esquema corto, de modo que los consumidores pueden obtener el esquema exacto con el que se escribió.

Apache Avro hace explícito el esquema del escritor: para decodificar se necesitan tanto el esquema del escritor como el del lector, y las reglas de resolución de Avro emparejan los campos por nombre, ignoran los campos del escritor que el lector no conoce y rellenan los campos del lector que faltan en los datos con sus valores por defecto declarados. De ahí la regla práctica: da un valor por defecto a todo campo que puedas añadir o eliminar. En el ejemplo de abajo, `currency` y `tip_pence` se añadieron en versiones posteriores con valores por defecto, así que los eventos antiguos se siguen decodificando con los consumidores nuevos.

Protocol Buffers identifica los campos por número, no por nombre, en la transmisión. Añadir campos es seguro, y los lectores antiguos conservan los campos desconocidos al volver a serializar. Renombrar un campo no afecta a la codificación binaria (aunque sí cambia la forma JSON). Las reglas fundamentales son no cambiar ni reutilizar nunca un número de campo, y marcar los números y nombres de los campos eliminados como `reserved` para que nadie los reutilice por accidente. Confluent recomienda la compatibilidad hacia atrás transitiva para Protobuf, porque añadir un tipo de mensaje nuevo no es compatible hacia adelante. También se puede registrar y comprobar JSON Schema, pero la apertura de JSON (¿se permite o no una propiedad desconocida?) hace que las reglas de compatibilidad sean más sutiles.

```
{
  "type": "record",
  "name": "OrderPlaced",
  "namespace": "food.orders.v1",
  "fields": [
    {"name": "order_id",       "type": "string"},
    {"name": "restaurant_id",  "type": "string"},
    {"name": "total_pence",    "type": "long"},
    {"name": "currency",       "type": "string", "default": "GBP"},
    {"name": "tip_pence",      "type": ["null", "long"], "default": null}
  ]
}
```

Un esquema Avro para un evento OrderPlaced; los campos añadidos después de la versión 1 llevan valores por defecto, así que los consumidores nuevos pueden leer los eventos antiguos.

Evolución de los esquemas de API y de eventos

Las API y los eventos siguen los mismos principios con herramientas distintas. En las API, prefiere los cambios aditivos, exige que los clientes sean lectores tolerantes y reserva los cambios incompatibles para una versión nueva que se ejecute junto a la antigua, con

monitorización del uso para ver cuándo puede retirarse la versión antigua (la Sección 6.5 trata los enfoques de versionado). Las pruebas de contratos dirigidos por el consumidor permiten que cada consumidor publique los campos de los que realmente depende, de modo que el proveedor sabe qué cambios son seguros.

Los eventos son más difíciles que las API porque se almacenan: un consumidor puede reproducir un topic desde el principio, y un sistema basado en event sourcing (Sección 7.5) reconstruye el estado a partir de eventos escritos hace años. Trata un esquema de eventos publicado como un contrato público, versiónalo en el registro y, cuando un cambio sea realmente incompatible, publica un tipo de evento o un topic nuevo (`OrderPlaced.v2`) durante un tiempo junto al antiguo, en lugar de modificar el existente. La Sección 6.4 trata el diseño de eventos de forma más general.

Lista de comprobación: antes de lanzar un cambio de esquema

- [ ] **¿Es aditivo el cambio?**
  Si no lo es, divídelo en pasos de expandir, migrar y contraer, cada uno desplegable por sí solo.
- [ ] **¿Funciona cada estado desplegado con las versiones de código anterior y siguiente?**
  Los despliegues graduales y los rollbacks implican que siempre se ejecutan dos versiones a la vez.
- [ ] **¿Quién más lee estos datos?**
  Busca informes, trabajos de extracción, otros servicios y consumidores de eventos; revisa los suscriptores del registro y el linaje en el catálogo de datos.
- [ ] **¿Se ha ejecutado la migración con datos de tamaño de producción?**
  Mide la duración de los bloqueos, las reescrituras de tablas y el volumen del log; usa la creación concurrente de índices y las restricciones NOT VALID donde estén disponibles.
- [ ] **¿Hay un tiempo de espera de bloqueo establecido?**
  Falla rápido y reintenta en lugar de dejar que un ALTER en espera bloquee todo el tráfico.
- [ ] **¿El backfill es separado, por lotes, con ritmo regulado, idempotente y reanudable?**
  Nunca debe bloquear un despliegue ni saturar las réplicas.
- [ ] **¿Tienen los campos nuevos valores por defecto y no se reutiliza ningún número ni nombre de campo?**
  Imprescindible para Avro y Protocol Buffers; lo impone la comprobación de compatibilidad del registro.
- [ ] **¿Ha cambiado el significado de algún campo?**
  Los cambios de unidades, de moneda o de semántica necesitan un nombre de campo nuevo, nunca uno reutilizado.
- [ ] **¿Se hace seguimiento del paso de contracción?**
  Registra un ticket o un ADR para que la estructura antigua se elimine de verdad cuando deje de usarse.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **Un equipo añade a un esquema Avro un campo obligatorio sin valor por defecto. ¿Qué compatibilidad rompe y por qué?**

La compatibilidad hacia atrás: un consumidor que usa el esquema nuevo para leer datos antiguos encuentra que falta el campo y no tiene ningún valor por defecto con el que rellenarlo.

▼ **¿Por qué continúa el paso de escritura dual después de que las lecturas pasen a la columna nueva?**

Para que, si se revierte al lanzamiento anterior, que todavía lee la columna antigua, esa columna esté al día.

▼ **¿Por qué un backfill grande no debe escribirse como una única sentencia UPDATE dentro de una migración?**

Una transacción enorme mantiene bloqueos durante mucho tiempo, genera una ráfaga de log que provoca retardo de replicación, deja mucho trabajo de limpieza, no puede reanudarse y bloquea el despliegue; los lotes pequeños confirmados evitan todo esto.

▼ **¿Qué evita marcar como reserved un número de campo de Protocol Buffers?**

Impide que alguien reutilice el número de un campo eliminado para un campo nuevo, lo que haría que los datos antiguos se decodificaran como el campo equivocado.

▼ **Inténtalo: Una tabla bookings tiene una columna, name, y necesitas first_name y last_name. Los servicios usan actualizaciones graduales y un trabajo nocturno lee name. Enumera los pasos para hacerlo sin tiempo de inactividad y di por qué una sola migración no es segura.**

Un cambio único rompería todas las instancias en ejecución del código antiguo. Usa expandir y contraer. Añade las dos nuevas columnas anulables. Despliega código que escriba tanto en las antiguas como en las nuevas. Haz un backfill de las filas antiguas en lotes pequeños, regulados e idempotentes. Cambia a los lectores, incluido el trabajo nocturno, a las columnas nuevas. Deja de escribir name y elimínala cuando ya nada la use. Cada paso funciona con dos versiones del código, y cualquiera se puede pausar con seguridad.

