Indeksy i plany wykonania w PostgreSQL

Poziom: podstawy **Dotyczy: PostgreSQL 9.6–17** **Czas: ~25 min**

PostgreSQL ma sześć typów indeksu, nie jeden — każdy rozwiązuje inny kształt zapytania, i wybór złego kończy się indeksem, który istnieje, kosztuje miejsce na dysku, a optymalizator i tak go nie użyje.

Skróty użyte na tej stronie (1)

Skrót	Co znaczy
WAL	Write-Ahead Log: dziennik PostgreSQL, zapis zmian przed zapisem danych

Fiszka

ZMIENIA SYSTEM

Utworzenie indeksu bez blokowania zapisów: CREATE INDEX CONCURRENTLY

```
CREATE INDEX CONCURRENTLY NazwaIndeksu ON NazwaTabeli (NazwaKolumny);
```

> **Pojęcie: CONCURRENTLY.** Zwykłe `CREATE INDEX` trzyma blokadę blokującą zapisy na tabeli przez cały czas budowy indeksu — na dużej tabeli produkcyjnej to może być niedopuszczalnie długie okno. Wariant `CONCURRENTLY` buduje indeks w kilku przebiegach, dopuszczając równoległe zapisy między nimi, kosztem dłuższego czasu całkowitej budowy i braku możliwości uruchomienia wewnątrz jawnej transakcji.

Przykład:

```
CREATE INDEX
```

Uważaj: jeśli budowa `CONCURRENTLY` zostanie przerwana (awaria, `Ctrl+C`, restart instancji), w katalogu zostaje niekompletny indeks oznaczony jako `INVALID` — taki indeks nie jest używany przez optymalizator i trzeba go usunąć ręcznie (`DROP INDEX`) przed ponowną próbą.

□□ ZMIENIA SYSTEM

Przebudowa indeksu bez blokowania zapisów: REINDEX CONCURRENTLY

```
REINDEX INDEX CONCURRENTLY NazwaIndeksu;
```

Przebudowujemy istniejący indeks od zera, bez blokady wyłącznej na tabeli — dostępne od PostgreSQL 12. Przydatne po dużej liczbie aktualizacji, kiedy indeks narósł fragmentacją opisaną na stronie Fragmentacja i przebudowa indeksów, a nie chcemy przerywać ruchu na tabeli na czas przebudowy.

Przykład:

```
REINDEX
```

Na wersjach starszych niż 12, gdzie `REINDEX CONCURRENTLY` jeszcze nie istnieje, równoważny efekt osiąga się ręcznie: budujemy nowy indeks `CONCURRENTLY` pod inną nazwą, a następnie usuwamy stary.

□□ TYLKO ODCZYT

Znalezienie nieużywanych indeksów: pg_stat_user_indexes

```
SELECT relname AS tabela, indexrelname AS indeks, idx_scan
FROM pg_stat_user_indexes
WHERE idx_scan = 0
ORDER BY pg_relation_size(indexrelid) DESC;
```

Sprawdzamy, które indeksy od uruchomienia statystyk (albo od ostatniego resetu licznika) nie zostały ani razu użyte przez żadne zapytanie — każdy z nich kosztuje miejsce na dysku i spowalnia każdy `INSERT`/`UPDATE` na tabeli, bo trzeba go aktualizować, mimo że nikomu nie służy do odczytu.

Przykład:

```
   tabela    |          indeks           | idx_scan
-------------+---------------------------+----------
 zamowienia  | idx_zamowienia_status_stary| 0
 klienci     | idx_klienci_telefon        | 0
(2 rows)
```

Uważaj: licznik `idx_scan` liczy od ostatniego restartu instancji albo od ręcznego `pg_stat_reset()` — indeks używany wyłącznie przez raport uruchamiany raz w miesiącu może chwilowo wyglądać jak nieużywany tuż po restarcie, mimo że w rzeczywistości jest potrzebny.

