**Engineer**

# Materialiserad vy

en vy som lagrar resultatet av sin fråga som en tabell

## Vad det är

En materialiserad vy är en fråga vars resultat databasen sparar, så att en läsning returnerar rader som räknats fram tidigare i stället för att köra frågan på nytt. På engelska heter den materialized view, och det är också namnet som kommandona använder. Den har sin plats mellan två äldre objekt. En vanlig vy lagrar bara sin fråga, och varje läsning kör den mot tabellerna därunder. En tabell som skapats med `CREATE TABLE AS` lagrar ett resultat men glömmer frågan som gav det. En materialiserad vy behåller båda: raderna, som läsningarna hämtar från, och frågan, som kan köras igen för att ersätta dem.

PostgreSQL fick materialiserade vyer i version 9.3, som släpptes den 9 september 2013, och versionsanteckningarna krediterar Kevin Grittner för arbetet. Enligt dokumentationen använder de regelsystemet på samma sätt som vanliga vyer men lagrar resultatet i en tabellik form. För frågeparsern är en materialiserad vy en relation som vilken tabell eller vy som helst, så en `SELECT` mot den läser de lagrade raderna direkt; den lagrade frågan används bara när vyn ska fyllas. Innehållet kan inte ändras direkt. För att ersätta innehållet med ett nytt frågeresultat måste den lagrade frågan köras igen.

Två kommandon sköter det arbetet. `CREATE MATERIALIZED VIEW` definierar vyn och kör frågan på en gång för att fylla den, om den inte får `WITH NO DATA`. `REFRESH MATERIALIZED VIEW` kör den lagrade frågan igen och ersätter hela innehållet med det nya resultatet. Version 9.4, som släpptes den 18 december 2014, lade till `CONCURRENTLY`, som låter en uppdatering köras utan att blockera de sessioner som läser vyn. Båda kommandona är PostgreSQL-tillägg och inte en del av SQL-standarden; andra databaser erbjuder idén med sin egen syntax och sina egna regler.

Använd begreppet i ett beslut

Använd ”Vad är en materialiserad vy?” i ett konkret beslut. Vi kan knyta begreppet till systemet ni bygger eller köper. Hittar databasfrågorna och de saknade indexen bakom ett system som har blivit tyngre för varje år, och åtgärdar dem som spelar roll.

**Se våra tjänster** Hör av er →

Granska antagandena bakom ordet

Använd förklaringen och det länkade arbetet för att formulera frågor till projektet. Vi kan undersöka antagandena tillsammans innan ett verktyg väljs. 30 dagars eldata på 180 ms — 180 ms för att aggregera en månads mätaravläsningar

**30 dagars eldata på 180 ms** Hör av er →

# Varför det spelar roll

Vissa frågor är dyra och ställs ofta: en dashboard som summerar ett års ordrar per dag, en rapport som joinar sex tabeller, en sida som rangordnar produkter efter försäljning. Körd mot källtabellerna betalar varje anrop för hela genomsökningen, joinen och aggregeringen igen, fast svaret knappt rör sig från ett anrop till nästa. En materialiserad vy betalar det priset en gång per uppdatering, och varje läsning därefter är en genomsökning av en liten tabell som räknats fram i förväg. Enligt PostgreSQL-dokumentationen går det ofta mycket snabbare att läsa de lagrade data än att gå till tabellerna därunder, vare sig det sker direkt eller via en vy.

Priset är aktualitet. Mellan två uppdateringar visar vyn data som de såg ut vid den senaste, och i PostgreSQL uppdaterar ingenting den om inte någon, eller något schemalagt, ber om det. Många användningar klarar sig med det. Gårdagens försäljning ändras inte i dag, och en månadsrapport behöver inte ordrarna från den senaste minuten. Det som en materialiserad vy tvingar fram är ett uttryckligt beslut: hur gammalt svaret får vara, och vems uppgift det är att hindra det från att bli äldre.

Den ger också index en plats att bo på: det lagrade resultatet kan få egna index, byggda för de frågor som läser det. PostgreSQL-dokumentationen visar en annan användning av samma sak: data som hämtas in från ett annat system via en foreign data wrapper, som kanske inte stöder index alls, kan kopieras över till en materialiserad vy som gör det.

För ett team som håller sitt schema normaliserat är det ofta här den platta formen hör hemma. Databasnormalisering lagrar varje uppgift på ett ställe och besvarar de flesta frågor med joins. En materialiserad vy kan hålla det joinade och summerade resultatet som rapporteringen vill ha, medan källtabellerna behåller sin enda kopia av varje uppgift och varje skrivning fortfarande går till ett ställe.

