Podzapytania: skalarne, wierszowe, skorelowane, EXISTS

Podzapytanie to SELECT schowany wewnątrz innego zapytania — może zwrócić jedną wartość, listę wartości albo całą tabelę, w zależności od miejsca, w którym stoi, i to ten kontekst decyduje, jakich operatorów wolno przy nim użyć.

Skróty użyte na tej stronie (2)

Skrót	Co znaczy
ISO	International Organization for Standardization: organizacja, która wydaje normy techniczne używane na całym świecie
SQL	Structured Query Language: język zapytań do relacyjnych baz danych

Fiszka

ZMIENIA SYSTEM

Podzapytanie skalarne w WHERE

```
SELECT k.nazwa, z.id AS zamowienie_id, z.wartosc
FROM zamowienia z
JOIN klienci k ON k.id = z.klient_id
WHERE z.wartosc > (SELECT AVG(wartosc) FROM zamowienia);
```

Pojęcie: podzapytanie skalarne. Podzapytanie, które zwraca dokładnie jedną wartość — jeden wiersz, jedna kolumna. Tylko takie podzapytanie wolno postawić po

jednej stronie zwykłego operatora porównania (`=`, `>`, `<`), bo tylko wtedy porównanie ma jednoznaczny sens.

Przykład:

```
# avg=1102.44
            nazwa             | zamowienie_id | wartosc
------------------------------+---------------+---------
 Nowak i Wspólnicy Sp. z o.o. |           101 | 1250.00
 Zielona Dolina S.A.          |           104 | 2890.00
 Baltic Trade Sp. z o.o.      |           107 | 3200.00
(3 rows)
```

Silnik liczy podzapytanie `SELECT AVG(wartosc) FROM zamowienia` raz, dostaje jedną liczbę — tu około 1102,44 — i porównuje z nią każdy wiersz zewnętrznego zapytania, dokładnie tak jak porównałby go z liczbą wpisaną wprost w kod.

join = połącz dwa pliki tekstowe po wspólnej kolumnie, jak złączenie tabel w bazie danych

TYLKO ODCZYT

Podzapytanie wielowierszowe: IN

```
SELECT nazwa
FROM klienci
WHERE id IN (SELECT klient_id FROM zamowienia WHERE status = 'zrealizowane');
```

Pojęcie: podzapytanie wielowierszowe. Podzapytanie zwracające jedną kolumnę, ale dowolną liczbę wierszy — używane z `IN`, `ANY` albo `ALL`, nigdy z gołym `=`, bo `=` nie wie, co zrobić z więcej niż jedną wartością po prawej stronie.

Przykład:

```
            nazwa
------------------------------
 Nowak i Wspólnicy Sp. z o.o.
 Baltic Trade Sp. z o.o.
(2 rows)
```

◫◫ ZMIENIA SYSTEM

ANY i ALL — porównanie z każdym albo z każdym naraz

```
SELECT nazwa, kategoria, cena
FROM produkty
WHERE cena > ANY (SELECT cena FROM produkty WHERE kategoria = 'elektronika');
```

`> ANY (...)` znaczy „większe niż co najmniej jedna wartość z listy" — wystarczy pokonać najsłabszego kandydata, żeby warunek dał prawdę.

Przykład:

```
         nazwa          |  kategoria  |  cena
------------------------+-------------+---------
 Monitor 24 cale        | elektronika |  650.00
 Klawiatura mechaniczna | elektronika |  220.00
 Biurko regulowane      | meble       | 1100.00
 Fotel biurowy          | meble       |  890.00
 Lampka biurkowa        | meble       |   90.00
(5 rows)
```

```
SELECT nazwa, kategoria, cena
FROM produkty
WHERE cena > ALL (SELECT cena FROM produkty WHERE kategoria = 'elektronika');
```

`> ALL (...)` znaczy „większe niż każda wartość z listy" — trzeba pokonać najmocniejszego kandydata, nie tylko najsłabszego.

