Engineer

Materialiseret view

et view, der gemmer forespørgslens resultat som en tabel

Hvad det er

Et materialiseret view er en forespørgsel, hvis resultat databasen gemmer, så en læsning returnerer rækker, der er beregnet tidligere, i stedet for at køre forespørgslen igen. På engelsk hedder det materialized view, og det er også det navn, kommandoerne bærer. Det placerer sig mellem to ældre objekter. Et almindeligt view gemmer kun sin forespørgsel, og hver læsning kører den mod tabellerne nedenunder. En tabel oprettet med `CREATE TABLE AS` gemmer et resultat, men glemmer den forespørgsel, der skabte det. Et materialiseret view beholder begge dele: rækkerne, som læsningerne kommer fra, og forespørgslen, som kan køres igen for at erstatte dem.

PostgreSQL fik materialiserede views i version 9.3, udgivet 9. september 2013, og udgivelsesnoterne krediterer Kevin Grittner for arbejdet. Ifølge dokumentationen bruger de regelsystemet på samme måde som almindelige views, men gemmer resultatet i en tabellignende form. For forespørgselsparseren er et materialiseret view en relation som enhver tabel eller ethvert view, så en `SELECT` mod det læser de gemte rækker direkte; den gemte forespørgsel bruges kun, når viewet skal fyldes. Selve indholdet kan ikke ændres direkte. For at erstatte indholdet med et nyt forespørgselsresultat skal den gemte forespørgsel køres igen.

To kommandoer klarer det arbejde. `CREATE MATERIALIZED VIEW` definerer viewet og kører straks forespørgslen for at fylde det, medmindre den får `WITH NO DATA` med. `REFRESH MATERIALIZED VIEW` kører den gemte forespørgsel igen og erstatter hele indholdet med det nye resultat. Version 9.4, udgivet 18. december 2014, tilføjede `CONCURRENTLY`, som lader en opdatering køre uden at blokere de sessioner, der læser viewet. Begge kommandoer er PostgreSQL-udvidelser og ikke en del af SQL-standarden; andre databaser tilbyder idéen med deres egen syntaks og deres egne regler.

Brug begrebet i en beslutning

Brug »Hvad er et materialiseret view?« i en konkret beslutning. Vi kan knytte begrebet til det system, I bygger eller køber. At finde de forespørgsler og manglende indeks bag et system, der er blevet tungere hvert år, og rette dem, der betyder noget.

Se vores ydelser Kontakt os →

Undersøg antagelserne bag ordet

Brug forklaringen og det tilknyttede arbejde til at formulere spørgsmål til jeres projekt. Vi kan undersøge antagelserne sammen, før et værktøj vælges. 30 dages eldata på 180 ms — 180 ms til at aggregere en måneds måleraflæsninger

30 dages eldata på 180 ms Kontakt os →

Hvorfor det betyder noget

Nogle forespørgsler er dyre og bliver stillet ofte: et dashboard, der summerer et års ordrer pr. dag, en rapport, der joiner seks tabeller, en side, der rangerer produkter efter salg. Kørt mod kildetabellerne betaler hver anmodning for hele scanningen, joinen og aggregeringen igen, selv om svaret knap flytter sig fra den ene anmodning til den næste. Et materialiseret view betaler den pris én gang pr. opdatering, og hver læsning derefter er en scanning af en lille tabel, der er beregnet på forhånd. Ifølge PostgreSQL-dokumentationen er det ofte meget hurtigere at læse de gemte data end at gå til de underliggende tabeller, hvad enten det sker direkte eller gennem et view.

Prisen er aktualitet. Mellem to opdateringer viser viewet data, som de så ud ved den seneste, og i PostgreSQL opdaterer intet det, medmindre nogen, eller noget planlagt, beder om det. Mange anvendelser kan leve med det. Gårsdagens salg ændrer sig ikke i dag, og en månedsrapport har ikke brug for ordrerne fra det seneste minut. Det, et materialiseret view tvinger frem, er en udtrykkelig beslutning: hvor gammelt svaret må være, og hvis opgave det er at forhindre, at det bliver ældre.

Det giver også indekser et sted at bo: det gemte resultat kan få sine egne indekser, bygget til de forespørgsler, der læser det. PostgreSQL-dokumentationen viser en anden anvendelse af det samme: data hentet ind fra et andet system gennem en foreign data wrapper, som måske slet ikke understøtter indekser, kan kopieres over i et materialiseret view, der gør.

