

# PRZEDMIOT: Baz danych

KLASA: 5i gr. 2

## Tydzień 1 Lekcja 1,2


**Temat:** Definicja Baz Danych. Powtórzenie terminów tabele, rekordy, pola. Relację między tabelami: 1:1, 1:N, N:M. Nadawanie, odbieranie uprawnień (GRANT, REVOKE )

### Definicja bazy danych i jej znaczenie:

#### Definicja bazy danych:


Baza danych to cyfrowy, uporządkowany zbiór informacji, zapisany i przechowywany w sposób ustrukturyzowany, który umożliwia łatwe i szybkie wyszukiwanie, pobieranie, dodawanie, modyfikowanie i usuwanie danych.


#### Znaczenie bazy danych:


 **Przechowywanie danych** – umożliwia gromadzenie dużych ilości informacji w jednym miejscu.


 **Szybki dostęp i wyszukiwanie** – dzięki językom zapytań (np. SQL) można błyskawicznie znaleźć potrzebne dane.

 **Relacje i spójność** – pozwala łączyć dane ze sobą (np. klient ↔ zamówienia), zachowując integralność.

 **Wielu użytkowników** – umożliwia jednoczesną pracę wielu osób/ aplikacji z tymi samymi danymi.

 **Bezpieczeństwo** – zapewnia mechanizmy kontroli dostępu i ochrony przed utratą danych.

 **Aktualność** – zmiany wprowadzane w jednym miejscu są natychmiast widoczne dla wszystkich użytkowników.

 **Uniwersalność** – używane w niemal każdej dziedzinie (bankowość, handel, medycyna, edukacja, serwisy internetowe).

## Bazy danych można podzielić według sposobu organizacji i przechowywania danych:

### ♦ 1. Bazy relacyjne (RDB – Relational Database)

- ☐ Najpopularniejszy typ.
- ☐ Dane są przechowywane w tabelach (wiersze = rekordy, kolumny = pola).
- ☐ Tabele są powiązane kluczami (np. użytkownik → zamówienia).
- ☐ Do zarządzania używa się języka SQL.
- ☐ Przykłady: MySQL, PostgreSQL, Oracle, MS SQL Server.

### ♦ 2. Bazy nierelacyjne (NoSQL)

- ☐ Dane przechowywane w innych formach niż tabele.
- ☐ Rodzaje/modele:
  - **Dokumentowe** dane przechowywane w formie **dokumentów** (np. JSON, BSON, XML).
  - **Grafowe** - dane są przechowywane w postaci grafu (Neo4j – dane jako grafy),
  - **Klucz–wartość** - dane przechowywane jako para: **klucz** → **wartość**. (Redis, DynamoDB),
  - **Kolumnowe** - dane zapisane w **kolumnach** zamiast wierszy (odwrotnie niż w SQL) (Cassandra, HBase).

### ♦ 3. Bazy obiektowe

- ☐ Dane przechowywane jako **obiekty** (tak jak w programowaniu obiektowym).
- ☐ Mogą przechowywać nie tylko liczby i tekst, ale także multimedia czy złożone struktury.
- ☐ Przykład: db4o, ObjectDB.

### ♦ 4. Bazy obiektowo-relacyjne

- ☐ Hybryda relacyjnych i obiektowych.
- ☐ Dane przechowywane są w postaci obiektów
- ☐ Obsługują tabele, ale także bardziej złożone typy danych.
- ☐ Przykład: PostgreSQL, Oracle.

### ♦ 5. Bazy hierarchiczne

- ☐ Dane są zorganizowane w strukturę **drzewa** (rodzic–dziecko).
- ☐ Każdy rekord ma jeden nadrzędny i wiele podrzędnych.
- ☐ Szybki dostęp, ale trudne do modyfikacji, mało elastyczne.
- ☐ Przykład: IBM IMS (starsze systemy bankowe).

## ♦ 5. Bazy sieciowe

- ☐ Dane zorganizowane w strukturze przypominającej **sieć** lub **graf** – rekordy mogą mieć wielu rodziców i wielu potomków.
- ☐ Stanowią one rozwinięcie modelu hierarchicznego
- ☐ Pozwalają na reprezentację danych, gdzie **jeden element może być powiązany z wieloma innymi elementami, a te z kolei mogą być powiązane z wieloma kolejnymi elementami**, tworząc złożoną, grafową strukturę.
- ☐ Przykład: IDS (Integrated Data Store).

## ♦ 6. Bazy rozproszone

- ☐ Dane nie są przechowywane w jednym miejscu (na jednym serwerze), tylko **rozsiane po wielu komputerach/serwerach**, często w różnych lokalizacjach geograficznych.
- ☐ Łatwo dodać nowe serwery, gdy rośnie liczba danych.
- ☐ Dane są **podzielone na części** i każda część jest przechowywana na innym serwerze pp. użytkownicy A–M są na serwerze 1, a N–Z na serwerze 2.

## Tworzenie nowej bazy:

### 1 Microsoft Access

- Access jest bazą plikową, więc baza danych to plik .accdb.
- Tworzenie bazy odbywa się **graficznie**, ale można też użyć SQL do tworzenia tabel w już otwartym pliku.

👉 Tworzenie nowej bazy:

1. Otwórz **Access** → **Plik** → **Nowy** → **Pusta baza danych**
2. Nadaj nazwę, np. **Sklep.accdb**
3. Access utworzy plik bazy danych i otworzy pustą bazę.