Przykład:

```
       nazwa       | kategoria |  cena
-------------------+-----------+---------
 Biurko regulowane | meble     | 1100.00
 Fotel biurowy     | meble     |  890.00
(2 rows)
```

Różnica między pięcioma a dwoma wierszami pokazuje odległość między „lepsze od któregoś" a „lepsze od wszystkich" — `Klawiatura mechaniczna` (220,00) jest droższa od najtańszego produktu elektronicznego (`Kabel HDMI 2m`, 25,00), więc przechodzi `ANY`, ale nie jest droższa od najdroższego (`Monitor 24 cale`, 650,00), więc odpada przy `ALL`.

🔒 TYLKO ODCZYT

Tabela pochodna w FROM

```
SELECT k.nazwa, suma.wartosc_lacznie
FROM klienci k
JOIN (
    SELECT klient_id, SUM(wartosc) AS wartosc_lacznie
    FROM zamowienia
    GROUP BY klient_id
) suma ON suma.klient_id = k.id;
```

> **Pojęcie: tabela pochodna (derived table).** Podzapytanie postawione w miejscu, gdzie normalnie stoi nazwa tabeli — w klauzuli `FROM`. Silnik traktuje jego wynik jak zwykłą tabelę tymczasową, istniejącą tylko na czas tego jednego zapytania, i wymaga dla niej aliasu — tu `suma`.

Przykład:

```
            nazwa             | wartosc_lacznie
------------------------------+-----------------
 Nowak i Wspólnicy Sp. z o.o. |         1680.50
 Kowalski Handel              |          150.00
 Zielona Dolina S.A.          |         2890.00
 Baltic Trade Sp. z o.o.      |         3200.00
 Podlaski Handel              |            0.00
(5 rows)
```

🔗 **join** = połącz dwa pliki tekstowe po wspólnej kolumnie, jak złączenie tabel w bazie danych

🔒 ZMIENIA SYSTEM

Podzapytanie skorelowane i EXISTS

```
SELECT nazwa
FROM klienci k
WHERE EXISTS (
    SELECT 1 FROM zamowienia z WHERE z.klient_id = k.id AND z.wartosc > 1000
```

```
);
```

> **Pojęcie: podzapytanie skorelowane.** Podzapytanie, które odwołuje się do kolumny zapytania zewnętrznego — tu `k.id` — i przez to nie da się go policzyć raz z góry, tak jak podzapytania skalarnego z pierwszej Fiszki. Silnik logicznie wykonuje je od nowa dla każdego wiersza zewnętrznego zapytania, podstawiając za każdym razem inną wartość `k.id`.

`EXISTS` nie pyta o żadną konkretną wartość — pyta wyłącznie, czy podzapytanie zwróciło choćby jeden wiersz. `SELECT 1` w środku jest umowną konwencją: liczba wybrana w `SELECT` nie ma żadnego znaczenia, bo `EXISTS` i tak jej nie czyta, tylko sprawdza samo istnienie wiersza.

Przykład:

```
            nazwa
------------------------------
 Nowak i Wspólnicy Sp. z o.o.
 Zielona Dolina S.A.
 Baltic Trade Sp. z o.o.
(3 rows)
```

`NOT EXISTS` odwraca pytanie na „czy podzapytanie nie zwróciło żadnego wiersza":

```
SELECT nazwa
FROM klienci k
WHERE NOT EXISTS (
    SELECT 1 FROM zamowienia z WHERE z.klient_id = k.id AND z.wartosc > 1000
);
```

Przykład:

```
          nazwa
-------------------------
 Kowalski Handel
 Podlaski Handel
 Elektro-System Kowalczyk
(3 rows)
```

Po co to istnieje

> **Pojęcie: podzapytanie (subquery).** Pełne zapytanie `SELECT`, umieszczone wewnątrz innego zapytania zamiast wewnątrz osobnego polecenia. Miejsce, w którym podzapytanie stoi, decyduje o tym, co wolno mu zwrócić i jakich operatorów wolno przy nim użyć — w `SELECT` i w prostym warunku `WHERE` musi być skalarne (jedna wartość), w `FROM` staje się tabelą pochodną, a przy `IN`, `ANY`, `ALL` i `EXISTS` może zwracać wiele wierszy.

