

Lekcja 11

Temat: Zaawansowane zapytania JOIN

```
DROP TABLE IF EXISTS zamowienia;  
DROP TABLE IF EXISTS klienci;
```

```
CREATE TABLE klienci (  
  id INT PRIMARY KEY,  
  imie VARCHAR(50)  
);
```

```
CREATE TABLE zamowienia (  
  id INT PRIMARY KEY,  
  id_klienta INT ,  
  produkt VARCHAR(50),  
  FOREIGN KEY (id_klienta) REFERENCES klienci(id)  
);
```

```
INSERT INTO klienci (id, imie) VALUES  
(1, 'Anna'),  
(2, 'Jan'),  
(3, 'Ola'),  
(4, 'Piotr');
```

```
INSERT INTO zamowienia (id, id_klienta, produkt) VALUES  
(1, 1, 'Laptop'),  
(2, 1, 'Myszka'),  
(3, 2, 'Telefon'),  
(4, null, 'Monitor'); -- ten klient (id=5) nie istnieje w tabeli klienci
```

● INNER JOIN

```
SELECT k.imie, z.produkt  
FROM klienci k  
INNER JOIN zamowienia z ON k.id = z.id_klienta;
```

| imie | produkt |
|------|---------|
| Anna | Laptop |
| Anna | Myszka |
| Jan | Telefon |

LEFT JOIN

```
SELECT k.imie, z.produkt
FROM klienci k
LEFT JOIN zamowienia z ON k.id = z.id_klienta;
```

| imie | produkt |
|-------|---------|
| Anna | Laptop |
| Anna | Myszka |
| Jan | Telefon |
| Ola | NULL |
| Piotr | NULL |

RIGHT JOIN

```
SELECT k.imie, z.produkt
FROM klienci k
RIGHT JOIN zamowienia z ON k.id = z.id_klienta;
```

| imie | produkt |
|------|---------|
| Anna | Laptop |
| Anna | Myszka |
| Jan | Telefon |
| NULL | Monitor |

FULL JOIN (symulowany)

```
SELECT k.imie, z.produkt
FROM klienci k
LEFT JOIN zamowienia z ON k.id = z.id_klienta
```

UNION

```
SELECT k.imie, z.produkt
FROM klienci k
RIGHT JOIN zamowienia z ON k.id = z.id_klienta;
```

| imie | produkt |
|------|---------|
|------|---------|

| | |
|-------|---------|
| Anna | Laptop |
| Anna | Myszka |
| Jan | Telefon |
| Ola | NULL |
| Piotr | NULL |
| NULL | Monitor |

```
DROP TABLE IF EXISTS zamowienia;  
DROP TABLE IF EXISTS produkty;  
DROP TABLE IF EXISTS sklepy;
```

```
CREATE TABLE sklepy (  
  id INT PRIMARY KEY,  
  nazwa VARCHAR(50)  
);
```

```
CREATE TABLE produkty (  
  id INT PRIMARY KEY,  
  nazwa VARCHAR(50),  
  id_sklepu INT,  
  FOREIGN KEY (id_sklepu) REFERENCES sklepy(id)  
);
```

```
CREATE TABLE zamowienia (  
  id INT PRIMARY KEY,  
  id_produkту INT ,  
  ilosc INT,  
  FOREIGN KEY (id_produkту) REFERENCES produkty(id)  
);
```

```
INSERT INTO sklepy (id, nazwa) VALUES  
(1, 'Sklep A'),  
(2, 'Sklep B'),  
(3, 'Sklep C');
```

```
INSERT INTO produkty (id, nazwa, id_sklepu) VALUES  
(1, 'Laptop', 1),  
(2, 'Myszka', 1),  
(3, 'Monitor', 2),  
(4, 'Klawiatura', 3);
```

```
INSERT INTO zamowienia (id, id_produkту, ilosc) VALUES  
(1, 1, 5),  
(2, 1, 3),  
(3, 2, 10),  
(4, 3, 2);
```

Zestawienie sklepów i produktów, łącznie z tymi, dla których nie odnotowano zamówień:

```
SELECT s.nazwa AS sklep,  
       p.nazwa AS produkt,  
       COALESCE(SUM(z.ilosc), 0) AS sprzedane_sztuki  
FROM sklepy s  
LEFT JOIN produkty p ON p.id_sklepu = s.id  
LEFT JOIN zamowienia z ON z.id_produktu = p.id  
GROUP BY s.id, p.id  
ORDER BY s.id, sprzedane_sztuki DESC;
```

| sklep | produkt | sprzedane_sztuki |
|---------|------------|------------------|
| Sklep A | Myszka | 10 |
| Sklep A | Laptop | 8 |
| Sklep B | Monitor | 2 |
| Sklep C | Klawiatura | 0 |

Wyjaśnienie:

- ☐ LEFT JOIN produkty → bierzemy wszystkie sklepy, nawet jeśli nie mają produktów.
- ☐ LEFT JOIN zamowienia → bierzemy wszystkie produkty, nawet jeśli nie mają zamówień.
- ☐ SUM(z.ilosc) → sumujemy liczbę sprzedanych sztuk dla każdego produktu.
- ☐ COALESCE(..., 0) → jeśli produkt nie ma zamówień, pokazujemy 0 zamiast NULL.
- ☐ GROUP BY s.id, p.id → agregujemy dane po sklepie i produkcie.
- ☐ ORDER BY s.id, sprzedane_sztuki DESC → sortujemy dane po sklepie i liczbie sprzedanych sztuk.

Pokazuje wszystkie zamówienia, nawet jeśli nie ma dopasowanego produktu lub sklepu:

```
SELECT s.nazwa AS sklep,  
       p.nazwa AS produkt,  
       z.ilosc AS sprzedane_sztuki  
FROM sklepy s  
RIGHT JOIN produkty p ON p.id_sklepu = s.id  
RIGHT JOIN zamowienia z ON z.id_produktu = p.id  
GROUP BY s.id, p.id;
```

| sklep | produkt | sprzedane_sztuki |
|---------|---------|------------------|
| Sklep A | Laptop | 3 |
| Sklep A | Laptop | 5 |
| Sklep A | Myszka | 10 |
| Sklep B | Monitor | 2 |

Wyjaśnienie:

- ☐ **RIGHT JOIN** produkty p ON p.id_sklepu = s.id → bierzemy wszystkie produkty, nawet jeśli nie mają sklepu.
- ☐ **RIGHT JOIN** zamówienia z ON z.id_produktu = p.id → bierzemy wszystkie zamówienia, nawet jeśli nie mają przypisanego produktu.
- ☐ Jeśli w tabeli **produkty** lub **sklepy** brakuje dopasowania → kolumny będą **NULL**.

Lekcja 12

Temat: Kategorie poleceń. Procedury w MySQL



Operatory logiczne

NOT - **negacja** np.: NOT A
AND - **koniunkcja** np.: A AND B
OR - **alternatywa** np.: A OR B



Wartość NULL

Null a testy logiczne

TRUE AND NULL - zwraca **NULL**
FALSE AND NULL - zwraca **NULL**
TRUE OR NULL - zwraca **TRUE**
FALSE OR NULL - zwraca **NULL**

Podstawowe kategorie poleceń w SQL to:

- **DDL (Data Definition Language)** – definiowanie struktury bazy danych (np. tworzenie tabel).
- **DML (Data Manipulation Language)** – manipulacja danymi (np. wstawianie, aktualizacja, usuwanie).
- **DCL (Data Control Language)** – zarządzanie uprawnieniami (np. GRANT, REVOKE).
- **DQL (Data Query Language)** – pobieranie danych (np. SELECT).
- **TCL (Transaction Control Language)** – zarządzanie transakcjami (np. COMMIT, ROLLBACK).



Procedury

Procedura - nazwany ciąg instrukcji wywoływany poprzez podanie jego nazwy, wykonujący określone zadania , a następnie zwracający sterowanie do programu wywołującego



Składnia procedury:





DELIMITER //

```
CREATE PROCEDURE nazwa_procedury([parametry])
[MODIFIER]
BEGIN
    -- Deklaracje zmiennych (opcjonalne)
    DECLARE zmienna1 typ_danych;
    DECLARE zmienna2 typ_danych DEFAULT wartość;

    -- Logika programu
    -- Instrukcje SQL, pętle, warunki itp.
END //
```

DELIMITER ;

Elementy składni:

-  **DELIMITER //**: Zmienia standardowy delimiter (domyślnie ;) na inny (np. //), aby MySQL nie interpretował średnika w procedurze jako końca polecenia. Po definicji procedury przywraca się standardowy delimiter (DELIMITER ;).
-  **CREATE PROCEDURE nazwa_procedury**: Definiuje nazwę procedury, która musi być unikalna w schemacie bazy danych.
-  **[parametry]** (opcjonalne): Lista parametrów w formacie:
 - IN** nazwa_parametru typ_danych: Parametr wejściowy (przekazywany do procedury).
 - OUT** nazwa_parametru typ_danych: Parametr wyjściowy (zwracany z procedury).
 - INOUT** nazwa_parametru typ_danych: Parametr dwukierunkowy (wejściowy i wyjściowy).
-  **[MODIFIER]** (opcjonalne): Opcje, takie jak:
 - DETERMINISTIC**: Procedura zwraca ten sam wynik dla tych samych danych wejściowych. Przykład: Funkcja obliczająca kwadrat liczby (liczba * liczba) jest deterministyczna, ponieważ dla tej samej wartości wejściowej (np. 5) zawsze zwróci ten sam wynik (25).

- **NOT DETERMINISTIC**: Wynik może się różnić dla tych samych danych. Przykład: Funkcja zwracająca aktualny czas (NOW()) lub losową wartość (RAND()) jest niedeterministyczna, ponieważ wynik zależy od zewnętrznych czynników (czasu lub losowości).
- **CONTAINS SQL, NO SQL, READS SQL DATA, MODIFIES SQL DATA**: Określają, czy procedura używa lub modyfikuje dane.

CONTAINS SQL:

- ☐ Oznacza, że procedura lub funkcja **zawiera instrukcje SQL, ale nie określa, czy odczytuje, czy modyfikuje dane.**
- ☐ Jest to domyślny modyfikator, jeśli żaden inny nie zostanie wybrany.

NO SQL:

- ☐ Oznacza, że procedura lub funkcja **nie zawiera żadnych poleceń SQL** ani nie wykonuje operacji na danych w bazie.
- ☐ Używane dla procedur/funkcji, które wykonują tylko operacje na zmiennych lokalnych lub parametrach, bez odwoływania się do bazy danych.

READS SQL DATA:

- ☐ Oznacza, że procedura lub funkcja **odczytuje dane z tabel** (np. za pomocą SELECT), ale ich nie modyfikuje.

MODIFIES SQL DATA:

- ☐ Oznacza, że procedura lub funkcja **modyfikuje dane w tabelach** (np. za pomocą INSERT, UPDATE, DELETE).

5.



BEGIN ... END: Zawiera logikę procedury, w tym:

- Deklaracje zmiennych (DECLARE).
- Instrukcje SQL (np. SELECT, INSERT, UPDATE).
- Struktury sterujące (np. IF, WHILE, LOOP).

6.



Wywołanie: Procedura jest wywoływana za pomocą CALL

`nazwa_procedury(parametry);`

Usuwanie procedury

`DROP PROCEDURE` nazwa_procedury;

1.



Przykład procedury:

```
DROP PROCEDURE pokaz_hello_world;
```

```
DELIMITER //
```

```
CREATE PROCEDURE pokaz_hello_world()
```

```
BEGIN
```

```
    SELECT 'Hello World' AS wiadomosc;
```

```
END //
```

```
DELIMITER ;
```

```
CALL pokaz_hello_world();
```

2.



Przykład procedury:

```
DROP TABLE IF EXISTS pracownicy;
```

```
CREATE TABLE pracownicy(
```

```
    id INT AUTO_INCREMENT PRIMARY KEY,
```

```
    imie VARCHAR(50),
```

```
    nazwisko VARCHAR(50),
```

```
    wynagrodzenie DECIMAL(10,2)
```

```
);
```

```
INSERT INTO pracownicy (imie, nazwisko, wynagrodzenie) VALUES
```

```
('Jan', 'Kowalski', 5000.00),
```

```
('Anna', 'Nowak', 6200.00),
```

```
('Piotr', 'Zieliński', 4800.00);
```

```
DELIMITER //
```

```
CREATE PROCEDURE aktualizuj_wynagrodzenie(IN id_pracownika INT, INOUT nowe_wynagrodzenie  
DECIMAL(10,2))
```

```
BEGIN
```

```
    DECLARE stare_wynagrodzenie DECIMAL(10,2);
```

```
    SELECT wynagrodzenie INTO stare_wynagrodzenie
```

```
    FROM pracownicy
```

```
    WHERE id = id_pracownika;
```

```
    SET nowe_wynagrodzenie = stare_wynagrodzenie * 1.1;
```

```
    UPDATE pracownicy
```

```
    SET wynagrodzenie = nowe_wynagrodzenie
```

```
WHERE id = id_pracownika;  
END //
```

```
DELIMITER ;
```

```
-- Wywołanie
```



```
SET @wynagrodzenie = 1000.00;
```

```
CALL aktualizuj_wynagrodzenie(1, @wynagrodzenie);
```

```
SELECT @wynagrodzenie;
```

Wyjaśnienie: Procedura zwiększa wynagrodzenie pracownika o 10% i zwraca nowe wynagrodzenie przez parametr INOUT.

3. Przykład procedury:

```
/*
```

Zadanie 1: Procedura do klasyfikacji uczniów na podstawie średniej ocen

Opis:

Stwórz procedurę, która klasyfikuje ucznia na podstawie średniej jego ocen (np. „Słaby”, „Średni”, „Dobry”).

