Engineer

Optimaliser databasen før du sharder den

14. oktober 2026

Når en database blir treg under last, er det første forslaget som regel flere maskiner. Noen ganger er det riktig. Oftere er den trege delen en håndfull setninger ingen har lest ennå, og å spre dataene over flere servere sprer de setningene over mer maskinvare uten å gjøre én eneste av dem billigere. Et problem som ser uløselig ut, er ofte én spørringsplan unna et elegant svar, og horisontal skalering velges iblant nettopp for at ingen skal måtte lese databasen du allerede har.

Dette er rekkefølgen vi arbeider i før vi deler opp noe: se, stram inn, ta lesingene bort fra primærnoden, partisjoner tabellene rundt det hete punktet, og shard først når én maskin er full av godt arbeid. Lynsalget, der hver kjøper skriver til én rad, er gjennomgått i Å selge den siste billetten nøyaktig én gang. Dette notatet er metoden rundt det, og den siste delen tar for seg det motsatte forslaget: selg fra hver server, og be om unnskyldning for enhetene som ble solgt to ganger.

Bruk erfaringen i systemet deres

Arbeidet bak «Optimaliser databasen før du sharder den» 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. Designet og driftet 6 ETL/ELT-pipelines — 6 ETL-pipelines på tvers av tre databasemotorer

Designet og driftet 6 ETL/ELT-pipelines Ta kontakt →

Se først

PostgreSQL registrerer allerede det meste det er å vite. Før du endrer noe, still den fire spørsmål.

Hvilke setninger koster mest totalt? `pg_stat_statements` , en utvidelse som lastes via `shared_preload_libraries` , tar vare på hver normaliserte setning med antall kall og tidsbruk. Sorter etter total, ikke etter gjennomsnitt: en spørring på 2 ms som kjøres en million ganger i timen, koster mer enn en rapport på to sekunder som kjøres to ganger.

```
SELECT calls,
       round(total_exec_time) AS total_ms,
       round(mean_exec_time::numeric, 1)
         AS mean_ms,
       left(query, 60) AS query
  FROM pg_stat_statements
 ORDER BY total_exec_time DESC
 LIMIT 10;
```

Hva gjør den øverste egentlig? `EXPLAIN (ANALYZE, BUFFERS)` kjører setningen og viser planen den valgte, radene hvert steg faktisk returnerte mot radene det forventet, og hvor mange sider den leste. En skanning av en million rader for å returnere ti, eller et estimat som bommer med en faktor hundre, er som regel hele historien. `ANALYZE` utfører det den forklarer, så pakk en skriving inn i `BEGIN` og `ROLLBACK` .

Hvem venter, og på hvem?

```
SELECT pid, wait_event_type, wait_event,
       pg_blocking_pids(pid) AS blocked_by,
       left(query, 60) AS query
  FROM pg_stat_activity
 WHERE wait_event_type = 'Lock';
```

Rader her i et travelt minutt betyr at transaksjoner venter på hverandre, ikke på maskinvare, og en større maskin fjerner ikke kravet om at oppdateringer av én felles rad utføres én om gangen. Flere applikasjonsservere kan føye skrivende

klienter til den køen.

Arbeider serveren, eller sjonglerer den? Tell de aktive øktene i `pg_stat_activity` mot kjernene. Når mange ganger flere er aktive enn det finnes kjerner, bruker serveren tiden sin på å bytte mellom dem, og en tilkoblingspool løser det før noen maskinvare gjør det. Les deretter `n_dead_tup` i `pg_stat_user_tables`: en tabell med flere døde rader enn levende blir skannet gjennom sitt eget søppel, og det er autovacuum som henger etter, ikke en database som er for liten.

Skriv ned det du finner. Det er utgangspunktet hvert senere steg måles mot, og det ærlige svaret på «må vi sharde?» ligger ofte allerede der.

Stram inn det som er der

Omtrent i rekkefølge etter gevinst for innsatsen.

