Лабораторна робота 12. Індексування в PostgreSQL. Пояснення плану виконання запиту

Мета: Ознайомити студентів із видами індексів та приципами їх роботи в СУБД PostgreSQL, а також із механізмами, що дозволяють отримати інформацію про виконання запиту за допомогою команди EXPLAIN.

Завдання:

1. Ознайомитися із принципами роботи індексів, їх різновидами в СУБД PostgreSQL, а також із командами, що дозволяють вивчити план виконання запиту;
2. Виконати SQL-оператори для створення індексів до таблиць, подивитися на їх роботу;
3. Навчитися аналізувати план виконання запиту і давати пояснення основним його елементам.

Результат:

Студенти повинні подати SQL-скрипти, що відображають створення індексів до таблиць, а також плани виконання запитів до і після створення індексних структур даних відповідно до завдання та предметної області, їх опис, а також звіт з результатами тестування.

Теоретичні відомості про індекси в PostgreSQL

У PostgreSQL індекс – це структура даних, яка збільшує швидкість пошуку даних, забезпечуючи швидкий спосіб пошуку рядків у таблиці. Індекс у PostgreSQL працює як покажчик у книзі, надаючи швидке посилання на сторінку, де можна знайти певний вміст (як приклад – організація різноманітних словників, де слова розташовані в алфавітному порядку і маємо розділ словника по буквам абетки).

Індекс – це окрема структура даних, яка підвищує швидкість отримання даних із таблиці за рахунок додаткових записів і зберігання, необхідних для її підтримки. Індекс дозволяє підвищити продуктивність запитів при його належному використанні, особливо у великих таблицях.

Припустимо, у вас є таблиця `contacts`, що містить `id` первинний ключ, ім’я особи та її контактний номер:

```
CREATE TABLE contacts (
   id INT PRIMARY KEY,
   name VACHAR(255) NOT NULL,
   phone VARCHAR(10) NOT NULL
);
```

І ви виконуєте наступний запит, щоб знайти контакти за іменами:

```
SELECT * FROM contacts
WHERE name = 'John Doe';
```

PostgreSQL має сканувати всю таблицю `contacts` щоб знайти відповідні рядки. Якщо таблиця `contacts` містить багато рядків, метод пошуку відповідних рядків є неефективним. Однак, маючи індекс у стовпці `name`, PostgreSQL може використовувати більш ефективний метод для пошуку відповідних рядків.

Наступний оператор створює індекс під назвою `contacts_name` для стовпця `name` таблиці `contacts`:

```
CREATE INDEX contacts_name
ON contacts (name);
```

Після створення індексу в стовпці `name`, PostgreSQL витягує дані зі стовпця `name` таблиці `contacts` і вставляє його в індексну структуру даних. Цей процес може зайняти час, залежно від кількості рядків у таблиці `contacts`. За замовчуванням PostgreSQL дозволяє вибір даних із таблиці під час створення індексу, але блокує операції INSERT, UPDATE і DELETE. Під час виконання наступного оператора SELECT PostgreSQL може використовувати індекс `contacts_name` для швидкого пошуку відповідних рядків у таблиці `contacts`:

```
SELECT * FROM contacts
WHERE name = 'John Doe';
```

Після створення індексу PostgreSQL має підтримувати його синхронізацію з таблицею. Наприклад, під час вставки, оновлення чи видалення даних із таблиці `contacts` PostgreSQL оновлює індекс, щоб відповідним чином відобразити зміни.

Типи індексів в PostgreSQL

PostgreSQL пропонує різні типи індексів, кожен з яких призначений для конкретних сценаріїв даних і шаблонів запитів. Розуміючи ці типи індексів, можна ефективніше підвищити продуктивність запитів.

Індекс B-дерева (B-tree)

B-дерево є типовим типом індексу в PostgreSQL. B-дерево – це самобалансуюче дерево, яке підтримує відсортовані дані та дозволяє здійснювати пошук, вставлення, видалення та послідовний доступ у логарифмічному часі. Планувальник запитів PostgreSQL розглядатиме використання індексу B-дерева щоразу, коли стовпці індексів залучаються до порівняння, яке використовує один із таких операторів:

- оператори порівняння (<, <=, =, >=, >);
- а також конструкції з операторами BETWEEN, IN, IS NULL, IS NOT NULL.

Крім того, планувальник запитів може використовувати індекс B-дерева для запитів, які включають оператор зіставлення шаблону LIKE та ~, якщо шаблон є константою та є прив’язкою на початку шаблону, наприклад:

- column_name LIKE 'foo%’;
- column_name LIKE ‘bar%’;
- column_name ~ '^foo'.

Крім того, планувальник запитів розгляне використання індексів B-дерева для ILIKE та ~*, якщо шаблон починається з неалфавітного символу, на який не впливає перетворення верхнього/нижнього регістру.

Хеш-індекс

Хеш-індекси підтримують 32-розрядний хеш-код, створений із значень індексованих стовпців. Тому хеш-індекси можуть обробляти лише прості порівняння рівності (=). Це означає, що щоразу, коли індексований стовпець бере участь у порівнянні за допомогою оператора рівності (=), планувальник запитів розглядатиме використання хеш-індексу.

Щоб створити хеш-індекс, використовується оператор CREATE INDEX із типом індексу HASH у реченні USING наступним чином:

```
CREATE INDEX index_name
ON table_name USING HASH (indexed_column);
```

Індекс GIN

GIN означає узагальнені інвертовані індекси (**g**eneralized **in**verted indexes). Індекси GIN – це інвертовані індекси, які підходять для таких складених значень, як масиви, дані JSONB, hstore, типи діапазонів і повнотекстовий пошук. Оскільки індекс GIN зберігає окремий запис для кожного компонента, він може обробляти запити, які перевіряють існування певного компонента.

