

# Lekcja

## Temat: Modele baz danych

Na potrzeby baz danych zostały zdefiniowane klasyczne techniki organizowania informacji, zwane modelami baz danych.



**Model danych to abstrakcyjny opis sposobu przedstawiania i wykorzystania danych. Definiuje on logiczną reprezentację danych oraz relacje między nimi.**

Na model danych składają się:

- struktura — opis sposobu przedstawiania obiektów (encji) modelowanego wycinka świata oraz ich związków,
- ograniczenia — reguły kontrolujące spójność i poprawność danych,
- operacje — zbiór działań, które umożliwiają dostęp do struktur.

### Głównymi modelami baz danych są:



#### 1. Tradycyjne (przedrelacyjne) modele

- a) **model hierarchiczny**,
- b) **model sieciowy**,



#### 2. **Model relacyjny** (najpopularniejszy od lat 80.)



#### 3. **Modele post-relacyjne / NoSQL**

##### a) **Model dokumentowy**

- Dane przechowywane jako dokumenty (JSON, BSON, XML)
- Struktura hierarchiczna wewnątrz dokumentu
- **Przykład:** MongoDB, Couchbase
- **Użycie:** katalogi produktów, profile użytkowników

##### b) **Model klucz-wartość**

- Proste pary: klucz → wartość (dowolne dane)
- Bardzo szybki dostęp po kluczu
- **Przykład:** Redis, Amazon DynamoDB
- **Użycie:** cache, sesje, konfiguracje

##### c) **Model szerokokolumnowy (kolumnowy)**

- Dane przechowywane w kolumnach (nie w wierszach)

- Optymalny dla agregacji i analiz
- **Przykład:** Cassandra, HBase, Google Bigtable
- **Użycie:** analityka, big data, systemy czasowych szeregów

#### d) Model grafowy

- Dane jako węzły, krawędzie i właściwości
- Idealny dla powiązań i relacji
- **Przykład:** Neo4j, Amazon Neptune
- **Użycie:** sieci społecznościowe, rekomendacje, wykrywanie oszustw



### 4. Nowoczesne / hybrydowe modele

#### a) Modele wielomodelowe

- Łączą różne modele w jednym systemie
- **Przykład:** PostgreSQL (relacyjny + JSON + graf), ArangoDB (dokumentowy + grafowy)

#### b) Bazy time-series

- Zoptymalizowane pod dane czasowe (znacznik czasu jako główny wymiar)
- **Przykład:** InfluxDB, TimescaleDB

#### c) Bazy przestrzenne/geograficzne

- Obsługa danych geograficznych i przestrzennych
- **Przykład:** PostGIS (rozszerzenie PostgreSQL)

#### d) Bazy pamięciowe (in-memory)

- Cała baza w pamięci RAM
- **Przykład:** Redis, MemSQL, SAP HANA



### 5. Model obiektowy i obiektowo-relacyjny



#### Model hierarchiczny

To jeden z najstarszych modeli baz danych (powstał w latach 60. XX wieku), w którym dane są organizowane w strukturę **drzewa** (hierarchii). Każde "drzewo" składa się z **węzłów**:

- **Jeden węzeł główny** (korzeń) – nie ma rodzica.
- **Węzły potomne** – każdy ma dokładnie **jednego rodzica**, ale może mieć wiele dzieci.
- Relacje są typu **jeden-do-wielu (1:N)**.
- Dostęp do danych często odbywa się poprzez ścieżki od korzenia do konkretnego rekordu.



## Przykład modelu hierarchicznego: Struktura Technikum

```
Technikum (korzeń)
|
|— Kierunek: Technikum Informatyczne
|   |
|   |— Przedmiot: Matematyka
|   |— Przedmiot: Język polski
|   |
|   |— Uczeń: Nowak Tadeusz
|
|— Kierunek: Technikum Programistyczne
|   |
|   |— Przedmiot: Matematyka
|   |— Przedmiot: Język polski
|   |
|   |— Uczeń: Kowalski Jan
```



## Model sieciowy

Model sieciowy powstał jako rozwinięcie modelu hierarchicznego, aby rozwiązać jego główne ograniczenia. Został sformalizowany przez grupę **CODASYL** (Conference on Data Systems Languages) w latach 60-70.



