SQL: anatomia zapytania

Skąd wziął się SQL, jak naprawdę wykonuje się SELECT, czym filtruje WHERE, co robi NULL i jak grupujemy wyniki.

- Skąd się wziął SQL i dlaczego każdy silnik mówi nim inaczej
- SELECT: kolejność zapisu a kolejność wykonania
- WHERE: filtrowanie wierszy
- NULL: trzecia wartość logiczna i pułapki, które z niej wynikają
- ORDER BY, LIMIT, TOP, ROWNUM, FETCH: porcjowanie wyniku
- Funkcje agregujące i GROUP BY
- HAVING: dlaczego nie da się tego zrobić w WHERE
- DISTINCT: i dlaczego zwykle jest objawem, nie rozwiązaniem

Skąd się wziął SQL i dlaczego każdy silnik mówi nim inaczej

> SQL ma jeden standard i osiem dialektów, a ta strona tłumaczy, skąd się wzięła ta rozbieżność i jak rozpoznać, które polecenie należy do której z czterech grup języka.

Skróty użyte na tej stronie (10)

Skrót	Co znaczy
ACM	AWS Certificate Manager: usługa wydająca i odnawiająca certyfikaty TLS dla równoważników, CloudFront i API Gateway
ANSI	American National Standards Institute: tu chodzi o kody sterujące kolorem i pozycją kursora w terminalu
DDL	Data Definition Language: polecenia SQL zmieniające strukturę (CREATE, ALTER, DROP)
DML	Data Manipulation Language: polecenia SQL zmieniające dane (INSERT, UPDATE, DELETE)
IBM	International Business Machines: firma, w której laboratorium powstał model relacyjny i pierwszy działający silnik SQL
ISO	International Organization for Standardization: organizacja, która wydaje normy techniczne używane na całym świecie
JSON	JavaScript Object Notation: tekstowy format danych z klamrami i cudzysłowami
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
XML	eXtensible Markup Language: format tekstowy do zapisu ustrukturyzowanych danych w znacznikach, tutaj używany przez PowerShell do zapisu zrzutu obiektów poleceniem Export-Clixml

Fiszka

□□ ZMIENIA SYSTEM

Rozpoznanie polecenia definiującego strukturę: DDL

```
CREATE TABLE test_ddl (id integer);
```

Polecenie tworzy, zmienia albo usuwa obiekt w bazie — tabelę, indeks, widok, schemat. Silnik odpowiada komunikatem o wykonanej operacji, nie zestawem wierszy.

Przykład:

```
CREATE TABLE
```

□□ **DDL** = Data Definition Language: polecenia SQL zmieniające strukturę (CREATE, ALTER, DROP)

□□ TYLKO ODCZYT

Rozpoznanie polecenia operującego na danych: DML

```
SELECT nazwa, miasto FROM klienci WHERE miasto = 'Warszawa';
```

Polecenie czyta albo zmienia zawartość tabel, nie ich strukturę. `SELECT`, `INSERT`, `UPDATE` i `DELETE` należą do tej samej grupy, mimo że tylko jedno z nich zwraca wiersze.

Przykład:

```
   nazwa                | miasto
------------------------+-----------
 Nowak i Wspólnicy Sp. z o.o. | Warszawa
(1 row)
```

□□ **DML** = Data Manipulation Language: polecenia SQL zmieniające dane (INSERT, UPDATE, DELETE)

□□ ZMIENIA SYSTEM

Rozpoznanie polecenia kontroli dostępu: DCL

```
GRANT SELECT ON zamowienia TO nazwa_usera;
```

Polecenie nadaje albo odbiera uprawnienia do obiektów bazy. Nie rusza ani struktury, ani danych — rusza wyłącznie to, kto ma do nich dostęp.

Przykład:

```
GRANT
```

□□ TYLKO ODCZYT

Rozpoznanie polecenia kontroli transakcji: TCL

```
COMMIT;
```

Polecenie zamyka albo cofa grupę zmian zebranych w jednej transakcji. `COMMIT`, `ROLLBACK` i `SAVEPOINT` do tej strony wracają jeszcze w rozdziale o transakcjach — tu liczy się tylko to, że stoją osobno od DML.

Przykład:

```
COMMIT
```

□□ **DML** = Data Manipulation Language: polecenia SQL zmieniające dane (INSERT, UPDATE, DELETE)

□□ OSTROŻNIE · NIEODWRACALNE

Rozpoznanie polecenia na granicy dwóch grup: TRUNCATE

```
TRUNCATE TABLE test_ddl;
```

Polecenie wygląda jak operacja na danych — usuwa wszystkie wiersze z tabeli, tak jak `DELETE` bez `WHERE`. Formalnie jednak większość silników traktuje je jako DDL: nie loguje usunięcia wiersz po wierszu, tylko zwalnia zajmowane przez tabelę strony danych naraz, przez co działa dużo szybciej niż `DELETE`, ale w niektórych silnikach nie da się go cofnąć poleceniem `ROLLBACK` wewnątrz otwartej transakcji.

Przykład:

```
TRUNCATE TABLE
```

□□ **DDL** = Data Definition Language: polecenia SQL zmieniające strukturę (CREATE, ALTER, DROP)

□□ TYLKO ODCZYT

Wszystkie grupy poleceń

Grupa	Rozwinięcie	Przykładowe polecenia
DDL	Data Definition Language	`CREATE`, `ALTER`, `DROP`, `TRUNCATE`
DML	Data Manipulation Language	`SELECT`, `INSERT`, `UPDATE`, `DELETE`, `MERGE`
DCL	Data Control Language	`GRANT`, `REVOKE`
TCL	Transaction Control Language	`COMMIT`, `ROLLBACK`, `SAVEPOINT`

> **Uważaj**
>
> - `TRUNCATE` wygląda jak operacja na danych, ale formalnie jest DDL — część silników nie da się jej cofnąć poleceniem `ROLLBACK`.
> - Różne książki zaliczają `MERGE` raz do DML, raz do osobnej kategorii — w praktyce liczy się to, że modyfikuje dane, nie strukturę.

DDL = Data Definition Language: polecenia SQL zmieniające strukturę (CREATE, ALTER, DROP) · **DML** = Data Manipulation Language: polecenia SQL zmieniające dane (INSERT, UPDATE, DELETE)

Po co to istnieje

> **Pojęcie: SQL.** Structured Query Language — deklaratywny język do opisywania, jakie dane mają się znaleźć w wyniku, oraz do definiowania i kontrolowania struktury, w której te dane leżą. Deklaratywny znaczy tyle, że zapytanie opisuje pożądany wynik, a nie kroki prowadzące do jego policzenia — do tego wracamy w SELECT: kolejność zapisu a kolejność wykonania.

Historia zaczyna się w 1970 roku, kiedy Edgar F. Codd, pracownik IBM, opublikował pracę opisującą model relacyjny — dane jako zbiory wierszy w tabelach powiązane przez wartości, a nie przez wskaźniki i fizyczne ścieżki dostępu, jak w ówczesnych bazach hierarchicznych i sieciowych. Ówczesny standard przemysłowy, opracowany przez organizację CODASYL, każdy sposób dotarcia do rekordu zapisywał wprost w strukturze bazy — program pytający o dane musiał znać fizyczną drogę do nich, wskaźnik po wskaźniku, rekord po rekordzie. Zmiana tej struktury oznaczała przepisanie każdego programu, który z niej korzystał. Model Codda odrywał sposób pytania o dane od sposobu ich fizycznego przechowywania — stąd zresztą wynika deklaratywność SQL, wyjaśniona niżej.

IBM uruchomił projekt badawczy System R, żeby sprawdzić, czy taki model da się w ogóle sensownie zaimplementować. Do zapytań do tego systemu Donald Chamberlin i Raymond Boyce zaprojektowali w 1974 roku język SEQUEL — Structured English Query Language, opisany w publikacji „SEQUEL: A Structured English Query Language" na konferencji ACM SIGMOD.

Nazwa SEQUEL nie przetrwała z powodów, które z bazami danych nie mają nic wspólnego: skrót był już zastrzeżony jako znak towarowy przez brytyjską firmę lotniczą Hawker Siddeley. IBM zmienił nazwę na SQL, zachowując wymowę zbliżoną do oryginału — stąd w wielu środowiskach nadal słyszy się „sequel" zamiast litera po literze.

System R był projektem badawczym i nie trafił do sprzedaży. Pierwszą komercyjną bazą danych z językiem SQL nie był produkt IBM, tylko Oracle w wersji 2, wydany w 1979 roku przez firmę Relational Software Inc. — cztery lata przed tym, jak sam IBM wypuścił DB2. Firma, która wymyśliła model relacyjny i pierwszy dialekt SQL, przegrała wyścig na rynek z konkurentem czytającym te same publikacje badawcze.

Równolegle z System R powstawał na Uniwersytecie Kalifornijskim w Berkeley konkurencyjny projekt badawczy, Ingres, z własnym językiem zapytań QUEL. Część badaczy tamtych lat uważała QUEL za czystszy składniowo niż SQL. Rynek zdecydował inaczej — to wokół SQL zbudowały swoje produkty Oracle i sam IBM, więc to SQL stał się w latach osiemdziesiątych faktycznym standardem branżowym, zanim jeszcze doczekał się standardu formalnego. Zwycięstwo SQL nad QUEL bywa przywoływane jako wczesny przykład tego, że o dominującym standardzie decyduje siła rynkowa produktów, które go używają, nie techniczna elegancja samego języka.

> **Pojęcie: standaryzacja.** Proces, w którym niezależna organizacja spisuje wspólną definicję języka, żeby polecenia napisane pod jeden silnik dało się — przynajmniej w teorii — uruchomić na innym. American National Standards Institute opublikował pierwszy standard SQL w 1986 roku, znany jako SQL-86 albo SQL-1. International Organization for Standardization przejęła standard rok później. Od tamtej pory standard był rewidowany regularnie: SQL-89 (drobne poprawki), SQL-92 (duże rozszerzenie, do dziś punkt odniesienia dla „podstawowej zgodności"), SQL:1999 (zapytania rekurencyjne, wyzwalacze, typy obiektowe), SQL:2003 (funkcje okna, kolumny generowane, XML), SQL:2006, SQL:2008, SQL:2011 (tabele czasowe), SQL:2016 (JSON, dopasowywanie wzorców w wierszach), SQL:2019 (dane wielowymiarowe) i SQL:2023 (typy JSON natywne, `GREATEST` / `LEAST` w standardzie).

Kolejne wydania standardu w skrócie:

Rok	Nazwa	Kluczowe dodatki
1986	SQL-86 / SQL-1	pierwszy standard, przyjęty przez ANSI
1989	SQL-89	integralność referencyjna
1992	SQL-92	duże rozszerzenie, do dziś punkt odniesienia zgodności

Rok	Nazwa	Kluczowe dodatki
1999	SQL:1999	zapytania rekurencyjne (`WITH RECURSIVE`), wyzwalacze, typy obiektowe
2003	SQL:2003	funkcje okna, kolumny generowane, integracja z XML
2006	SQL:2006	dalsza integracja z XML
2008	SQL:2008	`TRUNCATE` w standardzie, `INSTEAD OF` triggers
2011	SQL:2011	tabele czasowe (temporal)
2016	SQL:2016	JSON, dopasowywanie wzorców w wierszach
2019	SQL:2019	dane wielowymiarowe (MDA)
2023	SQL:2023	natywny typ JSON, `GREATEST`/`LEAST`

Standard nie jest jednym monolitem, który silnik albo wspiera, albo nie. Dzieli się na Core SQL — minimalny zestaw funkcji, bez którego producent nie ma prawa nazwać swojego produktu zgodnym z danym wydaniem standardu — oraz dziesiątki opcjonalnych pakietów funkcji, z których każdy silnik implementuje inny podzbiór. Dlatego pytanie „czy silnik jest zgodny z SQL:2016” nie ma sensownej odpowiedzi tak/nie — sensowne jest tylko pytanie o zgodność z konkretną, nazwaną funkcją standardu, na przykład z oknami analitycznymi z SQL:2003.

Jak to działa

Rozwiń: jak to działa pod spodem

Standard opisuje rdzeń języka — składnię, którą każdy silnik zobowiązuje się rozumieć podobnie. Żaden producent nie implementuje standardu w całości i żaden nie poprzestaje na standardzie. Powód jest ekonomiczny: silnik, który różni się tylko szybkością od konkurencji, trudno sprzedać, więc każdy producent dokłada własne funkcje, własne typy danych i własną składnię do rzeczy, których standard nie przewidział w momencie powstawania silnika — bo standard bywa młodszy od funkcji, którą opisuje, albo starszy o dekadę od potrzeby, która się dopiero pojawiła.

> **Pojęcie: dialekt.** Wariant SQL właściwy dla konkretnego silnika — nadzbiór rdzenia standardu rozszerzony o funkcje, składnię i zachowania, których nie ma nigdzie indziej. T-SQL w SQL Serverze, PL/pgSQL w PostgreSQL i PL/SQL w Oracle to trzy różne dialekty,

> a każdy z nich zawiera więcej niż sam standard SQL.

Rozbieżność widać najwyraźniej tam, gdzie standard w ogóle nie narzucił jednego rozwiązania albo narzucił je już po tym, jak silniki zdążyły wypracować własne.

Operacja	PostgreSQL	MySQL / MariaDB	SQL Server (T-SQL)	Oracle
Ograniczenie liczby wierszy	`LIMIT 5`	`LIMIT 5`	`SELECT TOP 5 ...`	`FETCH FIRST 5 ROWS ONLY` (12c+)
Konkatenacja tekstu	`||`	`CONCAT(a, b)`	`+`	`||`
Bieżąca data i czas	`now()` / `CURRENT_TIMESTAMP`	`NOW()`	`GETDATE()`	`SYSDATE`
Rzutowanie typu	`x::integer` / `CAST(x AS integer)`	`CAST(x AS SIGNED)`	`CAST(x AS INT)`	`CAST(x AS NUMBER)`
Sprawdzenie wersji silnika	`SELECT version();`	`SELECT VERSION();`	`SELECT @@VERSION;`	`SELECT * FROM v$version;`

Nawet coś tak podstawowego jak sprawdzenie, na jakiej wersji silnika w ogóle pracujemy, nie ma wspólnej składni — w T-SQL to zmienna systemowa poprzedzona podwójnym `@@`, w Oracle to zapytanie do widoku systemowego zwracającego kilka wierszy zamiast jednego tekstu.