FUENTES

- Danilo Sato: ParallelChange (martinfowler.com, 2014): https://martinfowler.com/bliki/ParallelChange.html

- Pramod Sadalage and Martin Fowler: Evolutionary Database Design: https://martinfowler.com/articles/evodb.html
- Confluent documentation: Schema evolution and compatibility: https://docs.confluent.io/platform/current/schema-registry/fundamentals/schema-evolution.html
- Protocol Buffers: Language Guide (proto3), updating a message type: https://protobuf.dev/programming-guides/proto3/
- PostgreSQL documentation: ALTER TABLE: https://www.postgresql.org/docs/current/sql-altertable.html

SECCIÓN 7.8 DE 7.8 · AVANZADO · 27 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Analítica, streaming, almacenes de datos, lagos de datos y data mesh

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.5 Almacenar datos
- 6.4 Dirigida por eventos
- 7.1 Propiedad de los datos

Términos anteriores que utiliza: Contrato de datos, Esquema, Clave foránea, Join

LA IDEA EN BREVE

Las cajas registradoras de un supermercado están hechas para atender rápido a cada cliente, un carrito cada vez. A nadie se le ocurriría pedir a los cajeros que se detuvieran a calcular qué productos se vendieron más en cada tienda durante los dos últimos años. En cambio, cada noche las ventas del día se envían a la oficina central, donde los analistas guardan años de historial organizados para ese tipo de preguntas. Algunas cifras no pueden esperar a mañana, como una avalancha repentina de compras de pan, así que también se vigilan en directo. La arquitectura de analítica consiste en construir la oficina central y la vigilancia en directo sin ralentizar las cajas registradoras.

LO QUE NECESITAS SABER

- OLTP atiende muchas transacciones pequeñas por clave; OLAP recorre y agrega grandes volúmenes; separarlos protege los sistemas operacionales y permite que la analítica use diseños columnares y desnormalizados.
- Inmon construye primero un almacén de datos empresarial integrado y normalizado; Kimball construye esquemas en estrella dimensionales por proceso de negocio, enlazados por dimensiones conformadas; toda tabla de hechos necesita un grano (grain) declarado.
- ELT carga los datos en bruto y los transforma dentro del almacén de datos; la captura de datos de cambios (CDC) transmite los cambios desde los logs de la base de datos, pero los contratos de datos publicados son más robustos que leer las tablas de otro servicio.
- Un lakehouse añade transacciones, evolución de esquemas, borrados y viaje en el tiempo (time travel) a archivos abiertos en almacenamiento de objetos mediante formatos de tabla abiertos como Delta Lake, Apache Iceberg y Apache Hudi.

- El procesamiento de flujos debe calcular por tiempo del evento, cortar los flujos en ventanas fijas (tumbling), deslizantes (sliding) o de sesión, y usar marcas de agua (watermarks) junto con una política explícita para los datos tardíos.
- Lambda (Marz) combina una capa de lotes y una capa rápida a costa de duplicar la lógica; Kappa (Kreps, 2014) usa un solo procesador de flujos y reprocesa reproduciendo un log conservado.
- El data mesh (Dehghani, 2019) se basa en la propiedad por dominio, los datos como producto, una plataforma de autoservicio y la gobernanza computacional federada; encaja en organizaciones grandes con equipos de dominio maduros.

Dos cargas de trabajo distintas: OLTP y OLAP

Todo lo visto hasta ahora en este capítulo ha tratado del procesamiento de transacciones en línea (OLTP): muchas lecturas y escrituras pequeñas y rápidas de registros individuales, como hacer un pedido, registrar un pago o actualizar la posición de un repartidor. La analítica hace otro tipo de preguntas. "¿Cuáles fueron los ingresos por tipo de cocina y región, semana a semana, durante los dos últimos años?" afecta a millones de filas, lee solo unas pocas columnas de cada una y no cambia nada. Esto es el procesamiento analítico en línea (OLAP), y es tan distinto del OLTP que ejecutar ambos en la misma base de datos no suele servir bien a ninguno: una sola consulta analítica grande puede saturar los discos y la caché de búfer de los que depende el proceso de pago, y el esquema normalizado que mantiene correctas las transacciones obliga a los analistas a unir una docena de tablas.

La respuesta arquitectónica estándar es separar las cargas de trabajo. Los sistemas operacionales mantienen sus propias bases de datos, de las que son propietarios sus servicios, como se describió en la Sección 7.1. Los datos se copian, de forma continua o periódica, a una plataforma analítica diseñada para recorridos completos: almacenamiento columnar que lee solo las columnas necesarias, compresión intensa, motores de consulta masivamente paralelos y esquemas con la forma adecuada para el análisis. El resto de esta sección trata de cómo se diseña esa plataforma, cómo llegan a ella los datos, cómo cambia el panorama el streaming y quién es el propietario del resultado.

OLTP y OLAP comparados

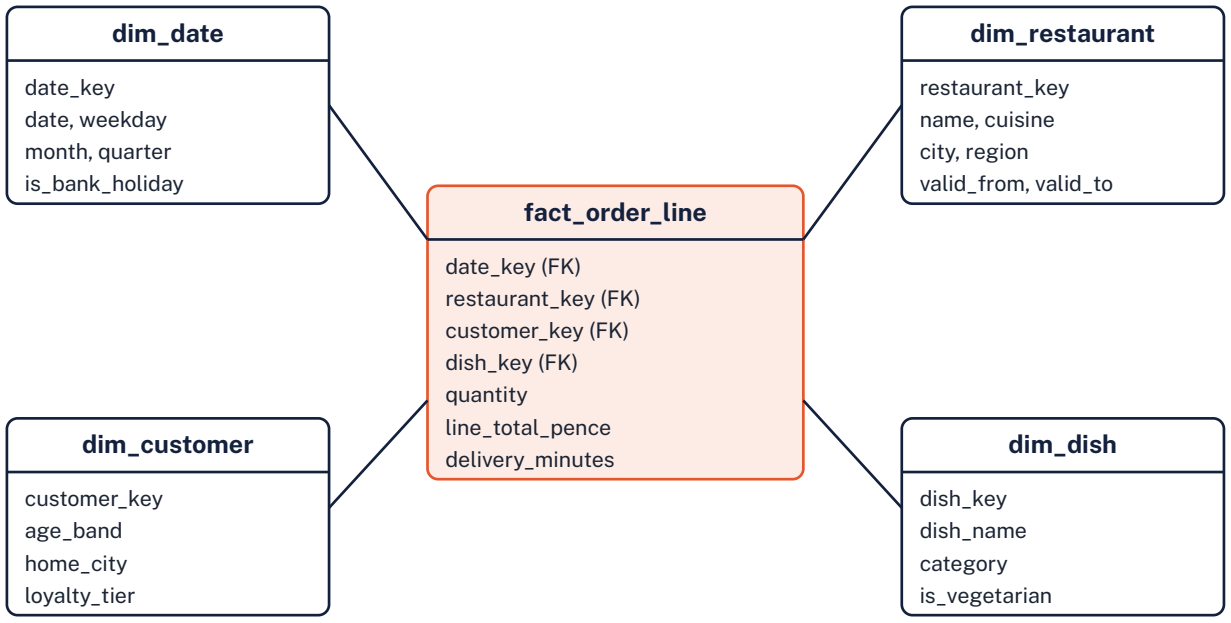
Aspecto	OLTP (operacional)	OLAP (analítico)
Pregunta típica	Obtener el pedido 81723; marcarlo como entregado	Tiempo medio de entrega por ciudad y por hora, últimos 12 meses
Filas afectadas por consulta	De una a unos cientos, por clave	De millones a miles de millones, por recorrido
Escrituras	Inserciones y actualizaciones pequeñas constantes	Cargas masivas o añadidos; pocas actualizaciones
Esquema	Normalizado por integridad (Sección 0.5)	Tablas desnormalizadas, dimensionales o anchas
Disposición del almacenamiento	Orientado a filas, indexado por clave	Orientado a columnas, comprimido, particionado por fecha
Público	Clientes y aplicaciones	Analistas, científicos de datos, paneles, pipelines de ML
Cualidades clave	Baja latencia, consistencia, disponibilidad	Rendimiento (throughput), flexibilidad de consulta, historial
Ejemplos	PostgreSQL, MySQL, DynamoDB	Snowflake, BigQuery, Redshift, ClickHouse, Databricks SQL

