Engineer Software

Vender la última entrada exactamente una vez

13 de octubre de 2026

Una venta relámpago ofrece diez mil unidades a un millón de personas a la vez. El fallo interesante no es que la tienda se caiga. Es que siga funcionando y venda diez mil cuarenta. El reflejo que viene después —la base de datos es demasiado pequeña, hay que fragmentarla— responde a una pregunta que nadie hizo. La base de datos nunca fue demasiado grande. Todos los compradores escribían en la misma fila.

Aquí desmenuzamos el problema antes de que llegue a un proyecto: de dónde viene el límite, la sentencia que elimina la mayor parte y qué construir cuando una sola base de datos realmente no basta.

Aplique la ingeniería a su sistema

La ingeniería detrás de «Vender la última entrada exactamente una vez» puede orientar su próximo cambio. Podemos investigar comportamientos similares en su software, datos o infraestructura. Una estructura que refleja cómo funciona el negocio para poder responder también a las preguntas que surjan más adelante.

Explore nuestros servicios Póngase en contacto →

Siga el razonamiento y sus límites

Lea el razonamiento y sus límites en esta nota. Podemos comprobar si el mismo diagnóstico se sostiene en su entorno. Diseño y operación de 6 flujos ETL/ELT — 6 flujos ETL en tres motores de bases de datos

Diseño y operación de 6 flujos ETL/ELT Póngase en contacto →

Una fila, todos los compradores

Veinte millones de filas repartidas entre un clúster son un problema resuelto. Una fila que debe modificar cada solicitud no lo es, y añadir máquinas no lo cambia porque la fila reside en una de ellas. Los almacenes distribuidos lo dicen

claramente: DynamoDB asigna a una partición unas 1.000 unidades de escritura por segundo, y toda escritura de un elemento llega a su partición, por muchas otras que haya.

La estructura tiene un teorema detrás. Bailis y sus colegas llaman invariante-confluente a una invariante cuando las réplicas pueden aceptar escrituras por separado y después fusionarse en un estado válido. `stock >= 0` es el ejemplo clásico de una que no lo es: dos réplicas que ven una unidad restante la venderán cada una, y ninguna fusión puede deshacer la segunda venta. Evitar siempre la sobreventa requiere coordinación. La única pregunta es cuánta paga cada venta.



La cifra que lo decide

El límite de un artículo es aproximadamente cuántas ventas permite un bloqueo dividido entre el tiempo que se mantiene:

```
max sales/s per item ≈ sales per lock ÷ lock hold time
```

Qué retiene la fila	Límite por fila
Bloquear y después llamar al pago	~2/s
Bloquear y después ejecutar lógica de aplicación	~100–200/s
Un `UPDATE` condicional	~500–2.000/s
Lotes de 100 por commit	~50.000/s
Un único escritor en memoria	~100.000+/s

Son órdenes de magnitud, no resultados de pruebas de rendimiento; el hardware los desplaza. Lo que no cambia es la estructura. La mayor parte de un bloqueo breve corresponde al commit esperando que su registro llegue al disco. Por eso los lotes ganan dos órdenes de magnitud: cien ventas comparten una escritura forzada.

Quedan cuatro palancas, y cada diseño posterior usa una:

- **Mantener el bloqueo menos tiempo**: una sentencia, nada de red dentro.
- **Hacer más por bloqueo**: agrupar los decrementos.
- **Tener más bloqueos**: dividir las existencias entre varias claves.
- **Enviar menos solicitudes al bloqueo**: rechazar a la mayoría de compradores antes de que lleguen.

  La última importa más de lo que parece. Con un millón de compradores y diez mil unidades, 990.000 solicitudes deben fallar. Los rechazos son la mayor parte del tráfico, así que «agotado» debe ser la respuesta más barata del sistema: una sala de espera antes de la venta, un control de admisión que deje pasar aproximadamente a tantos compradores como unidades, un indicador `sold_out` servido por el CDN sin consultar a nadie y un límite de frecuencia por cuenta.

Corregir primero la única base de datos

La versión que se arrastra o sobrevende suele tener este aspecto:

```
BEGIN;
SELECT available FROM inventory
 WHERE item_id = :item FOR UPDATE;
-- the application checks the number,
-- and perhaps calls a payment provider
UPDATE inventory SET available = :new
 WHERE item_id = :item;
COMMIT;
```

El bloqueo es correcto. Lo incorrecto es lo que abarca: cada milisegundo que la aplicación emplea entre el `SELECT` y el `UPDATE` es uno que espera el siguiente comprador. Sin `FOR UPDATE` es peor. Dos compradores leen 1, ambos escriben 0 y se ha vendido una unidad dos veces.