Rozdział ORDER BY, LIMIT, TOP, ROWNUM, FETCH: porcjowanie wyniku wraca do pierwszego wiersza tej tabeli ze szczegółami i przypadkami brzegowymi.

Ta sama rozbieżność dotyczy sposobu cytowania identyfikatorów — nazw tabel i kolumn, które kolidują ze słowem kluczowym albo zawierają wielkie litery czy spacje.

Silnik	Cytowanie identyfikatora
PostgreSQL	`"NazwaKolumny"`
MySQL / MariaDB	`` `NazwaKolumny` ``
SQL Server	`[NazwaKolumny]`
Oracle	`"NazwaKolumny"`

PostgreSQL i Oracle zgadzają się co do znaku cudzysłowu, bo obaj trzymają się tu litery standardu — SQL Server i MySQL poszły własną drogą, każdy w inną stronę.

Częste błędy

Traktowanie ANSI SQL jak języka w pełni przenośnego. Zapytanie napisane i przetestowane na jednym silniku po przeniesieniu na drugi potrafi nie zadziałać wcale, zadziałać z innym wynikiem albo zadziałać wolniej — konkatenacja przez `+` w T-SQL po przeniesieniu do PostgreSQL nie jest błędem składniowym, tylko operatorem dodawania, który dla dwóch tekstów zgłasza błąd typu.

Zakładanie, że NULL zachowuje się wszędzie tak samo. Oracle traktuje pusty ciąg znaków w kolumnie `VARCHAR2` jako `NULL`, podczas gdy PostgreSQL i MySQL rozróżniają pusty ciąg od braku wartości. Strona NULL: trzecia wartość logiczna i pułapki, które z niej wynikają opisuje tę pułapkę osobno.

Kopiowanie cudzysłowu identyfikatorów bez zmiany. PostgreSQL cytuje identyfikatory podwójnym cudzysłowem, MySQL domyślnie akceptuje odwrócony apostrof, a SQL Server nawiasy kwadratowe — zapytanie z `[NazwaTabeli]` przeniesione żywcem do PostgreSQL kończy się błędem składni, nie ostrzeżeniem.

Zakładanie identycznych nazw funkcji wbudowanych. Funkcja robiąca dokładnie to samo bywa dostępna pod inną nazwą w każdym silniku: wycięcie fragmentu tekstu to `SUBSTR` w Oracle i MySQL, `SUBSTRING` w SQL Server, a PostgreSQL rozumie akurat obie formy. Kod migrowany między silnikami trzeba przejrzeć pod kątem takich drobnych rozbieżności nazewniczych osobno od różnic w samej składni zapytania.

Traktowanie TRUNCATE jak bezpiecznego odpowiednika DELETE. `TRUNCATE` bywa nieodwracalny w ramach otwartej transakcji tam, gdzie `DELETE` dałoby się cofnąć — w MySQL z silnikiem InnoDB `TRUNCATE` wykonuje niejawny commit przed i po sobie, więc `ROLLBACK` po nim nie przywróci danych, nawet jeśli reszta transakcji się nie powiodła.

□□ Ćwiczenia

> Zadania sprawdzające.

Ćwiczenie 1 — rozpoznanie grupy

Dla każdego polecenia wskazujemy, czy jest to DDL, DML, DCL czy TCL: `ALTER TABLE zamowienia ADD COLUMN uwagi text;`, `REVOKE INSERT ON produkty FROM nazwa_usera;`, `DELETE FROM pozycje WHERE ilosc = 0;`, `SAVEPOINT przed_rabatem;`.

Odpowiedź

`ALTER TABLE` — DDL, zmienia strukturę tabeli. `REVOKE` — DCL, odbiera uprawnienie. `DELETE` — DML, usuwa dane, nie strukturę. `SAVEPOINT` — TCL, znacznik wewnątrz transakcji.

Ćwiczenie 2 — ta sama operacja, cztery silniki

Zapisujemy zapytanie sprawdzające bieżącą datę i pobierające pięć pierwszych wierszy z tabeli `klienci`, osobno w składni PostgreSQL i osobno w składni SQL Server.

Odpowiedź

PostgreSQL:

```
SELECT CURRENT_DATE, * FROM klienci LIMIT 5;
```

SQL Server:

```
SELECT TOP (5) GETDATE() AS CURRENT_DATE, * FROM klienci;
```

Obie wersje realizują to samo zadanie, ale różnią się miejscem, w którym stoi ograniczenie liczby wierszy, i nazwą funkcji zwracającej bieżącą datę — dokładnie tak, jak pokazuje tabela różnic w sekcji „Jak to działa”.

Pytania kontrolne

1. Dlaczego pierwszą komercyjną bazą z SQL nie był produkt IBM, mimo że to IBM sfinansował badania nad modelem relacyjnym i językiem SEQUEL?

Odpowiedź

Projekt System R był wewnętrznym projektem badawczym IBM, nie produktem. Firma Relational Software Inc. (późniejszy Oracle) przeczytała te same publikacje i wypuściła własną komercyjną implementację w 1979 roku, cztery lata przed komercyjnym DB2 od IBM.

2. Skąd wzięła się nazwa SQL, skoro pierwotny język nazywał się SEQUEL?

Odpowiedź

Nazwa SEQUEL kolidowała ze znakiem towarowym zastrzeżonym przez brytyjską firmę lotniczą Hawker Siddeley, więc IBM zmienił nazwę na SQL, zachowując zbliżoną wymowę.

3. Dlaczego zapytanie zgodne ze standardem SQL:2016 nie musi zadziałać identycznie na dwóch różnych silnikach?

Odpowiedź

Standard opisuje rdzeń języka, ale żaden silnik nie implementuje go w całości ani nie poprzestaje wyłącznie na nim — każdy dokłada własny dialekt z rozszerzeniami, których nie ma standard ani konkurencja.

4. Dlaczego TRUNCATE zalicza się do DDL, mimo że z punktu widzenia efektu — puste wiersze w tabeli — wygląda jak DELETE?

Odpowiedź

Bo mechanizm, jakim TRUNCATE osiąga ten efekt, jest inny niż w DELETE — nie usuwa wierszy pojedynczo z logowaniem każdej operacji, tylko zwalnia całe strony danych naraz, podobnie jak DROP zwalnia całą tabelę. To upodabnia go do operacji na strukturze, nie na pojedynczych rekordach, i tłumaczy, dlaczego w wielu silnikach nie da się go cofnąć tak samo łatwo jak DELETE.

5. Projekt Ingres z Berkeley i jego język QUEL przegrały z SQL na rynku, mimo że część badaczy uważała QUEL za lepiej zaprojektowany. Co z tego wynika dla oceny, dlaczego akurat SQL, a nie inny język, stał się standardem branżowym?

Odpowiedź

Wynika z tego, że o dominującym standardzie zdecydowała siła rynkowa produktów zbudowanych wokół danego języka — komercyjny Oracle i komercyjny DB2 od IBM — a nie techniczna elegancja samego języka zapytań. Standard formalny (ANSI, potem ISO) przyszedł już po tym, jak SQL faktycznie zdominował rynek, nie odwrotnie.

Powiązane: SELECT: kolejność zapisu a kolejność wykonania · NULL: trzecia wartość logiczna i pułapki, które z niej wynikają · ORDER BY, LIMIT, TOP, ROWNUM, FETCH: porcjowanie wyniku

Źródła: ISO/IEC 9075:2016 · dokumentacja PostgreSQL, MySQL, SQL Server, Oracle Database

SELECT: kolejność zapisu a kolejność wykonania

> `SELECT` opisujemy w jednej kolejności, a silnik wykonuje go w zupełnie innej — ta strona pokazuje obie kolejności obok siebie i tłumaczy, dlaczego alias z `SELECT` działa w `ORDER BY`, a w `WHERE` już nie.

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

TYLKO ODCZYT

Proste zapytanie

```
SELECT nazwa, miasto FROM klienci;
```

Zapytanie opisuje, jakie kolumny mają się znaleźć w wyniku i z jakiej tabeli pochodzą. Nic więcej — bez filtrowania, bez sortowania.

Przykład:

```
           nazwa               |  miasto
-------------------------------+-----------
 Nowak i Wspólnicy Sp. z o.o.  | Warszawa
 Kowalski Handel               | Kraków
```

```
 Zielona Dolina S.A.            | Wrocław
 Baltic Trade Sp. z o.o.        | Gdańsk
 Podlaski Handel                | Białystok
(5 rows)
```

TYLKO ODCZYT

Alias kolumny

```
SELECT nazwa AS klient, miasto AS lokalizacja FROM klienci;
```

Słowo kluczowe `AS` nadaje kolumnie wyniku inną nazwę niż nazwa kolumny źródłowej. `AS` jest opcjonalne w większości silników — `nazwa klient` bez niego działa tak samo — ale zostawiamy je, bo odróżnia alias od literówki w liście kolumn.

Przykład:

```
          klient                | lokalizacja
--------------------------------+-------------
 Nowak i Wspólnicy Sp. z o.o.   | Warszawa
 Kowalski Handel                | Kraków
 Zielona Dolina S.A.            | Wrocław
 Baltic Trade Sp. z o.o.        | Gdańsk
 Podlaski Handel                | Białystok
(5 rows)
```

TYLKO ODCZYT

Alias tabeli

```
SELECT k.nazwa, k.miasto FROM klienci AS k WHERE k.miasto = 'Warszawa';
```

Alias tabeli zastępuje jej pełną nazwę skróconym prefiksem przy każdym odwołaniu do kolumny. Przy jednej tabeli to wygoda, przy kilku połączonych złączeniem — jedyny sposób, żeby wskazać, z której tabeli pochodzi kolumna o niejednoznacznej nazwie. Rozdział Czym jest złączenie: iloczyn kartezjański i warunek, który go ogranicza wraca do tego, gdy w grze jest więcej niż jedna tabela.

Przykład:

```
           nazwa              |  miasto
------------------------------+-----------
 Nowak i Wspólnicy Sp. z o.o.  | Warszawa
(1 row)
```

TYLKO ODCZYT

Wszystkie kolumny: SELECT *

```
SELECT * FROM produkty;
```

Gwiazdka rozwija się w pełną listę kolumn tabeli, w kolejności, w jakiej są zdefiniowane w katalogu systemowym. Wygodne przy eksploracji danych z konsoli, ryzykowne w kodzie aplikacji — dodanie kolumny do tabeli zmienia liczbę i kolejność kolumn zwracanych do programu, który mógł się spodziewać stałego kształtu wyniku.

Przykład:

```
 id |         nazwa           |  kategoria  |  cena
----+-------------------------+-------------+---------
  1 | Monitor 24 cale         | elektronika |  650.00
  2 | Klawiatura mechaniczna  | elektronika |  220.00
  3 | Biurko regulowane       | meble       | 1100.00
  4 | Fotel biurowy           | meble       |  890.00
  5 | Kabel HDMI 2m           | elektronika |   25.00
  6 | Lampka biurkowa         | meble       |   90.00
(6 rows)
```

TYLKO ODCZYT

Wyrażenie w SELECT

```
SELECT nazwa, cena, cena * 1.23 AS cena_brutto FROM produkty WHERE kategoria = 'meble';
```

Lista w `SELECT` nie musi się ograniczać do nazw kolumn — może zawierać dowolne wyrażenie policzone dla każdego wiersza osobno: działanie arytmetyczne, sklejenie tekstu, wywołanie funkcji. Wynikowa kolumna nie istnieje w tabeli, istnieje wyłącznie w tym jednym wyniku.

Przykład:

```
        nazwa        |  cena   | cena_brutto
---------------------+---------+-------------
 Biurko regulowane   | 1100.00 |     1353.00
 Fotel biurowy       |  890.00 |     1094.70
 Lampka biurkowa     |   90.00 |      110.70
(3 rows)
```

Po co to istnieje

> **Pojęcie: zapytanie deklaratywne.** `SELECT` nie jest przepisem krok po kroku na to, jak policzyć wynik — jest opisem tego, jak wynik ma wyglądać. Piszemy, jakie kolumny, z jakich wierszy i w jakim porządku chcemy zobaczyć, a to, w jakiej kolejności silnik faktycznie przejdzie po danych, którego indeksu użyje i czy posortuje przed czy po filtrowaniu, zostaje jego decyzją — decyzją, którą da się podejrzeć w planie wykonania, opisanym na stronie Plan wykonania: jak go wywołać i jak go czytać. Z punktu widzenia autora zapytania liczy się wynik, nie droga do niego.

Ta różnica między opisem a przepisem tłumaczy, dlaczego kolejność, w jakiej piszemy klauzule `SELECT`, nie jest kolejnością, w jakiej silnik je liczy.

Jak to działa

Rozwiń: jak to działa pod spodem

> **Pojęcie: logiczna kolejność wykonania.** Ustalona przez standard SQL kolejność, w jakiej silnik logicznie przetwarza klauzule zapytania — niezależnie od tego, w jakiej kolejności fizycznie odczytuje dane z dysku czy z pamięci. „Logicznie” znaczy, że to model do rozumowania o zapytaniu, a nie dosłowny opis pracy silnika: optymalizator wolno mu zmienić kolejność operacji, o ile wynik końcowy wyjdzie taki, jakby trzymał się modelu.