### Kluczowe cechy:

1. **Struktura grafu** – dane organizowane są jako zbiór rekordów połączonych w dowolną strukturę grafową (nie tylko drzewo)
2. **Jeden rekord może mieć wielu rodziców** – w przeciwieństwie do hierarchicznego (tylko jeden rodzic)
3. **Relacje przechowywane jako wskaźniki fizyczne** między rekordami
4. **Dwa podstawowe elementy:**
  - a. **Typ rekordu** – odpowiednik tabeli w modelu relacyjnym
  - b. **Zbiór (set)** – nazwana relacja łącząca typy rekordów
5. **Obsługa relacji wiele-do-wielu (M:N)** – główna przewaga nad modelem hierarchicznym



### Przykład modelu sieciowego: System szkolny





## Podstawowe pojęcia:

1. **Relacja (tabela)** – zbiór krotek o takiej samej strukturze
2. **Krotka (wiersz, rekord)** – pojedynczy wpis w tabeli
3. **Atrybut (kolumna, pole)** – nazwane pole z określonym typem danych
4. **Klucz główny (Primary Key)** – unikalny identyfikator wiersza
5. **Klucz obcy (Foreign Key)** – odwołanie do klucza głównego innej tabeli
6. **Domeny** – zbiór dopuszczalnych wartości dla atrybutu



## Przykład modelu relacyjnego: System szkolny

### 1. Tabele i ich struktura:

Tabela UCZNIOWIE:

ID_ucznia	Imię	Nazwisko	ID_klasy
1	Jan	Kowalski	101
2	Anna	Nowak	101
3	Tomasz	Wiśniewski	102

PK: ID\_ucznia

FK: ID\_klasy → KLASY(ID\_klasy)

Tabela KLASY:

ID_klasy	Nazwa_klasy	Rok_szkolny
101	4TI	2024
102	3TP	2024
103	2TE	2024

PK: ID\_klasy

Tabela PRZEDMIOTY:

ID_przedmiotu	Nazwa_przedmiotu
1	Matematyka
2	Język polski
3	Bazy danych

PK: ID\_przedmiotu

Tabela OCENY:

ID_oceny	ID_ucznia	ID_przedmiotu	Ocena	Data
1	1	1	5	2024-01-15
2	1	2	4	2024-01-16
3	2	1	3	2024-01-15
4	3	3	5	2024-01-17

PK: ID\_oceny

FK: ID\_ucznia → UCZNIOWIE(ID\_ucznia)

FK: ID\_przedmiotu → PRZEDMIOTY(ID\_przedmiotu)

Tabela UCZNIOWIE\_PRZEDMIOTY (relacja wiele-do-wielu):

