Engineer

Optimera databasen innan du shardar den

14 oktober 2026

När en databas blir långsam under last är det första förslaget oftast fler maskiner. Ibland är det rätt. Oftare är den långsamma delen en handfull satser som ingen har läst ännu, och att sprida datan över flera servrar sprider de satserna över mer hårdvara utan att göra någon av dem billigare. Ett problem som ser olösligt ut är ofta bara en frågeplan från ett snyggt svar, och horisontell skalning väljs ibland just för att ingen ska behöva läsa den databas du redan har.

Det här är ordningen vi arbetar i innan vi delar upp något: titta, skärp, flytta läsningarna från primärnoden, partitionera tabellerna runt den heta punkten och sharda först när en maskin är full av bra arbete. Blixtrean, där varje köpare skriver till en enda rad, gås igenom i Att sälja den sista biljetten exakt en gång. Den här anteckningen är metoden runt den, och dess sista avsnitt tar sig an det motsatta förslaget: sälj från varje server och be om ursäkt för de enheter som såldes två gånger.

Använd erfarenheten i ert system

Arbetet bakom ”Optimera databasen innan du shardar den” 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. Utformade och drev 6 ETL/ELT-pipelines — 6 ETL-pipelines över tre databasmotorer

Utformade och drev 6 ETL/ELT-pipelines Hör av er →

Titta först

PostgreSQL registrerar redan det mesta som finns att veta. Ställ fyra frågor till den innan du ändrar något.

Vilka satser kostar mest totalt? `pg_stat_statements`, ett tillägg som läses in via `shared_preload_libraries`, sparar varje normaliserad sats med antalet anrop och tiden. Sortera på total tid, inte på medelvärde: en SQL-fråga på 2 ms som körs en miljon gånger i timmen kostar mer än en rapport på två sekunder som körs två gånger.

```
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;
```

Vad gör den översta faktiskt? `EXPLAIN (ANALYZE, BUFFERS)` kör satsen och visar planen den valde, de rader varje steg verkligen returnerade mot de rader den väntade sig, och hur många sidor den läste. En skanning av en miljon rader för att returnera tio, eller en uppskattning som slår fel med en faktor hundra, är oftast hela förklaringen. `ANALYZE` kör det den förklarar, så lägg en skrivning mellan `BEGIN` och `ROLLBACK`.

Vem väntar, och på vem?

```
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 här under en hektisk minut betyder att transaktioner väntar på varandra, inte på hårdvaran, och en större maskin tar inte bort kravet på att uppdateringar av en och samma delade rad utförs en i taget. Fler applikationsservrar kan öka

antalet skrivande klienter i den kön.

Arbetar servern eller jonglerar den? Räkna de aktiva sessionerna i `pg_stat_activity` mot antalet kärnor. När många gånger fler sessioner är aktiva än det finns kärnor ägnar servern sin tid åt att växla mellan dem, och en anslutningspool löser det innan någon hårdvara gör det. Läs sedan `n_dead_tup` i `pg_stat_user_tables` : en tabell med fler döda rader än levande skannas genom sitt eget skräp, vilket är autovacuum som halkar efter, inte en databas som är för liten.

Skriv ned vad du hittar. Det är utgångsläget som varje senare steg mäts mot, och det ärliga svaret på ”behöver vi sharda?” finns ofta redan i det.

Skärp det som redan finns

Ungefär i ordning efter vad det ger för insatsen.

Index som matchar frågorna. Planerna ovan pekar ut skanningarna. Ett index på de kolumner som en het SQL-fråga filtrerar på gör en skanning till ett uppslag, och ett partiellt index rymmer bara de rader en SQL-fråga kan vilja ha — enheter som fortfarande är till salu, till exempel — så att det krymper medan en rea pågår i stället för att växa:

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

Varje index betalas också vid varje skrivning, så ta bort dem vars `idx_scan` i `pg_stat_user_indexes` står kvar på noll.

Typer som säger vad värdet är. Scheman som skrivs snabbt, av en människa eller en modell, tar till `TEXT` där en `SMALLINT` eller en `BOOLEAN` skulle räcka. En `BOOLEAN` är en byte, en `SMALLINT` två, och `'yes'` lagrat som text är fyra. PostgreSQL lägger också in utfyllnad så att varje kolumn med fast bredd hamnar på sin egen justeringsgräns, så en kolumnordning som har valts av en slump kostar också byte. En lagerenhet, snabbt skriven:

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

Och samma enhet med typer som säger vad värdena är, bredast först:

```
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 mäter den första raden 72 byte och den andra 54, en fjärdedel mindre, så mer av tabellen ryms i minnet. Typerna vägrar också värden som `'maybe'` som applikationen annars skulle behöva försvara sig mot, och ett `CHECK` -villkor gör samma sak för intervall: `quantity smallint CHECK (quantity > 0)` .

