Engineer

Індекс бази даних

структура, що знаходить рядки без повного читання таблиці

Що це таке

Індекс бази даних — це окрема структура, що зберігається поряд із таблицею й дає базі даних змогу одразу дістатися до рядків, які задовольняють умову, замість читати таблицю від першого рядка до останнього. У повсякденній мові, як і в більшій частині документації PostgreSQL, його називають просто індексом. Документація порівнює його з предметним покажчиком наприкінці книжки: читач, якого цікавить одна тема, знаходить її там і відкриває вказані сторінки, а не гортає весь том.

Візьмімо таблицю замовлень і умову `WHERE customer_id = 42`. Без індексу на `customer_id` PostgreSQL має лише один спосіб відповісти: послідовне сканування, яке читає кожен рядок і залишає ті, що підходять. Воно коштує однаково, хоч у клієнта два замовлення, хоч дві тисячі, і дорожчає разом із таблицею. З індексом на цьому стовпці PostgreSQL спускається кількома рівнями дерева пошуку до записів для 42 і читає тільки ці рядки. Індекс створюють однією інструкцією, `DROP INDEX` його прибирає, і те, і інше можна зробити будь-коли.

Коли індекс уже існує, PostgreSQL оновлює його разом із таблицею під час кожного запису, а планувальник запитів для кожного запиту окремо вирішує, чи дешевше скористатися індексом, ніж сканувати. Сам SQL індексу не називає, тож один індекс може пришвидшити `SELECT`, `UPDATE` чи `DELETE` з умовою пошуку, а також з’єднання таблиць.

PostgreSQL має кілька типів, кожен для свого роду питань:

- **B-tree** — тип за замовчуванням, саме його будує `CREATE INDEX`, якщо не вказано інше. Він відповідає на рівність і порівняння діапазонів (`<`, `<=`, `=`, `>=`, `>`), `BETWEEN`, `IN`, `IS NULL` і `LIKE`, шаблон якого прив'язаний до початку рядка. Записи в ньому впорядковані, тож він може віддавати рядки вже відсортованими.
- **Hash** зберігає 32-бітний хеш-код кожного значення й відповідає лише на рівність.
- **GiST** і **SP-GiST** — каркаси для даних на кшталт геометрії, де питання звучить як «перетинається», «містить» чи «найближче до», а не «дорівнює».
- **GIN** — інвертований індекс для значень, що складаються з багатьох частин, наприклад масивів; він знаходить рядки, значення яких містить заданий елемент.
- **BRIN** зберігає лише зведення для кожного діапазону блоків таблиці. Він пасує стовпцю, значення якого йдуть у фізичному порядку рядків, як це часто буває з часом вставки.

Крім типу, індекс можна підлаштувати під питання. Унікальний індекс не пропускає дублікатів. Складений індекс охоплює кілька стовпців, до 32. Частковий індекс має умову `WHERE` і містить лише рядки, що їй відповідають. Індекс за виразом зберігає обчислене значення на кшталт `lower(email)`. А список `INCLUDE` додає до індексу стовпці, за якими не шукають, щоб запит міг отримати відповідь із самого індексу.

Застосуйте поняття до рішення

Застосуйте «Що таке індекс бази даних?» 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

Різниця, яку дає індекс, зростає разом із таблицею. На маленькій таблиці сканування дешеве, і планувальник часто цілком слушно обирає саме його: прочитати кілька сторінок поспіль дешевше, ніж добиратися до них деревом. На десятках мільйонів рядків різниця між скануванням і пошуком — це різниця між сторінкою, що відкривається, і тайм-аутом. Запит, який був швидким на запуску, через рік може стати повільним, хоча жоден рядок коду не змінився, — просто таблиця, яку він читає, виросла.

Індекси не безкоштовні, і рахунок приходить під час запису. Кожна вставка додає запис до кожного індексу таблиці. Оновлення може уникнути цього лише як heap-only tuple, або HOT-оновлення: нова версія рядка лишається на тій самій сторінці й не потребує нових записів в індексах, а PostgreSQL дозволяє таке тільки тоді, коли оновлення не змінює жодного стовпця, який використовують індекси таблиці. Індекси BRIN не враховуються, бо зберігають лише зведення. Тож кожен індекс звужує дешевий шлях, і саме тому документація радить видаляти індекси, якими користуються рідко або ніколи. Таблиця з дюжиною індексів платить дюжину разів за кожну вставку.