Kolejność zapisu klauzul w zapytaniu:

```
SELECT ...
FROM ...
WHERE ...
GROUP BY ...
HAVING ...
ORDER BY ...
LIMIT ...
```

Logiczna kolejność wykonania jest inna:

1. `FROM` — silnik ustala źródłowy zbiór wierszy.
2. `WHERE` — z tego zbioru odsiewa wiersze niespełniające warunku.
3. `GROUP BY` — zwija pozostałe wiersze w grupy.
4. `HAVING` — odsiewa grupy niespełniające warunku na agregacie.
5. `SELECT` — dla tego, co zostało, liczy listę kolumn i wyrażeń wyniku.
6. `DISTINCT` — usuwa zduplikowane wiersze wyniku, jeśli klauzula jest obecna.
7. `ORDER BY` — sortuje to, co zostało.
8. `LIMIT` / `OFFSET` — obcina wynik do żądanej liczby wierszy.

Weźmy zapytanie korzystające z większości tych kroków:

```
SELECT klient_id, SUM(wartosc) AS suma
FROM zamowienia
WHERE status = 'zrealizowane'
GROUP BY klient_id
HAVING SUM(wartosc) > 1000
ORDER BY suma DESC
LIMIT 5;
```

Przykład:

```
 klient_id |  suma
-----------+---------
         4 | 3200.00
         1 | 1680.50
(2 rows)
```

Silnik najpierw bierze całą tabelę `zamowienia` (krok 1), odrzuca wiersze ze statusem innym niż `zrealizowane` (krok 2), zwija pozostałe w grupy po `klient_id` (krok 3), odrzuca grupy, których suma nie przekracza 1000 (krok 4), dopiero teraz liczy kolumny wyniku łącznie z aliasem `suma`

(krok 5), sortuje po tym aliasie (krok 7) i obcina do pięciu wierszy (krok 8) — w tym przypadku i tak zostały tylko dwa.

Z tej kolejności wynika konkretna, praktyczna konsekwencja: alias zdefiniowany w `SELECT` nie istnieje jeszcze na etapie `WHERE`, bo `WHERE` wykonuje się w kroku 2, a `SELECT` dopiero w kroku 5.

```
SELECT klient_id, SUM(wartosc) AS suma
FROM zamowienia
WHERE suma > 1000
GROUP BY klient_id;
```

Przykład:

```
ERROR:  column "suma" does not exist
LINE 3: WHERE suma > 1000
              ^
```

Ten sam alias w `ORDER BY` działa bez zarzutu, bo `ORDER BY` wykonuje się w kroku 7, długo po tym, jak `SELECT` w kroku 5 policzył `suma` i nadał jej nazwę:

```
SELECT klient_id, SUM(wartosc) AS suma
FROM zamowienia
GROUP BY klient_id
ORDER BY suma DESC;
```

Filtr, który logicznie dotyczy agregatu, trafia zamiast tego do `HAVING` — strona HAVING: dlaczego nie da się tego zrobić w WHERE tłumaczy to dokładniej. Warunek dotyczący pojedynczego wiersza, zanim jeszcze trafi do grupowania, zawsze warto trzymać w `WHERE` — silnik odsieje go najwcześniej, jak to możliwe, i będzie miał mniej danych do zgrupowania.

Częste błędy

Filtrowanie po aliasie w WHERE. Opisane wyżej `column "suma" does not exist` — pierwszy odruch to poprawka na `HAVING SUM(wartosc) > 1000`, drugi to powtórzenie całego wyrażenia w WHERE`, jeśli warunek faktycznie dotyczy pojedynczego wiersza, a nie grupy.

Założenie, że kolejność zapisu klauzul jest dowolna. `WHERE` musi stać przed `GROUP BY`, a `GROUP BY` przed `HAVING` — to nie jest kwestia stylu, tylko wymóg składni, zgodny zresztą z logiczną kolejnością wykonania. Zapisanie ich w innej kolejności kończy się błędem składniowym, nie ostrzeżeniem.

Poleganie na kolejności zwracanej przez SELECT * bez ORDER BY. Bez jawnego `ORDER BY` kolejność wierszy w wyniku jest nieokreślona, nawet jeśli w praktyce wygląda stabilnie przez miesiące — strona ORDER BY, LIMIT, TOP, ROWNUM, FETCH: porcjowanie wyniku tłumaczy, dlaczego.

□□ Ćwiczenia

> Zadania sprawdzające.

Ćwiczenie 1 — kolejność wykonania

Dla zapytania `SELECT status, COUNT(*) AS ile FROM zamowienia WHERE klient_id IS NOT NULL GROUP BY status HAVING COUNT(*) > 1 ORDER BY ile DESC;` wypisujemy numerami kolejność, w jakiej silnik logicznie wykona poszczególne klauzule.

Odpowiedź

1. `FROM zamowienia`
2. `WHERE klient_id IS NOT NULL`
3. `GROUP BY status`
4. `HAVING COUNT(*) > 1`
5. `SELECT status, COUNT(*) AS ile`
6. `ORDER BY ile DESC`

`LIMIT` i `DISTINCT` nie występują w tym zapytaniu, więc kroki 6 i 8 z pełnej listy są pomijane.

Pytania kontrolne

1. Dlaczego `ORDER BY suma` działa, a `WHERE suma > 1000` w tym samym zapytaniu zgłasza błąd, skoro `suma` to alias zdefiniowany w tym samym SELECT?

Odpowiedź

Bo `WHERE` wykonuje się logicznie przed `SELECT`, więc alias jeszcze nie istnieje, a `ORDER BY` wykonuje się po `SELECT`, więc alias jest już policzony i dostępny.

2. Czy `SELECT *` gwarantuje tę samą kolejność kolumn w każdym wywołaniu tego samego zapytania na tej samej tabeli?

Odpowiedź

Tak, dopóki nikt nie zmieni struktury tabeli. Kolejność odpowiada kolejności kolumn w katalogu systemowym, ale `ALTER TABLE ... ADD COLUMN` dopisuje nową kolumnę na końcu, więc kod polegający na stałej liczbie i kolejności kolumn z `SELECT *` jest kruchy na zmiany struktury.

3. Zapytanie `SELECT nazwa AS n FROM klienci WHERE n = 'Kowalski Handel';` kończy się błędem. Dlaczego, skoro to samo zapytanie z `ORDER BY n` zamiast `WHERE n` działa bez zarzutu?

Odpowiedź

Alias `n` powstaje dopiero w kroku piątym logicznej kolejności wykonania, razem z resztą listy `SELECT`. `WHERE` wykonuje się w kroku drugim, więc `n` jeszcze nie istnieje i silnik zgłasza błąd „column n does not exist”. `ORDER BY` wykonuje się w kroku siódmym, długo po tym, jak `SELECT` policzył i nazwał `n`, więc w tym miejscu alias jest już dostępny.

Powiązane: WHERE: filtrowanie wierszy · HAVING: dlaczego nie da się tego zrobić w WHERE · Funkcje agregujące i GROUP BY · Plan wykonania: jak go wywołać i jak go czytać

Źródła: ISO/IEC 9075-2:2016 § 7.16 · dokumentacja PostgreSQL, rozdział „The Statistics Collector” i „Select Lists”

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

NULL: trzecia wartość logiczna i pułapki, które z niej wynikają

> `NULL` nie jest zerem ani pustym tekstem, tylko brakiem informacji, a to jedno zdanie tłumaczy większość zaskoczeń, jakie `NULL` sprawia w warunkach, agregatach i sortowaniu.

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

TYLKO ODCZYT

Test na brak wartości: IS NULL

```
SELECT id, status FROM zamowienia WHERE klient_id IS NULL;
```

`IS NULL` to jedyny poprawny sposób sprawdzenia, czy kolumna nie ma wartości. Zamówienie 106 zostało zarejestrowane bez przypisanego klienta — na przykład dlatego, że konto klienta zostało już usunięte z systemu.

Przykład:

```
 id | status
----+--------
106 | nowe
(1 row)
```

TYLKO ODCZYT

Test na obecność wartości: IS NOT NULL

```
SELECT id, klient_id FROM zamowienia WHERE klient_id IS NOT NULL ORDER BY id;
```

Odwrotność poprzedniego testu — zostają wiersze, w których kolumna ma jakąkolwiek wartość, niezależnie jaką.

Przykład:

```
 id | klient_id
----+-----------
101 |         1
102 |         1
103 |         2
104 |         3
105 |         2
107 |         4
108 |         5
(7 rows)
```

TYLKO ODCZYT

Podstawienie wartości domyślnej: COALESCE

```
SELECT id, COALESCE(klient_id, 0) AS klient_id_lub_zero FROM zamowienia ORDER BY id;
```

`COALESCE` przyjmuje dowolną liczbę argumentów i zwraca pierwszy, który nie jest `NULL`em. Przy dwóch argumentach działa jak „ta wartość, a jak jej nie ma, to tamta” — tu zastępujemy brakującego klienta wartością `0`, żeby dalsze przetwarzanie nie musiało osobno obsługiwać `NULL`.

Przykład:

```
 id | klient_id_lub_zero
----+--------------------
101 |                  1
102 |                  1
103 |                  2
104 |                  3
105 |                  2
106 |                  0
107 |                  4
108 |                  5
(8 rows)
```

🔒 TYLKO ODCZYT

Zamiana konkretnej wartości na NULL: NULLIF

```
SELECT id, wartosc, NULLIF(wartosc, 0) AS wartosc_bez_zera
FROM zamowienia WHERE status = 'anulowane';
```

`NULLIF(a, b)` zwraca `NULL`, jeśli `a` równa się `b`, w przeciwnym razie zwraca `a`. Działanie odwrotne do `COALESCE` — zamiast podstawiać wartość zamiast `NULL`a, podstawia `NULL` zamiast konkretnej wartości. Przydaje się przed dzieleniem, żeby dzielenie przez zero zamienić na `NULL` zamiast na błąd silnika.

Przykład:

```
 id | wartosc | wartosc_bez_zera
----+---------+------------------
103 |    0.00 |
108 |    0.00 |
(2 rows)
```

🔒 TYLKO ODCZYT

Dlaczego porównanie do NULL nigdy nie jest prawdą: = NULL

```
SELECT id FROM zamowienia WHERE klient_id = NULL;
```

`=` porównuje dwie wartości i zwraca prawdę albo fałsz. `NULL` nie jest wartością — jest brakiem wartości — więc pytanie „czy ten brak informacji równa się tamtemu brakowi informacji” nie ma odpowiedzi prawda ani fałsz. Wynikiem jest trzecia możliwość, `UNKNOWN`, a `WHERE` traktuje `UNKNOWN` tak samo jak `FALSE`: wiersz odpada z wyniku. Warunek `klient_id = NULL` nie zwróci żadnego wiersza, nawet jeśli kolumna faktycznie zawiera `NULL` — i żaden silnik z tej książki nie robi tu wyjątku.

Przykład:

```
 id
----
(0 rows)
```

Po co to istnieje

Pojęcie: NULL. Znacznik braku informacji w kolumnie — nie liczba zero, nie pusty ciąg znaków, nie żadna konkretna wartość, tylko stwierdzenie „tej wartości tu nie ma albo nie jest znana”. Wiek klienta zapisany jako `0` znaczy, że klient ma zero lat. Wiek zapisany jako `NULL` znaczy, że nikt go nie podał. To rozróżnienie ma konsekwencje w każdym miejscu, gdzie `NULL` wchodzi w interakcję z logiką, arytmetyką albo porównaniem.

Pojęcie: trójwartościowa logika. System logiczny SQL, w którym każdy warunek daje jeden z trzech wyników: `TRUE`, `FALSE` albo `UNKNOWN` — ten ostatni pojawia się zawsze, gdy w porównaniu albo działaniu logicznym uczestniczy `NULL`. Klasyczna logika dwuwartościowa zna tylko prawdę i fałsz; SQL dokłada trzecią możliwość, bo baza danych musi umieć reprezentować niekompletną wiedzę, a nie tylko fakty i ich zaprzeczenia.

Tabele prawdy dla `AND`, `OR` i `NOT` z uwzględnieniem `UNKNOWN`:

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN

OR	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

Wartość	NOT
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

Wzór do zapamiętania: `FALSE` wygrywa w `AND` z czymkolwiek, `TRUE` wygrywa w `OR` z czymkolwiek, a `UNKNOWN` przegrywa z obiema skrajnościami i przetrwa tylko starcie z drugim `UNKNOWN` albo z tym, co go akurat nie przebija.

Jak to działa

Rozwiń: jak to działa pod spodem

NULL w agregatach. Funkcje agregujące inne niż `COUNT(*)` pomijają `NULL`e przy liczeniu — nie traktują ich jako zero, tylko jako brak danych do policzenia.

```
SELECT COUNT(*) AS wszystkie, COUNT(klient_id) AS z_klientem
FROM zamowienia;
```

Przykład:

```
 wszystkie | z_klientem
-----------+------------
         8 |          7
(1 row)
```

`COUNT(*)` liczy wiersze — osiem zamówień, niezależnie od tego, co mają w kolumnach. `COUNT(klient_id)` liczy tylko wiersze, w których `klient_id` nie jest `NULL`em — siedem, bo zamówienie 106 zostało pominięte. To samo rozróżnienie działa dla `SUM`, `AVG`, `MIN`, `MAX` — każda z nich liczy po wartościach, które istnieją, a nie po wszystkich wierszach. Strona Funkcje agregujące i GROUP BY rozwija to zachowanie dla pozostałych funkcji.

NULL w ORDER BY. Sortowanie musi jakoś umieścić `NULL`e w porządku, mimo że `NULL` nie da się porównać z żadną wartością. Standard SQL zostawia to decyzji silnika, więc PostgreSQL i

Oracle dają jawną kontrolę przez `NULLS FIRST` / `NULLS LAST`.

```
SELECT id, klient_id FROM zamowienia ORDER BY klient_id NULLS FIRST, id;
```

Przykład:

```
 id | klient_id
----+-----------
106 |
101 |         1
102 |         1
103 |         2
105 |         2
104 |         3
107 |         4
108 |         5
(8 rows)
```

Bez jawnego `NULLS FIRST` PostgreSQL domyślnie stawia `NULL`e na końcu przy sortowaniu rosnącym i na początku przy malejącym — dokładnie odwrotnie niż mogłaby podpowiadać intuicja „brak wartości jest najmniejszy”. MySQL traktuje `NULL` jako mniejszy od każdej innej wartości i nie zna składni `NULLS FIRST` — tam kolejność wymusza się wyrażeniem w `ORDER BY`, na przykład `ORDER BY klient_id IS NULL, klient_id`.

