Engineer Data

At sælge den sidste billet præcis én gang

13. oktober 2026

Et flash-salg stiller ti tusind enheder foran en million mennesker på én gang. Den interessante fejl er ikke, at butikken går ned. Det er, at butikken bliver oppe og sælger ti tusind og fyrre. Instinktet bagefter — databasen er for lille, så shard den — svarer på et spørgsmål, ingen har stillet. Databasen var aldrig for stor. Alle købere skrev til den samme række.

Vi skiller problemet ad her, før det rammer et projekt: hvor loftet kommer fra, den ene sætning, der fjerner det meste af det, og hvad man bygger, når én database reelt ikke er nok.

Brug erfaringen i jeres system

Arbejdet bag »At sælge den sidste billet præcis én gang« kan bidrage til jeres næste ændring. Vi kan undersøge lignende adfærd i jeres software, jeres data eller jeres infrastruktur. En struktur, der passer til, hvordan forretningen faktisk fungerer, så de svar, I får brug for senere, stadig kan hentes.

Se vores ydelser Kontakt os →

Følg ræsonnementet og dets grænser

Læs ræsonnementet og dets begrænsninger i noten. Vi kan teste, om samme diagnose holder i jeres miljø. Har designet og drevet 6 ETL/ELT-pipelines — 6 ETL-pipelines på tre databasemotorer

Har designet og drevet 6 ETL/ELT-pipelines Kontakt os →

Én række, alle købere

Tyve millioner rækker fordelt over en klynge er et løst problem. Én række, som hver eneste forespørgsel skal ændre, er ikke, og flere maskiner rører den ikke, for rækken ligger på én af dem. Distribuerede lagre siger det ligeud: DynamoDB giver én partition omkring 1.000 skriveenheder i sekundet, og hver skrivning til én vare lander på varens partition, uanset hvor mange andre der findes.

Der ligger et teorem bag formen. Bailis og kolleger kalder en invariant invariant-konfluent, når replikaer kan tage imod skrivninger hver for sig og stadig flettes til en gyldig tilstand. `stock >= 0` er skoleeksemplet på én, der ikke er det: to replikaer, der hver ser én enhed tilbage, sælger den hver især, og ingen fletning kan tage det andet salg tilbage. At aldrig oversælge kræver koordinering. Spørgsmålet er kun, hvor meget af den hvert salg betaler for.

Tallet, der afgør det

Loftet for én vare er groft sagt, hvor mange salg én lås bærer, divideret med hvor længe låsen holdes:

```
maks. salg/s pr. vare ≈ salg pr. lås ÷ låsens holdetid
```

Hvad holder rækken	Loft pr. række
Lås, kald så betaling	~2/s
Lås, kør så app-logik	~100–200/s
Én betinget `UPDATE`	~500–2.000/s
Batches af 100 pr. commit	~50.000/s
Én skriver i hukommelsen	~100.000+/s

Det er størrelsesordener, ikke benchmarks; hardwaren flytter dem. Det, den ikke flytter, er formen. Det meste af en kort lås er committen, der venter på, at loggen når disken, og derfor vinder batching med to størrelsesordener: hundrede salg deler én flush.

Så er der fire håndtag tilbage, og hvert design herunder trækker i ét af dem:

- **Hold låsen kortere** — én sætning, intet over netværket inde i den.
- **Gør mere pr. lås** — saml nedskrivningerne i batches.
- **Hav flere låse** — del lageret over flere nøgler.
- **Send færre forespørgsler til låsen** — afvis de fleste købere, før de når frem.

Det sidste betyder mere, end det lyder til. Med en million købere og ti tusind enheder skal 990.000 forespørgsler fejle. Afvisninger er det meste af trafikken, så "udsolgt" skal være det billigste svar, systemet giver: et venteværelse foran salget, en port, der lukker omtrent lige så mange købere ind, som der er enheder, et `sold_out` -flag, som CDN'et serverer uden at spørge nogen, og en hastighedsgrænse pr. konto.

Ret den ene database først

Den version, der kryber af sted eller oversælger, ser som regel sådan ud:

```
BEGIN;
SELECT available FROM inventory
 WHERE item_id = :item FOR UPDATE;
-- applikationen tjekker tallet
-- og kalder måske en betalingsudbyder
UPDATE inventory SET available = :new
 WHERE item_id = :item;
COMMIT;
```

Låsen er rigtig. Det, den holdes hen over, er forkert: hvert millisekund, applikationen bruger mellem `SELECT` og `UPDATE` , er et millisekund, den næste køber venter. Uden `FOR UPDATE` er det værre. To købere læser 1, begge skriver 0, og én enhed er solgt to gange.