**W Access SQL nie ma polecenia typu CREATE DATABASE, bo baza to plik.**  
SQL w Access służy głównie do tworzenia tabel, zapytań, widoków.

### 2 PostgreSQL

- PostgreSQL jest serwerową bazą danych.
- Tworzenie bazy odbywa się komendą **CREATE DATABASE**.

👉 Przykład:

```
CREATE DATABASE sklep
WITH
OWNER = postgres
ENCODING = 'UTF8'
LC_COLLATE = 'pl_PL.UTF-8'
LC_CTYPE = 'pl_PL.UTF-8'
TEMPLATE = template0;
```

- **OWNER** – właściciel bazy (użytkownik PostgreSQL)
- **ENCODING** – kodowanie znaków
- **LC\_COLLATE** i **LC\_CTYPE** – lokalizacja i sortowanie znaków
- **TEMPLATE** – szablon bazy (zwykle template0 lub template1)

## Omówienie podstawowych koncepcji: tabele, rekordy, pola

### 📌 1. Tabela

To główna struktura w relacyjnej bazie danych. Można ją porównać do arkusza w Excelu – ma wiersze i kolumny. Każda tabela przechowuje dane dotyczące jednego typu obiektów.

👉 Przykład: Tabela Studenci przechowuje informacje o studentach.

### 📌 2. Rekord (wiersz, ang. row/record)

Pojedynczy wiersz w tabeli. Odpowiada jednej jednostce danych (np. jednemu studentowi). Składa się z pól (kolumn).

👉 Przykład rekordu w tabeli Studenci:


ID	Imię	Nazwisko	Wiek	Kierunek
1	Anna	Kowalska	21	Informatyka

Ten jeden wiersz to rekord opisujący Annę Kowalską.

### 3. Pole (kolumna, ang. field/column)

**To kolumna w tabeli, przechowująca określony typ danych.**

Każde pole ma nazwę i jest określonego typu danych (np. liczba, tekst, data).

 Przykłady pól w tabeli Studenci:

Imię – tekst,

Nazwisko – tekst,

Wiek – liczba całkowita,

Kierunek – tekst.

## Klucze

### Klucz główny (Primary Key, PK)

**To unikalny identyfikator rekordu w tabeli.**

Gwarantuje, że każdy wiersz można jednoznacznie odróżnić.

Kluczem głównym może być:

- ☐ liczba całkowita (np. ID = 1, 2, 3...),
- ☐ unikalny kod (np. PESEL, NIP),

 W tabeli Studenci:

ID	Imię	Nazwisko	Wiek
1	Anna	Kowalska	21

Tutaj ID jest kluczem głównym.

### Klucz obcy (Foreign Key, FK)

**To pole w tabeli, które wskazuje na klucz główny w innej tabeli.**

Dzięki temu możemy powiązać dane między tabelami.

 Przykład:

Tabela Zapisy (które kursy student wybrał) może mieć klucze obce:

StudentID → odwołanie do tabeli Studenci(ID),

KursID → odwołanie do tabeli Kursy(ID).

✓ **Podsumowanie w skrócie:**

- ☐ **Relacyjna baza danych** – dane w tabelach powiązane relacjami.
- ☐ **PK** – unikalny identyfikator w tabeli.
- ☐ **FK** – łączy jedną tabelę z drugą.

📌 **3. Relacje między tabelami**

1 **Jeden do jednego (1:1)**

**Każdy rekord w jednej tabeli odpowiada dokładnie jednemu rekordowi w drugiej.**

👉 Przykład: Osoby ↔ Pesel. Jedna osoba ma tylko jeden Pesel.

**Tabela: Osoby**

id_osoba	imie	nazwisko
1	Adam	Kowalski
2	Anna	Nowak
3	Patryk	Balicki

**Tabela: Pesele**

id_pesel	pesel	id_osoby
1	80010112345	1
2	92051267890	2
3	75032145678	3

## 2 Jeden do wielu (1:N)

**Jeden rekord w tabeli A może mieć wiele rekordów w tabeli B. Ale rekord w tabeli B należy tylko do jednego w tabeli A.**

👉 Przykład: Nauczyciele ↔ Przedmioty. Jeden nauczyciel prowadzi wiele przedmiotów, ale każdy przedmiot ma tylko jednego nauczyciela.

### ♦ Opis relacji

- **Jeden nauczyciel może uczyć wiele przedmiotów.**
- **Ale jeden przedmiot ma przypisanego tylko jednego nauczyciela.**

**Tabela: Nauczyciele**

id_nauczyciela	imie	nazwisko
1	Adam	Kowalski
2	Anna	Nowak
3	Patryk	Balicki

**Tabela: Przedmioty**

id_przedmiotu	nazwa	id_nauczyciela
1	Systemy Baz Danych	1
2	Matematyka	2
3	Fizyka	3
4	Chemia	1

### 3 Wiele do wielu (M:N)

**Rekordy w tabeli A mogą być powiązane z wieloma rekordami w tabeli B i odwrotnie.**

👉 Przykład:

Uczniowie ↔ Przedmioty. Uczeń może zapisać się na wiele przedmiotów, a przedmiot może mieć wielu uczniów.

