Engineer

Optimér databasen, før du sharder den

14. oktober 2026

Når en database bliver langsom under belastning, er det første forslag som regel flere maskiner. Nogle gange er det rigtigt. Oftere er den langsomme del en håndfuld sætninger, som ingen har læst endnu, og at sprede data over flere servere spreder de sætninger over mere hardware uden at gøre én eneste af dem billigere. Et problem, der ser uløseligt ud, ligger ofte kun én forespørgselsplan fra en pæn løsning, og horisontal skalering vælges nogle gange netop, så ingen behøver at læse den database, du allerede har.

Det her er rækkefølgen, vi arbejder i, før vi deler noget: kig, stram op, tag læsningerne væk fra den primære, partitionér tabellerne omkring det varme punkt, og shard først, når én maskine er fuld af godt arbejde. Flash-salget, hvor alle købere skriver til én række, er gennemgået i At sælge den sidste billet præcis én gang. Denne note beskriver metoden omkring det, og dens sidste afsnit tager det modsatte forslag op: sælg fra hver server, og undskyld for de enheder, der blev solgt to gange.

Brug erfaringen i jeres system

Arbejdet bag »Optimér databasen, før du sharder den« 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 →

Kig først

PostgreSQL registrerer allerede det meste af det, der er at vide. Stil den fire spørgsmål, før du ændrer noget.

Hvilke sætninger koster mest i alt? `pg_stat_statements`, en udvidelse, der indlæses gennem `shared_preload_libraries`, gemmer hver normaliseret sætning med antal kald og tid. Sortér efter total, ikke efter gennemsnit: en SQL-forespørgsel på 2 ms, der kører en million gange i timen, koster mere end en rapport på to sekunder, der kører to gange.

```
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;
```

Hvad gør den øverste egentlig? `EXPLAIN (ANALYZE, BUFFERS)` kører sætningen og viser den plan, den valgte, de rækker, hvert trin faktisk returnerede, mod dem, den forventede, og hvor mange sider den læste. En scanning af en million rækker for at returnere ti, eller et estimat, der rammer ved siden af med en faktor hundrede, er som regel hele historien. `ANALYZE` udfører det, den forklarer, så pak en skrivning ind 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';
```

Rækker her i et travlt minut betyder, at transaktioner venter på hinanden, ikke på hardware, og en større maskine fjerner ikke kravet om, at opdateringer af én fælles række udføres én ad gangen. Flere applikationsservere kan føje skrivende klienter til den kø.

Arbejder serveren, eller jonglerer den? Sammenlign antallet af aktive sessioner i `pg_stat_activity` med antallet af kerner. Når mange gange flere er aktive, end der er kerner, bruger serveren sin tid på at skifte mellem dem, og en forbindelsespulje løser det, før nogen hardware gør. Læs derefter `n_dead_tup` i

`pg_stat_user_tables` : en tabel med flere døde rækker end levende bliver scannet gennem sit eget affald, og det er autovacuum, der halter bagefter, ikke en database, der er for lille.

Skriv ned, hvad du finder. Det er udgangspunktet, som hvert senere skridt måles mod, og det ærlige svar på "skal vi sharde?" står ofte allerede i det.

Stram op i det, der er

Nogenlunde i rækkefølge efter udbytte i forhold til indsats.

Indekser, der passer til spørgsmålene. Planerne ovenfor udpeger scanningerne. Et indeks på de kolonner, en varm forespørgsel filtrerer på, gør en scanning til et opslag, og et partielt indeks rummer kun de rækker, en forespørgsel kan have brug for — for eksempel enheder, der stadig er til salg — så det skrumper, mens et salg skrider frem, i stedet for at vokse:

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

Hvert indeks betales også ved hver skrivning, så fjern de indekser, hvis `idx_scan` i `pg_stat_user_indexes` bliver ved med at stå på nul.

Typer, der siger, hvad værdien er. Skemaer skrevet hurtigt, af et menneske eller en model, griber efter `TEXT` , hvor en `SMALLINT` eller en `BOOLEAN` ville være nok. En `BOOLEAN` er én byte, en `SMALLINT` to, og `'yes'` gemt som tekst er fire. PostgreSQL indsætter desuden fyld, så hver kolonne med fast bredde begynder på sin egen justeringsgrænse, og en kolonnerækkefølge valgt tilfældigt koster derfor også bytes. Én lagerenhed, skrevet hurtigt:

```
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 enhed med typer, der siger, hvad værdierne 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 række 72 bytes og den anden 54, en fjerdedel mindre, så mere af tabellen kan ligge i hukommelsen. Typerne afviser også værdier som `'maybe'`, som applikationen ellers skulle forsvare sig imod, og en `CHECK`-begrænsning gør det samme for værdiområder: `quantity smallint CHECK (quantity > 0)`.