Färre nätverksrundor, och arbetet där datan finns. Varje sats som applikationen skickar kostar en nätverksrunda och en parsning. En loop som läser hundra rader och uppdaterar dem en i taget är hundraen nätverksrundor där en enda mängdbaserad sats skulle räcka, och en sida som hämtar varje enhet av en vara för att räkna de lediga skickar tusentals rader för att räkna ut ett enda tal. Regeln vi följer: det som måste se datan går till databasen — filtrering, räkning, en skrivning som beror på vad som finns där — och affärsregler stannar i applikationen, eftersom databasens CPU är den som är svårast att få mer av i hela systemet.

Uppdateringar som stannar på sin sida. En rad som ändras många gånger i minuten lämnar en ny version av sig själv efter sig varje gång. När en uppdatering inte ändrar någon indexerad kolumn och sidan har plats, behåller PostgreSQL den nya versionen på samma sida och rör inget index alls: en heap-only tuple, eller HOT-uppdatering. Håll därför de kolumner som en frekvent uppdatering ändrar utanför indexen, och lämna plats på sidorna:

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

Inställningen gäller sidor som skrivs från och med då, så sätt den innan raderna läses in. På tabellen ovan var 2 000 reservationer som satte `reserved`, `reserved_at` och `order_id` alla HOT, och av 2 000 uppdateringar som satte `sold`, som det partiella indexet läser, var ingen HOT. En försäljning sker en gång per enhet; en reservation kan ske många gånger. `n_tup_hot_upd` mot `n_tup_upd` i `pg_stat_user_tables` visar kvoten i ett system i drift.

En rad säger vem som var först

En enhet går till den första köpare vars sats når den, och reservationen löper ut enligt databasens klocka. Två kolumner bär den: `reserved`, en boolean, och `reserved_at`, tidpunkten då den sattes.

```
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;
```

En rad tillbaka: den här köparen har den. Ingen: någon annan var först, eller så är den såld. Två reservationer av samma enhet köar på dess radlås, och med READ COMMITTED, standardnivån i PostgreSQL, kontrollerar den andra `WHERE`-villkoret igen mot raden som den första committade, så att exakt en av dem får en rad tillbaka. En reservation som är äldre än tio minuter räknas som ledig, så ingenting behöver städa bort den för att svaret ska bli rätt.

Det gör booleanen ensam till ett ofullständigt svar. `reserved` förblir true efter att en reservation har löpt ut, och bara de två kolumnerna tillsammans säger om en enhet är reserverad, så varje SQL-fråga som vill veta det måste fråga efter

båda. En periodisk städning som nollställer utgångna reservationer håller tabellen ärlig för den som läser `reserved` ensam, men korrektheten väntar aldrig på den. Ett villkor håller paret konsistent:

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

Biljettanteckningen lagrar tidsgränsen, `held_until`, i stället för starten; det är samma lösning. När köparen väljer enheten — en sittplats, ett numrerat tryck — namnger satsen den, som här. När vilken enhet som helst duger väljer den anteckningens `SKIP LOCKED`-form en som ingen annan håller.



Flytta läsningarna från primärnoden

Det mesta av en butiks trafik är läsning: produktsidan, ”3 kvar”, varukorgen, orderhistoriken. Inget av det behöver primärnoden, den enda maskin som måste hållas fri för skrivningar. En strömmande replik tar de läsningarna, och att lägga till en är konfiguration där en shard är ett projekt.

Två förbehåll följer med. En replik ligger lite efter, så en köpare som just har reserverat en enhet och sedan läser sin varukorg från den kanske inte ser reservationen: läs det en användare just har skrivit från primärnoden, eller skicka den användaren till primärnoden i några sekunder efter varje skrivning. Och en lång rapport på en replik kan avbrytas när den håller upp ändringarna som repliken spelar upp. PostgreSQL-dokumentationen kallar det frågekonflikter och listar de inställningar som väger replikens eftersläpning mot dem.