Almacenes de datos y modelado dimensional

Un almacén de datos (data warehouse) es una base de datos construida para el análisis que integra datos depurados de muchos sistemas operacionales y conserva su historial. Dos escuelas han marcado cómo se diseñan los almacenes de datos, y los profesionales aún discuten sobre ellas. **Bill Inmon**, a menudo llamado el padre del almacén de datos (*Building the Data Warehouse*, 1992), defiende un diseño descendente: primero se construye un almacén de datos normalizado, integrado y de alcance empresarial como fuente única de verdad, y después se derivan de él los data marts departamentales. **Ralph Kimball** (*The Data Warehouse Toolkit*, publicado por primera vez en 1996, tercera edición con Margy Ross en 2013) defiende un diseño ascendente: construir modelos dimensionales de un proceso de negocio cada vez e integrarlos mediante **dimensiones conformadas** compartidas (una única definición acordada de cliente, fecha o restaurante que usan todos los modelos). El enfoque de Inmon ofrece una integración más sólida con un costo inicial mayor; el de Kimball da resultados utilizables antes, pero exige disciplina para mantener las dimensiones conformadas. Muchas plataformas modernas combinan ambos: una capa depurada e integrada que alimenta modelos dimensionales para su consumo.

El esquema en estrella de Kimball sigue siendo el modelo analítico que más se enseña. Una tabla de hechos central registra eventos de negocio con un **grano** (grain) declarado (una fila por línea de pedido, por pago, por ping del repartidor), con medidas numéricas como la cantidad, el importe y los minutos, y claves foráneas a tablas de dimensiones. Cada dimensión contiene atributos descriptivos que se usan para filtrar y agrupar: fechas (con indicadores precalculados de día de la semana, mes y día festivo), restaurantes (nombre, tipo de cocina, ciudad), clientes y platos. Las consultas pasan a ser una tabla de hechos unida a unas pocas dimensiones pequeñas, algo que los motores columnares ejecutan con mucha eficiencia. Los atributos que cambian con el tiempo, como un restaurante que se traslada de ciudad, se gestionan con técnicas de **dimensión de cambio lento** (slowly changing dimension); la más común, el tipo 2, añade una fila de dimensión nueva con fechas de validez, de modo que los hechos históricos siguen apuntando a los atributos que eran ciertos en su momento.

Un esquema en estrella para los pedidos de reparto de comida



Grano: una fila por plato de un pedido entregado. Los hechos son numéricos y aditivos; las dimensiones los describen.

Una tabla de hechos con un grano declarado, rodeada de las dimensiones que describen cada hecho.

```
-- Revenue and average delivery time by cuisine and weekday for last quarter.
-- One join per dimension; the fact table is scanned, never updated.
SELECT r.cuisine,
       d.weekday,
       SUM(f.line_total_pence) / 100.0 AS revenue_gbp,
       AVG(f.delivery_minutes)          AS avg_delivery_min
FROM   fact_order_line f
JOIN   dim_date       d ON d.date_key       = f.date_key
JOIN   dim_restaurant r ON r.restaurant_key = f.restaurant_key
WHERE  d.quarter = '2026-Q2'
  AND  r.region  = 'North West'
GROUP  BY r.cuisine, d.weekday
ORDER  BY revenue_gbp DESC;
-- Caveat: delivery_minutes is per order, so averaging it over order LINES
-- weights big orders more. Grain decisions matter (see the worked example).
```

Una consulta típica sobre un esquema en estrella: filtrar y agrupar por atributos de las dimensiones y agregar las medidas de los hechos.

EJEMPLO PRÁCTICO

Ejemplo práctico: declarar el grano

El equipo de analítica construye la tabla de hechos anterior con el grano de una fila por plato de un pedido entregado. La medida a nivel de pedido `delivery_minutes` se copió en cada línea. Se entregan dos pedidos: el pedido A tiene 1 línea y tardó 20 minutos; el pedido B tiene 4 líneas y tardó 40 minutos.

Tiempo medio de entrega correcto	Por pedido: (20 + 40) ÷ 2 = 30 minutos.
Media sobre las filas de hechos	Líneas: 20, 40, 40, 40, 40. Suma 180 ÷ 5 filas = 36 minutos. Incorrecto: los pedidos grandes cuentan cuatro veces.
Suma de los minutos de entrega	180, que no tiene ningún significado: la medida no es aditiva con este grano.
Corrección	Mantén las medidas de nivel de línea (cantidad, total de la línea) en la tabla de hechos de líneas de pedido, y pon las medidas de nivel de pedido (minutos de entrega, tarifa de entrega, propina) en una tabla fact_order separada con el grano de una fila por pedido, que comparta las mismas dimensiones conformadas.
Comprobar	En fact_order: (20 + 40) ÷ 2 = 30 minutos, y SUM(delivery_fee) ahora cuenta cada tarifa una sola vez.

El primer paso de diseño de Kimball es declarar el grano con precisión, antes de elegir las dimensiones o los hechos, porque cada medida debe ser cierta con ese grano.

Mezclar granos en una misma tabla es una de las causas más comunes de que los paneles no coincidan entre sí.

Cómo entran los datos: ETL, ELT y captura de datos de cambios

Los datos llegan al almacén de datos mediante pipelines. En el ETL clásico (extraer, transformar, cargar), una herramienta independiente extrae los datos de los sistemas de origen, los depura y les da forma en sus propios servidores, y carga solo el resultado. Desde que los almacenes de datos en la nube abarataron el almacenamiento y volvieron elástico el cómputo, el **ELT** se ha vuelto habitual: extraer y cargar primero los datos en bruto y después transformarlos dentro del almacén con SQL, a menudo gestionados como modelos versionados y probados con una herramienta como dbt. El ELT conserva los datos en bruto, así que las transformaciones pueden corregirse y volver a ejecutarse; su riesgo es que los datos personales en bruto se extiendan más de lo necesario.

Cómo se extraen los datos importa igual. Las copias completas nocturnas son sencillas, pero lentas y desactualizadas. La **captura de datos de cambios (CDC)** lee el propio log de replicación de la base de datos (el registro de escritura anticipada de PostgreSQL, el binlog de MySQL) con una herramienta como Debezium y transmite cada inserción, actualización y borrado como un evento, con baja latencia y poca carga en el origen. Sea cual sea el mecanismo, leer directamente las tablas de otro servicio acopla la plataforma de analítica al esquema privado de ese servicio, exactamente el problema que describió la Sección 7.1. El patrón más robusto es que cada servicio publique los datos deliberadamente, mediante eventos o una exportación acordada, bajo un contrato de datos.