NULL a ograniczenie UNIQUE. Skoro `NULL` nigdy nie równa się nawet drugiemu `NULL`owi, powstaje pytanie, czy kolumna z ograniczeniem `UNIQUE` może przyjąć więcej niż jeden wiersz z `NULL`em. Silniki nie są tu zgodne.

Silnik	Ile wierszy z NULL dopuszcza kolumna UNIQUE
PostgreSQL	dowolnie wiele — każdy NULL liczy się jako różny od pozostałych
MySQL / MariaDB	dowolnie wiele, tak samo jak PostgreSQL
Oracle	dowolnie wiele
SQL Server	tylko jeden — drugi wiersz z NULLem narusza ograniczenie

SQL Server jest tu wyjątkiem względem pozostałej trójki i względem litery standardu — traktuje `NULL` w kontekście unikalności inaczej niż w kontekście porównania `=`, co samo w sobie jest źródłem zaskoczeń przy migracji schematu między silnikami.

NULL w złączeniach. Kiedy złączenie nie znajduje pasującego wiersza po drugiej stronie, kolumny z tej strony wypełniają się `NULL`ami zamiast zniknąć z wyniku — mechanizm centralny dla `LEFT JOIN`, opisany na stronie LEFT JOIN i RIGHT JOIN.

Częste błędy

Pusty ciąg znaków traktowany jak NULL w Oracle. W PostgreSQL i MySQL pusty ciąg `''` i `NULL` to dwie różne rzeczy — `''` to znana wartość, po prostu o zerowej długości. Oracle przy wstawianiu pustego ciągu do kolumny typu `VARCHAR2` konwertuje go w locie na `NULL`.

```
-- Oracle
INSERT INTO klienci (id, nazwa, miasto, nip) VALUES (99, 'Test spółka', 'Poznań', '');
SELECT nip FROM klienci WHERE id = 99 AND nip IS NULL;
```

Przykład:

```
no rows selected
```

Wiersz istnieje, ale ten warunek go nie znajdzie, dopóki nie doda się `nip IS NULL` zamiast `nip = ''` — bo `nip = ''` nigdy nie da prawdy, z tego samego powodu co `= NULL` w pierwszej Fiszce tej strony. Kod przenoszony z PostgreSQL do Oracle, który rozróżnia `''` od `NULL`, jest jednym z częstszych źródeł błędów przy migracji.

Wykluczanie NULLi przy negacji warunku. `WHERE status <> 'anulowane'` wygląda jak „wszystko poza anulowanymi”, ale jeśli w kolumnie `status` trafi się `NULL`, ten wiersz nie pojawi się ani w wyniku tego warunku, ani w wyniku `WHERE status = 'anulowane'` — trójwartościowa logika odrzuca `UNKNOWN` w obu przypadkach. Jeśli kolumna może być pusta, a wiersze z `NULL`em mają się znaleźć w wyniku negacji, warunek musi to powiedzieć wprost: `WHERE status <> 'anulowane' OR status IS NULL`.

Mylenie COUNT(kolumna) z COUNT(*). Zamiana `COUNT(*)` na `COUNT(id_klienta)` w zapytaniu liczącym zamówienia po cichu zmienia znaczenie zapytania z „ile jest wierszy” na „ile wierszy ma wypełnioną tę konkretną kolumnę” — różnica niewidoczna, dopóki dane są kompletne, i myląca w raporcie w dniu, w którym przestają być.

□□ Ćwiczenia

> Zadania sprawdzające.

Ćwiczenie 1 — trójwartościowa logika

Zamówienie 106 ma `klient_id IS NULL` i `status = 'nowe'`. Przewidujemy, czy trafi do wyniku `SELECT id FROM zamowienia WHERE klient_id = 3 OR status = 'nowe';` oraz czy trafi do wyniku `SELECT id FROM zamowienia WHERE klient_id = 3 AND status = 'nowe';`, zanim uruchomimy zapytanie.

Odpowiedź

W pierwszym zapytaniu `klient_id = 3` daje `UNKNOWN` (bo `klient_id` jest `NULL`em), ale `status = 'nowe'` daje `TRUE` — z tabeli prawdy `OR` wynika, że `UNKNOWN OR TRUE = TRUE`, więc wiersz 106 trafia do wyniku. W drugim zapytaniu `UNKNOWN AND TRUE = UNKNOWN`, a `WHERE` traktuje `UNKNOWN` jak `FALSE` — wiersz 106 odpada.

Pytania kontrolne

1. Dlaczego `WHERE klient_id != 3` nie zwróci wiersza z zamówieniem 106, mimo że `klient_id` tego wiersza z pewnością nie jest równe 3?

Odpowiedź

Bo `klient_id` w tym wierszu to `NULL`, a każde porównanie z `NULL`, także `!=`, daje `UNKNOWN` zamiast `TRUE` czy `FALSE`. `WHERE` odrzuca wiersze z wynikiem `UNKNOWN` tak samo jak z wynikiem `FALSE`.

2. Czym różni się zachowanie COALESCE od NULLIF?

Odpowiedź

`COALESCE` zamienia `NULL` na podaną wartość zastępczą — działa, gdy chcemy się pozbyć braku danych. `NULLIF` robi coś przeciwnego: zamienia konkretną, podaną wartość na `NULL` — działa, gdy chcemy oznaczyć jakąś wartość jako brak danych, na przykład przed dzieleniem.

3. Ile wierszy z `NULL`em w kolumnie `nip` może przyjąć tabela `klienci`, jeśli kolumna `nip` ma ograniczenie `UNIQUE`, a tabela stoi w PostgreSQL i w SQL Server?

Odpowiedź

W PostgreSQL dowolnie wiele — każdy `NULL` liczy się jako różny od pozostałych, więc ograniczenie `UNIQUE` nigdy go nie odrzuci. W SQL Server tylko jeden — drugi wiersz z `NULL`em w kolumnie `nip` naruszy ograniczenie unikalności, mimo że logicznie oba `NULL`e oznaczają tylko brak numeru NIP, nie tę samą wartość.

Powiązane: WHERE: filtrowanie wierszy · Funkcje agregujące i GROUP BY · LEFT JOIN i RIGHT JOIN · Ograniczenia: NOT NULL, UNIQUE, CHECK, FOREIGN KEY, DEFAULT

Źródła: ISO/IEC 9075-2:2016 § 6.3, 8.2 · dokumentacja PostgreSQL, rozdział „Comparison Operators” · dokumentacja Oracle Database SQL Language Reference, sekcja o NULL i VARCHAR2

ORDER BY, LIMIT, TOP, ROWNUM, FETCH: porcjowanie wyniku

> Bez `ORDER BY` kolejność wierszy jest przypadkowym efektem sposobu przechowywania danych, a każdy z czterech silników tej książki ogranicza liczbę zwracanych wierszy własną, niekompatybilną składnią — ta strona pokazuje obie sprawy razem, bo w praktyce zawsze występują w parze.

Skróty użyte na tej stronie (4)

Skrót	Co znaczy
ASCII	American Standard Code for Information Interchange: podstawowy zestaw znaków (litery, cyfry) używany w komputerach
ISO	International Organization for Standardization: organizacja, która wydaje normy techniczne używane na całym świecie
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

TYLKO ODCZYT

Sortowanie rosnące i malejące

```
SELECT id, wartosc FROM zamowienia ORDER BY wartosc DESC;
```

`ORDER BY` sortuje wynik według wskazanej kolumny. `DESC` sortuje malejąco, `ASC` — rosnąco i jest wartością domyślną, więc w praktyce pomija się je, jeśli sortowanie ma iść od najmniejszej wartości.

Przykład:

```
 id | wartosc
----+---------
107 | 3200.00
104 | 2890.00
101 | 1250.00
106 |  899.00
102 |  430.50
105 |  150.00
103 |    0.00
108 |    0.00
(8 rows)
```

□□ **ASC** = Automatic Self-Calibration: automatyczna samokalibracja czujnika CO2, która zakłada, że co tydzień czujnik zobaczy czyste powietrze o stężeniu około 400 ppm

□□ **TYLKO ODCZYT**

Sortowanie po wielu kolumnach

```
SELECT id, status, wartosc FROM zamowienia ORDER BY status ASC, wartosc DESC;
```

Kolumny w `ORDER BY` rozstrzygają kolejność po kolei — pierwsza ustala główne grupy, kolejna sortuje wewnątrz wierszy, które w pierwszej kolumnie mają tę samą wartość. Tu zamówienia najpierw grupują się alfabetycznie po statusie, a dopiero w obrębie jednego statusu sortują się malejąco po kwocie.

Przykład:

```
 id |    status     | wartosc
----+---------------+---------
103 | anulowane     |    0.00
108 | anulowane     |    0.00
106 | nowe          |  899.00
105 | nowe          |  150.00
104 | w realizacji  | 2890.00
```

```
107 | zrealizowane  | 3200.00
101 | zrealizowane  | 1250.00
102 | zrealizowane  |  430.50
(8 rows)
```

ASC = Automatic Self-Calibration: automatyczna samokalibracja czujnika CO2, która zakłada, że co tydzień czujnik zobaczy czyste powietrze o stężeniu około 400 ppm

TYLKO ODCZYT

Ograniczenie liczby wierszy z przesunięciem: LIMIT / OFFSET

```
SELECT id, wartosc FROM zamowienia ORDER BY wartosc DESC LIMIT 3 OFFSET 2;
```

Składnia PostgreSQL i MySQL. `LIMIT` ustala, ile wierszy zwrócić, `OFFSET` — ile wierszy pominąć od początku posortowanego wyniku, zanim zacznie się liczenie zwracanych. To druga strona wyniku wartości z poprzedniej Fiszki, pomijająca dwa najwyższe zamówienia.

Przykład:

```
 id | wartosc
----+---------
101 | 1250.00
106 |  899.00
102 |  430.50
(3 rows)
```

TYLKO ODCZYT

Ograniczenie liczby wierszy w SQL Server: TOP

```
SELECT TOP (3) id, wartosc FROM zamowienia ORDER BY wartosc DESC;
```

`TOP` stoi zaraz po `SELECT`, nie na końcu zapytania jak `LIMIT`. Bez `ORDER BY` `TOP` zwraca wiersze w kolejności, w jakiej silnik akurat je odczytał — nieokreślonej z tych samych powodów, co w części „Po co to istnieje” niżej.

Przykład:

```
 id | wartosc
----+---------
107 | 3200.00
104 | 2890.00
101 | 1250.00
(3 rows)
```

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

TYLKO ODCZYT

Przesunięcie i limit w SQL Server: OFFSET ... FETCH NEXT

```
SELECT id, wartosc FROM zamowienia
ORDER BY wartosc DESC
OFFSET 2 ROWS FETCH NEXT 3 ROWS ONLY;
```

Odpowiednik `LIMIT`/`OFFSET` w nowoczesnym T-SQL, dostępny od SQL Server 2012. W odróżnieniu od `TOP`, `OFFSET ... FETCH` wymaga obowiązkowego `ORDER BY` — bez niego zapytanie nie skompiluje się w ogóle, co samo w sobie chroni przed poleganiem na przypadkowej kolejności.

Przykład:

```
 id | wartosc
----+---------
101 | 1250.00
106 |  899.00
102 |  430.50
(3 rows)
```

SQL = Structured Query Language: język zapytań do relacyjnych baz danych · **NEXT** = Near-End Crosstalk: przesłuch zbliżny, czyli ile sygnału z jednej pary przecieka do sąsiedniej pary na tym samym końcu kabla · **T-SQL** = Transact-SQL: dialekt SQL używany w Microsoft SQL Server, wywodzący się z języka zapytań oryginalnego silnika Sybase

TYLKO ODCZYT

Ograniczenie liczby wierszy w starszym Oracle: ROWNUM

```
SELECT id, wartosc FROM (
    SELECT id, wartosc FROM zamowienia ORDER BY wartosc DESC
) WHERE ROWNUM <= 3;
```

> **Pojęcie: ROWNUM.** Pseudokolumna Oracle przypisująca każdemu wierszowi kolejny numer w miarę jego pobierania przez silnik — przed sortowaniem, nie po nim. `ROWNUM` w podzapytaniu wewnętrznym nie istnieje, dopóki podzapytanie się nie wykona, więc numerowanie w zapytaniu zewnętrznym dotyczy już posortowanego wyniku. Zagnieżdżenie w podzapytaniu jest tu obowiązkowe — bez niego `ROWNUM` numerowałby wiersze przed sortowaniem, co pokazuje sekcja „Częste błędy”.

Przykład:

```
 ID | WARTOSC
----+---------
107 |    3200
104 |    2890
101 |    1250
```

□□ TYLKO ODCZYT

Ograniczenie liczby wierszy w Oracle 12c i nowszych: FETCH FIRST

```
SELECT id, wartosc FROM zamowienia ORDER BY wartosc DESC FETCH FIRST 3 ROWS ONLY;
```

Składnia zgodna ze standardem SQL:2008, dostępna w Oracle od wersji 12c. Robi dokładnie to, co konstrukcja z `ROWNUM` w podzapytaniu, bez zagnieżdżania — Oracle rozwiązał we własnym silniku ten sam problem, jaki SQL Server rozwiązał przez `OFFSET ... FETCH`.

Przykład:

```
 ID | WARTOSC
----+---------
107 |    3200
104 |    2890
101 |    1250
```

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

◫◫ ZMIENIA SYSTEM

Ograniczenie liczby wierszy w Sybase: SET ROWCOUNT

```
SET ROWCOUNT 3;
SELECT id, wartosc FROM zamowienia ORDER BY wartosc DESC;
SET ROWCOUNT 0;
```

`SET ROWCOUNT` nie jest częścią zapytania, tylko oddzielnym poleceniem sesji — ustawia limit obowiązujący dla wszystkich kolejnych instrukcji, dopóki ktoś go nie zresetuje. Zapomnienie o `SET ROWCOUNT 0` na końcu zostawia limit działający na następne, niepowiązane zapytania w tej samej sesji — częsty błąd w skryptach, które kończą się przedwcześnie po wyjątku, zanim reset zdąży się wykonać.