Den billigaste läsningen är en som aldrig når en databas. När varje enhet av en vara är reserverad kan ingen köpa den förrän en reservation löper ut eller släpps, och databasen kan säga exakt när det sker:

```
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;
```

När `free` är noll är `until` tidpunkten då den tidigaste reservationen löper ut: cacha ”fullt reserverad” med den som utgångstid, aldrig längre, så avvisas varje besökare fram till dess av cachen i stället för av databasen. När `free` är noll och `until` är tom är varje enhet såld och svaret slutgiltigt. En reservation som släpps i förtid — ett nekat kort, en övergiven varukorg — tar bort cacheposten, så att enheten genast är till salu igen.

Cachen säger aldrig något annat än nej. Ett ja kommer alltid från den villkorade `UPDATE` ovan, så en inaktuell cache kan kosta en försäljning i några sekunder och kan aldrig sälja en enhet två gånger. Det är biljettanteckningens cachade ”slutsålt”, med en tidsgräns.

Partitionera tabellerna runt den heta raden

Partitionering delar upp en tabell i flera efter en nyckel — en månad, en tenant — bakom ett enda namn. Det är värt att säga rakt ut vad den inte gör: den gör inte en rad som flera transaktioner konkurrerar om något snabbare. Den raden finns fortfarande i en partition, på en maskin, bakom ett lås.

Det den gör är att hålla tabellerna runt den heta punkten små. Ordrar, reservationshistorik och händelser växer för alltid, och de flesta SQL-frågor vill ha den senaste 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 SQL-fråga som anger ett datumintervall läser bara partitionerna inom det, och den här månadens index förblir små nog att ligga i minnet. Gallring slutar vara en `DELETE` av miljontals rader, med den uppsvällning och vacuum-körning som följer, och blir en enda sats:

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

Primärnyckeln måste innehålla partitionsnyckeln, och därför är den `(id, created_at)`. En SQL-fråga som inte anger nyckeln läser varje partition, så partitionera på det som SQL-frågorna redan filtrerar på.



När sharding är rimligt

Det finns en gräns som sharding inte kan flytta: varje vara har en ägare. En enhet kan inte reserveras på två servrar samtidigt, så sharding flyttar hela varor, aldrig halva. När en enda vara växer ur sin maskin delar biljettanteckningens hinkar upp dess lagersaldo i mindre delar, och varje del har återigen exakt en ägare.

Inom den ramen kallar vi sharding rimligt när något av följande stämmer, och inte förr:

- **En maskin är full av bra arbete.** Satserna är redan slimmade — en villkorad skrivning var, ingenting externt inuti en transaktion, uppdateringar som stannar på sin sida — och primärnodens CPU eller write-ahead-logg är ändå mättad, på den största maskin som är värd att köpa, över många varor snarare än en. En het vara är ett hinkproblem, inte ett shardproblem.
- **Datan har vuxit ur sitt underhåll.** En säkerhetskopia kan inte återställas inom den tid som verksamheten har blivit lovad, ett index kan inte byggas eller en tabell städas med vacuum under de lugna timmarna, eller så ryms arbetsmängden inte längre i minnet.
- **Isolering är kravet.** En tenant vars avtal eller tillsynsmyndighet säger att dess data ska ligga för sig, eller vars rea inte får dela disk med någon annans.

Kostnaden hamnar på allt annat. Migreringar, säkerhetskopiering, återställning och rapporter måste ta hänsyn till separata shards; en join över shards kan flyttas till applikationskoden; och en SQL-fråga som inte känner till sin shardnyckel kan behöva fråga alla. Det är den nota som avsnitten ovan finns till för att undvika, och när den väl är betald är det svårt att ta sig tillbaka.

När en ursäkt är billigare än ett lås

Det finns ett seriöst argument för att gå åt andra hållet. Låt varje server sälja av det lager den ser, acceptera att en enhet då och då går till två köpare, och låt kundtjänst återbetala en av dem med en ursäkt. Affärslogiken blir slutligt konsistent, byggd för de 99,9 % av begärandena som aldrig krockar, och en asynkron loop hittar krockarna i efterhand och gottgör dem. I gengäld behöver butiken aldrig säga ”försök igen”. Helland och Campbell för fram den starkaste formen av argumentet i *Building on Quicksand*: ett system som bekräftar arbete innan dess reservnod har fått det gör varje löfte probabilistiskt, och företag har i alla tider gjort upp om löften de inte kunnat hålla genom att be om ursäkt.