La solución une la comprobación y el decremento en una sentencia:

```
UPDATE inventory
   SET available = available - :qty
 WHERE tenant_id = :tenant
   AND item_id = :item
   AND available >= :qty
RETURNING available;
-- one row back: reserved. none: sold out.
```

En PostgreSQL con READ COMMITTED, el segundo comprador que llega mientras el primero retiene la fila espera y después vuelve a evaluar `WHERE` sobre la fila confirmada por el primero. Si aquella venta consumió la última unidad, la condición falla y el segundo no recibe ninguna fila —agotado— en vez de un número negativo. Con REPEATABLE READ o SERIALIZABLE recibe un error de serialización: es seguro, pero convierte la contención en reintentos. La sentencia está escrita para READ COMMITTED. InnoDB de MySQL llega a la misma respuesta porque su `UPDATE` lee la última fila confirmada en cualquier nivel. Añada también la restricción para que un error en otro lugar falle de forma visible en vez de vender existencias inexistentes:

```
ALTER TABLE inventory
  ADD CONSTRAINT available_not_negative
  CHECK (available >= 0);
```

Después, el resto del trabajo con una sola base de datos, aproximadamente en orden de rentabilidad:

- **Nada externo dentro de la transacción.** Ninguna llamada de pago, solicitud HTTP ni publicación de mensajes. Eso va antes o después.
- **Toque la fila conflictiva al final.** Inserte primero el pedido y todo lo demás para que el bloqueo de la fila disputada dure solo la última sentencia y el commit.
- **Nada de bloqueo optimista en un contador muy solicitado.** Una columna de versión convierte la contención en reintentos, y durante una venta los reintentos son la carga.
- **Sirva las lecturas desde otro lugar.** El «quedan 12» de la página de producto puede venir de una réplica o caché con un segundo de retraso. Solo la venta necesita el primario.
- **Dé al inventario su propia base de datos**, para que una venta nunca espere detrás del catálogo por el mismo disco.
- **Agrupe las conexiones.** Cien mil compradores deberían ser unas decenas de conexiones mediante PgBouncer, no miles de procesos de backend esperando un bloqueo.

Los asientos tienen otra estructura porque cada unidad es distinta y requieren otra sentencia. Cada asiento es una fila y el comprador salta las retenidas por otra transacción en lugar de hacer cola detrás:

```
UPDATE seats
   SET held_by = :order,
       held_until = now() + interval '10 min'
 WHERE id = (
   SELECT id FROM seats
    WHERE event_id = :event
      AND sold_at IS NULL
      AND (held_until IS NULL
           OR now() > held_until)
    LIMIT 1
    FOR UPDATE SKIP LOCKED)
RETURNING id;
```

Una reserva caducada vuelve a estar disponible sin más, así que no hace falta un proceso de limpieza para que la respuesta sea correcta. Además, `now()` es el reloj de la base de datos: los servidores de aplicación con relojes distintos no pueden discrepar sobre el vencimiento.

Para muchas tiendas, este apartado es toda la solución. Aplíquelo antes de lo que sigue, porque el sharding es una puerta de un solo sentido: fácil de atravesar y caro de desandar.



Reservar primero, pagar después

Un comprador que toma la última unidad y después abandona el formulario de tarjeta no la ha comprado. Por eso una venta no es una escritura, sino un ciclo breve:

```
AVAILABLE → HELD, for minutes → SOLD
                              → RELEASED
```

- **La reserva se identifica por la solicitud, no por el intento.** Cada checkout lleva una clave de idempotencia y la tabla de reservas incluye `UNIQUE (tenant_id, idempotency_key)`. Así, un clic reintentado recibe la reserva existente en vez de una segunda unidad. Un detalle de PostgreSQL puede sorprender: `INSERT ... ON CONFLICT DO NOTHING RETURNING id` no devuelve nada si la fila ya existía. «Ninguna fila» significa «búsquela», no «ha fallado».

- **Cada transición es condicional.** Confirmar una venta pasa una reserva de `held` a `sold` solo si sigue en `held`, y una liberación devuelve stock únicamente si su actualización cambió exactamente una fila. Una liberación ejecutada dos veces —por ejemplo, el tiempo de espera y el botón de cancelar — devuelve la unidad una sola vez.
- **Las reservas caducan en minutos.** Lo habitual son entre cinco y diez. Amplíela una vez al comenzar el pago para que una tarjeta lenta no pierda la unidad que está pagando.
- **Limite las unidades por comprador al reservar**, para que una cuenta no pueda retener todo el stock mientras decide.