ID_ucznia	ID_przedmiotu
1	1
1	2
2	1
3	3

PK: (ID\_ucznia, ID\_przedmiotu) - klucz złożony

## 2. Relacje między tabelami (diagram ER):

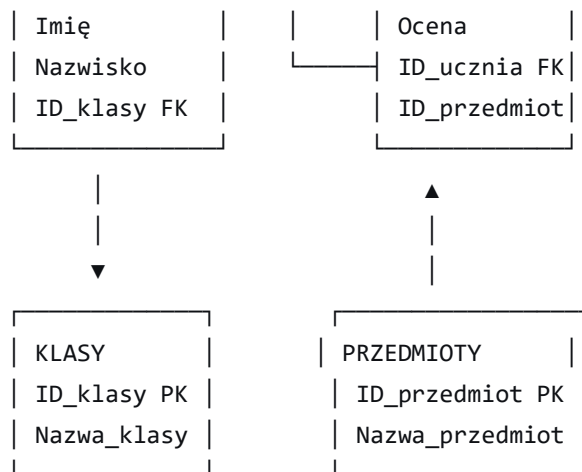
UCZNIOWIE

ID_ucznia PK
--------------

OCENY

ID_oceny PK
-------------





### Rodzaje relacji:

- **1:N** (jeden do wielu) – jedna klasa ma wielu uczniów
- **M:N** (wiele do wielu) – uczeń ma wiele przedmiotów, przedmiot ma wielu uczniów
- **1:1** (jeden do jednego) – rzadziej stosowane

```

CREATE TABLE UCZNIOWIE (
  ID_ucznia INT PRIMARY KEY,
  Imię VARCHAR(50),
  Nazwisko VARCHAR(50),
  ID_klasy INT,
  FOREIGN KEY (ID_klasy) REFERENCES KLASY(ID_klasy)
);
  
```

```

CREATE TABLE OCENY (
  ID_oceny INT PRIMARY KEY,
  ID_ucznia INT,
  ID_przedmiotu INT,
  Ocena INT CHECK (Ocena BETWEEN 1 AND 6),
  Data DATE,
  FOREIGN KEY (ID_ucznia) REFERENCES UCZNIOWIE(ID_ucznia),
  FOREIGN KEY (ID_przedmiotu) REFERENCES PRZEDMIOTY(ID_przedmiotu)
);
  
```





## Model dokumentowy

Dane przechowywane jako **dokumenty** w formatach JSON, BSON, XML. Każdy dokument jest samodzielną jednostką zawierającą wszystkie dane o danym obiekcie.

### Kluczowe cechy:

- **Samo opisujące dokumenty** – struktura zawarta w dokumencie
- **Dynamiczny schemat** – różne dokumenty mogą mieć różne pola
- **Zagnieżdżanie** – możliwość przechowywania obiektów w obiektach
- **Indeksowanie** – po dowolnych polach dokumentu



## Przykład modelu dokumentowego: System blogowy (MongoDB)

```
// Kolekcja "posts"
{
  "_id": "507f1f77bcf86cd799439011",
  "title": "Wprowadzenie do NoSQL",
  "author": {
    "name": "Jan Kowalski",
    "email": "jan@example.com",
    "bio": "Ekspert baz danych"
  },
  "tags": ["NoSQL", "MongoDB", "Bazy danych"],
  "content": "NoSQL to nowoczesne podejście...",
  "comments": [
    {
      "user": "Anna Nowak",
      "text": "Świetny artykuł!",
      "date": "2024-01-15",
      "likes": 5
    },
    {
      "user": "Tomasz Wiśniewski",
      "text": "Brakuje przykładów",
      "date": "2024-01-16"
      // Brak pola 'likes' - to OK w modelu dokumentowym!
    }
  ],
  "metadata": {
    "views": 1250,
    "reading_time": "5 min",
  }
}
```

```
    "published": true
  }
}

// Inny post może mieć zupełnie inną strukturę:
{
  "_id": "507f1f77bcf86cd799439012",
  "title": "Nowości w JavaScript",
  "author": "Maria Zielińska", // String zamiast obiektu!
  "category": "programming",
  "body": "ES2024 wprowadza...", // Inna nazwa pola niż 'content'
  "created_at": "2024-01-20T10:30:00Z"
}
```



## Model klucz-wartość

Najprostszy model – jak **słownik** lub **hashmapa**. Klucz (unikalny identyfikator) → Wartość (dowolne dane).