# Så ser det ut i praktiken

Ta en webbutiks ordertabeller: `orders`, med en rad per order och datumet då den lades, och `order_lines`, med artikelnummer, antal och styckpris för varje rad. En försäljningsdashboard ska visa sålda enheter och intäkter per produkt och dag. Frågan är kort, men över flera års ordrar läser den varenda rad:

```
CREATE MATERIALIZED VIEW daily_sales AS
SELECT o.ordered_on                   AS day,
       l.sku,
       sum(l.quantity)                AS units,
       sum(l.quantity * l.unit_price) AS revenue
FROM   orders o
JOIN   order_lines l USING (order_id)
WHERE  o.ordered_on < CURRENT_DATE
GROUP  BY o.ordered_on, l.sku;
```

`WHERE`-villkoret utesluter dagens försäljning, vars siffror fortfarande ändras, så att dashboarden bara visar hela dagar; exemplet i PostgreSQL:s egen dokumentation behandlar en dag med ofullständig försäljning på samma sätt. Kommandot kör frågan en gång och lagrar resultatet. Sedan kommer ett index:

```
CREATE UNIQUE INDEX daily_sales_sku_day ON daily_sales (sku, day);
```

Indexet har två uppgifter. Det låter dashboarden hitta en produkts dagar utan att läsa hela vyn, och det är det som senare gör en samtidig uppdatering möjlig. Frågan grupperar på dag och artikelnummer, så varje par förekommer en gång, och därför kan indexet vara unikt. Från och med nu läser dashboarden vyn som en tabell:

```
SELECT day, units, revenue
FROM   daily_sales
WHERE  sku = 'TEA-250'
ORDER  BY day;
```

Varje natt, strax efter midnatt, uppdaterar ett schemalagt jobb den:

```
REFRESH MATERIALIZED VIEW CONCURRENTLY daily_sales;
```

Utan `CONCURRENTLY` använder en uppdatering som berör många rader oftast mindre resurser och blir klar snabbare, men den kan blockera dashboardens läsningar tills den är klar. Med `CONCURRENTLY` fortsätter läsningarna under uppdateringen, och dokumentationen påpekar att det till och med kan vara det snabbare valet när bara få rader ändras, som här, där en ny dag tillkommer varje natt. Valet har två villkor. Vyn måste redan innehålla data, och den måste ha minst ett unikt index byggt av rena kolumnnamn som täcker alla rader, alltså inget uttrycksindex och inget `WHERE` -villkor. Indexet ovan uppfyller det andra; att vyn skapades med sina data uppfyllde det första.

Andra databaser drar gränsen på andra ställen. I BigQuery underhålls materialiserade vyer i bakgrunden: ändringar i bastabellerna läggs till inkrementellt utan att någon ber om det, en fråga mot vyn fogar in de senaste ändringarna så att resultatet förblir aktuellt, och smart tuning kan skicka en fråga som skrivits mot bastabellerna vidare till vyn i stället. Det gäller inkrementella vyer, som bara godtar en begränsad del av SQL. En vy som definierats med `allow_non_incremental_definition = true` godtar de flesta frågor, men varje uppdatering kör hela frågan, och resultaten är inte alltid aktuella.

SQL Server når ett liknande resultat med indexerade vyer. Skapar man ett unikt klustrat index på en vy, som måste vara skapad `WITH SCHEMABINDING` och ha en deterministisk definition, lagras resultatet på samma sätt som en tabell med ett klustrat index, och frågeoptimeraren kan använda det även för en fråga som inte nämner vyn. SQL Server uppdaterar den indexerade vyn tillsammans med varje insert, update och delete i de tabeller den är beroende av, så det finns ingen uppdatering att schemalägga; kostnaden flyttar till skrivningarna, och Microsofts dokumentation varnar för att deras prestanda kan försämras avsevärt.

Samma avvägning syns i produktionsarbete. I en tidigare roll utvecklade grundaren ETL/ELT-pipelines i PL/pgSQL, som utnyttjade partitionerade tabeller och materialiserade vyer för att optimera läsprestandan för stora datamängder, tillsammans med villkor för primärnycklar och främmande nycklar för referensintegriteten och unika och sammansatta index för JOIN-operationerna.

# Var det går fel

**Gamla siffror som ser aktuella ut.** Ett uppdateringsjobb som slutade köras för en vecka sedan lämnar kvar en vy som besvarar varje fråga, snabbt och fel, och ingenting i resultatet säger hur gammalt det är. Låt jobbet registrera tiden för varje lyckad uppdatering någonstans där dashboarden kan visa den, och larma när den tiden är äldre än läsarna har godtagit.