Індекс GiST

GiST означає узагальнене дерево пошуку (**G**eneralized **S**earch **T**ree). Індекси GiST дозволяють будувати загальні деревоподібні структури. Індекси GiST є універсальними та підтримують широкий спектр типів даних, включаючи геометричні та повнотекстові дані. Індекси GiST дозволяють використовувати різні стратегії пошуку, такі як пошук найближчого сусіда та частковий пошук (partial match), що робить їх корисними для спеціалізованих програм.

Індекс SP-GiST

SP-GiST розшифровується як GiST з розділеним простором (**s**pace-**p**artitioned GiST). Індекси SP-GiST корисні для індексування даних з ієрархічними структурами або складними типами даних. Індекси SP-GiST поділяють простір індексів на області, що не перекриваються, пропонуючи ефективні можливості пошуку для спеціалізованих структур даних.

SP-GiST підтримує розділені дерева пошуку, які полегшують розробку широкого діапазону різноманітних незбалансованих структур даних. Індекси SP-GiST найбільш корисні для даних, які мають природний елемент кластеризації, а також не є однаково збалансованим деревом, наприклад, ГІС, мультимедіа, телефонна маршрутизація та IP-маршрутизація.

Індекс BRIN (Block Range Index)

BRIN означає індекси діапазону блоків (**b**lock **r**ange **in**dexes). BRIN набагато менший і дешевший у обслуговуванні порівняно з індексом B-дерева. Індекси BRIN розроблені для дуже великих таблиць, де індексувати кожен рядок є недоцільним.

Індекс BRIN ділить таблицю на діапазони сторінок і зберігає підсумкову інформацію про кожен діапазон, що робить їх ефективними для запитів діапазонів у великих наборах даних, використовуючи мінімальний простір. BRIN дозволяє використовувати індекс у дуже великій таблиці, що раніше було б непрактичним за допомогою B-дерева без горизонтального поділу. BRIN часто використовується для стовпця, який має лінійний порядок сортування, наприклад, створений стовпець дати таблиці замовлень на продаж.

Створення індексу CREATE INDEX

Для створення індексів в PostgreSQL використовується оператор CREATE INDEX, синтаксис якого виглядає наступним чином:

```
CREATE INDEX [IF NOT EXISTS] index_name
ON table_name (column1, column2, ...);
```

де:

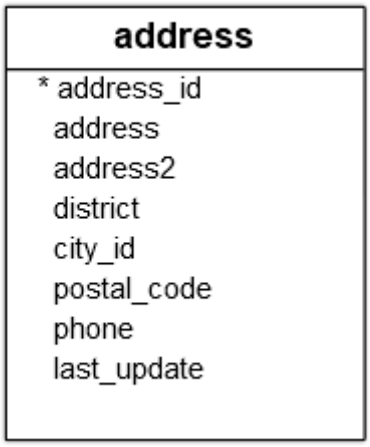
- після пропозиції CREATE INDEX вказується назва індексу;
- опція IF NOT EXISTS використовується, щоб запобігти помилці, якщо індекс уже існує;
- *table_name* – назва таблиці, до якої належить індекс;
- *column1*, *column2*, ... – один або більше індексованих стовпців усередині дужок () після назви таблиці.

За замовчуванням PostgreSQL дозволяє вибір даних із таблиці під час створення індексу, але блокує операції вставки, оновлення та видалення. За замовчуванням оператор CREATE INDEX створює індекс B-дерева, що підходить для більшості випадків, але можна створити й інші типи індексів (із пропозицією USING та 1 із: hash, gist, spgist, gin, brin).

Після створення індексу PostgreSQL має підтримувати його синхронізацію з таблицею. Наприклад, під час вставки, оновлення чи видалення даних із таблиці `contacts` PostgreSQL оновлює індекс, щоб відповідним чином відобразити зміни. У результаті індекс додає витрати на запис до операцій маніпулювання даними (вставлення, оновлення, видалення).

Приклад створення індексу і його вплив на виконання запиту в PostgreSQL

Для демонстрації наступних прикладів створення індексів ми використаємо демонстраційну таблицю `address`, що містить стовпці: первинний ключ, перша і друга адреса, район, ідентифікатор міста, поштовий індекс, телефон, а також дату останнього оновлення інформації:



Виконаємо наступний запит, щоб знайти адресу, номер телефону якої є `223664661973`:

```
SELECT
  address_id,
  address,
  district,
```

```
  phone
FROM
  address
WHERE
  phone = '223664661973';
```

Результат:

```
address_id |      address       | district  |    phone
------------+--------------------+-----------+--------------
        85 | 320 Baiyin Parkway | Mahajanga | 223664661973
(1 row)
```

Щоб знайти рядок, значення якого в стовпці телефону дорівнює `223664661973`, PostgreSQL має просканувати всю таблицю `address`. Покажемо план виконання запиту за допомогою наступного оператора EXPLAIN (детальніше про цей оператор буде розказано нижче):

```
EXPLAIN SELECT
  address_id,
  address,
  district,
  phone
FROM
  address
WHERE
  phone = '223664661973';
```

В результаті отримуємо наступний план виконання запиту:

```
QUERY PLAN
-------------------------------------------------------------
 Seq Scan on address  (cost=0.00..15.54 rows=1 width=45)
   Filter: ((phone)::text = '223664661973'::text)
(2 rows)
```

Цей план вказує на те, що оптимізатор запитів має виконати послідовне сканування (Seq Scan) таблиці `address`, що може бути трудомісткою задачею для великих таблиць. Також у дужках вказується вартість виконання операції (на виведення першого рядка та всіх рядків через …), кількість рядків, що буде виведена (1) та їх середня ширина у байтах (45).

Створимо індекс для значень у стовпці `phone` таблиці `address` за допомогою оператора CREATE INDEX:

```
CREATE INDEX idx_address_phone
```

```
ON address (phone);
```

Коли виконується оператор CREATE INDEX, PostgreSQL сканує таблицю `address`, витягує дані зі стовпця `phone` і вставляє їх в індекс `idx_address_phone`. Цей процес називається створенням індексу. За замовчуванням PostgreSQL дозволяє читання з таблиці `address` та блокує операції запису під час створення індексу.