TYLKO ODCZYT

Uruchomienie planu z realnym zużyciem buforów: EXPLAIN (ANALYZE, BUFFERS)

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM NazwaTabeli WHERE NazwaKolumny =
WartoscWarunku;
```

Ogólny mechanizm `EXPLAIN` i sposób czytania planu opisuje strona Plan wykonania: jak go wywołać i jak go czytać. Opcja `BUFFERS`, dostępna razem z `ANALYZE`, dokłada do każdego węzła planu liczbę stron odczytanych z pamięci podręcznej (`shared hit`) i liczbę odczytanych faktycznie z dysku (`read`) — rozróżnienie, którego sam czas wykonania nie pokazuje, a które tłumaczy, dlaczego to samo zapytanie bywa błyskawiczne za drugim razem.

Przykład:

```
Index Scan using idx_zamowienia_klient_id on zamowienia  (cost=0.42..8.53 rows=12
width=48) (actual time=0.031..0.089 rows=14 loops=1)
  Index Cond: (klient_id = 42)
  Buffers: shared hit=16 read=3
Planning Time: 0.112 ms
Execution Time: 0.121 ms
```

`shared hit=16 read=3` oznacza, że z szesnastu potrzebnych stron trzynaście było już w pamięci podręcznej, a trzy trzeba było sprowadzić z dysku — przy powtórzeniu tego samego zapytania `read` zwykle spada do zera, bo strony zostają już w pamięci.



Planner wybiera Index Scan na podstawie EXPLAIN (ANALYZE, BUFFERS), a liczniki shared hit i read pokazują skąd przyszły strony (kliknięcie otwiera pełny rozmiar)

ZMIENIA SYSTEM

Zainstalowanie rozszerzenia do śledzenia zapytań: pg_stat_statements

```
CREATE EXTENSION pg_stat_statements;
```

> **Pojęcie: pg_stat_statements.** Rozszerzenie zbierające statystyki wykonania dla każdego unikalnego kształtu zapytania przechodzącego przez instancję — nie dla pojedynczego wywołania, tylko dla znormalizowanej postaci zapytania (z parametrami zastąpionymi placeholderami), agregując liczbę wywołań, łączny i średni czas oraz zużycie buforów. Wymaga dopisania `pg_stat_statements` do parametru `shared_preload_libraries` w `postgresql.conf` i restartu instancji, zanim samo `CREATE EXTENSION` zadziała.

Przykład:

```
CREATE EXTENSION
```

TYLKO ODCZYT

Odnalezienie najkosztowniejszych zapytań: pg_stat_statements

```
SELECT query, calls, round(total_exec_time::numeric, 1) AS laczny_czas_ms,
       round(mean_exec_time::numeric, 2) AS sredni_czas_ms
FROM pg_stat_statements
ORDER BY total_exec_time DESC
LIMIT 5;
```

Sprawdzamy, które zapytania łącznie zjadają najwięcej czasu silnika — nie musi to być najwolniejsze pojedyncze zapytanie, tylko takie, które wykonywane bardzo często sumuje się

do największego obciążenia. To pierwsze zapytanie, od którego zaczyna się poszukiwanie kandydata do optymalizacji, zamiast zgadywania na podstawie wrażenia „coś jest wolne".

Przykład:

```
                          query                           | calls | laczny_czas_ms |
sredni_czas_ms
----------------------------------------------------------+-------+----------------
+-----------------
 SELECT * FROM zamowienia WHERE klient_id = $1              | 48213 |        192852.4
|            4.00
 UPDATE zamowienia SET status = $1 WHERE id = $2            |  9021 |         81204.1
|            9.00
(2 rows)
```

Uważaj: kolumny `total_exec_time` i `mean_exec_time` noszą te nazwy od PostgreSQL 13 — na wersjach 9.6–12 to samo rozszerzenie zwraca te dane pod nazwami `total_time` i `mean_time`, bez przyrostka `_exec`.

Z życia wzięte

Z ŻYCIA WZIĘTE

Raport Marka z magazynu, wcześniej błyskawiczny, zaczyna trwać kilkanaście sekund po dużym imporcie danych do bazy cennik na pg01. `EXPLAIN (ANALYZE, BUFFERS)` pokazuje sekwencyjne skanowanie całej tabeli zamiast użycia istniejącego indeksu, bo statystyki optymalizatora są nieaktualne. Ręczne `ANALYZE` na tabeli przywraca poprzedni, szybki plan zapytania.

Typy indeksów w PostgreSQL

Ogólne pojęcie indeksu, powód jego istnienia i mechanikę B-drzewa opisują strony Czym jest indeks — i dlaczego to nie jest „przyspieszacz tabeli" i B-drzewo od środka. PostgreSQL implementuje kilka odrębnych metod dostępu (`access method`), z których każda jest osobną strukturą danych na dysku, dobraną pod inny rodzaj warunku w zapytaniu.

Typ	Do czego służy

`btree`	domyślny, ogólnego przeznaczenia — równość, zakresy, sortowanie; właściwy wybór, dopóki nie mamy konkretnego powodu, żeby sięgnąć po inny
`hash`	wyłącznie równość (`=`), szybszy niż btree dla tego jednego przypadku, ale bez wsparcia dla zakresów i sortowania
`GiST`	struktury nieliniowe — dane geometryczne, zakresy (`range types`), wyszukiwanie najbliższego sąsiada
`GIN`	wartości złożone, wielowartościowe w jednym wierszu — tablice, `jsonb`, pełnotekstowe wyszukiwanie (`tsvector`)
`BRIN`	bardzo duże tabele, w których wartość kolumny koreluje z fizycznym porządkiem wierszy na dysku, np. kolumna z datą w tabeli dopisywanej chronologicznie
`SP-GiST`	dane o nierównomiernym rozkładzie przestrzennym — drzewa czwórkowe, k-d-drzewa, dane hierarchiczne

Uważaj: indeks `hash` przed PostgreSQL 10 nie był bezpiecznie replikowany (operacje na nim nie trafiały do WAL) — od wersji 10 ten problem nie istnieje, ale w starszych środowiskach bywał to powód, dla którego administratorzy unikali go całkowicie na rzecz zwykłego `btree` nawet przy czystej równości.

`BRIN` (Block Range INdex) jest szczególnym przypadkiem — zamiast wskaźnika na każdy pojedynczy wiersz trzyma minimum i maksimum wartości dla całego zakresu stron dyskowych, przez co zajmuje ułamek miejsca `btree` na tej samej tabeli, kosztem mniejszej precyzji. Sprawdza się tylko wtedy, gdy dane są fizycznie uporządkowane zgodnie z indeksowaną kolumną — na tabeli, do której wiersze trafiają losowo względem tej kolumny, `BRIN` nie daje żadnej korzyści.

Częste błędy

Zakładanie GIN albo GiST tam, gdzie wystarczy btree. Indeksy GIN i GiST mają wyższy koszt aktualizacji niż btree — sięgamy po nie dopiero wtedy, gdy typ danych albo rodzaj warunku (tablica, `jsonb`, pełny tekst, geometria) faktycznie tego wymaga, nie z góry „na wszelki wypadek".

Trzymanie nieużywanych indeksów znalezionych w pg_stat_user_indexes bez dalszej weryfikacji. Zanim usunie się indeks o `idx_scan = 0`, warto sprawdzić harmonogram rzadko uruchamianych raportów i procesów batchowych — indeks nieużywany od restartu instancji sprzed tygodnia nie musi być nieużywany w cyklu miesięcznym.

Powiązane: Czym jest indeks — i dlaczego to nie jest „przyspieszacz tabeli" · B-drzewo od środka · Plan wykonania: jak go wywołać i jak go czytać · Fragmentacja i przebudowa indeksów · Statystyki i optymalizator kosztowy

Źródła: dokumentacja PostgreSQL, rozdział „Indexes" i „The Statistics Collector" · dokumentacja rozszerzenia `pg_stat_statements`

Revision #43
Created 2026-09-24 23:08:14 UTC by Tomasz Konieczny
Updated 2026-10-26 14:12:41 UTC by Tomasz Konieczny