Indekser som passer til spørsmålene. Planene ovenfor navngir skanningene. En indeks på kolonnene en het spørring filtrerer på, gjør en skanning om til et oppslag, og en partiell indeks inneholder bare radene en spørring kan være ute etter — enheter som fortsatt er til salgs, for eksempel — så den krymper etter hvert som et salg skrider frem, i stedet for å vokse:

```
CREATE INDEX units_unsold
    ON units (item_id)
 WHERE NOT sold;
```

Hver indeks betales også ved hver skriving, så fjern dem der `idx_scan` i `pg_stat_user_indexes` blir stående på null.

Typer som sier hva verdien er. Skjemaer skrevet i en fart, av et menneske eller en modell, griper etter `TEXT` der en `SMALLINT` eller en `BOOLEAN` ville holdt. En `BOOLEAN` er én byte, en `SMALLINT` to, og `'yes'` lagret som tekst er fire. PostgreSQL fyller også ut slik at hver kolonne med fast bredde havner på sin egen justeringsgrense, så en kolonnerekkefølge som er valgt tilfeldig, koster også byte. Én lagerenhet, skrevet i en fart:

```
CREATE TABLE units (
  sold        text,  -- 'yes' eller 'no'
  id          bigint,
  reserved    text,  -- 'yes' eller 'no'
  reserved_at timestamptz,
  item_id     bigint,
  order_id    bigint
);
```

Og den samme enheten med typer som sier hva verdiene er, de bredeste først:

```
CREATE TABLE units (
  id          bigint PRIMARY KEY,
  reserved_at timestamptz,
  order_id    bigint,
  item_id     integer NOT NULL,
  sold        boolean NOT NULL DEFAULT false,
  reserved    boolean NOT NULL DEFAULT false
);
```

På PostgreSQL 18 måler den første raden 72 byte og den andre 54, en fjerdedel mindre, så mer av tabellen får plass i minnet. Typene avviser også verdier som `'maybe'`, som applikasjonen ellers måtte forsvare seg mot, og en `CHECK`-skranke gjør det samme for verdiområder: `quantity smallint CHECK (quantity > 0)`.

Færre rundturer, og arbeidet der dataene er. Hver setning applikasjonen sender, koster en rundtur og en parsing. En løkke som leser hundre rader og oppdaterer dem én etter én, er hundre og én rundturer der én mengdebasert setning ville holdt, og en side som henter hver enhet av en vare for å telle de ledige, sender tusenvis av rader for å regne ut ett tall. Regelen vi bruker: det som må se dataene, går til databasen — filtrering, telling, en skriving som avhenger av hva som er der — og forretningsregler blir i applikasjonen, fordi databasens CPU er den vanskeligste i systemet å få mer av.

Oppdateringer som blir på siden sin. En rad som endres mange ganger i minuttet, etterlater en ny versjon av seg selv hver gang. Når en oppdatering ikke endrer noen indeksert kolonne og siden har plass, beholder PostgreSQL den nye versjonen på samme side og rører ingen indeks i det hele tatt: en heap-only tuple, eller HOT-oppdatering. Hold derfor kolonnene en hyppig oppdatering endrer, utenfor indeksene, og la det være plass igjen på sidene:

```
ALTER TABLE units SET (fillfactor = 90);
```

Innstillingen gjelder sider som skrives fra da av, så sett den før radene lastes inn. På tabellen ovenfor var alle 2 000 reservasjoner som satte `reserved`, `reserved_at` og `order_id`, HOT, og av 2 000 oppdateringer som satte `sold`, som den partielle indeksen leser, var ingen HOT. Salg skjer én gang per enhet; reservasjoner kan skje mange ganger. `n_tup_hot_upd` mot `n_tup_upd` i `pg_stat_user_tables` viser forholdet på et system i drift.



Én rad sier hvem som var først

En enhet går til den første kjøperen hvis setning når den, og reservasjonen løper ut etter databasens klokke. To kolonner bærer den: `reserved`, en boolean, og `reserved_at`, tidspunktet den ble satt.

```
UPDATE units
   SET reserved = true,
       reserved_at = now(),
       order_id = :order
 WHERE id = :unit
   AND NOT sold
   AND (NOT reserved
        OR reserved_at
           < now() - interval '10 minutes')
RETURNING id;
```

Én rad tilbake: denne kjøperen har den. Ingen: noen andre var først, eller den er solgt. To reservasjoner på samme enhet står i kø på radlåsen dens, og på READ COMMITTED, standarden i PostgreSQL, sjekker den andre `WHERE`-leddet på nytt mot raden den første committet, så nøyaktig én av dem får en rad tilbake. En reservasjon som er eldre enn ti minutter, regnes som ledig, så ingenting trenger å rydde den bort for at svaret skal bli riktig.

