

PRZEDMIOT: Systemy baz danych

KLASA: 5i gr. 2

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). 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/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.



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: Pesel

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
```



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**

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] |
+=====+
```


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.

Lekcja 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  
);
```

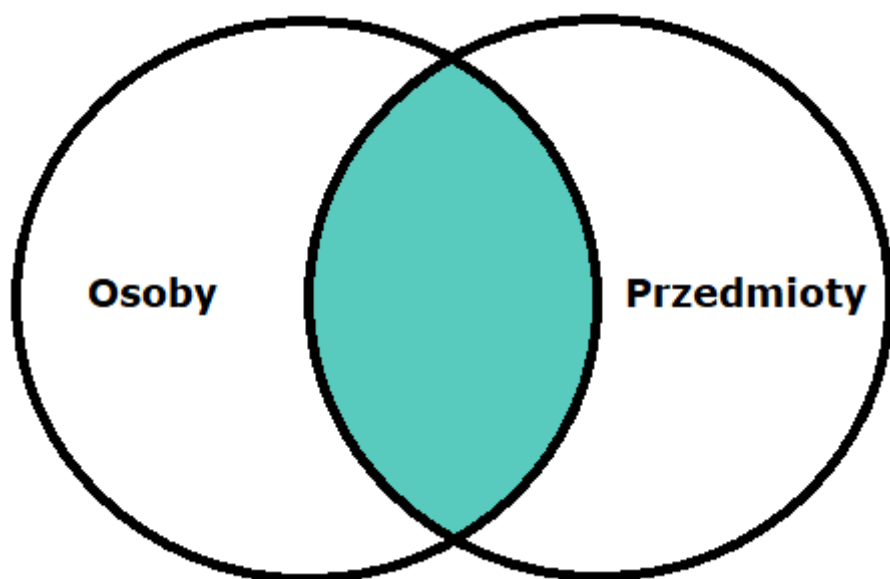
```
CREATE TABLE Przedmioty (  
    przedmiot_id INT AUTO_INCREMENT PRIMARY KEY,  
    nazwa VARCHAR(100) NOT NULL,  
    osoba_id INT,  
    CONSTRAINT fk_przedmiot_osoba FOREIGN KEY (osoba_id)  
REFERENCES Osoby(osoba_id)  
);
```

```
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

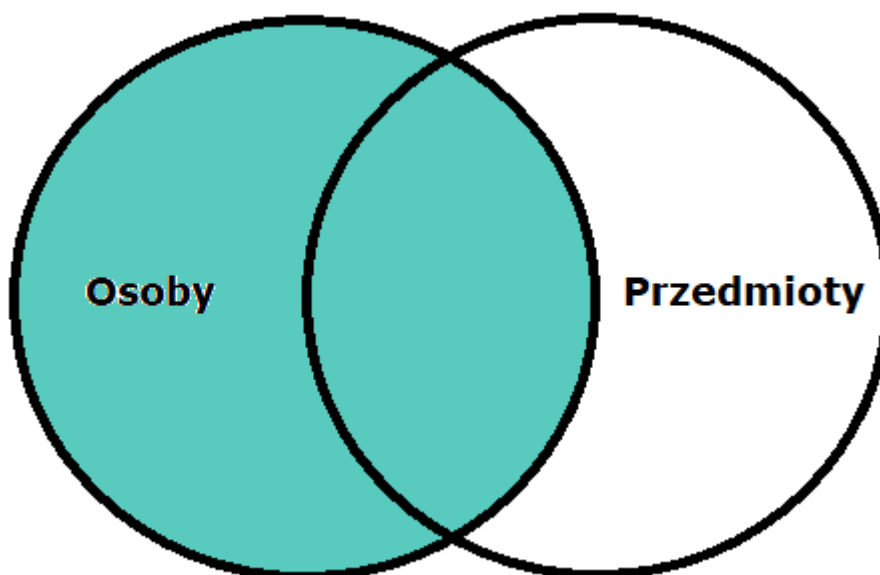


```
select Osoby.imie, Osoby.nazwisko, Przedmioty.nazwa  
from Osoby  
inner join Przedmioty on Osoby.osoba_id = Przedmioty.osoba_id;
```

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.