### Kluczowe cechy:

- **Ekstremalnie szybki dostęp**  $O(1)$  po kluczu
- **Brak schematu** – wartość może być czymkolwiek
- **Proste operacje**: GET, SET, DELETE
- **TTL** (Time To Live) – automatyczne usuwanie po czasie



## Przykład modelu klucz-wartość: System cache i sesji (Redis)

```
# Cache stron internetowych
SET "page:/products/laptop" "<html>...duża strona...</html>"
EXPIRE "page:/products/laptop" 300 # Usunąć po 5 minutach

# Koszyk zakupowy użytkownika
HSET "cart:user123" "laptop" 2 "phone" 1 "headphones" 1
EXPIRE "cart:user123" 3600 # Wygaśnięcie koszyka po 1 godzinę

# Licznik odwiedzin
INCR "page_views:/homepage"
INCRBY "page_views:/homepage" 5

# Leaderboard gry
```

```
ZADD "game_leaderboard" 1500 "player1" 2300 "player2" 1800 "player3"
ZREVRANGE "game_leaderboard" 0 2 WITHSCORES # Top 3 graczy

# Sesja użytkownika
SET "session:abc123" '{"user_id": 456, "username": "janek", "role": "admin"}'
EXPIRE "session:abc123" 1800 # Sesja wygasa po 30 minut
```



## Model szerokokolumnowy (kolumnowy)

Dane przechowywane w **kolumnach** zamiast wierszy. Każda kolumna przechowywana osobno, optymalna dla agregacji.

### Kluczowe cechy:

- **Przechowywanie kolumnowe** – szybkie skanowanie kolumn
- **Skalowanie poziome** – partycjonowanie
- **Zoptymalizowany dla zapytań agregujących**
- **Tolerancja na brak spójności** (eventual consistency)



## Przykład modelu szerokokolumnowy: Dane sensorów IoT (Cassandra)

```
-- Tworzenie tabeli (Cassandra Query Language - podobne do SQL)
CREATE TABLE sensor_readings (
    sensor_id uuid,
    date date,
    timestamp timestamp,
    temperature decimal,
    humidity decimal,
    pressure decimal,
    location text,
    PRIMARY KEY ((sensor_id, date), timestamp)
) WITH CLUSTERING ORDER BY (timestamp DESC);

-- Wstawianie danych
INSERT INTO sensor_readings
(sensor_id, date, timestamp, temperature, humidity, location)
VALUES (
    uuid(),
    '2024-01-20',
    '2024-01-20 10:30:00',
    23.5,
    65.0,
    'Warsaw, Poland'
);
```

```
22.5,  
45.0,  
'Warszawa'  
);  
  
SELECT location, AVG(temperature) as avg_temp  
FROM sensor_readings  
WHERE date = '2024-01-20'  
GROUP BY location;
```



## Model grafowy

Dane jako **węzły** (encje) i **krawędzie** (relacje). Idealny do reprezentowania powiązań.



## Kluczowe cechy:

- **Węzły** – encje z właściwościami
- **Krawędzie** – relacje z typem i właściwościami
- **Przechodzenie grafu** – znajdowanie ścieżek
- **Wykrywanie wzorców** – znajdowanie podgrafów



## Przykład model grafowy

*// Cypher - język zapytań dla grafów*

*// Tworzenie użytkowników (węzłów)*

```
CREATE (jan:User {  
  id: 1,  
  name: "Jan Kowalski",  
  age: 28,  
  city: "Warszawa"  
})
```

```
CREATE (anna:User {  
  id: 2,
```

```

    name: "Anna Nowak",
    age: 32,
    city: "Kraków"
})

CREATE (tomek:User {
    id: 3,
    name: "Tomasz Wiśniewski",
    age: 25,
    city: "Warszawa"
})

// Tworzenie relacji
CREATE (jan)-[:FRIENDS_WITH {since: "2022-05-10"}]->(anna)
CREATE (jan)-[:FRIENDS_WITH {since: "2023-01-15"}]->(tomek)
CREATE (anna)-[:FOLLOWS]->(tomek)

// Jan lubi programowanie
CREATE (java:Skill {name: "Java", category: "programming"})
CREATE (python:Skill {name: "Python", category: "programming"})
CREATE (jan)-[:KNOWS {level: "expert"}]->(java)
CREATE (jan)-[:KNOWS {level: "intermediate"}]->(python)
CREATE (anna)-[:KNOWS {level: "beginner"}]->(python)

// Zapytanie: znajdź znajomych Jana którzy znają Javę
MATCH (jan:User {name: "Jan Kowalski"})-[:FRIENDS_WITH]-(friend:User)
WHERE (friend)-[:KNOWS]->(:Skill {name: "Java"})
RETURN friend.name

// Zapytanie: znajdź ścieżkę między Janem a osobą znającą Python
MATCH path = (jan:User {name: "Jan Kowalski"})-[*..3]-(pythonExpert:User)
WHERE (pythonExpert)-[:KNOWS]->(:Skill {name: "Python"})
RETURN path

