Engineer Plattform

Å selge den siste billetten nøyaktig én gang

13. oktober 2026

Et lynsalg setter ti tusen enheter foran en million mennesker samtidig. Den interessante feilen er ikke at butikken går ned. Det er at butikken holder seg oppe og selger ti tusen og førti. Instinktet som følger — databasen er for liten, shard den — svarer på et spørsmål ingen stilte. Databasen var aldri for stor. Hver kjøper skrev til den samme raden.

Her tar vi problemet fra hverandre før det når et prosjekt: hvor taket kommer fra, den ene setningen som fjerner det meste av det, og hva du bygger når én database virkelig ikke er nok.

Bruk erfaringen i systemet deres

Arbeidet bak «Å selge den siste billetten nøyaktig én gang» kan bidra til deres neste endring. Vi kan undersøke lignende oppførsel i programvaren, dataene eller infrastrukturen deres. En struktur som samsvarer med hvordan virksomheten faktisk fungerer, slik at spørsmålene du trenger svar på senere, fortsatt er mulige å stille.

Se tjenestene våre Ta kontakt →

Følg resonnementet og grensene

Les resonnementet og begrensningene i notatet. Vi kan teste om samme diagnose holder i deres miljø. Bygde tilgangsrettigheter for Stripe og appkjøp — 11 tabeller som avgjør hvem som får se hva

Bygde tilgangsrettigheter for Stripe og appkjøp Ta kontakt →

Én rad, alle kjøpere

Tjue millioner rader fordelt over en klynge er et løst problem. Én rad som hver forespørsel må endre, er det ikke, og flere maskiner endrer ingenting ved det, fordi raden bor på én av dem. Distribuerte datalagre sier det rett ut: DynamoDB gir én enkelt partisjon rundt 1 000 skriveenheter i sekundet, og hver skriving til ett element havner på partisjonen til det elementet, uansett hvor mange andre det finnes.

Det ligger et teorem bak denne formen. Bailis og kolleger kaller en invariant invariant-konfluent når replikaer kan ta imot skrivinger på egen hånd og likevel flettes sammen til en gyldig tilstand. `stock >= 0` er skoleeksempelet på en som ikke er det: to replikaer som hver ser én enhet igjen, vil hver selge den, og ingen fletting kan ta det andre salget tilbake. Å aldri overselge krever koordinering. Det eneste spørsmålet er hvor mye av den hvert salg betaler for.

Tallet som avgjør det

Taket for én vare er omtrent hvor mange salg én lås bærer, delt på hvor lenge låsen holdes:

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

Hva som holder raden	Tak per rad
Lås, så kall til betaling	~2/s
Lås, så applikasjonslogikk	~100–200/s
Én betinget `UPDATE`	~500–2 000/s
Batcher på 100 per commit	~50 000/s
Én skriver i minnet	~100 000+/s

Dette er størrelsesordener, ikke ytelsesmålinger; maskinvaren flytter dem. Det den ikke flytter, er formen. Det meste av en kort lås er commiten som venter på at loggen skal nå disken, og det er derfor batching vinner med to størrelsesordener: hundre salg deler én diskskriving.

Det gir fire spaker, og hvert design nedenfor trekker i én av dem:

- **Hold låsen kortere** — én setning, ingenting over nettverket inne i den.
- **Gjør mer per lås** — samle nedtellingene i batcher.
- **Ha flere låser** — fordel beholdningen på flere nøkler.
- **Send færre forespørsler til låsen** — avvis de fleste kjøperne før de når den.

Den siste betyr mer enn den høres ut som. Med en million kjøpere og ti tusen enheter må 990 000 forespørsler mislykkes. Avvisninger er det meste av trafikken, så «utsolgt» må være det billigste svaret systemet gir: et venterom foran salget, en port som slipper inn omtrent like mange kjøpere som det finnes enheter, et `sold_out` -flagg som CDN-et serverer uten å spørre noen, og en hastighetsbegrensning per konto.

Få den ene databasen i orden først

Versjonen som går i sneglefart, eller som overselger, ser som regel slik ut:

```
BEGIN;
SELECT available FROM inventory
 WHERE item_id = :item FOR UPDATE;
-- applikasjonen sjekker tallet,
-- og kaller kanskje en betalingsleverandør
UPDATE inventory SET available = :new
 WHERE item_id = :item;
COMMIT;
```

Låsen er riktig. Det den holdes over, er feil: hvert millisekund applikasjonen bruker mellom `SELECT` og `UPDATE` , er et millisekund den neste kjøperen venter. Uten `FOR UPDATE` er det verre. To kjøpere leser 1, begge skriver 0, og én enhet er solgt to ganger.