Det gjør boolean-verdien alene til et ufullstendig svar. `reserved` forblir sann etter at en reservasjon har løpt ut, og bare de to kolonnene sammen sier om en enhet er reservert, så hver spørring som spør, må spørre om begge. En periodisk

opprydding som nullstiller utløpte reservasjoner, holder tabellen ærlig for alle som leser `reserved` alene, men korrektheten venter aldri på den. En skranke holder paret konsistent:

```
ALTER TABLE units
  ADD CONSTRAINT reserved_has_time
  CHECK (NOT reserved
         OR reserved_at IS NOT NULL);
```

Billettnotatet lagrer fristen, `held_until`, i stedet for starttidspunktet; det er det samme designet. Når kjøperen velger enheten — et sete, et nummerert trykk — nevner setningen den, som her. Når en hvilken som helst enhet duger, velger det notatets `SKIP LOCKED`-variant en som ingen andre holder.

Lesinger bort fra primærnoden

Mesteparten av trafikken i en nettbutikk er lesing: produktsiden, «3 igjen», handlekurven, ordrehistorikken. Ingenting av det trenger primærnoden, den ene maskinen som må holdes fri for skrivinger. En strømmende replika tar de lesingene, og å legge til en er konfigurasjon, der en shard er et prosjekt.

To forbehold følger med. En replika ligger litt etter, så en kjøper som nettopp har reservert en enhet og så leser handlekurven sin fra den, ser kanskje ikke reservasjonen: les det en bruker nettopp har skrevet, fra primærnoden, eller send den brukeren til primærnoden i noen sekunder etter hver skriving. Og en lang rapport på en replika kan bli avbrutt når den holder igjen endringene replikaen spiller av. PostgreSQL-dokumentasjonen kaller dette spørringskonflikter og lister opp innstillingene som veier replikaforsinkelse opp mot dem.

Den billigste lesingen er en som aldri når en database. Når hver enhet av en vare er reservert, kan ingen kjøpe den før en reservasjon løper ut eller frigis, og databasen kan si nøyaktig når det er:

```
SELECT count(*) FILTER (
         WHERE NOT reserved
            OR reserved_at
               < now() - interval '10 minutes')
```

```
        AS free,
      min(reserved_at) + interval '10 minutes'
        AS until
  FROM units
 WHERE item_id = :item
   AND NOT sold;
```

Når `free` er null, er `until` øyeblikket da den tidligste reservasjonen løper ut: legg «fullt reservert» i hurtigbufferen med det som utløpstid, aldri lenger, og hver besøkende fram til da blir avvist av hurtigbufferen i stedet for av databasen. Når `free` er null og `until` er tom, er hver enhet solgt, og svaret er endelig. En reservasjon som frigis tidlig — et avvist kort, en forlatt handlekurv — sletter oppføringen i hurtigbufferen, så enheten er straks til salgs igjen.

Hurtigbufferen sier bare nei. Et ja kommer alltid fra den betingede `UPDATE` -setningen ovenfor, så en utdatert hurtigbuffer kan koste et salg i noen sekunder og kan aldri selge en enhet to ganger. Det er billettnotatets hurtigbufrede «utsolgt», med en frist.



Partisjoner tabellene rundt den hete raden

Partisjonering deler én tabell i flere etter en nøkkel — en måned, en leietaker — bak ett navn. Det er verdt å si rett ut hva den ikke gjør: den gjør ikke én rad som det kjempes om, raskere. Den raden bor fortsatt i én partisjon, på én maskin, bak én lås.