Procedura używa instrukcji IF do określenia kategorii i zwraca wynik przez parametr OUT.

```
*/
```

```
CREATE TABLE uczniowie (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    imie VARCHAR(50),  
    nazwisko VARCHAR(50)  
);  
  
CREATE TABLE oceny (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    id_ucznia INT,  
    ocena DECIMAL(2,1),  
    przedmiot VARCHAR(50),  
    FOREIGN KEY (id_ucznia) REFERENCES uczniowie(id)  
);
```

```
-- Wstawianie danych
```

```
INSERT INTO uczniowie (imie, nazwisko) VALUES  
( 'Jan', 'Kowalski'),  
( 'Anna', 'Nowak'),  
( 'Piotr', 'Wiśniewski');
```

```
INSERT INTO oceny (id_ucznia, ocena, przedmiot) VALUES  
(1, 4.5, 'Matematyka'),  
(1, 3.0, 'Język polski'),  
(1, 5.0, 'Fizyka'),  
(2, 2.0, 'Matematyka'),  
(2, 3.5, 'Język polski'),
```

```
(3, 4.0, 'Chemia'),  
(3, 4.5, 'Biologia');
```

```
DELIMITER //
```

```
CREATE PROCEDURE klasyfikuj_ucznia(IN id_ucznia INT, OUT kategoria VARCHAR(50))  
BEGIN
```

```
    DECLARE srednia_ocen DECIMAL(3,1);
```

```
    -- Obliczanie średniej ocen ucznia
```

```
    SELECT AVG(ocena) INTO srednia_ocen
```

```
    FROM oceny
```

```
    WHERE id_ucznia = id_ucznia;
```

```
    -- Klasyfikacja za pomocą IF
```

```
    IF srednia_ocen IS NULL THEN
```

```
        SET kategoria = 'Brak ocen';
```

```
    ELSEIF srednia_ocen < 3.0 THEN
```

```
        SET kategoria = 'Słaby';
```

```
    ELSEIF srednia_ocen >= 3.0 AND srednia_ocen < 4.5 THEN
```

```
        SET kategoria = 'Średni';
```

```
    ELSE
```

```
        SET kategoria = 'Dobry';
```

```
    END IF;
```

```
END //
```

```
DELIMITER ;
```

```
-- Testowanie procedury
```

```
SET @kategoria = '';
```

```
CALL klasyfikuj_ucznia(1, @kategoria); -- Jan Kowalski: średnia ok. 4.17
```

```
SELECT @kategoria; -- Wynik: 'Średni'
```

```
SET @kategoria = '';
```

```
CALL klasyfikuj_ucznia(2, @kategoria); -- Anna Nowak: średnia ok. 2.75
```

```
SELECT @kategoria; -- Wynik: 'Słaby'
```

```
SET @kategoria = '';
```

```
CALL klasyfikuj_ucznia(3, @kategoria); -- Piotr Wiśniewski: średnia ok. 4.25
```

```
SELECT @kategoria; -- Wynik: 'Średni'
```

```
SET @kategoria = '';
```

```
CALL klasyfikuj_ucznia(4, @kategoria); -- Nieistniejący uczeń
```

```
SELECT @kategoria; -- Wynik: 'Brak ocen'
```

Lekcja 13

Temat: Funkcję w MySQL



Procedura - nazwany ciąg instrukcji wywoływany poprzez podanie jego nazwy, wykonujący określone zadania , a następnie zwracający sterowanie do programu wywołującego



Funkcja - podobnie jak procedura z tą różnicą iż zawsze zwraca co najmniej jedną wartość określonego typu.



Składnia funkcji:


DELIMITER //

```
CREATE FUNCTION nazwa_funkcji([parametry])  
RETURNS typ_danych  
[MODIFIER]  
BEGIN  
    -- Deklaracje zmiennych (opcjonalne)  
    DECLARE zmienna1 typ_danych;  
  
    -- Logika programu  
    -- Instrukcje SQL, obliczenia  
    RETURN wartość;  
END //
```

DELIMITER ;



Elementy składni:

1.  **DELIMITER //** mówi: „kończ polecenie dopiero przy //, nie przy ;”
DELIMITER; przywraca normalne zachowanie po zakończeniu tworzenia funkcji.

Przykład:

```
BEGIN
  SET x = 10;
  RETURN x;
END;
```

W MySQL **średnik (;)** jest domyślnym **znakiem końca polecenia SQL**.
MySQL bez zmiany delimitera **pomyśli, że SET x = 10; kończy całe polecenie** i wyświetli błąd składni:

#1064 - Something is wrong in your syntax obok 'SET x = 10' w linii 2





♦ **Rozwiązanie — tymczasowa zmiana delimitera**

Zmieniasz delimiter na coś innego (np. //, \$\$, ###), żeby MySQL wiedział, że **cała funkcja kończy się dopiero tam**, gdzie Ty wskażesz.

```
DELIMITER //
```

```
CREATE FUNCTION oblicz_vat(cena DECIMAL(10,2))
RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
  DECLARE wynik DECIMAL(10,2);
  SET wynik = cena * 0.23;
  RETURN wynik;
END//
```

```
DELIMITER ;
```

2.  **CREATE FUNCTION nazwa_funkcji:** Definiuje nazwę funkcji, unikalną w schemacie.
3.  **[parametry]** (opcjonalne): Parametry wejściowe (tylko IN, bez OUT czy INOUT).
4.  **RETURNS typ_danych:** Określa typ zwracanej wartości (np. INT, VARCHAR, DECIMAL).
5.  **[MODIFIER]** (opcjonalne): Opcje, takie jak:

- **DETERMINISTIC**: Procedura zwraca ten sam wynik dla tych samych danych wejściowych. Przykład: Funkcja obliczająca kwadrat liczby (liczba * liczba) jest deterministyczna, ponieważ dla tej samej wartości wejściowej (np. 5) zawsze zwróci ten sam wynik (25).
- **NOT DETERMINISTIC**: Wynik może się różnić dla tych samych danych. Przykład: Funkcja zwracająca aktualny czas (NOW()) lub losową wartość (RAND()) jest niedeterministyczna, ponieważ wynik zależy od zewnętrznych czynników (czasu lub losowości).
- **CONTAINS SQL, NO SQL, READS SQL DATA, MODIFIES SQL DATA**: Określają, czy funkcja używa lub modyfikuje dane.

CONTAINS SQL:

- Oznacza, że procedura lub funkcja **zawiera instrukcje SQL, ale nie określa, czy odczytuje, czy modyfikuje dane**.
- Jest to domyślny modyfikator, jeśli żaden inny nie zostanie wybrany.

NO SQL:

- ☐ Oznacza, że procedura lub funkcja **nie zawiera żadnych poleceń SQL** ani nie wykonuje operacji na danych w bazie.
- ☐ Używane dla procedur/funkcji, które wykonują tylko operacje na zmiennych lokalnych lub parametrach, bez odwoływania się do bazy danych.

READS SQL DATA:

- ☐ Oznacza, że procedura lub funkcja **odczytuje dane z tabel** (np. za pomocą SELECT), ale ich nie modyfikuje.

MODIFIES SQL DATA:

- ☐ Oznacza, że procedura lub funkcja **modyfikuje dane w tabelach** (np. za pomocą INSERT, UPDATE, DELETE).

6.  **BEGIN ... END**: Zawiera logikę funkcji, w tym:

- Deklaracje zmiennych (DECLARE).
- Instrukcje SQL i obliczenia.
- Obowiązkowe RETURN wartość zwracającą pojedynczą wartość.

7.  **Wywołanie**: Funkcję wywołuje się w wyrażeniach SQL, np. SELECT nazwa_funkcji(parametry);.

Przykład funkcji:

DELIMITER //

```

CREATE FUNCTION oblicz_vat(kwota DECIMAL(10,2), stawka DECIMAL(4,2))
RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
    DECLARE podatek DECIMAL(10,2);
    SET podatek = kwota * stawka;
    RETURN podatek;
END //

```

```

DELIMITER ;

```

```

-- Wywołanie
SELECT oblicz_vat(100.00, 0.23) AS podatek; -- Zwraca 23.00

```

Instrukcja IF

Składnia:

```

IF warunek THEN
    -- instrukcje, jeśli warunek jest prawdziwy
[ELSEIF warunek THEN
    -- instrukcje dla dodatkowego warunku]
[ELSE
    -- instrukcje, jeśli żaden warunek nie jest prawdziwy]
END IF;

```

Przykład w procedurze:

```

DELIMITER //

CREATE PROCEDURE sprawdz_wiek(IN id_ucznia INT, OUT komunikat VARCHAR(100))
BEGIN
    DECLARE wiek INT;

    SELECT wiek INTO wiek FROM uczniowie WHERE id = id_ucznia;

    IF wiek < 18 THEN
        SET komunikat = 'Uczeń jest niepełnoletni';
    ELSEIF wiek >= 18 AND wiek < 21 THEN
        SET komunikat = 'Uczeń jest pełnoletni, ale poniżej 21 lat';
    ELSE
        SET komunikat = 'Uczeń ma 21 lat lub więcej';
    END IF;
END //

```

```
END //
```

```
DELIMITER ;
```

```
-- Wywołanie
```



```
SET @komunikat = "";  
CALL sprawdz_wiek(1, @komunikat);  
SELECT @komunikat;
```

Przykład w funkcji:

```
DELIMITER //
```

```
CREATE FUNCTION kategoria_wieku(wiek INT)  
RETURNS VARCHAR(50)  
DETERMINISTIC  
BEGIN  
    DECLARE komunikat VARCHAR(50);  
  
    IF wiek < 18 THEN  
        SET komunikat = 'Niepełnoletni';  
    ELSEIF wiek >= 18 AND wiek < 21 THEN  
        SET komunikat = 'Młody dorosły';  
    ELSE  
        SET komunikat = 'Dorosły';  
    END IF;  
  
    RETURN komunikat;  
END //
```

```
DELIMITER ;
```

```
-- Wywołanie
```



```
SELECT kategoria_wieku(20) AS kategoria;
```

Instrukcja CASE

Składnia (wyszukująca forma):

```
CASE
```

```

WHEN warunek1 THEN
    -- instrukcje
WHEN warunek2 THEN
    -- instrukcje
[ELSE
    -- instrukcje, jeśli żaden warunek nie jest prawdziwy]
END CASE;

```

Przykład w procedurze (prosta forma):

DELIMITER //

```

CREATE PROCEDURE ocen_uczniow(IN ocena INT, OUT komunikat VARCHAR(100))
BEGIN
    CASE ocena
        WHEN 1 THEN
            SET komunikat = 'Niedostateczny';
        WHEN 2 THEN
            SET komunikat = 'Dopuszczający';
        WHEN 3 THEN
            SET komunikat = 'Dostateczny';
        WHEN 4 THEN
            SET komunikat = 'Dobry';
        WHEN 5 THEN
            SET komunikat = 'Bardzo dobry';
        WHEN 6 THEN
            SET komunikat = 'Celujący';
        ELSE
            SET komunikat = 'Nieprawidłowa ocena';
        END CASE;
    END //

```

DELIMITER ;

```

-- Wywołanie
SET @komunikat = "";
CALL ocen_uczniow(4, @komunikat);
SELECT @komunikat;

```

Przykład w funkcji (wyszukująca forma):

DELIMITER //

```

CREATE FUNCTION kategoria_oceny(ocena INT)

```

```

RETURNS VARCHAR(50)
DETERMINISTIC
BEGIN
    DECLARE komunikat VARCHAR(50);

    CASE
        WHEN ocena = 1 THEN
            SET komunikat = 'Niedostateczny';
        WHEN ocena = 2 THEN
            SET komunikat = 'Dopuszczający';
        WHEN ocena BETWEEN 3 AND 4 THEN
            SET komunikat = 'Średni';
        WHEN ocena = 5 THEN
            SET komunikat = 'Dobry';
        WHEN ocena = 6 THEN
            SET komunikat = 'Celujący';
        ELSE
            SET komunikat = 'Nieprawidłowa ocena';
    END CASE;

    RETURN komunikat;
END //

DELIMITER ;

-- Wywołanie

```



```
SELECT kategoria_oceny(3) AS kategoria;
```

Błędy w programach:

☐ Składniowe

- ☐ spowodowane użyciem niewłaściwego polecenia przez programistę
- ☐ wykrywane automatycznie

☐ Logiczne

- ☐ program wykonuje się lecz rezultaty jego działania są dalekie od oczekiwań
- ☐ wykrywane przez programistę/testera/użytkownika końcowego

Lekcja 14



Temat: Wyzwalacze (triggery) w MySQL

Definicja:

Wyzwalacz (trigger) w MySQL to specjalny rodzaj procedury składowanej, która jest automatycznie wywoływana w odpowiedzi na określone zdarzenia w tabeli, takie jak wstawianie (INSERT**), aktualizacja (**UPDATE**) lub usuwanie (**DELETE**) danych.** Wyzwalacze służą do automatycznego wykonywania operacji w bazie danych, np. do zapewnienia spójności danych, logowania zmian czy automatycznego wypełniania pól.

Rodzaje wyzwalaczy w MySQL

Wyzwalacze w MySQL można podzielić na podstawie dwóch kryteriów: **czasu wywołania** i **zdarzenia**, na które reagują.

1.  **Czas wywołania:**
 - **BEFORE:** Wyzwalacz jest uruchamiany przed wykonaniem operacji (np. przed wstawieniem rekordu).
 - **AFTER:** Wyzwalacz jest uruchamiany po wykonaniu operacji (np. po wstawieniu rekordu).
2.  **Zdarzenia:**
 - **INSERT:** Wyzwalacz **reaguje na wstawienie nowego rekordu do tabeli.**
 - **UPDATE:** Wyzwalacz **reaguje na aktualizację istniejącego rekordu.**
 - **DELETE:** Wyzwalacz **reaguje na usunięcie rekordu z tabeli.**

W efekcie można stworzyć sześć kombinacji wyzwalaczy:

- BEFORE INSERT
- AFTER INSERT
- BEFORE UPDATE
- AFTER UPDATE
- BEFORE DELETE
- AFTER DELETE



Składnia wyzwalacza w MySQL

```
CREATE TRIGGER nazwa_wyzwalacza
[BEFORE | AFTER] [INSERT | UPDATE | DELETE]
ON nazwa_tabeli
FOR EACH ROW
BEGIN
    -- Kod wyzwalacza (operacje do wykonania)
END;
```

- **nazwa_wyzwalacza:** Unikalna nazwa wyzwalacza.
- **nazwa_tabeli:** Tabela, do której wyzwalacz jest przypisany.
- **FOR EACH ROW:** Wyzwalacz jest wykonywany dla każdego wiersza, który podlega operacji.
- Wewnątrz wyzwalacza można używać słów kluczowych NEW (dla nowych danych w INSERT i UPDATE) oraz OLD (dla starych danych w UPDATE i DELETE).



Przykłady zastosowania wyzwalaczy

1. Automatyczne logowanie zmian w tabeli (AFTER UPDATE)

Cel: Rejestrowanie zmian w kolumnie cena w tabeli produkty w osobnej tabeli log_zmian.

Struktura tabel:

```
CREATE TABLE produkty (
    id INT PRIMARY KEY,
    nazwa VARCHAR(100),
    cena DECIMAL(10,2)
);
```

```
CREATE TABLE log_zmian (
    id INT AUTO_INCREMENT PRIMARY KEY,
    produkt_id INT,
    stara_cena DECIMAL(10,2),
    nowa_cena DECIMAL(10,2),
    data_zmiany TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
INSERT INTO produkty (id, nazwa, cena) VALUES (1, "Telefon", 222);
```

Wyzwalacz:

```
DELIMITER //
CREATE TRIGGER log_zmiana_ceny
AFTER UPDATE ON produkty
FOR EACH ROW
BEGIN
    IF OLD.cena != NEW.cena THEN
        INSERT INTO log_zmian (produkt_id, stara_cena, nowa_cena)
        VALUES (OLD.id, OLD.cena, NEW.cena);
    END IF;
END //
DELIMITER ;
```

Działanie:

- Po każdej aktualizacji ceny w tabeli produkty, wyzwalacz zapisuje stary i nowy poziom ceny w tabeli log_zmian.
- Przykład: Jeśli zmienimy cenę produktu o ID 1 z 100.00 na 120.00, w tabeli log_zmian pojawi się nowy rekord z tymi wartościami.

