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_produktu INT ,
    ilosc INT,
    FOREIGN KEY (id_produktu) 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_produktu, 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 zamowienia 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

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:

1. **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 ;).
2. **CREATE PROCEDURE nazwa_procedury**: Definiuje nazwę procedury, która musi być unikalna w schemacie bazy danych.
3. **[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).
4. **[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:**

```sql
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:

```sql
/*
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:
        i. Oznacza, że procedura lub funkcja **zawiera instrukcje SQL, ale nie określa, czy odczytuje, czy modyfikuje dane.**
        ii. 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 zwracające 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 //

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
);
```

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
);
```

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_usunieciu
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 transakcje**
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 zapisan**e 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 ❌

🔹 **Przykład 4**

```
START TRANSACTION;

-- operacje 1
SAVEPOINT punkt1;

-- operacje 2
SAVEPOINT punkt2;

-- operacje 3
ROLLBACK TO punkt2;  -- cofa tylko operacje 3
ROLLBACK TO punkt1;  -- cofa także operacje 2, ale nie operacje 1

COMMIT;          -- zapisuje wszystko do punktu1
```

🧩 **RELEASE usunięcie SAVEPOINT ;**

```
RELEASE SAVEPOINT po_pierwszej_operacji;
```

🧠**Ważne uwagi**

1. **Działa tylko w silnikach transakcyjnych** — np. InnoDB.
   Jeśli używasz MyISAM, transakcje (a więc i ROLLBACK) **nie działają**.

2. **Autocommit**:
   Domyślnie MySQL działa w trybie autocommit = 1, co oznacza, że każda instrukcja SQL jest automatycznie zatwierdzana po wykonaniu.
   Aby używać transakcji, musisz:

   ```
   SET autocommit = 0;
   ```

Lekcja

Temat: Having, funkcje agregujące. Przykłady zapytań z datami, kwartałami 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(ilosc * kwota) AS laczna_kwota
FROM zamowienia
WHERE status = 'zrealizowane'
GROUP BY id_klienta;
```

id_klienta	laczna_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(ilosc * kwota),2) AS srednia_wartosc_zamowienia
FROM zamowienia
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 zamowienia
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 *realnego* wykonania	2025-11-05 14:32:11
CURTIME()	Tylko czas	14:32:11