Løsningen gjør sjekken og nedtellingen til én setning:

```
UPDATE inventory
   SET available = available - :qty
 WHERE tenant_id = :tenant
   AND item_id = :item
   AND available >= :qty
RETURNING available;
-- én rad tilbake: reservert. ingen: utsolgt.
```

I PostgreSQL på READ COMMITTED venter en annen kjøper som når raden mens den første holder den, og evaluerer så `WHERE` -leddet på nytt mot raden den første har committet. Tok det salget den siste enheten, slår betingelsen feil, og den andre kjøperen får ingen rad tilbake — utsolgt — i stedet for et negativt tall.

På REPEATABLE READ eller SERIALIZABLE får den andre kjøperen i stedet en serialiseringsfeil, som er trygt, men gjør kampen om raden til nye forsøk; READ COMMITTED er nivået denne setningen er skrevet for. InnoDB i MySQL kommer til samme svar, fordi en `UPDATE` der leser den sist committede raden på alle nivåer. Legg også til begrensningen, slik at en feil et annet sted feiler høylytt i stedet for å selge beholdning som ikke finnes:

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

Så resten av arbeidet på den ene databasen, omtrent i rekkefølge etter gevinst:

- **Ingenting eksternt inne i transaksjonen.** Ingen betalingskall, ingen HTTP-forespørsel, ingen publisert melding. Det hører hjemme før transaksjonen eller etter den.
- **Rør den hete raden sist.** Sett inn ordren og alt annet først, slik at låsen på raden som flere transaksjoner konkurrerer om, bare varer gjennom den siste setningen og commiten.
- **Ingen optimistisk låsing på en het teller.** En versjonskolonne gjør kampen om raden til nye forsøk, og under et salg er det de nye forsøkene som er lasten.
- **Hent lesingene fra et annet sted.** «12 igjen» på produktsiden kan komme fra en replika eller en hurtigbuffer som ligger et sekund etter. Bare salget trenger primærnoden.
- **Gi lagerbeholdningen en egen database,** slik at et salg aldri venter bak katalogen på den samme disken.
- **Samle tilkoblingene i en pool.** Hundre tusen kjøpere bør bli noen dusin tilkoblinger gjennom PgBouncer, ikke tusenvis av databaseprosesser som venter på én lås.

Seter har en annen form, fordi hver enhet er unik, og de trenger en annen setning. Hvert sete er en rad, og en kjøper hopper over radene en annen transaksjon holder, i stedet for å vente bak 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 utløpt reservasjon er rett og slett ledig igjen, så det trengs ingen oppryddingsjobb for at svaret skal bli riktig. Og `now()` er databasens klokke, så applikasjonsservere med klokker som ikke stemmer overens, kan ikke være uenige om når en reservasjon gikk ut.

For mange butikker er denne delen hele løsningen. Gjør den før noe av det som følger, fordi sharding er en enveisdør: lett å gå gjennom, dyr å gå tilbake gjennom.

Reserver først, betal etterpå

En kjøper som tar den siste enheten og så forlater kortskjemaet, har ikke kjøpt den. Et salg er derfor ikke én skriving, men en kort livssyklus:

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

- **Reservasjonen er knyttet til forespørselen, ikke til forsøket.** Hver utsjekking har en idempotensnøkkel, og reservasjonstabellen har `UNIQUE (tenant_id, idempotency_key)`, slik at et klikk som prøves på nytt, får tilbake reservasjonen det allerede har, i stedet for en enhet til. Én PostgreSQL-detalj biter her: `INSERT ... ON CONFLICT DO NOTHING RETURNING id` returnerer ingenting når raden fantes fra før, så «ingen rad tilbake» betyr «slå den opp», ikke «feilet».
- **Hver overgang er betinget.** Å bekrefte et salg flytter en reservasjon fra `held` til `sold` bare der den fortsatt er `held`, og en frigivelse fører beholdningen tilbake bare hvis dens egen oppdatering endret nøyaktig én rad. En frigivelse som kjører to ganger — tidsavbruddet og avbryt-knappen, for eksempel — gir enheten tilbake én gang.

- **Reservasjoner utløper etter minutter.** Fem til ti er vanlig. Forleng reservasjonen én gang når betalingen starter, slik at et tregt kort ikke mister enheten det betaler for.
- **Sett et tak på enheter per kjøper ved reservasjonen,** slik at én konto ikke kan holde hele beholdningen mens den bestemmer seg.