Det den gjør, er å holde tabellene rundt det hete punktet små. Ordrer, reservasjonshistorikk og hendelser vokser i det uendelige, og de fleste spørringer vil ha den nyeste delen:

```
CREATE TABLE orders (
  id          bigint NOT NULL,
  created_at  timestamptz NOT NULL
              DEFAULT now(),
  item_id     integer NOT NULL,
  quantity    smallint NOT NULL
              CHECK (quantity > 0),
  paid        boolean NOT NULL DEFAULT false,
  PRIMARY KEY (id, created_at)
) PARTITION BY RANGE (created_at);
```

```
CREATE TABLE orders_2026_10
  PARTITION OF orders
  FOR VALUES FROM ('2026-10-01')
              TO ('2026-11-01');
```

En spørring som angir et datointervall, leser bare partisjonene innenfor det, og denne månedens indekser holder seg små nok til å ligge i minnet. Å kvitte seg med gamle data er ikke lenger en `DELETE` av millioner av rader, med oppblåsingen og vacuumkjøringen som følger, men én setning:

```
ALTER TABLE orders
  DETACH PARTITION orders_2025_10;
```

Primærnøkkelen må inneholde partisjonsnøkkelen, og derfor er den `(id, created_at)` . En spørring som ikke angir nøkkelen, leser hver partisjon, så partisjoner etter det spørringene allerede filtrerer på.

Når sharding er fornuftig

Det finnes én grense sharding ikke kan flytte: hver vare har én eier. En enhet kan ikke reserveres på to servere samtidig, så sharding flytter hele varer, aldri halvparten av én. Når én enkelt vare vokser fra maskinen sin, deler billettnotatets bøtter beholdningen dens i mindre biter, og hver bit har igjen nøyaktig én eier.

Innenfor det kaller vi sharding fornuftig når én av disse er sann, og ikke før:

- **Én maskin er full av godt arbeid.** Setningene er allerede slanke — én betinget skriving hver, ingenting eksternt inne i en transaksjon, oppdateringer som blir på siden sin — og primærnodens CPU eller write-ahead-logg er fortsatt mettet, på den største maskinen det er verdt å kjøpe, på tvers av mange varer heller enn én. Én het vare er et bøtteproblem, ikke et shardproblem.
- **Dataene har vokst fra vedlikeholdet sitt.** En sikkerhetskopi kan ikke gjenopprettes innen tiden som er lovet virksomheten, en indeks kan ikke bygges eller en tabell vacuumes i de stille timene, eller arbeidssettet får ikke lenger plass i minnet.

- **Isolasjon er kravet.** En leietaker hvis kontrakt eller tilsynsmyndighet sier at dataene skal ligge for seg selv, eller hvis salg ikke får dele disk med noen andres.

Kostnaden havner på alt annet. Migreringer, sikkerhetskopiering, gjenoppretting og rapporter må ta hensyn til separate shards; en join på tvers av shards kan flyttes til applikasjonskoden; og en spørring som ikke kjenner shardnøkkelen sin, kan måtte spørre alle. Det er regningen delene ovenfor finnes for å unngå, og når den først er betalt, er den vanskelig å få tilbake.

Når en unnskyldning er billigere enn en lås

Det finnes et seriøst argument for å gå den andre veien. La hver server selge fra beholdningen den kan se, godta at en enhet nå og da går til to kjøpere, og la kundeservice refundere én av dem med en unnskyldning. Forretningslogikken blir konsistent over tid, bygd for de 99,9 % av forespørslene som aldri kolliderer, og en asynkron løkke finner kollisjonene etterpå og retter opp i dem. Til gjengjeld trenger butikken aldri å si «prøv igjen». Helland og Campbell legger fram den sterkeste formen av saken i *Building on Quicksand*: et system som kvitterer for arbeid før reservesystemet har fått det, gjør hvert løfte sannsynlighetsbasert, og bedrifter har alltid gjort opp løftene de ikke kunne holde, med unnskyldninger.

Vi er enige i det meste av det. Vi er uenige om hvor det skal brukes.

Det meste av en butikk bør være konsistent over tid. Katalogen, søkeindeksen, «3 igjen»-merket, handlekurven, anbefalingene, e-postene, lagerstrømmen og rapportene kan alle ligge sekunder bak, og å la dem vente på primærnoden koster fart og kjøper ingenting. Dynamo ble bygd for nettopp dette: handlekurven til Amazon glemmer eller avviser aldri en «legg i handlekurven», selv mens maskiner svikter, og prisen er at en sammenslåing kan bringe tilbake en vare kjøperen hadde fjernet. Etter reservasjonen er utsjekkingen i billettnotatet allerede asynkron: en saga av lokale steg, meldinger levert minst én gang, og konsumenter som er idempotente, med ordren som nøkkel. En avstemmingsløkke hører hjemme i ethvert design som snakker med en betalingsleverandør, fordi webhooks går tapt.

