Engineer

Матеріалізоване подання

подання, що зберігає результат свого запиту як таблицю

Що це таке

Матеріалізоване подання — це запит, результат якого база даних зберігає, тож читання повертає рядки, обчислені раніше, а не виконує запит заново. Трапляється й назва «матеріалізоване представлення», а англійською це materialized view, як і в назвах команд. Воно стоїть між двома старшими об'єктами. Звичайне подання зберігає лише свій запит, і кожне читання виконує його над таблицями, що лежать під ним. Таблиця, створена командою `CREATE TABLE AS`, зберігає результат, але забуває запит, який його дав. Матеріалізоване подання тримає і те, і те: рядки, з яких читають, і запит, який можна виконати знову, щоб їх замінити.

PostgreSQL отримав матеріалізовані подання у версії 9.3, що вийшла 9 вересня 2013 року, і примітки до випуску приписують цю роботу Кевіну Ґріттнеру (Kevin Grittner). Згідно з документацією, вони використовують систему правил так само, як звичайні подання, але зберігають результат у формі, подібній до таблиці. Для парсера запитів матеріалізоване подання — це відношення, як будь-яка таблиця чи подання, тож `SELECT` до нього читає збережені рядки напряму; збережений запит потрібен лише тоді, коли подання заповнюють. Сам вміст не можна змінити безпосередньо. Щоб замінити вміст новим результатом запиту, збережений запит потрібно виконати знову.

Цю роботу виконують дві команди. `CREATE MATERIALIZED VIEW` визначає подання й одразу виконує запит, щоб його заповнити, якщо не вказано `WITH NO DATA`. `REFRESH MATERIALIZED VIEW` виконує збережений запит знову й замінює весь вміст новим результатом. Версія 9.4, що вийшла 18 грудня 2014 року, додала параметр `CONCURRENTLY`, який дає змогу оновлювати подання, не блокуючи сеанси, що його читають. Обидві команди — розширення PostgreSQL, а не частина стандарту SQL; інші бази даних пропонують цю ідею зі своїм синтаксисом і своїми правилами.

Застосуйте поняття до рішення

Застосуйте «Що таке матеріалізоване подання?» до конкретного рішення. Ми можемо пов'язати поняття із системою, яку ви створюєте або купуєте. Пошук запитів і відсутніх індексів у системі, що рік за роком важчає, та виправлення тих, що справді мають значення.

Переглянути наші послуги Зв'язатися з нами →

Перевірте припущення за терміном

Використайте пояснення та пов'язані роботи, щоб сформулювати запитання для проєкту. Ми можемо перевірити припущення разом до вибору інструмента. 30 днів даних про електроенергію за 180 мс — 180 ms на агрегацію місяця показань лічильників

30 днів даних про електроенергію за 180 мс Зв'язатися з нами →

Чому це важливо

Деякі запити дорогі, а потрібні часто: дашборд, що підсумовує замовлення за рік по днях, звіт, що з’єднує шість таблиць, сторінка, що ранжує товари за продажами. Якщо виконувати їх над вихідними таблицями, кожне звернення знову оплачує повне сканування, з’єднання й агрегацію, хоча відповідь від одного звернення до наступного ледве змінюється. Матеріалізоване подання платить цю ціну один раз за оновлення, а кожне читання після цього — це сканування невеликої таблиці, обчисленої заздалегідь. Документація PostgreSQL зазначає, що читати збережені дані часто набагато швидше, ніж звертатися до таблиць під ними напряму чи через подання.

Ціна — свіжість даних. Між двома оновленнями подання показує дані такими, якими вони були під час останнього, і в PostgreSQL його оновлює лише той, хто про це попросить: людина або завдання за розкладом. Багатьом сценаріям цього досить. Учорашні продажі сьогодні не зміняться, а місячному звіту не потрібні замовлення останньої хвилини. Матеріалізоване подання змушує ухвалити явне рішення: наскільки старою може бути відповідь і хто відповідає за те, щоб вона не старішала далі.

Воно також дає індексам місце, де жити: збережений результат може мати власні індекси, побудовані під запити, що його читають. Документація PostgreSQL показує ще одне застосування того самого: дані, отримані з іншої системи через обгортку зовнішніх даних (foreign data wrapper), який може взагалі не підтримувати індексів, можна скопіювати в матеріалізоване подання, яке їх підтримує.

Для команди, що тримає свою схему нормалізованою, саме тут часто місце для пласкої форми. Нормалізація баз даних зберігає кожен факт в одному місці й відповідає на більшість запитань через з’єднання. Матеріалізоване

подання може тримати з’єднаний і підсумований результат, якого хоче звітність, тоді як вихідні таблиці зберігають свою єдину копію кожного факту, а кожен запис і далі йде в одне місце.