Data lakes, lakehouses y formatos de tabla abiertos

Un data lake (lago de datos) (el término se atribuye a James Dixon, 2010) almacena datos en bruto de muchas fuentes en su forma original, normalmente como archivos en un almacenamiento de objetos barato como Amazon S3, a menudo en formatos columnares como Apache Parquet. Aplaza el esquema hasta que se leen los datos, lo que conviene a los científicos de datos que quieren el detalle en bruto y a los datos cuyo uso aún no se conoce. Los lagos se ganaron la fama de convertirse en “pantanos de datos” (data swamps): sin propietarios, documentación ni comprobaciones de calidad, nadie se fía de lo que contienen. Los archivos en almacenamiento de objetos también carecen de lo que ofrece un almacén de datos: transacciones, actualizaciones y borrados, imposición del esquema y lecturas consistentes mientras se escriben los datos.

El lakehouse (Armbrust, Ghodsi, Xin y Zaharia, CIDR 2021) añade esas funciones de almacén de datos sobre archivos abiertos en el lago. La tecnología que lo hace posible es el formato de tabla abierto: **Delta Lake** (creado en Databricks), **Apache Iceberg** (creado en Netflix) y **Apache Hudi** (creado en Uber). Cada uno mantiene una capa de metadatos (un log de transacciones o un árbol de archivos de manifiesto) que registra qué archivos de datos forman cada versión de una tabla. Eso aporta confirmaciones atómicas, escritores concurrentes con concurrencia optimista, evolución de esquemas, actualizaciones y borrados a nivel de fila (necesarios para las solicitudes de supresión) y **viaje en el tiempo** (time travel): consultar una tabla tal como era en una versión anterior. Como el formato es abierto, varios motores (Spark, Trino, Flink y, cada vez más, los propios almacenes de datos en la nube) pueden leer y escribir las mismas tablas, lo que reduce la dependencia del proveedor (lock-in). Muchos lakehouses organizan los datos en capas, lo que Databricks popularizó como arquitectura medallón (medallion): bronce para los datos en bruto ingeridos, plata para los datos depurados y conformados, y oro para los modelos a nivel de negocio, como los esquemas en estrella.

Procesamiento de flujos: tiempo del evento, ventanas y marcas de agua

Los pipelines por lotes responden a preguntas sobre ayer. Algunas preguntas no pueden esperar: ¿se está usando esta tarjeta de forma fraudulenta ahora mismo?, ¿llegan tarde esta noche los repartidores de Manchester?, ¿qué restaurantes deberían pausarse porque sus cocinas están desbordadas? Los procesadores de flujos como Apache Flink, Kafka Streams y Spark Structured Streaming calculan de forma continua sobre flujos de eventos no acotados. Los conceptos esenciales proceden del modelo Dataflow de Google (Akidau y colegas, VLDB 2015).

El primero es la diferencia entre el tiempo del evento, cuando algo ocurrió realmente, y el **tiempo de procesamiento**, cuando el sistema lo observa. Difieren por los retrasos de red, los reintentos, el procesamiento por lotes y los dispositivos que se quedan sin conexión: el teléfono de un repartidor dentro de un ascensor registra una entrega a las 19:03, pero envía el evento a las 19:13. Cualquier resultado que deba reflejar la realidad (pedidos cada cinco minutos, tiempo medio de entrega por hora) debe calcularse por tiempo del evento, o se desplazará cada vez que el pipeline vaya lento y será incorrecto cuando se reproduzca.

El segundo son las **ventanas**, que cortan un flujo no acotado en trozos finitos para agregarlos: las ventanas **fijas** (tumbling) tienen un tamaño fijo y no se solapan (cada cinco minutos), las ventanas **deslizantes** (sliding) se solapan (los últimos diez minutos, actualizados cada minuto) y las ventanas **de sesión** agrupan ráfagas de actividad por clave separadas por un intervalo de inactividad (la sesión de navegación de un cliente).

El tercero es la marca de agua (watermark): la estimación móvil del procesador de que probablemente ya han llegado todos los eventos hasta un tiempo del evento dado. Cuando la marca de agua pasa el final de una ventana, puede emitirse el resultado de esa ventana. Las marcas de agua son heurísticas, así que algunos eventos llegan aun así después de que su ventana se haya cerrado. El pipeline necesita una política explícita para esos **datos tardíos**: descartarlos, enviarlos a una salida lateral (side output) para corregirlos más tarde, o mantener las ventanas abiertas durante un retraso permitido (allowed lateness) y emitir resultados actualizados. Esperar más da respuestas más completas; emitir antes da respuestas más frescas. Ese compromiso entre latencia y completitud es la versión del procesamiento de flujos de un compromiso entre atributos de calidad y debe decidirse con el negocio.

Tiempo del evento, ventanas y un evento tardío

Ventanas fijas de 5 minutos por tiempo del evento; cada punto, en su tiempo de evento



T. evento
cuando el teléfono del repartidor registró la entrega: 19:03

T. procesamiento
cuando lo vio el trabajo de streaming, al recuperar la señal: 19:13

La ventana de 19:00 ya se cerró cuando la marca de agua pasó de 19:05; el evento tardío necesita una regla: descartarlo, enviarlo a una salida lateral o actualizar la ventana si está en el retraso permitido.

Las ventanas se definen por tiempo del evento; la marca de agua decide cuándo se cierran y los eventos tardíos necesitan una política explícita.

SI ESTO ES NUEVO PARA TI

Paso a paso: contar según cuándo ocurrió

La idea más difícil de esta sección es que un trabajo de streaming debe agrupar los eventos según cuándo ocurrieron, no según cuándo llegaron, y después decidir cuánto esperar a los rezagados. Sigue un evento retrasado a lo largo de un recuento sencillo.

- Un trabajo de procesamiento de flujos (stream processing) cuenta los pedidos que acepta la tableta de cocina de cada restaurante en ventanas de cinco minutos, para detectar las cocinas desbordadas. Entre las 20:40 y las 20:45, un restaurante con mucho trabajo acepta 25 pedidos. Su tableta pierde el wifi justo después de aceptar uno a las 20:43, y no envía esa aceptación hasta las 20:57.
- Si se cuenta por tiempo de procesamiento, cuando el trabajo ve cada evento, la ventana de las 20:40 muestra 24 y la ventana de las 20:55 gana un pedido aceptado mucho antes. Al reproducirlo mañana, toda la noche se lee en pocos minutos y los recuentos vuelven a cambiar.
- Si se cuenta por tiempo del evento, la hora registrada en el propio evento, la aceptación retrasada va a la ventana de las 20:40, donde le corresponde, y una reproducción da siempre la misma respuesta.
- Eso plantea una pregunta nueva: ¿cuándo puede publicar el trabajo el recuento de las 20:40, si aún podría aparecer un rezagado? Usa una marca de agua, su estimación móvil de que probablemente ya han llegado todos los eventos hasta un momento dado. Cuando la marca de agua pasa de las 20:45, el trabajo publica 24.
- A las 20:57 llega la aceptación de las 20:43, después de que su ventana se haya cerrado. El pipeline necesita una regla decidida de antemano: descartarla, apartarla para una corrección posterior o permitir cierto retraso y publicar un recuento actualizado de 25.
- **La lección:** calcula por tiempo del evento para que los resultados reflejen lo que ocurrió de verdad, y acuerda con el negocio cuánto esperar. Esperar más da recuentos más completos; publicar antes da recuentos más frescos.