Покажемо індекси, які належать до таблиці `address` за допомогою системного представлення pg_indexes (цей вид представлення буде описано далі):

```
SELECT
  indexname,
  indexdef
FROM
  pg_indexes
WHERE
  tablename = 'address';
```

В результаті отримуємо інформацію про всі індекси таблиці `address`:

```
indexname     |                                  indexdef
------------------+-----------------------------------------------------------------------------
 address_pkey       | CREATE UNIQUE INDEX address_pkey ON public.address USING btree
(address_id)
 idx_fk_city_id    | CREATE INDEX idx_fk_city_id ON public.address USING btree (city_id)
 idx_address_phone | CREATE INDEX idx_address_phone ON public.address USING btree
(phone)
(3 rows)
```

Результат показує, що `idx_address_phone` було створено успішно. Два інших індекси `address_pkey` та `idx_fk_city_id` були створені неявно під час створення таблиці `address`. Точніше, індекс `address_pkey` було створено для стовпця первинного ключа `address_id`, а `idx_fk_city_id` — для стовпця `city_id` зовнішнього ключа.

Покажемо план виконання попереднього запиту після створення індексу, щоб побачити, що змінилось:

```
EXPLAIN SELECT
  address_id,
  address,
  district,
  phone
FROM
  address
WHERE
  phone = '223664661973';
```

Отримаємо наступний план виконання запиту:

```
QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using idx_address_phone on address  (cost=0.28..8.29 rows=1 width=45)
   Index Cond: ((phone)::text = '223664661973'::text)
(2 rows)
```

Бачимо, що PostgreSQL використовує індекс `idx_address_phone` для пошуку, тобто для доступу до таблиці `address` використовується індексне сканування (Index Scan) замість послідовного, що було показано у попередньому прикладі. Як можна побачити, вартість виконання операції читання внаслідок створення індексу також знизилась (із 15.54 до 8.29).

Оператор EXPLAIN

Оператор EXPLAIN повертає план виконання, який планувальник PostgreSQL генерує для даного запиту. EXPLAIN показує, як таблиці, задіяні в запиті, скануватимуться індексно або послідовним скануванням тощо, і якщо використовується кілька таблиць, який тип алгоритму з'єднання таблиць буде використано (хеш-з'єднання чи інші).

Найважливішою та корисною інформацією, яку повертає оператор EXPLAIN, є початкова вартість до повернення першого рядка та загальна вартість повернення повного набору результатів. Нижче показано синтаксис оператора EXPLAIN:

EXPLAIN [(*option* [, ...])] *sql_statement*;

де опція *option* може бути однією із наступних:

ANALYZE [boolean]
VERBOSE [boolean]
COSTS [boolean]
BUFFERS [boolean]
TIMING [boolean]
SUMMARY [boolean]
FORMAT { TEXT | XML | JSON | YAML }

Логічне значення *boolean* вказує, чи слід увімкнути чи вимкнути вибраний параметр. Можна використовувати TRUE, ON або 1, щоб увімкнути опцію, і FALSE, OFF або 0, щоб вимкнути її. Якщо опустити логічне значення, за замовчуванням буде ON.

ANALYZE

Опція ANALYZE спричиняє виконання спочатку SQL запиту, а потім видачу фактичної статистики під час виконання запиту у поверненій інформації, включаючи загальний час, витрачений у кожному вузлі плану, і кількість рядків, які він повертає. Параметр ANALYZE виконує запит SQL і відкидає вихідну інформацію, отже, якщо ви хочете проаналізувати будь-який запит оновлення даних, наприклад INSERT, UPDATE або DELETE, не впливаючи на дані, вам слід обернути EXPLAIN ANALYZE в транзакцію наступним чином (додавши ROLLBACK):

```
BEGIN;
  EXPLAIN ANALYZE sql_statement;
```

ROLLBACK;

VERBOSE

Параметр VERBOSE дозволяє показати додаткову інформацію щодо плану. За замовчуванням цей параметр має значення FALSE.

COSTS

Параметр COSTS містить приблизну вартість запуску (*cost=перше число*) та загальну вартість (*число після …*) кожного вузла плану, а також приблизну кількість рядків (*rows*) і приблизну ширину (*width*) кожного рядка (у байтах) у плані запиту. COSTS за умовчанням має значення TRUE.

BUFFERS

Цей параметр додає інформацію про використання буфера. BUFFERS можна використовувати, лише якщо ввімкнено ANALYZE. За замовчуванням параметр BUFFERS має значення FALSE.

TIMING

Цей параметр включає фактичний час запуску та час, витрачений на кожному вузлі у вихідних даних. Для параметра TIMING за замовчуванням встановлено TRUE, і його можна використовувати лише тоді, коли ввімкнено ANALYZE.

SUMMARY

Параметр SUMMARY додає підсумкову інформацію, таку як загальний час після плану запиту. Зауважте, що коли використовується параметр ANALYZE, підсумкова інформація включається за замовчуванням.

FORMAT

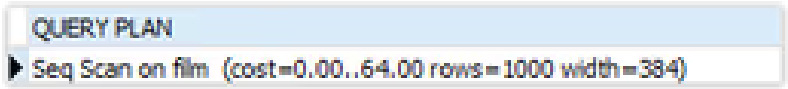
Укажіть вихідний формат плану запиту, наприклад TEXT, XML, JSON і YAML. За замовчуванням цей параметр має значення TEXT.

