Skan sekwencyjny, skan indeksu, lookup: sposoby dotarcia do wiersza

Ten sam warunek WHERE da się zaspokoić na kilka różnych sposobów fizycznego dotarcia do danych — ta strona przechodzi po metodach dostępu do pojedynczej tabeli, a potem po metodach złączenia dwóch tabel, i pokazuje, jakie warunki skłaniają optymalizator do wyboru jednej metody zamiast drugiej.

Skróty użyte na tej stronie (1)

Skrót	Co znaczy
SQL	Structured Query Language: język zapytań do relacyjnych baz danych

Czym jest metoda dostępu

Pojęcie: metoda dostępu. Konkretny sposób, w jaki silnik fizycznie dociera do wierszy tabeli potrzebnych do policzenia wyniku zapytania — czyta wszystko po kolei, nawiguje po indeksie, czy łączy kilka technik naraz. To samo zapytanie logiczne, `SELECT ... WHERE ...`, nie narzuca jednej metody dostępu — optymalizator wybiera ją na podstawie statystyk, istniejących indeksów i szacowanej liczby pasujących wierszy, dokładnie tak, jak opisuje strona Statystyki i optymalizator kosztowy.

Seq Scan / Table Scan

> **Pojęcie: skan sekwencyjny (Seq Scan / Table Scan).** Odczyt wszystkich stron tabeli po kolei, od pierwszej do ostatniej, ze sprawdzeniem warunku dla każdego napotkanego wiersza z osobna. To najprostsza możliwa metoda dostępu — nie wymaga żadnego indeksu i nie zależy od tego, jak wartości są rozłożone w tabeli.

Skan sekwencyjny bywa najlepszym wyborem w trzech sytuacjach: kiedy zapytanie i tak potrzebuje większości albo całości wierszy tabeli, kiedy tabela jest na tyle mała, że mieści się w garstce stron niezależnie od metody, albo kiedy żaden istniejący indeks nie pasuje do warunku. Kluczowy jest pierwszy przypadek — optymalizator kosztowy świadomie rezygnuje z istniejącego, pasującego indeksu, kiedy szacuje, że warunek zwróci dużą część tabeli, bo w tej sytuacji koszt tysięcy pojedynczych, rozproszonych odczytów przez indeks przewyższa koszt jednego uporządkowanego przejścia po danych leżących fizycznie obok siebie na dysku.

Przykład planu z tabeli `zamowienia` liczącej cztery miliony wierszy, gdzie warunek `status <> 'anulowane'` obejmuje niemal trzy czwarte tabeli:

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on zamowienia  (cost=0.00..98450.00 rows=2980000 width=42)
   Filter: (status <> 'anulowane'::text)
(2 rows)
```

Mimo istnienia indeksu na `status`, optymalizator wybrał skan sekwencyjny — bo prawie trzy miliony pasujących wierszy oznaczałyby prawie trzy miliony pojedynczych doskoków do tabeli, gdyby przejść przez indeks, znacznie drożej niż jedno uporządkowane przejście po całej tabeli.

Index Scan, Index Only Scan, Bitmap Index Scan w PostgreSQL

> **Pojęcie: Index Scan.** Metoda dostępu, w której silnik najpierw nawiguje po B-drzewie indeksu, żeby znaleźć pozycje pasujące do warunku, a potem dla każdej znalezionej pozycji osobno doskakuje do wiersza w tabeli, żeby odczytać kolumny spoza indeksu. Ten drugi krok, doskok do tabeli, jest kosztem, który rośnie z liczbą pasujących wierszy — im więcej trafień, tym więcej rozproszonych, pojedynczych odczytów.

```
                                QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using ix_zamowienia_klient on zamowienia  (cost=0.43..24.63 rows=32 width=42)
```

```
   Index Cond: (klient_id = 4)
(2 rows)
```

> **Pojęcie: Index Only Scan.** Metoda dostępu, która pomija doskok do tabeli w ogóle, bo indeks jest pokrywający dla danego zapytania — wszystkie potrzebne kolumny leżą już w samym indeksie, dokładnie jak opisuje strona Indeks złożony, pokrywający, częściowy, funkcyjny.