Przykład:

```
 id | wartosc
----+---------
107 | 3200.00
104 | 2890.00
101 | 1250.00
(3 rows affected)
```

◫◫ **SET** = Switch Embedded Teaming: zespół kart sieciowych zbudowany w przełączniku wirtualnym Hyper-V; łączy dwie karty w jedno łącze bez ustawień po stronie przełącznika fizycznego, więc bez LACP

◫◫ TYLKO ODCZYT

Wszystkie warianty ograniczenia wyniku

Silnik	Ograniczenie liczby wierszy	Przesunięcie
PostgreSQL	`LIMIT n`	`OFFSET m`
MySQL / MariaDB	`LIMIT n` (albo `LIMIT m, n`)	`OFFSET m`
SQL Server	`TOP (n)` lub `OFFSET ... FETCH NEXT n ROWS ONLY`	`OFFSET m ROWS` (tylko z FETCH)
Oracle 12c+	`FETCH FIRST n ROWS ONLY`	`OFFSET m ROWS`
Oracle starsze	`ROWNUM <= n` w podzapytaniu	dwa zagnieżdżone `ROWNUM`
Sybase ASE	`SET ROWCOUNT n` przed zapytaniem	brak wbudowanego odpowiednika

Uważaj

- `TOP` bez `ORDER BY` i `LIMIT` bez `ORDER BY` kompilują się bez ostrzeżenia, ale zwracają wiersze w kolejności, której nikt nie gwarantuje.
- `ROWNUM <= n` zastosowany przed `ORDER BY`, zamiast w podzapytaniu, filtruje pierwsze `n` wierszy w kolejności odczytu, a dopiero potem je sortuje.

NEXT = Near-End Crosstalk: przesłuch zbliżny, czyli ile sygnału z jednej pary przecieka do sąsiedniej pary na tym samym końcu kabla

Po co to istnieje

Pojęcie: nieokreślona kolejność. Tabela relacyjna to zbiór wierszy, nie lista — a zbiór z definicji nie ma kolejności. Silnik przechowuje wiersze fizycznie w jakiejś kolejności (zwykle w kolejności wstawienia, na stronach danych na dysku), ale ta fizyczna kolejność nie jest częścią żadnej umowy z zapytaniem. `VACUUM` w PostgreSQL, przebudowa indeksu, `UPDATE` przenoszący wiersz w nowe miejsce po przekroczeniu rozmiaru strony, równoległe wykonanie zapytania na kilku wątkach naraz — każde z tych zdarzeń może zmienić kolejność, w jakiej wiersze trafiają do wyniku zapytania bez `ORDER BY`. Zapytanie, które przez pół roku zwracało dane „w kolejności wstawienia", może zacząć zwracać je inaczej z dnia na dzień bez żadnej zmiany w samym zapytaniu — bo to nigdy nie była gwarancja, tylko przypadkowy efekt uboczny.

Jak to działa

Rozwiń: jak to działa pod spodem

Sortowanie po numerze kolumny. Zamiast nazwy kolumny `ORDER BY` przyjmuje też jej numer pozycyjny w liście `SELECT`.

```
SELECT nazwa, miasto FROM klienci ORDER BY 2;
```

Przykład:

```
           nazwa               |  miasto
-------------------------------+-----------
 Podlaski Handel               | Białystok
 Baltic Trade Sp. z o.o.       | Gdańsk
 Kowalski Handel               | Kraków
 Nowak i Wspólnicy Sp. z o.o.  | Warszawa
 Zielona Dolina S.A.           | Wrocław
(5 rows)
```

Działa identycznie jak `ORDER BY miasto`, ale nie mówi tego wprost. Nie sięgamy po tę składnię w kodzie, który ma przetrwać dłużej niż jedno zapytanie napisane naprędce w konsoli — jeśli ktoś później zmieni kolejność kolumn w `SELECT`, na przykład doda kolumnę na początek listy, `ORDER BY 2` zacznie sortować po zupełnie innej kolumnie, bez żadnego błędu i bez żadnego ostrzeżenia od silnika.

Collation. Pojęcie: collation. Zestaw reguł określających, jak silnik porównuje i sortuje wartości tekstowe — które znaki uznaje za „mniejsze”, jak traktuje wielkość liter i znaki diakrytyczne. Collation `C` (albo `POSIX`) sortuje bajtowo, według wartości liczbowej kodu znaku, więc wielkie litery trafiają przed małymi, a `Ż` jako znak spoza podstawowego zakresu ASCII może wylądować za literą `z`. Collation `pl_PL` sortuje lingwistycznie — `Żabka` znajdzie się w sąsiedztwie innych słów na `Z`, tak jak w polskim słowniku. Ten sam `ORDER BY nazwa` na tej samej tabeli może dać dwa różne porządki w zależności wyłącznie od collation ustawionego dla bazy albo kolumny — temat wart świadomej decyzji przy zakładaniu bazy, nie domyślnego ustawienia zostawionego przypadkowi.

Paginacja przez OFFSET i dlaczego zwalnia. Pojęcie: paginacja. Podział dużego wyniku na kolejne, mniejsze porcje — strony — pobierane osobnymi zapytaniami, zamiast jednym zapytaniem zwracającym wszystko naraz. Najprostszy sposób jej realizacji to rosnący `OFFSET`: strona pierwsza to `LIMIT 20 OFFSET 0`, druga to `LIMIT 20 OFFSET 20`, setna to `LIMIT 20 OFFSET 1980`. Problem w tym, że `OFFSET` nie mówi silnikowi „przeskocz od razu do wiersza numer 1980” — silnik i tak musi posortować cały kwalifikujący się zbiór i policzyć kolejno pierwsze 1980 wierszy, żeby wiedzieć, które odrzucić, zanim zwróci kolejne 20. Im dalsza strona, tym więcej pracy wykonanej wyłącznie po to, żeby zostać wyrzuconą.

> **Pojęcie: paginacja kluczowa.** Alternatywa dla `OFFSET`, w której kolejna strona nie liczy się od początku wyniku, tylko od ostatniego wiersza poprzedniej strony — stąd angielska nazwa keyset pagination, po polsku czasem „paginacja przez klucz” albo „metoda seek”.

```
SELECT id, wartosc FROM zamowienia
WHERE wartosc < 1250.00
```

```
ORDER BY wartosc DESC
LIMIT 3;
```

Przykład:

```
 id | wartosc
----+---------
106 |  899.00
102 |  430.50
105 |  150.00
(3 rows)
```

Zamiast pomijać wiersze przez `OFFSET`, warunek `WHERE wartosc < 1250.00` odwołuje się do wartości z ostatniego wiersza poprzedniej strony (`101`, `1250.00`) i od razu każe silnikowi szukać dalej w indeksie od tego miejsca — bez liczenia i odrzucania wszystkiego, co było wcześniej. Koszt takiego zapytania nie rośnie wraz z numerem strony, tak jak rośnie przy `OFFSET`. Cena tej metody to brak swobodnego skoku na dowolny numer strony — działa dobrze dla „następna strona”, gorzej dla „przejdź od razu na stronę 47”.

Częste błędy

ROWNUM przed ORDER BY zamiast po nim. `SELECT id, wartosc FROM zamowienia WHERE ROWNUM <= 3 ORDER BY wartosc DESC;` wygląda logicznie, ale `ROWNUM` przypisuje numery w kolejności fizycznego odczytu, przed sortowaniem — więc filtr wybiera trzy dowolne wiersze, a `ORDER BY` tylko je sortuje, nie zmieniając, które trzy zostały wybrane. Poprawka to zagnieżdżenie zapytania z `ORDER BY` w podzapytaniu i filtrowanie `ROWNUM` dopiero na zewnątrz, tak jak w Fiszce tej strony.

Rosnący OFFSET w paginacji rankingu z częstymi zmianami danych. Jeśli między pobraniem strony pierwszej a drugiej ktoś wstawi nowy wiersz, który trafia przed granicą sortowania, `OFFSET` przesuwa się o jeden wiersz i ten sam rekord może pojawić się na obu stronach albo zniknąć z obu. Paginacja kluczowa nie ma tego problemu, bo nie liczy pozycji, tylko porównuje wartość klucza.

TOP bez PERCENT i bez ORDER BY w SQL Server. `SELECT TOP (10) * FROM zamowienia;` bez `ORDER BY` zwróci dziesięć wierszy w kolejności, jaką akurat wybrał optymalizator — często zgodną z kolejnością klucza głównego, ale to przypadek, nie gwarancja, i pierwsza zmiana planu wykonania może ją zmienić.

□□ Ćwiczenia

> Zadania sprawdzające.

Ćwiczenie 1 — druga strona wyniku

Zamówienia z Fiszki tej strony sortujemy malejąco po `wartosc`. Piszemy zapytanie w składni PostgreSQL zwracające trzecią stronę wyniku, przy założeniu, że jedna strona ma dwa wiersze.

Odpowiedź

```
SELECT id, wartosc FROM zamowienia ORDER BY wartosc DESC LIMIT 2 OFFSET 4;
```

Strona pierwsza to wiersze 1–2 (`OFFSET 0`), strona druga to wiersze 3–4 (`OFFSET 2`), strona trzecia to wiersze 5–6, czyli `OFFSET 4`:

```
 id | wartosc
----+---------
102 |  430.50
105 |  150.00
(2 rows)
```

Pytania kontrolne

1. Dlaczego zapytanie bez ORDER BY, które przez rok zwracało wiersze w kolejności wstawienia, może pewnego dnia zacząć zwracać je w innej kolejności bez żadnej zmiany w samym zapytaniu?

Odpowiedź

Kolejność wierszy bez `ORDER BY` nigdy nie była gwarantowana — była przypadkowym efektem fizycznego sposobu przechowywania danych. Operacje takie jak `VACUUM`, przebudowa indeksu czy zmiana planu wykonania na równoległy mogą zmienić tę fizyczną kolejność, nie naruszając przy tym żadnej reguły SQL.

2. Dlaczego paginacja przez rosnący OFFSET zwalnia na dalszych stronach, a paginacja kluczowa nie?

Odpowiedź

`OFFSET` każe silnikowi policzyć i odrzucić wszystkie wiersze przed żądaną stroną, więc koszt rośnie liniowo z numerem strony. Paginacja kluczowa filtruje `WHERE` względem wartości ostatniego wiersza poprzedniej strony, więc silnik może od razu wejść w odpowiednie miejsce indeksu, niezależnie od tego, która to strona z kolei.

Powiązane: SELECT: kolejność zapisu a kolejność wykonania · Czym jest indeks: i dlaczego to nie jest „przyspieszacz tabeli” · Skąd się wziął SQL i dlaczego każdy silnik mówi nim inaczej

Źródła: dokumentacja PostgreSQL, rozdział „LIMIT and OFFSET” · dokumentacja Microsoft, „ORDER BY Clause (Transact-SQL)” · Oracle Database SQL Language Reference, „Row Limiting Clause” · ISO/IEC 9075-2:2016 § 7.13

Funkcje agregujące i GROUP BY

Agregacja zwija wiele wierszy w jeden, a `GROUP BY` decyduje, według jakiego klucza to zwijanie ma przebiegać — ta strona pokazuje obie rzeczy razem, bo funkcja agregująca bez `GROUP BY` liczy jedną wartość dla całej tabeli, a z `GROUP BY` — jedną wartość dla każdej grupy osobno.

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

TYLKO ODCZYT

Zliczenie wszystkich wierszy: COUNT(*)

```
SELECT COUNT(*) AS liczba_zamowien FROM zamowienia;
```

`COUNT(*)` liczy wiersze, niezależnie od tego, co mają w kolumnach — nawet jeśli wszystkie kolumny w danym wierszu są `NULL`em, wiersz i tak się liczy.

Przykład:

```
 liczba_zamowien
-----------------
               8
```

```
(1 row)
```

TYLKO ODCZYT

Zliczenie wypełnionych wartości: COUNT(kolumna)

```
SELECT COUNT(klient_id) AS zamowien_z_klientem FROM zamowienia;
```

`COUNT` z nazwą kolumny liczy tylko wiersze, w których ta kolumna nie jest `NULL`em. Strona NULL: trzecia wartość logiczna i pułapki, które z niej wynikają opisuje to zachowanie wraz z tabelami prawdy, które je tłumaczą.

Przykład:

```
 zamowien_z_klientem
----------------------
                    7
(1 row)
```

TYLKO ODCZYT

Suma: SUM

```
SELECT SUM(wartosc) AS suma_zamowien FROM zamowienia WHERE status = 'zrealizowane';
```

`SUM` dodaje wartości z kolumny liczbowej po wszystkich wierszach, które przeszły przez `WHERE`. Wiersze z `NULL`em w sumowanej kolumnie są pomijane, tak jakby ich nie było — nie liczą się jako zero.

Przykład:

```
 suma_zamowien
---------------
       4880.50
(1 row)
```

TYLKO ODCZYT

Średnia: AVG

```
SELECT AVG(cena) AS srednia_cena FROM produkty WHERE kategoria = 'elektronika';
```

`AVG` liczy średnią arytmetyczną z wartości niepustych. Mianownikiem jest liczba wierszy z wypełnioną kolumną, nie liczba wszystkich wierszy — `NULL`e nie wchodzą ani do sumy, ani do liczby dzielników.

Przykład:

```
    srednia_cena
---------------------
 298.3333333333333333
(1 row)
```

□□ TYLKO ODCZYT

Minimum i maksimum: MIN, MAX

```
SELECT MIN(cena) AS najtansza, MAX(cena) AS najdrozsza FROM produkty;
```

`MIN` i `MAX` działają na dowolnym typie porównywalnym — liczbach, datach, tekście — nie tylko na liczbach. Podobnie jak pozostałe agregaty, ignorują `NULL`e.