Vi håller med om det mesta av det. Vi är oense om var det ska tillämpas.

Det mesta av en butik bör vara slutligt konsistent. Katalogen, sökindexet, märket med ”3 kvar”, varukorgen, rekommendationerna, e-postmeddelandena, flödet till lagret och rapporterna kan alla ligga sekunder efter, och att låta dem vänta på primärnoden kostar snabbhet och köper ingenting. Dynamo byggdes för just detta: Amazons kundvagn glömmer eller avvisar aldrig ett ”lägg i kundvagnen”, inte ens när maskiner fallerar, och priset är att en sammanslagning kan återställa en vara som köparen hade tagit bort. Efter reservationen är biljettanteckningens köpflöde redan asynkront: en saga av lokala steg, meddelanden som levereras minst en gång, och konsumenter som är idempotenta, nycklade på ordern. En avstämningsloop hör hemma i varje lösning som talar med en betalningsleverantör, eftersom webhooks går förlorade.

Reservationen är det enda ställe där det inte håller. ”Sällsynt” förutsätter att krockar är sällsynta, och i en blixtrea är krockarna själva rean. En server som säljer utifrån sin egen bild fortsätter sälja tills den hör vad de andra sålde, så översäljningen blir ungefär försöksfrekvensen gånger replikeringsfördröjningen. I en enkel simulering delar två regioner på 2 000 försök i sekunden, var och en säljer utifrån sin egen räkning och hör om den andras försäljningar 200 millisekunder senare. Över fem körningar av en rea på 10 000 enheter sålde varje

körning mellan 191 och 205 enheter som inte fanns. Räknestycket förutsäger 200: försöksfrekvensen, gånger fördröjningen, gånger andelen försäljningar som görs i den andra regionen. Med en ägare per vara sålde samma modell exakt lagret. Det händer varje gång något säljer slut, oavsett om något gick sönder eller inte, och det inträffar vid klimax, när flest köpare tittar.

Upptiden är mindre än den låter. Gilbert och Lynch bevisade den avvägning som slutlig konsistens gör: medan nätverket är partitionerat kan ett system svara eller vara konsistent, inte båda. Abadis PACELC lägger till den andra halvan. När ingenting är partitionerat är avvägningen latens mot konsistens, och en ägare per vara, med slimmade satser, är redan snabb. Ingen av lösningarna når 100 %: betalningsleverantören, DNS och nätverken mellan dem har sina egna avbrott. Det den strikta lösningen ger upp är smalt. När ägaren av en varas lagersaldo inte går att nå slutar den varan att säljas under de sekunder som en failover tar, och biljettanteckningens avsnitt om att pausa varan avgränsar pausen till varorna på maskinen som gick ned. Resten av butiken fortsätter sälja.

En ursäkt är inte en återbetalning. Återbetalningen är den billiga delen. Köparen fick ett ja, och kan redan ha bokat tåget, tagit ledigt och berättat för sina vänner. Att höra ”slutsålt” en sekund för tidigt är en besvikelse. Att höra att en bekräftad order har dragits tillbaka är ett sviket löfte. Kundtjänst är bemannad för en vanlig dag, och översäljningar kommer samtidigt, under årets mest hektiska minut. Innan du planerar att dra tillbaka bekräftade ordrar, fråga en jurist vad en bekräftelse förpliktar butiken till.

Behåll den asynkrona delen, och flytta den före jaet. Det som förslaget har rätt i är att köparen inte behöver vänta på ett lås. Ta emot varje klick i en beständig, replikerad logg, en partition per vara, och säg till köparen att svaret är på väg. Att ta emot ett klick beror då bara på loggen, och en logg som replikeras över tre maskiner fortsätter ta emot skrivningar när en av dem går sönder. En ägare per vara läser loggen i ordning och avgör, i minnets hastighet, och svaret går tillbaka som ett meddelande inom några sekunder. När den ägaren gör failover väntar begärandena i loggen i stället för att avvisas eller säljas två gånger. Jaet är en reservation, och betalningen följer på den som i biljettanteckningens köpflöde, så ingen betalar för en enhet utan att få den. Avstämningen sker fortfarande, men före löftet i stället för efter det.