For et team, der holder sit skema normaliseret, er det ofte her, den flade form hører hjemme. Databasenormalisering gemmer hver oplysning ét sted og besvarer de fleste spørgsmål gennem joins. Et materialiseret view kan rumme

det joinede og summerede resultat, som rapporteringen vil have, mens kildetabellerne beholder deres ene kopi af hver oplysning, og hver skrivning stadig går ét sted hen.

Sådan ser det ud i praksis

Tag en webshops ordretabeller: `orders` med én række pr. ordre og den dato, den blev afgivet, og `order_lines` med varenummer, antal og stykpris for hver linje. Et salgsdashboard skal vise solgte enheder og omsætning pr. produkt pr. dag. Forespørgslen er kort, men over flere års ordrer læser den hver eneste linje:

```
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` -betingelsen udelader dagens salg, hvis tal stadig ændrer sig, så dashboardet kun viser hele dage; eksemplet i PostgreSQL's egen dokumentation behandler en dag med ufærdigt salg på samme måde. Kommandoen kører forespørgslen én gang og gemmer resultatet. Derefter kommer et indeks:

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

Indekset har to opgaver. Det lader dashboardet finde ét produkts dage uden at læse hele viewet, og det er det, der senere gør en samtidig opdatering mulig. Forespørgslen grupperer på dag og varenummer, så hvert par optræder én gang, og derfor kan indekset være unikt. Fra nu af læser dashboardet viewet som en tabel:

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

Hver nat, kort efter midnat, bringer et planlagt job det ajour:

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

Uden `CONCURRENTLY` bruger en opdatering, der rører mange rækker, typisk færre ressourcer og bliver hurtigere færdig, men den kan blokere dashboardets læsninger, indtil den er færdig. Med `CONCURRENTLY` fortsætter læsningerne under opdateringen, og dokumentationen bemærker, at det endda kan være det hurtigste valg, når kun få rækker ændres, som her, hvor der kommer én ny dag til hver nat. Valget har to betingelser. Viewet skal allerede indeholde data, og det skal have mindst ét unikt indeks bygget af rene kolonnenavne, som dækker alle rækker, altså intet udtryksindeks og ingen `WHERE` -betingelse. Indekset ovenfor opfylder den anden; at viewet blev oprettet med sine data, opfyldte den første.

Andre databaser trækker grænsen andre steder. I BigQuery vedligeholdes materialiserede views i baggrunden: ændringer i basistabellerne lægges trinvist til, uden at nogen beder om det, en forespørgsel mod viewet fletter de seneste ændringer ind, så resultatet forbliver aktuelt, og smart tuning kan sende en forespørgsel, der er skrevet mod basistabellerne, videre til viewet i stedet. Det gælder inkrementelle views, som kun accepterer et begrænset udvalg af SQL. Et view defineret med `allow_non_incremental_definition = true` accepterer de fleste forespørgsler, men hver opdatering kører hele forespørgslen, og resultaterne er ikke altid aktuelle.

SQL Server når et lignende resultat gennem indekserede views. Opretter man et unikt clustered indeks på et view, som skal være oprettet `WITH SCHEMABINDING` og have en deterministisk definition, gemmes resultatet på samme måde som en tabel med et clustered indeks, og forespørgselsoptimeringen kan bruge det, også til en forespørgsel, der ikke nævner viewet. SQL Server opdaterer det indekserede view sammen med hver insert, update og delete på de tabeller, det afhænger af, så der er ingen opdatering at planlægge; omkostningen flytter over på skrivningerne, og Microsofts dokumentation advarer om, at deres ydeevne kan falde betydeligt.

Den samme afvejning viser sig i produktionsarbejde. I en tidligere rolle udviklede grundlæggeren ETL/ELT-pipelines i PL/pgSQL, som brugte partitionerede tabeller og materialiserede views til at optimere læseydeevnen for store datasæt, sammen med primærnøgler og fremmednøgler for den referentielle integritet og unikke og sammensatte indeks til JOIN-operationerne.

Hvor det går galt

Forældede tal, der ser aktuelle ud. Et opdateringsjob, der holdt op med at køre for en uge siden, efterlader et view, der besvarer hver forespørgsel, hurtigt og forkert, og intet i resultatet fortæller, hvor gammelt det er. Lad jobbet registrere tidspunktet for hver vellykket opdatering et sted, hvor dashboardet kan vise det, og send en alarm, når tidspunktet er ældre, end læserne har accepteret.