Rozwiązanie: Tabela Zapisy z polami: id\_ucznia (FK do tabeli Uczniowie)  
id\_przedmiotu (FK do tabeli Przedmioty ). Trzeba pamiętać, że jednego ucznia nie można przypisać wiele razy do tego samego przedmiotu

**Tabela: Uczniowie**

id_ucznia	imie	nazwisko
1	Adam	Kowalski
2	Anna	Nowak
3	Patryk	Balicki

**Tabela: Przedmioty**

id_przedmiotu	nazwa
1	Systemy Baz Danych
2	Matematyka
3	Fizyka
4	Chemia



## Tabela Zapisy (tabela pośrednia)

id_przedmiotu	id_ucznia
1	1
2	1
3	1
2	1
2	2
2	3
3	3
4	1

## ◆ Podstawowe polecenia do sortowania

### ORDER BY

```
SELECT nazwisko, imie  
FROM pracownicy  
ORDER BY nazwisko ASC; -- rosnąco
```

```
SELECT nazwisko, imie  
FROM pracownicy  
ORDER BY nazwisko DESC; -- malejąco
```

### Sortowanie po wielu kolumnach

```
SELECT nazwisko, imie, pensja  
FROM pracownicy  
ORDER BY nazwisko ASC, pensja DESC;
```

👉 Najpierw sortuje po nazwisku rosnąco, a w ramach tego – po pensji malejąco.

## Zarządzanie bezpieczeństwem bazy danych.

### ♦ Definicje

- **GRANT** – służy do nadawania uprawnień użytkownikom bazy danych (np. prawa do odczytu, zapisu, aktualizacji, usuwania, tworzenia tabel).
- **REVOKE** – służy do odbierania wcześniej nadanych uprawnień.

### ♦ Składnia

#### Nadawanie uprawnień (GRANT)

```
GRANT <uprawnienia>  
ON <nazwa_bazy_danych>.<nazwa_tabeli>  
TO <nazwa_uzytkownika>@<host>;
```

#### Odbieranie uprawnień (REVOKE)

```
REVOKE <uprawnienia>  
ON <nazwa_bazy_danych>.<nazwa_tabeli>  
FROM <nazwa_uzytkownika>@<host>;
```

## Tydzień 2 Lekcja 3,4

**Temat:** SQL JOINS, CONSTRAINT. Zastosowanie wyświetlania liczb porządkowych dla wszystkich wierszy  
ROW\_NUMBER()

```
DROP TABLE Przedmioty;  
DROP TABLE Osoby;
```

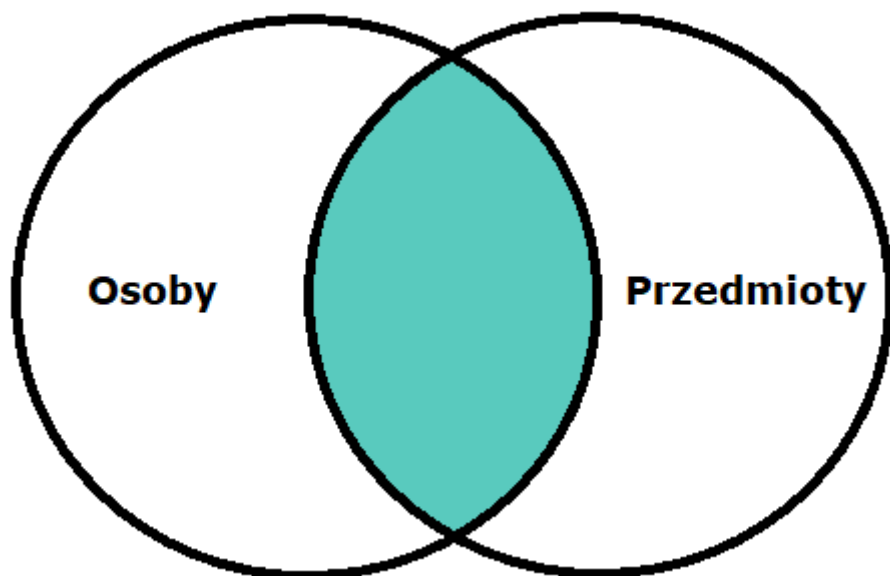
```
CREATE TABLE Osoby (  
    osoba_id INT AUTO_INCREMENT PRIMARY KEY,  
    imie VARCHAR(50) NOT NULL,  
    nazwisko VARCHAR(50) NOT NULL  
);
```

```
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);
```

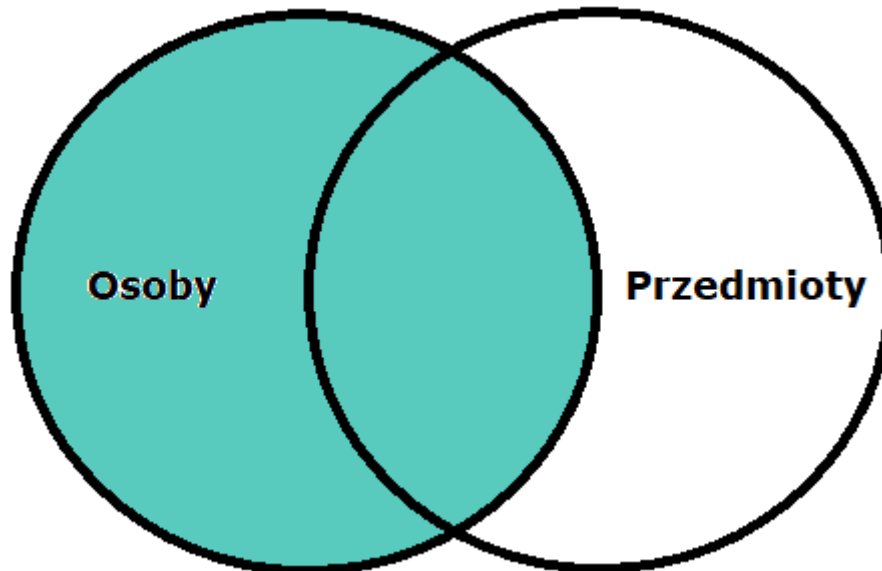
- ◆ **INNER JOIN** - czyli wszystkie wspólne rekordy, bez NULL



```
select Osoby.imie, Osoby.nazwisko, Przedmioty.nazwa  
from Osoby  
inner join Przedmioty on Osoby.osoba_id = Przedmioty.osoba_id;
```