Dela lagret, inte räkningen. När verksamheten behöver att regioner säljer utan att fråga varandra, ge varje region en kvot av lagret och låt den sälja av sin egen kvot utan att fråga någon. En region vars kvot tar slut ber om mer, och mot slutet

av rean slås kvoterna ihop till en ägare igen. Bailis med kolleger reder ut vilka regler servrar kan upprätthålla utan att tala med varandra, och ett lagersaldo som inte får gå under noll är inte en av dem. Kvoter tar inte bort det samtalet; de flyttar det bort från den heta vägen, till det sällsynta ögonblick då en kvot fylls på. Det är den slutligt konsistenta lösning som behåller invarianten. Dess enda kostnad är svansen, där en köpare avvisas i en region medan en annan region fortfarande har lager.

När en översäljning är rätt policy. Vissa varor kan tillverkas igen: ett tryck på beställning, en t-shirt som leverantören fortfarande har, en licensnyckel. Att sälja förbi räkningen är då en restorder med ett ärligt datum, och slutlig konsistens är rätt lösning. Flygbolag överbokar med flit. De känner till hur stor andel av passagerarna som uteblir, prissätter risken och kompenserar enligt en regel som är skriven i förväg. Det är testet. En översäljning är acceptabel när verksamheten valde dess storlek och dess gottgörelse före rean. Den är en defekt när den drabbar den som råkade klicka sist.

Om du väljer ursäkter, bygg dem som betalningar. Loopen som förslaget beskriver är en kedja för återbetalningar, och den behöver det som allt som flyttar pengar behöver:

- **Ett id på varje försäljning,** så att ett omförsök är samma försäljning, och en liggare över försäljningar där poster bara läggs till. Lagersaldot härleds ur liggaren och redigeras aldrig.
- **En regel för vem som behåller enheten** som du skulle kunna förklara för den köpare som förlorade. Ordna efter loggen, inte efter klockorna på olika servrar, där några millisekunders avvikelse skulle avgöra vem som får återbetalningen.
- **Kompensationer som är idempotenta och loggade:** avbryt, annullera eller återbetala, meddela köparen. De är sagans ångra-steg, körda av en maskin.
- **En tidsgräns.** Loopen måste hitta varje översäljning innan något skickas, och innan köparen hittar den.
- **En budget, mätt.** Översäljningar per rea, mot det tal som verksamheten skrev under på, bredvid invarianten på samma instrumentpanel, med ett larm när budgeten är förbrukad.

Vårt svar är att vara slutligt konsistent överallt runt reservationen, strikt vid reservationen och asynkron i hur svaret når köparen. Bygg för det vanliga fallet, men i en blixtrea är krocken det vanliga fallet. Om verksamheten vill ha några

översäljningar i utbyte mot något, gör det till ett tal som verksamheten har valt, med en gottgörelse som är skriven före rean, och mät det.

Vad vi skulle säga till nästa person

- **Läs innan du köper.** `pg_stat_statements` och en enda `EXPLAIN` besvarar fler kapacitetsfrågor än en ny server gör.
- **Låt typerna säga vad datan är.** En boolean som lagras som boolean är mindre, och den vägrar nonsens.
- **Skicka till databasen det som måste se datan, och inget annat.** Filtrering och räkning hör hemma där. Affärsregler gör det inte.
- **Cachen säger nej; bara databasen säger ja.** Då kostar en inaktuell cache sekunder, aldrig en enhet.
- **Partitionering är städning.** Den håller tabellerna runt en het rad små och gör ingenting för raden.
- **Sharda när en maskin är full av bra arbete,** inte när den är full av väntan.
- **Stäm av före jaet, inte efter.** Allt runt reservationen kan vara slutligt konsistent. Själva reservationen, gjord på det sättet, översäljer med försöksfrekvensen gånger fördröjningen, vid varje rea.

Källor

- PostgreSQL: pg_stat_statements
- PostgreSQL: att använda EXPLAIN
- PostgreSQL: det kumulativa statistiksystemet
- PostgreSQL: numeriska typer
- PostgreSQL: teckentyper
- PostgreSQL: typen boolean
- PostgreSQL: partiella index
- PostgreSQL: heap-only tuples (HOT)
- PostgreSQL: tabellpartitionering
- PostgreSQL: hot standby och frågekonflikter
- Pat Helland och Dave Campbell, Building on Quicksand
- Seth Gilbert och 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 och Kenneth Salem, Sagas
- Bailis et al., Coordination Avoidance in Database Systems

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 ”Optimera databasen innan du shardar den” 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://engineer.company/sv/notes/optimise-before-you-shard/