**Test:**

```
UPDATE produkty SET cena = 120.00 WHERE id = 1;
SELECT * FROM log_zmian;
```

**2. Automatyczne ustawianie daty modyfikacji (BEFORE UPDATE)**

Cel: Automatyczne ustawianie kolumny data_modyfikacji na aktualną datę i godzinę przy każdej aktualizacji rekordu.

Struktura tabeli:

```
CREATE TABLE klienci (
  id INT PRIMARY KEY,
  imie VARCHAR(50),
  data_modyfikacji TIMESTAMP
);
INSERT INTO klienci (id, imie, data_modyfikacji) VALUES (1, 'Anna', NOW());
```

Wyzwalacz:

```
DELIMITER //
CREATE TRIGGER aktualizuj_date
BEFORE UPDATE ON klienci
FOR EACH ROW
BEGIN
  SET NEW.data_modyfikacji = CURRENT_TIMESTAMP;
END;
DELIMITER ;
```

Działanie:

- Przed każdą aktualizacją rekordu w tabeli klienci, wyzwalacz ustawia wartość kolumny data_modyfikacji na bieżącą datę i godzinę.

**Test:**

```
UPDATE klienci SET imie = 'Jan' WHERE id = 1;
SELECT * FROM klienci;
```

**3. Zapobieganie usuwaniu rekordów (BEFORE DELETE)**

Cel: Uniemożliwienie usuwania rekordów z tabeli zamowienia, jeśli mają status "zrealizowane".

Struktura tabeli:

```
CREATE TABLE zamowienia (
```

```
id INT PRIMARY KEY,  
status VARCHAR(20)  
);
```

Wyzwalacz:

```
DELIMITER //  
CREATE TRIGGER zapobiegaj_usuniecieu  
BEFORE DELETE ON zamowienia  
FOR EACH ROW  
BEGIN  
    IF OLD.status = 'zrealizowane' THEN  
        SIGNAL SQLSTATE '45000';  
        SET MESSAGE_TEXT = 'Nie można usunąć zrealizowanego zamówienia!';  
    END IF;  
END;  
DELIMITER ;
```

Działanie:

- Jeśli spróbujemy usunąć rekord, którego status to "zrealizowane", wyzwalacz zgłosi błąd i zablokuje operację.



Test:

```
DELETE FROM zamowienia WHERE id = 1; -- Błąd, jeśli status = 'zrealizowane'
```

Uwagi i ograniczenia



1. **Brak wyzwalaczy dla SELECT:** MySQL nie obsługuje wyzwalaczy dla operacji odczytu.
2. **Unikanie rekurencji:** Wyzwalacz nie powinien modyfikować tej samej tabeli, na której działa, aby uniknąć pętli (chyba że jest to kontrolowane).
3. **Debugowanie:** Wyzwalacze mogą być trudne do debugowania, więc warto logować działania do osobnej tabeli.
4. **Wydajność:** Nadmierne użycie wyzwalaczy może spowolnić operacje na bazie danych.

Podsumowanie

Wyzwalacze w MySQL są potężnym narzędziem do automatyzacji i zapewnienia spójności danych. Mogą być używane do logowania, walidacji danych, automatycznego wypełniania pól czy zapobiegania niepożądanym operacjom. Kluczowe jest rozważne ich stosowanie, aby nie skomplikować logiki bazy danych.

Lekcja

Temat: Sesja w MySQL. Transakcje w MySQL

Sesja to połączenie klienta z serwerem MySQL, które trwa od momentu zalogowania się do bazy (np. przez `mysql -u root -p` lub przez aplikację) aż do momentu, gdy to połączenie zostanie **zamknięte**.

W jednej sesji użytkownik może uruchamiać wiele transakcji, jedna po drugiej — ale tylko jedną naraz.

✖ Przykład — poprawny przebieg w jednej sesji:

```
-- sesja 1
START TRANSACTION;

UPDATE konto SET saldo = saldo - 100 WHERE id = 1;
UPDATE konto SET saldo = saldo + 100 WHERE id = 2;
COMMIT; -- kończymy pierwszą transakcję

-- teraz możemy rozpocząć drugą
START TRANSACTION;
DELETE FROM historia WHERE data < '2024-01-01';
COMMIT;
```

● Tutaj wszystko jest OK — transakcje wykonywane jedna po drugiej.

Transakcja to zestaw kilku poleceń SQL (np. INSERT, UPDATE, DELETE), które są wykonywane jako jedna całość.

Czyli:

albo wszystkie operacje się udają (zostają zapisane w bazie),
albo żadna z nich — jeśli coś pójdzie nie tak (wszystko się cofa).



♦ Podstawowe polecenia transakcyjne:

| | |
|-----------------------------------|---|
| START TRANSACTION; - | rozpoczyna transakcję |
| COMMIT; - | zatwierdza wszystkie zmiany |
| ROLLBACK; - | cofnięcie wszystkich zmian do początku transakcji |
| SAVEPOINT nazwa; - | tworzy punkt przywracania transakcji |
| ROLLBACK TO nazwa; - | cofnięcie zmian tylko do danego punktu |
| RELEASE SAVEPOINT nazwa; - | usuwa punkt przywracania |

✖ Przykład podstawowej transakcji

```
CREATE TABLE konto (  
  id INT PRIMARY KEY,
```

```
    imie VARCHAR(50),  
    saldo DECIMAL(10,2)  
);
```

INSERT INTO konto VALUES

```
(1, 'Adam', 1000.00),  
(2, 'Beata', 2000.00);
```



♦ Przykład 1 – przelew między kontami:

```
START TRANSACTION;
```

```
UPDATE konto SET saldo = saldo - 200 WHERE id = 1; -- Adam traci 200 zł  
UPDATE konto SET saldo = saldo + 200 WHERE id = 2; -- Beata dostaje 200 zł
```

```
COMMIT; -- Zatwierdzenie zmian
```

➡ Jeśli **wszystko się uda**, zmiany zostaną na stałe zapisane w bazie.



♦ Przykład 2 – błąd w trakcie transakcji

```
START TRANSACTION;
```

```
UPDATE konto SET saldo = saldo - 200 WHERE id = 1; -- Adam traci 200 zł  
UPDATE konto SET saldo = saldo + 200 WHERE id = 99; -- ❌ konto 99 nie istnieje
```

```
ROLLBACK; -- cofnięcie wszystkich zmian
```

➡ W efekcie **Adam nie traci 200 zł**, bo cała transakcja zostaje cofnięta.
To jest **bezpieczeństwo danych** – nic się nie "rozjedzie".

✂ SAVEPOINT — punkt przywracania

Czasem chcesz cofnąć **tylko część transakcji**, a nie całość.



♦ Przykład 3 – użycie SAVEPOINT:

```
START TRANSACTION;
```

```
UPDATE konto SET saldo = saldo - 100 WHERE id = 1;  
SAVEPOINT po_pierwszej_operacji;
```

```
UPDATE konto SET saldo = saldo - 500 WHERE id = 2;  
ROLLBACK TO po_pierwszej_operacji; -- Cofamy tylko drugą zmianę
```

COMMIT; -- Zatwierdzamy pierwszą zmianę

➡ W efekcie:

- Adamowi zabrano 100 zł ✓
- Druga operacja (z konta 2) została cofnięta ✗



✚ **RELEASE usunięcie SAVEPOINT ;**

RELEASE SAVEPOINT po_pierwszej_operacji;

Lekcja

Temat: Tabela tymczasowa w MySQL

Tabela tymczasowa (temporary table) w MySQL to specjalny rodzaj tabeli, która istnieje tylko w ramach bieżącej sesji połączenia z bazą danych. **Jest ona automatycznie usuwana po zakończeniu sesji** (np. po rozłączeniu się z serwerem MySQL).

Tabele tymczasowe są przydatne do przechowywania pośrednich wyników zapytań, przetwarzania danych tymczasowo lub unikania konfliktów z trwałymi tabelami. Są widoczne tylko dla użytkownika, który je utworzył, i nie wpływają na inne sesje.

Polecenie tworzące tabele tymczasową:

CREATE TEMPORARY TABLE

♦ Tworzenie tabeli tymczasowej z wyniku zapytania

CREATE TEMPORARY TABLE nazwa_tabeli_tymczasowej **AS**
SELECT * FROM nazwa_tabeli;

♦ Tworzenie tabeli tymczasowej

Składnia jest prawie taka sama jak dla zwykłej tabeli, z dodatkiem słowa kluczowego **TEMPORARY**

CREATE TEMPORARY TABLE nazwa_tabeli_tymczasowej (
kolumna1 typ_danych [opcje],
kolumna2 typ_danych [opcje],
-- itd.

);

Polecenie usuwające tabele tymczasową:

```
DROP TEMPORARY TABLE IF EXISTS temp_tab;
```

Kiedy używać tabel tymczasowych?

- ☐ Do przetwarzania dużych zbiorów danych w złożonych zapytaniach (np. w procedurach składowanych).
- ☐ Do tymczasowego przechowywania wyników podzapytań.
- ☐ W raportach lub analizach, gdzie nie chcesz modyfikować stałych tabel.
- ☐ Aby uniknąć blokad w wieloużytkownikowych środowiskach.

Ograniczenia

- Nie można tworzyć indeksów pełnotekstowych ani widoków na tabelach tymczasowych.
- W niektórych silnikach (np. InnoDB) mogą być wolniejsze dla bardzo dużych zbiorów.
- Jeśli sesja się zakończy nieoczekiwanie, tabela zniknie.

Uwaga: jeśli utworzysz tymczasową tabelę o takiej samej nazwie jak istniejąca stała tabela, MySQL **będzie używać wersji tymczasowej** w danej sesji. Po jej usunięciu znów zobaczysz oryginalną tabelę.

♦ Widoczność i izolacja

- Tabela tymczasowa jest **widoczna tylko w ramach bieżącego połączenia**.
- **Inne sesje** (nawet ten sam użytkownik) **nie mają do niej dostępu**.
- Dzięki temu nie musisz martwić się o kolizje nazw między użytkownikami lub zapytaniami.

♦ Wydajność i miejsce przechowywania

- MySQL tworzy tymczasowe tabele w **pamięci RAM (MEMORY)** lub **na dysku (InnoDB / MyISAM)** – zależnie od ich rozmiaru i typu danych.
- Dla małych zbiorów danych (bez kolumn typu **TEXT** czy **BLOB**) tabela będzie w pamięci.
- Gdy przekroczy limit **tmp_table_size** lub **max_heap_table_size**, MySQL **przeniesie ją automatycznie na dysk**.

♦ Indeksy i klucze

Możesz w tabelach tymczasowych:

definiować **PRIMARY KEY, UNIQUE, INDEX** itp.

```
CREATE TEMPORARY TABLE produkty_tmp (  
  id INT PRIMARY KEY AUTO_INCREMENT,  
  nazwa VARCHAR(100),  
  cena DECIMAL(10,2),  
  INDEX (cena)  
);
```

- Jednak nie możesz tworzyć **indeksów FULLTEXT** ani **SPATIAL** w tabelach tymczasowych.

♦ Typowe zastosowania

- ✓ **Przechowywanie wyników pośrednich** — np. podczas tworzenia raportów lub obliczeń.
- ✓ **Łączenie dużych danych etapami** (np. przez JOIN-y z danymi wstępnie przefiltrowanymi).
- ✓ **Przyspieszanie złożonych zapytań** (zamiast tworzyć tymczasowe widoki).
- ✓ **Izolacja danych dla konkretnego użytkownika lub procesu** — szczególnie przy analizie danych sesyjnych.
- ✓ **Wielokrotne użycie danych w obrębie jednej transakcji** bez konieczności ponownego zapytania do głównej tabeli.

♦ Ograniczenia

- ⚠ **Brak replikacji:** dane z tabel tymczasowych nie są replikowane między serwerami Master–Slave.
- ⚠ **Brak trwałości:** po restarcie serwera MySQL tabele tymczasowe znikają.
- ⚠ **Nie można używać ALTER TABLE** do zmiany struktury w niektórych wersjach MySQL.
- ⚠ **Uważaj na nazwy:** jeśli zapomnisz o **TEMPORARY**, możesz nadpisać istniejącą tabelę trwałą o tej samej nazwie.

Lekcja

Temat: Group by

GROUP BY w **MySQL** służy do **grupowania rekordów**, które mają te same wartości w określonych kolumnach. Zazwyczaj używa się go **razem z funkcjami agregującymi**, takimi jak:

- **COUNT()** – zlicza ilość rekordów,
- **SUM()** – sumuje wartości,
- **AVG()** – liczy średnią,
- **MAX()** – zwraca wartość maksymalną,
- **MIN()** – zwraca wartość minimalną.

♦ Przykład praktyczny

```
CREATE TABLE orders (  
  id INT AUTO_INCREMENT PRIMARY KEY,  
  customer_name VARCHAR(50),  
  product_name VARCHAR(50),  
  quantity INT,  
  price DECIMAL(10, 2)  
);
```

```
INSERT INTO orders (customer_name, product_name, quantity, price)  
VALUES  
( 'Jan Kowalski', 'Laptop', 1, 3500.00),  
( 'Anna Nowak', 'Mysz', 2, 50.00),  
( 'Jan Kowalski', 'Mysz', 1, 50.00),  
( 'Piotr Wiśniewski', 'Klawiatura', 1, 120.00),  
( 'Anna Nowak', 'Laptop', 1, 3400.00),  
( 'Jan Kowalski', 'Laptop', 1, 3500.00);
```

♦ **Przykład 1 – Suma wartości zamówień dla każdego klienta**

```
SELECT customer_name, SUM(price * quantity) AS total_spent  
FROM orders  
GROUP BY customer_name;
```

♦ **Przykład 2 – Ile produktów kupił każdy klient**

```
SELECT customer_name, COUNT(*) AS total_orders  
FROM orders  
GROUP BY customer_name;
```

♦ **Przykład 3 – Średnia cena produktów kupionych przez każdego klienta**

```
SELECT customer_name, AVG(price) AS avg_price  
FROM orders  
GROUP BY customer_name;
```

♦ **Przykład 4 – Grupowanie po dwóch kolumnach (klient + produkt)**

```
SELECT customer_name, product_name, SUM(quantity) AS total_quantity  
FROM orders  
GROUP BY customer_name, product_name;
```

♦ **Przykład 5 GROUP BY z HAVING**

Założmy, że chcemy zobaczyć **tylko tych klientów**, którzy **wydali łącznie więcej niż 1000 zł**.

```
SELECT customer_name, SUM(price * quantity) AS total_spent  
FROM orders  
GROUP BY customer_name
```

HAVING SUM(price * quantity) > 1000;

♦ Przykład 6 filtracja po liczbie zamówień

Chcemy zobaczyć klientów, którzy złożyli więcej niż jedno zamówienie:

```
SELECT customer_name, COUNT(*) AS total_orders
FROM orders
GROUP BY customer_name
HAVING COUNT(*) > 1;
```

♦ Różnica między WHERE a HAVING

- WHERE filtruje **pojedyncze rekordy przed grupowaniem**,
- HAVING filtruje **całe grupy po agregacji**.