```



## Model obiektowy i obiektowo-relacyjny



### 1. Model obiektowy

Model obiektowy przenosi zasady **programowania obiektowego (OOP)** do bazy danych. Dane przechowywane są jako **obiekty** z:

- **Atrybutami** (pola, właściwości)
- **Metodami** (zachowanie)
- **Dziedziczeniem** (hierarchie klas)
- **Enkapsulacją** (hermetyzacja)



### Kluczowe cechy:

- **Obiekty z identyfikatorami OID** (Object ID)
- **Bezpośrednie mapowanie** obiektów aplikacji na obiekty bazy
- **Brak potrzeby ORM** (Object-Relational Mapping)
- **Zachowanie z danymi** – metody przechowywane w bazie



### Przykład modelu obiektowego: System biblioteczny (db4o)

```
// Klasy Java (również przechowywane w bazie)
public class Osoba {
    private String id;
    private String imie;
    private String nazwisko;

    public Osoba(String imie, String nazwisko) {
        this.id = UUID.randomUUID().toString();
        this.imie = imie;
        this.nazwisko = nazwisko;
    }

    // Metody - również dostępne w zapytaniach
    public String getPełnaNazwa() {
        return imie + " " + nazwisko;
    }
}

public class Czytelnik extends Osoba {
    private Date dataRejestracji;
    private List<Wypożyczenie> wypożyczenia = new ArrayList<>();

    public Czytelnik(String imie, String nazwisko) {
        super(imie, nazwisko);
        this.dataRejestracji = new Date();
    }

    public void wypożyczKsiążkę(Książka książka) {
        Wypożyczenie w = new Wypożyczenie(this, książka);
        wypożyczenia.add(w);
        książka.setWypożyczona(true);
    }
}
```

```

        public int getLiczbaWypozycczen() {
            return wypozycczenia.size();
        }
    }

    public class Ksiazka {
        private String isbn;
        private String tytul;
        private Autor autor;
        private boolean wypozyczona;

        // Metoda w klasie przechowywanej w bazie
        public boolean jestDostepna() {
            return !wypozyczona;
        }
    }

    // Operacje na bazie obiektowej (db4o)
    ObjectContainer db = Db4oEmbedded.openFile("biblioteka.db");

    // Zapis obiektu - bez mapowania!
    Czytelnik czytelnik = new Czytelnik("Jan", "Kowalski");
    Ksiazka ksiazka = new Ksiazka("123-456", "Programowanie w Java");
    db.store(czytelnik);
    db.store(ksiazka);

    // Zapytanie natywne - użycie metod obiektów!
    List<Czytelnik> czytelnicy = db.query(new Predicate<Czytelnik>() {
        public boolean match(Czytelnik czytelnik) {
            // Używamy metody obiektu w zapytaniu!
            return czytelnik.getLiczbaWypozycczen() > 5;
        }
    });

    // Zapytanie QBE (Query by Example)
    Czytelnik przykład = new Czytelnik(null, "Kowalski");
    List<Czytelnik> wynik = db.queryByExample(przykład);

    // Pobranie powiązanych obiektów
    czytelnik.wypozyczKsiazke(ksiazka); // Użycie metody obiektu
    db.store(czytelnik); // Zapis całego grafu obiektów

    db.close();