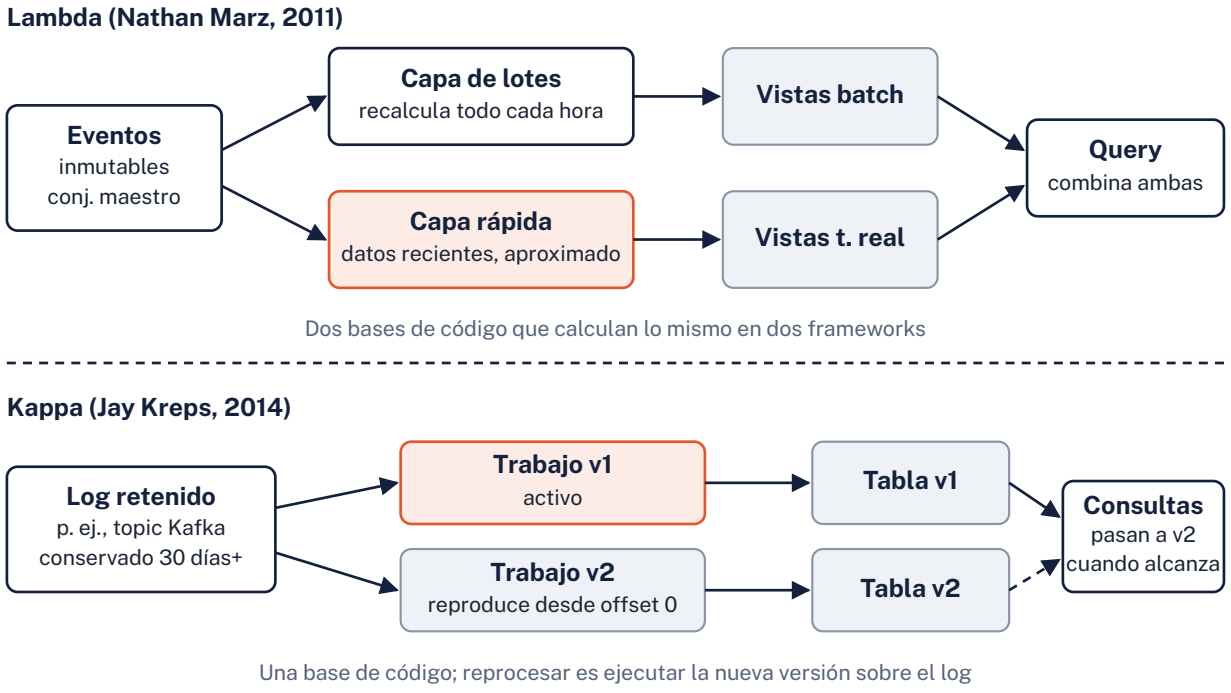
Las arquitecturas Lambda y Kappa

¿Cómo deben combinarse los resultados por lotes y los de streaming? El ensayo de Nathan Marz de 2011 “How to beat the CAP theorem” describió lo que pasó a conocerse como la **arquitectura Lambda**: un conjunto de datos maestro inmutable alimenta una **capa de lotes** (batch layer), que recalcula periódicamente vistas exactas a partir de todos los datos, y una **capa rápida** (speed layer), que calcula en tiempo real vistas aproximadas de los datos recientes; las consultas combinan ambas, y cada ejecución por lotes sustituye las

aproximaciones de la capa rápida. Lambda ofrece exactitud y baja latencia, y la capa de lotes actúa como red de seguridad ante los bugs del streaming. Su costo es que la misma lógica debe escribirse, y mantenerse idéntica, en dos frameworks distintos.

En 2014, Jay Kreps, uno de los creadores de Apache Kafka, cuestionó este diseño y propuso la **arquitectura Kappa**, un nombre que ofreció medio en broma. Usar solo un procesador de flujos y conservar la entrada en un log durante el tiempo suficiente para reproducirla. Para cambiar la lógica, se inicia una segunda instancia del trabajo nuevo que lee desde el principio del log y escribe en una tabla de salida nueva, y se pasan los lectores a ella cuando se ha puesto al día. Una sola base de código, y reprocesar es simplemente reproducir. Kappa depende de una retención larga del log y de procesadores de flujos lo bastante buenos como para gestionar correctamente el tiempo del evento y el estado, algo que ofrecen los motores modernos. Hoy, la mayoría de los diseños nuevos se inclinan por un streaming al estilo Kappa con un lakehouse para el historial, mientras que los diseños al estilo Lambda persisten donde recalcular por lotes historiales muy largos es más barato o donde se confía en pipelines por lotes existentes.

Lambda y Kappa, lado a lado



Dos bases de código que calculan lo mismo en dos frameworks

Una base de código; reprocesar es ejecutar la nueva versión sobre el log

Lambda mantiene rutas separadas de lotes y de velocidad y las combina en el momento de la consulta; Kappa tiene una sola ruta de streaming y reprocesa mediante la reproducción.

Data mesh: la propiedad de los datos analíticos

A medida que las organizaciones crecían, los equipos de datos centrales se convirtieron en cuellos de botella: cada fuente nueva y cada pregunta nueva hacían cola detrás de un equipo que no entendía muy bien ni los sistemas de origen ni las preguntas del negocio. En 2019,

Zhamak Dehghani, entonces en Thoughtworks, propuso el data mesh (ampliado después en su libro de 2022 *Data Mesh*). Aplica a los datos analíticos la misma descentralización que los microservicios aplicaron a los sistemas operacionales, y se basa en cuatro principios.

Propiedad por dominio: los equipos más cercanos a los datos, como los de pedidos, pagos o repartidores, son propietarios de sus datos analíticos y los sirven, igual que son propietarios de sus servicios. **Los datos como producto**: cada equipo publica productos de datos que se pueden descubrir y direccionar, y que están documentados, son fiables y son seguros, con un propietario y objetivos de calidad, y trata a los demás equipos como clientes. **Plataforma de datos de autoservicio**: un equipo de plataforma proporciona el almacenamiento, los pipelines, el catálogo y el control de acceso que abaratan la publicación de un producto de datos, para que cada dominio no tenga que reinventarlos. **Gobernanza computacional federada**: las reglas globales (identificadores, clasificaciones de privacidad, estándares de interoperabilidad) se acuerdan conjuntamente y la plataforma las aplica automáticamente, en lugar de un comité de revisión central.

El data mesh es tanto un diseño organizativo como una arquitectura, y las opiniones difieren sobre cuándo compensa. Sus defensores señalan la escala: elimina el cuello de botella central y pone la calidad de los datos donde está el conocimiento. Sus críticos señalan que necesita equipos de dominio maduros, una plataforma sólida y una inversión real, y que a las organizaciones más pequeñas suele servirles mejor una plataforma central bien gestionada con una propiedad clara de cada conjunto de datos. La ley de Conway (Sección 3.7) se aplica directamente: la arquitectura de datos reflejará la estructura que la organización adopte realmente.