Debajo de todo hay una línea, la más útil del diseño:

```
available + held + sold = total
```

Se cumple después de cada transacción, una consulta la verifica y, cuando deja de cumplirse, algo ha sobrevendido o perdido una unidad, antes de que lo diga un cliente.

Cuando una base de datos no basta

A veces realmente no basta: muchas organizaciones cliente, muchas ventas simultáneas, varias regiones. La estructura que escala conserva una regla: cada artículo tiene exactamente un responsable de sus existencias, un servicio de inventario, y nada más decide. Ese servicio se fragmenta por `(tenant_id, item_id)` para que las escrituras de un artículo nunca abarquen dos fragmentos. Las organizaciones cliente mayores reciben celdas propias, de modo que su venta no prive de recursos a las demás. Solo la reserva debe ser síncrona; todo lo posterior puede ser un mensaje.

Para el artículo más solicitado dentro de un fragmento hay cuatro herramientas, cada una con una de las palancas anteriores.

Un control de admisión delante. Un contador Redis, decrementado por un script Lua para que comprobar y decrementar sean un paso, admite aproximadamente a tantos compradores como unidades y rechaza al resto en microsegundos. No es el árbitro. Redis replica de forma asíncrona, así que después de una conmutación

por error el nuevo primario puede haber olvidado algunas admisiones y dejar pasar a unos pocos de más. El `UPDATE` condicional de la base de datos sigue decidiendo y un conciliador restablece el control desde la cifra de la base de datos.

Dividir las existencias. Diez mil unidades pueden ser cien buckets de cien, cada uno con su propia fila o fragmento, y cada comprador dirigido a uno: cien bloqueos en vez de uno. La idea es antigua: transacciones escrow de O'Neil en 1986, después el protocolo de demarcación de Barbará y Garcia-Molina y, más tarde, los contadores acotados de Balegas y sus colegas. Su único coste real es el tramo final. El bucket de un comprador se vacía mientras otros conservan stock, así que hay que probar otro o mover el sobrante y volver a uno solo al final de la venta. La misma idea da a cada región su cuota para que la mayoría de las ventas no crucen un océano para preguntar.

Un escritor por artículo. Ponga cada solicitud de un artículo en una partición Kafka, leída por un consumidor que conserva la cuenta en memoria y confirma sus decisiones por lotes. Un hilo y ningún bloqueo: la estructura LMAX. Las solicitudes llevan un identificador y el escritor descarta las que ya decidió, porque una cola entrega dos veces.

Unidades como filas. Si las unidades son pocas y distintas —asientos, ediciones numeradas—, cree una fila por unidad de antemano y resérvelas con `SKIP LOCKED` como arriba. O conserve sus identificadores en un conjunto Redis y extraiga uno con `SPOP` . Ya no queda un contador por el que competir.

El resto del checkout

El pago, el pedido y el recibo son sistemas separados y ninguna transacción los abarca. El commit en dos fases es la respuesta de manual y aquí la equivocada: bloquea si cualquier participante va lento, añade viajes de ida y vuelta a la ruta más solicitada y los proveedores de pago no participan. Una saga resuelve el trabajo: una secuencia de pasos locales, cada uno con otro que lo deshace:

```
1. reserve stock       undo: release the hold
2. pending order       undo: cancel the order
3. extend the hold     (expires on its own)
```

```
4. authorise payment   undo: void it
5. confirm HELD → SOLD
6. capture payment
```

Autorizar primero y capturar al final significa que un pago que llega tras caducar su reserva se anula en lugar de reembolsarse: el dinero nunca se movió. Cada paso escribe su mensaje en una tabla outbox dentro de la misma transacción que el cambio anunciado, y la captura de cambios de datos lo publica. Así no se pierde nada entre guardar y enviar. Todos los consumidores son idempotentes y se identifican por el pedido. Los mensajes llegan al menos una vez y la idempotencia lo convierte, en la práctica, en una vez.

Aquí no corresponde ningún bloqueo distribuido. Kleppmann señala que su titular puede quedar pausado más allá de su concesión —una recolección de basura, una máquina virtual detenida— y seguir escribiendo. La solución es un token de vallado que compruebe el almacén. Pero un almacén capaz de comprobar un token también puede comprobar `available >= :qty`, y entonces nunca necesitó el bloqueo.



Pausar el artículo, nunca sobrevenderlo

Consistencia y disponibilidad se eligen por pregunta, no por sistema. «¿Puedo vender esta unidad?» debe responderse con consistencia: si no se puede alcanzar al responsable del stock de un artículo, ese artículo deja de venderse. Todo lo demás —el catálogo, el aviso «quedan 3», el historial de pedidos— puede seguir disponible y algo desactualizado.