📌 Przykład błędu:

-- ❌ Nie zadziała:

```
SELECT customer_name, SUM(price)
FROM orders
WHERE SUM(price) > 1000
GROUP BY customer_name;
```

📌 Poprawnie:

```
SELECT customer_name, SUM(price)
FROM orders
GROUP BY customer_name
HAVING SUM(price) > 1000;
```

♦ Podsumowanie

- GROUP BY **grupuje** dane na podstawie wartości w kolumnach.
- Zazwyczaj używa się go z **funkcjami agregującymi** (SUM, COUNT, AVG, itp.).
- Może grupować po **jednej lub wielu kolumnach**.
- Często łączy się z HAVING, aby filtrować wyniki po agregacji (np. „pokaż tylko klientów, którzy wydali więcej niż 1000 zł”).

Lekcja

Temat: Having, funkcje agregujące. Przykłady zapytań z datami, kwartalami i czasem

```
CREATE TABLE zamowienia (  
  id INT AUTO_INCREMENT PRIMARY KEY,  
  id_produktu INT NOT NULL,  
  id_klienta INT NOT NULL,  
  ilosc INT NOT NULL,  
  kwota DECIMAL(10,2) NOT NULL,  
  data_zamowienia DATE NOT NULL,  
  status ENUM('oczekujące', 'zrealizowane', 'anulowane')  
);
```

```
INSERT INTO zamowienia (id_produktu, id_klienta, ilosc, kwota, data_zamowienia, status)  
VALUES  
(1, 1, 2, 200.00, '2025-04-01', 'zrealizowane'),  
(1, 1, 1, 200.00, '2025-05-01', 'zrealizowane'),  
(2, 1, 5, 300.00, '2025-10-05', 'oczekujące'),  
(3, 2, 3, 400.00, '2025-10-06', 'zrealizowane'),  
(3, 2, 1, 400.00, '2025-09-15', 'oczekujące'),  
(3, 2, 2, 400.50, '2025-11-05', 'anulowane'),  
(4, 3, 3, 600.00, '2025-10-07', 'zrealizowane'),  
(4, 3, 1, 250.00, '2025-11-02', 'anulowane');
```

HAVING to słowo kluczowe w MySQL, które często bywa mylone z WHERE.

W skrócie:

- ➡ **WHERE** filtruje pojedyncze wiersze przed grupowaniem,
- ➡ **HAVING** filtruje całe grupy po wykonaniu **GROUP BY**.

♦ Składnia

```
SELECT kolumna, funkcja_agregująca(...)  
FROM tabela  
[WHERE warunek]  
GROUP BY kolumna  
HAVING warunek_na_grupie;
```

Różnica między WHERE a HAVING

| Etap | Kiedy działa | Co filtruje |
|---------------|---------------------------------------|---------------------|
| WHERE | Przed grupowaniem (GROUP BY) | Pojedyncze wiersze |
| HAVING | Po grupowaniu | Całe grupy wynikowe |

Krok po kroku

1. Na początku chcesz zobaczyć sumę zamówień każdego klienta

```
SELECT id_klienta, SUM(kwota) AS suma_zamowien
FROM zamowienia
GROUP BY id_klienta;
```

| id_klienta | suma_zamowien |
|------------|---------------|
| 1 | 700.00 |
| 2 | 1200.50 |
| 3 | 850.00 |

2. Teraz chcesz tylko klientów, którzy wydali więcej niż 300 zł

```
SELECT id_klienta, SUM(kwota) AS suma_zamowien
FROM zamowienia
GROUP BY id_klienta
HAVING SUM(kwota) > 300;
```

| id_klienta | suma_zamowien |
|------------|---------------|
| 1 | 700.00 |
| 2 | 1200.50 |
| 3 | 850.00 |

Można używać HAVING bez GROUP BY

Jeśli nie masz **GROUP BY**, **HAVING** może nadal działać, ale wtedy traktuje cały zestaw wyników jako jedną grupę.

```
SELECT SUM(kwota) AS suma
FROM zamowienia
HAVING SUM(kwota) > 1000;
```

| suma |
|---------|
| 2750.50 |

Funkcje agregujące

1. **SUM()** z warunkiem i **GROUP BY**

Suma wartości zamówień (ilość * kwota) dla każdego klienta, tylko dla zamówień „zrealizowanych”.

```
SELECT id_klienta,  
       SUM(ilość * kwota) AS łączna_kwota  
FROM zamówienia  
WHERE status = 'zrealizowane'  
GROUP BY id_klienta;
```

| id_klienta | łączna_kwota |
|------------|--------------|
| 1 | 600.00 |
| 2 | 1200.00 |
| 3 | 1800.00 |

2. AVG() + ROUND()

Średnia wartość pojedynczego zamówienia w zaokrągleniu do 2 miejsc po przecinku.

```
SELECT id_klienta,  
       ROUND(AVG(ilość * kwota),2) AS srednia_wartosc_zamowienia  
FROM zamówienia  
GROUP BY id_klienta;
```

| id_klienta | srednia_wartosc_zamowienia |
|------------|----------------------------|
| 1 | 700.00 |
| 2 | 800.33 |
| 3 | 1025.00 |

3. COUNT(DISTINCT ...)

Ile różnych produktów zamówił każdy klient.

```
SELECT id_klienta,  
       COUNT(DISTINCT id_produktu) AS unikalne_produkty  
FROM zamówienia  
GROUP BY id_klienta;
```

| id_klienta | unikalne_produkty |
|------------|-------------------|
| 1 | 2 |

| | |
|---|---|
| 2 | 1 |
| 3 | 1 |

4. **MIN()** i **MAX()** z datami

Najstarsze i najnowsze zamówienie dla każdego klienta.

```
SELECT id_klienta,
       MIN(data_zamowienia) AS pierwsze_zamowienie,
       MAX(data_zamowienia) AS ostatnie_zamowienie
FROM zamowienia
GROUP BY id_klienta;
```

| id_klienta | pierwsze_zamowienie | ostatnie_zamowienie |
|------------|---------------------|---------------------|
| 1 | 2025-04-01 | 2025-10-05 |
| 2 | 2025-09-15 | 2025-11-05 |
| 3 | 2025-10-07 | 2025-11-02 |

5. **GROUP_CONCAT()** 🌟 (często pojawia się na egzaminie!)

Wypisanie wszystkich statusów zamówień dla każdego klienta w jednej kolumnie.

```
SELECT id_klienta,
       GROUP_CONCAT(DISTINCT status ORDER BY status SEPARATOR ', ') AS statusy
FROM zamowienia
GROUP BY id_klienta;
```

| id_klienta | unikalne_produkty |
|------------|-------------------------------------|
| 1 | oczekujące, zrealizowane |
| 2 | oczekujące, zrealizowane, anulowane |
| 3 | zrealizowane, anulowane |

Podzapytanie z agregacją

Klient, który wydał najwięcej pieniędzy łącznie.

```

SELECT id_klienta, SUM(ilosc * kwota) AS suma
FROM zamowienia
GROUP BY id_klienta
HAVING suma = (
    SELECT MAX(suma_kwot)
    FROM (
        SELECT SUM(ilosc * kwota) AS suma_kwot
        FROM zamowienia
        GROUP BY id_klienta
    ) AS t
);

```

| id_klienta | suma |
|------------|---------|
| 2 | 2401.00 |

rozkładamy na czynniki

```

SELECT SUM(ilosc * kwota) AS suma_kwot
FROM zamowienia
GROUP BY id_klienta

```

Klient 1:

$(2 * 200.00) + (1 * 200.00) + (5 * 300.00)$
 $= 400 + 200 + 1500$
 $= 2100.00$

Klient 2:

$(3 * 400.00) + (1 * 400.00) + (2 * 400.50)$
 $= 1200 + 400 + 801$
 $= 2401.00$

Klient 3:

$(3 * 600.00) + (1 * 250.00)$
 $= 1800 + 250$
 $= 2050.00$

| id_klienta | suma_kwot |
|------------|-----------|
| 1 | 2100 |
| 2 | 2401 |
| 3 | 2050 |

Teraz wybieramy największą wartość:

```

SELECT MAX(suma_kwot)
FROM (

```

```
SELECT SUM(ilosc * kwota) AS suma_kwot
FROM zamowienia
GROUP BY id_klienta
) AS t
```

Następnie pokaż tylko tych klientów, których łączna suma = największej sumie z całej tabeli.

Przykłady zapytań z datami, kwartałami i czasem

1. Zamówienia z ostatniego miesiąca

Pokazuje wszystkie zamówienia z ostatnich 30 dni względem bieżącej daty (**CURDATE()**):

```
SELECT *
FROM zamowienia
WHERE data_zamowienia >= DATE_SUB(CURDATE(), INTERVAL 1 MONTH);
```

2. Suma wartości zamówień w każdym kwartale

To klasyczne zapytanie egzaminacyjne.

```
SELECT
    YEAR(data_zamowienia) AS rok,
    QUARTER(data_zamowienia) AS kwartal,
    SUM(ilosc * kwota) AS suma_kwartalu
FROM zamowienia
GROUP BY rok, kwartal
ORDER BY rok, kwartal;
```

3. Liczba zamówień według miesiąca

Często spotykane na INF.03: raport miesięczny.

```
SELECT
    YEAR(data_zamowienia) AS rok,
    MONTH(data_zamowienia) AS miesiac,
    COUNT(*) AS liczba_zamowien
FROM zamowienia
GROUP BY rok, miesiac
ORDER BY rok, miesiac;
```

4. Zamówienia, które miały miejsce więcej niż 2 miesiące temu

Dobre na testy z **DATE_SUB()**:

```
SELECT *
FROM zamowienia
WHERE data_zamowienia < DATE_SUB(CURDATE(), INTERVAL 2 MONTH);
```

5. Zamówienia z bieżącego kwartału

Egzaminowe pytanie: „Wyświetl wszystkie zamówienia z bieżącego kwartału”

```
SELECT *
FROM zamowienia
WHERE QUARTER(data_zamowienia) = QUARTER(CURDATE())
AND YEAR(data_zamowienia) = YEAR(CURDATE());
```

6. Łączna wartość zamówień w każdym kwartale

„Podaj sumę wartości wszystkich zamówień w poszczególnych kwartałach 2025 roku.”

```
SELECT
    QUARTER(data_zamowienia) AS kwartal,
    ROUND(SUM(ilosc * kwota), 2) AS wartosc_zamowien
FROM zamowienia
WHERE YEAR(data_zamowienia) = 2025
GROUP BY kwartal
ORDER BY kwartal;
```

7. Średnia wartość zamówienia w każdym miesiącu

„Wyznacz średnią wartość zamówienia dla każdego miesiąca 2025 roku.”

```
SELECT
    DATE_FORMAT(data_zamowienia, '%Y-%m') AS miesiac,

    ROUND(AVG(ilosc * kwota), 2) AS srednia_kwota
FROM zamowienia
GROUP BY miesiac
ORDER BY miesiac;
```

✚ Funkcja `DATE_FORMAT()` formatuje datę — tutaj do postaci 2025-10 itd.

8. W którym kwartale było najwięcej zamówień?

„Znajdź kwartał, w którym złożono najwięcej zamówień.”

```
SELECT
    QUARTER(data_zamowienia) AS kwartal,
    COUNT(*) AS liczba_zamowien
```

```
FROM zamowienia
GROUP BY kwartal
ORDER BY liczba_zamowien DESC
LIMIT 1;
```

9. Zamówienia złożone w weekendy

„Wyświetl zamówienia, które złożono w sobotę lub niedzielę.”

```
SELECT *
FROM zamowienia
WHERE DAYOFWEEK(data_zamowienia) IN (1, 7);
```

✳ **DAYOFWEEK()** zwraca numer dnia tygodnia (1 = niedziela, 7 = sobota).

Różnice między CURDATE() a innymi podobnymi funkcjami

| Funkcja | Zwraca | Przykład wyniku |
|-----------------------|--|----------------------------|
| CURDATE() | Tylko datę (rok-miesiąc-dzień) | 2025-11-05 |
| CURRENT_DATE() | To samo co CURDATE() | 2025-11-05 |
| NOW() | Datę i czas | 2025-11-05 14:32:11 |
| SYSDATE() | Datę i czas w momencie <i>realnego</i> wykonania | 2025-11-05 14:32:11 |
| CURTIME() | Tylko czas | 14:32:11 |

Lekcja

Temat: Group by

GROUP BY w **MySQL** służy do **grupowania rekordów**, które mają te same wartości w określonych kolumnach. Zazwyczaj używa się go **razem z funkcjami agregującymi**, takimi jak:

- **COUNT()** – zlicza ilość rekordów,
- **SUM()** – sumuje wartości,
- **AVG()** – liczy średnią,
- **MAX()** – zwraca wartość maksymalną,
- **MIN()** – zwraca wartość minimalną.

- ♦ **Przykład praktyczny**

```
CREATE TABLE orders (  
  id INT AUTO_INCREMENT PRIMARY KEY,  
  customer_name VARCHAR(50),  
  product_name VARCHAR(50),  
  quantity INT,  
  price DECIMAL(10, 2)  
);
```

```
INSERT INTO orders (customer_name, product_name, quantity, price)  
VALUES  
( 'Jan Kowalski', 'Laptop', 1, 3500.00),  
( 'Anna Nowak', 'Mysz', 2, 50.00),  
( 'Jan Kowalski', 'Mysz', 1, 50.00),  
( 'Piotr Wiśniewski', 'Klawiatura', 1, 120.00),  
( 'Anna Nowak', 'Laptop', 1, 3400.00),  
( 'Jan Kowalski', 'Laptop', 1, 3500.00);
```

- ♦ **Przykład 1 – Suma wartości zamówień dla każdego klienta**

```
SELECT customer_name, SUM(price * quantity) AS total_spent  
FROM orders  
GROUP BY customer_name;
```

- ♦ **Przykład 2 – Ile produktów kupił każdy klient**

```
SELECT customer_name, COUNT(*) AS total_orders  
FROM orders  
GROUP BY customer_name;
```

- ♦ **Przykład 3 – Średnia cena produktów kupionych przez każdego klienta**

```
SELECT customer_name, AVG(price) AS avg_price  
FROM orders  
GROUP BY customer_name;
```

- ♦ **Przykład 4 – Grupowanie po dwóch kolumnach (klient + produkt)**

```
SELECT customer_name, product_name, SUM(quantity) AS total_quantity  
FROM orders  
GROUP BY customer_name, product_name;
```

- ♦ **Przykład 5 GROUP BY z HAVING**

Załóżmy, że chcemy zobaczyć **tylko tych klientów**, którzy **wydali łącznie więcej niż 1000 zł**.

```
SELECT customer_name, SUM(price * quantity) AS total_spent
```

```
FROM orders
GROUP BY customer_name
HAVING SUM(price * quantity) > 1000;
```

♦ Przykład 6 filtracja po liczbie zamówień

Chcemy zobaczyć klientów, którzy złożyli więcej niż jedno zamówienie:

```
SELECT customer_name, COUNT(*) AS total_orders
FROM orders
GROUP BY customer_name
HAVING COUNT(*) > 1;
```

♦ Różnica między WHERE a HAVING

- WHERE filtruje **pojedyncze rekordy przed grupowaniem**,
- HAVING filtruje **całe grupy po agregacji**.