Conservación, archivo y retención legal (legal hold)

Los datos necesitan una fecha de fin además de un propietario. El principio de limitación del plazo de conservación del RGPD del Reino Unido (UK GDPR) exige, por lo general, que los datos personales se conserven de forma identificable durante no más tiempo del necesario, por lo que cada clase de datos necesita un plazo de conservación acordado con los asesores jurídicos: los recorridos de ubicación de los repartidores pueden ser de corta duración, mientras que los registros fiscales duran años. El **archivo** traslada a un almacenamiento más barato los datos que deben conservarse pero que rara vez se leen, y necesita una forma de recuperarlos para su lectura. La eliminación debe alcanzar a las réplicas, los almacenes de datos, los logs de eventos y las copias de seguridad (Sección 8.7). La **retención legal** (legal hold) prevalece sobre el calendario: puede ser obligatorio preservar los registros pertinentes para un litigio, por lo que las tareas de eliminación deben comprobar primero si hay retenciones. Amazon S3 Object Lock, por ejemplo, ofrece tanto períodos de conservación como retenciones legales sin fecha de vencimiento. Modela las retenciones como datos, por cliente o pedido, en lugar de pausar una tarea. Los plazos dependen de la ley aplicable.

Calidad, linaje, catálogos y privacidad

Los datos analíticos solo son útiles si la gente confía en ellos. Las comprobaciones de **calidad de los datos** se ejecutan dentro de los pipelines y prueban la frescura (¿llegó la carga de hoy?), el volumen (¿se redujo a la mitad el número de filas?), el esquema (¿cambió de tipo una columna?), la validez (¿son positivos los importes y están bien formados los códigos postales?) y la unicidad. Las comprobaciones que fallan deben detener un pipeline o poner los datos en cuarentena, en lugar de publicarlos sin avisar. El linaje de datos registra de dónde procede cada conjunto de datos y qué transformaciones lo produjeron, de modo que, cuando una cifra parece errónea o cambia un esquema de origen, puedes rastrear el impacto aguas arriba y aguas abajo; la especificación OpenLineage estandariza cómo lo comunican los pipelines. Un **catálogo de datos** hace que los conjuntos de datos se puedan descubrir, con descripciones, propietarios, clasificaciones y linaje en un solo lugar.

Las plataformas de analítica concentran datos, lo que las hace atractivas para los atacantes y arriesgadas según la legislación de protección de datos. Aplica la minimización de datos (no copies campos personales que ningún análisis necesite), seudonimiza los identificadores siempre que sea posible, clasifica las columnas para que las políticas de acceso sigan a los datos, impón los periodos de conservación con borrado automatizado y asegúrate de que las solicitudes de supresión lleguen al lago y al almacén de datos además de a los sistemas operacionales; los formatos de tabla abiertos ayudan porque admiten borrados a nivel de fila. La Sección 8.7 trata las obligaciones en detalle.

ATENCIÓN

Errores comunes

- Ejecutar consultas analíticas pesadas contra la base de datos OLTP de producción, o contra una réplica de la que la aplicación también depende para el failover.
- Construir tablas de hechos sin declarar el grano, de modo que las medias y las sumas son erróneas sin que nadie lo note.
- Leer las tablas privadas de otros servicios directamente en el almacén de datos, de modo que cada cambio interno de esquema rompe un pipeline.
- Calcular agregados de streaming por tiempo de procesamiento, de modo que los resultados cambian cada vez que el pipeline se retrasa o se reproduce.
- Tratar un data lake como un vertedero sin propietarios, comprobaciones de calidad ni catálogo.
- Adoptar el vocabulario del data mesh sin financiar la plataforma de autoservicio ni dar a los dominios una propiedad real.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Por qué es mala idea ejecutar un informe de ingresos de dos años contra la base de datos del proceso de pago?

El informe recorre millones de filas y compite por disco, memoria y CPU con las transacciones pequeñas y sensibles a la latencia que necesita el proceso de pago, y el esquema normalizado hace que la consulta sea compleja y lenta.

▼ ¿Cuál es la primera decisión que pide el método de Kimball al diseñar una tabla de hechos?

Declarar el grano: exactamente qué representa una fila, como un plato de un pedido entregado.

▼ Un trabajo de streaming cuenta los pedidos por minuto por tiempo de procesamiento. ¿Qué ocurre cuando se queda diez minutos atrás y luego se pone al día?

Los pedidos se cuentan en los minutos en que se procesaron y no en los que ocurrieron, así que los recuentos son erróneos, con una caída seguida de un pico; calcular por tiempo del evento lo evita.

▼ ¿Cómo gestiona la arquitectura Kappa un cambio en la lógica de procesamiento?

Ejecuta la nueva versión del trabajo de streaming desde el principio del log conservado hacia una salida nueva, después pasa los lectores a ella cuando se ha puesto al día y retira el trabajo antiguo.

▼ Inténtalo: Una tabla de hechos tiene una fila por entrada, y cada fila copia el cargo de la reserva. La reserva A tiene 1 entrada y un cargo de £3; la reserva B tiene 3 entradas y un cargo de £5. Compara los resultados correctos y los ingenuos. ¿Qué cambiarías?

Lo correcto: comisiones totales de £8, media de £4 por reserva. Sumando las filas de hechos: 3 + 5 + 5 + 5 = £18; promediándolas: 18 / 4 = £4.50. La comisión pertenece al grano de reserva, no al de entrada. Declara el grano y luego, o bien mantén una tabla de hechos separada con una fila por reserva, o bien reparte la comisión entre las entradas, de modo que £5 pasen a ser unos £1.67 cada una, y etiquétalo claramente.

FUENTES

- Zhamak Dehghani: How to Move Beyond a Monolithic Data Lake to a Distributed Data Mesh (2019): https://martinfowler.com/articles/data-monolith-to-mesh.html
- Jay Kreps: Questioning the Lambda Architecture (O'Reilly Radar, 2014): https://www.oreilly.com/radar/questioning-the-lambda-architecture/

- Akidau et al.: The Dataflow Model (VLDB 2015): https://www.vldb.org/pvldb/vol8/p1792-Akidau.pdf
- Armbrust et al.: Lakehouse (CIDR 2021): https://www.cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf
- Kimball Group: Dimensional modeling techniques: https://www.kimballgroup.com/data-warehouse-business-intelligence-resources/kimball-techniques/dimensional-modeling-techniques/
- UK GDPR Article 5 (legislation.gov.uk): https://www.legislation.gov.uk/eur/2016/679/article/5
- Amazon S3 User Guide: Object Lock: https://docs.aws.amazon.com/AmazonS3/latest/userguide/object-lock.html