Приклади використання команди PostgreSQL EXPLAIN

Наступний оператор показує план для простого запиту, що повертає всі стовпці таблиці `film`:

```
EXPLAIN SELECT * FROM film;
```

План виконання запиту виглядає наступним чином, бачимо, що виконується послідовне сканування таблиці (Seq Scan), буде повернено 1000 рядків середньою шириною 384B, вартість запуску становить 0.00, повернення всіх рядків 64.00:



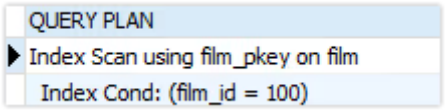
У наведеному нижче прикладі показано план для запиту, який повертає фільм за конкретним `film_id`:

```
EXPLAIN SELECT * FROM film WHERE film_id = 100;
```

Оскільки `film_id` проіндексовано (первинний ключ – унікальний індекс, про що буде нижче сказано), запит повернув інший план. У вихідних даних планувальник використав індексне сканування (Index Scan) використовуючи первинний ключ `film_pkey` замість послідовного сканування всієї таблиці `film`.

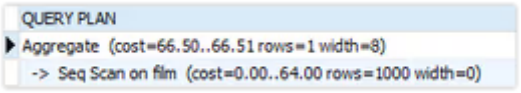
Щоб не показувати вартість, можна вимкнути цей параметр, вказавши FALSE в COSTS:

```
EXPLAIN (COSTS FALSE)
SELECT    * FROM    film
WHERE
   film_id = 100;
```



У наступному прикладі показано план для запиту, який використовує агрегатну функцію:

```
EXPLAIN SELECT COUNT(*) FROM film;
```



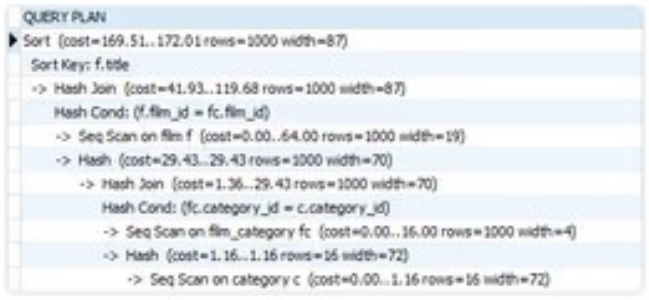
Зазвичай план виконання читається знизу вверх, бачимо, що спочатку виконується послідовне сканування таблиці `film` (Seq Scan), а потім виконання агрегатної функції (Aggregate).

Наступний приклад повертає план для оператора, який об'єднує кілька таблиць (`film, film_category, category`):

```
EXPLAIN
SELECT
   f.film_id,
   title,
   name category_name
FROM
   film f
   INNER JOIN film_category fc
      ON fc.film_id = f.film_id
   INNER JOIN category c
      ON c.category_id = fc.category_id
ORDER BY
   title;
```

Отримаємо наступний план виконання запиту, для з'єднання таблиць використовується алгоритми хеш-з'єднання (Hash Join) за умовами хешування (Hash Cond) спочатку c.category_id = fc.category_id, а потім fc.film_id = f.film_id, після цього виконується сортування ORDER BY результату (Sort) за ключовим параметром (f.title). Послідовно

знизу вгору можна побачити, як обчислюється і зрозстає вартість виконання кожного вузла плану (cost):



Щоб додати фактичну статистику часу виконання до виводу, потрібно виконати EXPLAIN за допомогою параметра ANALYZE:

```
EXPLAIN ANALYZE
   SELECT
      f.film_id,
      title,
      name category_name
   FROM
      film f
      INNER JOIN film_category fc
         ON fc.film_id = f.film_id
      INNER JOIN category c
         ON c.category_id = fc.category_id
   ORDER BY
      title;
```

Унікальні індекси UNIQUE в PostgreSQL

Унікальний індекс PostgreSQL забезпечує унікальність значень в одному або кількох стовпцях. Щоб створити унікальний індекс, використовуємо оператор CREATE UNIQUE INDEX:

```
CREATE UNIQUE INDEX index_name
ON table_name (column [, ...])
[ NULLS [ NOT ] DISTINCT ];
```

де:

- *index_name* – назва індексу в операторі CREATE UNIQUE INDEX;
- *table_name* (*column* [, ...]) – назва індексованої таблиці разом зі списком індексованих стовпців у реченні ON;
- параметр NULLS NOT DISTINCT розглядає порожні значення як рівні, тоді як NULLS DISTINCT розглядає порожні як різні значення. За замовчуванням оператор використовує NULLS DISTINCT, тобто індексований стовпець може містити кілька порожніх значень.

PostgreSQL пропонує кілька типів індексів, але *лише тип індексу B-tree підтримує унікальні індекси*. Коли визначається унікальний індекс для стовпця, стовпець не може зберігати кілька рядків з однаковими значеннями.

Якщо визначається унікальний індекс для двох або більше стовпців, об'єднані значення в цих стовпцях не можна дублювати в кількох рядках.

Коли задається обмеження *первинний ключ* (PRIMARY KEY) або *унікальне обмеження* (UNIQUE) для таблиці, PostgreSQL *автоматично створює відповідний унікальний індекс*.

Приклади створення унікальних індексів PostgreSQL UNIQUE index

Давайте розглянемо кілька прикладів використання унікальних індексів PostgreSQL.

1. Унікальні індекси для стовпця первинного ключа та стовпця з унікальним обмеженням.

Спочатку створимо наступну таблицю `employees`, що містить інформацію про ім'я (`first_name`), прізвище (`last_name`) та електронну пошту працівника (`email`), а також первинний ключ:

```
CREATE TABLE employees (
   employee_id SERIAL PRIMARY KEY,
   first_name VARCHAR(255) NOT NULL,
   last_name VARCHAR(255) NOT NULL,
   email VARCHAR(255) UNIQUE
);
```

У цьому запиті, `emploee_id` є стовпцем первинного ключа, а стовпець `email` має унікальне обмеження, тому PostgreSQL створює два UNIQUE індекси, по одному для кожного стовпця.