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.

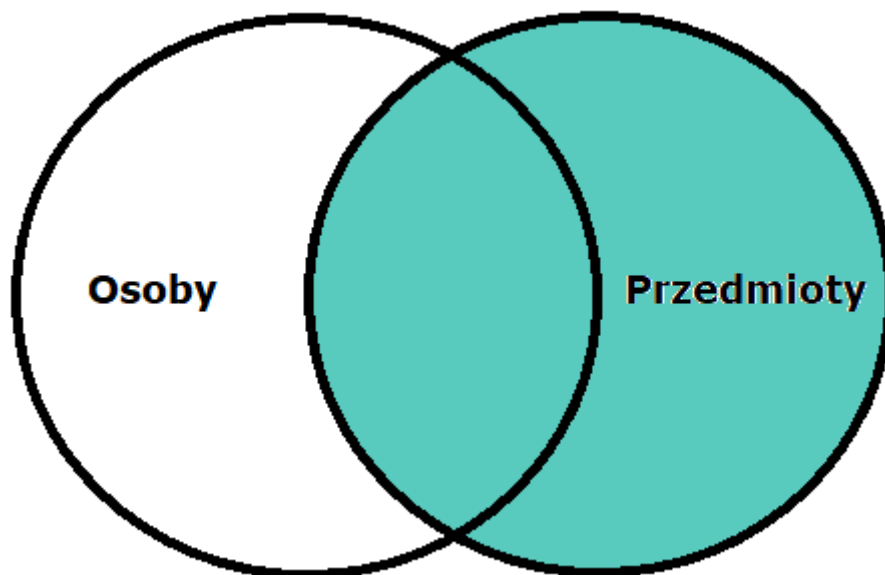


```
SELECT Osoby.imie, Osoby.nazwisko, Przedmioty.nazwa  
FROM Osoby  
LEFT JOIN Przedmioty ON Osoby.osoba_id = Przedmioty.osoba_id;
```

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.



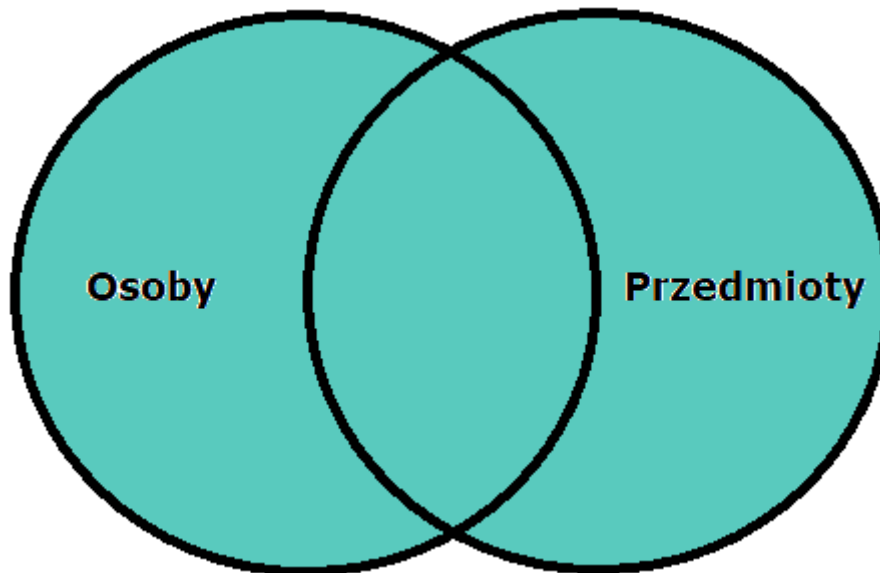
```
SELECT Osoby.imie, Osoby.nazwisko, Przedmioty.nazwa  
FROM Osoby  
RIGHT JOIN Przedmioty ON Osoby.osoba_id = Przedmioty.osoba_id;
```