Przykład:

```
 najtansza | najdrozsza
-----------+------------
     25.00 |    1100.00
(1 row)
```

□□ TYLKO ODCZYT

Sklejenie tekstu z grupy: STRING_AGG

```
SELECT kategoria, STRING_AGG(nazwa, ', ' ORDER BY nazwa) AS produkty
FROM produkty
GROUP BY kategoria;
```

`STRING_AGG` skleja wartości tekstowe z wielu wierszy jednej grupy w jeden ciąg znaków, rozdzielony podanym separatorem. Nazwa i dokładna składnia różnią się między silnikami mocniej niż w przypadku pozostałych agregatów.

Przykład:

```
  kategoria  |                    produkty
-------------+-------------------------------------------------
 elektronika | Kabel HDMI 2m, Klawiatura mechaniczna, Monitor 24 cale
 meble       | Biurko regulowane, Fotel biurowy, Lampka biurkowa
(2 rows)
```

Silnik	Funkcja sklejająca tekst z grupy
PostgreSQL	`STRING_AGG(kolumna, separator)`
SQL Server (2017+)	`STRING_AGG(kolumna, separator)`
MySQL / MariaDB	`GROUP_CONCAT(kolumna SEPARATOR separator)`
Oracle	`LISTAGG(kolumna, separator)`

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

TYLKO ODCZYT

Grupowanie po jednej kolumnie

```
SELECT kategoria, COUNT(*) AS liczba, AVG(cena) AS srednia_cena
FROM produkty
GROUP BY kategoria;
```

`GROUP BY` zwija wiersze o tej samej wartości wskazanej kolumny w jeden wiersz wyniku, a funkcje agregujące w `SELECT` liczą się osobno dla każdej takiej grupy.

Przykład:

```
  kategoria  | liczba |    srednia_cena
-------------+--------+--------------------
 elektronika |      3 | 298.3333333333333333
 meble       |      3 | 693.3333333333333333
(2 rows)
```

TYLKO ODCZYT

Grupowanie po wielu kolumnach

```
SELECT klient_id, status, COUNT(*) AS liczba, SUM(wartosc) AS suma
FROM zamowienia
GROUP BY klient_id, status
ORDER BY klient_id, status;
```

Kombinacja wartości ze wszystkich kolumn `GROUP BY` razem definiuje jedną grupę — dwa wiersze trafiają do tej samej grupy tylko wtedy, gdy zgadzają się na wszystkich wymienionych kolumnach naraz.

Przykład:

```
 klient_id |    status     | liczba |  suma
-----------+---------------+--------+---------
         1 | zrealizowane  |      2 | 1680.50
         2 | anulowane     |      1 |    0.00
         2 | nowe          |      1 |  150.00
         3 | w realizacji  |      1 | 2890.00
         4 | zrealizowane  |      1 | 3200.00
         5 | anulowane     |      1 |    0.00
           | nowe          |      1 |  899.00
(7 rows)
```

Zamówienie 106, z `klient_id` równym `NULL`, dostało własną grupę zamiast zniknąć z wyniku — `GROUP BY` traktuje `NULL` jako jedną, konkretną wartość do celów grupowania, w odróżnieniu od `WHERE`, gdzie porównanie z `NULL`em nigdy nie daje prawdy.

TYLKO ODCZYT

Grupowanie po wyrażeniu

```
SELECT EXTRACT(YEAR FROM data) AS rok, EXTRACT(MONTH FROM data) AS miesiac, COUNT(*) AS
liczba
FROM zamowienia
GROUP BY EXTRACT(YEAR FROM data), EXTRACT(MONTH FROM data)
ORDER BY rok, miesiac;
```

`GROUP BY` nie ogranicza się do nazw kolumn — może grupować po dowolnym wyrażeniu, o ile to samo wyrażenie da się policzyć dla każdego wiersza. Tu grupujemy zamówienia po miesiącu, mimo że w tabeli nie ma osobnej kolumny na miesiąc.

Przykład:

```
 rok | miesiac | liczba
-----+---------+--------
2024 |       1 |      2
2024 |       2 |      1
2024 |       3 |      3
2024 |       4 |      2
(4 rows)
```

Uważaj

- Kolumna w `SELECT`, której nie ma ani w `GROUP BY`, ani wewnątrz funkcji agregującej, jest błędem w PostgreSQL i Oracle — i niebezpiecznym wyjątkiem w starszym MySQL, opisanym niżej.
- `COUNT(kolumna)` i `COUNT(*)` w zapytaniu bez `GROUP BY` dają różne liczby tylko wtedy, gdy kolumna ma choć jeden `NULL` — przy kompletnych danych są identyczne, co maskuje pomyłkę do dnia, w którym dane przestają być kompletne.

Po co to istnieje

Pojęcie: agregacja. Zwinięcie wielu wierszy w jeden przez policzenie z nich pojedynczej wartości — sumy, średniej, licznika, wartości skrajnej albo sklejonego tekstu. Bez `GROUP BY` cała tabela (albo to, co z niej zostało po `WHERE`) jest jedną wielką grupą i funkcja agregująca zwraca dokładnie jeden wiersz wyniku. Z `GROUP BY` tabela dzieli się na tyle grup, ile różnych kombinacji wartości występuje w kolumnach grupujących, a agregacja liczy się osobno w każdej z nich.

Jak to działa

Rozwiń: jak to działa pod spodem

Co wolno w SELECT przy grupowaniu. Jeśli zapytanie ma `GROUP BY`, każda kolumna wymieniona w `SELECT` musi być albo kolumną grupującą, albo argumentem funkcji agregującej. Powód jest logiczny, nie arbitralny: grupa złożona z kilku wierszy nie ma jednej wspólnej wartości kolumny, która nie była częścią grupowania — `status` w grupie o `klient_id = 1` mógłby dla różnych wierszy tej grupy być inny, więc pytanie „który status pokazać w tym jednym wierszu wyniku” nie ma jednoznacznej odpowiedzi bez dodatkowej reguły.

```
SELECT klient_id, status, wartosc FROM zamowienia GROUP BY klient_id;
```

Przykład:

```
ERROR:  column "zamowienia.status" must appear in the GROUP BY clause
        or be used in an aggregate function
LINE 1: SELECT klient_id, status, wartosc FROM zamowienia GROUP BY...
                          ^
```

MySQL i ONLY_FULL_GROUP_BY. MySQL przed wersją 5.7 domyślnie pozwalał na dokładnie to, co wyżej kończy się błędem w PostgreSQL — zwracał wtedy wartość `status` i `wartosc` z jednego, dowolnie wybranego wiersza danej grupy, bez żadnej gwarancji, z którego. Zapytanie nie zgłaszało błędu i wyglądało na działające, dopóki ktoś nie porównał wyniku z oczekiwaniem. Od wersji 5.7.5 tryb `ONLY_FULL_GROUP_BY` jest domyślnie włączony i MySQL zaczyna zgłaszać ten sam błąd co PostgreSQL. Zapytania napisane pod starsze MySQL i migrowane na nowszą wersję albo na inny silnik są jednym z częstszych źródeł awarii przy takiej migracji — kod latami działał, bo nikt nie sprawdził, że wynik był przypadkowy.

ROLLUP, CUBE, GROUPING SETS. Trzy rozszerzenia `GROUP BY`, które oprócz zwykłych grup dokładają do wyniku wiersze podsumowań.

> **Pojęcie: ROLLUP.** Rozszerzenie `GROUP BY`, które oprócz każdej grupy dokłada wiersze sum częściowych, cofając się po kolei wzdłuż listy kolumn grupujących, aż do jednego wiersza sumy całkowitej.

```
SELECT status, SUM(wartosc) AS suma
FROM zamowienia
GROUP BY ROLLUP (status);
```

Przykład:

```
    status     |  suma
---------------+---------
 anulowane     |    0.00
 nowe          | 1049.00
 w realizacji  | 2890.00
 zrealizowane  | 4880.50
               | 8819.50
(5 rows)
```

Ostatni wiersz, z pustym `status`, to suma po wszystkich statusach naraz — `NULL` w tym miejscu nie znaczy „brak danych o statusie”, tylko „ten wiersz to podsumowanie, nie pojedyncza grupa”. Rozróżnienie tych dwóch znaczeń tego samego `NULL`a jest źródłem pomyłek opisanych niżej.

`CUBE` rozszerza ten pomysł na wszystkie możliwe kombinacje kolumn grupujących, nie tylko kolejne cofnięcia od pełnej listy do pustej. `GROUPING SETS` pozwala wprost wymienić, które kombinacje grupowania mają się znaleźć w wyniku, zamiast polegać na automatycznym schemacie `ROLLUP` czy `CUBE`.

Częste błędy

Mylenie NULL z ROLLUP z prawdziwym brakiem danych. Wiersz podsumowania z `ROLLUP` ma `NULL` w kolumnie grupującej z zupełnie innego powodu niż wiersz zamówienia 106, w którym `klient_id` naprawdę jest nieznany. Kod, który filtruje `WHERE status IS NOT NULL` po zapytaniu z `ROLLUP`, usunie wiersz podsumowania razem z prawdziwymi brakami danych, jeśli oba źródła `NULL` wystąpią w tej samej kolumnie.

Filtrowanie po agregacie w WHERE zamiast w HAVING. `WHERE SUM(wartosc) > 1000` nie skompiluje się — `WHERE` działa przed grupowaniem, więc agregat jeszcze nie istnieje. Strona HAVING: dlaczego nie da się tego zrobić w WHERE rozwija ten temat.

Zakładanie, że COUNT(kolumna) i COUNT(*) zawsze dają to samo. Różnią się tylko wtedy, gdy kolumna zawiera `NULL`e — dopóki dane są kompletne, różnica jest niewidoczna, co usypia czujność do dnia, w którym w danych pojawi się pierwszy brak.

□□ Ćwiczenia

> Zadania sprawdzające.

Ćwiczenie 1 — grupowanie po wielu kolumnach

Na podstawie tabeli `pozycje(id, zamowienie_id, produkt_id, ilosc, cena)` piszemy zapytanie liczące, ile sztuk każdego produktu sprzedano łącznie (`SUM(ilosc)`), grupując po `produkt_id`.

Odpowiedź

```
SELECT produkt_id, SUM(ilosc) AS sztuk_lacznie
FROM pozycje
GROUP BY produkt_id
ORDER BY produkt_id;
```

`SELECT` zawiera tylko `produkt_id` (kolumna grupująca) i `SUM(ilosc)` (agregat) — dokładnie to, co reguła „co wolno w SELECT przy grupowaniu” dopuszcza.

Pytania kontrolne

1. Dlaczego `SELECT klient_id, wartosc FROM zamowienia GROUP BY klient_id;` jest błędem, mimo że `wartosc` istnieje w tabeli?

Odpowiedź

Bo `wartosc` nie jest kolumną grupującą ani argumentem funkcji agregującej. Grupa dla danego `klient_id` może zawierać kilka wierszy o różnej `wartosc`, więc silnik nie ma jednoznacznej odpowiedzi, którą z nich pokazać w jednym wierszu wyniku.

2. Co się stanie z wierszem, w którym kolumna grupująca ma wartość NULL?

Odpowiedź

Dostanie własną grupę, tak jakby `NULL` był jedną, konkretną wartością do porównania z innymi `NULL`ami w tej samej kolumnie — inaczej niż w `WHERE`, gdzie porównanie z `NULL` nigdy nie daje prawdy.

Powiązane: NULL: trzecia wartość logiczna i pułapki, które z niej wynikają · HAVING: dlaczego nie da się tego zrobić w WHERE · SELECT: kolejność zapisu a kolejność wykonania

Źródła: dokumentacja PostgreSQL, rozdział „Aggregate Functions” i „GROUPING SETS, CUBE, and ROLLUP” · dokumentacja MySQL, sekcja „ONLY_FULL_GROUP_BY SQL Mode” · ISO/IEC 9075-2:2016 § 10.9

HAVING: dlaczego nie da się tego zrobić w WHERE

> `HAVING` filtruje grupy, `WHERE` filtruje wiersze, a granica między nimi wynika wprost z logicznej kolejności wykonania zapytania — ta strona pokazuje, kiedy warunek należy do jednej klauzuli, kiedy do drugiej, i co się dzieje, gdy `HAVING` pojawia się bez `GROUP BY`.

Skróty użyte na tej stronie (1)

Skrót	Co znaczy
ISO	International Organization for Standardization: organizacja, która wydaje normy techniczne używane na całym świecie

Fiszka

TYLKO ODCZYT

Filtrowanie wierszy przed grupowaniem: WHERE

```
SELECT klient_id, COUNT(*) AS liczba_zamowien
FROM zamowienia
WHERE status = 'zrealizowane'
GROUP BY klient_id;
```

Warunek dotyczy pojedynczego wiersza — statusu konkretnego zamówienia — więc trafia do `WHERE`, zanim silnik zacznie cokolwiek grupować. Wiersze niespełniające warunku odpadają najwcześniej, jak to możliwe, i nigdy nie trafiają do grupowania.

Przykład:

```
 klient_id | liczba_zamowien
-----------+------------------
```

```
         1 |                2
         4 |                1
(2 rows)
```

⚠ ZMIENIA SYSTEM

Filtrowanie grup po agregacie: HAVING

```
SELECT klient_id, COUNT(*) AS liczba_zamowien
FROM zamowienia
GROUP BY klient_id
HAVING COUNT(*) > 1;
```

Warunek dotyczy nie pojedynczego wiersza, tylko wyniku policzonego dla całej grupy — liczby zamówień danego klienta. Taki warunek nie ma prawa trafić do `WHERE`, bo `COUNT(*)` dla grupy nie istnieje, dopóki grupowanie się nie wykona, a `WHERE` działa przed nim.

