Engineer Plattform

Att sälja den sista biljetten exakt en gång

13 oktober 2026

En blixtrea ställer tiotusen enheter framför en miljon människor på en gång. Det intressanta felet är inte att butiken går ned. Det är att butiken håller sig uppe och säljer tiotusenfyrtio. Instinkten som följer — databasen är för liten, sharda den — svarar på en fråga som ingen har ställt. Databasen var aldrig för stor. Varje köpare skrev till samma rad.

Vi plockar här isär problemet innan det når ett projekt: var taket kommer ifrån, den enda sats som tar bort det mesta av det, och vad man bygger när en enda databas verkligen inte räcker.

Använd erfarenheten i ert system

Arbetet bakom ”Att sälja den sista biljetten exakt en gång” kan bidra till er nästa förändring. Vi kan undersöka liknande beteenden i er programvara, era data eller er infrastruktur. En struktur som speglar hur verksamheten faktiskt fungerar, så att de frågor du behöver svar på senare fortfarande går att ställa.

Se våra tjänster Hör av er →

Följ resonemanget och dess gränser

Läs resonemanget och dess begränsningar i anteckningen. Vi kan testa om samma diagnos håller i er miljö. Byggde åtkomsträttigheter via Stripe och appköp — 11 tabeller som avgör vem som får se vad

Byggde åtkomsträttigheter via Stripe och appköp Hör av er →

En rad, alla köpare

Tjugo miljoner rader utspridda över ett kluster är ett löst problem. En rad som varje begäran måste ändra är det inte, och fler maskiner rör den inte, eftersom raden finns på en av dem. Distribuerade lagringssystem säger det rakt ut: DynamoDB ger en enskild partition ungefär 1 000 skrivenheter per sekund, och varje skrivning till ett objekt hamnar på det objektets partition, hur många andra det än finns.

Det finns ett teorem bakom formen. Bailis med kolleger kallar en invariant för invariant-confluent när repliker kan ta emot skrivningar var för sig och ändå slås samman till ett giltigt tillstånd. `stock >= 0` är skolboksexemplet på en som inte är det: två repliker som var och en ser en enhet kvar säljer den båda, och ingen sammanslagning kan ta tillbaka den andra försäljningen. Att aldrig översälja kräver samordning. Den enda frågan är hur mycket av den varje försäljning betalar för.

Talet som avgör

Taket för en vara är ungefär hur många försäljningar ett lås bär, delat med hur länge låset hålls:

```
max försäljningar/s per vara ≈ försäljningar per lås ÷ låsets hålltid
```

Vad som håller raden	Tak per rad
Lås, anropa sedan betalning	~2/s
Lås, kör sedan applogik	~100–200/s
En villkorad `UPDATE`	~500–2 000/s
Batchar om 100 per commit	~50 000/s
En skrivare i minnet	~100 000+/s

Det här är storleksordningar, inte benchmarkresultat; hårdvaran flyttar dem. Det den inte flyttar är formen. Det mesta av ett kort lås är commiten som väntar på att dess logg ska nå disken, och därför vinner batchning med två storleksordningar: hundra försäljningar delar på en enda skrivning till disk.

Det lämnar fyra spakar, och varje lösning nedan drar i en av dem:

- **Håll låset kortare tid** — en sats, ingenting över nätverket inuti den.
- **Gör mer per lås** — batcha nedräkningarna.
- **Ha fler lås** — dela upp lagersaldot på flera nycklar.
- **Skicka färre begäranden till låset** — avvisa de flesta köpare innan de når det.

Den sista betyder mer än det låter. Med en miljon köpare och tiotusen enheter måste 990 000 begäranden misslyckas. Avvisningar är det mesta av trafiken, så ”slutsålt” måste vara det billigaste svar systemet ger: ett väntrum framför försäljningen, en grind som släpper in ungefär lika många köpare som det finns enheter, en `sold_out`-flagga som CDN:et serverar utan att fråga någon, och en hastighetsbegränsning per konto.

Åtgärda den enda databasen först

Versionen som kryper fram, eller översäljer, ser oftast ut så här:

```
BEGIN;
SELECT available FROM inventory
 WHERE item_id = :item FOR UPDATE;
-- applikationen kontrollerar talet,
-- och anropar kanske en betalningsleverantör
UPDATE inventory SET available = :new
 WHERE item_id = :item;
COMMIT;
```

Låset är rätt. Det som det hålls över är fel: varje millisekund som applikationen tillbringar mellan `SELECT` och `UPDATE` är en millisekund som nästa köpare väntar. Utan `FOR UPDATE` är det värre. Två köpare läser 1, båda skriver 0, och en enhet har sålts två gånger.