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

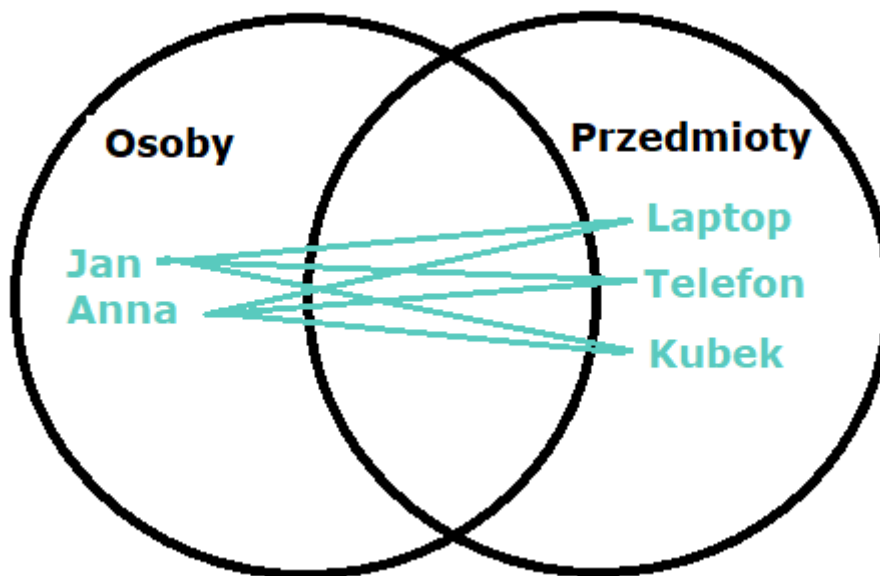
```
SELECT o.imie, o.nazwisko, p.nazwa  
FROM Osoby o  
RIGHT JOIN Przedmioty p ON o.osoba_id = p.osoba_id;
```

Wynik:

imie	nazwisko	nazwa
Jan	Kowalski	Laptop
Jan	Kowalski	Telefon
Anna	Nowak	Rower
Piotr	Zieliński	Książka

Kasia	Wiśniewska	Plecak
Patryk	Nowakowski	<i>NULL</i>
<i>NULL</i>	<i>NULL</i>	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;
```

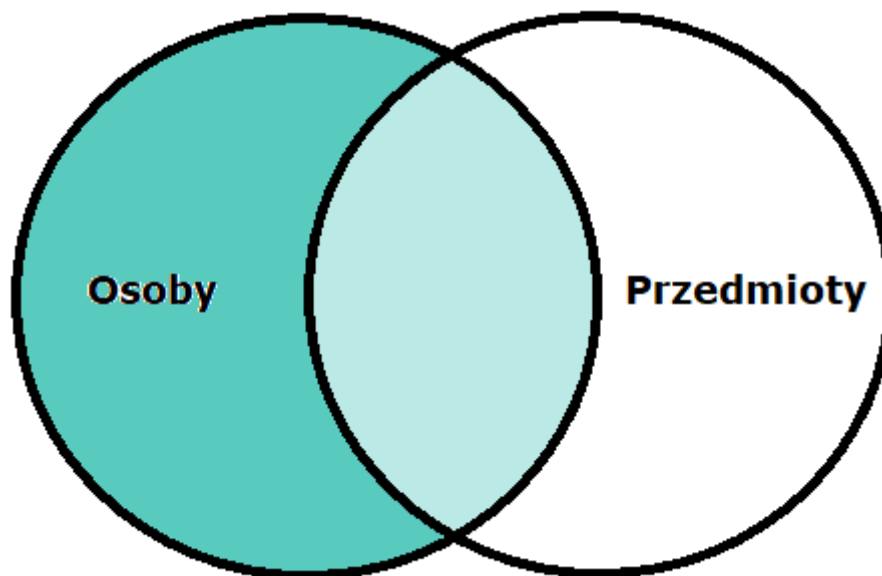
Wynik:

<b>imie</b>	<b>nazwa</b>
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

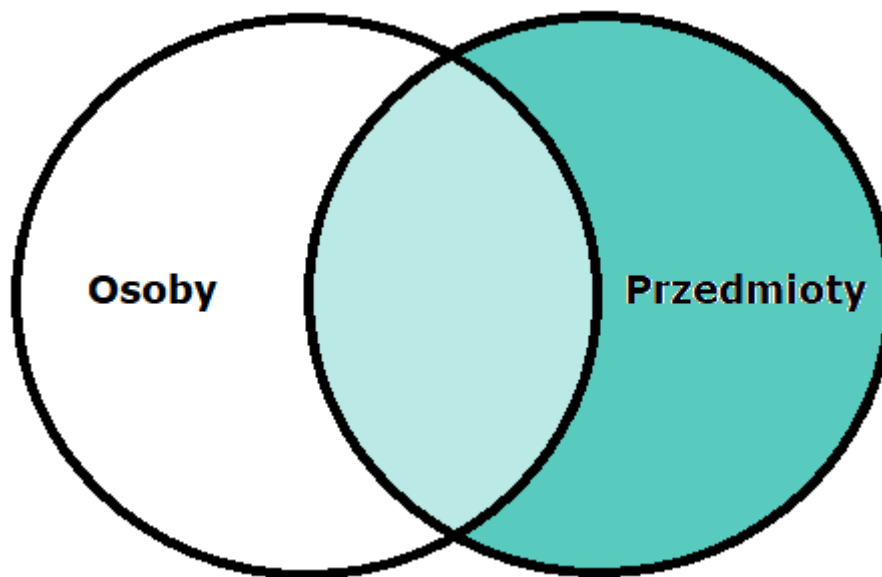