```
                                QUERY PLAN
---------------------------------------------------------------------------------
 Index Only Scan using ix_zamowienia_klient_status on zamowienia  (cost=0.43..8.63 rows=32
width=12)
   Index Cond: (klient_id = 4)
   Heap Fetches: 0
(3 rows)
```

`Heap Fetches: 0` potwierdza, że silnik ani razu nie musiał zajrzeć do tabeli — cała odpowiedź wyszła z samego indeksu.

> **Pojęcie: Bitmap Index Scan.** Metoda dwuetapowa, w której silnik najpierw przechodzi po indeksie i buduje w pamięci mapę bitową stron tabeli zawierających pasujące wiersze, zamiast doskakiwać do każdego wiersza z osobna, a potem w drugim etapie (`Bitmap Heap Scan`) czyta te strony w kolejności fizycznej, jedna po drugiej, sprawdzając warunek dokładnie tam, gdzie mapa wskazała trafienie. Ten sposób jest kompromisem między czystym `Index Scan` a `Seq Scan` — sprawdza się przy średniej selektywności, kiedy pasujących wierszy jest zbyt dużo na tanie pojedyncze doskoki, ale wciąż wyraźnie mniej niż cała tabela.

```
                                QUERY PLAN
---------------------------------------------------------------------------------
 Bitmap Heap Scan on zamowienia  (cost=980.22..42450.10 rows=41200 width=42)
   Recheck Cond: (status = 'nowe'::text)
   ->  Bitmap Index Scan on ix_zamowienia_status  (cost=0.00..970.02 rows=41200 width=0)
         Index Cond: (status = 'nowe'::text)
(4 rows)
```

Dodatkowa zaleta bitmapowego dostępu ujawnia się przy kilku warunkach na różnych indeksach naraz — silnik potrafi zbudować osobne mapy bitowe z dwóch indeksów i połączyć je operacją bitową `AND` (`BitmapAnd`) albo `OR` (`BitmapOr`), zamiast wybierać tylko jeden z indeksów i filtrować

resztę przez odczyt wierszy.

Clustered Index Seek/Scan i Key Lookup w MSSQL

SQL Server nazywa te same idee nieco inaczej, bo domyślnie tabela jest fizycznie zorganizowana wokół indeksu klastrowego. **Pojęcie: Clustered Index Seek.** Nawigacja po B-drzewie indeksu klastrowego wprost do właściwego zakresu wierszy — ponieważ indeks klastrowy jest tabelą, ten krok od razu daje pełny wiersz, bez osobnego doskoku, odpowiadając funkcjonalnie `Index Only Scan` z PostgreSQL. Clustered Index Scan to odpowiednik `Seq Scan` — przejście po wszystkich wierszach indeksu klastrowego po kolei.

Kiedy zapytanie filtruje po kolumnie objętej indeksem nieklastrowanym, ale potrzebuje też kolumn spoza tego indeksu, SQL Server najpierw wykonuje `Index Seek` na indeksie nieklastrowanym, a potem dla każdego znalezionego wiersza osobno sięga do indeksu klastrowego po resztę kolumn — ten drugi krok nazywa się Key Lookup.

```
|--Nested Loops(Inner Join)
     |--Index Seek(OBJECT:([Zamowienia].[IX_Zamowienia_KlientId]), SEEK:([KlientId]=4))
     |--Key Lookup(OBJECT:([Zamowienia].[PK_Zamowienia]))
```

`Key Lookup` kosztuje tym więcej, im więcej wierszy trzeba doszukać — dokładnie ten sam mechanizm co doskok do tabeli w `Index Scan` z PostgreSQL. Istnieje punkt przegięcia, powyżej którego SQL Server przestaje opłacalnie korzystać z indeksu nieklastrowanego z osobnym `Key Lookup` na każdy wiersz i przechodzi zamiast tego na pełny `Clustered Index Scan` — w praktyce dzieje się to zwykle w okolicach pojedynczych procent całej tabeli, dokładna granica zależy od względnego kosztu odczytu losowego i sekwencyjnego na danym sprzęcie oraz od statystyk tabeli, nie jest sztywno zapisana w dokumentacji jako stała liczba.