📌 Przykład błędu:

```
-- ❌ Nie zadziała:
SELECT customer_name, SUM(price)
FROM orders
WHERE SUM(price) > 1000
GROUP BY customer_name;
```

📌 Poprawnie:

```
SELECT customer_name, SUM(price)
FROM orders
GROUP BY customer_name
HAVING SUM(price) > 1000;
```

♦ Podsumowanie

- GROUP BY **grupuje** dane na podstawie wartości w kolumnach.
- Zazwyczaj używa się go z **funkcjami agregującymi** (SUM, COUNT, AVG, itp.).
- Może grupować po **jednej lub wielu kolumnach**.
- Często łączy się z HAVING, aby filtrować wyniki po agregacji (np. „pokaż tylko klientów, którzy wydali więcej niż 1000 zł”).

Lekcja

Temat: Funkcje związane z czasem, datą, operatorami łańcuchowymi

Funkcje daty i czasu

Link do dokumentacji MySQL:

<https://dev.mysql.com/doc/refman/8.4/en/date-and-time-functions.html>

| Metoda | Wyjaśnienie | Przykład SQL | Wynik |
|----------------------|---------------------------|--|------------------|
| ADDDATE() | Dodaje interwał do daty | SELECT ADDDATE('2024-01-01', INTERVAL 5 DAY); | 2024-01-06 |
| ADDTIME() | Dodaje czas | SELECT ADDTIME('10:00:00','02:30:00'); | 12:30:00 |
| CONVERT_TZ() | Konwersja strefy czasowej | SELECT CONVERT_TZ('2024-01-01 12:00','UTC','Europe/Warsaw'); | 2024-01-01 13:00 |
| CURDATE() | Bieżąca data | SELECT CURDATE(); | 2025-11-10 |
| CURTIME() | Bieżący czas | SELECT CURTIME(); | np. 14:22:01 |
| DATE() | Zwraca część datową | SELECT DATE('2024-01-01 10:00:00'); | 2024-01-01 |
| DATE_ADD() | Dodaje interwał do daty | SELECT DATE_ADD('2024-01-01', INTERVAL 1 MONTH); | 2024-02-01 |
| DATE_FORMAT() | Formatuje datę | SELECT DATE_FORMAT('2024-01-15','%d-%m-%Y'); | 15-01-2024 |
| DATE_SUB() | Odejmuje interwał | SELECT DATE_SUB('2024-01-10', INTERVAL 3 DAY); | 2024-01-07 |
| DATEDIFF() | Różnica między datami | SELECT DATEDIFF('2024-02-01','2024-01-01'); | 31 |

| | | | |
|------------------------|-------------------------|---|---------------------|
| DAY() | Dzień miesiąca | SELECT DAY('2024-01-15'); | 15 |
| DAYNAME() | Nazwa dnia | SELECT DAYNAME('2024-01-15'); | Tuesday |
| DAYOFMONTH() | Dzień miesiąca | SELECT DAYOFMONTH('2024-01-15'); | 15 |
| DAYOFWEEK() | Numer dnia tyg. (1=nd) | SELECT DAYOFWEEK('2024-01-15'); | 3 |
| DAYOFYEAR() | Dzień roku | SELECT DAYOFYEAR('2024-01-15'); | 15 |
| EXTRACT() | Wyodrębnia część daty | SELECT EXTRACT(YEAR FROM '2024-01-15'); | 2024 |
| FROM_DAYS() | Dni → data | SELECT FROM_DAYS(750000); | 2044-01-22 |
| FROM_UNIXTIME() | UNIX → data | SELECT FROM_UNIXTIME(1700000000); | 2023-11-14 22:13:20 |
| HOUR() | Pobiera godzinę | SELECT HOUR('12:45:00'); | 12 |
| LAST_DAY() | Ostatni dzień miesiąca | SELECT LAST_DAY('2024-02-10'); | 2024-02-29 |
| MAKEDATE() | Tworzy datę z dnia roku | SELECT MAKEDATE(2024,32); | 2024-02-01 |
| MAKETIME() | Tworzy czas | SELECT MAKETIME(10,20,30); | 10:20:30 |
| MICROSECOND() | Mikrosekundy | SELECT MICROSECOND('10:00:00.123456'); | 123456 |

| | | | |
|----------------------|--------------------------------|--|---------------------|
| MINUTE() | Minuta | SELECT MINUTE('12:45:30'); | 45 |
| MONTH() | Numer miesiąca | SELECT MONTH('2024-05-10'); | 5 |
| MONTHNAME() | Nazwa miesiąca | SELECT MONTHNAME('2024-05-10'); | May |
| NOW() | Aktualny datetime | SELECT NOW(); | 2025-11-10 14:20:xx |
| PERIOD_ADD() | Dodaje miesiące do YYYYMM | SELECT PERIOD_ADD(202401,2); | 202403 |
| PERIOD_DIFF() | Ilość miesięcy między okresami | SELECT PERIOD_DIFF(202402,202401); | 1 |
| QUARTER() | Kwartał | SELECT QUARTER('2024-05-10'); | 2 |
| SEC_TO_TIME() | Sekundy → czas | SELECT SEC_TO_TIME(3661); | 01:01:01 |
| SECOND() | Sekundy | SELECT SECOND('12:45:59'); | 59 |
| STR_TO_DATE() | Tekst → data | SELECT STR_TO_DATE('31-01-2024','%d-%m-%Y'); | 2024-01-31 |
| SUBTIME() | Odejmuje czas | SELECT SUBTIME('10:00:00','01:30:00'); | 08:30:00 |
| SYSDATE() | Czas wykonania | SELECT SYSDATE(); | 2025-11-10... |

| | | | |
|-------------------------|--------------------------|--|---------------------|
| TIME() | Czas z datetime | SELECT TIME('2024-01-01 12:30:45'); | 12:30:45 |
| TIME_FORMAT() | Formatuje czas | SELECT TIME_FORMAT('12:30:45','%H:%i'); | 12:30 |
| TIME_TO_SEC() | Czas → sekundy | SELECT TIME_TO_SEC('01:00:00'); | 3600 |
| TIMEDIFF() | Różnica czasu | SELECT TIMEDIFF('12:00:00','10:00:00'); | 02:00:00 |
| TIMESTAMP() | Tworzy datetime | SELECT TIMESTAMP('2024-01-01'); | 2024-01-01 00:00:00 |
| TIMESTAMPADD() | Dodaje interwał | SELECT TIMESTAMPADD(HOUR,2,'2024-01-01 10:00'); | 2024-01-01 12:00 |
| TIMESTAMPDIFF() | Różnica datetime | SELECT TIMESTAMPDIFF(DAY,'2024-01-01','2024-01-10'); | 9 |
| TO_DAYS() | Data → dni od roku 0 | SELECT TO_DAYS('2024-01-01'); | 739252 |
| TO_SECONDS() | Data → sekundy od roku 0 | SELECT TO_SECONDS('2024-01-01'); | 64092288000 |
| UNIX_TIMESTAMP() | Aktualny UNIX time | SELECT UNIX_TIMESTAMP(); | np. 1768060000 |
| UTC_DATE() | Data UTC | SELECT UTC_DATE(); | 2025-11-10 |

| | | | |
|------------------------|--------------------|---|---------------------|
| UTC_TIME() | Czas UTC | <code>SELECT UTC_TIME();</code> | 13:14:xx |
| UTC_TIMESTAMP() | Datetime UTC | <code>SELECT UTC_TIMESTAMP();</code> | 2025-11-10 13:14:xx |
| WEEK() | Numer tygodnia | <code>SELECT WEEK('2024-01-10');</code> | 1 |
| WEEKDAY() | Dzień tyg. (0=pon) | <code>SELECT WEEKDAY('2024-01-10');</code> | 3 |
| WEEKOFYEAR() | Tydzień ISO | <code>SELECT WEEKOFYEAR('2024-01-10');</code> | 2 |
| YEAR() | Rok | <code>SELECT YEAR('2024-01-10');</code> | 2024 |
| YEARWEEK() | Rok + tydzień | <code>SELECT YEARWEEK('2024-01-10');</code> | 202402 |

UTC (Uniwersalny Czas Koordynowany) to światowy standard czasu atomowego, który służy jako podstawa do ustalania lokalnego czasu w różnych strefach czasowych. Polska znajduje się w strefie czasowej UTC+1 (czas środkowoeuropejski, CET) zimą i UTC+2 (czas środkowoeuropejski letni, CEST) latem, a lokalny czas w Polsce jest o 1 lub 2 godziny późniejszy od czasu UTC.

- **Co to jest UTC:**
 - UTC to międzynarodowy standard czasu, który jest niezależny od ruchu obrotowego Ziemi i oparty na bardzo precyzyjnym czasie atomowym.

- Jest to punkt odniesienia, taki sam na całym świecie, do którego dodaje się lub od którego odejmuje się czas, aby uzyskać lokalny czas dla danej strefy czasowej.

- **UTC w Polsce:**

- Polska leży w strefie czasowej UTC+1 (czas zimowy) lub UTC+2 (czas letni).
- Czas zimowy (CET): Obowiązuje od ostatniej niedzieli października do ostatniej niedzieli marca. Czas lokalny w Polsce jest o 1 godzinę późniejszy niż UTC. (np. jeśli UTC to 12:00, w Polsce jest 13:00).
- Czas letni (CEST): Obowiązuje od ostatniej niedzieli marca do ostatniej niedzieli października. Czas lokalny w Polsce jest o 2 godziny późniejszy niż UTC. (np. jeśli UTC to 12:00, w Polsce jest 14:00).

Zastosowania:

- Programowanie - przechowywanie dat i czasu w bazach danych
- Lotnictwo - koordynacja lotów międzynarodowych
- Internet - synchronizacja serwerów
- Telekomunikacja - koordynacja transmisji
- Nauka - precyzyjne pomiary czasu

W praktyce: Gdy widzisz znacznik czasu typu `2025-11-11T14:30:00Z`, litera "Z" na końcu oznacza właśnie UTC (od "Zulu time" - wojskowego określenia UTC).

Przykłady:

- Polska: UTC+1 (zimą) lub UTC+2 (latem)
- Nowy Jork: UTC-5 (zimą) lub UTC-4 (latem)
- Tokio: UTC+9
- Londyn: UTC+0 (zimą) lub UTC+1 (latem)

Funkcje i operatory łańcuchowe

Link do dokumentacji MySQL:

<https://dev.mysql.com/doc/refman/8.4/en/string-functions.html>

| Metoda | Wyjaśnienie | Przykład | Wynik |
|---------------------------|---|-------------------------------------|---------|
| ASCII() | Zwraca kod ASCII pierwszego znaku | SELECT ASCII('A'); | 65 |
| BIN() | Zwraca liczbę w postaci binarnej | SELECT BIN(10); | 1010 |
| BIT_LENGTH() | Zwraca długość napisu w bitach | SELECT BIT_LENGTH('ABC'); | 24 |
| CHAR() | Zwraca znak odpowiadający podanemu kodowi ASCII | SELECT CHAR(65); | 'A' |
| CHAR_LENGTH() | Liczba znaków (nie bajtów) | SELECT CHAR_LENGTH('Łódź'); | 4 |
| CHARACTER_LENGTH() | To samo co CHAR_LENGTH() | SELECT CHARACTER_LENGTH('Test'); | 4 |
| CONCAT() | Łączy napisy | SELECT CONCAT('A', 'B', 'C'); | 'ABC' |
| CONCAT_WS() | Łączy napisy z separatorem | SELECT CONCAT_WS('-', 'A','B','C'); | 'A-B-C' |

| | | | |
|----------------------|--|--|---------------|
| ELT() | Zwraca element listy na indeksie (1-based) | SELECT ELT(2,'jeden','dwa','trzy'); | 'dwa' |
| EXPORT_SET() | Zamienia liczby bitowe na tekst ON/OFF | SELECT EXPORT_SET(5, 'ON', 'OFF', ',', 4); | ON,OFF,ON,OFF |
| FIELD() | Zwraca pozycję pierwszego argumentu w liście | SELECT FIELD('kot','pies','kot','mysz'); | 2 |
| FIND_IN_SET() | Pozycja elementu w liście CSV | SELECT FIND_IN_SET('B', 'A,B,C'); | 2 |
| FORMAT() | Formatuje liczbę z przecinkami | SELECT FORMAT(12345.678, 2); | '12,345.68' |
| FROM_BASE64() | Dekoduje Base64 | SELECT FROM_BASE64('SGVsbG8='); | 'Hello' |
| HEX() | Zamienia liczbę lub tekst na hex | SELECT HEX('ABC'); | 414243 |
| INSERT() | Wstawia podciąg w podaną pozycję, zastępując określoną liczbę znaków | SELECT INSERT('abcdef', 3, 2, 'XYZ'); | 'abXYZef' |
| INSTR() | Pozycja pierwszego wystąpienia podciągu | SELECT INSTR('abcabc','ca'); | 3 |
| LCASE() | To samo co LOWER() – zamienia na małe litery | SELECT LCASE('Test'); | 'test' |