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

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

Den holder etter hver transaksjon, én spørring sjekker den, og når den slutter å holde, har noe oversolgt eller mistet en enhet — før en kunde sier fra.

Når én database ikke er nok

Noen ganger er den virkelig ikke det: mange leietakere, mange salg samtidig, flere regioner. Strukturen som skalerer, holder fast ved én regel — hver vare har nøyaktig én eier av beholdningen sin, en lagertjeneste, og ingenting annet bestemmer — og sharder den tjenesten etter `(tenant_id, item_id)`, slik at skrivingene for én vare aldri spenner over to shards. De største leietakerne får egne celler, slik at salget deres ikke kan sulte ut noen andres. Bare reservasjonen må være synkron; alt etter den kan være en melding.

For den hete varen inne i en shard finnes det fire verktøy, og hvert av dem trekker i én av spakene over.

En port foran. En Redis-teller, som telles ned av et Lua-skript slik at sjekken og nedtellingen er ett steg, slipper inn omtrent like mange kjøpere som det finnes enheter og avviser alle andre på mikrosekunder. Den er ikke dommeren. Redis replikerer asynkront, så etter en failover kan den nye primærnoden ha glemt noen innslipp og latt noen for mange komme gjennom. Databasens betingede `UPDATE` avgjør fortsatt, og en avstemmingsjobb nullstiller porten ut fra databasens tall.

Del opp beholdningen. Ti tusen enheter kan være hundre bøtter med hundre, hver sin egen rad eller sin egen shard, og hver kjøper sendes til én av dem: hundre låser i stedet for én. Ideen er gammel — O'Neils escrow-transaksjoner i

1986, så demarkasjonsprotokollen til Barbará og Garcia-Molina, så de begrensede tellerne til Balegas og kolleger — og den eneste reelle kostnaden er halen. Bøtten til en kjøper går tom mens andre fortsatt har beholdning, så prøv enten en annen bøtte eller flytt restene, og slå dem sammen til én igjen mot slutten av et salg. Den samme ideen gir hver region en egen kvote, slik at de fleste salg aldri krysser et hav for å spørre.

Én skriver per vare. Legg hver forespørsel for en vare på én Kafka-partisjon, lest av én konsument som holder tellingen i minnet og committer beslutningene sine i batcher. Én tråd og ingen låser: LMAX-formen. Forespørsler har en id, og skriveren forkaster en den allerede har avgjort, fordi en kø leverer to ganger.

Enheter som rader. Når enhetene er få og unike — seter, nummererte utgaver — opprett én rad for hver på forhånd og reserver dem med `SKIP LOCKED` som over, eller hold enhets-id-ene i et Redis-sett og hent én med `SPOP` . Da finnes det ingen teller igjen å slåss om.

Resten av utsjekkingen

Betaling, ordre og kvittering er separate systemer, og ingen transaksjon spenner over dem. Tofase-commit er læreboksvaret og feil svar her: den blokkerer så lenge en deltaker er treg, den legger rundturer til den heteste stien, og betalingsleverandører deltar ikke i den. En saga gjør jobben, en rekke lokale steg, hvert med et steg som angrer det:

```
1. reserver beholdning     angre: frigi reservasjonen
2. ventende ordre          angre: kanseller ordren
3. forleng reservasjonen   (utløper av seg selv)
4. autoriser betalingen    angre: annuller den
5. bekreft HELD → SOLD
6. trekk betalingen
```

Å autorisere først og trekke sist betyr at en betaling som kommer etter at reservasjonen har utløpt, annulleres i stedet for å refunderes: pengene flyttet seg aldri. Hvert steg skriver meldingen sin til en outbox-tabell i den samme transaksjonen som endringen den kunngjør, og change data capture publiserer den, så ingenting går tapt mellom lagret og sendt. Hver konsument er idempotent, med ordren som nøkkel. Meldinger kommer minst én gang, og idempotens gjør det i praksis til én gang.

En distribuert lås hører ikke hjemme noe sted i dette. Kleppmanns poeng er at den som holder en lås, kan bli stående på pause forbi leieperioden sin — en søppelinnsamling, en virtuell maskin som henger — og fortsette å skrive, og kuren er et fencing-token som datalageret sjekker. Men et datalager som kan sjekke et token, kan sjekke `available >= :qty`, og da trengte det aldri låsen.