Покажемо всі індекси таблиці `employees`, використовуючи представлення *pg_indexes*:

```
SELECT
   tablename,
   indexname,
   indexdef
FROM
   pg_indexes
WHERE
   tablename = 'employees';
```

Отримуємо інформацію про всі індекси таблиці `employees`:

```
tablename |     indexname     |                          indexdef
----------+-------------------+-------------------------------------------------------------------------------
-
 employees | employees_pkey    | CREATE UNIQUE INDEX employees_pkey ON public.employees USING btree (employee_id)
```

```
 employees | employees_email_key | CREATE UNIQUE INDEX employees_email_key ON public.employees USING btree (email)
(2 rows)
```

2. Використання унікального індексу PostgreSQL для одного стовпця.

Спочатку додамо стовпець з назвою `mobile_phone` до таблиці `employees`:

```
ALTER TABLE employees
ADD mobile_phone VARCHAR(20);
```

Щоб переконатися, що номери мобільних телефонів є різними для всіх співробітників, можна визначити унікальний індекс для стовпця `mobile_phone` за допомогою оператора CREATE INDEX.

Створимо унікальний індекс у стовпці `mobile_phone` таблиці `employees`:

```
CREATE UNIQUE INDEX idx_employees_mobile_phone
ON employees (mobile_phone);
```

Вставимо новий рядок у таблицю `employees`:

```
INSERT INTO employees(first_name, last_name, email, mobile_phone)
VALUES ('John','Doe','[[email protected]](../cdn-cgi/l/email-protection.html)', '(408)-555-1234');
```

Спробуємо вставити інший рядок із тим же номером телефону:

```
INSERT INTO employees(first_name, last_name, email, mobile_phone)
VALUES ('Jane','Doe','[[email protected]](../cdn-cgi/l/email-protection.html)', '(408)-555-1234');
```

PostgreSQL видає наступну помилку через повторюваний номер мобільного телефону, отже наш унікальний індекс `idx_employees_mobile_phone` працює:

```
ERROR:  duplicate key value violates unique constraint "idx_employees_mobile_phone"
DETAIL:  Key (mobile_phone)=((408)-555-1234) already exists.
```

Резюме

– використовуйте унікальний індекс PostgreSQL, щоб забезпечити унікальність значень у стовпці або наборі стовпців;

– PostgreSQL автоматично створює унікальний індекс для стовпця первинного ключа або стовпця з унікальним обмеженням.

Частковий індекс в PostgreSQL

Коли створюється індекс для стовпця таблиці, PostgreSQL використовує всі значення з цього стовпця для створення індексу. Іноді можна включити в індекс лише деякі значення з індексованого стовпця. Для цього використовується частковий індекс (*Partial Index*).

Частковий індекс – це індекс, побудований на підмножині даних індексованих стовпців. Щоб визначити підмножину даних, використовуєтmcz предикат, який є умовним

виразом часткового індексу. PostgreSQL створить індекс для рядків, які задовольняють предикат.

Частковий індекс може підвищити продуктивність запитів, одночасно зменшивши розмір індексу. Це також може покращити оновлення таблиць, оскільки PostgreSQL не потрібно підтримувати індекс у всіх випадках.

Щоб створити частковий індекс, використовується оператор CREATE INDEX із пропозицією WHERE:

CREATE [IF NOT EXISTS] INDEX *index_name*
ON *table_name* (*column1, column2, ...*)
WHERE *predicate*;

де:

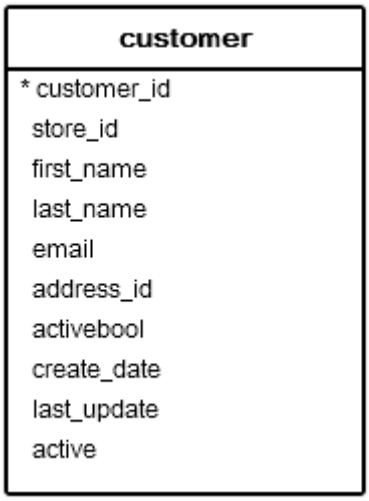
– *index_name* – назву індексу в операторі CREATE INDEX. Використовуйте IF NOT EXISTS, щоб запобігти помилці створення індексу, який уже існує;

– *table_name* (*column1, column2, ...*) – назва таблиці разом з індексованими стовпцями в реченні ON;

– *predicate* – предикат у реченні WHERE, щоб визначити умову для рядків, які будуть включені в індекс.

Приклад використання часткового індексу в PostgreSQL

Візьмемо для прикладу таблицю `customer`, що містить дані про клієнтів, а саме первинний ключ, айді сховища, ім'я, прізвище, поштову скриньку, адресу, дату створення, останнього оновлення даних, а також чи є клієнт активним:



Стовпець `active` має 2 значення:

– 0: неактивний;
– 1: активний.

Запитуючи дані з бази даних, часто працюють з неактивними клієнтами, а не з активними. Наприклад, ви можете зв'язатися з неактивними клієнтами, щоб змусити їх замовити більше фільмів.

Щоб пришвидшити запит, який отримує неактивного клієнта, можна створити частковий індекс.

Спочатку створимо частковий індекс для стовпця `active` таблиці клієнтів `customer`, задавши умовою предикату рівність 0:

```
CREATE INDEX customer_active
ON customer (active)
WHERE active = 0;
```

Покажемо план запиту, який отримує неактивних клієнтів:

```
EXPLAIN
SELECT  * FROM  customer
WHERE
 active = 0;
```

В результаті бачимо, що відубвається індексне сканування таблиці `customer`:

```
QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using customer_active on customer  (cost=0.14..16.12 rows=15 width=70)
```

Частковий індекс `customer_active` покращує продуктивність запиту, включаючи лише ті рядки, які часто шукаються.

Резюме

Використовуйте частковий індекс PostgreSQL, щоб створити індекс, який включає підмножину рядків у таблиці, задану умовою.

Багатоколонкові індекси в PostgreSQL

Коли ви створюєте індекс на двох або більше стовпцях у таблиці, цей тип індексу називається багатостовпцевим (багатоколонковим) індексом. Багатостовпцевий індекс часто називають складеним індексом, комбінованим індексом або конкатенованим індексом.