```
|--Clustered Index Scan(OBJECT:([Zamowienia].[PK_Zamowienia]), WHERE:([KlientId]=4))
```

Ten sam warunek `KlientId = 4`, przy tabeli, w której klient ma bardzo dużo zamówień, może więc dostać zupełnie inny plan niż identyczny warunek dla klienta z garstką zamówień — dokładnie ten sam mechanizm doboru metody na podstawie szacowanej liczby wierszy, co opisuje strona Statystyki i optymalizator kosztowy.

Metody złączenia

Poza dostępem do pojedynczej tabeli, optymalizator wybiera też metodę, jaką dwie strony złączenia zostaną ze sobą zestawione — strona Czym jest złączenie: iloczyn kartezjański i warunek, który go ogranicza wprowadza te trzy metody skrótowo, tutaj rozwijamy je pełniej.

> **Pojęcie: Nested Loop.** Dla każdego wiersza tabeli zewnętrznej silnik osobno przeszukuje tabelę wewnętrzną w poszukiwaniu dopasowań. Sprawdza się najlepiej, kiedy tabela zewnętrzna ma niewiele wierszy i kiedy tabela wewnętrzna ma indeks na kolumnie złączenia — wtedy każde pojedyncze przeszukanie jest tanim zejściem po B-drzewie, a nie skanem całej tabeli wewnętrznej.

```
 Nested Loop  (cost=0.43..612.30 rows=32 width=48)
   ->  Index Scan using ix_zamowienia_klient on zamowienia z  (cost=0.43..24.63 rows=32
width=24)
         Index Cond: (klient_id = 4)
   ->  Index Scan using klienci_pkey on klienci k  (cost=0.15..0.17 rows=1 width=32)
         Index Cond: (id = z.klient_id)
```

> **Pojęcie: Hash Join.** Silnik buduje w pamięci tablicę haszującą z mniejszej z dwóch stron złączenia, a potem przechodzi raz po większej stronie, sprawdzając każdy jej wiersz w tej tablicy. Sprawdza się przy dużych zbiorach po obu stronach, kiedy nie ma przydatnego indeksu na kolumnie złączenia i kiedy wynik nie musi wyjść posortowany.

```
 Hash Join  (cost=15234.11..98234.55 rows=41200 width=48)
   Hash Cond: (z.klient_id = k.id)
   ->  Seq Scan on zamowienia z  (cost=0.00..78450.00 rows=41200 width=24)
   ->  Hash  (cost=12.60..12.60 rows=180 width=32)
         ->  Seq Scan on klienci k  (cost=0.00..12.60 rows=180 width=32)
```

> **Pojęcie: Merge Join.** Obie strony złączenia są posortowane po kolumnie złączenia — z indeksu albo z jawnego sortowania — a silnik przesuwa się po obu jednocześnie, jak zamek błyskawiczny, zestawiając kolejne pasujące wartości bez budowania żadnej dodatkowej struktury pomocniczej. Sprawdza się, kiedy obie strony i tak są już posortowane albo kiedy wynik zapytania sam musi wyjść posortowany po tej kolumnie.

```
 Merge Join  (cost=1024.30..4210.80 rows=41200 width=48)
   Merge Cond: (z.klient_id = k.id)
   ->  Index Scan using ix_zamowienia_klient on zamowienia z  (cost=0.43..3120.00 rows=41200
width=24)
```