Як це виглядає на практиці

Візьмімо таблиці замовлень інтернет-магазину: `orders`, з одним рядком на замовлення й датою, коли його оформлено, і `order_lines`, з артикулом, кількістю й ціною за одиницю для кожної позиції. Дашборд продажів має показувати продані одиниці й виручку за кожним товаром за кожен день. Запит короткий, але на замовленнях за кілька років він читає кожну позицію:

```
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;
```

Умова `WHERE` відкидає сьогоднішній день, цифри якого ще змінюються, тож дашборд показує лише повні дні; приклад у власній документації PostgreSQL так само обходиться з днем незавершених продажів. Команда виконує запит один раз і зберігає результат. Далі йде індекс:

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

Індекс має два завдання. Він дає дашборду знайти дні одного товару, не читаючи всього подання, і саме він згодом робить можливим оновлення без блокування читання. Запит групує за днем і артикулом, тож кожна пара трапляється один раз, і тому індекс може бути унікальним. Відтепер дашборд читає подання як таблицю:

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

Щоночі, невдовзі після опівночі, завдання за розкладом оновлює його:

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

Без `CONCURRENTLY` оновлення, що зачіпає багато рядків, зазвичай використовує менше ресурсів і завершується швидше, але може блокувати читання дашборда, доки не завершиться. З `CONCURRENTLY` читання тривають під час оновлення, а документація зауважує, що цей варіант може бути навіть швидшим, коли змінюється небагато рядків, як тут, де щоночі додається один новий день. Параметр має дві умови. Подання вже має містити дані, і воно мусить мати принаймні один унікальний індекс, побудований лише з назв стовпців, що охоплює всі рядки, тобто без виразів в індексі й без умови `WHERE`. Індекс вище виконує другу умову; те, що подання створили разом із даними, виконало першу.

Інші бази даних проводять межу деінде. У BigQuery матеріалізовані подання підтримуються у фоні: зміни в базових таблицях додаються інкрементно, без жодного прохання, запит до подання підмішує найсвіжіші зміни, тож результат лишається актуальним, а smart tuning може перенаправити до подання запит, написаний до базових таблиць. Це стосується інкрементних подань, які приймають лише обмежену частину SQL. Подання, визначене з `allow_non_incremental_definition = true`, приймає більшість запитів, але кожне оновлення виконує запит повністю, і результати не завжди свіжі.

SQL Server досягає схожого результату через індексовані подання. Якщо створити унікальний кластеризований індекс на поданні, яке має бути створене `WITH SCHEMABINDING` і мати детерміноване визначення, його результат зберігається так само, як таблиця з кластеризованим індексом, а оптимізатор запитів може використати його навіть для запиту, який подання не згадує. SQL Server оновлює індексоване подання разом із кожною вставкою, зміною й видаленням у таблицях, від яких воно залежить, тож оновлення за розкладом не потрібне; вартість переходить на записи, і документація Microsoft попереджає, що їхня продуктивність може суттєво погіршитися.

Той самий компроміс видно й у виробничій роботі. У попередній ролі засновниця розробила ETL/ELT-пайплайни на PL/pgSQL, які використовували партиціоновані таблиці й матеріалізовані подання, щоб оптимізувати

продуктивність читання великих наборів даних, а поряд із ними — обмеження первинного та зовнішнього ключів для посилальної цілісності й унікальні та складені індекси для JOIN.

Де виникають проблеми

Застарілі цифри, що виглядають свіжими. Завдання оновлення, яке перестало запускатися тиждень тому, залишає подання, що відповідає на кожен запит швидко й неправильно, і ніщо в результаті не каже, наскільки він старий. Нехай завдання записує час кожного успішного оновлення туди, де дашборд може його показати, і нехай спрацьовує тривога, коли цей час старший, ніж погодилися терпіти читачі.

Оновлення, що блокує читачів. Звичайний `REFRESH MATERIALIZED VIEW` для великого подання може блокувати всі сеанси, що намагаються його читати, доки оновлення не завершиться. Додайте унікальний індекс і використовуйте `CONCURRENTLY` або плануйте звичайні оновлення на години, коли подання ніхто не читає. Якщо більшість рядків змінюється під час кожного запуску, звичайне оновлення все одно може бути дешевшим.