| | | | |
|--------------------------|---|-------------------------------------|--------------------------|
| LEFT() | Zwraca określoną liczbę znaków od lewej | SELECT LEFT('abcdef', 3); | 'abc' |
| LENGTH() | Długość napisu w bajtach | SELECT LENGTH('ABC'); | 3 |
| LIKE | Sprawdza dopasowanie wzorca | SELECT 'Ala' LIKE 'A%'; | 1 |
| LOAD_FILE() | Wczytuje zawartość pliku (jeśli SQL ma dostęp) | SELECT LOAD_FILE('/path/file.txt'); | <i>treść pliku</i> |
| LOCATE() | Pozycja podciągu (jak INSTR, ale kolejność argumentów odwrotna) | SELECT LOCATE('b','abc'); | 2 |
| LOWER() | Zamienia na małe litery | SELECT LOWER('TEST'); | 'test' |
| LPAD() | Uzupełnia z lewej do zadanej długości | SELECT LPAD('7', 3, '0'); | '007' |
| LTRIM() | Usuwa spacje z lewej | SELECT LTRIM(' test'); | 'test' |
| MAKE_SET() | Zwraca listę elementów pasujących do bitów liczby | SELECT MAKE_SET(5,'A','B','C'); | 'A,C' |
| MATCH() AGAINST() | Pełnotekstowe wyszukiwanie | SELECT MATCH(text) AGAINST('kot'); | <i>ocena dopasowania</i> |

| | | | |
|-----------------------|--|---|--------------|
| MID() | Alias SUBSTRING() | SELECT MID('abcdef', 2, 3); | 'bcd' |
| NOT LIKE | Odwrotność LIKE | SELECT 'Ala' NOT LIKE 'K%'; | 1 |
| NOT REGEXP | Odwrotność REGEXP | SELECT 'abc' NOT REGEXP '^[0-9]+\$'; | 1 |
| OCT() | Zamienia liczbę na system ósemkowy | SELECT OCT(15); | '17' |
| OCTET_LENGTH() | Alias LENGTH() | SELECT OCTET_LENGTH('ABC'); | 3 |
| ORD() | Kod ASCII pierwszego znaku | SELECT ORD('A'); | 65 |
| POSITION() | Alias LOCATE() | SELECT POSITION('a' IN 'banan'); | 2 |
| QUOTE() | Zwraca tekst w bezpiecznej formie (escape) | SELECT QUOTE("Ala's cat"); | 'Ala\'s cat' |
| REGEXP | Dopasowanie wyrażenia regularnego | SELECT 'abc123' REGEXP '[0-9]+'; | 1 |
| REGEXP_INSTR() | Pozycja dopasowania regexu | SELECT REGEXP_INSTR('abc123','[0-9]+'); | 4 |
| REGEXP_LIKE() | Czy pasuje regex | SELECT REGEXP_LIKE('test123','[a-z]+'); | 1 |

| | | | |
|-------------------------|------------------------------------|--|---------------|
| REGEXP_REPLACE() | Zamienia dopasowane fragmenty | SELECT REGEXP_REPLACE('a1b2c3','[0-9]','X'); | 'aXbXcX' |
| REGEXP_SUBSTR() | Zwraca fragment pasujący do regexu | SELECT REGEXP_SUBSTR('abc123','[0-9]+'); | '123' |
| REPEAT() | Powtarza tekst | SELECT REPEAT('A',3); | 'AAA' |
| REPLACE() | Podmienia tekst | SELECT REPLACE('ala ma kota','a','X'); | 'XIX mX kotX' |
| REVERSE() | Odwraca napis | SELECT REVERSE('kota'); | 'atok' |
| RIGHT() | Znaki od prawej | SELECT RIGHT('abcdef', 2); | 'ef' |
| RLIKE | Alias REGEXP | SELECT 'abc' RLIKE '[a-z]+'; | 1 |
| RPAD() | Uzupełnia napis z prawej | SELECT RPAD('A', 4, '.'); | 'A...' |
| RTRIM() | Usuwa spacje z prawej | SELECT RTRIM('test '); | 'test' |
| SOUNDEX() | Kod fonetyczny słów | SELECT SOUNDEX('Robert'); | 'R163' |
| SOUNDS LIKE | Porównanie brzmienia | SELECT 'Robert' SOUNDS LIKE 'Rupert'; | 1 |
| SPACE() | Generuje spacje | SELECT SPACE(5); | ' ' |
| STRCMP() | Porównuje napisy | SELECT STRCMP('abc','abd'); | -1 |

| | | | |
|--------------------------|--|--|-------------|
| SUBSTR() | Podciąg (alias SUBSTRING) | SELECT SUBSTR('abcdef',2,3); | 'bcd' |
| SUBSTRING() | Podciąg | SELECT SUBSTRING('abcdef',3); | 'cdef' |
| SUBSTRING_INDEX() | Podciąg do N-tego separatora | SELECT SUBSTRING_INDEX('a,b,c',';',2); | 'a,b' |
| TO_BASE64() | Kodowanie Base64 | SELECT TO_BASE64('Hello'); | 'SGVsbG8=' |
| TRIM() | Usuwa spacje z obu stron | SELECT TRIM(' test '); | 'test' |
| UCASE() | Alias UPPER() | SELECT UCASE('abc'); | 'ABC' |
| UNHEX() | Hex → tekst | SELECT UNHEX('414243'); | 'ABC' |
| UPPER() | Zamienia na wielkie litery | SELECT UPPER('kot'); | 'KOT' |
| WEIGHT_STRING() | Zwraca wewnętrzną wagę znaków (techniczne) | SELECT WEIGHT_STRING('A'); | (hex bajty) |

Lekcja

Temat: ERD (Diagram związków encji ang. Entity Relationship Diagram)

ERD — diagram związków encji

To graficzny sposób przedstawienia struktury bazy danych:

- jakie **tabele (encje)** istnieją,
- jakie mają **atrybuty (kolumny)**,
- jakie występują **relacje** między tabelami:

- **1:1**
- **1:N**
- **N:M**

ERD jest tworzony zanim powstanie baza danych, aby zaplanować jej strukturę.

Encja (Entity) = obiekt, który ma znaczenie w systemie i który chcesz zapisać w bazie.

Inaczej mówiąc:

👉 **Encja** = tabela w bazie danych

👉 **Atrybut** = kolumna w tabeli

Przykłady encji:

- **User** (użytkownik)
- **Product** (produkt)
- **Order** (zamówienie)
- **Invoice** (faktura)
- **Department** (dział firmy)

Każda encja ma klucz główny (Primary Key, PK) – unikalny identyfikator, np. id.

Tworzenie krok po kroku diagramu związków encji

Krok 1: Zidentyfikuj encje (tabele)

Krok 2: Określ atrybuty

Dla każdej encji określasz pola.

Przykład:

Customer

- id
- first_name
- last_name
- email

Krok 3: Ustal klucze główne

Każda encja ma PK:

Krok 4: Określ relacje między encjami

1) Relacja 1:1 (One to One)

Jeden rekord odpowiada dokładnie jednemu rekordowi w drugiej tabeli.

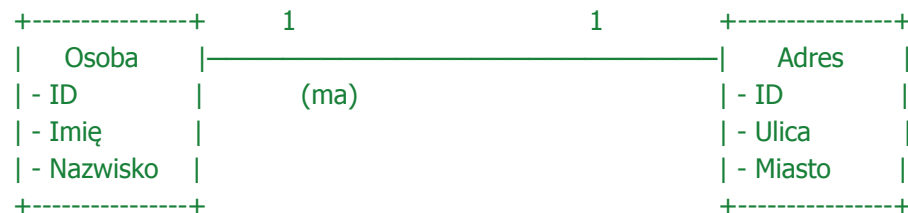
2) Relacja 1:N (One to Many)

Jeden klient może mieć wiele zamówień.

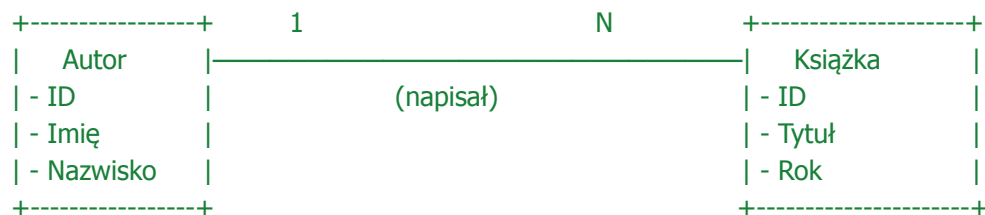
3) Relacja N:M (Many to Many)

Tworzy się tabelę pośredniczącą.

✓ 1. Relacja 1 : 1 (Osoba — Adres)

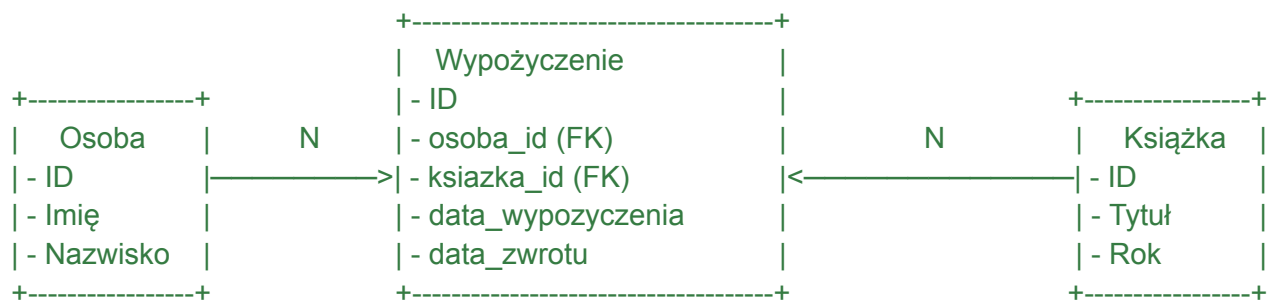


✓ 2. Relacja 1 : N (Autor — Książka)



✓ 3. Relacja N : N (Osoba — Książka) przez tabelę Wypożyczenie

W MySQL/SQL relacja N:N **zawsze wymaga tabeli pośredniej**.

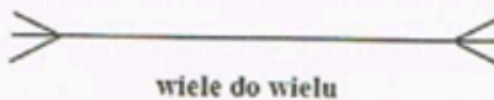
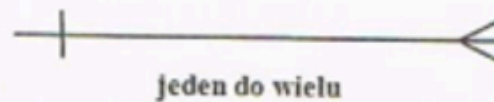
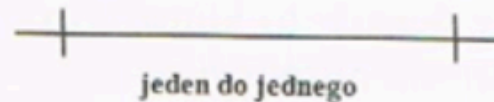


N : N
(wiele osób wypożycza wiele książek)

Opis reprezentacji graficznej stopnia związku został pokazany na rysunku

Rysunek

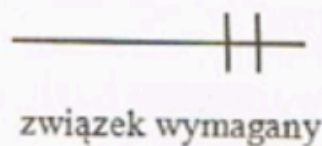
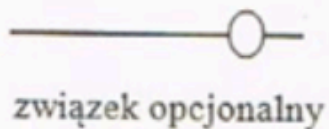
Graficzna reprezentacja
związków zachodzą-
cych między encjami



Opis reprezentacji graficznej opcjonalności związku został pokazany na rysunku

Rysunek

Graficzna reprezentacja
opcjonalności związku



Diagramy ERD możemy tworzyć za pomocą różnych notacji. Najpopularniejsze są diagramy w zapisie według Martina i Chena.

Lekcja

Temat: UNIQUE w MySQL

UNIQUE oznacza, że wartości w danej kolumnie (lub w zestawie kolumn) muszą być **unikalne** — nie mogą się powtarzać. To **nie jest klucz główny**, ale działa podobnie.

1. UNIQUE na jednej kolumnie

```
CREATE TABLE klienci (  
  id INT PRIMARY KEY,  
  email VARCHAR(255) UNIQUE  
);
```

Oznacza:

- każdy **email** musi być **inny**
- nie można dodać dwóch klientów z tym samym emailiem
- **NULL** jest dozwolony (i może być więcej niż jeden, bo MySQL traktuje NULL jako wartość nieporównywalną)

2. UNIQUE na wielu kolumnach (unikalna kombinacja)

Można zrobić również **unikalność złożoną**, podobnie jak composite key:

```
CREATE TABLE zapis (  
  uczen_id INT,  
  kurs_id INT,  
  UNIQUE (uczen_id, kurs_id)  
);
```

Oznacza:

- ten sam uczeń nie może zapisać się drugi raz na ten sam kurs
- ale może zapisać się na inny

3. Dodanie UNIQUE do istniejącej tabeli

ALTER TABLE klienci

ADD UNIQUE (email);

4. Różnica: PRIMARY KEY vs UNIQUE

| Cecha | PRIMARY KEY | UNIQUE |
|------------------------------|----------------------------------|----------------------|
| Musi być unikalne | ✓ Tak | ✓ Tak |
| Może być NULL | ✗ Nie | ✓ Tak |
| Można mieć więcej niż jeden? | ✗ Nie (tylko jeden PK na tabelę) | ✓ Tak (wiele UNIQUE) |
| Tworzy indeks | ✓ Tak | ✓ Tak |

Podsumowanie

UNIQUE:

- zapewnia **unikalność wartości**
- można stosować na **jednej** lub **wielu kolumnach**
- pozwala uniknąć duplikacji danych
- ale **nie zastępuje klucza głównego**, tylko go uzupełnia

Lekcja

Temat: Replair. Akrónim ACID, kategorie poleceń w SQL. Polecenie **DELETE i DROP**. System Zarządzania Bazą Danych (DBMS – DataBase Management System)

♦ Polecenie **REPAIR TABLE** w MySQL służy do **naprawy uszkodzonych tabel** oraz do **optymalizacji** pewnych typów tabel. Działa jednak tylko dla wybranych silników — głównie **MyISAM** oraz **ARCHIVE**.

Jeśli tabela MyISAM została uszkodzona (np. po awarii serwera), **REPAIR TABLE** próbuje:

- odbudować indeksy,
- odtworzyć strukturę danych,
- odzyskać jak najwięcej wierszy.

Składnia

```
REPAIR TABLE nazwa_tabeli;
```

Dodatkowe opcje:

- **QUICK** – naprawia tylko indeksy, bez skanowania danych
- **EXTENDED** – dogłębna naprawa, rekonstruuje plik danych (najwolniejsza)
- **USE_FRM** – odbudowuje indeksy na podstawie pliku .frm (tylko MyISAM)

REPAIR TABLE nie naprawia tabel InnoDB.

♦ **Akrónim ACID w SQL oznacza cztery kluczowe właściwości transakcji w systemach baz danych:**

A – Atomicity (Atomowość)

Transakcja jest niepodzielna: albo wykonuje się w całości, albo wcale.

C – Consistency (Spójność)

Transakcja musi pozostawić bazę danych w stanie zgodnym z regułami i ograniczeniami (constraints).

I – Isolation (Izolacja)

Równocześnie wykonywane transakcje nie powinny wzajemnie sobie przeszkadzać — każda działa tak, jakby była wykonywana osobno.

D – Durability (Trwałość)

Po zatwierdzeniu transakcji (COMMIT) jej skutki są trwałe i nie zostaną utracone, nawet w przypadku awarii.