```
->  Index Scan using klienci_pkey on klienci k  (cost=0.15..48.20 rows=180 width=32)
```

Metoda złączenia	Kiedy optymalizator wybiera
Nested Loop	mała tabela po jednej stronie, indeks na kolumnie złączenia po drugiej stronie
Hash Join	obie strony duże, brak przydatnego indeksu na kolumnie złączenia, wynik nie musi być posortowany
Merge Join	obie strony już posortowane po kolumnie złączenia, zwykle dzięki istniejącym indeksom, albo wymagane sortowanie wyniku

Częste błędy

Traktowanie Seq Scan jako oznaki źle zaprojektowanej bazy. Skan sekwencyjny na dużej tabeli po zapytaniu, które i tak zwraca większość jej wierszy, jest poprawną, świadomą decyzją optymalizatora, a nie objawem brakującego indeksu — dodanie indeksu w takiej sytuacji nic nie zmieni w wyborze planu.

Zdziwienie zmianą planu dla tego samego zapytania z inną wartością parametru. `WHERE klient_id = 4` i `WHERE klient_id = 812` mogą dostać zupełnie różne plany, jeśli jeden klient ma trzydzieści zamówień, a drugi trzydzieści tysięcy — optymalizator dobiera metodę pod szacowaną liczbę wierszy dla konkretnej wartości, nie pod samą strukturę zapytania. Zjawisko to wraca szerzej jako parameter sniffing na stronie Statystyki i optymalizator kosztowy.

Oczekiwanie Merge Join tam, gdzie żadna ze stron nie jest posortowana. Bez istniejącego indeksu albo jawnego sortowania po kolumnie złączenia silnik musiałby najpierw posortować obie strony, co samo w sobie kosztuje — Hash Join bywa wtedy tańszy mimo braku sortowania na wyjściu.

□□ Ćwiczenia

> Zadania sprawdzające.

Ćwiczenie 1 — dobór metody dostępu

Tabela `zamowienia` ma cztery miliony wierszy i indeks na `klient_id`. Klient o `id = 4` ma dwadzieścia osiem zamówień. Jaką metodę dostępu prawdopodobnie wybierze optymalizator dla `WHERE klient_id = 4`, a jaką dla `WHERE klient_id IS NOT NULL`, skoro tylko jedno zamówienie w całej

tabeli ma `klient_id` równe `NULL`?

Odpowiedź

Dla `klient_id = 4`, z dwudziestoma ośmioma pasującymi wierszami z czterech milionów, optymalizator wybierze `Index Scan` — niewielka liczba trafień czyni doskoki do tabeli tanimi. Dla `klient_id IS NOT NULL`, gdzie pasują niemal wszystkie cztery miliony wierszy poza jednym, optymalizator wybierze `Seq Scan` — przy tak wysokim odsetku pasujących wierszy przejście po całej tabeli jest tańsze niż niemal cztery miliony pojedynczych doskoków przez indeks.

Pytania kontrolne

1. Czym różni się Index Scan od Index Only Scan?

Odpowiedź

`Index Scan` po znalezieniu pozycji w indeksie doskakuje do tabeli, żeby odczytać kolumny spoza indeksu. `Index Only Scan` pomija ten doskok całkowicie, bo indeks jest pokrywający dla zapytania — wszystkie potrzebne kolumny leżą już w samym indeksie.

2. Dlaczego Merge Join wymaga posortowanych danych po obu stronach, a Hash Join nie?

Odpowiedź

`Merge Join` przesuwa się po obu stronach jednocześnie i porównuje bieżące wartości — ten mechanizm działa tylko wtedy, gdy obie strony są w tej samej kolejności sortowania po kolumnie złączenia. `Hash Join` nie zależy od żadnego porządku — buduje tablicę haszującą z jednej strony i sprawdza w niej wartości drugiej strony niezależnie od kolejności, w jakiej te wiersze faktycznie się pojawiają.

Powiązane: Statystyki i optymalizator kosztowy · Plan wykonania: jak go wywołać i jak go czytać · Indeks złożony, pokrywający, częściowy, funkcyjny · Czym jest złączenie: iloczyn kartezjański i warunek, który go ogranicza

Źródła: dokumentacja PostgreSQL, rozdział „Query Planning” i „Join Strategies” · Microsoft Docs, „Showplan Logical and Physical Operators Reference” · Oracle Database SQL Tuning Guide, rozdział „Optimizer Access Paths”

Revision #47
Created 2026-09-24 22:42:43 UTC by Tomasz Konieczny

Updated 2026-10-26 14:11:45 UTC by Tomasz Konieczny