```
SELECT o.imie, o.nazwisko, p.nazwa  
FROM Osoby o  
LEFT JOIN Przedmioty p ON o.osoba_id = p.osoba_id  
WHERE p.osoba_id IS NULL;
```

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



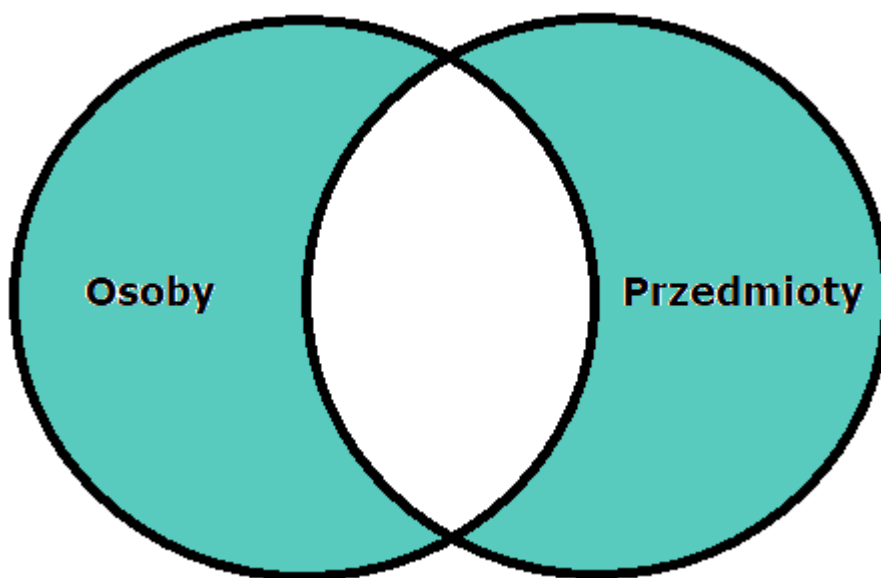
```
SELECT o.imie, o.nazwisko, p.nazwa  
FROM Osoby o  
RIGHT JOIN Przedmioty p ON o.osoba_id = p.osoba_id  
WHERE o.osoba_id IS NULL;
```

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	<i>NULL</i>
<i>NULL</i>	<i>NULL</i>	Kubek

## CONSTRAINT

W MySQL jeśli sam nie dodasz CONSTRAINT zostaje automatycznie dodany z nazwą.

Polecenie:

```
SHOW CREATE TABLE nazwa_tabeli;
```

**zwraca pełną instrukcję polecenia CREATE TABLE**

- **zwraca nazwę** CONSTRAINT
- **nazwy kolumn,**
- **typy danych,**
- **klucze** (PRIMARY KEY, FOREIGN KEY, UNIQUE, INDEX),
- **ustawienia tabeli** (ENGINE=InnoDB, DEFAULT CHARSET, itp.).

## Wyświetlenie liczby porządkowej dla każdego rekordu

### 1. Za pomocą ROW\_NUMBER()

```
SELECT
  ROW_NUMBER() OVER (ORDER BY o.osoba_id) AS lp,
  o.imie,
  o.nazwisko,
  p.nazwa
FROM Osoby o
INNER JOIN Przedmioty p ON o.osoba_id = p.osoba_id;
```

### 2. Za pomocą zmiennej sesyjnej

```
SET @lp := 0;

SELECT
  @lp := @lp + 1 AS lp,
  o.imie,
  o.nazwisko,
  p.nazwa
FROM Osoby o
INNER JOIN Przedmioty p ON o.osoba_id = p.osoba_id;
```

## Tydzień 2 Lekcja 5,6

**Temat:** Obliczanie sumy

```
CREATE TABLE Klienci (  
    klient_id INT PRIMARY KEY AUTO_INCREMENT,  
    imie VARCHAR(50) NOT NULL,  
    nazwisko VARCHAR(50) NOT NULL  
);
```

```
CREATE TABLE Zakupy (  
    zakup_id INT PRIMARY KEY AUTO_INCREMENT,  
    klient_id INT NOT NULL,  
    kwota DECIMAL(10,2) NOT NULL,  
    FOREIGN KEY (klient_id) REFERENCES Klienci(klient_id)  
);
```

```
CREATE TABLE Zwroty (  
    zwrot_id INT PRIMARY KEY AUTO_INCREMENT,  
    klient_id INT NOT NULL,  
    kwota DECIMAL(10,2) NOT NULL,  
    FOREIGN KEY (klient_id) REFERENCES Klienci(klient_id)  
);
```

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

```
INSERT INTO Zakupy (klient_id, kwota) VALUES  
(1, 300.00),    -- Jan Kowalski  
(1, 150.00),    -- Jan Kowalski  
(2, 500.00),    -- Anna Nowak  
(3, 200.00);    -- Piotr Wiśniewski
```

```
INSERT INTO Zwroty (klient_id, kwota) VALUES  
(1, 50.00),     -- Jan Kowalski  
(2, 100.00);    -- Anna Nowak
```

## 1. Suma zakupów i zwrotów per klient

```
SELECT k.klient_id, k.imie, k.nazwisko,  
       COALESCE(s.kwota, 0) AS suma_zakupow,  
       COALESCE(z.kwota, 0) AS suma_zwrotow  
FROM Klienci k  
LEFT JOIN (  
    SELECT klient_id, SUM(kwota) AS kwota  
    FROM Zakupy  
    GROUP BY klient_id  
) s ON k.klient_id = s.klient_id  
LEFT JOIN (  
    SELECT klient_id, SUM(kwota) AS kwota  
    FROM Zwroty  
    GROUP BY klient_id  
) z ON k.klient_id = z.klient_id;
```

lub