Reservasjonen er det ene stedet der det ikke holder. «Sjelden» forutsetter at kollisjoner er sjeldne, og i et lynsalg er kollisjonene salget. En server som selger ut fra sitt eget bilde, fortsetter å selge til den hører hva de andre har solgt, så oversalget er omtrent forsøksraten ganger replikeringsforsinkelsen. I en enkel simulering deler to regioner 2 000 forsøk i sekundet, hver selger ut fra sin egen telling og hører om den andres salg 200 millisekunder senere. Over fem kjøringer av et salg på 10 000 enheter solgte hver kjøring mellom 191 og 205 enheter som ikke fantes. Regnestykket forutsier 200: forsøksraten, ganger forsinkelsen, ganger andelen salg gjort i den andre regionen. Med én eier per vare solgte den samme modellen nøyaktig beholdningen. Det skjer hver gang noe blir utsolgt, enten noe svikter eller ikke, og det rammer på høydepunktet, når flest kjøpere ser på.

Oppetiden er mindre enn den høres ut. Gilbert og Lynch beviste avveiningen bak konsistens over tid: mens nettverket er partisjonert, kan et system svare eller være konsistent, ikke begge deler. Abadis PACELC legger til den andre halvdelen. Når ingenting er partisjonert, står avveiningen mellom latens og konsistens, og én eier per vare, med slanke setninger, er allerede rask. Ingen av designene når 100 %: betalingsleverandøren, DNS og nettverkene mellom dem har sine egne driftsavbrudd. Det det strenge designet gir opp, er smalt. Når eieren av beholdningen til én vare ikke kan nås, slutter den varen å selge i de sekundene en failover tar, og delen i billettnotatet om å sette varen på pause avgrenser pausen til varene på maskinen som feilet. Resten av butikken fortsetter å selge.

En unnskyldning er ikke en refusjon. Refusjonen er den billige delen. Kjøperen fikk ja, og kan allerede ha bestilt toget, tatt fri fra jobben og fortalt det til vennene sine. Å høre «utsolgt» et sekund for tidlig er en skuffelse. Å høre at en bekreftet ordre er trukket tilbake, er et brutt løfte. Kundeservice er bemannet for en vanlig dag, og oversalg kommer samlet, i årets travleste minutt. Før du planlegger å trekke tilbake bekreftede ordrer, spør en advokat hva en bekreftelse forplikter butikken til.

Behold den asynkrone delen, og flytt den til før ja-svaret. Det forslaget treffer riktig, er at kjøperen ikke trenger å vente på en lås. Ta imot hvert klikk i en varig, replikert logg, én partisjon per vare, og fortell kjøperen at svaret er på vei. Å ta imot et klikk avhenger da bare av loggen, og en logg som er replikert over tre maskiner, fortsetter å ta imot skrivinger når én av dem svikter. Én eier per vare leser loggen i rekkefølge og avgjør, i minnehastighet, og svaret går tilbake som en melding i løpet av sekunder. Under en failover for den eieren venter forespørslene i loggen i stedet for å bli avvist eller solgt to ganger. Ja-svaret er

en reservasjon, og betalingen følger etter den som i utsjekkingen i billettnotatet, så ingen betaler for en enhet de ikke fikk. Avstemmingen skjer fortsatt, men før løftet i stedet for etter det.

Del beholdningen, ikke tellingen. Når virksomheten trenger at regioner selger uten å spørre hverandre, gi hver region en kvote av beholdningen og la den selge fra sin egen kvote uten å spørre noen. En region som går tom for kvote, ber om mer, og mot slutten av salget faller kvotene sammen til én eier igjen. Bailis og kolleger finner ut hvilke regler servere kan overholde uten å snakke med hverandre, og en beholdning som ikke får gå under null, er ikke én av dem. Kvoter fjerner ikke den samtalen; de flytter den bort fra den hete stien, til det sjeldne øyeblikket da en kvote fylles på. Det er designet som er konsistent over tid og likevel holder invarianten. Den eneste kostnaden er halen, der en kjøper blir avvist i én region mens en annen region fortsatt har beholdning.