Sett varen på pause, overselg den aldri

Konsistens og tilgjengelighet velges per spørsmål, ikke per system. «Kan jeg selge denne enheten?» må være konsistent: når eieren av en vares beholdning ikke kan nås, slutter den varen å selges. Alt annet — katalogen, «3 igjen»-merket, ordrehistorikken — kan forbli tilgjengelig og være litt utdatert.

Det stilleste oversalget kommer fra en failover. Med asynkron replikering bekrefter en primærnode et salg og dør før replikaen har fått det; replikaen forfremmes, salget er borte, og enheten selges på nytt. Løsningen er å gjøre et salg varig på mer enn én maskin før det bekreftes: en synkron standby i PostgreSQL (`synchronous_commit` med `synchronous_standby_names`), eller et datalager bygd på Raft. Det koster latens på hver commit, noe som forlenger nøyaktig den holdetiden for låsen som taket deles på, og batching og porten kjøper den tilbake. Avgrens pausen også: en shard som feiler, skal stoppe varene på den, ikke butikken.

Å bevise det

Ingenting av dette bør tas på tro, heller ikke vårt.

- **Overvåk invarianten.** Kjør `available + held + sold = total` etter en tidsplan, vis den i en graf og send varsel ved ethvert avvik. Null oversalg er et tjenestenivåmål med en alarm, ikke et håp.
- **Fremprovoser kappløpet med vilje.** Ti tusen samtidige kjøpere til hundre enheter, i testsuiten, med automatiske kontroller av at nøyaktig hundre er solgt og at ingen tall er under null.
- **Ødelegg ting under kappløpet.** Drep primærnoden, kutt nettverket, sett en prosess på pause — i Jepsen-stil — og sjekk invarianten etterpå.
- **Lasttest avvisningene.** Utsolgt-stien må holde seg rask når den er 99 % av trafikken, og kanten må ta imot den.

- **Samle målingene på ett dashbord:** låseventing, utløpte reservasjoner og avstanden mellom portens telling og databasens.

I hvilken rekkefølge

- Mål først. `pg_stat_statements` og låseventingen sier om den hete raden i det hele tatt er problemet.
- Gjør skrivingen til én betinget setning.
- Kvitt deg med last: et venterom, en port, et hurtigbufret «utsolgt».
- Gi lagerbeholdningen sin egen tjeneste og sin egen eier.
- Shard etter `(tenant_id, item_id)`, og bare hvis de fire første ikke var nok.
- Legg til en modus for hete varer, bøtter eller én enkelt skriver, for salget du vet kommer.
- Regioner til slutt, med kvoter.

Det vi ville sagt til den neste

- **Sharding svarer på et annet spørsmål.** Den fordeler mange rader, og et oversolgt salg er én rad. Finn ut hvilket av dem du har, før du bygger.
- **Gjør sjekken og nedtellingen til én setning.** Det meste av avstanden mellom i orden og oversolgt er tiden mellom å lese et tall og å skrive det.
- **Det meste av trafikken er svaret nei.** Gjør avvisning til det billigste systemet gjør.
- **Sett på pause før du overselger.** En vare som slutter å selges i tretti sekunder, er en unnskyldning. En vare som er solgt to ganger, er en refusjon, en supportkø og en kunde som husker det.
- **Skriv ned invarianten og overvåk den.** Den er én linje, og den er det eneste som sier at designet virker i produksjon og ikke bare i gjennomgangen.

Kilder

- PostgreSQL: isolasjonsnivået Read Committed
- 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: utforming av partisjonsnøkler
- DynamoDB: partisjoner og datafordeling
- microservices.io: Saga
- microservices.io: Transactional outbox
- Redis: replikering
- Jepsen

Alle blogginnlegg

Mål før dere endrer

Ta med et representativt datasett, spørsmålene det skal besvare og dagens flaskehals. Vi kan avtale målinger av kvalitet, gjennomstrømning og gjenoppretting.

Se hva det koster Ta kontakt →

Drøft observasjonene deres

Drøft «Å selge den siste billetten nøyaktig én gang» i sammenheng med deres eget prosjekt. Fortell hva dere observerte og hva et nyttig resultat ville være.

Se tjenestene våre

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

Ta med deg denne siden

Visittkort

Laget med kjærlighet av D.

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

Opphavsrett © 2025 – nå · Engineer ApS · Alle rettigheter forbeholdt.

Skrevet ut fra https://platform.engineer.company/nb/notes/selling-limited-stock-without-overselling/