Багатостовпцевий індекс може містити максимум 32 стовпці (обмеження можна змінити, змінивши файл конфігурації pg_config_manual.h).

Крім того, багатоколонкові індекси підтримують лише типи індексів B-tree, GIST, GIN і BRIN.

Нижче показано синтаксис для створення багатоколонкового індексу:

CREATE INDEX [IF NOT EXISTS] *index_name*
ON *table_name* (*column1, column2*, ...);

де:

– *index_name* – назва індексу в реченні CREATE INDEX. Використовуйте опцію IF NOT EXISTS, щоб запобігти помилці під час створення індексу, ім’я якого вже існує;

– після ON вводиться назва таблиці *table_name* разом із стовпцями індексу в дужках (*column1, column2*, ...).

Визначаючи багатостовпцевий індекс, маємо розмістити стовпці, які часто використовуються в реченні WHERE, на початку списку стовпців, а потім стовпці, які рідше використовуються в реченні WHERE.

У наведеному вище синтаксисі оптимізатор запитів розглядатиме використання багатоколонкового індексу в наступних випадках:

WHERE column1 = v1 AND column2 = v2 AND column3 = v3;

Або коли звертаємось до частини стовпців в порядку згадування у дужках ():

WHERE column1 = v1 AND column2 = v2;

Або коли звертаємось до першого стовпця складеного індексу:

WHERE column1 = v1;

Однак він *не розглядатиме* використання багатоколонкового індексу в наведених нижче випадках, коли звертаємось до стовпців індексу не по порядку їх згадування в індексі:

WHERE column3 = v3;

Або

WHERE column2 = v2 AND column3 = v3;

Зауважимо, що також можна використовувати речення WHERE, щоб визначити частково багатостовпцевий індекс.

Приклад створення багатоколонкового індексу в PostgreSQL

Спочатку створимо нову таблицю `people`, яка складається з 3 стовпців: `id`, ім'я `first_name` та прізвище `last_name`:

```
CREATE TABLE people (
   id INT GENERATED BY DEFAULT AS IDENTITY,
   first_name VARCHAR(50) NOT NULL,
   last_name VARCHAR(50) NOT NULL
);
```

Покажемо план запиту, який знаходить особу на прізвище `Adams`:

```
EXPLAIN SELECT
  id,
  first_name,
  last_name
FROM
  people
WHERE
  last_name = 'Adams';
```

План виконання запиту:

```
QUERY PLAN
-------------------------------------------------------
 Seq Scan on people  (cost=0.00..83.88 rows=9 width=240)
   Filter: ((last_name)::text = 'Adams'::text)
(2 rows)
```

Результат показує, що PostgreSQL виконує послідовне сканування (Seq Scan) таблиці `people`, щоб знайти відповідні рядки, оскільки для стовпця `last_name` не визначено індекс.

Створимо індекс, який включає стовпці `last_name` і `first_name`. Припускаючи, що пошук людей за прізвищем є більш поширеним, ніж за іменем, ми визначаємо індекс із таким порядком стовпців:

```
CREATE INDEX idx_people_names
ON people (last_name, first_name);
```

Покажемо план запиту, який шукає особу на прізвище `Adams`:

```
EXPLAIN SELECT
  id,
  first_name,
  last_name
FROM
  people
WHERE
  last_name = 'Adams';
```

План виконання запиту:

```
QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on people  (cost=4.42..44.07 rows=18 width=17)
   Recheck Cond: ((last_name)::text = 'Adams'::text)
   ->  Bitmap Index Scan on idx_people_names  (cost=0.00..4.42 rows=18 width=0)
         Index Cond: ((last_name)::text = 'Adams'::text)
(4 rows)
```

Результат показує, що оптимізатор запитів використовує індекс `idx_people_names` для пошуку значень в таблиці, а також алгоритм бітової карти (bitmap), не зважаючи на те, що у запиті ми використали лише прізвище (перший стовпець багатоколонкового індексу `idx_people_names`).

Покажемо план запиту, який шукає людину, чиє прізвище `Adams`, а ім'я `Lou`.

```
EXPLAIN SELECT
  id,
  first_name,
  last_name
FROM
  people
```

```
WHERE
  last_name = 'Adams'
  AND first_name = 'Lou';
```

План виконання запиту:

```
QUERY PLAN
-----------------------------------------------------------------------------------------
 Index Scan using idx_people_names on people  (cost=0.29..8.30 rows=1 width=17)
   Index Cond: (((last_name)::text = 'Adams'::text) AND ((first_name)::text = 'Lou'::text))
(2 rows)
```

Результат показує, що оптимізатор запитів виконує індексне сканування таблиці `people`, використовуючи індекс `idx_people_names`, оскільки обидва стовпці в фразі WHERE (`first_name` і `last_name`) входять до індексу.

Покажемо план запиту, який шукає людину чиє ім'я `Lou`:

```
EXPLAIN SELECT
  id,
  first_name,
  last_name
FROM
  people
WHERE
  first_name = 'Lou';
```

План виконання запиту:

```
QUERY PLAN
-------------------------------------------------------------
 Seq Scan on people  (cost=0.00..186.00 rows=32 width=17)
   Filter: ((first_name)::text = 'Lou'::text)
(2 rows)
```

Результат показує, що PostgreSQL виконує послідовне сканування (Seq Scan) таблиці `people` замість використання індексу, хоча стовпець `first_name` є частиною індексу, але не перший по порядку згадування стовпців в індексі.

Резюме

Використовуйте багатостовпцевий індекс PostgreSQL, щоб визначити індекс, що включає два або більше стовпців із таблиці.

Розмістіть стовпці, які часто використовуються в реченні WHERE, на початку списку стовпців багатоколнкового індексу.

Оператор PostgreSQL REINDEX

На практиці індекс може бути пошкоджений і більше не містити дійсних даних через апаратні збої або помилки програмного забезпечення.