Lecturas recomendadas

- *Designing Data-Intensive Applications (2nd edition)*, Martin Kleppmann and Chris Riccomini (2026), O'Reilly Media. La referencia moderna de cabecera sobre replicación, particionamiento, transacciones, procesamiento de flujos y datos derivados; la mayoría de los mecanismos de este capítulo se explican allí en profundidad.
  https://martin.kleppmann.com/2026/03/24/designing-data-intensive-applications-2e.html
- *Software Architecture: The Hard Parts*, Neal Ford, Mark Richards, Pramod Sadalage and Zhamak Dehghani (2021), O'Reilly Media. Propiedad de los datos, técnicas de propiedad conjunta, patrones de acceso a datos distribuidos y desintegradores e integradores de datos, con un caso práctico recurrente.
- *A Critique of ANSI SQL Isolation Levels*, Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil and Patrick O'Neil (1995), Microsoft Research technical report MSR-TR-95-51 (also ACM SIGMOD 1995). El breve artículo que definió el aislamiento de instantánea y las anomalías que faltan en el estándar SQL.
  https://www.microsoft.com/en-us/research/publication/a-critique-of-ansi-sql-isolation-levels/
- *Dynamo: Amazon's Highly Available Key-value Store*, Giuseppe DeCandia and colleagues (2007), ACM Symposium on Operating Systems Principles (SOSP). El artículo en el que se basan la replicación sin líder, los quórums y el hashing consistente con nodos virtuales en Cassandra y sus parientes.
  https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
- *Choose Boring Technology*, Dan McKinley (2015), mcfunley.com. Un ensayo breve e influyente sobre los tokens de innovación y el costo a largo plazo de añadir tecnologías, en especial bases de datos.
  https://mcfunley.com/choose-boring-technology
- *The Data Warehouse Toolkit (3rd edition)*, Ralph Kimball and Margy Ross (2013), Wiley. La guía definitiva sobre el modelado dimensional, los esquemas en estrella y las tablas de hechos y de dimensiones para la analítica.
- *Data Mesh: Delivering Data-Driven Value at Scale*, Zhamak Dehghani (2022), O'Reilly Media. El relato, con la extensión de un libro, de los datos analíticos cuyo propietario es el dominio, los datos como producto, una plataforma de autoservicio y la gobernanza federada.

Glosario

Actualización perdida

Anomalía en la que dos transacciones leen un valor, cada una calcula uno nuevo y lo vuelve a escribir, de modo que un cambio sobrescribe al otro sin aviso.

Agrupación de solicitudes (request coalescing)
Dejar que una sola solicitud obtenga un elemento que falta mientras las demás solicitudes del mismo elemento esperan y comparten su resultado; también se llama single flight.

Aislamiento de instantánea (snapshot isolation)
Nivel de aislamiento en el que cada transacción lee una instantánea coherente de los datos confirmados desde su inicio, y se rechazan las escrituras en conflicto sobre la misma fila. Evita las actualizaciones perdidas, pero permite el write skew.

Almacén de datos (data warehouse)
Una base de datos construida para el análisis que integra datos depurados de muchos sistemas operativos y conserva su historial.

Anomalía (fenómeno)
Un resultado incorrecto causado por la interferencia entre transacciones concurrentes, como una lectura sucia, una actualización perdida o un sesgo de escritura (write skew); los niveles de aislamiento se definen según las anomalías que evitan.

Backfill (relleno de datos)
Rellenar una nueva columna o tabla con valores para las filas existentes, normalmente en lotes pequeños para no perturbar el sistema en funcionamiento.

Base de datos de integración
Una base de datos compartida por varias aplicaciones, que se integran a través de ella leyendo y escribiendo las mismas tablas. Martin Fowler (2004) la desaconseja porque acopla profundamente las aplicaciones.

Base de datos de series temporales
Base de datos optimizada para mediciones con marca de tiempo que llegan en orden, con compresión, reducción de resolución (downsampling) y eliminación barata de datos antiguos.

Base de datos vectorial
Almacén de embeddings (listas de números que representan significado) que encuentra los elementos más similares a una consulta, normalmente con índices de vecinos más cercanos aproximados.

Bloqueo pesimista
Prevenir los conflictos bloqueando las filas cuando se leen (por ejemplo, con SELECT … FOR UPDATE), de modo que otras transacciones esperan hasta que se libera el bloqueo.

Cache stampede (estampida de caché)
Una oleada de solicitudes que fallan en la caché en el mismo momento, normalmente cuando caduca un elemento popular, y llegan todas a la vez a la fuente más lenta; está estrechamente relacionada con el problema de la manada atronadora (thundering herd).

Caché write-behind
Patrón de caché en el que las escrituras se confirman en cuanto llegan a la caché y se escriben en el almacén subyacente más tarde, en segundo plano; también se llama write-back.

Caché write-through
Patrón de caché en el que cada escritura va a la caché y al almacén subyacente antes de confirmarse, de modo que la caché nunca es más antigua que el almacén.

Cache-aside
Un patrón de caché en el que la propia aplicación consulta la caché, carga desde la base de datos cuando hay un fallo de caché y coloca el resultado en la caché; también se denomina carga diferida (lazy loading).

Compatibilidad hacia adelante
Un cambio de esquema es compatible hacia adelante cuando el código que usa el esquema antiguo todavía puede leer datos escritos con el nuevo, normalmente ignorando los campos que no conoce.

Compatibilidad hacia atrás
Un cambio de esquema es compatible hacia atrás cuando el código que usa el nuevo esquema todavía puede leer datos escritos con el antiguo.

Contrato de datos
Un acuerdo explícito y versionado entre el propietario de unos datos y sus consumidores que abarca el esquema, el significado, la calidad, la actualidad, la propiedad y cómo se anuncian los cambios.

Control de concurrencia multiversión (MVCC)
Técnica en la que las escrituras crean nuevas versiones de las filas en lugar de sobrescribirlas, de modo que los lectores ven una instantánea coherente sin bloquear a los escritores.

Control de concurrencia optimista
Detectar los conflictos en lugar de prevenirlos: una escritura solo tiene éxito si un número de versión o una marca de tiempo no ha cambiado desde que se leyeron los datos; de lo contrario, quien escribe lo reintenta.

Crypto-shredding (destrucción criptográfica)
Hacer ilegibles los datos destruyendo la clave de cifrado que los protege; se usa para borrar datos personales de registros y copias de seguridad que no pueden editarse.

Data lake (lago de datos)
Un almacén grande y barato, normalmente almacenamiento de objetos, que guarda datos sin procesar de muchas fuentes en su forma original para analizarlos más adelante.

Data mesh
El enfoque de Zhamak Dehghani para los datos analíticos en el que los equipos de dominio son propietarios de sus datos y los publican como productos en una plataforma de autoservicio bajo una gobernanza federada.

Esquema en estrella
Modelo dimensional con una tabla de hechos central de mediciones rodeada de tablas de dimensiones que las describen, como fecha, cliente y producto.

ETag
Una cabecera de respuesta HTTP que contiene un identificador de una versión de un recurso, que un cliente puede devolver en If-None-Match para preguntar si ha cambiado.

Expandir y contraer (expand and contract)
Hacer un cambio incompatible en pasos seguros: añadir la nueva estructura junto a la antigua, migrar los lectores, los escritores y los datos, y luego eliminar la estructura antigua. También se llama cambio paralelo (parallel change).

Flujo de eventos
La secuencia ordenada de eventos de un agregado, numerada por versión, a partir de la cual se reconstruye el estado de ese agregado.