```
SELECT k.imie, k.nazwisko,  
       COALESCE((SELECT SUM(kwota) FROM Zakupy WHERE klient_id = k.klient_id),  
0) AS suma_zakupow,  
       COALESCE( (SELECT SUM(kwota) FROM Zwroty WHERE klient_id = k.klient_id),  
0) AS suma_zwrotow  
FROM Klienci k;
```

Wynik:

klient_id	imie	nazwisko	suma_zakupow	suma_zwrotow
1	Jan	Kowalski	450	50
2	Anna	Nowak	500	100
3	Piotr	Wiśniewski	200	0



## 2. Bilans (zakupy – zwroty)

```
SELECT k.imie, k.nazwisko,  
       COALESCE(SUM(zak.kwota), 0) - COALESCE(SUM(zw.kwota), 0) AS bilans  
FROM Klienci k  
LEFT JOIN Zakupy zak ON k.klient_id = zak.klient_id  
LEFT JOIN Zwroty zw ON k.klient_id = zw.klient_id  
GROUP BY k.klient_id, k.imie, k.nazwisko;
```

Wynik:

imie	nazwisko	bilans
Jan	Kowalski	400
Anna	Nowak	400
Piotr	Wiśniewski	200

## 1. Suma zakupów, ilość zakupów i zwrotów per klient

```
SELECT k.klient_id, k.imie, k.nazwisko,  
       COALESCE(s.suma_zakupow, 0) AS suma_zakupow,  
       COALESCE(s.ilosc_zakupow, 0) AS ilosc_zakupow,  
       COALESCE(z.suma_zwrotow, 0) AS suma_zwrotow  
FROM Klienci k  
LEFT JOIN (  
    SELECT klient_id,  
           SUM(kwota) AS suma_zakupow,  
           COUNT(*) AS ilosc_zakupow  
    FROM Zakupy  
    GROUP BY klient_id  
) s ON k.klient_id = s.klient_id  
LEFT JOIN (  
    SELECT klient_id,  
           SUM(kwota) AS suma_zwrotow  
    FROM Zwroty  
    GROUP BY klient_id  
) z ON k.klient_id = z.klient_id;
```

Wynik:

klient_id	imie	nazwisko	suma_zakupow	ilosc_zakupow	suma_zwrotow
1	Jan	Kowalski	450	2	50
2	Anna	Nowak	500	1	100
3	Piotr	Wiśniewski	200	1	0

## ♦ 1. Funkcje warunkowe w IF

### ✓ IFNULL(expr, value)

```
SELECT IFNULL(SUM(kwota), 0) AS suma_zakupow  
FROM Zakupy;
```

### ✓ IF(expr, true\_value, false\_value)

```
SELECT IF(SUM(kwota) IS NULL, 0, SUM(kwota)) AS suma_zakupow  
FROM Zakupy;
```

### ✓ NULLIF(expr1, expr2)

```
SELECT NULLIF(kwota, 0) AS wynik  
FROM Zakupy;
```

➡ Jeśli `kwota = 0`, wynik to `NULL`.

## ✓ CASE WHEN ... THEN ... ELSE ... END

SELECT

CASE

WHEN kwota > 500 THEN 'VIP zakup'

WHEN kwota > 100 THEN 'średni zakup'

ELSE 'mały zakup'

END AS kategoria

FROM Zakupy;

## Tydzień 3 Lekcja 7,8

### Temat: Kategorie poleceń. Procedury i funkcje

#### 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 i funkcje

**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 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);***.

### Przykład procedury:

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.

### 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 //**: Jak w procedurach, zmienia 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):
  - DETERMINISTIC: Funkcja zwraca ten sam wynik dla tych samych danych wejściowych (zalecane dla optymalizacji).
  - NOT DETERMINISTIC: Wynik może się różnić (np. dla funkcji używających RAND()).
  - Inne modyfikatory, jak w procedurach.
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

## Tydzień 3 Lekcja 9,10

**Temat:** Definicja podzapytań skorelowanych w SQL i MySQL. Funkcje agregujące w SQL – Użycie funkcji takich jak COUNT, SUM, AVG, MIN, MAX oraz grupowanie danych (GROUP BY, HAVING).

♦ **Podzapytanie skorelowane** (ang. *correlated subquery*) to **rodzaj podzapytania** w języku SQL, **które jest zależne od wartości z zapytania zewnętrznego**.

**Oznacza to, że podzapytanie odnosi się do kolumn lub wartości z tabeli zapytania głównego, co powoduje, że jest ono wykonywane wielokrotnie** – raz dla każdego wiersza przetwarzanego przez zapytanie zewnętrzne. W przeciwieństwie do podzapytań nieskorelowanych (nieskorelowanych subqueries), które są wykonywane tylko raz i niezależnie od zapytania zewnętrznego.

Podzapytania skorelowane mogą być mniej wydajne, ponieważ wymagają iteracji po wierszach.

### ♦ Różnica między podzapytaniem skorelowanym a nieskorelowanym

- **Nieskorelowane:** **Podzapytanie jest samodzielne**, np. zwraca stałą wartość lub listę, która jest używana w zapytaniu zewnętrznym. **Wykonywane raz.**
- **Skorelowane:** **Podzapytanie używa wartości z zewnętrznego zapytania** (np. poprzez alias tabeli zewnętrznej), co sprawia, że jest "skorelowane" z każdym wierszem zewnętrznym. **Wykonywane wielokrotnie.**

### ✓ Przykłady

Zakładamy prostą bazę danych z dwiema tabelami:

- `pracownicy` (id, imie, pensja, dzial\_id)
- `dzialy` (id, nazwa, srednia\_pensja)

