

# PRZEDMIOT: Systemy baz danych

KLASA: 2A 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. Polecenie Order By. Nadawanie, odbieranie uprawnień (GRANT, REVOKE ). Pojęcie CRUD

### 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/modelce:
  - **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.



## 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.

📌 Sortować można po **numerach kolumn** (niezalecane, ale działa):


```
SELECT nazwisko, imie  
FROM pracownicy  
ORDER BY 2 ASC, 4 DESC
```

👉 2 kolumną będzie imie. Nie bierze pod uwagę faktyczna kolejność kolumn w tabeli



## 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>;
```

### CRUD

CRUD to skrót od angielskich słów:


- **C – Create** → tworzenie nowych rekordów (np. **INSERT**)
- **R – Read** → odczytywanie danych (np. **SELECT**)
- **U – Update** → aktualizowanie istniejących rekordów (np. **UPDATE**)
- **D – Delete** → usuwanie rekordów (np. **DELETE**)

### Odpowiedniki w SQL

- **Create** → **INSERT INTO uczniowie (...) VALUES (...)**
- **Read** → **SELECT \* FROM uczniowie**
- **Update** → **UPDATE uczniowie SET klasa='3B' WHERE id=1**
- **Delete** → **DELETE FROM uczniowie WHERE id=1**

## Tydzień 2 Lekcja 3

### Temat: Struktura Bazy Danych MySQL

 **Schemat bazy danych to struktura i organizacja bazy danych, która definiuje jej tabele, pola, relacje, ograniczenia i typy danych.** Organizacja baz danych może się różnić od siebie.

**MySQL ma silnik InnoDB.**

InnoDB zarządza danymi na własny sposób, korzystając z **tablespace (przestrzeni tabel)**, które są fizycznymi plikami na dysku. W ich wnętrzu dane są przechowywane w postaci stron (ang. *pages*) i segmentów.

```

+=====+
|                                     |
|          TABLESPACE (np. .ibd / ibdata1)          |
|-----|
| SEGMENT DANYCH (indeks klastrowany = PRIMARY KEY) |
| └─ Extent #1 (1 MB) └─ Page (16 KB) → rekordy      |
| |                                     |
| |                                     └─ Page (16 KB) → rekordy |
| |                                     └─ ...                |
| └─ Extent #2 (1 MB) └─ Page (16 KB) → rekordy      |
| |                                     └─ ...                |
| └─ ...                                                |
|-----|
| SEGMENT INDEKSU POMOCNICZEGO (np. idx_nazwisko)    |
| └─ Extent #1 (1 MB) └─ Page (16 KB) → węzły B-Tree (klucze→PK) |
| |                                     └─ ...                |
| └─ Extent #2 (1 MB) └─ Page (16 KB) → węzły B-Tree      |
| |                                     └─ ...                |
|-----|
| SEGMENT UNDO (dla cofania transakcji)                |
| └─ Extent #1 (1 MB) └─ Page (16 KB) → wpisy UNDO      |
| |                                     └─ ...                |
| └─ ...                                                |
|-----|
| [inne segmenty/metadane; wolne extenty do przydziału] |
+=====+

```



👉 Tabela **zawsze ma co najmniej 2 segmenty** – jeden na dane i jeden na indeksy.

## Gdzie fizycznie są dane tabeli w InnoDB?

### 1. W pliku tablespace:

- jeśli masz `innodb_file_per_table=ON` (domyślnie) → każda tabela ma **własny plik .ibd**,
- jeśli `innodb_file_per_table=OFF` → wszystkie tabele są w **system tablespace (ibdata1)**.



#### Plik .ibd (tablespace dla tabeli "users")

- └─ **Segment** danych (PRIMARY KEY = indeks klastrowany)
  - └─ **Extenty** (1 MB = 64 stron)
    - └─ **Strony** (16 KB każda)
      - └─ **Page 1** → rekordy użytkowników (np. **id=1...20**)
      - └─ **Page 2** → rekordy użytkowników (np. **id=21...40**)
      - └─ ...

## Rodzaje przestrzeni tabel:

- **System tablespace** – główna przestrzeń tabel (zwykle ibdata1) zawierająca metadane, UNDO logi, i ewentualnie dane tabel, jeśli nie korzystasz z trybu „file-per-table”.
- **File-per-table tablespace** – osobny plik .ibd dla każdej tabeli (jeśli włączone `innodb_file_per_table=ON`).
- **Temporary tablespaces** – dla tabel tymczasowych.

- **Undo tablespaces** – do przechowywania danych potrzebnych przy cofnięciu transakcji.

## ◆ Extent

- **Extent** = blok ciągłych stron (*pages*).
- Rozmiar: **1 MB = 64 strony po 16 KB**.
- Extent to czysto fizyczne pojęcie – sposób zarządzania przestrzenią w tablespace.
- Extenty są jednostką, którą InnoDB rezerwuje i przydziela tabelom lub indeksom.

👉 Możesz to porównać do „klocka” miejsca na dysku.

## ◆ Segment

- **Segment** = logiczna struktura w InnoDB, zbudowana z extentów.
- Segmenty są używane do przechowywania różnych rzeczy, np.:
  - segment **danych** (rekordy tabeli),
  - segment **indeksów**,
  - segment **undo logu**.
- Każda tabela w InnoDB ma co najmniej **2 segmenty**:
  - segment danych,
  - segment indeksu klastrowanego (PRIMARY KEY).

👉 Segment to bardziej „pojemnik logiczny”, a extenty to jego fizyczne części.

## ◆ Relacja segment ↔ extent

- Segment składa się z extentów.
- Extenty zawierają strony (*pages*).

- Strony zawierają rekordy, indeksy, itp.

### Przykład różnej organizacji systemów baz danych:

W Oracle Blok (block) to Strona (page)

#### ♦ Oracle

- **Blok (Block)** = podstawowa jednostka przechowywania danych.
- Rozmiar bloku w Oracle jest konfigurowalny (np. **2 KB, 4 KB, 8 KB, 16 KB, 32 KB**).
- W bloku są wiersze tabel, wpisy indeksów, nagłówki itd.

#### ♦ InnoDB (MySQL)

- **Strona (Page)** = podstawowa jednostka przechowywania danych.
- Rozmiar strony jest prawie zawsze **16 KB** (od MySQL 5.7 można zmienić, ale zwykle 16 KB).
- W stronie są rekordy, sloty, metadane – bardzo podobnie jak w bloku Oracle.



#### ♦ 1. Tworzymy prostą tabelę

```
CREATE TABLE uczniowie (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    imie VARCHAR(50),  
    nazwisko VARCHAR(50)  
) ENGINE=InnoDB;
```

👉 Co powstaje w .ibd?

- segment danych (dla clustered index = PK **id**),
- segment indeksu klastra.

Czyli minimum **2 segmenty**.





## ♦ 2. Dodajemy nowy indeks