Rettelsen gør tjekket og nedskrivningen til én sætning:

```
UPDATE inventory
   SET available = available - :qty
 WHERE tenant_id = :tenant
   AND item_id = :item
   AND available >= :qty
RETURNING available;
-- én række tilbage: reserveret. ingen: udsolgt.
```

I PostgreSQL på READ COMMITTED venter en anden køber, der når rækken, mens den første holder den, og evaluerer så `WHERE` -betingelsen igen mod den række, den første committede. Tog det salg den sidste enhed, fejler betingelsen, og den anden køber får ingen række tilbage — udsolgt — i stedet for et negativt

tal. På REPEATABLE READ eller SERIALIZABLE får den anden køber i stedet en serialiseringsfejl, hvilket er sikkert, men gør konkurrence til genforsøg; READ COMMITTED er det niveau, sætningen er skrevet til. MySQL's InnoDB når samme svar, fordi en `UPDATE` dér læser den senest committede række på ethvert niveau. Tilføj også begrænsningen, så en fejl et andet sted fejler højlydt i stedet for at sælge lager, der ikke findes:

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

Så resten af arbejdet i den ene database, nogenlunde efter udbytte:

- **Intet eksternt inde i transaktionen.** Intet betalingskald, ingen HTTP-forespørgsel, ingen besked sendt. Det hører til før transaktionen eller efter den.
- **Rør den varme række sidst.** Indsæt ordren og alt andet først, så låsen på den række, som flere transaktioner konkurrerer om, kun varer under den sidste sætning og committen.
- **Ingen optimistisk låsning på en varm tæller.** En versionskolonne gør konkurrence til genforsøg, og under et salg er genforsøgene belastningen.
- **Server læsningerne fra et andet sted.** Produktsidens "12 tilbage" kan komme fra en replika eller en cache, der er et sekund bagud. Kun salget behøver den primære.
- **Giv lageret sin egen database,** så et salg aldrig skal konkurrere med kataloget om den samme disk.
- **Pool forbindelserne.** Hundrede tusind købere bør være et par dusin forbindelser gennem PgBouncer, ikke tusindvis af backends, der venter på én lås.

Pladser har en anden form, fordi hver enhed er unik, og de vil have en anden sætning. Hver plads er en række, og en køber springer de rækker over, som en anden transaktion holder, i stedet for at vente på 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 udløbet reservation er simpelthen ledig igen, så der skal ingen oprydder til, for at svaret er rigtigt. Og `now()` er databasens ur, så applikationsservere, hvis ure er uenige, ikke kan være uenige om, hvornår en reservation udløb.

For mange butikker er dette afsnit hele rettelsen. Gør det før alt herunder, for sharding er en dør, der kun går én vej: let at gå igennem, dyr at gå tilbage igennem.

Reservér først, betal bagefter

En køber, der tager den sidste enhed og så forlader kortformularen, har ikke købt den. Så et salg er ikke én skrivning, men et kort livsforløb:

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

- **Reservationen har forespørgslens nøgle, ikke forsøgets.** Hvert checkout bærer en idempotensnøgle, og reservationstabellen bærer `UNIQUE (tenant_id, idempotency_key)`, så et gentaget klik får den reservation, det allerede har, og ikke en enhed mere. En detalje i PostgreSQL bider her: `INSERT ... ON CONFLICT DO NOTHING RETURNING id` returnerer intet, når rækken fandtes i forvejen, så "ingen række tilbage" betyder "slå den op", ikke "fejlede".
- **Hver overgang er betinget.** At bekræfte et salg flytter en reservation fra `held` til `sold` kun dér, hvor den stadig er `held`, og en frigivelse giver kun lageret tilbage, hvis dens egen opdatering ændrede præcis én række. En frigivelse, der kører to gange — timeouten og annullér-knappen, for eksempel — giver enheden tilbage én gang.
- **Reservationer udløber efter minutter.** Fem til ti er det almindelige. Forlæng reservationen én gang, når betalingen starter, så et langsomt kort ikke mister den enhed, det er ved at betale for.
- **Begræns antallet af enheder pr. køber ved reservationen,** så én konto ikke kan holde hele lageret, mens den tænker sig om.