Formato de tabla abierto
Especificación, como Apache Iceberg, Delta Lake o Apache Hudi, que organiza los archivos de datos y los metadatos para que muchos motores puedan tratarlos como tablas con transacciones y viaje en el tiempo.

Hashing consistente
Una forma de asignar claves a nodos en un anillo de hash de modo que añadir o quitar un nodo mueva solo alrededor de 1/N de las claves (Karger y colaboradores, 1997).

Índice secundario
Índice sobre un atributo distinto de la clave primaria o de partición. En un almacén particionado es local a cada partición o global y particionado por el valor indexado.

Invalidación de caché
Eliminar o marcar como obsoleta una copia almacenada en caché cuando cambian los datos originales, para que las lecturas posteriores obtengan el nuevo valor.

Lakehouse
Arquitectura que añade funciones de almacén de datos, como transacciones, esquemas y consultas rápidas, a datos guardados en formatos de archivo abiertos sobre almacenamiento de objetos.

Lector tolerante (tolerant reader)
Consumidor que lee solo los campos que necesita e ignora todo lo que no reconoce, de modo que los proveedores pueden añadir campos sin romperlo.

Linaje de datos
Un registro de dónde procede un dato y de cada transformación por la que ha pasado en el camino.

Marca de agua (watermark)
La estimación móvil de un procesador de flujos de que probablemente han llegado todos los eventos hasta un determinado tiempo de evento, que se usa para decidir cuándo puede emitirse el resultado de una ventana.

Menos frecuentemente usado (LFU)
Política de desalojo que elimina el elemento que se ha leído menos veces cuando la caché está llena.

Menos recientemente usado (LRU)
Política de desalojo que elimina el elemento que lleva más tiempo sin leerse cuando la caché está llena.

Migración
Script versionado que cambia el esquema de una base de datos o sus datos de una versión a la siguiente, guardado en el control de versiones y aplicado automáticamente.

Motor de búsqueda
Almacén construido en torno a un índice invertido de palabras a documentos, que ofrece búsqueda de texto completo ordenada por relevancia y tolerante a erratas; normalmente se alimenta desde un sistema de registro.

Nivel de aislamiento
Un ajuste que indica cuánto pueden afectarse entre sí las transacciones concurrentes: read uncommitted, read committed, repeatable read, snapshot o serializable. Los nombres significan cosas distintas en bases de datos distintas.

OLAP
Online analytical processing (procesamiento analítico en línea): cargas de trabajo de consultas grandes que recorren y agregan muchos registros, como los ingresos por región y por mes.

OLTP
Online transaction processing (procesamiento de transacciones en línea): cargas de trabajo de muchas lecturas y escrituras pequeñas de registros individuales, como hacer pedidos o registrar pagos.

Particionamiento (sharding)
Dividir un conjunto de datos de modo que cada máquina guarde solo una parte, elegida mediante una clave de partición, para escalar las escrituras y el almacenamiento más allá de una sola máquina.

Persistencia políglota
Usar distintos tipos de base de datos para distintos datos dentro de un mismo sistema, elegidos según cómo se usa cada conjunto de datos, a costa de tener más tecnologías que operar.

Producto de datos
Un conjunto de datos que un equipo de dominio publica para que otros lo usen, con un propietario, documentación, garantías de calidad y una interfaz estable.

Propiedad conjunta
Situación en la que varios servicios escriben todos los mismos datos. Puede resolverse dividiendo la tabla, con un pequeño dominio de datos compartido, delegando las escrituras en un único propietario o fusionando los servicios.

Proyección
Código que escucha cambios o eventos y mantiene actualizado un modelo de lectura; también se usa para designar el modelo de lectura que produce.

Registro de escritura anticipada (write-ahead log, WAL)
Log de solo anexado en disco en el que una base de datos registra cada cambio antes de aplicarlo, para que el trabajo confirmado sobreviva a una caída; las réplicas suelen alimentarse de él.

Registro de esquemas
Servicio que almacena todas las versiones de cada esquema de mensaje y rechaza las versiones nuevas que rompen la regla de compatibilidad acordada.

Replicación
Mantener copias de los mismos datos en varias máquinas para lograr disponibilidad, capacidad de lectura y menor latencia cerca de los usuarios.

Replicación líder-seguidor
Esquema de replicación en el que un nodo (el líder o primario) acepta todas las escrituras y transmite sus cambios a los seguidores (réplicas), que pueden atender lecturas.

Replicación multilíder
Esquema en el que varios nodos aceptan escrituras y se replican entre sí, lo que exige una forma de detectar y resolver escrituras en conflicto.

Replicación sin líder
Esquema al estilo de Dynamo en el que los clientes escriben y leen directamente en varias réplicas y usan quórums en lugar de un único líder.

Retardo de replicación (replication lag)
El retardo entre una escritura en el líder y su aparición en un seguidor; hace que los usuarios lean datos desactualizados.

Separación comando-consulta (CQS)
Principio de Bertrand Meyer según el cual un método debe cambiar el estado (un comando) o devolver una respuesta (una consulta), pero no ambas cosas.

Serializabilidad
La garantía de aislamiento más fuerte: las transacciones concurrentes producen el mismo resultado que si se hubieran ejecutado de una en una en algún orden.

Sistema de registro (system of record)
La fuente autoritativa de un dato, cuyo valor prevalece cuando las copias no coinciden, como el libro mayor central de un banco para los saldos de las cuentas.

Snapshot (instantánea)
Copia almacenada del estado de un agregado en una versión de evento conocida, que se usa para no tener que reproducir un flujo largo desde el principio; es una caché, no una fuente de verdad.

Split brain (cerebro dividido)
Fallo en el que dos nodos creen ser el líder y ambos aceptan escrituras, de modo que sus datos divergen.

SQL distribuido (NewSQL)
Bases de datos relacionales, como Spanner y CockroachDB, que conservan SQL y las transacciones ACID mientras particionan y replican los datos en muchos nodos o regiones.

Tabla de dimensiones
Una tabla de atributos descriptivos, como el nombre, el tipo de cocina y la ciudad de un restaurante, que se usa para filtrar y agrupar hechos.

Tabla de hechos
La tabla central de un esquema en estrella, que contiene una fila por cada evento de negocio con una granularidad (grain) establecida, con medidas numéricas y claves hacia las dimensiones.

Tasa de aciertos de caché
La fracción de consultas que la caché puede responder con su propio contenido; el resto son fallos de caché (misses) que van a la fuente más lenta.

Tiempo de vida (TTL)
Cuánto tiempo puede usarse un elemento en caché antes de considerarse obsoleto y tener que actualizarse o revalidarse.

Tiempo del evento
El momento en que un evento ocurrió realmente, a diferencia del tiempo de procesamiento, cuando un sistema lo observa.

Upcasting
Convertir un evento almacenado con una versión antigua del esquema a la versión actual en el momento de leerlo, para que el código de la aplicación solo tenga que manejar una forma.

Write skew (sesgo de escritura)
Anomalía en la que dos transacciones leen datos que se solapan, cada una decide que un cambio es seguro y actualizan filas distintas, de modo que juntas incumplen una regla que cada una había comprobado.