Færre rundture, og arbejdet der, hvor data er. Hver sætning, applikationen sender, koster en rundtur og en parsning. En løkke, der læser hundrede rækker og opdaterer dem én ad gangen, er hundrede og én rundture, hvor én mængdebaseret sætning ville gøre det, og en side, der henter hver enhed af en vare for at tælle de ledige, sender tusindvis af rækker for at beregne ét tal. Reglen, vi bruger: det, der skal se data, går til databasen — filtrering, optælling, en skrivning, der afhænger af, hvad der er der — og forretningsregler bliver i applikationen, fordi databasens CPU er den, det er sværest at få mere af i hele systemet.

Opdateringer, der bliver på deres side. En række, der ændres mange gange i minuttet, efterlader en ny version af sig selv hver gang. Når en opdatering ikke ændrer nogen indekseret kolonne, og siden har plads, beholder PostgreSQL den nye version på samme side og rører slet ikke noget indeks: en heap-only tuple, eller HOT-opdatering. Hold derfor de kolonner, en hyppig opdatering ændrer, ude af indeksene, og efterlad plads på siderne:

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

Indstillingen gælder for sider, der skrives fra da af, så sæt den, før rækkerne indlæses. På tabellen ovenfor var 2.000 reservationer, der satte `reserved`, `reserved_at` og `order_id`, alle HOT, og af 2.000 opdateringer, der satte `sold`, som det partielle indeks læser, var ingen HOT. Et salg sker én gang pr. enhed; en reservation kan ske mange gange. `n_tup_hot_upd` mod `n_tup_upd` i `pg_stat_user_tables` viser forholdet på et system i drift.

Én række siger, hvem der kom først

En enhed går til den første køber, hvis sætning når den, og reservationen udløber på databasens ur. Den hviler på to kolonner: `reserved`, en boolean, og `reserved_at`, det øjeblik, den blev sat.

```
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 række tilbage: denne køber har den. Ingen: en anden kom først, eller den er solgt. To reservationer på samme enhed står i kø ved dens rækkelås, og under READ COMMITTED, PostgreSQL's standard, tjekker den anden `WHERE`-betingelsen igen mod den række, den første committede, så præcis én af dem får en række tilbage. En reservation, der er ældre end ti minutter, tæller som ledig, så intet behøver at rydde den op, for at svaret er rigtigt.

Det gør boolean'en alene til et ufuldstændigt svar. `reserved` forbliver sand, efter en reservation er udløbet, og kun de to kolonner sammen siger, om en enhed er optaget, så hver forespørgsel, der spørger, skal spørge til begge. En periodisk oprydning, der nulstiller udløbne reservationer, holder tabellen ærlig for den, der læser `reserved` alene, men korrektheden venter aldrig på den. En begrænsning holder parret konsistent:

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

Billetnoten gemmer fristen, `held_until`, i stedet for starten; det er det samme design. Når køberen vælger enheden — et sæde, et nummereret tryk — navngiver sætningen den, som her. Når en hvilken som helst enhed er god nok, vælger varianten med `SKIP LOCKED` i den note en enhed, som ingen andre har reserveret.

Læsninger væk fra den primære

Det meste af en butiks trafik er læsning: produktsiden, "3 tilbage", kurven, ordrehistorikken. Intet af det har brug for den primære, den ene maskine, der skal holdes fri til skrivninger. En streaming-replika tager de læsninger, og at tilføje én er et spørgsmål om konfiguration, mens en shard er et projekt.

To forbehold følger med. En replika halter en smule bagefter, så en køber, der lige har reserveret en enhed og så læser sin kurv fra den, ser måske ikke reservationen: læs det, en bruger lige har skrevet, fra den primære, eller send den bruger til den primære i nogle sekunder efter hver skrivning. Og en lang rapport på en replika kan blive annulleret, når den holder de ændringer tilbage, replikaen afspiller. PostgreSQL-dokumentationen kalder det forespørgselskonflikter og nævner de indstillinger, der afvejer replikaens forsinkelse mod dem.

Den billigste læsning er den, der aldrig når en database. Når hver enhed af en vare er reserveret, kan ingen købe den, før en reservation udløber eller frigives, og databasen kan sige præcis, hvornår 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 nul, er `until` det øjeblik, den tidligste reservation udløber: cache "fuldt reserveret" med det som udløbstid, aldrig længere, og hver besøgende indtil da bliver afvist af cachen i stedet for af databasen. Når `free` er nul, og `until` er tom, er hver enhed solgt, og svaret er endeligt. En reservation, der frigives tidligt — et afvist kort, en forladt kurv — sletter cache-posten, så enheden er til salg igen med det samme.

Cachen siger kun nej. Et ja kommer altid fra den betingede `UPDATE` ovenfor, så en forældet cache kan koste et salg i nogle sekunder og kan aldrig sælge en enhed to gange. Det er billetnotens cachede "udsolgt", med en frist.

Partitionér tabellerne omkring den varme række

Partitionering deler én tabel i flere efter en nøgle — en måned, en lejer — bag ét navn. Det er værd at sige ligeud, hvad den ikke gør: den gør ikke én række, som flere transaktioner kæmper om, hurtigere. Den række ligger stadig i én partition, på én maskine, bag én lås.

Det, den gør, er at holde tabellerne omkring det varme punkt små. Ordrer, reservationshistorik og hændelser vokser uden ophør, og de fleste forespørgsler vil have den nyeste del:

```
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 forespørgsel, der nævner et datointerval, læser kun partitionerne inden for det, og denne måneds indekser forbliver små nok til at ligge i hukommelsen. Oprydning holder op med at være en `DELETE` af millioner af rækker, med den oppustning og vacuum, der følger, og bliver én sætning:

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

Primærnøglen skal indeholde partitionsnøglen, og derfor er den `(id, created_at)`. En forespørgsel, der ikke nævner nøglen, læser hver partition, så partitionér efter det, forespørgslerne allerede filtrerer på.



Når sharding er rimeligt

Der er én grænse, sharding ikke kan flytte: hver vare har én ejer. En enhed kan ikke reserveres på to servere på samme tid, så sharding flytter hele varer, aldrig halvdelen af én. Når en enkelt vare vokser fra sin maskine, deler billetnotens spande dens lager i mindre stykker, og hvert stykke har igen præcis én ejer.

Inden for det kalder vi sharding rimeligt, når én af disse ting er sand, og ikke før:

- **Én maskine er fuld af godt arbejde.** Sætningerne er allerede slanke — én betinget skrivning hver, intet eksternt inde i en transaktion, opdateringer, der bliver på deres side — og den primæres CPU eller write-ahead-log er stadig mættet, på den største maskine, det er værd at købe, på tværs af mange varer og ikke kun én. Én varm vare er et spandeproblem, ikke et shardproblem.
- **Datamængden er vokset fra vedligeholdelsen.** En backup kan ikke gendannes inden for den tid, forretningen er lovet, et indeks kan ikke bygges eller en tabel vacuumes i de stille timer, eller arbejdssættet passer ikke længere i hukommelsen.
- **Isolation er kravet.** En lejer, hvis kontrakt eller tilsynsmyndighed siger, at deres data skal ligge for sig selv, eller hvis salg ikke må dele disk med nogen andens.

Prisen lander på alt andet. Migreringer, backup, gendannelser og rapporter skal tage højde for separate shards; en joinoperation på tværs af shards kan flyttes til applikationskoden; og en forespørgsel, der ikke kender sin shardnøgle, kan være nødt til at spørge dem alle. Det er regningen, afsnittene ovenfor er der for at undgå, og når den først er betalt, er den svær at få tilbage.

Når en undskyldning er billigere end en lås

Der findes et seriøst argument for at gå den anden vej. Lad hver server sælge af det lager, den kan se, acceptér, at en enhed nu og da går til to købere, og lad kundeservice refundere den ene af dem med en undskyldning. Forretningslogikken bliver konsistent med tiden (eventual consistency), bygget til de 99,9 % af forespørgslerne, der aldrig støder sammen, og en asynkron løkke finder sammenstødene bagefter og retter op på dem. Til gengæld behøver butikken aldrig at sige "prøv igen". Helland og Campbell giver argumentet dets stærkeste form i *Building on Quicksand*: et system, der kvitterer for arbejde, før dets backup har det, gør hvert løfte til en sandsynlighed, og virksomheder har altid gjort de løfter, de ikke kunne holde, op med en undskyldning.