◆ Podstawowe kategorie poleceń w SQL to:

- **DDL (Data Definition Language)** – definiowanie struktury bazy danych (np. tworzenie tabel).
- **DML (Data Manipulation Language)** – manipulacja danymi (np. wstawianie, aktualizacja, usuwanie).
- **DCL (Data Control Language)** – zarządzanie uprawnieniami (np. GRANT, REVOKE).
- **DQL (Data Query Language)** – pobieranie danych (np. SELECT).
- **TCL (Transaction Control Language)** – zarządzanie transakcjami (np. COMMIT, ROLLBACK).

◆ DELETE FROM

Polecenie:

DELETE FROM nazwa_tabeli;

Usuwa rekordy (wiersze) z tabeli, ale:

- **nie usuwa struktury tabeli**, kolumn ani jej definicji,
- **nie resetuje auto_increment** (chyba że użyjesz TRUNCATE),
- może usuwać pojedyncze wiersze lub wszystkie — zależnie od warunku WHERE.

Przykłady:

Usuń wszystkie rekordy:

DELETE FROM users;

Usuń tylko wybrane:

DELETE FROM users WHERE id = 5;

◆ DROP

Polecenie:

DROP TABLE nazwa_tabeli;

Usuwa całą tabelę z bazy danych, czyli:

- usuwa wszystkie dane,
- usuwa strukturę tabeli (kolumny, indeksy, klucze),
- usuwa definicję tabeli z katalogu bazy.

Po wykonaniu DROP tabela **przestaje istnieć**.

Przykłady:

Usuń tabelę:

DROP TABLE users;

Usuń całą bazę danych:

DROP DATABASE sklep;

- ◆ Polecenie ustawiające określoną wartość dla kolumny dla **wszystkich rekordów**:

UPDATE nazwa_tabeli
SET nazwa_kolumny = **WARTOSC**;

♦ **System Zarządzania Bazą Danych (DBMS – DataBase Management System)**

to oprogramowanie, które umożliwia:

- tworzenie baz danych,
- zapisywanie, modyfikowanie i usuwanie danych,
- zarządzanie dostępem użytkowników,
- zapewnianie bezpieczeństwa i integralności danych,
- wykonywanie zapytań (np. SQL),
- jednoczesny dostęp wielu użytkowników.

Prościej:

👉 DBMS to program do zarządzania danymi w bazie – np. MySQL, PostgreSQL, Oracle, SQL Server.

✅ **Jakie mechanizmy są NIEZBĘDNE dla Systemu Zarządzania Bazą Danych?**
Wszystkie SZBD muszą mieć pewne podstawowe mechanizmy — zwykle wymienia się:

1. Mechanizm składowania danych

Przechowywanie danych na dysku, w tabelach, indeksach itp.

2. Mechanizm dostępu do danych / język zapytań (np. SQL)

Możliwość pobierania, wstawiania, usuwania, aktualizowania danych.

3. Mechanizmy bezpieczeństwa

- autoryzacja i autentykacja,
- role, użytkownicy,
- uprawnienia.

4. Mechanizmy kontroli współbieżności (concurrency control)

Zapewniają poprawną pracę wielu użytkowników *jednocześnie*.

5. Mechanizmy zapewnienia integralności danych

- klucze główne,
- klucze obce,
- ograniczenia (NOT NULL, UNIQUE, CHECK).

Chronią przed niepoprawnymi danymi.

6. Mechanizmy odtwarzania po awarii (recovery)

Przywracają działanie po:

- awarii systemu,
- utracie zasilania,

- błędach sprzętu.

Zapisywanie logów transakcyjnych, backupy itp.

7. Mechanizmy zarządzania transakcjami (ACID)

Każdy SZBD musi obsługiwać transakcje zgodnie z zasadą:

- **A atomicity** – niepodzielność
- **C consistency** – spójność
- **I isolation** – izolacja
- **D durability** – trwałość

To fundament poprawnej pracy.

```
/*
DROP TABLE Przedmioty;
DROP TABLE Osoby;
*/

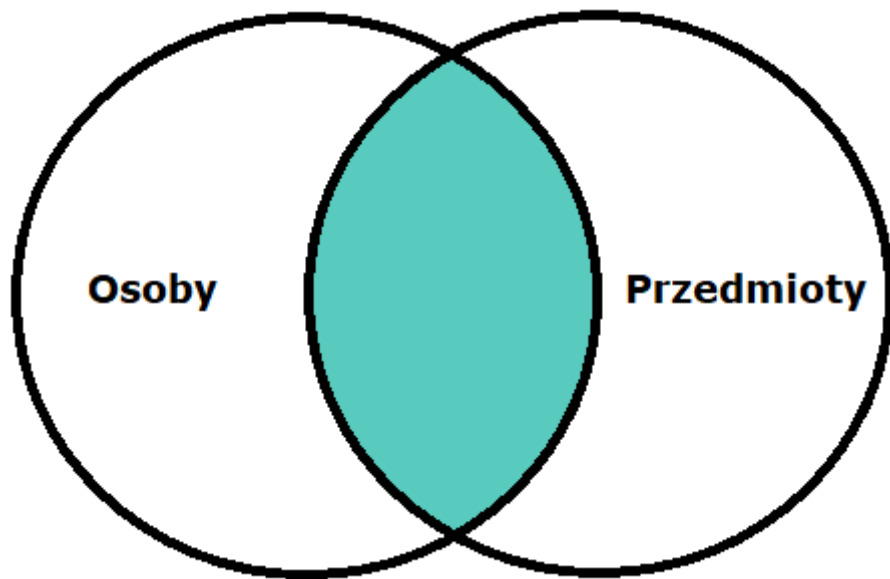
CREATE TABLE Osoby (
    osoba_id INT AUTO_INCREMENT PRIMARY KEY,
    imie VARCHAR(50) NOT NULL,
    nazwisko VARCHAR(50) NOT NULL
);

CREATE TABLE Przedmioty (
    przedmiot_id INT AUTO_INCREMENT PRIMARY KEY,
    nazwa VARCHAR(100) NOT NULL,
    osoba_id INT,
    CONSTRAINT fk_przedmiot_osoba FOREIGN KEY (osoba_id) REFERENCES
Osoby(osoba_id)
);

INSERT INTO Osoby (imie, nazwisko) VALUES
('Jan', 'Kowalski'),
('Anna', 'Nowak'),
('Piotr', 'Zieliński'),
('Kasia', 'Wiśniewska'),
('Patryk', 'Nowakowski');

INSERT INTO Przedmioty (nazwa, osoba_id) VALUES
('Laptop', 1),
('Telefon', 1),
('Rower', 2),
('Książka', 3),
('Plecak', 4),
('Kubek', null);
```

- ♦ **INNER JOIN** - czyli wszystkie wspólne rekordy, bez NULL

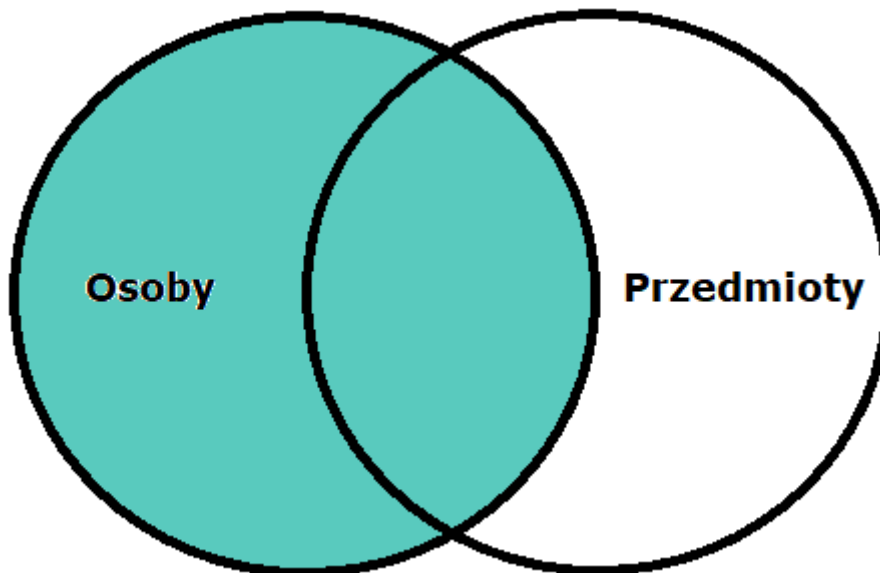


```
SELECT Osoby.imie, Osoby.nazwisko, Przedmioty.nazwa  
FROM Osoby  
INNER JOIN Przedmioty ON Osoby.osoba_id = Przedmioty.osoba_id;
```

Wynik:

| imie | nazwisko | nazwa |
|-------|------------|---------|
| Jan | Kowalski | Laptop |
| Jan | Kowalski | Telefon |
| Anna | Nowak | Rower |
| Piotr | Zieliński | Książka |
| Kasia | Wiśniewska | Plecak |

♦ **LEFT JOIN** - czyli wszystkie rekordy z lewej tabeli. W naszym przypadku lewa tabela to Osoby. Jeśli Osoba jest a nie ma dopasowania w tabeli Przedmioty również się wyświetli.

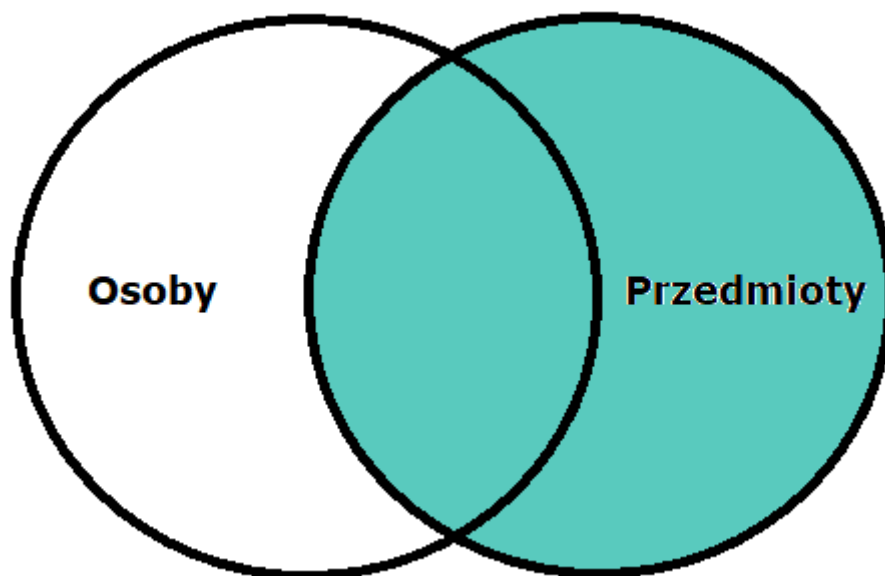


```
SELECT Osoby.imie, Osoby.nazwisko, Przedmioty.nazwa  
FROM Osoby  
LEFT JOIN Przedmioty ON Osoby.osoba_id = Przedmioty.osoba_id;
```

Wynik:

| imie | nazwisko | nazwa |
|--------|------------|---------|
| Jan | Kowalski | Laptop |
| Jan | Kowalski | Telefon |
| Anna | Nowak | Rower |
| Piotr | Zieliński | Książka |
| Kasia | Wiśniewska | Plecak |
| Patryk | Nowakowski | NULL |

♦ **RIGHT JOIN** - czyli wszystkie rekordy z prawej tabeli. W naszym przypadku prawa tabela to Przedmioty. Jeśli Przedmiot nie ma dopasowania w tabeli Osoby również się wyświetli.



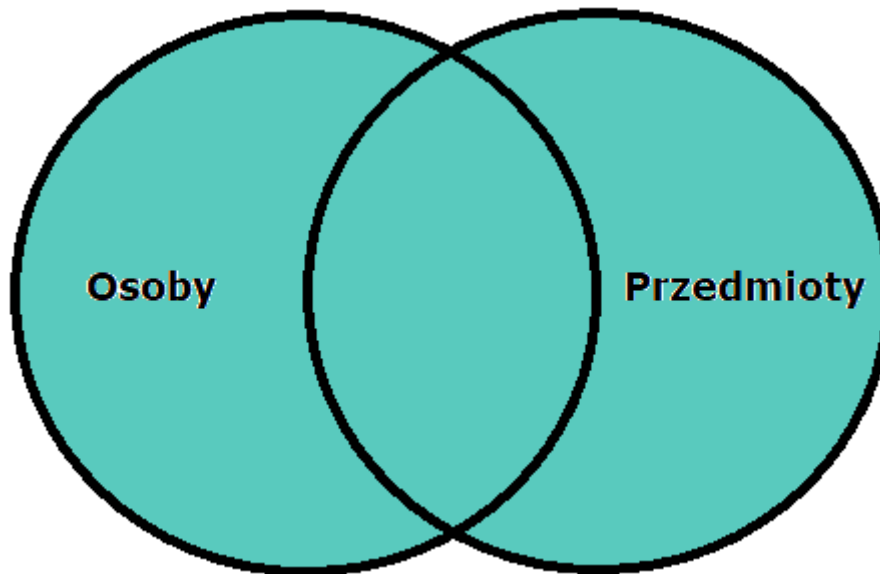
```
SELECT Osoby.imie, Osoby.nazwisko, Przedmioty.nazwa  
FROM Osoby  
RIGHT JOIN Przedmioty ON Osoby.osoba_id = Przedmioty.osoba_id;
```

Wynik:

| imie | nazwisko | nazwa |
|-------|------------|---------|
| Jan | Kowalski | Laptop |
| Jan | Kowalski | Telefon |
| Anna | Nowak | Rower |
| Piotr | Zieliński | Książka |
| Kasia | Wiśniewska | Plecak |
| NULL | NULL | Kubek |

♦ **FULL OUTER JOIN (LEFT JOIN, UNION, RIGHT JOIN)** - czyli wszystkie rekordy z prawej i lewej tabeli połączone.

