WHERE: filtrowanie wierszy

> `WHERE` decyduje, który wiersz zostaje w wyniku, sprawdzając dla każdego z osobna jeden warunek logiczny — ta strona przechodzi przez operatory, którymi ten warunek się buduje, i przez to, kiedy taki warunek przestaje pozwalać silnikowi skorzystać z indeksu.

◫ Skróty użyte na tej stronie (4)

Skrót	Co znaczy
ISO	International Organization for Standardization: organizacja, która wydaje normy techniczne używane na całym świecie
POSIX	Portable Operating System Interface: wspólny zestaw zasad, dzięki któremu polecenie działa tak samo na różnych Uniksach
SQL	Structured Query Language: język zapytań do relacyjnych baz danych
T-SQL	Transact-SQL: dialekt SQL używany w Microsoft SQL Server, wywodzący się z języka zapytań oryginalnego silnika Sybase

Fiszka

◫ ZMIENIA SYSTEM

Porównanie: =, <>, <, >, <=, >=

```
SELECT id, status, wartosc FROM zamowienia WHERE wartosc > 1000;
```

Silnik sprawdza warunek dla każdego wiersza z osobna i zostawia w wyniku tylko te, dla których warunek dał wartość prawda. `<>` i `!=` to ten sam operator różności — obie formy działają we wszystkich czterech silnikach z tej książki.

Przykład:

```
 id |    status     | wartosc
----+---------------+---------
```

```
101 | zrealizowane  | 1250.00
104 | w realizacji  | 2890.00
107 | zrealizowane  | 3200.00
(3 rows)
```

◫◫ ZMIENIA SYSTEM

Łączenie warunków: AND, OR, NOT

```
SELECT id, status, wartosc FROM zamowienia
WHERE status = 'zrealizowane' OR status = 'nowe' AND wartosc > 500
ORDER BY id;
```

`AND` wiąże mocniej niż `OR`, tak samo jak mnożenie wiąże mocniej niż dodawanie w arytmetyce. To zapytanie silnik czyta jako `status = 'zrealizowane' OR (status = 'nowe' AND wartosc > 500)`, nie jako `(status = 'zrealizowane' OR status = 'nowe') AND wartosc > 500` — różnica robi różnicę w wyniku, co widać poniżej.

Przykład:

```
 id |    status     | wartosc
----+---------------+---------
101 | zrealizowane  | 1250.00
102 | zrealizowane  |  430.50
106 | nowe          |  899.00
107 | zrealizowane  | 3200.00
(4 rows)
```

Ten sam warunek z nawiasem wymuszającym inną kolejność:

```
SELECT id, status, wartosc FROM zamowienia
WHERE (status = 'zrealizowane' OR status = 'nowe') AND wartosc > 500
ORDER BY id;
```

Przykład:

```
 id |    status     | wartosc
----+---------------+---------
101 | zrealizowane  | 1250.00
106 | nowe          |  899.00
```

```
107 | zrealizowane  | 3200.00
(3 rows)
```

Zamówienie 102 (zrealizowane, 430.50) wypadło z wyniku, bo tym razem warunek na kwotę dotyczy obu statusów naraz, nie tylko statusu `nowe`. Nawias nie jest tu kosmetyką — zmienia treść zapytania. `NOT` odwraca wynik warunku, który następuje po nim: `NOT status = 'anulowane'` znaczy to samo co `status <> 'anulowane'`, ale przy złożonych warunkach w nawiasie `NOT (a AND b)` czyta się wyraźnie inaczej niż `NOT a AND NOT b`.

TYLKO ODCZYT

IN — dopasowanie do listy wartości

```
SELECT id, status FROM zamowienia WHERE status IN ('nowe', 'w realizacji');
```

`IN` zastępuje ciąg warunków połączonych `OR` na tej samej kolumnie. `status IN ('nowe', 'w realizacji')` to dokładnie to samo, co `status = 'nowe' OR status = 'w realizacji'`, tylko czytelniejsze przy dłuższej liście.

Przykład:

```
 id |    status
----+--------------
104 | w realizacji
105 | nowe
106 | nowe
(3 rows)
```

TYLKO ODCZYT

BETWEEN — zakres domknięty z obu stron

```
SELECT id, data, wartosc FROM zamowienia WHERE data BETWEEN '2024-01-01' AND '2024-02-29';
```

`BETWEEN a AND b` znaczy `>= a AND <= b` — obie granice wliczone. Dla dat to wygodne, dopóki granica górna nie ma godziny: `BETWEEN '2024-01-01' AND '2024-01-31'` na kolumnie typu `timestamp` pominie wiersze z 31 stycznia po północy, bo `2024-01-31` bez godziny to `2024-01-31 00:00:00`.