Vi er enige i det meste af det. Vi er uenige om, hvor det skal bruges.

Det meste af en butik bør være konsistent med tiden. Kataloget, søgeindekset, mærket "3 tilbage", kurven, anbefalingerne, mailene, lagerfeedet og rapporterne kan alle køre sekunder bagud, og at lade dem vente på den primære koster fart og giver intet igen. Dynamo blev bygget til netop det: Amazons indkøbskurv glemmer eller afviser aldrig et "læg i kurven", heller ikke mens maskiner fejler, og prisen er, at en fletning kan bringe en vare tilbage, som køberen havde fjernet. Efter reservationen er billetnotens checkout allerede asynkron: en saga af lokale trin, beskeder, der leveres mindst én gang, og forbrugere, der er idempotente, med ordren som nøgle. En afstemningsløkke hører hjemme i ethvert design, der taler med en betalingsudbyder, for webhooks går tabt.

Reservationen er det ene sted, hvor det ikke holder. "Sjældent" forudsætter, at sammenstød er sjældne, og i et flash-salg er sammenstødene salget. En server, der sælger ud fra sit eget billede, bliver ved med at sælge, indtil den hører, hvad de andre har solgt, så oversalget er omtrent forsøgsraten gange replikeringsforsinkelsen. I en simpel simulering deler to regioner 2.000 forsøg i sekundet, hver sælger ud fra sin egen optælling og hører om den andens salg

200 millisekunder senere. Over fem kørsler af et salg på 10.000 enheder solgte hver kørsel mellem 191 og 205 enheder, der ikke fandtes. Regnestykket forudsiger 200: forsøgsraten gange forsinkelsen gange den andel af salget, der sker i den anden region. Med én ejer pr. vare solgte den samme model præcis det, der var på lager. Det sker, hver gang lageret bliver solgt ud, uanset om noget fejlede, og det sker på højdepunktet, når flest købere ser med.

Oppetiden er mindre, end den lyder. Gilbert og Lynch beviste den byttehandel, som konsistens med tiden indgår: mens netværket er delt, kan et system svare, eller det kan være konsistent, ikke begge dele. Abadis PACELC tilføjer den anden halvdel. Når intet er delt, står valget mellem latens og konsistens, og én ejer pr. vare med slanke sætninger er allerede hurtig. Ingen af de to designs når 100 %: betalingsudbyderen, DNS og netværkene imellem har deres egne nedbrud. Det, det strenge design giver afkald på, er begrænset. Når ejeren af én vares lager ikke kan nås, holder den vare op med at sælge i de sekunder, en failover tager, og billetnotens afsnit om at sætte varen på pause begrænser pausen til varerne på den maskine, der fejlede. Resten af butikken sælger videre.

En undskyldning er ikke en refusion. Refusionen er den billige del. Køberen fik et ja og har måske allerede bestilt toget, taget fri og fortalt det til vennerne. At høre "udsolgt" et sekund for tidligt er en skuffelse. At høre, at en bekræftet ordre er trukket tilbage, er et brudt løfte. Kundeservice er bemandet til en almindelig dag, og oversalg kommer samlet, i årets travleste minut. Før du planlægger at trække bekræftede ordrer tilbage, så spørg en jurist, hvad en bekræftelse forpligter butikken til.

Behold den asynkrone del, og flyt den før ja'et. Det, forslaget har ret i, er, at køberen ikke skal vente på en lås. Tag imod hvert klik i en holdbar, replikeret log, én partition pr. vare, og fortæl køberen, at svaret er på vej. At tage imod et klik afhænger så kun af loggen, og en log, der er replikeret over tre maskiner, bliver ved med at tage imod skrivninger, når én af dem fejler. Én ejer pr. vare læser loggen i rækkefølge og beslutter i hukommelsens tempo, og svaret går tilbage som en besked inden for sekunder. Når den ejer får en failover, venter forespørgslerne i loggen i stedet for at blive afvist eller solgt to gange. Ja'et er en reservation, og betalingen følger efter, som i billetnotens checkout, så ingen betaler for en enhed, de ikke fik. Afstemningen sker stadig, men før løftet i stedet for efter det.