`CONCURRENTLY` **відхилено.** Параметр вимагає унікального індексу лише за стовпцями, що охоплює всі рядки. Унікальний індекс на `lower(email)` або індекс з умовою `WHERE` не підходить. Також потрібне подання, яке вже містить дані, тож на поданні, створеному `WITH NO DATA`, команда завершується помилкою, а поєднувати ці два параметри не можна. Додайте індекс, що відповідає вимогам, і один раз заповніть подання звичайним оновленням.

Подання, яке не можна сканувати. `WITH NO DATA` створює подання, не виконуючи запиту, і позначає його як непридатне для сканування, тож будь-який запит до нього завершується помилкою до першого оновлення. Це може бути правильним вибором у міграції, яка не повинна чекати на довгий запит, але тоді перше оновлення треба спланувати, а не забути.

Змінене визначення, яке так і не дійшло. `CREATE MATERIALIZED VIEW IF NOT EXISTS` у міграції, коли подання з такою назвою вже є, лише видає notice і більше нічого не робить, а документація не гарантує, що наявне подання схоже на нове визначення. Міграція, яка так змінює запит, лишає старий на місці. Видаліть подання командою `DROP MATERIALIZED VIEW` і створіть його знову в тій самій міграції.

Завдання оновлення, що накладаються. Для одного матеріалізованого подання одночасно може виконуватися лише одне оновлення, навіть із `CONCURRENTLY`, тож розклад, щільніший за саме оновлення, лише дедалі більше відкидає подання назад. Виміряйте, скільки триває оновлення, і складайте розклад відповідно.

Довіра до порядку рядків. Якщо визначальний запит має `ORDER BY`, перший вміст з'являється в цьому порядку, але оновлення не обіцяє його зберегти. Пишіть `ORDER BY` у запитах, що читають подання.

Функції, що гублять свої таблиці. Поки виконується визначальний запит, під час створення чи оновлення, PostgreSQL тимчасово встановлює `search_path` у `pg_catalog, pg_temp` і обмежує, що запиту дозволено: функція, яка створює тимчасові таблиці, завершується помилкою. Функція, що знаходить свої таблиці за некваліфікованим іменем, може пройти кожну ручну перевірку і зламатися лише всередині оновлення. Указуйте схему перед іменами таблиць у всіх функціях, які викликає подання.

Завдання запускає не та роль. Оновлення вимагає привілею `MAINTAIN` на матеріалізованому поданні. Завдання, що працює від імені ролі для звітів, якій дозволено читати подання, але не обслуговувати його, щоночі завершується помилкою. Надайте цей привілей ролі, від імені якої виконується оновлення.

Збережена копія там, де вистачило б індексу. Якщо запит повільний через відсутній індекс, матеріалізоване подання приховує проблему й додає до неї застарілі дані. Спершу прочитайте план запиту. Коли потрібно лише дати запиту назву, вистачить звичайного подання, і воно завжди актуальне; залишайте матеріалізовані подання для результатів, які дорого обчислювати і які безпечно читати трохи застарілими.

Що почитати далі

- Materialized Views — розділ документації PostgreSQL про те, як матеріалізовані подання співвідносяться з поданнями й таблицями, з підсумком продажів, що оновлюється щоночі, і прикладом із foreign data wrapper.
- CREATE MATERIALIZED VIEW — довідка з команди, зокрема `WITH NO DATA`, `IF NOT EXISTS` та обмежений `search_path` під час виконання запиту.

- REFRESH MATERIALIZED VIEW — команда оновлення, умови для `CONCURRENTLY` і привілей `MAINTAIN`, якого вона вимагає.
- PostgreSQL 9.3 release notes — випуск вересня 2013 року, який додав матеріалізовані подання.
- Introduction to materialized views — інкрементні й неінкрементні матеріалізовані подання в BigQuery, автоматичне оновлення і smart tuning.
- Create indexed views — індексовані подання в SQL Server, вимоги до них і їхня ціна для записів.

Словник

Обирайте за фактичними умовами

Надайте репрезентативний набір даних, запитання до нього та поточне вузьке місце. Узгодимо вимірювання якості, пропускної здатності та відновлення.

Дивитися, скільки це коштує Зв'язатися з нами →

Зробіть термінологію практичною

Запитайте, як «Що таке матеріалізоване подання?» стосується вашої організації. Конкретного прикладу з поточної системи достатньо для початку розмови.

Переглянути наші послуги

welcome@engineer.company +45 50 10 61 84 Копенгаген, Данія

Візьміть цю сторінку із собою

Візитівка

Дбайливо зроблено з любов'ю — D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Авторське право © 2025 – дотепер · Engineer ApS · Усі права захищено.

Надруковано з https://engineer.company/uk/glossary/materialized-view/