```
ALTER TABLE uczniowie ADD INDEX idx_nazwisko (nazwisko);
```

👉 Powstaje **nowy segment** na ten indeks.  
Teraz w pliku `.ibd` są 3 segmenty:

- dane (clustered index),
- indeks klastra,
- dodatkowy indeks `idx_nazwisko`.



## ♦ 3. Dodajemy kolumnę typu LOB

```
ALTER TABLE uczniowie ADD COLUMN opis TEXT;
```

👉 Dla kolumny `TEXT` tworzony jest osobny segment LOB (Large Object).  
Teraz mamy 4 segmenty:

- dane,
- indeks klastra,
- dodatkowy indeks,
- segment LOB dla `opis`.



## 4. Tabela zaczyna rosnać (np. milion rekordów)

```
INSERT INTO uczniowie (imie, nazwisko, opis)  
SELECT 'Jan', 'Kowalski', REPEAT('x', 1000)  
FROM generate_series(1, 1000000);
```

👉 Segmenty się **nie zmieniają** – dalej są 4.  
Po prostu każdy segment dostaje **więcej extentów (1 MB)** i plik `.ibd` rośnie.

# Tydzień 2 Lekcja 3

## **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
);

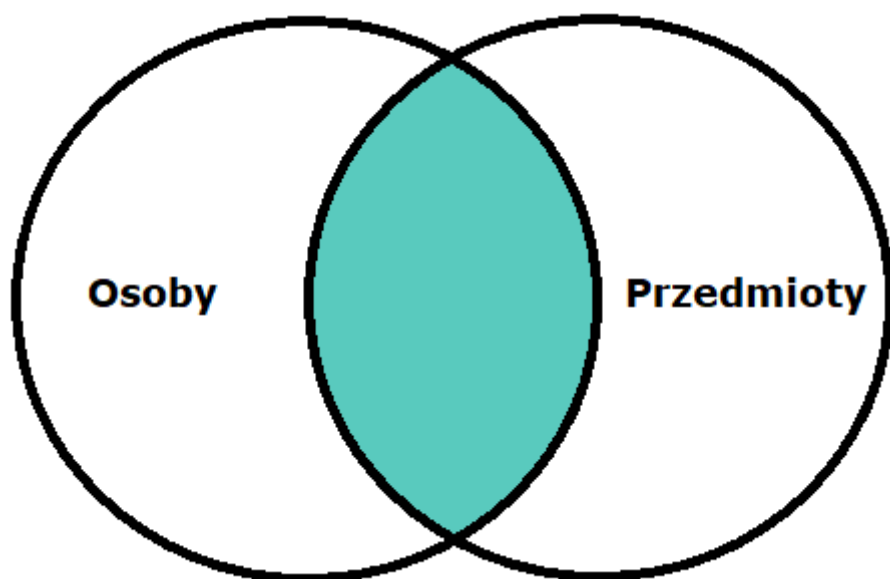
CREATE TABLE Przedmioty (
    przedmiot_id INT AUTO_INCREMENT PRIMARY KEY,
    nazwa VARCHAR(100) NOT NULL,
    osoba_id INT,
    CONSTRAINT fk_przedmiot_osoba FOREIGN KEY (osoba_id)
REFERENCES Osoby(osoba_id)
);

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

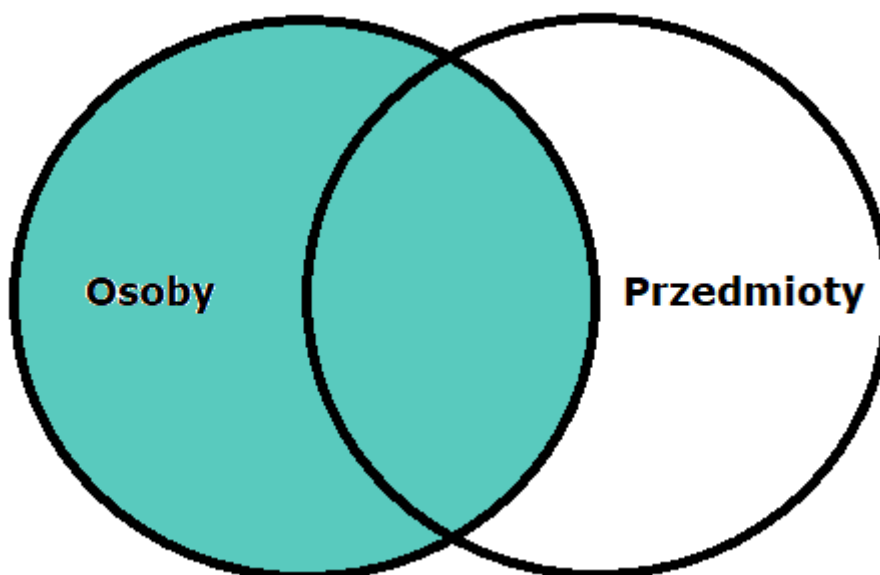