```



## Struktura danych w bazie obiektowej:

Obiekt Czytelnik [OID: 0x1234]:

Nagłówek: OID=0x1234, Typ=Czytelnik
Dane:
- id: "550e8400-e29b-41d4-a716-446655"
- imie: "Jan"

- nazwisko: "Kowalski"
- dataRejestracji: 2024-01-20
- wypozyczenia: [OID:0x5678, OID:0x9ABC]

Metody (wskazania):
- getPelnaNazwa()
- wypożyczKsiążke()
- getLiczbaWypozycczen()

↓

Obiekt Wypozyczenie [OID: 0x5678]...

Obiekt Książka [OID: 0x9ABC]...



## 2. Model obiektowo-relacyjny (OR)

Rozszerzenie modelu relacyjnego o cechy obiektowe. Łączy zalety obu światów:

- **Tabele** z relacyjnego
- **Typy złożone**, dziedziczenie, metody z obiektowego



### Kluczowe cechy:

1. **UDT** (User-Defined Types) – własne typy danych
2. **Tabele zagnieżdżone** – tabele w tabelach
3. **Dziedziczenie tabel** – hierarchie tabel
4. **Metody w tabelach** – zachowanie w bazie
5. **REF** – referencje do wierszy



## Przykład modelu obiektowo-relacyjny: System kadrowy (PostgreSQL z rozszerzeniami OR)

-- PostgreSQL z rozszerzeniami obiektowo-relacyjnymi

-- 1. TWORZENIE TYPÓW ZŁOŻONYCH (UDT)

```
CREATE TYPE adres_typ AS (  
  ulica VARCHAR(100),  
  numer VARCHAR(10),  
  miasto VARCHAR(50),  
  kod_pocztowy VARCHAR(6)
```



```
);
```

```
CREATE TYPE kontakt_typ AS (  
    email VARCHAR(100),  
    telefon VARCHAR(20),  
    telefon_komórkowy VARCHAR(20)  
);
```

```
-- 2. TABELA Z UDT I METODAMI
```

```
CREATE TABLE pracownicy (  
    id SERIAL PRIMARY KEY,  
    imie VARCHAR(50),  
    nazwisko VARCHAR(50),  
    adres adres_typ, -- Typ złożony!  
    kontakt kontakt_typ,  
    data_zatrudnienia DATE,  
    wynagrodzenie DECIMAL(10,2),
```

```
-- Metoda jako funkcja w tabeli
```

```
FUNCTION pelny_adres() RETURNS VARCHAR  
AS $$  
BEGIN  
    RETURN (adres).ulica || ' ' || (adres).numer || ', ' ||  
        (adres).kod_pocztowy || ' ' || (adres).miasto;  
END;  
$$ LANGUAGE plpgsql  
);
```

```
-- 3. DZIEDZICZENIE TABEL
```

```
CREATE TABLE osoby (  
    id SERIAL PRIMARY KEY,  
    imie VARCHAR(50),  
    nazwisko VARCHAR(50),  
    pesel VARCHAR(11)  
);
```

```
CREATE TABLE klienci (  
    numer_karty VARCHAR(20),  
    data_rejestracji DATE  
) INHERITS (osoby); -- Dziedziczenie!
```

```
CREATE TABLE dostawcy (  
    nip VARCHAR(10),  
    regon VARCHAR(9)  
) INHERITS (osoby);
```

```
-- 4. TABELA ZAGNIEŻDŻONE (kolekcje)
```

```
CREATE TABLE zamowienia (  
    id SERIAL PRIMARY KEY,  
    data_zamowienia DATE,  
    klient_id INTEGER,
```

```
-- Zagnieżdżona tabela z pozycjami zamówienia
```

```
pozycje_zamowienia pozycja_zamowienia_typ[]  
);
```

```