La sobreventa más silenciosa viene de una conmutación por error. Con replicación asíncrona, un primario confirma una venta y muere antes de que la réplica la tenga; la réplica se promociona, la venta ha desaparecido y la unidad se vende otra vez. La solución es hacer duradera la venta en más de una máquina antes de confirmarla: un standby síncrono en PostgreSQL (`synchronous_commit` con `synchronous_standby_names`) o un almacén basado en Raft. Eso añade latencia a cada commit y alarga precisamente el tiempo de bloqueo por el que se divide el límite. Los lotes y el control de admisión lo compensan. Limite también la pausa: un fragmento averiado debería detener sus artículos, no la tienda.

Demostrarlo

Nada de esto debe aceptarse por confianza, tampoco lo nuestro.

- **Vigile la invariante.** Ejecute `available + held + sold = total` de forma programada, grafíquela y active una alerta ante cualquier desviación. Cero sobreventas es un objetivo de nivel de servicio con una alerta, no una esperanza.
- **Provoque la carrera.** Diez mil compradores simultáneos para cien unidades en la batería de pruebas, comprobando exactamente cien vendidas y ninguna cifra por debajo de cero.
- **Rompa cosas durante la carrera.** Detenga el primario, corte la red, pause un proceso, al estilo Jepsen, y compruebe después la invariante.
- **Pruebe la carga de los rechazos.** La ruta de agotado debe seguir siendo rápida cuando representa el 99 % del tráfico y la infraestructura de borde debe absorberlo.
- **Reúna las métricas en un panel:** esperas de bloqueo, reservas caducadas y distancia entre el contador del control de admisión y el de la base de datos.

En qué orden

- Mida primero. `pg_stat_statements` y las esperas de bloqueo indican si la fila conflictiva es siquiera el problema.
- Convierta la escritura en una sentencia condicional.
- Descargue tráfico: sala de espera, control de admisión, «agotado» en caché.
- Dé al inventario un servicio y un responsable propios.
- Fragmente por `(tenant_id, item_id)`, solo si los cuatro pasos anteriores no bastaron.
- Añada un modo para artículos muy solicitados, buckets o un único escritor, para la venta que sabe que llegará.
- Las regiones al final, con cuotas.

Qué le diríamos a la siguiente persona

- **El sharding responde a otra pregunta.** Reparte muchas filas; una venta con sobreventa es una fila. Averigüe cuál es su caso antes de construir.

- **Una la comprobación y el decremento en una sentencia.** La mayor parte de la distancia entre correcto y sobrevendido es el tiempo entre leer una cifra y escribirla.
- **La mayor parte del tráfico es la respuesta no.** Haga del rechazo lo más barato del sistema.
- **Pause antes de sobrevender.** Un artículo que deja de venderse treinta segundos requiere una disculpa. Uno vendido dos veces supone un reembolso, una cola de atención al cliente y un cliente que lo recuerda.
- **Escriba la invariante y vigílela.** Es una línea y lo único que demuestra que el diseño funciona en producción, no solo en la revisión.

Fuentes

- PostgreSQL: aislamiento Read Committed
- PostgreSQL: `FOR UPDATE` y `SKIP LOCKED`
- PostgreSQL: replicación síncrona
- Bailis et al., Coordination Avoidance in Database Systems
- Balegas et al., Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants
- Martin Kleppmann: How to do distributed locking
- Martin Fowler: The LMAX Architecture
- DynamoDB: diseño de claves de partición
- DynamoDB: particiones y distribución de datos
- microservices.io: Saga
- microservices.io: Transactional outbox
- Redis: replicación
- Jepsen

Todas las entradas del blog

Mida antes de cambiar

Traiga un conjunto de datos representativo, las preguntas que debe responder y el cuello de botella actual. Podemos acordar cómo medir la calidad, la capacidad de procesamiento y la recuperación.

Vea cuánto cuesta Póngase en contacto →

Hable de sus observaciones

Hablemos de «Vender la última entrada exactamente una vez» en el contexto de su proyecto. Cuéntenos qué observó y cómo sería un resultado útil.

Explore nuestros servicios

welcome@engineer.company +45 50 10 61 84 Copenhague, Dinamarca

Llévese esta página

Tarjeta de contacto

Creado con esmero y cariño por D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Derechos de autor © 2025 – actualidad · Engineer ApS · Todos los derechos reservados.

Impreso desde https://software.engineer.company/es/notes/selling-limited-stock-without-overselling/