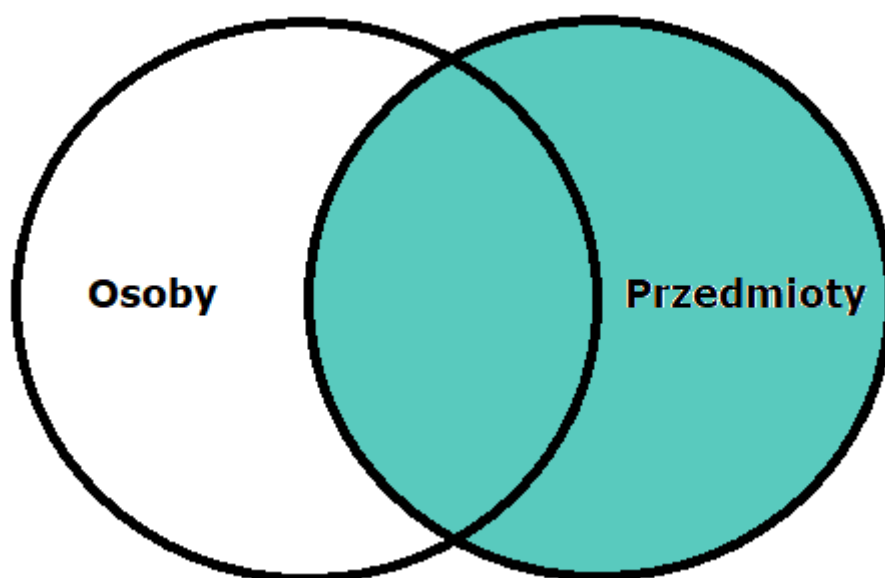
```
SELECT Osoby.imie, Osoby.nazwisko, Przedmioty.nazwa  
FROM Osoby  
LEFT JOIN Przedmioty ON Osoby.osoba_id = Przedmioty.osoba_id;
```

Wynik:

imie	nazwisko	nazwa
Jan	Kowalski	Laptop
Jan	Kowalski	Telefon
Anna	Nowak	Rower
Piotr	Zieliński	Książka
Kasia	Wiśniewska	Plecak
Patryk	Nowakowski	<i>NULL</i>



♦ **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:

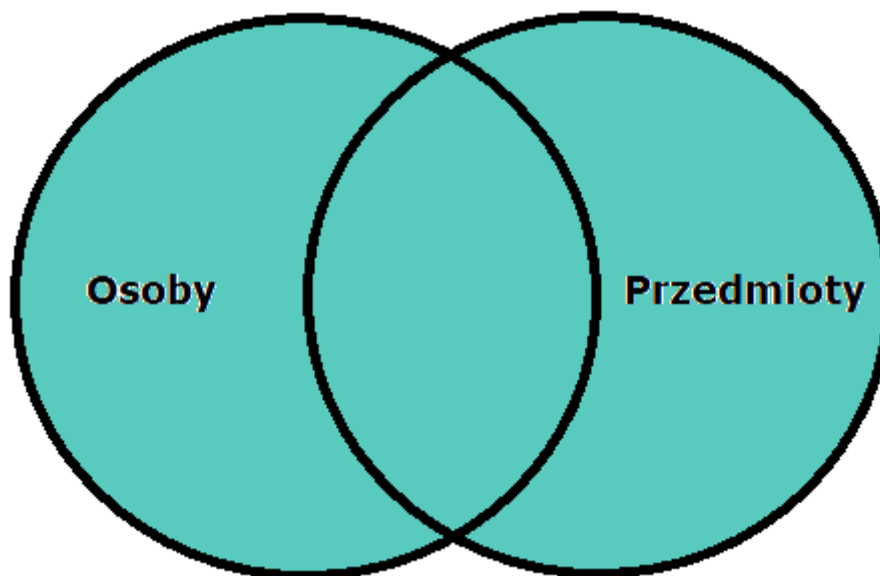
imie	nazwisko	nazwa
Jan	Kowalski	Laptop
Jan	Kowalski	Telefon
Anna	Nowak	Rower
Piotr	Zieliński	Książka
Kasia	Wiśniewska	Plecak
NULL	NULL	Kubek