Lösningen gör kontrollen och nedräkningen till en enda sats:

```
UPDATE inventory
   SET available = available - :qty
 WHERE tenant_id = :tenant
   AND item_id = :item
   AND available >= :qty
RETURNING available;
-- en rad tillbaka: reserverad. ingen: slutsåld.
```

I PostgreSQL med READ COMMITTED väntar en andra köpare som når raden medan den första håller den, och utvärderar sedan `WHERE`-villkoret på nytt mot raden som den första committade. Om den försäljningen tog den sista enheten uppfylls inte villkoret, och den andra köparen får ingen rad tillbaka — slutsålt — i

stället för ett negativt tal. Med REPEATABLE READ eller SERIALIZABLE får den andra köparen i stället ett serialiseringsfel, vilket är säkert men gör konkurrensen om raden till omförsök; READ COMMITTED är den nivå som den här satsen är skriven för. InnoDB i MySQL kommer fram till samma svar, eftersom en `UPDATE` där läser den senast committade raden på alla nivåer. Lägg också till integritetsvillkoret, så att en bugg någon annanstans misslyckas högljutt i stället för att sälja lager som inte finns:

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

Därefter resten av arbetet med en enda databas, ungefär i ordning efter vad det ger:

- **Inget externt inuti transaktionen.** Inget betalningsanrop, ingen HTTP-begäran, inget publicerat meddelande. De hör hemma före transaktionen eller efter den.
- **Rör den heta raden sist.** Infoga ordern och allt annat först, så att låset på raden som flera transaktioner konkurrerar om bara varar under den sista satsen och commiten.
- **Ingen optimistisk låsning på en het räknare.** En versionskolumn gör konkurrens till omförsök, och under en rea är omförsöken själva lasten.
- **Servera läsningarna från något annat ställe.** Produktsidans ”12 kvar” kan komma från en replik eller en cache som ligger en sekund efter. Bara försäljningen behöver primärnoden.
- **Ge lagret en egen databas,** så att en försäljning aldrig köar bakom katalogen för samma disk.
- **Poola anslutningarna.** Hundratusen köpare ska vara några dussin anslutningar genom PgBouncer, inte tusentals backendprocesser som köar för ett enda lås.

Sittplatser har en annan form, eftersom varje enhet är unik, och de kräver en annan sats. Varje plats är en rad, och en köpare hoppar över de rader som en annan transaktion håller i stället för att köa bakom dem:

```
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;
```

En reservation som har löpt ut är helt enkelt tillgänglig igen, så det behövs ingen städprocess för att svaret ska bli rätt. Och `now()` är databasens klocka, så applikationsservrar vars klockor går olika kan inte vara oense om när en reservation tog slut.

För många butiker är det här avsnittet hela lösningen. Gör det före allt nedan, eftersom sharding är en enkelriktad dörr: lätt att gå igenom, dyr att gå tillbaka genom.



Reservera först, betala sedan

En köpare som tar den sista enheten och sedan överger kortformuläret har inte köpt den. En försäljning är alltså inte en skrivning utan en kort livscykel:

```
AVAILABLE → HELD, i minuter → SOLD
                            → RELEASED
```

- **Reservationen nycklas på begäran, inte på försöket.** Varje köp i kassan bär en idempotensnyckel, och reservationstabellen har `UNIQUE (tenant_id, idempotency_key)`, så att ett upprepat klick får tillbaka den reservation det redan har i stället för en andra enhet. En detalj i PostgreSQL biter här: `INSERT ... ON CONFLICT DO NOTHING RETURNING id` returnerar ingenting när raden redan fanns, så ”ingen rad tillbaka” betyder ”slå upp den”, inte ”misslyckades”.
- **Varje övergång är villkorad.** Att bekräfta en försäljning flyttar en reservation från `held` till `sold` bara där den fortfarande är `held`, och att släppa en reservation lägger tillbaka enheten i lagersaldot bara om den egna uppdateringen ändrade exakt en rad. Ett släpp som körs två gånger — t.ex. timeouten och avbrytknappen — lämnar tillbaka enheten en gång.

- **Reservationer löper ut efter minuter.** Fem till tio är vanligt. Förläng reservationen en gång när betalningen startar, så att ett långsamt kort inte förlorar den enhet det betalar för.
- **Begränsa antalet enheter per köpare vid reservationen,** så att ett konto inte kan hålla hela lagret medan det bestämmer sig.