```
select Osoby.imie, Osoby.nazwisko, Przedmioty.nazwa  
from Osoby  
inner join Przedmioty on Osoby.osoba_id = Przedmioty.osoba_id;
```

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.



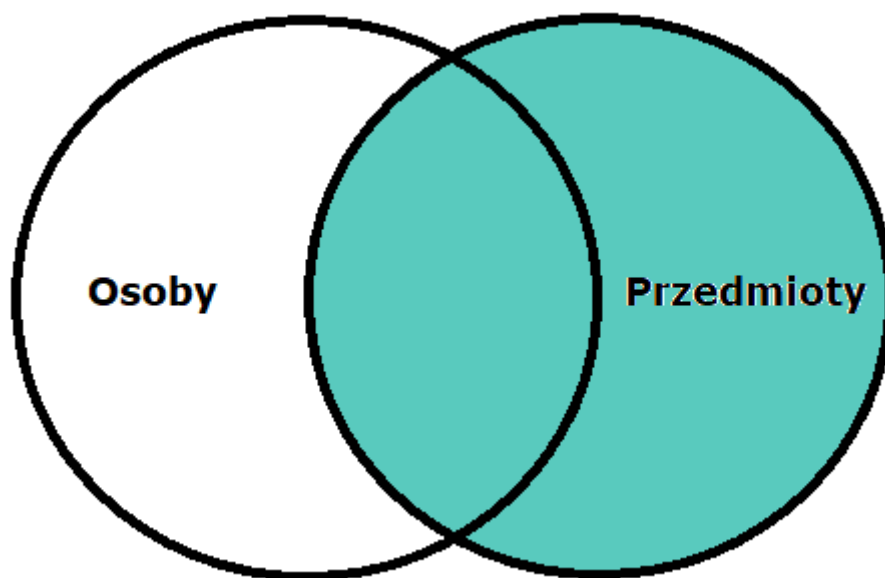
```
SELECT Osoby.imie, Osoby.nazwisko, Przedmioty.nazwa  
FROM Osoby  
LEFT JOIN Przedmioty ON Osoby.osoba_id = Przedmioty.osoba_id;
```

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:

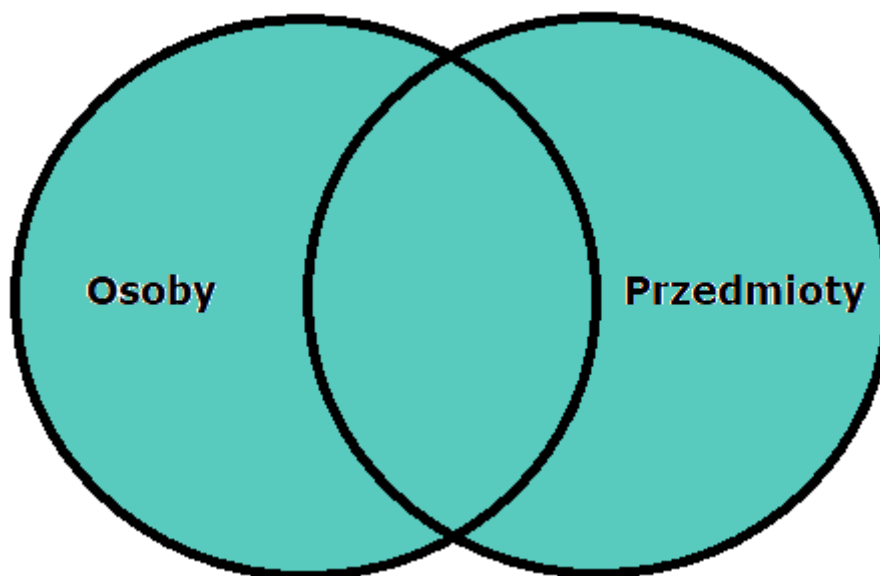
imie	nazwisko	nazwa
Jan	Kowalski	Laptop
Jan	Kowalski	Telefon
Anna	Nowak	Rower
Piotr	Zieliński	Książka
Kasia	Wiśniewska	Plecak
NULL	NULL	Kubek



♦ **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

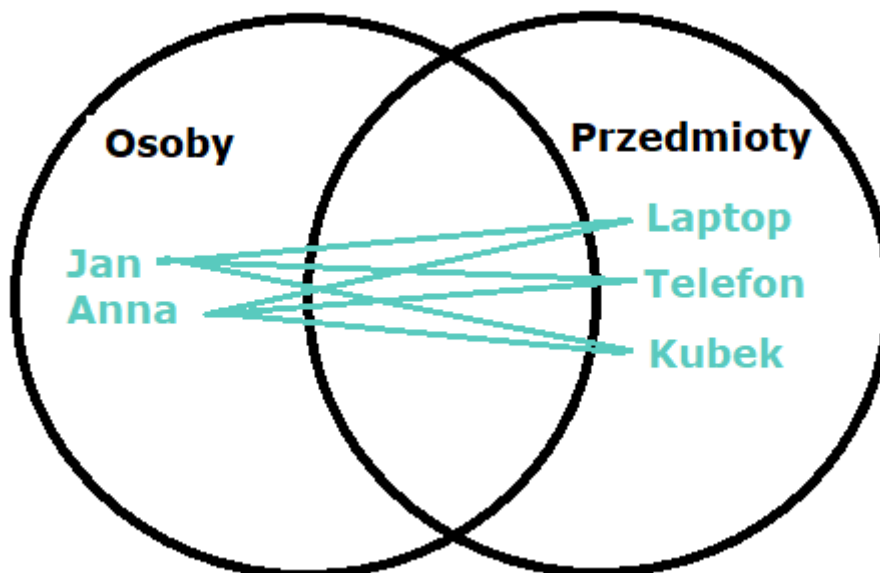
Piotr	Zieliński	Książka
Kasia	Wiśniewska	Plecak
Patryk	Nowakowski	<i>NULL</i>
<i>NULL</i>	<i>NULL</i>	Kubek

**Uwaga:**

SELECT \* FROM db FULL JOIN event

Działa, bo FULL JOIN jest bez ON, MySQL traktuje je jako CROSS JOIN

♦ **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
------	-------

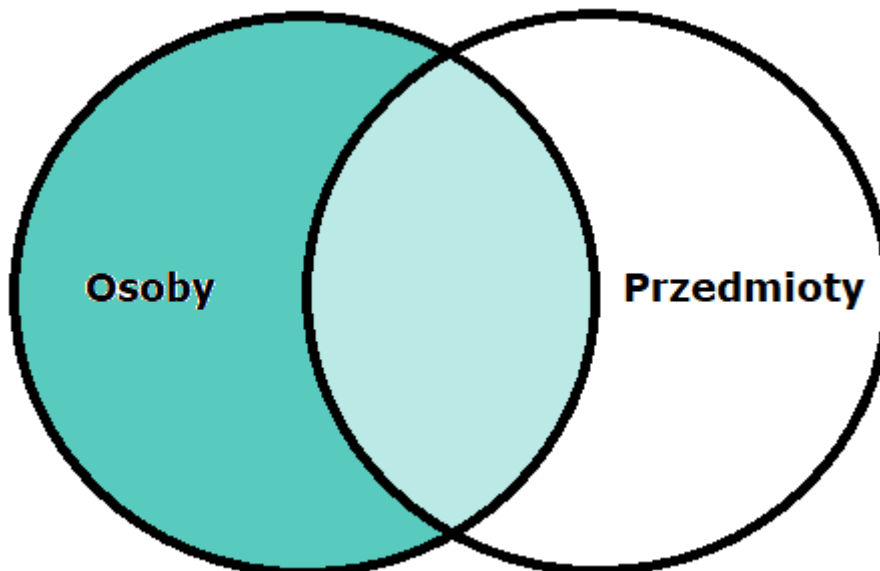
Piotr	Kubek
-------	-------

Kasia	Kubek
-------	-------





♦ **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



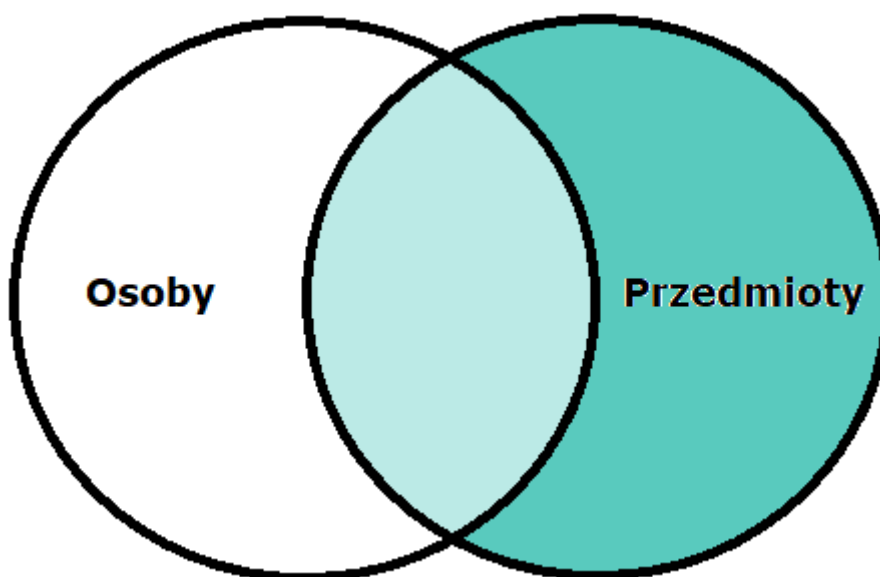
```
SELECT o.imie, o.nazwisko, p.nazwa
FROM Osoby o
LEFT JOIN Przedmioty p ON o.osoba_id = p.osoba_id
WHERE p.osoba_id IS NULL;
```

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;
```