♦ **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

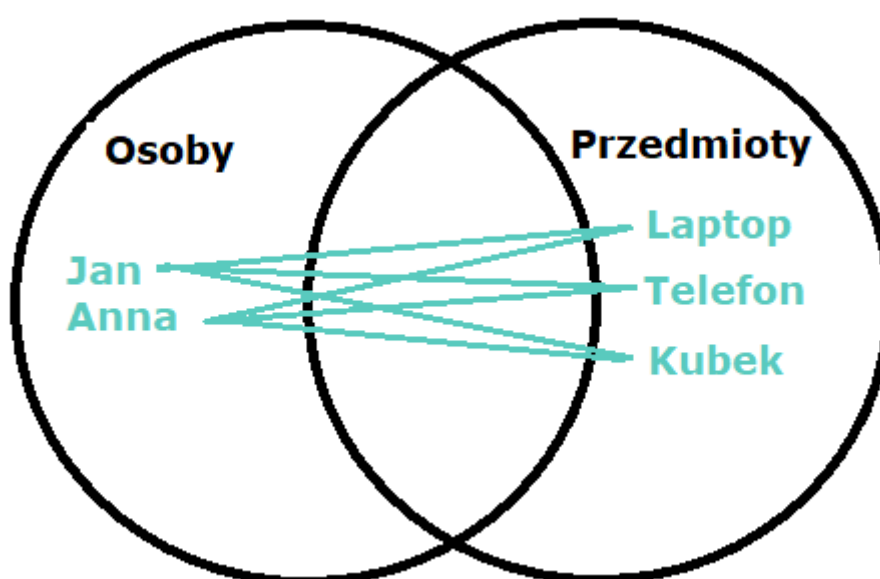
```
SELECT o.imie, o.nazwisko, p.nazwa  
FROM Osoby o  
RIGHT JOIN Przedmioty p ON o.osoba_id = p.osoba_id;
```

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:

imie	nazwa
Jan	Laptop
Anna	Laptop
Piotr	Laptop
Kasia	Laptop
Patryk	Laptop

Jan	Telefon
-----	---------

Anna	Telefon
------	---------

Piotr	Telefon
-------	---------

Kasia	Telefon
-------	---------

Patryk	Telefon
--------	---------

Jan	Rower
-----	-------

Anna	Rower
------	-------

Piotr	Rower
-------	-------

Kasia	Rower
-------	-------

Patryk	Rower
--------	-------

Jan	Książka
-----	---------

Anna	Książka
------	---------

Piotr	Książka
-------	---------

Kasia	Książka
-------	---------

Patryk	Książka
--------	---------

Jan	Plecak
-----	--------

Anna	Plecak
------	--------

Piotr	Plecak
-------	--------

Kasia	Plecak
-------	--------

Patryk	Plecak
--------	--------

Jan	Kubek
-----	-------

Anna	Kubek
------	-------

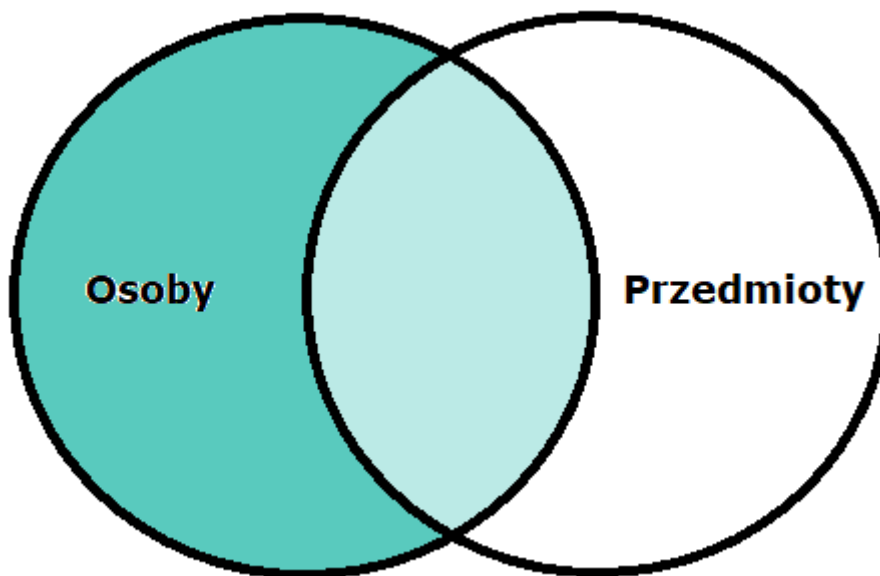
Piotr	Kubek
-------	-------

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

Patryk	Kubek
--------	-------



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



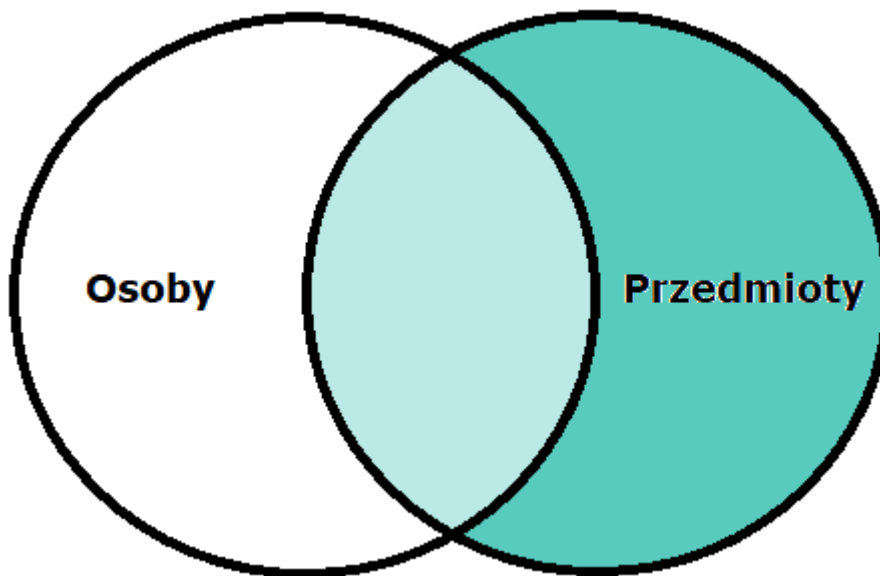
```
SELECT o.imie, o.nazwisko, p.nazwa  
FROM Osoby o  
LEFT JOIN Przedmioty p ON o.osoba_id = p.osoba_id  
WHERE p.osoba_id IS NULL;
```

Wynik:

imie	nazwisko	nazwa
Patryk	Nowakowski	NULL



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



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

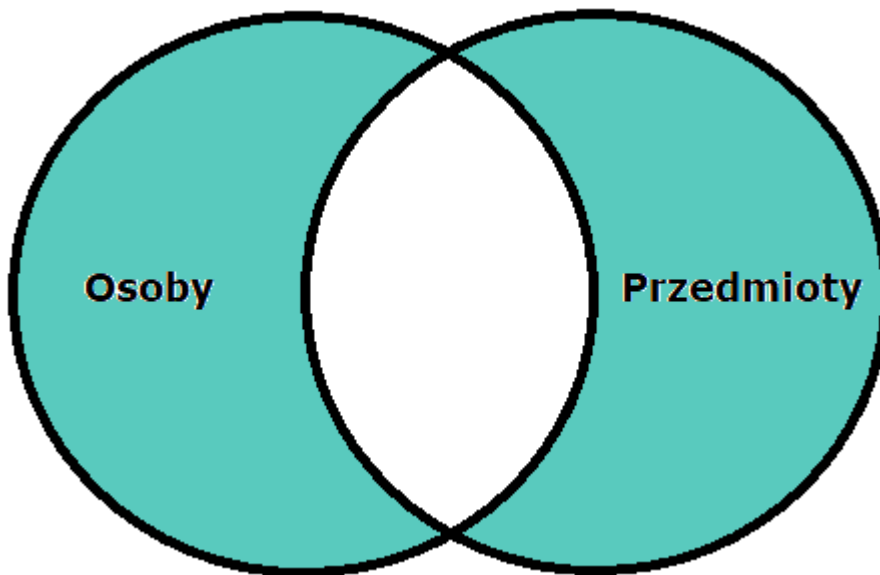
Wynik:

imie	nazwisko	nazwa
NULL	NULL	Kubek



♦ **FULL OUTER JOIN** excluding **INNER JOIN** (**LEFT JOIN** **wykluczający wiersze dopasowane**, **UNION**, **RIGHT JOIN** **wykluczający wiersze dopasowane**) - czyli wszystkie rekordy z prawej i lewej tabeli połączone. Następnie odrzucamy te wiersze, które mają dopasowanie w obu tabelach.

W MySQL nie ma instrukcji FULL OUTER JOIN. Dla MySQL należy zastosować **UNION**. Czyli left join z wartościami nie mających dopasowania oraz right join z wartościami nie mających dopasowania łączymy z **UNION**.



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

UNION

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

Wynik:

imie	nazwisko	nazwa
Patryk	Nowakowski	<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;
```

Lekcja 5

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	123.456789 → ~123.4568	123.456789 → ~123.456789
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).

```
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).

```
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.

Lekcja 6

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  
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;
```

Lekcja 7

Temat: Obsługa wyjątków

W MySQL **obsługa wyjątków jest zdefiniowana w procedurach składowanych lub funkcjach za pomocą konstrukcji DECLARE ... HANDLER.** Mechanizm ten pozwala określić, co zrobić, gdy pojawi się określony warunek (np. błąd SQL, ostrzeżenie lub brak danych).

Jak sprawdzić, czy procedura istnieje?

```
SHOW PROCEDURE STATUS WHERE Db = DATABASE();
```

Składnia podstawowa

DECLARE {CONTINUE | EXIT | UNDO} HANDLER FOR {warunek} {akcja};

- **CONTINUE:** Po obsłudze wyjątku wykonanie procedury jest kontynuowane.
- **EXIT:** Po obsłudze wyjątku procedura natychmiast się kończy.
- **UNDO:** Nieobsługiwane w MySQL (zarezerwowane na przyszłość).
- **warunek:** Rodzaj błędu lub sytuacji, którą chcemy przechwycić, np.:
 - **SQLSTATE** 'kod_błędu': Kod błędu SQLSTATE (np. '23000' dla naruszenia ograniczenia).
 - **condition_name:** Nazwa warunku zdefiniowana wcześniej przez DECLARE CONDITION.
 - **SQLException:** Wszystkie błędy SQL.
 - **SQLWARNING:** Wszystkie ostrzeżenia SQL.
 - **NOT FOUND:** Sytuacja, gdy zapytanie nie zwraca wyników (np. brak wierszy w SELECT).
- **akcja:** Kod SQL do wykonania, gdy warunek zostanie spełniony (np. ustawienie zmiennej, logowanie błędu, wykonanie alternatywnego zapytania).

Przykłady obsługi wyjątków

Przykład 1: Obsługa naruszenia ograniczenia UNIQUE (EXIT HANDLER dla SQLException)

Cel: Napisz procedurę AddEmployee, która dodaje nowego pracownika. Jeśli e-mail już istnieje (naruszenie ograniczenia UNIQUE), procedura przechwytuje błąd i zwraca komunikat o błędzie zamiast powodować przerwanie programu.