Under alltihop ligger en rad, den mest användbara i lösningen:

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

Den gäller efter varje transaktion, en enda SQL-fråga kontrollerar den, och när den slutar gälla har något översålt eller tappat en enhet — innan en kund säger det.

När en databas inte räcker

Ibland räcker den verkligen inte: många tenants, många reor samtidigt, flera regioner. Strukturen som skalar behåller en regel — varje vara har exakt en ägare av sitt lagersaldo, en lagertjänst, och ingenting annat bestämmer — och shardar den tjänsten på `(tenant_id, item_id)`, så att en varas skrivningar aldrig spänner över två shards. De största tenanterna får egna celler, så att deras rea inte kan svälta ut någon annans. Bara reservationen måste vara synkron; allt efter den kan vara ett meddelande.

För den heta varan inuti en shard finns det fyra verktyg, och vart och ett drar i en av spakarna ovan.

En grind framför. En Redis-räknare, som räknas ned av ett Lua-skript så att kontrollen och nedräkningen blir ett enda steg, släpper in ungefär lika många köpare som det finns enheter och avvisar alla andra på mikrosekunder. Den är inte den som avgör. Redis replikerar asynkront, så efter en failover kan den nya primärnoden ha glömt några insläpp och släppt igenom några för många. Databasens villkorade `UPDATE` avgör fortfarande, och en avstämningsprocess återställer grinden från databasens tal.

Dela upp lagersaldot. Tiotusen enheter kan vara hundra hinkar om hundra, var och en sin egen rad eller sin egen shard, med varje köpare skickad till en av dem: hundra lås i stället för ett. Idén är gammal — O'Neils escrow-transaktioner 1986, sedan demarkationsprotokollet från Barbará och Garcia-Molina, sedan de begränsade räknarna från Balegas med kolleger — och dess enda verkliga kostnad är svansen. En köpares hink tar slut medan andra fortfarande har lager, så antingen prövar man en annan hink eller flyttar resterna, och mot slutet av en rea slår man ihop dem till en igen. Samma idé ger varje region en egen kvot, så att de flesta försäljningar aldrig korsar ett hav för att fråga.

En skrivare per vara. Lägg varje begäran för en vara på en Kafka-partition, läst av en enda konsument som håller räkningen i minnet och committar sina beslut i batchar. En tråd och inga lås: LMAX-formen. Begäranden bär ett id, och skrivaren kastar en som den redan har avgjort, eftersom en kö levererar två gånger.

Enheter som rader. När enheterna är få och unika — sittplatser, numrerade utgåvor — skapa en rad per enhet i förväg och ta dem med `SKIP LOCKED` som ovan, eller håll enheternas id i en Redis-mängd och ta en med `SPOP`. Det finns ingen räknare kvar att slåss om.

Resten av köpflödet

Betalningen, ordern och kvittot är separata system, och ingen transaktion spänner över dem. Tvåfas-commit är skolboksvaret och fel svar här: den blockerar så länge någon deltagare är långsam, den lägger till fler nätverksrundor på den hetaste vägen, och betalningsleverantörer deltar inte i den. En saga gör jobbet, en sekvens av lokala steg, vart och ett med ett steg som ångrar det:

```
1. reservera lager       ångra: släpp reservationen
2. väntande order        ångra: avbryt ordern
3. förläng reservationen (löper ut av sig själv)
4. auktorisera betalning ångra: annullera den
5. bekräfta HELD → SOLD
6. dra betalningen
```

Att auktorisera först och dra betalningen sist betyder att en betalning som kommer fram efter att dess reservation har löpt ut annulleras i stället för att återbetalas: pengarna flyttades aldrig. Varje steg skriver sitt meddelande till en outbox-tabell i samma transaktion som ändringen det meddelar, och change data

capture publicerar det, så att ingenting går förlorat mellan sparat och skickat. Varje konsument är idempotent, nycklad på ordern. Meddelanden kommer fram minst en gång, och idempotensen gör det i praktiken till en gång.

Ett distribuerat lås hör inte hemma någonstans i det här. Kleppmanns poäng är att den som håller ett lås kan bli pausad längre än låsets lease-tid — en skräpsamling, en virtuell maskin som står still — och fortsätta skriva, och botemedlet är en fencing-token som lagringssystemet kontrollerar. Men ett lagringssystem som kan kontrollera en token kan kontrollera `available >= :qty`, och då behövde det aldrig låset.