Under det hele ligger én linje, den mest nyttige i designet:

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

Den gælder efter hver transaktion, én forespørgsel tjekker den, og når den holder op med at gælde, har noget oversolgt eller tabt en enhed — før en kunde siger det.

Når én database ikke er nok

Nogle gange er den reelt ikke det: mange lejere, mange salg på én gang, flere regioner. Den struktur, der skalerer, holder fast i én regel — hver vare har præcis én ejer af sit lager, en lagertjeneste, og intet andet bestemmer — og sharder den tjeneste efter `(tenant_id, item_id)`, så én vares skrivninger aldrig spænder over to shards. De største lejere får deres egne celler, så deres salg ikke kan sulte andres ud. Kun reservationen behøver at være synkron; alt efter den kan være en besked.

Til den varme vare inde i en shard er der fire værktøjer, og hvert trækker i ét af håndtagene ovenfor.

En port foran. En tæller i Redis, nedskrevet af et Lua-script, så tjek og nedskrivning er ét trin, lukker omtrent lige så mange købere ind, som der er enheder, og afviser alle andre på mikrosekunder. Den er ikke dommeren. Redis replikerer asynkront, så efter en failover kan den nye primære have glemt nogle adgange og lukket lidt for mange ind. Databasens betingede `UPDATE` bestemmer stadig, og en afstemningsproces nulstiller porten ud fra databasens tal.

Del lageret. Ti tusind enheder kan være hundrede spande med hundrede, hver sin række eller sin shard, med hver køber sendt til én: hundrede låse i stedet for én. Idéen er gammel — O'Neils escrow-transaktioner fra 1986, så Barbará og Garcia-Molinas demarkationsprotokol, så Balegas og kollegers begrænsede tællere — og dens ene reelle pris er halen. En købers spand løber tør, mens andre stadig har lager, så prøv enten en anden spand eller flyt resterne, og saml dem til én igen, når salget nærmer sig sin afslutning. Samme idé giver hver region sin egen kvote, så de fleste salg aldrig krydser et hav for at spørge.

Én skriver pr. vare. Læg hver forespørgsel på en vare på én Kafka-partition, læst af én forbruger, der holder tallet i hukommelsen og committer sine afgørelser i batches. Én tråd og ingen låse: LMAX-formen. Forespørgsler bærer et id, og skriveren smider én væk, den allerede har afgjort, for en kø leverer to gange.

Enheder som rækker. Når enhederne er få og unikke — pladser, nummererede udgaver — så opret én række til hver på forhånd og gør krav på dem med `SKIP LOCKED` som ovenfor, eller hold enhedernes id'er i et Redis-sæt og tag én med `SPOP`. Så er der ingen tæller tilbage at slås om.

Resten af checkout

Betaling, ordre og kvittering er separate systemer, og ingen transaktion spænder over dem. Tofaset commit er lærebogens svar og det forkerte her: det blokerer, mens en deltager er langsom, det lægger rundture til den varmeste sti, og betalingsudbydere deltager ikke i det. En saga klarer opgaven, en række lokale trin, hver med et trin, der fortryder det:

```
1. reservér lager        fortryd: frigiv reservationen
2. afventende ordre      fortryd: annullér ordren
3. forlæng reservationen (udløber af sig selv)
4. autorisér betaling    fortryd: annullér den
5. bekræft HELD → SOLD
6. hæv betalingen
```

At autorisere først og hæve sidst betyder, at en betaling, der kommer efter sin reservations udløb, annulleres i stedet for at refunderes: pengene flyttede sig aldrig. Hvert trin skriver sin besked i en outbox-tabel i samme transaktion som den ændring, den melder, og change data capture udgiver den, så intet går tabt mellem gemt og sendt. Hver forbruger er idempotent, med ordren som nøgle. Beskeder kommer mindst én gang, og idempotens gør det til i praksis én gang.

En distribueret lås hører ingen steder hjemme her. Kleppmanns pointe er, at en låseholder kan gå i stå forbi sin lejeperiode — en garbage collection, en virtuel maskine, der hænger — og blive ved med at skrive, og kuren er et fencing-token, som lageret tjekker. Men et lager, der kan tjekke et token, kan tjekke `available >= :qty`, og så har det aldrig haft brug for låsen.