DELIMITER //

```
CREATE PROCEDURE AddEmployee(
  IN p_id INT,
  IN p_first_name VARCHAR(50),
  IN p_last_name VARCHAR(50),
  IN p_email VARCHAR(100),
  OUT p_message VARCHAR(255)
)
BEGIN
  -- Deklaracja zmiennej na komunikat
  SET p_message = 'Pracownik dodany pomyślnie';

  -- Deklaracja handlera dla SQLException
  DECLARE EXIT HANDLER FOR SQLException
  BEGIN
    SET p_message = 'Błąd: Nie można dodać pracownika, e-mail już istnieje!';
  END;
```



```

-- Próba wstawienia rekordu
INSERT INTO employees (id, first_name, last_name, email)
VALUES (p_id, p_first_name, p_last_name, p_email);

END //

DELIMITER ;

-- Test procedury
CALL AddEmployee(7, 'Ewa', 'Zając', 'jan.kowalski@example.com', @message);
SELECT @message AS message;

```

W MySQL wszystkie deklaracje (DECLARE) muszą być zgrupowane na początku bloku, przed jakimikolwiek operacjami typu SET, INSERT czy SELECT.

Przykład 2: Obsługa braku rekordu (CONTINUE HANDLER dla NOT FOUND)

Cel: Napisz procedurę GetEmployeeEmail, która pobiera e-mail pracownika na podstawie ID. Jeśli pracownik nie istnieje, handler NOT FOUND ustawi domyślny komunikat zamiast zwracać NULL.

```

DELIMITER //

CREATE PROCEDURE GetEmployeeEmail(
    IN p_id INT,
    OUT p_email VARCHAR(100)
)
BEGIN
    -- Deklaracja handlera dla NOT FOUND
    DECLARE CONTINUE HANDLER FOR NOT FOUND
    BEGIN
        SET p_email = 'Nie znaleziono pracownika';
    END;

    -- Próba pobrania e-maila
    SELECT email INTO p_email
    FROM employees
    WHERE id = p_id;
END //

DELIMITER ;

-- Test procedury
CALL GetEmployeeEmail(5, @email);
SELECT @email AS email;

```

Przykład 3: Obsługa konkretnego błędu SQLSTATE (EXIT HANDLER dla SQLSTATE)

Cel: Napisz procedurę UpdateEmployeeEmail, która aktualizuje e-mail pracownika.

Jeśli e-mail już istnieje w innym rekordzie (naruszenie UNIQUE), handler przechwytyje błąd SQLSTATE '23000' i loguje go do tabeli error_log.

Najpierw utwórz tabelę na logi błędów:

```
CREATE TABLE error_log (  
  id INT AUTO_INCREMENT PRIMARY KEY,  
  error_message VARCHAR(255),  
  error_time DATETIME  
);  
  
DELIMITER //  
  
CREATE PROCEDURE UpdateEmployeeEmail(  
  IN p_id INT,  
  IN p_new_email VARCHAR(100),  
  OUT p_message VARCHAR(255)  
)  
BEGIN  
  -- Deklaracja handlera dla SQLSTATE '23000' (naruszenie UNIQUE)  
  DECLARE EXIT HANDLER FOR SQLSTATE '23000'  
  BEGIN  
    SET p_message = 'Błąd: E-mail już istnieje w bazie!';  
    INSERT INTO error_log (error_message, error_time)  
    VALUES (p_message, NOW());  
  END;  
  
  -- Domyślny komunikat  
  SET p_message = 'E-mail zaktualizowany pomyślnie';  
  
  -- Próba aktualizacji  
  UPDATE employees  
  SET email = p_new_email  
  WHERE id = p_id;  
  
END //  
  
DELIMITER ;  
  
-- Test procedury  
CALL UpdateEmployeeEmail(2, 'jan.kowalski@example.com', @message);  
SELECT @message AS message;  
SELECT * FROM error_log
```