**En uppdatering som blockerar läsarna.** En vanlig `REFRESH MATERIALIZED VIEW` över en stor vy kan blockera alla sessioner som försöker läsa den tills uppdateringen är klar. Lägg till ett unikt index och använd `CONCURRENTLY`, eller lägg de vanliga uppdateringarna till timmar då ingen läser vyn. När de flesta rader ändras vid varje körning kan den vanliga uppdateringen ändå vara den billigaste.

`CONCURRENTLY` **avvisas.** Valet kräver ett unikt index på rena kolumner som täcker alla rader. Ett unikt index på `lower(email)`, eller ett som begränsas av ett `WHERE`-villkor, duger inte. Det kräver också en vy som redan innehåller data, så det misslyckas på en vy som skapats `WITH NO DATA`, och de två valen kan inte kombineras. Lägg till ett index som uppfyller kraven, och fyll vyn en gång med en vanlig uppdatering.

**En vy som inte går att läsa.** `WITH NO DATA` skapar vyn utan att köra frågan och markerar den som oläsbar, och varje fråga mot den misslyckas fram till den första uppdateringen. Det kan vara rätt val i en migrering som inte ska vänta på en lång fråga, men då måste den första uppdateringen planeras, inte glömmas.

**En ändrad definition som aldrig kom fram.** `CREATE MATERIALIZED VIEW IF NOT EXISTS` i en migrering gör ingenting utöver att ge en notice när en vy med det namnet redan finns, och dokumentationen ger ingen garanti för att den befintliga vyn liknar den nya definitionen. En migrering som ändrar frågan på det sättet låter den gamla stå kvar. Ta bort vyn med `DROP MATERIALIZED VIEW`, och skapa den igen i samma migrering.

**Uppdateringsjobb som överlappar.** Bara en uppdatering i taget kan köras mot en given materialiserad vy, även med `CONCURRENTLY`, så ett schema som är tätare än själva uppdateringen lämnar bara vyn längre efter. Mät hur lång tid en uppdatering tar, och schemalägg utifrån det.

**Att lita på radernas ordning.** Har den definierande frågan en `ORDER BY` kommer det första innehållet i den ordningen, men en uppdatering lovar inte att behålla den. Lägg `ORDER BY` i de frågor som läser vyn.

**Funktioner som tappar bort sina tabeller.** Medan den definierande frågan körs, när vyn skapas eller vid en uppdatering, sätter PostgreSQL tillfälligt `search_path` till `pg_catalog, pg_temp` och begränsar vad frågan får göra: en funktion som skapar temporära tabeller misslyckas. En funktion som hittar sina tabeller via ett okvalificerat namn kan klara varje test för hand och gå sönder först inne i uppdateringen. Kvalificera tabellnamnen med schemat i alla funktioner som vyn anropar.

**Fel roll kör jobbet.** En uppdatering kräver behörigheten `MAINTAIN` på den materialiserade vyn. Ett jobb som körs under en rapportroll med rätt att läsa vyn men inte att underhålla den misslyckas varje natt. Ge den roll som kör uppdateringen den behörigheten.

**En lagrad kopia där ett index hade räckt.** Är en fråga långsam för att ett index saknas döljer en materialiserad vy problemet och lägger gamla data ovanpå. Läs frågeplanen först. När en fråga bara behöver ett namn räcker en vanlig vy, och den är alltid aktuell; spara materialiserade vyer till resultat som är dyra att räkna fram och ofarliga att läsa lite gamla.

## Läs mer

- Materialized Views — kapitlet i PostgreSQL-manualen om hur materialiserade vyer förhåller sig till vyer och tabeller, med en försäljningssammanställning som uppdateras varje natt och ett exempel med en foreign data wrapper.
- CREATE MATERIALIZED VIEW — kommandoreferensen, bland annat `WITH NO DATA`, `IF NOT EXISTS` och den begränsade `search_path` medan frågan körs.
- REFRESH MATERIALIZED VIEW — uppdateringskommandot, villkoren för `CONCURRENTLY` och behörigheten `MAINTAIN` som det kräver.
- PostgreSQL 9.3 release notes — versionen från september 2013 som lade till materialiserade vyer.
- Introduction to materialized views — inkrementella och icke-inkrementella materialiserade vyer i BigQuery, automatisk uppdatering och smart tuning.
- Create indexed views — indexerade vyer i SQL Server, kraven på dem och vad de kostar för skrivningarna.

Ordlista

Välj utifrån era faktiska förutsättningar

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 →

# Gör begreppen praktiska

Fråga hur ”Vad är en materialiserad vy?” gäller i er organisation. Ett konkret exempel från ert nuvarande system räcker för att börja samtalet.

**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/glossary/materialized-view/**