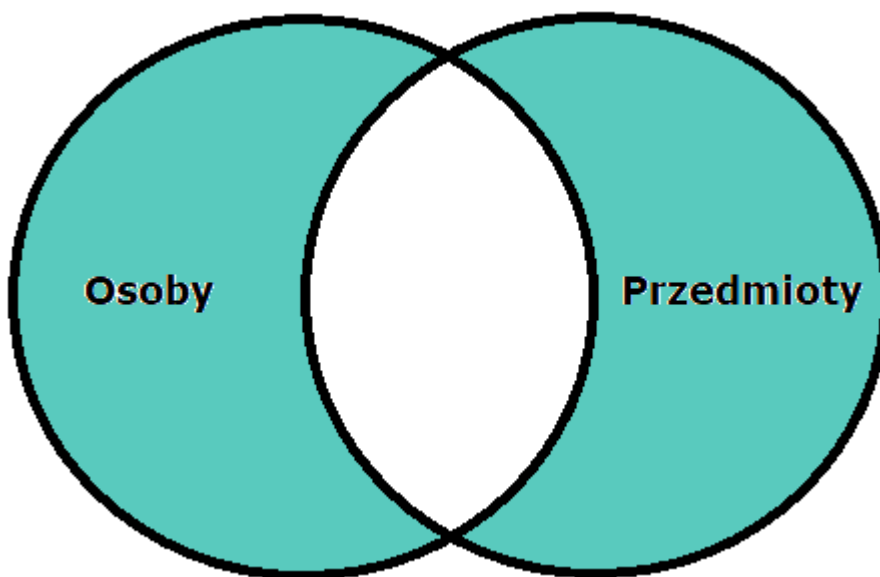
Wynik:

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

## 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 4

**Temat:** Typy danych w MySQL. Tryb ścisły (strict mode). UNSIGNED i SIGNED

**Link do dokumentacji:**

<https://dev.mysql.com/doc/refman/9.1/en/data-types.html>



**MySQL obsługuje typy danych SQL w kilku kategoriach:**

1. typy **numeryczne**,
2. typy **daty** i **godziny**,
3. typy **ciągów znaków** (znakowe i bajtowe),
4. typy **przestrzenne**
5. typ danych **JSON**



♦ Typy **numeryczne**:

### 1. Liczby całkowite (Integer Types):

- **TINYINT** – 1 bajt, zakres: **-128 do 127** (lub 0 do 255 dla UNSIGNED).
- **SMALLINT** – 2 bajty, zakres: **-32.768 do 32.767** (lub 0 do 65,535 dla UNSIGNED).
- **MEDIUMINT** – 3 bajty, zakres: **-8.388.608 do 8.388.607** (lub 0 do 16,777,215 dla UNSIGNED).

- **INT (lub INTEGER) – 4 bajty, zakres: -2.147.483.648 do 2.147.483.647** (lub 0 do 4,294,967,295 dla UNSIGNED).
- **BIGINT – 8 bajtów, zakres: -9.223.372.036.854.775.808 do 9.223.372.036.854.775.807** (lub 0 do 18,446,744,073,709,551,615 dla UNSIGNED).



#### Przykład:

CREATE TABLE produkty (

```
id INT UNSIGNED NOT NULL AUTO_INCREMENT, -- id, tylko wartości >=0
nazwa VARCHAR(50) NOT NULL , -- nazwa produktu
ilosc SMALLINT UNSIGNED NOT NULL DEFAULT 0, -- ilość sztuk, tylko >=0
cena DECIMAL(10,2) UNSIGNED NOT NULL DEFAULT 0.00, -- cena tylko >=0
ocena TINYINT UNSIGNED DEFAULT 0, -- ocena produktu 0-255
PRIMARY KEY(id)
```

);



#### ♦ Wyjaśnienie:

- INT UNSIGNED → od 0 do 4 294 967 295
- SMALLINT UNSIGNED → od 0 do 65 535
- TINYINT UNSIGNED → od 0 do 255
- DECIMAL(10,2) UNSIGNED → liczba dziesiętna z 2 miejscami po przecinku, tylko wartości nieujemne



#### Przykład:

```
CREATE TABLE example (
    id SMALLINT,
    age SMALLINT
);
```

Dodajemy wartość poza zakresem

```
INSERT INTO `example` VALUES ( -32768, 33769);
```

zwraca komunikat



Warning: #1264 Out of range value for column 'age' at row 1

**Wartość została dodana, ale o maksymalnym rozmiarze**

```
SELECT * FROM example;
```

```
+-----+-----+
| id     | age   |
+-----+-----+
| -32768 | 32767 |
+-----+-----+
```

**Jak wymusić sprawdzanie zakresu (strict mode)?**

Jeśli chcesz, aby MySQL blokował wstawianie wartości spoza zakresu, włącz tryb ścisły (strict mode) w konfiguracji serwera lub sesji:

```
SET sql_mode = 'STRICT_ALL_TABLES';
```

Teraz przy próbie wstawienia wartości 33769 do kolumny SMALLINT otrzymasz błąd:



ERROR 1264 (22003): Out of range value for column 'age' at row 1





## 2. Liczby zmiennoprzecinkowe (Floating-Point Types):

- **FLOAT** – liczba zmiennoprzecinkowa o pojedynczej precyzji, przybliżona, z opcjonalnym określeniem precyzji (np. `FLOAT(p)`).
- **DOUBLE** (lub **DOUBLE PRECISION**, **REAL**) – liczba zmiennoprzecinkowa o podwójnej precyzji, przybliżona.

### Pojedyncza precyzja:

- **Pojedyncza precyzja** oznacza, że liczba jest przechowywana w formacie zgodnym ze standardem IEEE 754, używając **32 bitów** (4 bajty). Składa się z:
  - **1 bit na znak** (+ lub -).
  - **8 bitów na wykładnik** (określa "skalę" liczby, np. czy to  $10^3$ ,  $10^{-5}$ ).
  - **23 bity na mantysę** (określa cyfry znaczące liczby).