Побудова індексу теж має ціну. Звичайний `CREATE INDEX` дозволяє іншим сеансам читати таблицю, але блокує запис у неї, доки побудова не завершиться, а документація попереджає, що на дуже великій таблиці це може тривати багато годин.

І нарешті, індекс корисний рівно настільки, наскільки планувальник вирішить ним скористатися, а планувальник спирається на статистику, яку збирає `ANALYZE`. Питати треба не «чи є індекс?», а «чи користуються ним запити, що мають значення?», і відповідь на це дає `EXPLAIN`.

Як це виглядає на практиці

Сторінка облікового запису інтернет-магазину показує двадцять останніх замовлень клієнта:

```
SELECT id, created_at, total
FROM   orders
WHERE  customer_id = 42
ORDER  BY created_at DESC
LIMIT  20;
```

Якщо є лише первинний ключ на `id`, PostgreSQL мусить прочитати всю таблицю `orders`, відібрати рядки клієнта, відсортувати їх і відкинути все, крім двадцяти. `EXPLAIN` це показує: `Seq Scan` по `orders`, де умова стоїть як `Filter`, а над ним сортування. `EXPLAIN ANALYZE` ще й виконує запит і виводить справжню кількість рядків і час поруч з оцінками.

Індекс, що відповідає цьому питанню, ставить стовпець рівності першим, а стовпець сортування — другим:

```
CREATE INDEX CONCURRENTLY orders_customer_created_idx
ON orders (customer_id, created_at);
```

Запустіть `EXPLAIN` ще раз, і сканування таблиці стане скануванням індексу `orders_customer_created_idx`, де `customer_id = 42` тепер показано як `Index Cond`, а не як фільтр. Зникає й сортування. B-tree тримає записи клієнта 42 упорядкованими за `created_at`, PostgreSQL може прочитати їх у зворотному порядку, щоб отримати найновіші першими, а з `LIMIT` зупиняється після двадцяти записів і решти не торкається.

Порядок стовпців має значення. Той самий індекс обслуговує запит лише за `customer_id`, бо це його перший стовпець, а от звіт про всі вчорашні замовлення, що фільтрує тільки за `created_at`, отримує від нього значно менше. У складеному B-tree частину індексу, яку доведеться прочитати, звужують рівність на перших стовпцях і діапазон на наступному. PostgreSQL 18 уміє skip scan, тобто повторює пошук для кожного окремого значення першого стовпця, але планувальник зазвичай обирає його лише тоді, коли таких значень мало, а ідентифікаторів клієнтів багато. Якщо звіт важливий, йому потрібен власний індекс на `created_at`. А в зворотному порядку, `(created_at, customer_id)`, індекс розкидав би замовлення клієнта 42 по всьому своєму обсягу.

Нотатка про те, як оптимізувати базу, перш ніж її шардувати, ставить індекси першими серед виправлень, які впорядковує за віддачею, щойно `EXPLAIN (ANALYZE, BUFFERS)` показав, які сканування виконує навантажений запит. Її власний приклад — частковий індекс `units_unsold`, що містить лише одиниці, які ще в продажу, тож під час розпродажу він зменшується, а не росте. Нотатка рахує й ціну. На її таблиці всі 2000 резервувань, що задавали `reserved`, `reserved_at` і `order_id`, були HOT-оновленнями, а з 2000 оновлень, що задавали `sold`, стовпець, який читає частковий індекс, — жодне. Вона радить видаляти індекси, у яких `idx_scan` у `pg_stat_user_indexes` так і лишається нулем.

Кейс про кращий вибір ключа показує, чому значення, які потрапляють у B-tree, важать не менше за стовпці. Випадкові UUID розпорошують вставки по всьому індексу, що може спричиняти фрагментацію. На платформі впорядкований за часом UUID v7 став типовим вибором для первинних ключів сутностей; мітка часу в старших бітах забезпечує часовий порядок нових ідентифікаторів. Зменшення фрагментації та пришвидшення запитів

були цілями проєктування; зафіксований результат — налаштоване правило вибору ключів, а порівняльного бенчмарку робочого навантаження до й після зміни в кейсі немає.