Przykład:

```
 klient_id | liczba_zamowien
-----------+-----------------
         1 |                2
         2 |                2
(2 rows)
```

⚠ ZMIENIA SYSTEM

WHERE i HAVING w jednym zapytaniu

```
SELECT klient_id, SUM(wartosc) AS suma
FROM zamowienia
WHERE status <> 'anulowane'
GROUP BY klient_id
HAVING SUM(wartosc) > 500
ORDER BY klient_id;
```

Nic nie stoi na przeszkodzie, żeby jedno zapytanie odsiewało wiersze przed grupowaniem i grupy po grupowaniu naraz — to normalny, częsty przypadek, nie wyjątek. `WHERE` odrzuca anulowane zamówienia, zanim cokolwiek zostanie zgrupowane, a `HAVING` odrzuca klientów,

których suma pozostałych zamówień nie przekracza 500.

Przykład:

```
 klient_id |  suma
-----------+---------
         1 | 1680.50
         3 | 2890.00
         4 | 3200.00
           |  899.00
(4 rows)
```

Ostatni wiersz to grupa zamówień bez przypisanego klienta (`klient_id IS NULL`) — `GROUP BY` traktuje `NULL` jako osobną, prawidłową grupę, tak jak opisuje to strona Funkcje agregujące i GROUP BY.

◫◫ ZMIENIA SYSTEM

HAVING bez GROUP BY

```
SELECT COUNT(*) AS liczba, SUM(wartosc) AS suma
FROM zamowienia
HAVING SUM(wartosc) > 5000;
```

Bez `GROUP BY` cała tabela liczy się jako jedna, jedyna grupa — `HAVING` filtruje wtedy tę jedną grupę tak samo, jak filtrowałby jedną z wielu. Jeśli warunek jest spełniony, zapytanie zwraca dokładnie jeden wiersz z policzonymi agregatami; jeśli nie, zwraca zero wierszy zamiast jednego.

Przykład:

```
 liczba |  suma
--------+---------
      8 | 8819.50
(1 row)
```

◫◫ ZMIENIA SYSTEM

Kilka warunków na agregatach naraz

```
SELECT klient_id, COUNT(*) AS liczba, SUM(wartosc) AS suma
FROM zamowienia
GROUP BY klient_id
HAVING COUNT(*) > 1 AND SUM(wartosc) > 1000;
```

`HAVING` łączy warunki tak samo jak `WHERE` — przez `AND`, `OR` i `NOT`, z tymi samymi priorytetami. Nic nie stoi na przeszkodzie, żeby jeden warunek odwoływał się do jednego agregatu, a drugi do innego, o ile oba dotyczą tej samej grupy.

Przykład:

```
 klient_id | liczba |  suma
-----------+--------+---------
         1 |      2 | 1680.50
(1 row)
```

Klient 2 odpada mimo dwóch zamówień, bo ich suma (`0.00 + 150.00 = 150.00`) nie przekracza tysiąca — oba warunki muszą być spełnione naraz.

□□ ZMIENIA SYSTEM

Filtrowanie po liczbie unikalnych wartości w grupie

```
SELECT klient_id, COUNT(DISTINCT status) AS liczba_statusow
FROM zamowienia
GROUP BY klient_id
HAVING COUNT(DISTINCT status) > 1;
```

`HAVING` sprawdza dowolne wyrażenie agregujące, nie tylko gołe `COUNT(*)` czy `SUM`. Tu szukamy klientów, których zamówienia miały więcej niż jeden różny status — sygnał, że historia zamówień takiego klienta jest bardziej zróżnicowana niż u pozostałych.

Przykład:

```
 klient_id | liczba_statusow
-----------+-----------------
         2 |               2
(1 row)
```

Klient 2 ma jedno zamówienie anulowane i jedno nowe — dwa różne statusy. Pozostali klienci mają albo jedno zamówienie, albo kilka zamówień o tym samym statusie.

> **Uważaj**
>
> - Agregat w `WHERE` (`WHERE COUNT(*) > 1`) kończy się błędem składniowym — agregaty nie istnieją jeszcze na etapie `WHERE`.
> - Warunek bez agregatu w `HAVING` (`HAVING status = 'nowe'`) często działa, ale filtruje później, niż mógłby, marnując pracę włożoną w pogrupowanie wierszy, które i tak zostaną odrzucone.

Po co to istnieje

Strona SELECT: kolejność zapisu a kolejność wykonania ustaliła logiczną kolejność wykonania zapytania: `FROM` → `WHERE` → `GROUP BY` → `HAVING` → `SELECT` → `ORDER BY`. `WHERE` wykonuje się w kroku drugim, długo zanim jakiekolwiek grupowanie miało szansę się zdarzyć — dlatego warunek w `WHERE` może odwoływać się wyłącznie do kolumn pojedynczego wiersza, nigdy do wyniku funkcji agregującej. `HAVING` wykonuje się w kroku czwartym, po `GROUP BY` — dopiero wtedy agregaty policzone dla każdej grupy istnieją i mogą wejść w warunek.

Z tej różnicy w momencie wykonania wynika praktyczna zasada podziału pracy między obiema klauzulami, rozwinięta niżej.

Jak to działa

Kiedy warunek trafia do WHERE. Zawsze wtedy, gdy dotyczy wartości dostępnej w pojedynczym wierszu, zanim cokolwiek zostanie zgrupowane — statusu, daty, identyfikatora klienta, dowolnej kolumny źródłowej albo prostego wyrażenia na niej. Filtrowanie w `WHERE` jest tańsze, bo silnik odrzuca niepotrzebne wiersze najwcześniej, jak to możliwe — mniej wierszy trafia do grupowania, mniej pracy wykonuje sortowanie czy budowanie tabeli haszującej pod agregację.

Kiedy warunek musi trafić do HAVING. Zawsze wtedy, gdy odwołuje się do wyniku funkcji agregującej — sumy, liczby wierszy w grupie, średniej, wartości minimalnej czy maksymalnej policzonej dla całej grupy. Taki warunek fizycznie nie może zostać sprawdzony wcześniej niż grupowanie, bo przed grupowaniem nie ma jeszcze niczego do policzenia — pojedynczy wiersz nie ma własnej sumy zamówień klienta, ma tylko wartość swojego jednego zamówienia.

Reguła praktyczna: jeśli da się przenieść warunek z `HAVING` do `WHERE` bez zmiany znaczenia zapytania, warto to zrobić — silnik odsieje wiersze wcześniej i grupowanie będzie miało mniej

materiału do przetworzenia. Warunek zostaje w `HAVING` tylko wtedy, gdy naprawdę nie da się go inaczej wyrazić — bo dotyczy czegoś, co istnieje wyłącznie na poziomie grupy.

Różnicę widać w planie wykonania, opisanym dokładniej na stronie Plan wykonania: jak go wywołać i jak go czytać. Warunek przeniesiony z `HAVING` do `WHERE` przesuwa się w planie z filtra nałożonego na węzeł grupujący (`GroupAggregate` albo `HashAggregate` z dodatkowym `Filter`) na filtr nałożony na sam odczyt tabeli, często jeszcze przed jakimkolwiek sortowaniem czy haszowaniem — mniej wierszy wchodzi do droższego etapu, więc ten etap kończy się szybciej.

Częste błędy

Agregat wstawiony do WHERE z przyzwyczajenia.

```
SELECT klient_id, COUNT(*) AS liczba
FROM zamowienia
WHERE COUNT(*) > 1
GROUP BY klient_id;
```

Przykład:

```
ERROR:  aggregate functions are not allowed in WHERE
LINE 3: WHERE COUNT(*) > 1
              ^
```

Poprawka to przeniesienie warunku do `HAVING`, jak w drugiej Fiszce tej strony.

Warunek na pojedynczym wierszu trzymany w HAVING zamiast w WHERE.

```
SELECT klient_id, COUNT(*) AS liczba
FROM zamowienia
GROUP BY klient_id
HAVING klient_id IS NOT NULL;
```

Zapytanie zadziała i da poprawny wynik, ale silnik najpierw pogrupuje wszystkie wiersze, łącznie z tymi o `klient_id IS NULL`, i dopiero po fakcie odrzuci niepotrzebną grupę. Ten sam warunek w `WHERE` odsiałby wiersz 106 przed grupowaniem, więc grupowanie miałoby mniej pracy — przy ośmiu wierszach różnica jest niemierzalna, przy ośmiu milionach już nie.

HAVING bez GROUP BY mylone z pustym wynikiem WHERE. Zapytanie `SELECT SUM(wartosc) FROM zamowienia HAVING SUM(wartosc) > 100000;` przy niespełnionym warunku zwraca zero wierszy, nie jeden wiersz z wartością `NULL` czy zerem — łatwo to pomylić z sytuacją, w której zapytanie się

nie wykonało, podczas gdy wykonało się poprawnie i po prostu odrzuciło jedyną istniejącą grupę.

Odwołanie do aliasu z SELECT wewnątrz HAVING.

```
SELECT klient_id, COUNT(*) AS liczba
FROM zamowienia
GROUP BY klient_id
HAVING liczba > 1;
```

Przykład:

```
ERROR:  column "liczba" does not exist
LINE 4: HAVING liczba > 1;
               ^
```

Ten sam mechanizm, który w SELECT: kolejność zapisu a kolejność wykonania tłumaczy, dlaczego alias z `SELECT` nie działa w `WHERE`, obowiązuje też tutaj — `HAVING` wykonuje się w kroku czwartym logicznej kolejności, `SELECT` dopiero w piątym, więc alias `liczba` jeszcze nie istnieje w momencie sprawdzania `HAVING`. Poprawka to powtórzenie pełnego wyrażenia agregującego: `HAVING COUNT(*) > 1`.

□□ Ćwiczenia

> Zadania sprawdzające.

Ćwiczenie 1 — dobór klauzuli

Piszemy zapytanie zwracające kategorie produktów, w których średnia cena przekracza 300, pomijając przy tym z obliczeń produkty z kategorii `elektronika` tańsze niż 50 (traktujemy je jako akcesoria, nie główny asortyment).

Odpowiedź

```
SELECT kategoria, AVG(cena) AS srednia_cena
FROM produkty
WHERE cena >= 50
GROUP BY kategoria
```

```
HAVING AVG(cena) > 300;
```

Warunek na pojedynczym produkcie (`cena >= 50`) trafia do `WHERE`, bo dotyczy pojedynczego wiersza przed grupowaniem. Warunek na średniej dla całej kategorii trafia do `HAVING`, bo `AVG(cena)` nie istnieje przed zgrupowaniem produktów.

Pytania kontrolne

1. Dlaczego `WHERE SUM(wartosc) > 1000` jest błędem składniowym, a `HAVING SUM(wartosc) > 1000` nie?

Odpowiedź

`WHERE` wykonuje się przed `GROUP BY` w logicznej kolejności zapytania, więc `SUM(wartosc)` dla grupy jeszcze nie istnieje w tym momencie. `HAVING` wykonuje się po `GROUP BY`, kiedy agregaty są już policzone.

2. Co zwróci zapytanie z HAVING, ale bez GROUP BY, jeśli warunek w HAVING nie jest spełniony?

Odpowiedź

Zero wierszy. Cała tabela liczy się wtedy jako jedna grupa, a `HAVING` odrzuca tę jedną grupę w całości, zamiast zwrócić wiersz z wartością `NULL` czy zerem.

3. Dlaczego `HAVING liczba > 1` zgłasza błąd, skoro `liczba` to nazwa kolumny widocznej w wyniku tego samego zapytania?

Odpowiedź

`liczba` to alias nadany w klauzuli `SELECT`, a `SELECT` wykonuje się logicznie po `HAVING`. W momencie sprawdzania warunku `HAVING` ten alias jeszcze nie istnieje — trzeba w to miejsce wstawić pełne wyrażenie, na przykład `COUNT(*)`.

Powiązane: SELECT: kolejność zapisu a kolejność wykonania · Funkcje agregujące i GROUP BY · WHERE: filtrowanie wierszy

Źródła: ISO/IEC 9075-2:2016 § 7.9 · dokumentacja PostgreSQL, rozdział „The HAVING Clause”

DISTINCT: i dlaczego zwykle jest objawem, nie rozwiązaniem

> `DISTINCT` usuwa zduplikowane wiersze z wyniku, ale w zapytaniu ze złączeniem duplikaty najczęściej biorą się z błędu w samym złączeniu — a wtedy naprawiamy złączenie, zamiast doklejać `DISTINCT` jako plaster.

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

TYLKO ODCZYT

Usunięcie duplikatów z jednej kolumny

```
SELECT DISTINCT status FROM zamowienia;
```

`DISTINCT` porównuje kompletne wiersze wyniku i zostawia po jednym egzemplarzu z każdej grupy identycznych. Osiem zamówień ma tylko cztery różne statusy — reszta to powtórzenia.

Przykład:

```
    status
--------------
 zrealizowane
 anulowane
 w realizacji
 nowe
(4 rows)
```

TYLKO ODCZYT

Usunięcie duplikatów po wielu kolumnach

```
SELECT DISTINCT klient_id, status FROM zamowienia ORDER BY klient_id, status;
```

Przy kilku kolumnach za duplikat uznaje się dopiero identyczną kombinację wszystkich wymienionych wartości naraz. Zamówienia 101 i 102 mają tę samą parę `(klient_id, status)` — `(1, zrealizowane)` — więc kolapsują do jednego wiersza; pozostałe kombinacje są unikalne już same z siebie.

Przykład:

```
 klient_id |    status
-----------+--------------
           | nowe
         1 | zrealizowane
         2 | anulowane
         2 | nowe
         3 | w realizacji
         4 | zrealizowane
         5 | anulowane
(7 rows)
```

TYLKO ODCZYT

Pierwszy wiersz z każdej grupy: DISTINCT ON

```
SELECT DISTINCT ON (klient_id) klient_id, id, data, wartosc
FROM zamowienia
```

```
WHERE klient_id IS NOT NULL
ORDER BY klient_id, data DESC;
```

Rozszerzenie właściwe PostgreSQL. `DISTINCT ON (klient_id)` zostawia dokładnie jeden wiersz dla każdej wartości `klient_id` — ten, który jako pierwszy pojawia się w kolejności zadanej przez `ORDER BY`. Tu `ORDER BY klient_id, data DESC` sprawia, że dla każdego klienta zostaje jego najnowsze zamówienie.