### Podwójna precyzja:

- **Podwójna precyzja** oznacza, że liczba jest przechowywana zgodnie ze standardem IEEE 754, używając **64 bitów** (8 bajtów), w przeciwieństwie do `FLOAT`, który używa 32 bitów (pojedyncza precyzja). Składa się z:
  - **1 bit na znak** (+ lub -).
  - **11 bitów na wykładnik** (określa skalę liczby, np.  $10^5$ ,  $10^{-10}$ ).
  - **52 bity na mantysę** (określa cyfry znaczące liczby).



Cecha	FLOAT (23 bity na mantysę)	DOUBLE (52 bity na mantysę)
Liczba bitów na mantysę	23 + 1 ukryty = 24 bity	52 + 1 ukryty = 53 bity
Precyzja (cyfry dziesiętne)	~6-7 cyfr znaczących	~15-16 cyfr znaczących
Przykład liczby	<b>123.456789</b> → <b>~123.4568</b>	<b>123.456789</b> → <b>~123.456789</b>
Zastosowanie	Mniej precyzyjne obliczenia	Precyzyjne obliczenia



### 3. Liczby o stałej precyzji (Fixed-Point Types):

- **DECIMAL** (lub NUMERIC) – liczba o stałej precyzji, używana do dokładnych obliczeń (np. finansowych), z określonym miejscem na cyfry przed i po przecinku (np. DECIMAL(10,2)).



### 4. Typ bitowy (Bit Type):

- **BIT** – przechowuje wartości bitowe (od 1 do 64 bitów), używane do przechowywania sekwencji bitów. Wartości bitowe można zapisywać w formacie binarnym (np. b'1010') lub dziesiętnym.



## ♦ Typy **daty** i **godziny**,

### DATE

- **Opis:** Przechowuje datę w formacie **YYYY-MM-DD** (rok-miesiąc-dzień).
- **Zakres:** Od 1000-01-01 do 9999-12-31.

### DATETIME

- **Opis:** Przechowuje datę i godzinę w formacie **YYYY-MM-DD HH:MM:SS**.
- **Zakres:** Od 1000-01-01 00:00:00 do 9999-12-31 23:59:59.

### TIMESTAMP

- **Opis:** Przechowuje datę i godzinę w formacie **YYYY-MM-DD HH:MM:SS**, ale automatycznie konwertuje na UTC podczas zapisu i z powrotem na lokalną strefę czasową podczas odczytu. Często używany do rejestrowania czasu modyfikacji.
- **Zakres:** Od 1970-01-01 00:00:01 UTC do 2038-01-19 03:14:07 UTC.

### TIME

- **Opis:** Przechowuje czas w formacie **HH:MM:SS** (godziny:minuty:sekundy).
- **Zakres:** Od -838:59:59 do 838:59:59.

### YEAR

- **Opis:** Przechowuje rok w formacie **YYYY** (4 cyfry) lub **YY** (2 cyfry).
- **Zakres:** Od 1901 do 2155 (dla 4 cyfr) lub od 70 (1970) do 69 (2069) dla 2 cyfr.



## ♦ Typy **ciągów znaków** (znakowe i bajtowe)

### 1. Typy danych dla ciągów znaków (tekstowych)

#### CHAR(n)

- **Opis:** Stała długość ciągu znaków, gdzie n to liczba znaków (od 0 do 255). Jeśli dane są krótsze niż n, MySQL uzupełnia je spacjami.

## VARCHAR(n)

- **Opis:** Zmienna długość ciągu znaków, gdzie n to maksymalna liczba znaków (od 0 do 65,535, zależnie od kodowania i limitu wiersza).

## TINYTEXT

- **Opis:** Przechowuje tekst o maksymalnej długości 255 znaków.

## TEXT

- **Opis:** Przechowuje tekst o maksymalnej długości 65,535 znaków.

## MEDIUMTEXT

- **Opis:** Przechowuje tekst o maksymalnej długości 16,777,215 znaków.

## LONGTEXT

- **Opis:** Przechowuje tekst o maksymalnej długości 4,294,967,295 znaków.

## ENUM

- **Opis:** Przechowuje jedną wartość z predefiniowanej listy ciągów znaków (do 65,535 wartości).

Przykład:

```
CREATE TABLE shirts (
```

```
    name VARCHAR(40),
```

```
    size ENUM('x-small', 'small', 'medium', 'large', 'x-large')
```

```
);
```

```
INSERT INTO shirts (name, size) VALUES ('dress shirt','large'), ('t-shirt','medium'),
```

```
    ('polo shirt','small');
```

```
SELECT name, size FROM shirts;
```

```
+-----+-----+
```

```
| name          | size    |
```

```

+-----+
| dress shirt | large |
| t-shirt    | medium |
| polo shirt  | small  |
+-----+

```

## SET

- **Opis:** Przechowuje zbiór wartości z predefiniowanej listy (do 64 elementów).

Przykład:

```
CREATE TABLE myset (col SET('a', 'b', 'c', 'd'));
```

```
INSERT INTO myset (col) VALUES ('a,d'), ('d,a'), ('a,d,a'), ('a,d,d'), ('d,a,d');
```

```
SELECT col FROM myset;
```

```

+-----+
| col |
+-----+
| a,d |
| a,d |
| a,d |
| a,d |
| a,d |
+-----+

```



## 2. Typy danych dla ciągów bajtowych (binarnych)

### BINARY(n)

- **Opis:** Stała długość ciągu bajtów, gdzie n to liczba bajtów (od 0 do 255). Uzupełniane zerami binarnymi, jeśli dane są krótsze.

```
CREATE TABLE t (c BINARY(3));
```

```
INSERT INTO t SET c = 'a';
```

```
SELECT HEX(c), c = 'a', c = 'a\0\0', c FROM t;
```

HEX(C)	c='a'	c='a\0\0'	c
610000	0	1	0x610000

## VARBINARY(n)

- **Opis:** Zmienna długość ciągu bajtów, gdzie n to maksymalna liczba bajtów (od 0 do 65,535).

## TINYBLOB

- **Opis:** Przechowuje dane binarne o maksymalnej długości 255 bajtów.

## BLOB

- **Opis:** Przechowuje dane binarne o maksymalnej długości 65,535 bajtów.

## MEDIUMBLOB

- **Opis:** Przechowuje dane binarne o maksymalnej długości 16,777,215 bajtów.

## LOBLOB

- **Opis:** Przechowuje dane binarne o maksymalnej długości 4,294,967,295 bajtów.



## ♦ Typy przestrzenne

## GEOMETRY

- **Opis:** Typ ogólny, który może przechowywać dowolny obiekt geometryczny (np. POINT, LINESTRING, POLYGON itp.). Jest to nadrzędny typ dla wszystkich innych typów przestrzennych.