CREATE TYPE pozycja_zamowienia_typ AS (  
    produkt_id INTEGER,  
    nazwa_produktu VARCHAR(100),  
    ilosc INTEGER,  
    cena_jednostkowa DECIMAL(10,2),  
  
    -- Metoda w typie  
    FUNCTION wartosc_calkowita() RETURNS DECIMAL  
    AS $$  
    BEGIN  
        RETURN $1.ilosc * $1.cena_jednostkowa;  
    END;  
    $$ LANGUAGE plpgsql  
);
```

```
-- 5. OBIEKTOWE ZAPYTANIA
```

```
-- Wstawianie z typami złożonymi
```

```
INSERT INTO pracownicy (imie, nazwisko, adres, kontakt)  
VALUES (  
    'Jan',  
    'Kowalski',  
    ROW('Warszalkowska', '123', 'Warszawa', '00-001')::adres_typ,  
    ROW('jan@firma.pl', '222333444', '555666777')::kontakt_typ  
);
```

```
-- Dostęp do pól typów złożonych
```

```
SELECT  
    imie,  
    nazwisko,  
    (adres).miasto, -- Dostęp do pola typu złożonego  
    (kontakt).email,  
    pelny_adres() -- Wywołanie metody  
FROM pracownicy  
WHERE (adres).miasto = 'Warszawa';
```

```
-- Zapytanie na zagnieżdżonych danych
```

```
SELECT  
    z.id,  
    z.data_zamowienia,  
    pz.nazwa_produktu,  
    pz.ilosc,  
    pz.cena_jednostkowa,  
    pz.wartosc_calkowita() -- Metoda na typie zagnieżdżonym  
FROM zamowienia z,  
    UNNEST(z.pozycje_zamowienia) AS pz -- Rozwijanie zagnieżdżonej tabeli  
WHERE pz.wartosc_calkowita() > 1000;
```

```
-- 6. REFERENCJE OBIEKTÓW (OID)
```

```
CREATE TABLE produkty OF produkt_typ ( -- Tabela obiektów  
    PRIMARY KEY (id)
```

```

) WITH OIDS; -- Obiektowe identyfikatory

CREATE TABLE zamowienia_ref (
  id SERIAL,
  produkt REF produkt_typ -- Referencja do obiektu
);

-- 7. POLIMORFICZNE FUNKCJE
CREATE OR REPLACE FUNCTION oblicz_podatek(osoba osoby)
RETURNS DECIMAL AS $$
BEGIN
  -- Różne implementacje w zależności od typu
  IF TG_OP = 'klienci' THEN
    RETURN 0; -- Klienci nie płacą podatku
  ELSIF TG_OP = 'dostawcy' THEN
    RETURN 1000; -- Stały podatek
  END IF;
  RETURN 0;
END;
$$ LANGUAGE plpgsql;

```

## Lekcja

### Temat: Projektowanie baz danych



**Projektowanie baz danych to proces tworzenia logicznej i fizycznej struktury bazy danych, która efektywnie przechowuje, organizuje i zarządza danymi, spełniając wymagania konkretnej aplikacji lub systemu.**

Kluczowym aspektem tego procesu jest świadomość, że **w bazie danych przechowujemy tylko niektóre informacje o świecie rzeczywistym**. Wybór właściwych wycinków rzeczywistości i dotyczących ich danych jest bardzo istotny — od niego zależy prawidłowe działanie bazy. Aby ten wybór był właściwy, należy wskazać informacje, które powinny być przechowywane w bazie danych, oraz określić ich strukturę.