Przykład:

```
 klient_id | id |    data    | wartosc
-----------+----+------------+---------
         1 | 102| 2024-02-03 |  430.50
         2 | 105| 2024-03-11 |  150.00
         3 | 104| 2024-03-10 | 2890.00
         4 | 107| 2024-04-02 | 3200.00
         5 | 108| 2024-04-05 |    0.00
(5 rows)
```

Kolumna, po której grupujemy w `DISTINCT ON`, musi być pierwsza w `ORDER BY` — inaczej PostgreSQL zgłosi błąd, że wyrażenia `DISTINCT ON` i `ORDER BY` nie są zgodne.

□□ TYLKO ODCZYT

Deduplikacja przez GROUP BY zamiast DISTINCT

```
SELECT klient_id, status FROM zamowienia GROUP BY klient_id, status ORDER BY klient_id,
status;
```

`GROUP BY` bez żadnej funkcji agregującej daje dokładnie ten sam wynik co `DISTINCT` na tych samych kolumnach — obie konstrukcje zostawiają po jednym wierszu z każdej unikalnej kombinacji wartości. Wybór między nimi bywa kwestią czytelności: `GROUP BY` czytelniej zapowiada, że w tym samym miejscu może się pojawić agregat.

Przykład:

```
 klient_id |    status
-----------+--------------
           | nowe
         1 | zrealizowane
         2 | anulowane
```

```
          2 | nowe
          3 | w realizacji
          4 | zrealizowane
          5 | anulowane
(7 rows)
```

🔒 TYLKO ODCZYT

Sprawdzenie istnienia zamiast złączenia: EXISTS

```
SELECT nazwa, miasto
FROM klienci k
WHERE EXISTS (
    SELECT 1 FROM zamowienia z WHERE z.klient_id = k.id
);
```

Zakładamy, że do tabeli `klienci` dopisano szóstego klienta, Górski Serwis z Katowic, który nie złożył jeszcze żadnego zamówienia. `EXISTS` sprawdza dla każdego klienta, czy istnieje choć jedno pasujące zamówienie, i zwraca wiersz klienta dokładnie raz — niezależnie od tego, ile zamówień faktycznie ma. W odróżnieniu od złączenia z `zamowienia` nie ma tu mowy o duplikacji ani o potrzebie sprzątania jej przez `DISTINCT`, bo `EXISTS` nigdy nie wprowadza do wyniku kolumn z podzapytania.

Przykład:

```
           nazwa               |  miasto
-------------------------------+-----------
 Nowak i Wspólnicy Sp. z o.o.  | Warszawa
 Kowalski Handel               | Kraków
 Zielona Dolina S.A.           | Wrocław
 Baltic Trade Sp. z o.o.       | Gdańsk
 Podlaski Handel               | Białystok
(5 rows)
```

Górski Serwis nie pojawia się w wyniku — nie dlatego, że ktoś go odfiltrował `DISTINCT`em, tylko dlatego, że podzapytanie `EXISTS` dla niego nie znajduje żadnego wiersza.

🔒 TYLKO ODCZYT

Deduplikacja funkcją okna: ROW_NUMBER

```
SELECT klient_id, id, data, wartosc
FROM (
    SELECT klient_id, id, data, wartosc,
           ROW_NUMBER() OVER (PARTITION BY klient_id ORDER BY data DESC) AS
numer_w_grupie
    FROM zamowienia
    WHERE klient_id IS NOT NULL
) AS ponumerowane
WHERE numer_w_grupie = 1
ORDER BY klient_id;
```

Ten sam efekt co `DISTINCT ON`, ale przenośny między silnikami — `ROW_NUMBER()` jest częścią standardu i działa tak samo w PostgreSQL, SQL Server, Oracle i MySQL od wersji 8. Funkcje okna mają własną stronę, Funkcje okna — OVER, PARTITION BY, ROW_NUMBER, RANK, LAG/LEAD, bo mechanizm `PARTITION BY` wykracza poza temat tej strony.

Przykład:

```
 klient_id | id |    data    | wartosc
-----------+----+------------+---------
         1 | 102| 2024-02-03 |  430.50
         2 | 105| 2024-03-11 |  150.00
         3 | 104| 2024-03-10 | 2890.00
         4 | 107| 2024-04-02 | 3200.00
         5 | 108| 2024-04-05 |    0.00
(5 rows)
```

> **Uważaj**
>
> - `SELECT DISTINCT *` na szerokiej tabeli porównuje każdą kolumnę każdego wiersza z każdym innym — koszt rośnie z liczbą kolumn, nie tylko z liczbą wierszy.
> - `ORDER BY` przy `DISTINCT` musi sortować po czymś, co jest w liście `SELECT` — inaczej silnik zgłasza błąd, opisany w sekcji „Częste błędy”.

□□ **SQL** = Structured Query Language: język zapytań do relacyjnych baz danych ·**LAG** = Link Aggregation Group: grupa łączy spięta w jedno logiczne; wariant MC-LAG rozciąga ją na dwa przełączniki

Po co to istnieje

> **Pojęcie: DISTINCT.** Deduplikacja wyniku zapytania — usunięcie z wyniku wierszy, które po policzeniu wszystkich kolumn z listy `SELECT` są identyczne z którymś wcześniejszym wierszem. `DISTINCT` działa na gotowym wyniku, dokładnie w kroku szóstym logicznej kolejności wykonania opisanej na stronie SELECT: kolejność zapisu a kolejność wykonania — po tym, jak `SELECT` policzył wszystkie kolumny i wyrażenia wynikowe, a przed sortowaniem przez `ORDER BY`.

Jak to działa

Rozwiń: jak to działa pod spodem

Skąd biorą się duplikaty w zapytaniu ze złączeniem. Złączenie tabeli nadrzędnej z tabelą podrzędną, w której jednemu wierszowi nadrzędnemu odpowiada wiele wierszy podrzędnych, mnoży wiersz nadrzędny tyle razy, ile ma dopasowań — mechanizm ten szczegółowo opisuje strona Czym jest złączenie: iloczyn kartezjański i warunek, który go ogranicza. Zapytanie, które łączy `zamowienia` z `pozycje` tylko po to, żeby pokazać kolumny samego zamówienia, produkuje duplikat zamówienia za każdą pozycję, jaką to zamówienie ma.

```
SELECT z.id, z.status, z.wartosc
FROM zamowienia z
JOIN pozycje poz ON poz.zamowienie_id = z.id
ORDER BY z.id;
```

Przykład:

```
 id |    status     | wartosc
----+---------------+---------
101 | zrealizowane  | 1250.00
101 | zrealizowane  | 1250.00
102 | zrealizowane  |  430.50
102 | zrealizowane  |  430.50
104 | w realizacji  | 2890.00
104 | w realizacji  | 2890.00
```

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

Zamówienia 101, 102, 104 i 107 mają po dwie pozycje, więc każde z nich pojawia się dwukrotnie. Doklejenie `DISTINCT` do tego zapytania rzeczywiście usunie widoczne powtórzenia:

```
SELECT DISTINCT z.id, z.status, z.wartosc
FROM zamowienia z
JOIN pozycje poz ON poz.zamowienie_id = z.id
ORDER BY z.id;
```

Przykład:

```
 id |    status     | wartosc
----+---------------+---------
101 | zrealizowane  | 1250.00
102 | zrealizowane  |  430.50
104 | w realizacji  | 2890.00
107 | zrealizowane  | 3200.00
(4 rows)
```

Wygląda na naprawiony wynik, ale zamówienia 103, 105, 106 i 108 zniknęły z niego całkowicie — nie dlatego, że nie istnieją, tylko dlatego, że złączenie z `pozycje` jest złączeniem wewnętrznym, a te cztery zamówienia nie mają żadnej pozycji, więc nigdy nie trafiają do wyniku złączenia. `DISTINCT` naprawił objaw, który było widać — powtórzone wiersze — i ukrył objaw, którego widać nie było — brakujące wiersze. Prawdziwa naprawa nie potrzebuje `DISTINCT` w ogóle, bo zapytanie w ogóle nie potrzebowało `pozycje`:

```
SELECT id, status, wartosc FROM zamowienia ORDER BY id;
```

Ta wersja zwraca poprawnie wszystkie osiem zamówień, bez duplikatów i bez ukrytych braków, bo nigdy nie łączyła się z tabelą, której kolumn nawet nie używała w wyniku. Gdyby zapytanie rzeczywiście potrzebowało wiedzieć, czy zamówienie ma jakąkolwiek pozycję, właściwym narzędziem jest `EXISTS` z Fiszki tej strony, nie złączenie doprawione `DISTINCT`em.

Koszt DISTINCT. Żeby znaleźć duplikaty, silnik musi porównać każdy wiersz z każdym innym — w praktyce robi to, sortując cały wynik (żeby identyczne wiersze znalazły się obok siebie) albo budując dla niego tabelę haszującą. Obie metody kosztują dodatkową pracę proporcjonalną do rozmiaru całego wyniku, widoczną w planie wykonania jako krok `Sort` albo `HashAggregate` — strona Plan wykonania: jak go wywołać i jak go czytać pokazuje, jak ten krok

rozpoznać.

Kiedy DISTINCT jest właściwym narzędziem. Wtedy, gdy zapytanie rzeczywiście pyta o zbiór unikalnych wartości, a nie maskuje efekt uboczny złączenia — pierwsza Fiszka tej strony, lista unikalnych statusów zamówień bez żadnego złączenia w tle, jest dokładnie takim przypadkiem. `DISTINCT` jest też uzasadniony przy łączeniu wielu źródeł przez `UNION` (który domyślnie i tak usuwa duplikaty), przy raportach agregujących dane z wielu systemów, gdzie te same rekordy mogły trafić do zapytania dwoma różnymi drogami, albo po prostu przy eksploracyjnym zapytaniu w konsoli, gdzie liczy się szybka odpowiedź, nie plan wykonania.

Częste błędy

Doklejanie DISTINCT odruchowo, gdy wynik ma „za dużo” wierszy. Zamiast sprawdzić, skąd biorą się powtórzenia — czy złączenie ma właściwy warunek, czy tabela podrzędna faktycznie powinna brać udział w zapytaniu — `DISTINCT` ląduje na końcu listy `SELECT` jako pierwszy odruch. Efekt widać w sekcji „Jak to działa”: symptom znika, przyczyna zostaje, a dodatkowo znikają wiersze, które nie powinny były zniknąć.

ORDER BY po kolumnie spoza SELECT DISTINCT.

```
SELECT DISTINCT status FROM zamowienia ORDER BY wartosc;
```

Przykład:

```
ERROR:  for SELECT DISTINCT, ORDER BY expressions must appear in select list
LINE 1: SELECT DISTINCT status FROM zamowienia ORDER BY wartosc;
                                                        ^
```

`DISTINCT` deduplikuje po kolumnach z `SELECT` — `wartosc` w tym zapytaniu w ogóle nie istnieje w wyniku po deduplikacji, więc nie ma po czym sortować. Poprawka to dopisanie `wartosc` do listy `SELECT` albo usunięcie jej z `ORDER BY`.

SELECT DISTINCT * na tabeli z kolumną tekstową o zmiennej długości bez potrzeby. Porównanie całych, długich wartości tekstowych kolumna po kolumnie dla każdej pary wierszy jest dużo droższe niż porównanie samych kluczy — jeśli deduplikacja ma sens tylko względem części kolumn, `DISTINCT ON` albo `GROUP BY` na tej węższej liście kosztuje mniej.

□□ Ćwiczenia

> Zadania sprawdzające.

Ćwiczenie 1 — znalezienie przyczyny zamiast objawu

Zapytanie `SELECT DISTINCT p.nazwa, p.kategoria FROM produkty p JOIN pozycje poz ON poz.produkt_id = p.id;` ktoś napisał, żeby dostać listę produktów, które kiedykolwiek się sprzedały. Oceniamy, czy to jest właściwe użycie `DISTINCT`, czy plaster na inny problem.

Odpowiedź

To właściwe użycie. Złączenie z `pozycje` z natury rzeczy powtórzy produkt tyle razy, ile ma pozycji sprzedaży, a zapytanie świadomie pyta o zbiór unikalnych produktów, nie o listę transakcji — `DISTINCT` tu nie maskuje błędu w złączeniu, tylko realizuje właściwe pytanie. Alternatywą o tym samym znaczeniu byłoby `EXISTS` w podzapytaniu do `pozycje`, bez `JOIN` i bez `DISTINCT` w ogóle.

Pytania kontrolne

1. Dlaczego DISTINCT dodany do zapytania z Fiszki „Sprawdzenie przyczyny” usunął widoczne duplikaty, ale nie naprawił zapytania?

Odpowiedź

Bo problemem nie były duplikaty same w sobie, tylko niepotrzebne złączenie z tabelą podrzędną, które je wywołało — to samo złączenie wewnętrzne po cichu usunęło z wyniku zamówienia bez żadnej pozycji. `DISTINCT` scalił powtórzone wiersze, ale nie przywrócił brakujących.

2. Jaki krok w planie wykonania zdradza obecność DISTINCT w zapytaniu?

Odpowiedź

`Sort` albo `HashAggregate` (zależnie od tego, którą metodę wybrał optymalizator) — DISTINCT wymaga porównania całego wyniku ze sobą, więc silnik musi go najpierw posortować albo zahaszować.

Powiązane: SELECT: kolejność zapisu a kolejność wykonania · Czym jest złączenie: iloczyn kartezjański i warunek, który go ogranicza · Podzapytania: skalarne, wierszowe, skorelowane, EXISTS · Funkcje okna — OVER, PARTITION BY, ROW_NUMBER, RANK, LAG/LEAD

Źródła: ISO/IEC 9075-2:2016 § 7.12 · dokumentacja PostgreSQL, rozdział „SELECT DISTINCT”