Når oversalg er riktig strategi. Noen varer kan lages på nytt: et trykk på bestilling, en T-skjorte leverandøren fortsatt har, en lisensnøkkel. Å selge forbi tellingen er da en restordre med en ærlig dato, og konsistens over tid er riktig design. Flyselskaper overbooker med vilje. De kjenner andelen som ikke møter opp, priser risikoen og kompenserer etter en regel skrevet på forhånd. Det er testen. Et oversalg er akseptabelt når virksomheten valgte størrelsen og botemiddelet før salget. Det er en feil når det rammer den som tilfeldigvis klikket sist.

Velger du unnskyldninger, så bygg dem som betalinger. Løkken forslaget beskriver, er en refusjonsprosess, og den trenger det alt som flytter penger, trenger:

- **En id på hvert salg,** slik at et nytt forsøk er det samme salget, og en hovedbok over salg som bare kan utvides. Beholdningstallet utledes av hovedboken og redigeres aldri.
- **En regel for hvem som beholder enheten** som du kunne forklart for kjøperen som tapte. Ordne etter loggen, ikke etter klokkene på ulike servere, der noen få millisekunders avvik ville avgjort hvem som får refusjonen.
- **Kompensasjoner som er idempotente og loggført:** kanseller, annuller eller refunder, gi kjøperen beskjed. De er angre-stegene i sagaen, kjørt av en maskin.
- **En frist.** Løkken må finne hvert oversalg før noe sendes, og før kjøperen finner det.

- **Et budsjett, målt.** Oversalg per salg, mot tallet virksomheten skrev under på, ved siden av invarianten på det samme dashbordet, med et varsel når budsjettet er brukt opp.

Svaret vårt er å være konsistent over tid overalt rundt reservasjonen, streng ved reservasjonen og asynkron i hvordan svaret når kjøperen. Design for den vanlige veien, men i et lynsalg er kollisjonen den vanlige veien. Vil virksomheten ha noen få oversalg i bytte mot noe, så gjør det til et tall virksomheten valgte, med et botemiddel skrevet før salget, og mål det.

Det vi ville sagt til den neste

- **Les før du kjøper.** `pg_stat_statements` og én `EXPLAIN` svarer på flere kapasitetsspørsmål enn en ny server gjør.
- **La typene si hva dataene er.** En boolean lagret som boolean er mindre, og den avviser tull.
- **Send databasen det som må se dataene, og ikke noe annet.** Filtrering og telling hører hjemme der. Forretningsregler gjør det ikke.
- **Hurtigbufferen sier nei; bare databasen sier ja.** Da koster en utdatert hurtigbuffer sekunder, aldri en enhet.
- **Partisjonering er husarbeid.** Den holder tabellene rundt en het rad små og gjør ingenting for raden.
- **Shard når én maskin er full av godt arbeid,** ikke når den er full av venting.
- **Avstem før ja-svaret, ikke etter.** Alt rundt reservasjonen kan være konsistent over tid. Reservasjonen selv, gjort på den måten, overselger med forsøksraten ganger forsinkelsen, ved hvert salg.

Kilder

- PostgreSQL: pg_stat_statements
- PostgreSQL: bruk av EXPLAIN
- PostgreSQL: det kumulative statistikksystemet
- PostgreSQL: numeriske typer
- PostgreSQL: tegntyper
- PostgreSQL: boolean-typen
- PostgreSQL: partielle indekser
- PostgreSQL: heap-only tuples (HOT)
- PostgreSQL: tabellpartisjonering

- PostgreSQL: hot standby og spørringskonflikter
- Pat Helland og Dave Campbell, Building on Quicksand
- Seth Gilbert og Nancy Lynch, Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services
- Daniel Abadi, Consistency Tradeoffs in Modern Distributed Database System Design
- DeCandia et al., Dynamo: Amazon’s Highly Available Key-value Store
- Hector Garcia-Molina og Kenneth Salem, Sagas
- Bailis et al., Coordination Avoidance in Database Systems

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 «Optimaliser databasen før du sharder den» 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://engineer.company/nb/notes/optimise-before-you-shard/