Sæt varen på pause, oversælg den aldrig

Konsistens og tilgængelighed vælges pr. spørgsmål, ikke pr. system. “Må jeg sælge denne enhed?” skal være konsistent: når ejeren af en vares lager ikke kan nås, holder den vare op med at sælge. Alt andet — kataloget, mærket med “3 tilbage”, ordrehistorikken — kan forblive tilgængeligt og være en smule forældet.

Det stilleste oversalg kommer fra en failover. Med asynkron replikering bekræfter en primær et salg og dør, før dens replika har det; replikaen forfremmes, salget er væk, og enheden sælges igen. Rettelsen er at gøre et salg holdbart på mere end én maskine, før det bekræftes: en synkron standby i PostgreSQL (`synchronous_commit` med `synchronous_standby_names`) eller et lager bygget på Raft. Det koster latens på hver commit, hvilket forlænger præcis den holdetid, loftet divideres med, og batching og porten køber den tilbage. Afgræns også pausen: en shard, der fejler, skal stoppe varerne på den, ikke butikken.

Bevis det

Intet af det her skal tages på tro, heller ikke vores.

- **Hold øje med invarianten.** Kør `available + held + sold = total` efter en tidsplan, tegn den, og alarmér ved enhver afvigelse. Nul oversalg er et serviceniveaumål med en alarm, ikke et håb.
- **Fremprovokér kapløbet.** Ti tusind samtidige købere til hundrede enheder, i testsuiten, der kræver præcis hundrede solgte og intet tal under nul.
- **Ødelæg ting under kapløbet.** Dræb den primære, kap netværket, sæt en proces på pause — i Jepsens ånd — og tjek invarianten bagefter.
- **Belastningstest afvisningerne.** Udsolgt-stien skal forblive hurtig, når den er 99 procent af trafikken, og kanten skal kunne tage den.
- **Saml målingerne på ét dashboard:** låseventetider, udløbne reservationer og afstanden mellem portens tal og databasens.

I hvilken rækkefølge

- Mål først. `pg_stat_statements` og låseventetiderne siger, om den varme række overhovedet er problemet.

- Gør skrivningen til én betinget sætning.
- Skær belastningen ned: et venteværelse, en port, et cachet "udsolgt".
- Giv lageret sin egen tjeneste og sin egen ejer.
- Shard efter `(tenant_id, item_id)`, og kun hvis de første fire ikke var nok.
- Tilføj en tilstand til varme varer, spande eller én skriver, til det salg, du ved kommer.
- Regioner til sidst, med kvoter.

Hvad vi ville sige til den næste

- **Sharding svarer på et andet spørgsmål.** Det spreder mange rækker, og et oversolgt salg er én række. Find ud af, hvilken af delene du har, før du bygger.
- **Gør tjek og nedskrivning til én sætning.** Det meste af afstanden mellem fint og oversolgt er tiden mellem at læse et tal og at skrive det.
- **Det meste af trafikken er svaret nej.** Gør afvisning til det billigste, systemet gør.
- **Sæt på pause, før du oversælger.** En vare, der holder op med at sælge i tredive sekunder, er en undskyldning. En vare, der er solgt to gange, er en refusion, en supportkø og en kunde, der husker det.
- **Skriv invarianten ned, og hold øje med den.** Den er én linje, og den er det eneste, der siger, at designet virker i drift og ikke kun i review.

Kilder

- PostgreSQL: Read Committed-isolation
- PostgreSQL: `FOR UPDATE` og `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: design af partitionsnøgler
- DynamoDB: partitioner og datafordeling
- microservices.io: Saga
- microservices.io: Transactional outbox
- Redis: replikering
- Jepsen

Alle blogindlæg

Mål, før I ændrer

Medbring et repræsentativt datasæt, de spørgsmål det skal besvare, og den aktuelle flaskehals. Vi kan aftale målinger af kvalitet, gennemløb og gendannelse.

Se, hvad det koster Kontakt os →

Drøft jeres observationer

Drøft »At sælge den sidste billet præcis én gang« i forbindelse med jeres eget projekt. Fortæl, hvad I har observeret, og hvad et nyttigt resultat ville være.

Se vores ydelser

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

Tag denne side med dig

Visitkort

Lavet med kærlighed af D.

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

Copyright © 2025 – nu · Engineer ApS · Alle rettigheder forbeholdes.

Udskrevet fra https://data.engineer.company/da/notes/selling-limited-stock-without-overselling/