## POINT

- **Opis:** Przechowuje pojedynczy punkt w przestrzeni 2D z współrzędnymi (x, y).

## LINESTRING

- **Opis:** Przechowuje sekwencję punktów tworzących linię (ciągłą lub łamaną).

## POLYGON

- **Opis:** Przechowuje wielokąt zdefiniowany przez zamknięty zbiór punktów (obszar otoczony linią).

## MULTIPOINT

- **Opis:** Przechowuje zbiór punktów.

## MULTILINESTRING

- **Opis:** Przechowuje zbiór linii (LINESTRING).

## MULTIPOLYGON

- **Opis:** Przechowuje zbiór wielokątów (POLYGON).

## GEOMETRYCOLLECTION

- **Opis:** Przechowuje zbiór różnych obiektów geometrycznych (np. punkty, linie, wielokąty).

## ♦ Typ danych **JSON**

### Przykład użycia

Tworzenie tabeli z kolumną JSON

```
CREATE TABLE Uzytkownicy (  
  Id INT AUTO_INCREMENT PRIMARY KEY,  
  DaneUzytkownika JSON  
);
```

```
INSERT INTO Uzytkownicy (DaneUzytkownika)  
VALUES ('{'
```

```
"imie": "Jan",  
"nazwisko": "Kowalski",  
"wiek": 30,  
"adres": {  
    "miasto": "Warszawa",  
    "ulica": "Prosta 1"  
},  
"zainteresowania": ["sport", "muzyka"]  
});
```

W MySQL typ danych JSON służy do przechowywania danych w formacie JSON (JavaScript Object Notation), który jest lekki i pozwala na przechowywanie strukturalnych danych, takich jak obiekty, tablice, liczby, ciągi znaków, wartości logiczne czy null.

## Tydzień 3 Lekcja 5

### Temat: SQL - podzapytania

Podzapytania w MySQL to zapytania SQL którą są zagnieżdżone wewnątrz innego zapytania

```
-- Tworzenie tabeli uczniowie  
CREATE TABLE uczniowie (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    imie VARCHAR(50),  
    nazwisko VARCHAR(50)  
);
```

```
-- Tworzenie tabeli oceny  
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');
```

Przykłady użycia:

 **Podzapytanie skalarne w WHERE** (nieskorelowane):

- Znajdź uczniów z oceną wyższą niż średnia wszystkich ocen.

```
SELECT imie, ocena  
FROM uczniowie, oceny  
WHERE ocena > (SELECT AVG(ocena) FROM oceny);
```

 **Podzapytanie tabelaryczne w FROM:**

- Użyj podzapytania jako "wirtualnej tabeli".

```
SELECT avg_ocena.id_ucznia, avg_ocena.srednia  
FROM (  
    SELECT id_ucznia, AVG(ocena) AS srednia  
    FROM oceny  
    GROUP BY id_ucznia) AS avg_ocena
```

```
WHERE avg_ocena.srednia > 4.0;
```

### Podzapytanie skorelowane z EXISTS:

- Sprawdź, czy istnieją oceny dla ucznia.

```
SELECT imie  
FROM uczniowie u  
WHERE EXISTS (SELECT 1 FROM oceny o WHERE o.id_ucznia = u.id);
```

### Podzapytanie w SELECT:

- Dodaj kolumnę z wynikiem podzapytania.

```
SELECT imie, (SELECT COUNT(*)  
              FROM oceny  
              WHERE id_ucznia = uczniowie.id) AS liczba_ocen  
FROM uczniowie;
```

## Tydzień 3 Lekcja 6

**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 podzapytaniami skorelowanymi a nieskorelowanymi

- **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 p.imie, p.pensja, p.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 p.imie, p.pensja, p.id_dzial  
FROM pracownicy p  
JOIN (  
    SELECT id_dzial, AVG(pensja) AS srednia
```

```

FROM pracownicy
GROUP BY id_dzial
) s ON p.id_dzial = s.id_dzial
WHERE p.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'
```

## Lekcja 7

**Temat:** Funkcje tekstowe w MySQL – Praca z danymi tekstowymi: CONCAT, SUBSTRING, REPLACE, UPPER, LOWER i ich zastosowanie.

Dokumentacja

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

Funkcje tekstowe w MySQL to wbudowane funkcje, które pozwalają na manipulację ciągami znaków (stringami).

Funkcje te działają na kolumnach typu VARCHAR, CHAR, TEXT itp. i są często używane w klauzulach SELECT, WHERE czy ORDER BY. Większość z nich jest multibyte safe, co oznacza, że obsługują znaki wielobajtowe (np. w UTF-8).

```
/* Drop TABLE employees; */
```

```
CREATE TABLE employees (  
    id INT PRIMARY KEY,  
    first_name VARCHAR(50),  
    last_name VARCHAR(50),  
    email VARCHAR(100)  
);
```

```
INSERT INTO employees (id, first_name, last_name, email) VALUES  
(1, 'Jan', 'Kowalski', 'jan.kowalski@example.com'),  
(2, 'Anna', 'Nowak', 'anna.nowak@example.com'),  
(3, 'Michał', 'Wiśniewski', 'michal.wisniewski@przyklad.pl'),  
(4, 'Natalia', null, 'Natalia@przyklad.pl'),  
(5, null, 'Nowakowski', 'Nowakowski@przyklad.pl'),  
(6, 'Katarzyna', null, null);
```



### 1. CONCAT(str1, str2, ...)

- **Wyjaśnienie:** Łączy dwa lub więcej ciągów w jeden. Jeśli którykolwiek argument jest NULL, zwraca NULL. Przydatne do tworzenia pełnych nazw, adresów czy złożonych ciągów.
- **Składnia:** CONCAT(str1, str2, ...)

**Przykład zastosowania:** Łączenie imienia i nazwiska w jedną kolumnę "full\_name".

```
SELECT id, CONCAT(first_name, ' ', last_name) AS full_name  
FROM employees;
```

```
SELECT CONCAT('Ala ma kota', null, 'psa') AS wynik; – NULL
```



### 2. SUBSTRING(str, pos [, len])

- **Wyjaśnienie:** Wyodrębnia podciąg z ciągu zaczynając od pozycji pos (1 to pierwsza pozycja). Jeśli podano len, ogranicza długość. Negatywna pos liczy od końca. Przydatne do wyciągania fragmentów tekstu, np. inicjałów czy kodów.
- **Składnia:** SUBSTRING(str, pos [, len]) (lub alternatywnie SUBSTRING(str FROM pos FOR len))

**Przykład zastosowania:** Wyodrębnianie pierwszych 3 liter nazwiska.