Pausa varan, översälj den aldrig

Konsistens och tillgänglighet väljs per fråga, inte per system. ”Kan jag sälja den här enheten?” måste vara konsistent: när ägaren av en varas lagersaldo inte går att nå slutar den varan att säljas. Allt annat — katalogen, märket med ”3 kvar”, orderhistoriken — kan förbli tillgängligt och vara lite inaktuellt.

Den tystaste översäljningen kommer från en failover. Med asynkron replikering bekräftar en primärnod en försäljning och dör innan dess replik har fått den; repliken befordras, försäljningen är borta, och enheten säljs igen. Lösningen är att göra en försäljning beständig på mer än en maskin innan den bekräftas: en synkron standby i PostgreSQL (`synchronous_commit` med `synchronous_standby_names`), eller ett lagringssystem byggt på Raft. Det kostar latens vid varje commit, vilket förlänger precis den låshålltid som taket delas med, och batchning och grinden köper tillbaka den. Avgränsa också pausen: en shard som har gått ned ska stoppa varorna på den, inte butiken.

Bevisa det

Inget av detta ska tas på tro, inte ens det vi säger.

- **Bevaka invarianten.** Kör `available + held + sold = total` schemalagt, visa den i ett diagram och larma vid varje avvikelse. Noll översäljningar är ett servicenivåmål med ett larm, inte en förhoppning.

- **Framkalla kapplöpningen med flit.** Tiotusen samtidiga köpare för hundra enheter, i testsviten, med kontroll av att exakt hundra såldes och att inget tal gick under noll.
- **Ha sönder saker under kapplöpningen.** Döda primärnoden, kapa nätverket, pausa en process — i Jepsen-stil — och kontrollera invarianten efteråt.
- **Lasttesta avvisningarna.** Vägen för slutsålt måste förbli snabb när den är 99 % av trafiken, och edge-nätverket måste ta upp den.
- **Samla mätvärdena på en instrumentpanel:** låsväntetider, utgångna reservationer och avståndet mellan grindens räkning och databasens.



I vilken ordning

- Mät först. `pg_stat_statements` och låsväntetiderna visar om den heta raden alls är problemet.
- Gör skrivningen till en enda villkorad sats.
- Minska lasten: ett väntrum, en grind, ett cachat ”slutsålt”.
- Ge lagret en egen tjänst och en egen ägare.
- Sharda på `(tenant_id, item_id)`, och bara om de fyra första inte räckte.
- Lägg till ett läge för heta varor, hinkar eller en enda skrivare, för den rea du vet kommer.
- Regioner sist, med kvoter.



Vad vi skulle säga till nästa person

- **Sharding svarar på en annan fråga.** Den sprider många rader, och en översåld rea är en enda rad. Ta reda på vilket du har innan du bygger.
- **Gör kontrollen och nedräkningen till en sats.** Det mesta av avståndet mellan bra och översålt är tiden mellan att läsa ett tal och att skriva det.
- **Det mesta av trafiken är svaret nej.** Gör avvisningen till det billigaste systemet gör.
- **Pausa innan du översäljer.** En vara som slutar säljas i trettio sekunder blir en ursäkt. En vara som säljs två gånger blir en återbetalning, en supportkö och en kund som minns.
- **Skriv ned invarianten och bevaka den.** Den är en rad, och den är det enda som visar att lösningen fungerar i produktion snarare än i granskningen.

Källor

- PostgreSQL: isoleringsnivån Read Committed
- PostgreSQL: `FOR UPDATE` och `SKIP LOCKED`
- PostgreSQL: synkron replikering
- 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: utformning av partitionsnycklar
- DynamoDB: partitioner och datafördelning
- microservices.io: Saga
- microservices.io: Transactional outbox
- Redis: replikering
- Jepsen

Alla blogginlägg

Mät innan ni ändrar

Ta med ett representativt dataset, frågorna det ska besvara och dagens flaskhals. Vi kan avtala mått på kvalitet, genomströmning och återställning.

Se vad det kostar Hör av er →

Diskutera era observationer

Diskutera ”Att sälja den sista biljetten exakt en gång” i samband med ert eget projekt. Berätta vad ni observerade och vad ett användbart resultat skulle vara.

Se våra tjänster

welcome@engineer.company +45 50 10 61 84 Köpenhamn, Danmark

Ta med dig den här sidan

Visitkort

Omsorgsfullt skapad med kärlek av D.

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

Upphovsrätt © 2025 – nu · Engineer ApS · Alla rättigheter förbehållna.

Utskrivet från https://platform.engineer.company/sv/notes/selling-limited-stock-without-overselling/