```

CREATE TABLE dzialy (
    id INT PRIMARY KEY,
    nazwa VARCHAR(100) NOT NULL,
    srednia_pensja DECIMAL(10,2)
);

CREATE TABLE pracownicy (
    id INT PRIMARY KEY,
    imie VARCHAR(100) NOT NULL,
    pensja DECIMAL(10,2) NOT NULL,
    id_dzial INT ,
    FOREIGN KEY (id_dzial) REFERENCES dzialy(id)
);

INSERT INTO dzialy (id, nazwa, srednia_pensja) VALUES
(1, 'IT', NULL),
(2, 'Finanse', NULL),
(3, 'Marketing', NULL);

INSERT INTO pracownicy (id, imie, pensja, id_dzial) VALUES
(1, 'Adam', 8000.00, 1),
(2, 'Beata', 9500.00, 1),
(3, 'Cezary', 7200.00, 1),
(4, 'Daria', 6500.00, 2),
(5, 'Edward', 7000.00, 2),
(6, 'Fiona', 9000.00, 2),
(7, 'Grzegorz', 6000.00, 3),
(8, 'Hanna', 7500.00, 3),
(9, 'Igor', 5000.00, 3);

```

### Przykład 1: Podstawowe podzapytanie skorelowane (używane w klauzuli WHERE)

To zapytanie znajduje pracowników, których pensja jest wyższa niż średnia pensja w ich własnym dziale.

```

SELECT imie, pensja, id_dzial
FROM pracownicy p
WHERE pensja > (
    SELECT AVG(pensja)
    FROM pracownicy
    WHERE id_dzial= p.id_dzial -- Tutaj skorelowane: odnosi się do
    'p.dzial_id' z zewnętrznego zapytania
);

```

### ♦ Optymalizacja 1 – JOIN + GROUP BY

```
SELECT imie, pensja, id_dzial
FROM pracownicy e
JOIN (
    SELECT id_dzial, AVG(pensja) AS srednia
    FROM pracownicy
    GROUP BY id_dzial
) s ON e.id_dzial = s.id_dzial
WHERE e.pensja > s.srednia;
```

✓ Tutaj **AVG()** liczony jest tylko raz dla każdego działu, a nie powtarzany w pętli.

### ♦ Optymalizacja 2 – WITH (czytelniejsza wersja)

```
WITH srednie AS (
    SELECT id_dzial, AVG(pensja) AS srednia
    FROM pracownicy
    GROUP BY id_dzial
)
SELECT p.imie, p.pensja, p.id_dzial
FROM pracownicy p
JOIN srednie s ON p.id_dzial = s.id_dzial
WHERE p.pensja > s.srednia;
```

👉 **WITH** pozwala zdefiniować **tympczasowy zestaw danych (jakby wirtualną tabelę)**, który możesz potem wykorzystać w głównym zapytaniu.

Możesz to traktować jak **alias dla podzapytania**, tyle że:

- jest bardziej czytelny,
- można go wielokrotnie używać w tym samym zapytaniu,
- może być rekurencyjny (np. do pracy z hierarchiami: drzewami, strukturami organizacyjnymi).

### ♦ Optymalizacja 3 – WINDOW FUNCTION (najlepsza, MySQL 8+)

```
SELECT imie, pensja, id_dzial
FROM (
    SELECT p.*,
           AVG(pensja) OVER (PARTITION BY id_dzial) AS srednia
    FROM pracownicy p
) t
WHERE pensja > srednia;
```

podpowiedzi:

```
// Window functions (funkcje okienkowe, w SQL nazywane też analytical functions) to mechanizm, który pozwala wykonywać obliczenia na zbiorze wierszy powiązanych z bieżącym wierszem – bez grupowania całej tabeli.  
// OVER → ale nie dla całej tabeli, tylko w „oknie”,  
// PARTITION BY id_dzial → podziel dane na grupy według id_dzial (czyli każdy dział to osobne „okno”),
```

✅ Zalety:

- Nie trzeba pisać JOIN ani GROUP BY.
- Baza najpierw liczy średnią dla działów, a potem filtruje wiersze.
- Bardzo szybkie na dużych danych.

## Przykład 2: Podzapytanie skorelowane w klauzuli SELECT

To zapytanie wyświetla działy wraz z liczbą pracowników zarabiających powyżej średniej w danym dziale.

```
SELECT nazwa,  
       (SELECT COUNT(*)  
        FROM pracownicy p  
        WHERE p.id_dzial = d.id AND p.pensja > d.srednia_pensja --  
        Skorelowane: odnosi się do 'd.id' i 'd.srednia_pensja'  
       ) AS liczba_nad_srednia  
FROM dzialy d;
```

## Przykład 3: Podzapytanie skorelowane z EXISTS (sprawdzenie istnienia)

To zapytanie znajduje działy, które mają co najmniej jednego pracownika z pensją powyżej 5000.

```
SELECT nazwa  
FROM dzialy d  
WHERE EXISTS (  
    SELECT 1  
    FROM pracownicy p  
    WHERE p.id_dzial = d.id AND p.pensja > 5000 -- Skorelowane: odnosi się  
    do 'd.id'  
);
```