En opdatering, der blokerer læserne. En almindelig `REFRESH MATERIALIZED VIEW` over et stort view kan blokere alle sessioner, der prøver at læse det, indtil opdateringen er færdig. Tilføj et unikt indeks og brug `CONCURRENTLY`, eller læg de almindelige opdateringer i timer, hvor ingen læser viewet. Når de fleste rækker ændres ved hver kørsel, kan den almindelige opdatering stadig være den billigste.

`CONCURRENTLY` **bliver afvist.** Valget kræver et unikt indeks på rene kolonner, som dækker alle rækker. Et unikt indeks på `lower(email)`, eller et, der er begrænset af en `WHERE`-betingelse, tæller ikke. Det kræver også et view, der allerede indeholder data, så det fejler på et view oprettet `WITH NO DATA`, og de to valg kan ikke kombineres. Tilføj et indeks, der opfylder kravene, og fyld viewet én gang med en almindelig opdatering.

Et view, der ikke kan scannes. `WITH NO DATA` opretter viewet uden at køre forespørgslen og markerer det som ikke-scanbart, og enhver forespørgsel mod det fejler frem til den første opdatering. Det kan være det rigtige valg i en migrering, der ikke skal vente på en lang forespørgsel, men så skal den første opdatering planlægges, ikke glemmes.

En ændret definition, der aldrig nåede frem. `CREATE MATERIALIZED VIEW IF NOT EXISTS` i en migrering gør intet ud over at udsende en notice, når et view med det navn allerede findes, og dokumentationen garanterer ikke, at det eksisterende view ligner den nye definition. En migrering, der ændrer forespørgslen på den måde, lader den gamle blive stående. Fjern viewet med `DROP MATERIALIZED VIEW`, og opret det igen i samme migrering.

Opdateringsjob, der overlapper. Der kan kun køre én opdatering ad gangen mod et givet materialiseret view, også med `CONCURRENTLY`, så en tidsplan, der er tættere end selve opdateringen, efterlader kun viewet længere bagud. Mål, hvor lang tid en opdatering tager, og læg tidsplanen ud fra det.

At stole på rækkernes rækkefølge. Har den definerende forespørgsel en `ORDER BY`, kommer det første indhold i den rækkefølge, men en opdatering lover ikke at bevare den. Sæt `ORDER BY` i de forespørgsler, der læser viewet.

Funktioner, der mister deres tabeller. Mens den definerende forespørgsel kører, ved oprettelsen eller ved en opdatering, sætter PostgreSQL midlertidigt `search_path` til `pg_catalog, pg_temp` og begrænser, hvad forespørgslen må: en funktion, der opretter midlertidige tabeller, fejler. En funktion, der finder sine tabeller ved et ukvalificeret navn, kan bestå hver test i hånden og først gå i stykker inde i opdateringen. Kvalificér tabelnavnene med skemaet i alle funktioner, viewet kalder.

Den forkerte rolle kører jobbet. En opdatering kræver privilegiet `MAINTAIN` på det materialiserede view. Et job, der kører som en rapporteringsrolle, der må læse viewet, men ikke vedligeholde det, fejler hver nat. Giv den rolle, der kører opdateringen, det privilegium.

En gemt kopi, hvor et indeks ville være nok. Er en forespørgsel langsom, fordi der mangler et indeks, skjuler et materialiseret view problemet og lægger forældede data oven i. Læs forespørgselsplanen først. Når en forespørgsel blot skal have et navn, er et almindeligt view nok, og det er altid aktuelt; gem materialiserede views til resultater, der er dyre at beregne og ufarlige at læse lidt gamle.

Læs mere

- Materialized Views — PostgreSQL's kapitel om, hvordan materialiserede views forholder sig til views og tabeller, med et salgsoverblik, der opdateres hver nat, og et eksempel med en foreign data wrapper.
- CREATE MATERIALIZED VIEW — kommandoreferencen, herunder `WITH NO DATA`, `IF NOT EXISTS` og den begrænsede `search_path`, mens forespørgslen kører.
- REFRESH MATERIALIZED VIEW — opdateringskommandoen, betingelserne for `CONCURRENTLY` og privilegiet `MAINTAIN`, som den kræver.
- PostgreSQL 9.3 release notes — udgivelsen fra september 2013, der tilføjede materialiserede views.
- Introduction to materialized views — BigQuerys inkrementelle og ikke-inkrementelle materialiserede views, automatisk opdatering og smart tuning.
- Create indexed views — SQL Servers indekserede views, kravene til dem og deres pris for skrivningerne.

Ordliste

Vælg ud fra jeres faktiske vilkår

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 →

Gør begreberne praktiske

Spørg, hvordan »Hvad er et materialiseret view?« gælder i jeres organisation. Et konkret eksempel fra jeres nuværende system er nok til at begynde samtalen.

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/glossary/materialized-view/