Del lageret, ikke optællingen. Når forretningen har brug for, at regioner sælger uden at spørge hinanden, så giv hver region en kvote af lageret, og lad den sælge af sin egen kvote uden at spørge nogen. En region, hvis kvote slipper op, beder om mere, og mod slutningen af salget falder kvoterne tilbage til én ejer. Bailis og kolleger kortlægger, hvilke regler servere kan overholde uden at tale sammen, og et lager, der ikke må gå under nul, er ikke en af dem. Kvoter fjerner ikke den samtale; de flytter den væk fra det enkelte salg, til det sjældne øjeblik, hvor en kvote fyldes op. Det er det design, der er konsistent med tiden og alligevel holder invarianten. Dets eneste pris er halen, hvor en køber bliver afvist i én region, mens en anden region stadig har lager.

Når oversalg er den rigtige politik. Nogle varer kan laves igen: et print on demand, en T-shirt, som leverandøren stadig har, en licensnøgle. At sælge ud over lagertallet er så en restordre med en ærlig dato, og konsistens med tiden er det rigtige design. Flyselskaber overbooker med vilje. De kender deres udeblivelsesrater, prissætter risikoen og kompenserer efter en regel, der er skrevet på forhånd. Det er testen. Et oversalg er acceptabelt, når forretningen valgte dets størrelse og dets afhjælpning før salget. Det er en fejl, når det rammer den, der tilfældigvis klikkede sidst.

Hvis du vælger undskyldninger, så byg dem som betalinger. Løkken, forslaget beskriver, er en refusionspipeline, og den har brug for det samme som alt andet, der flytter penge:

- **Et id på hvert salg,** så et nyt forsøg er det samme salg, og en journal over salg, som der kun tilføjes til. Lagertallet udledes af journalen og redigeres aldrig.
- **En regel for, hvem der beholder enheden,** som du kunne forklare den køber, der tabte. Ordn efter loggen, ikke efter forskellige serveres ure, hvor få millisekunders forskydning ville afgøre, hvem der får refusionen.
- **Kompensationer, der er idempotente og logget:** annullér, frigiv eller refundér, og fortæl køberen det. De er sagaens fortryd-trin, kørt af en maskine.
- **En frist.** Løkken skal finde hvert oversalg, før noget bliver sendt, og før køberen finder det.
- **Et budget, målt.** Oversalg pr. salg, mod det tal, forretningen skrev under på, ved siden af invarianten på samme dashboard, med en alarm, når budgettet er brugt.

Vores svar er at være konsistent med tiden overalt omkring reservationen, streng ved reservationen og asynkron i den måde, svaret når køberen på. Design efter det almindelige forløb, men i et flash-salg er sammenstødet det almindelige forløb. Hvis forretningen vil have nogle få oversalg til gengæld for noget, så gør det til et tal, forretningen valgte, med en afhjælpning skrevet før salget, og mål det.

Hvad vi ville sige til den næste

- **Læs, før du køber.** `pg_stat_statements` og én `EXPLAIN` besvarer flere kapacitetsspørgsmål end en ny server.
- **Lad typerne sige, hvad data er.** En boolean gemt som boolean er mindre, og den afviser vrøvl.
- **Send databasen det, der skal se data, og intet andet.** Filtrering og optælling hører hjemme dér. Forretningsregler gør ikke.
- **Cachen siger nej; kun databasen siger ja.** Så koster en forældet cache sekunder, aldrig en enhed.
- **Partitionering er oprydning.** Den holder tabellerne omkring en varm række små og gør intet for rækken.
- **Shard, når én maskine er fuld af godt arbejde,** ikke når den er fuld af venten.
- **Afstem før ja'et, ikke efter.** Alt omkring reservationen kan være konsistent med tiden. Selve reservationen, lavet på den måde, oversælger med forsøgsraten gange forsinkelsen, ved hvert salg.

Kilder

- PostgreSQL: pg_stat_statements
- PostgreSQL: brug af EXPLAIN
- PostgreSQL: det kumulative statistiksystem
- PostgreSQL: numeriske typer
- PostgreSQL: tegntyper
- PostgreSQL: boolean-typen
- PostgreSQL: partielle indekser
- PostgreSQL: heap-only tuples (HOT)
- PostgreSQL: tabelpartitionering
- PostgreSQL: hot standby og forespørgselskonflikter
- 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 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 »Optimér databasen, før du sharder den« 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://engineer.company/da/notes/optimise-before-you-shard/