Przykład:

```
 id |    data    | wartosc
----+------------+---------
101 | 2024-01-15 | 1250.00
102 | 2024-02-03 |  430.50
103 | 2024-01-20 |    0.00
(3 rows)
```

TYLKO ODCZYT

LIKE — dopasowanie wzorca: % i _

```
SELECT nazwa FROM klienci WHERE nazwa LIKE 'K%';
```

> **Pojęcie: wzorzec LIKE.** Ciąg znaków, w którym `%` zastępuje dowolny fragment tekstu, także pusty, a `_` zastępuje dokładnie jeden znak. `'K%'` dopasowuje każdy tekst zaczynający się od litery `K`, niezależnie od długości reszty.

Przykład:

```
     nazwa
----------------
 Kowalski Handel
(1 row)
```

`_` pasuje do dokładnie jednego znaku, co przydaje się, gdy wiemy, ile znaków ma brakujący fragment:

```
SELECT nazwa FROM produkty WHERE nazwa LIKE 'Kabel HDMI _m';
```

Przykład:

```
      nazwa
------------------
 Kabel HDMI 2m
(1 row)
```

Gdyby zapytanie użyło `%` zamiast `_`, dopasowałoby też hipotetyczny `Kabel HDMI 15m` — `_` ogranicza dopasowanie do dokładnie jednego znaku w tym miejscu.

TYLKO ODCZYT

ILIKE — dopasowanie bez rozróżniania wielkości liter

```
SELECT nazwa FROM klienci WHERE nazwa ILIKE 'zielona%';
```

`ILIKE` to rozszerzenie PostgreSQL — działa jak `LIKE`, ale ignoruje wielkość liter. Standard SQL i pozostałe trzy silniki z tej książki tego operatora nie znają; w MySQL to samo osiąga się przez `LIKE` na kolumnie z porównaniem case-insensitive (domyślne dla większości collation), w SQL Server i Oracle — przez `LOWER()` po obu stronach porównania albo odpowiednie collation.

Przykład:

```
        nazwa
----------------------
 Zielona Dolina S.A.
(1 row)
```

SQL = Structured Query Language: język zapytań do relacyjnych baz danych

TYLKO ODCZYT

Wyrażenia regularne w skrócie

```
SELECT nazwa FROM produkty WHERE nazwa ~ '[0-9]';
```

Operator `~` w PostgreSQL sprawdza dopasowanie do wyrażenia regularnego POSIX — pełniejszego narzędzia niż wzorzec `LIKE`, bo pozwala na klasy znaków, powtórzenia i alternatywy. `[0-9]` dopasowuje dowolną cyfrę gdziekolwiek w tekście. MySQL ma `REGEXP`, SQL Server nie ma wbudowanego operatora regex w rdzeniu T-SQL (dopiero przez CLR albo `PATINDEX` z ograniczonym wzorcem), Oracle ma `REGEXP_LIKE()`.

Przykład:

```
       nazwa
-------------------
 Monitor 24 cale
```

```
 Kabel HDMI 2m
(2 rows)
```

POSIX = Portable Operating System Interface: wspólny zestaw zasad, dzięki któremu polecenie działa tak samo na różnych Uniksach · **SQL** = Structured Query Language: język zapytań do relacyjnych baz danych · **T-SQL** = Transact-SQL: dialekt SQL używany w Microsoft SQL Server, wywodzący się z języka zapytań oryginalnego silnika Sybase

TYLKO ODCZYT

Wszystkie operatory

Operator	Działanie
`=`, `<>`	równość, różność
`<`, `>`, `<=`, `>=`	porównanie
`AND`, `OR`, `NOT`	łączenie i negacja warunków
`IN (...)`	dopasowanie do listy wartości
`BETWEEN a AND b`	zakres domknięty z obu stron
`LIKE 'wzorzec'`	dopasowanie tekstu, `%` i `_`
`ILIKE 'wzorzec'`	jak `LIKE`, bez rozróżniania wielkości liter (PostgreSQL)
`~`, `REGEXP_LIKE()`	wyrażenie regularne (składnia zależna od silnika)

Uważaj

- `IN` z listą zawierającą `NULL` nie dopasuje wierszy z `NULL` w kolumnie testowanej — trójwartościowa logika opisana na stronie NULL: trzecia wartość logiczna i pułapki, które z niej wynikają.
- `LIKE '%tekst%'` z wiodącym `%` nie da się wesprzeć zwykłym indeksem B-drzewa — silnik musi przejrzeć każdy wiersz.

Po co to istnieje

> **Pojęcie: predykat.** Wyrażenie logiczne policzone osobno dla każdego wiersza, które zwraca prawdę, fałsz albo — gdy w grę wchodzi `NULL` — wartość nieznaną. `WHERE` zostawia w wyniku wyłącznie wiersze, dla których predykat dał prawdę; wiersze z wynikiem fałsz i wiersze z wynikiem nieznana są odrzucane tak samo. Ta trzecia możliwość nazywa się trójwartościową logiką i ma własną stronę, NULL: trzecia wartość logiczna i pułapki, które z niej wynikają, bo konsekwencje są nieoczywiste na tyle, żeby zasługiwać na osobne omówienie.