Крім того, коли створюється індекс без параметра CONCURRENTLY, індекс може стати недійсним, якщо побудова індексу не вдасться.

У таких випадках можна перебудувати індекс. Щоб перебудувати індекс, використовується оператор REINDEX наступним чином:

REINDEX [(*option*, ...)]
{ INDEX | TABLE | SCHEMA | DATABASE | SYSTEM }
name;

де:

– опція *option* може мати одне або декілька значень:

– VERBOSE [boolean] – показувати прогрес по мірі перебудови кожного індексу;

– TABLESPACE *new_tablespace* – новий табличний простір, на якому будуть перебудовані індекси;

– CONCURRENTLY – перебудувати індекс без жодних блокувань. Якщо не використовувати, REINDEX блокуватиме записи, але не читання в таблиці, доки не буде завершено.

– INDEX | TABLE | SCHEMA | DATABASE | SYSTEM – вказуємо, чи хочемо перебудувати вказаний індекс / всі індекси в таблиці / схемі / базі даних / системному каталозі поточної БД.

Щоб перебудувати один індекс, вказуємо ім’я індексу після оператору REINDEX INDEX:

REINDEX INDEX *index_name*;

Щоб перебудувати всі індекси таблиці, використовуємо ключове слово TABLE і вказуємо назву таблиці:

REINDEX TABLE *table_name*;

Аналогічно і для інших варіантів SCHEMA | DATABASE | SYSTEM.

REINDEX порівняно з DROP INDEX та CREATE INDEX

Оператор REINDEX перебудовує індекс з нуля, що має подібний ефект до видалення DROP INDEX та повторного створення індексу CREATE INDEX. Однак механізми блокування у них різні.

Оператор REINDEX:

– заблокувати запис, але не читання з таблиці, до якої належить індекс;

– взяти ексклюзивне блокування індексу, який обробляється, що блокує читання, яке намагається використати індекс.

Оператори DROP INDEX та CREATE INDEX:

– по-перше, DROP INDEX блокує як запис, так і читання таблиці, до якої належить індекс, шляхом отримання ексклюзивного блокування таблиці;

– потім наступний оператор CREATE INDEX блокує запис, але не читання з батьківської таблиці індексу. Однак читання може бути дорогим під час створення індексу.

Оператор DROP INDEX

Іноді може знадобитися видалити існуючий індекс із системи бази даних. Для цього використовується оператор DROP INDEX з наступним синтаксисом:

DROP INDEX [CONCURRENTLY] [IF EXISTS] *index_name*
[CASCADE | RESTRICT];

де:

– *index_name* – назва індексу, який потрібно видалити після пропозиції DROP INDEX;

– *IF EXISTS* – спроба видалити неіснуючий індекс призведе до помилки. Щоб уникнути цього, можна використовувати опцію IF EXISTS. Якщо спробувати видалити неіснуючий індекс із використанням IF EXISTS, PostgreSQL замість помилки видасть повідомлення;

– *CASCADE* – якщо в індексі є залежні об'єкти, використовується параметр CASCADE, щоб автоматично видалити ці об'єкти та всі об'єкти, які залежать від цих об'єктів;

– *RESTRICT* – параметр RESTRICT наказує PostgreSQL відмовитися від видалення індексу, якщо від нього залежать будь-які об'єкти. DROP INDEX за замовчуванням використовує RESTRICT;

– *CONCURRENTLY* – коли виконується оператор DROP INDEX, PostgreSQL отримує ексклюзивне блокування таблиці та блокує інші доступи до завершення видалення індексу. Щоб змусити команду чекати завершення конфліктної транзакції перед видаленням індексу, ви можете скористатися параметром CONCURRENTLY. Це дасть змогу завершити всі команди зміни даних, а потім видалити індекс.

DROP INDEX CONCURRENTLY має деякі обмеження:

– по-перше, параметр CASCADE не підтримується;

– по-друге, виконання в блоці транзакцій також не підтримується.

Зауважте, що можна видалити кілька індексів одночасно, розділяючи індекси комами (,):

DROP INDEX *index_name*, *index_name2*, ...;

Список індексів PostgreSQL

PostgreSQL не надає такої команди, як SHOW INDEXES, для перерахування інформації про індекси таблиці або бази даних.

Однак PostgreSQL надає доступ до системного представлення *pg_indexes*, щоб можна було запитувати інформацію про індекси.

Якщо використовується psql для взаємодії з базою даних PostgreSQL, можна використовувати команду \d для перегляду інформації про індекси в таблиці. Розглянемо обидва випадки нижче.

Список індексів PostgreSQL використовуючи представлення pg_indexes

Представлення *pg_indexes* дозволяє отримати доступ до корисної інформації про кожен індекс у базі даних PostgreSQL.

Представлення *pg_indexes* складається з 5 стовпців:

– *schemaname*: зберігає назву схеми, яка містить таблиці та індекси;

– *tablename*: вказує назву таблиці, до якої належить індекс;

– *indexname:* представляє назву індексу;

– *tablespace*: визначає назву табличного простору, який містить індекси;

– *indexdef*: містить команду визначення індексу у формі оператора CREATE INDEX.