Podzapytania i JOIN-y często odpowiadają na to samo pytanie z dwóch różnych kierunków. Zapytanie z tabelą pochodną z Fiszki da się zapisać bez podzapytania wcale, złączeniem po zwykłym `GROUP BY`:

```
SELECT k.nazwa, SUM(z.wartosc) AS wartosc_lacznie
FROM klienci k
JOIN zamowienia z ON z.klient_id = k.id
GROUP BY k.nazwa;
```

Przykład:

```
            nazwa             |  wartosc_lacznie
------------------------------+-----------------
 Nowak i Wspólnicy Sp. z o.o. |         1680.50
 Kowalski Handel              |          150.00
 Zielona Dolina S.A.          |         2890.00
 Baltic Trade Sp. z o.o.      |         3200.00
 Podlaski Handel              |            0.00
(5 rows)
```

Wynik identyczny, droga inna. Ogólna zasada: gdy chodzi o dołączenie dodatkowych kolumn z drugiej tabeli do każdego wiersza — złączenie. Gdy chodzi o samo pytanie „czy istnieje", „ile jest" albo o filtr oparty na zagregowanej wartości z drugiej tabeli, bez potrzeby wyciągania z niej żadnych dodatkowych kolumn — podzapytanie, zwłaszcza `EXISTS`, bywa czytelniejsze i czasem szybsze, bo silnik może przerwać sprawdzanie przy pierwszym trafieniu, zamiast liczyć pełne złączenie.

Jak to działa

Podzapytanie nieskorelowane — takie, które nie odwołuje się do niczego z zapytania zewnętrznego, jak podzapytanie skalarne z pierwszej Fiszki — silnik może policzyć raz, niezależnie od tego, ile wierszy ma zapytanie zewnętrzne. Podzapytanie skorelowane odwołuje się do kolumny z zewnątrz, więc logicznie musi zostać przeliczone dla każdego wiersza zewnętrznego z osobna — `k.id` w `WHERE z.klient_id = k.id` zmienia się przy każdym wierszu `klienci`. W praktyce optymalizator często przepisuje takie zapytanie na złączenie pod spodem, żeby uniknąć dosłownego przeliczania podzapytania w pętli — ale logicznie, z punktu widzenia autora zapytania, warto myśleć o nim właśnie jako o powtarzanym dla każdego wiersza.

Częste błędy

NOT IN z podzapytaniem, które może zwrócić NULL. To jest jedna z najbardziej podstępnych pułapek w całym SQL, bo nie zgłasza żadnego błędu — po prostu cicho zwraca pusty wynik tam, gdzie ktoś spodziewał się konkretnych wierszy.

```
SELECT nazwa FROM klienci
WHERE id NOT IN (SELECT klient_id FROM zamowienia);
```

Zamierzeniem jest znalezienie klientów bez żadnego zamówienia — powinien to być Elektro-System Kowalczyk, jedyny klient bez wpisu w `zamowienia`. Ale zamówienie 106 ma `klient_id IS NULL`, więc podzapytanie zwraca listę zawierającą `NULL` obok liczb.

Przykład:

```
 id | nazwa | miasto | nip
----+-------+--------+-----
(0 rows)
```

Zero wierszy, nie jeden. Powód leży w trójwartościowej logice ze strony NULL: trzecia wartość logiczna i pułapki, które z niej wynikają: `x NOT IN (a, b, NULL)` rozwija się logicznie do `x <> a AND x <> b AND x <> NULL` — a `x <> NULL` zawsze daje `UNKNOWN`, niezależnie od `x`. `UNKNOWN` w `AND` z czymkolwiek innym niż `FALSE` też daje `UNKNOWN`, więc cały warunek dla każdego wiersza staje się `UNKNOWN`, a `WHERE` odrzuca `UNKNOWN` tak samo jak `FALSE`. Efekt: żaden wiersz nigdy nie przejdzie tego `NOT IN`, dopóki podzapytanie może zwrócić choć jeden `NULL`.