Jak to działa

> **Pojęcie: sargowalność.** Właściwość warunku w `WHERE`, która mówi, czy optymalizator może go wykorzystać do przeszukania indeksu, zamiast czytać każdy wiersz tabeli po kolei. Nazwa pochodzi od angielskiego skrótu SARG — Search ARGument. Warunek jest sargowalny, kiedy silnik potrafi z niego bezpośrednio wyznaczyć zakres wartości do sprawdzenia w indeksie; przestaje być sargowalny, kiedy kolumna indeksowana trafia pod funkcję albo wyrażenie.

```
SELECT id, data, wartosc FROM zamowienia WHERE EXTRACT(YEAR FROM data) = 2024;
```

Indeks założony na kolumnie `data` przechowuje posortowane wartości `data`, nie wartości `EXTRACT(YEAR FROM data)`. Żeby sprawdzić ten warunek, silnik musiałby policzyć `EXTRACT(YEAR FROM data)` dla każdego wiersza z osobna — a to oznacza pełne przejście po tabeli, mimo istniejącego indeksu.

```
SELECT id, data, wartosc FROM zamowienia WHERE data >= '2024-01-01' AND data < '2025-01-01';
```

To samo kryterium selekcji, zapisane jako zakres bezpośrednio na kolumnie `data`, bez żadnej funkcji. Silnik czyta z indeksu ciągły zakres wartości od `2024-01-01` włącznie do `2025-01-01` wyłącznie — dokładnie do tego indeksy B-drzewa są stworzone. Rozdział Kiedy indeks szkodzi wraca do sargowalności przy okazji innych pułapek na indeksy.

Częste błędy

Owijanie kolumny indeksowanej funkcją w WHERE. `WHERE UPPER(nazwa) = 'KOWALSKI HANDEL'` psuje sargowalność tak samo jak `EXTRACT(YEAR FROM data)`. Rozwiązanie zależy od silnika: indeks funkcyjny na `UPPER(nazwa)` w PostgreSQL i Oracle, kolumna wyliczana i zaindeksowana w SQL Server, albo — najprościej — porównanie bez owijania kolumny, jeśli dane i tak są już

znormalizowane co do wielkości liter.

Wiodący `%` we wzorcu LIKE. `LIKE '%Handel'` wymaga przejrzenia każdego wiersza, bo dopasowanie może zaczynać się w dowolnym miejscu tekstu. `LIKE 'Handel%'` bez wiodącego znacznika pozwala silnikowi użyć indeksu na kolumnie tekstowej tak samo jak zwykłe porównanie `>=`.

Mieszanie AND i OR bez nawiasów w warunku, który ma więcej niż dwa człony. Pierwszy przykład w Fiszce pokazuje różnicę w wyniku między zapisem z nawiasem i bez niego — przy trzech i więcej warunkach ryzyko pomyłki w priorytecie rośnie, a nawias kosztuje jeden znak.

□□ Ćwiczenia

> Zadania sprawdzające.

Ćwiczenie 1 — priorytety operatorów

Bez uruchamiania zapytania przewidujemy, które zamówienia zwróci `SELECT id FROM zamowienia WHERE status = 'anulowane' OR status = 'nowe' AND wartosc = 0;`, mając dane z Fiszki tej strony.

Odpowiedź

`AND` wiąże mocniej, więc warunek czyta się jako `status = 'anulowane' OR (status = 'nowe' AND wartosc = 0)`. Zamówienia o statusie `anulowane` to 103 i 108 — oba trafiają do wyniku niezależnie od kwoty. Zamówień o statusie `nowe` i kwocie równej zero nie ma (105 ma 150.00, 106 ma 899.00), więc wynik to `103, 108`.

Pytania kontrolne

1. Dlaczego `WHERE UPPER(miasto) = 'WARSZAWA'` bywa wolniejsze niż `WHERE miasto = 'Warszawa'`, mimo że logicznie sprawdzają to samo przy jednolitych danych?

Odpowiedź

Funkcja `UPPER()` owija kolumnę indeksowaną, więc silnik nie może użyć indeksu na `miasto` wprost — musiałby policzyć `UPPER(miasto)` dla każdego wiersza. Warunek przestaje być sargowalny.

2. Czym różni się `IN ('nowe', 'w realizacji')` od `BETWEEN`?

Odpowiedź

`IN` sprawdza przynależność do jawnie wypisanej listy dyskretnych wartości, `BETWEEN` sprawdza przynależność do ciągłego zakresu domkniętego z obu stron. `IN` ma sens dla tekstu i kategorii, `BETWEEN` — dla liczb i dat.

Powiązane: NULL: trzecia wartość logiczna i pułapki, które z niej wynikają · Czym jest indeks: i dlaczego to nie jest „przyspieszacz tabeli” · Kiedy indeks szkodzi

Źródła: dokumentacja PostgreSQL, rozdział „Pattern Matching” i „Comparison Functions and Operators” · ISO/IEC 9075-2:2016 § 6.34

Revision #47
Created 2026-09-24 22:41:46 UTC by Tomasz Konieczny
Updated 2026-10-26 14:10:48 UTC by Tomasz Konieczny