

PRZEDMIOT: Systemy baz danych

KLASA: 3A gr. 2

Tydzień 1 Lekcja 1

Temat: Definicja Baz Danych. Powtórzenie terminów tabele, rekordy, pola. Relację między tabelami: 1:1, 1:N, N:M.

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: 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 |

Tydzień 2 Lekcja 2

Temat: Polecenie Order By. Nadawanie, odbieranie uprawnień (GRANT, REVOKE). Pojęcie CRUD

♦ 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>;
```

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 3 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 rosnąć (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 4 Lekcja 4

Temat: SQL JOINS, CONSTRAINT. Zastosowanie wyświetlania liczb porządkowych dla wszystkich wierszy `ROW_NUMBER()`


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

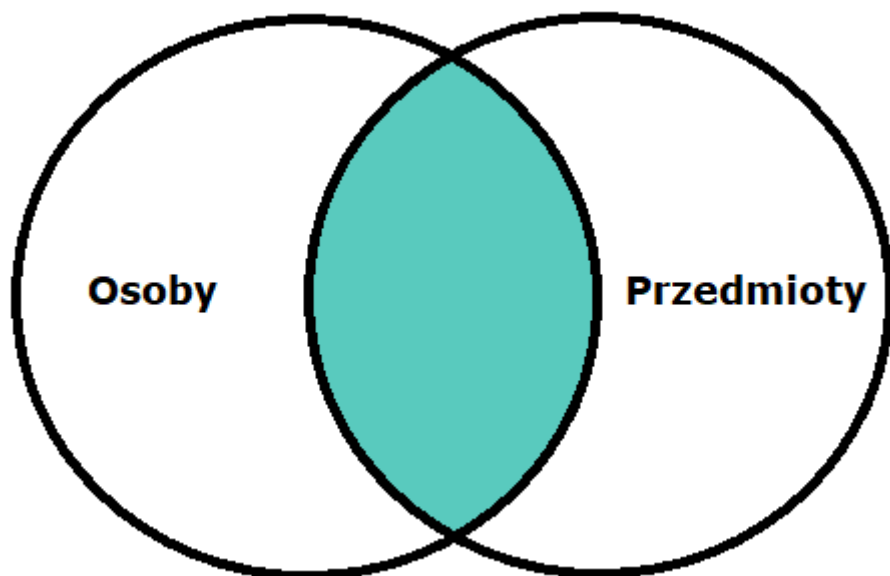
```
CREATE TABLE Osoby (  
    osoba_id INT AUTO_INCREMENT PRIMARY KEY,  
    imie VARCHAR(50) NOT NULL,  
    nazwisko VARCHAR(50) NOT NULL  
);
```

```
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);
```

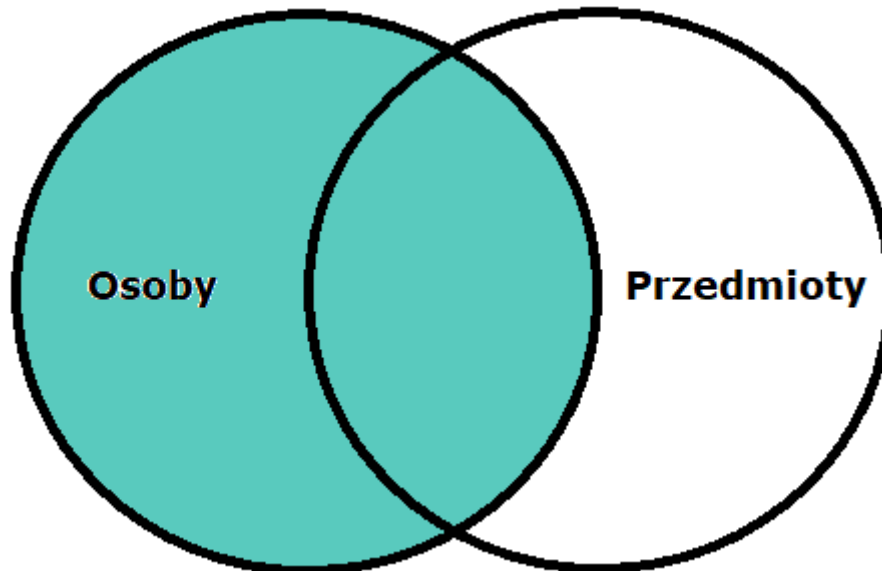
- ◆ **INNER JOIN** - czyli wszystkie wspólne rekordy, bez NULL



```
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 | 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

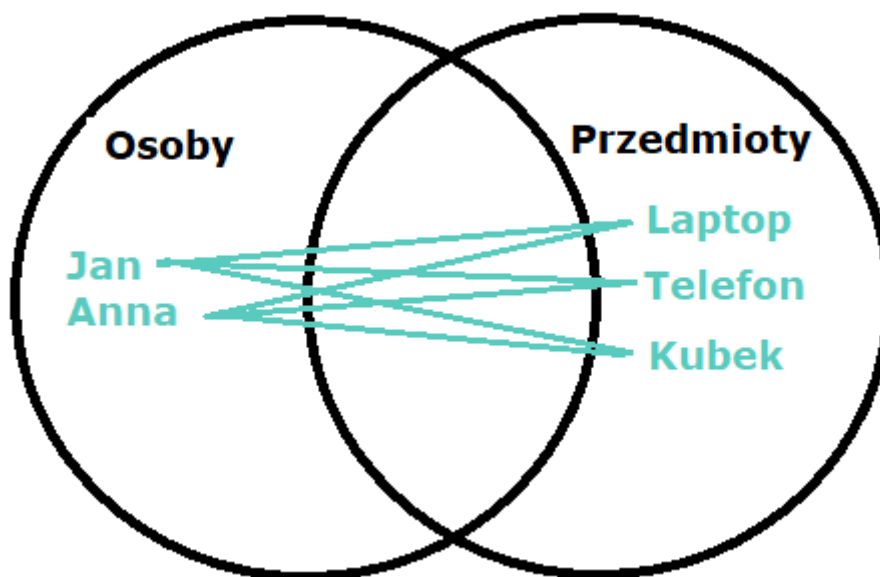
```
SELECT o.imie, o.nazwisko, p.nazwa  
FROM Osoby o  
RIGHT JOIN Przedmioty p ON o.osoba_id = p.osoba_id;
```

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:

| 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;
```