Де виникають проблеми

Новий індекс зупиняє запис у таблицю. Звичайний `CREATE INDEX` на навантаженій таблиці блокує вставки, оновлення й видалення на весь час побудови, і рутинне розгортання перетворюється на простій. Використовуйте `CREATE INDEX CONCURRENTLY` : він сканує таблицю двічі й чекає на транзакції, які могли б скористатися індексом, тож триває значно довше, зате запис не зупиняється. Його не можна виконати всередині блоку транзакції, тож інструменту міграцій, який загортає кожен крок у транзакцію, треба окремо вказати виконати цю інструкцію самостійно. На одній таблиці одночасно може будуватися лише один такий індекс.

Паралельна побудова падає й лишає недійсний індекс. Взаємоблокування або дубльоване значення під час побудови унікального індексу змушує `CREATE INDEX CONCURRENTLY` завершитися помилкою, але розпочатий індекс лишається в каталозі з позначкою недійсного. Запити його ігнорують, проте кожен запис і далі платить за його підтримку, а недійсний унікальний індекс може й надалі забезпечувати унікальність. Команда `\d` у psql показує його з позначкою `INVALID` після визначення. Видаліть його й запустіть паралельну побудову знову або перебудуйте через `REINDEX INDEX CONCURRENTLY` .

Запит ховає стовпець усередині функції. Індекс на `email` нічим не допоможе умові `WHERE lower(email) = 'ana@example.com'` , бо в індексі лежить `email` , а не `lower(email)` . Проіндексуйте саме той вираз, який використовує запит:

```
CREATE INDEX users_lower_email_idx ON users (lower(email));
```

Вираз обчислюється під час кожної вставки й кожного оновлення, що не є HOT, тому документація радить індекси за виразами там, де швидкість вибірки важить більше, ніж швидкість запису.

Накопичуються індекси, якими ніхто не користується. Кожен індекс, доданий «про всяк випадок», коштує під час кожного запису й може унеможливити HOT-оновлення. `pg_stat_user_indexes` рахує сканування, розпочаті на кожному індексі, у стовпці `idx_scan`, а час останнього записує в `last_idx_scan`:

```
SELECT relname, indexrelname, idx_scan, last_idx_scan
FROM   pg_stat_user_indexes
WHERE  idx_scan = 0
ORDER  BY relname;
```

Перш ніж видаляти, придивіться. Індекс, на якому тримається первинний ключ чи обмеження унікальності, робить свою справу, навіть якщо жоден запит його не читає, а нуль мало що означає на сервері, який щойно почав приймати трафік.

Планувальник ігнорує добрий індекс. Після масового завантаження статистика відстає від даних, оцінки хибні, і сканування перемагає на папері. Виконайте `ANALYZE`: документація вважає, що без нього судити про використання індексів марно. Тестуйте й на реалістичних обсягах: на сотні рядків, які вміщуються на одній сторінці диска, жоден індекс не обжене читання цієї сторінки. Щоб з'ясувати, чи індекс узагалі може бути використаний, задайте `enable_seqscan = off` у тестовому сеансі й запустіть `EXPLAIN` знову. Якщо план і далі сканує таблицю, умова не відповідає індексу.

Що почитати далі

- Indexes: Introduction — для чого потрібен індекс, як його створити й видалити і чим він обтяжує кожен запис.
- Index Types — B-tree, Hash, GiST, SP-GiST, GIN і BRIN та оператори, на які відповідає кожен із них.
- Multicolumn Indexes — як порядок стовпців визначає, що може звузити B-tree, і коли спрацьовує skip scan.
- CREATE INDEX — повний синтаксис, зокрема `CONCURRENTLY`, `INCLUDE`, часткові індекси та відновлення після невдалої побудови.
- Using EXPLAIN — як читати план: типи сканування, оцінки вартості й справжні цифри з `EXPLAIN ANALYZE`.
- Examining Index Usage — чому спершу потрібні `ANALYZE` і реальні дані та як перевірити, чи індекс узагалі можна використати.

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 індекс бази даних?» 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/database-index/