Наступний запит перераховує всі індекси схеми `public` в поточній базі даних:

```
SELECT
   tablename,
   indexname,
   indexdef
FROM
   pg_indexes
WHERE
   schemaname = 'public'
ORDER BY
   tablename,
   indexname;
```

Отримаємо наступне представлення:

```
tablename     |                 indexname                  |                          indexdef
--------------------+----------------------------------------------------+----------------------------------------
----------------------------------------------------------------------------------------------------
 accounts            | accounts_email_key                          | CREATE UNIQUE INDEX accounts_email_key ON public.accounts USING btree (email)
 accounts          | accounts_pkey                         | CREATE UNIQUE INDEX accounts_pkey ON public.accounts USING btree (user_id)
 accounts            | accounts_username_key                          | CREATE UNIQUE INDEX accounts_username_key ON public.accounts USING btree (username)
 actor             | actor_pkey                          | CREATE UNIQUE INDEX actor_pkey ON public.actor USING btree (actor_id)
 actor             | idx_actor_first_name                        | CREATE INDEX idx_actor_first_name ON public.actor USING btree (first_name)
 actor             | idx_actor_last_name                         | CREATE INDEX idx_actor_last_name ON public.actor USING btree (last_name)
...
```

Щоб показати всі індекси конкретної таблиці, можна скористатись наступним запитом:

```
SELECT
  indexname,
  indexdef
FROM
```

```
 pg_indexes
WHERE
 tablename = 'table_name';
```

Наприклад, щоб перерахувати всі індекси для таблиці `customer`, використовуйте наступний вираз:

```
SELECT
   indexname,
   indexdef
FROM
   pg_indexes
WHERE
   tablename = 'customer';
```

Отримаємо інформацію про всі індекси таблиці `customer`:

```
indexname    |                                  indexdef
-----------------+-----------------------------------------------------------------------------
 customer_pkey    | CREATE UNIQUE INDEX customer_pkey ON public.customer USING btree (customer_id)
 idx_fk_address_id | CREATE INDEX idx_fk_address_id ON public.customer USING btree (address_id)
 idx_fk_store_id  | CREATE INDEX idx_fk_store_id ON public.customer USING btree (store_id)
 idx_last_name        | CREATE INDEX idx_last_name ON public.customer USING btree (last_name)
(4 rows)
```

Список індексів PostgreSQL використовуючи команду psql

Якщо використовується psql для підключення до бази даних PostgreSQL і ми бажаємо отримати список усіх індексів таблиці, можна використати команду \d psql наступним чином:

```
\d table_name
```

Команда поверне всю інформацію про таблицю, включаючи структуру таблиці, індекси, обмеження та тригери.

Наприклад, наступний оператор повертає детальну інформацію про таблицю `customer`:

```
\d customer
```

Результат показує індекси таблиці в розділі «Індекси»:

Table "public.customer"

```
  Column    |            Type             | Collation | Nullable |                  Default
------------+-----------------------------+-----------+----------+-------------------------------------------
 customer_id | integer                     |           | not null | nextval('customer_customer_id_seq'::regclass)
 store_id    | smallint                    |           | not null |
 first_name  | character varying(45)       |           | not null |
 last_name   | character varying(45)       |           | not null |
 email       | character varying(50)       |           |          |
 address_id  | smallint                    |           | not null |
 activebool  | boolean                     |           | not null | true
 create_date | date                        |           | not null | 'now'::text::date
 last_update | timestamp without time zone |           |          | now()
 active      | integer                     |           |          |
```

Indexes:

"customer_pkey" PRIMARY KEY, btree (customer_id)

"idx_fk_address_id" btree (address_id)

"idx_fk_store_id" btree (store_id)

"idx_last_name" btree (last_name)

Foreign-key constraints:

"customer_address_id_fkey" FOREIGN KEY (address_id) REFERENCES address(address_id) ON UPDATE CASCADE ON DELETE RESTRICT

Referenced by:

TABLE "payment" CONSTRAINT "payment_customer_id_fkey" FOREIGN KEY (customer_id) REFERENCES customer(customer_id) ON UPDATE CASCADE ON DELETE RESTRICT

TABLE "rental" CONSTRAINT "rental_customer_id_fkey" FOREIGN KEY (customer_id) REFERENCES customer(customer_id) ON UPDATE CASCADE ON DELETE RESTRICT

Triggers:

last_updated BEFORE UPDATE ON customer FOR EACH ROW EXECUTE FUNCTION last_updated()

Висновки

Індексування та правильний вибір типів даних відіграють ключову роль у забезпеченні високої продуктивності та ефективності роботи з базою даних PostgreSQL. Розробникам та адміністраторам баз даних слід ретельно проектувати індекси, оптимізувати запити та регулярно підтримувати індекси в актуальному стані для забезпечення найкращого функціонування системи.

Структура звіту до лабораторної роботи

Для кожного з запитів представити:

1) словесна постановка задачі, що вирішується;

2) SQL-код рішення;

3) скриншот отриманого результату.

Завдання до лабораторної роботи

1. Створіть запит, який вибирає із таблиці за варіантом предментої області для лабораторних робіт значення за числовим діапазоном (від-до) одного із стовпців таблиці (не з первинним чи зовнішнім ключем). Виконайте команду пояснення плану виконання запиту. Опишіть план виконання запиту.

2. Створіть до стовпця із завдання 1 індекс. Виконайте команду пояснення плану виконання того ж самого запиту, як і в завданні 1. Опишіть зміни, що відбулись.

3. Виконайте запит, що вибирає значення із різних 2 стовпців іншої таблиці (наприклад, за цілочисельним і символьним значенням чи іншим). Виконайте команду пояснення плану виконання запиту. Опишіть план виконання запиту.

4. Створіть до стовпців із завдання 3 багатостовпчиковий індекс. Виконайте команду пояснення плану виконання того ж самого запиту, як і в завданні 3. Опишіть зміни, що відбулись.

5. Виконайте запит до таблиці із завдання 3, використовуючи 1 індексований стовпець із завдання 4 та 1 неіндексований, поясніть зміни у плані виконання запиту.

6. Створіть індекс, що індексуватиме *частину* значень стовпчика третьої таблиці (не повторюючи таблиці із попередніх завдань). Виконайте пояснення плану виконання запиту, що використовує цей індекс.

7. Додайте до таблиці із першого завдання 5 рядків, виконайте команду реіндексування.

8. Перегляньте список створених індексів до таблиць із завдань 2, 4 і 6 за допомогою представлення pg_indexes або `\d`.

Звіт до лабораторної роботи 12 можна здати онлайн на сайті ДО edu.op.edu.ua до початку вашого заняття.