`NOT EXISTS` nie ma tego problemu, bo nie porównuje wartości między sobą — sprawdza tylko istnienie dopasowania:

```
SELECT nazwa FROM klienci k
WHERE NOT EXISTS (
    SELECT 1 FROM zamowienia z WHERE z.klient_id = k.id
```

```
);
```

Przykład:

```
 id |          nazwa           | miasto |    nip
----+--------------------------+--------+------------
  6 | Elektro-System Kowalczyk | Poznań | 7123456780
(1 row)
```

Poprawny wynik, bo `z.klient_id = k.id` po prostu nigdy nie jest prawdą dla wiersza z `NULL`-em w `z.klient_id` — nie tworzy to żadnego `UNKNOWN`, który mógłby zatruć cały warunek zewnętrzny. Praktyczna reguła: gdy kolumna użyta w podzapytaniu do `NOT IN` może zawierać `NULL`, bezpieczniejszy jest `NOT EXISTS` — albo jawne `WHERE kolumna IS NOT NULL` wewnątrz podzapytania, jeśli `NOT IN` ma zostać z jakiegoś powodu.

Podzapytanie skalarne zwracające więcej niż jeden wiersz. `WHERE wartosc = (SELECT wartosc FROM zamowienia WHERE klient_id = 2)` wybucha w chwili wykonania, jeśli klient 2 ma więcej niż jedno zamówienie — a ma, dwa.

```
ERROR:  more than one row returned by a subquery used as an expression
```

Silnik pozwala sobie na taki błąd dopiero w czasie wykonania, nie składni, bo z samej treści zapytania nie widać z góry, ile wierszy podzapytanie faktycznie zwróci.

□□ Ćwiczenia

> Zadania sprawdzające.

Ćwiczenie 1 — NOT IN kontra NOT EXISTS

Kolumna `zamowienia.klient_id` zawiera jeden `NULL`. Bez uruchamiania zapytania przewidujemy, czy `SELECT nazwa FROM klienci WHERE id NOT IN (SELECT klient_id FROM zamowienia WHERE klient_id IS NOT NULL);` zwróci klienta bez zamówień.

Odpowiedź

Tak, zwróci — bo dodanie `WHERE klient_id IS NOT NULL` wewnątrz podzapytania usuwa `NULL` z listy, zanim `NOT IN` w ogóle ją zobaczy. To drugi, równie poprawny sposób obejścia pułapki `NOT IN`

, obok zamiany na `NOT EXISTS` — działa, dopóki ktoś pamięta, żeby ten warunek dopisać.

Pytania kontrolne

1. Dlaczego podzapytanie z pierwszej Fiszki (średnia wartość zamówień) można policzyć raz, a podzapytanie z EXISTS trzeba logicznie liczyć dla każdego wiersza z osobna?

Odpowiedź

Bo podzapytanie skalarne z `AVG(wartosc)` nie odwołuje się do niczego z zapytania zewnętrznego — jego wynik jest taki sam niezależnie od tego, który wiersz `zamowienia` akurat jest sprawdzany. Podzapytanie w `EXISTS` odwołuje się do `k.id` z zapytania zewnętrznego, więc jego wynik zależy od tego, który konkretnie wiersz `klienci` jest właśnie rozpatrywany — to jest definicja podzapytania skorelowanego.

2. Kiedy warto sięgnąć po podzapytanie zamiast po JOIN, mimo że oba potrafią odpowiedzieć na podobne pytanie?

Odpowiedź

Gdy pytanie sprowadza się do „czy istnieje" albo do filtra opartego na agregacie z drugiej tabeli, a nie do potrzeby wyciągnięcia z niej dodatkowych kolumn do wyniku. `EXISTS` i `NOT EXISTS` bywają wtedy czytelniejsze niż złączenie z `GROUP BY`, i pozwalają silnikowi przerwać sprawdzanie przy pierwszym dopasowaniu zamiast liczyć pełne złączenie wszystkich wierszy.

Powiązane: CROSS JOIN i SELF JOIN · INTERSECT i EXCEPT / MINUS · NULL: trzecia wartość logiczna i pułapki, które z niej wynikają · CTE: WITH i zapytania rekurencyjne

Źródła: ISO/IEC 9075-2:2016 § 7.15 (subquery) · dokumentacja PostgreSQL, rozdział „Subquery Expressions"

Revision #47
Created 2026-09-24 22:42:06 UTC by Tomasz Konieczny
Updated 2026-10-26 14:11:09 UTC by Tomasz Konieczny