```
SELECT id, SUBSTRING(last_name, 1, 3) AS short_last_name  
FROM employees;
```

```
SELECT SUBSTRING('Ala ma kota', 2, 1) AS wynik; – l
```



### 3. REPLACE(str, from\_str, to\_str)

- **Wyjaśnienie:** Zastępuje wszystkie wystąpienia from\_str w ciągu str na to\_str. Funkcja rozróżnia małe i duże litery jako inne znaki. Przydatne do korekty danych, np. zamiany domen e-mail.
- **Składnia:** REPLACE(str, from\_str, to\_str)

**Przykład zastosowania:** Zmiana domeny e-mail na inną (np. z "example.com" na "firma.pl").

```
SELECT id, REPLACE(email, 'example.com', 'firma.pl') AS new_email
```



FROM employees;



#### 4. UPPER(str)

- **Wyjaśnienie:** Konwertuje ciąg na wielkie litery według bieżącego kodowania znaków (domyślnie utf8mb4). Nie działa na binarnych stringach. Przydatne do normalizacji danych, np. do porównań bez rozróżniania wielkości liter.
- **Składnia:** UPPER(str)

**Przykład zastosowania:** Konwersja imienia na wielkie litery.

```
SELECT id, UPPER(first_name) AS upper_first_name  
FROM employees;
```



#### 5. LOWER(str)

- **Wyjaśnienie:** Konwertuje ciąg na małe litery według bieżącego kodowania znaków. Nie działa na binarnych stringach. Przydatne do standaryzacji tekstu, np. w wyszukiwaniach.
- **Składnia:** LOWER(str)

**Przykład zastosowania:** Konwersja nazwiska na małe litery.

```
SELECT id, LOWER(last_name) AS lower_last_name  
FROM employees;
```



#### 6. LENGTH(str)

- **Wyjaśnienie:** Zwraca długość ciągu w bajtach (multibyte znaki liczą się jako wiele bajtów). Przydatne do walidacji długości pól.
- **Składnia:** LENGTH(str)

**Przykład zastosowania:** Obliczanie długości e-maila.

```
SELECT id, LENGTH(email) AS email_length  
FROM employees;
```



**Przykład zastosowania:** Załóżmy, że baza używa UTF-8 (gdzie polskie znaki zajmują 2 bajty, a emoji mogą zajmować nawet 4 bajty).



SELECT

```
LENGTH('kot') AS bajty1,
CHAR_LENGTH('kot') AS znaki1,

LENGTH('ślimak') AS bajty2,
CHAR_LENGTH('ślimak') AS znaki2,

LENGTH('😊') AS bajty3,
CHAR_LENGTH('😊') AS znaki3;
```

Wynik:

```
bajty1 = 3    znaki1 = 3    -- zwykłe litery ASCII
bajty2 = 7    znaki2 = 6    -- "ś" zajmuje 2 bajty
bajty3 = 4    znaki3 = 1    -- emoji zajmuje 4 bajty, ale to 1 znak
```



## 7. TRIM([{BOTH | LEADING | TRAILING} [remstr] FROM] str)

- **Wyjaśnienie:** Usuwa spacje (lub inne znaki) z początku i/lub końca ciągu. Domyślnie usuwa spacje z obu stron. Przydatne do czyszczenia danych.
- **Składnia:** TRIM([remstr FROM] str) (lub z BOTH/LEADING/TRAILING)

**Przykład zastosowania:** Usuwanie spacji z imienia (zakładając, że dane mają nadmiarowe spacje).

*-- Zakładamy, że dodajemy rekord z spacjami*

```
INSERT INTO employees (id, first_name, last_name, email) VALUES (7, ' Ewa ',
'Zajac', 'ewa.zajac@example.com');
INSERT INTO employees (id, first_name, last_name, email) VALUES (8, ' ', 'Sosna
', 'Sosna@example.com');
```

```
SELECT id, TRIM(first_name) AS trimmed_first_name
FROM employees WHERE id = 4;
```



## 8. LEFT(str, len)

- **Wyjaśnienie:** Zwraca lewą część ciągu o długości len. Przydatne do wyciągania prefiksów.
- **Składnia:** LEFT(str, len)

**Przykład zastosowania:** Wyciąganie pierwszych 5 znaków e-maila.

```
SELECT id, LEFT(email, 5) AS email_prefix  
FROM employees;
```



## 9. RIGHT(str, len)

- **Wyjaśnienie:** Zwraca prawą część ciągu o długości len. Przydatne do wyciągania sufiksów, np. rozszerzeń plików.
- **Składnia:** RIGHT(str, len)

**Przykład zastosowania:** Wyciąganie ostatnich 3 znaków nazwiska.

```
SELECT id, RIGHT(last_name, 3) AS last_name_suffix FROM employees;
```



## 10. LOCATE(substr, str [, pos])

- **Wyjaśnienie:** Zwraca pozycję pierwszego wystąpienia substr w str (zaczynając od 1). Jeśli nie znaleziono, zwraca 0. Opcjonalna pos określa start wyszukiwania.
- **Składnia:** LOCATE(substr, str [, pos])

**Przykład zastosowania:** Znajdowanie pozycji "@" w e-mailu.

```
SELECT id, LOCATE('@', email) AS at_position  
FROM employees;
```



## 11. INSTR(str, substr)

- **Wyjaśnienie:** Zwraca pozycję pierwszego wystąpienia podciągu substr w ciągu str. Pozycje są numerowane od 1 (pierwszy znak w ciągu to pozycja 1). Jeśli podciąg nie zostanie znaleziony, funkcja zwraca 0. Jest to funkcja case-sensitive (uwzględnia wielkość liter). Przydatna do wyszukiwania określonego fragmentu tekstu w ciągu, np. w celu

analizy zawartości pól tekstowych.

- **Składnia:** INSTR(str, substr)

**Przykład zastosowania:** Znajdowanie pozycji znaku "@" w adresie e-mail.

```
SELECT id, INSTR(email, '@') AS at_position  
FROM employees;
```

```
SELECT LOCATE('ma', 'Ala ma kota') AS wynik;      -- 5  
SELECT LOCATE('ma', 'Ala ma kota', 6) AS wynik;   -- 0 (bo od 6 już nie ma "ma")  
SELECT LOCATE('a', 'Ala ma kota', 6) AS wynik;    -- 9 (szuka od 6 znaku)
```



Różnice między **INSTR** a **LOCATE**

- **Kolejność argumentów:**
  - **INSTR**(tekst, szukany)
  - **LOCATE**(szukany, tekst [, start])
- **Dodatkowy parametr – LOCATE** pozwala podać **start** (od której pozycji szukać). **INSTR** tego nie ma.

## Lekcja 8

**Temat:** Funkcje daty i czasu – Manipulacja datami: NOW(), DATEADD, DATEDIFF, FORMAT i ich użycie w raportach.

W MySQL funkcje daty i czasu są kluczowymi narzędziami do manipulacji danymi czasowymi przechowywanymi w kolumnach typu DATE, DATETIME, TIMESTAMP itp. Umożliwiają one pobieranie bieżącej daty i czasu, wykonywanie operacji arytmetycznych na datach, obliczanie różnic między datami oraz formatowanie

danych czasowych w czytelny sposób. Są szczególnie przydatne w raportach, gdzie dane czasowe muszą być analizowane, grupowane lub prezentowane w określonym formacie.