W MySQL nie ma instrukcji FULL OUTER JOIN. Jednak można wykonać ten mechanizm za pomocą połączenia poleceń right join, left join i UNION.



```
SELECT o.imie, o.nazwisko, p.nazwa
FROM Osoby o
LEFT JOIN Przedmioty p ON o.osoba_id = p.osoba_id
```

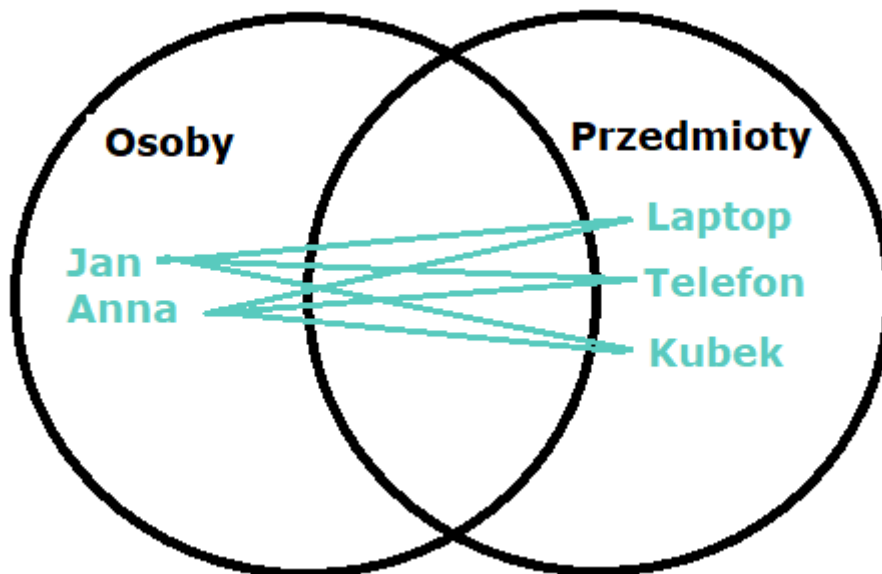
UNION

```
SELECT o.imie, o.nazwisko, p.nazwa
FROM Osoby o
RIGHT JOIN Przedmioty p ON o.osoba_id = p.osoba_id;
```

Wynik:

| imie | nazwisko | nazwa |
|--------|------------|---------|
| Jan | Kowalski | Laptop |
| Jan | Kowalski | Telefon |
| Anna | Nowak | Rower |
| Piotr | Zieliński | Książka |
| Kasia | Wiśniewska | Plecak |
| Patryk | Nowakowski | NULL |
| NULL | NULL | Kubek |

♦ **CROSS JOIN** - łączy **każdy wiersz z pierwszej tabeli z każdym wierszem z drugiej tabeli**.



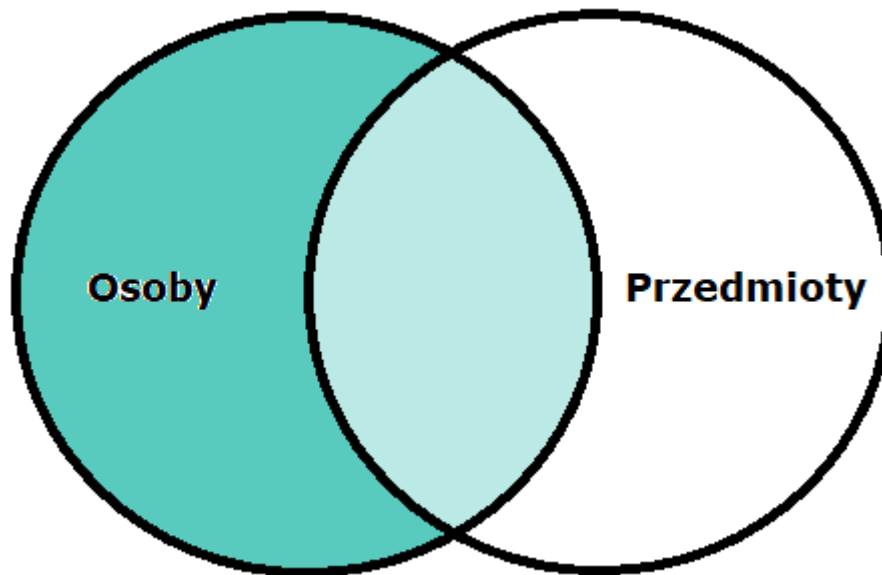
```
SELECT o.imie, p.nazwa  
FROM Osoby o  
CROSS JOIN Przedmioty p;
```

Wynik:

| imie | nazwa |
|--------|---------|
| Jan | Laptop |
| Anna | Laptop |
| Piotr | Laptop |
| Kasia | Laptop |
| Patryk | Laptop |
| Jan | Telefon |
| Anna | Telefon |
| Piotr | Telefon |
| Kasia | Telefon |
| Patryk | Telefon |
| Jan | Rower |

| | |
|--------|---------|
| Anna | Rower |
| Piotr | Rower |
| Kasia | Rower |
| Patryk | Rower |
| Jan | Książka |
| Anna | Książka |
| Piotr | Książka |
| Kasia | Książka |
| Patryk | Książka |
| Jan | Plecak |
| Anna | Plecak |
| Piotr | Plecak |
| Kasia | Plecak |
| Patryk | Plecak |
| Jan | Kubek |
| Anna | Kubek |
| Piotr | Kubek |
| Kasia | Kubek |
| Patryk | Kubek |

♦ **LEFT JOIN excluding INNER JOIN** (LEFT JOIN wykluczający wiersze dopasowane) - **na początku wykonuje zapytanie LEFT JOIN. Następnie filtruje wynik wyświetlając z lewej tabeli wartości nie mających dopasowania w tabeli prawej.** Czyli w naszym przypadku z tabeli Osoby wyświetli wartości, które nie mają dopasowania

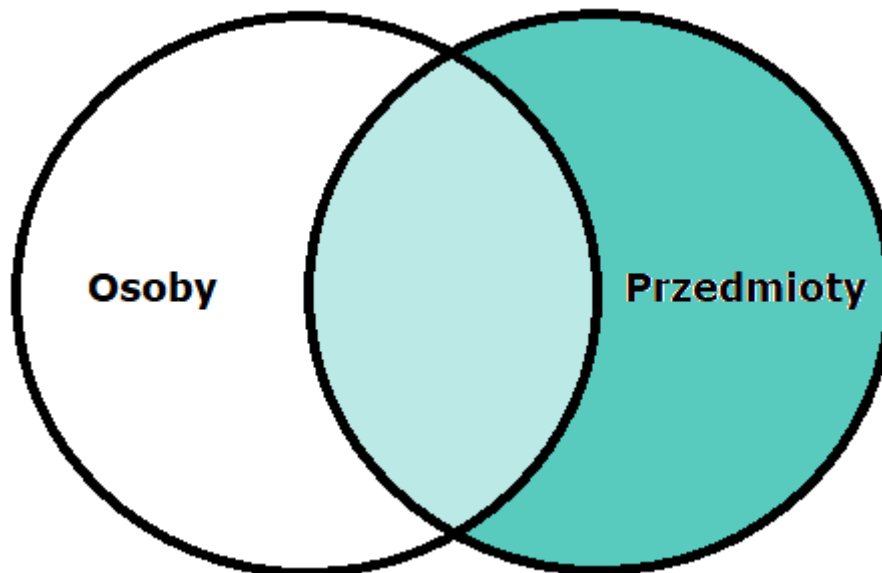


```
SELECT o.imie, o.nazwisko, p.nazwa
FROM Osoby o
LEFT JOIN Przedmioty p ON o.osoba_id = p.osoba_id
WHERE p.osoba_id IS NULL;
```

Wynik:

| imie | nazwisko | nazwa |
|--------|------------|-------|
| Patryk | Nowakowski | NULL |

♦ **RIGHT JOIN excluding INNER JOIN (RIGHT JOIN wykluczający wiersze dopasowane)** - **na początku wykonuje zapytanie RIGHT JOIN. Następnie filtruje wynik wyświetlając z prawej tabeli wartości nie mających dopasowania w tabeli lewej.** Czyli w naszym przypadku z tabeli Przedmioty wyświetli wartości, które nie mają dopasowania

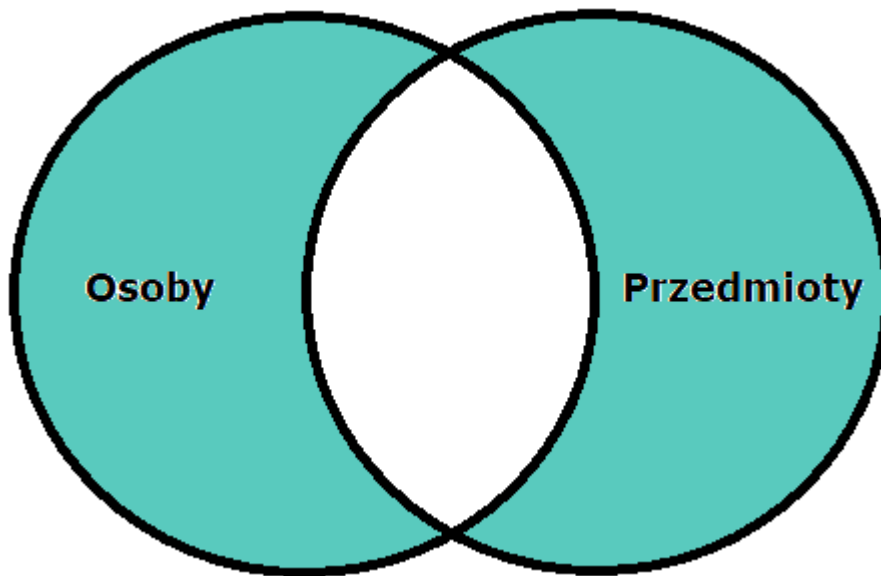


```
SELECT o.imie, o.nazwisko, p.nazwa
FROM Osoby o
RIGHT JOIN Przedmioty p ON o.osoba_id = p.osoba_id
WHERE o.osoba_id IS NULL;
```

Wynik:

| imie | nazwisko | nazwa |
|------|----------|-------|
| NULL | NULL | Kubek |

♦ **FULL OUTER JOIN excluding INNER JOIN (LEFT JOIN wykluczający wiersze dopasowane, UNION, RIGHT JOIN wykluczający wiersze dopasowane)** - czyli wszystkie rekordy z prawej i lewej tabeli połączone. Następnie odrzucamy te wiersze, które mają dopasowanie w obu tabelach.
W MySQL nie ma instrukcji FULL OUTER JOIN. Dla MySQL należy zastosować UNION. Czyli left join z wartościami nie mających dopasowania oraz right join z wartościami nie mających dopasowania łączymy z UNION.



```
SELECT o.imie, o.nazwisko, p.nazwa
FROM Osoby o
LEFT JOIN Przedmioty p ON o.osoba_id = p.osoba_id
WHERE p.osoba_id IS NULL
```

UNION

```
SELECT o.imie, o.nazwisko, p.nazwa
FROM Osoby o
RIGHT JOIN Przedmioty p ON o.osoba_id = p.osoba_id
WHERE o.osoba_id IS NULL;
```

Wynik:

| imie | nazwisko | nazwa |
|--------|------------|-------|
| Patryk | Nowakowski | NULL |
| NULL | NULL | Kubek |

Lekcja

Temat: Właściwości kolumn (pól) w MySQL: NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, DEFAULT, CHECK, AUTO_INCREMENT, ENUM, COMMENT

W MySQL możesz nałożyć **wiele rodzajów właściwości (constraints)** na pojedynczą kolumnę albo na kilka kolumn naraz, żeby wymusić reguły zachowania danych.

✓ 1. NOT NULL

Kolumna **nie może przyjmować wartości NULL**.
Wymusza, że musisz zawsze podać wartość.

Przykład:

```
CREATE TABLE osoby (  
  id INT NOT NULL,  
  imie VARCHAR(100) NOT NULL  
);
```

Wyjaśnienie:

- imie i id **musi** być podane.

✓ 2. UNIQUE

Wymusza **unikalne wartości** w kolumnie — nie mogą się powtarzać.

Przykład:

```
CREATE TABLE klienci (  
  id INT PRIMARY KEY,  
  email VARCHAR(255) UNIQUE  
);
```

Wyjaśnienie:

Dwa takie same maile → **✗** błąd.

Można też ustawić UNIQUE na **kilka kolumn naraz**:

```
UNIQUE (uczen_id, kurs_id)
```

Dodanie UNIQUE do istniejącej tabeli

```
ALTER TABLE klienci  
ADD UNIQUE (email);
```

Różnica: PRIMARY KEY vs UNIQUE

| Cecha | PRIMARY KEY | UNIQUE |
|------------------------------|----------------------------------|----------------------|
| Musi być unikalne | ✓ Tak | ✓ Tak |
| Może być NULL | ✗ Nie | ✓ Tak |
| Można mieć więcej niż jeden? | ✗ Nie (tylko jeden PK na tabelę) | ✓ Tak (wiele UNIQUE) |
| Tworzy indeks | ✓ Tak | ✓ Tak |

✓ 3. PRIMARY KEY

- jednoznacznie identyfikuje każdy wiersz (unikalny),
- automatycznie ma **UNIQUE + NOT NULL**.

Przykład:

```
CREATE TABLE produkty (
  produkt_id INT PRIMARY KEY,
  nazwa VARCHAR(100)
);
```

Możesz też zrobić klucz **złożony z kilku kolumn**:
PRIMARY KEY (zamowienie_id, produkt_id)

✓ 4. FOREIGN KEY

Łączy tabele — kolumna musi wskazywać na wartość z innej tabeli.

Przykład:

```
CREATE TABLE zamowienia (
  id INT PRIMARY KEY
);
```

```
CREATE TABLE produkty_w_zamowieniu (
  zamowienie_id INT,
  produkt_id INT,
  FOREIGN KEY (zamowienie_id) REFERENCES zamowienia(id)
);
```

Nie można dodać produktu do zamówienia, które nie istnieje.

✓ 5. DEFAULT

Ustawia **wartość domyślną**, jeśli użytkownik nie poda swojej.

Przykład:

```
CREATE TABLE artykuly (
  id INT PRIMARY KEY,
  status VARCHAR(20) DEFAULT 'aktywny'
```

);

Jeśli nie podasz statusu → automatycznie będzie „aktywny”.

✅ 6. CHECK

Wymusza spełnienie **logicznego warunku**.

Przykład:

```
CREATE TABLE pracownicy (  
  id INT PRIMARY KEY,  
  wiek INT CHECK (wiek >= 18 AND wiek <= 65)  
);
```

Próba dodania `wiek = 10` → ❌ błąd.

✅ 7. AUTO_INCREMENT

Automatycznie zwiększa wartość w kolumnie liczbowej przy każdym INSERT.

Przykład:

```
CREATE TABLE logi (  
  id INT PRIMARY KEY AUTO_INCREMENT,  
  opis VARCHAR(255)  
);
```

Dodajesz 5 logów → id będą: 1, 2, 3, 4, 5.

✅ 8. ENUM

Ogranicza wartości w kolumnie do **zamkniętej listy dopuszczalnych opcji**.

Przykład:

```
CREATE TABLE uzytkownicy (  
  id INT PRIMARY KEY,  
  plec ENUM('M', 'K', 'INNE') DEFAULT 'INNE'  
);
```

Próba zapisania `plec = 'ABC'` → ❌ błąd.

✅ 9. COMMENT

Pozwala dopisać **komentarz** do kolumny — bardzo przydatne przy dokumentowaniu schematu.

Przykład:

```
CREATE TABLE produkty (  
  id INT PRIMARY KEY,  
  cena DECIMAL(10,2) COMMENT 'Cena brutto w zł'  
);
```

W narzędziach typu phpMyAdmin, DBeaver zobaczysz komentarz przy kolumnie.

2 osoby - piszą sprawdzian na następnej lekcji Od GROUP BY lekcja