```
CREATE TABLE orders (  
  order_id INT PRIMARY KEY,  
  customer_name VARCHAR(100),  
  order_date DATETIME,  
  total_amount DECIMAL(10, 2)  
);
```

```
INSERT INTO orders (order_id, customer_name, order_date, total_amount) VALUES  
(1, 'Jan Kowalski', '2025-09-01 10:00:00', 150.50),  
(2, 'Anna Nowak', '2025-09-10 14:30:00', 299.99),  
(3, 'Michał Wiśniewski', '2025-09-15 09:15:00', 75.00),  
(4, 'Ewa Zając', '2025-09-20 16:45:00', 200.00);
```



## 1. NOW()

- **Wyjaśnienie:** Zwraca bieżącą datę i czas w formacie YYYY-MM-DD HH:MM:SS lub YYYYMMDDHHMMSS (w zależności od kontekstu). Używa strefy czasowej ustawionej w systemie MySQL (domyślnie strefa systemowa lub ustawiona przez @@session.time\_zone). Przydatna do rejestrowania bieżącego czasu w raportach, porównywania z datami w bazie lub oznaczania aktualnych operacji.
- **Składnia:** NOW()

**Przykład zastosowania:** Dodanie kolumny z bieżącą datą i czasem do raportu zamówień.

```
SELECT order_id, customer_name, order_date, NOW() AS "bieżący_czas"  
FROM orders;
```



## 2. DATE\_ADD(date, INTERVAL expr unit)

- **Wyjaśnienie:** Dodaje określony interwał czasowy (np. dni, godziny, miesiące) do daty. Parametr expr to liczba, a unit to jednostka czasu (np. DAY, MONTH, YEAR, HOUR, MINUTE). Przydatna do obliczania dat ważności, terminów dostaw lub przesunięć czasowych w raportach.
- **Składnia:** DATE\_ADD(date, INTERVAL expr unit)

**Przykład zastosowania:** Obliczenie daty dostawy zakładając, że dostawa następuje 3 dni po złożeniu zamówienia.

```
SELECT order_id, customer_name, order_date,  
       DATE_ADD(order_date, INTERVAL 3 DAY) AS data_dostawy  
FROM orders;
```



### 3. DATEDIFF(date1, date2)

- **Wyjaśnienie:** Zwraca liczbę dni między dwiema datami (date1 - date2). Ignoruje godziny, minuty i sekundy, uwzględnia tylko datę. Wynik jest dodatni, jeśli date1 jest późniejsza niż date2, ujemny w przeciwnym razie. Przydatna do obliczania wieku zamówienia, czasu realizacji lub opóźnień.
- **Składnia:** DATEDIFF(date1, date2)

**Przykład zastosowania:** Obliczenie, ile dni minęło od złożenia zamówienia do bieżącej daty.

```
SELECT order_id, customer_name, order_date,  
       DATEDIFF(NOW(), order_date) AS dni_od_zamowienia  
FROM orders;
```



### 4. DATE\_FORMAT(date, format)

- **Wyjaśnienie:** Formatuje datę według określonego wzorca format. Umożliwia prezentację dat w czytelnej formie, np. z nazwami miesięcy, dni tygodnia lub w niestandardowych formatach. Popularne specyfikatory formatu to: %Y (rok 4-cyfrowy), %m (miesiąc 2-cyfrowy), %d (dzień 2-cyfrowy), %H (godzina 24h), %i (minuty), %S (sekundy), %W (nazwa dnia tygodnia), %M (nazwa miesiąca). Przydatna w raportach dla użytkowników końcowych.
- **Składnia:** DATE\_FORMAT(date, format)

**Przykład zastosowania:** Formatowanie daty zamówienia na format "Dzień, DD Miesiąc YYYY, HH:MM".

– **Ustaw zmienną sesyjną języka** przed zapytaniem:

```
SET lc_time_names = 'pl_PL';  
SELECT order_id, customer_name,  
       DATE_FORMAT(order_date, '%W, %d %M %Y, %H:%i') AS sformatowana_data  
FROM orders;
```



Specyfikator	Opis	Przykład (dla 2025-09-01 10:00:00)
%a	Skrócona nazwa dnia tygodnia (Sun-Sat)	Mon
%b	Skrócona nazwa miesiąca (Jan-Dec)	Sep
%c	Miesiąc w formie numerycznej (0-12)	9
%D	Dzień miesiąca z angielskim sufiksem (1st, 2nd, 3rd, ...)	1st
%d	Dzień miesiąca, 2 cyfry (00-31)	01
%e	Dzień miesiąca, bez wiodącego zera (0-31)	1
%f	Mikrosekundy, 6 cyfr (000000-999999)	000000
%H	Godzina w formacie 24h, 2 cyfry (00-23)	10
%h	Godzina w formacie 12h, 2 cyfry (01-12)	10
%I	Godzina w formacie 12h, 2 cyfry (01-12, identyczne jak %h)	10
%i	Minuty, 2 cyfry (00-59)	00
%j	Dzień roku (001-366)	244
%k	Godzina w formacie 24h, bez wiodącego zera (0-23)	10
%l	Godzina w formacie 12h, bez wiodącego zera (1-12)	10
%M	Pełna nazwa miesiąca (January-December)	September
%m	Miesiąc, 2 cyfry (01-12)	09
%p	AM/PM dla formatu 12-godzinnego	AM
%r	Pełny czas w formacie 12h (hh:mm:ss AM/PM)	10:00:00 AM
%S	Sekundy, 2 cyfry (00-59)	00

%S	Sekundy, 2 cyfry (00-59, identyczne jak %S)	00
%T	Pełny czas w formacie 24h (hh:mm:ss)	10:00:00
%U	Numer tygodnia w roku (00-53, niedziela jako pierwszy dzień)	35
%u	Numer tygodnia w roku (00-53, poniedziałek jako pierwszy dzień)	36
%V	Numer tygodnia w roku (01-53, niedziela jako pierwszy dzień, używane z %X)	35
%v	Numer tygodnia w roku (01-53, poniedziałek jako pierwszy dzień, używane z %x)	36
%W	Pełna nazwa dnia tygodnia (Sunday-Saturday)	Monday
%w	Dzień tygodnia (0-6, 0=niedziela, 6=sobota)	1
%X	Rok dla numeru tygodnia, 4 cyfry (używane z %V)	2025
%x	Rok dla numeru tygodnia, 4 cyfry (używane z %v)	2025
%Y	Rok, 4 cyfry	2025
%y	Rok, 2 cyfry	25
%%	Literalny znak %	%