### Główne etapy projektowania

Cały proces projektowania bazy danych możemy podzielić na kilka etapów:



## 1. Planowanie bazy danych

- Analiza wymagań systemu i użytkowników
- Określenie, które **fragmenty rzeczywistości** mają być modelowane
- Identyfikacja **konkretnych danych** potrzebnych systemowi
- Definiowanie celów, zakresu i ograniczeń projektu
- Określenie przyszłych potrzeb roszszerzania bazy



## 2. Tworzenie modelu konceptualnego (diagramy ERD)

- Identyfikacja **encji** (tabel) reprezentujących wybrane byty rzeczywiste
- Określenie **atrybutów** (kolumn) opisujących te byty
- Definiowanie **relacji** między encjami
- Tworzenie diagramów ERD (Entity-Relationship Diagrams)
- **Selekcja** - co włączyć, co pominąć w modelu danych



## 3. Transformacja modelu konceptualnego na model relacyjny

- Przekształcenie encji na tabele
- Konwersja atrybutów na kolumny
- Zamiana relacji na klucze obce i tabele łączące
- Definiowanie typów danych dla kolumn
- Określenie ograniczeń (constraints)



## 4. Proces normalizacji bazy danych

- **1NF** (Pierwsza postać normalna): eliminacja powtarzających się grup
- **2NF**: eliminacja zależności częściowych od klucza
- **3NF**: eliminacja przechodnich zależności
- Ocena konieczności dalszych postaci normalnych (BCNF, 4NF, 5NF)
- Rozważenie celowej denormalizacji dla poprawy wydajności



## 5. Wybór struktur i określenie zasad dostępu do bazy danych

- Wybór silnika bazy danych w MySQL (InnoDB, MyISAM)
- Definiowanie indeksów dla optymalizacji zapytań
- Określenie zasad bezpieczeństwa i uprawnień
- Planowanie kopii zapasowych i recovery
- Optymalizacja struktur pod kątem wydajności



## Podstawowe pojęcia w projektowaniu baz danych



### Encja (Entity)

Encją jest każdy przedmiot, zjawisko, stan lub pojęcie, czyli każdy obiekt, który potrafimy odróżnić od innych obiektów (na przykład: osoba, samochód, książka, stan pogody).



### Zbiór encji

Encje podobne do siebie (opisywane za pomocą podobnych parametrów) grupujemy w **zbiory encji**. W praktyce relacyjnej każdy zbiór encji staje się tabelą w bazie danych.

#### Przykład:

- Zbiór encji "Studenci" - zawiera poszczególne encje: student Jan Kowalski, student Anna Nowak, itd.
- Zbiór encji "Książki" - zawiera: "Pan Tadeusz", "Kamienie na szaniec", "Dzieci z Bullerbyn "

**Wskazówka:** Projektując bazę danych, należy precyzyjnie zdefiniować encje i określić parametry, przy użyciu których będą opisywane.



### Atrybut (Attribute)

**Atrybut** to cecha opisująca encję. Encje mają określone cechy wynikające z ich natury. Cechy te nazywamy atrybutami.

**Atrybuty encji stają się kolumnami w tabeli.** Każdy atrybut reprezentuje jedną kolumnę w tabeli, która przechowuje wartości tego atrybutu dla poszczególnych encji.

**Zestaw atrybutów**, które określamy dla encji, zależy od potrzeb bazy danych. Nie wszystkie cechy encji muszą być przechowywane - tylko te istotne z punktu widzenia systemu.



### Dziedzina (Domena)

**Dziedzina** to zbiór dopuszczalnych wartości atrybutu. Atrybuty encji mogą przyjmować różne wartości. Projektując bazę danych, możemy określić, jakie wartości może przyjmować dany atrybut. Zbiór wartości atrybutu nazywamy dziedziną (domeną).



### Problem z projektowaniem bazy danych: Relacje wiele-do-wielu



### Przykład problemu: System rezerwacji hotelu

#### Błędne podejście:

```
CREATE TABLE reservations (  
    reservation_id INT PRIMARY KEY,  
    guest_name VARCHAR(100),  
    room_number VARCHAR(10),  
    check_in DATE,  
    check_out DATE  
);
```

**Problem:** Jeden gość może mieć wiele rezerwacji, a w jednej rezerwacji może być wielu gości (rodzina, grupa). Jeśli wpiszemy wszystkich gości w jednym polu `guest_name` jako "Jan Kowalski, Anna Kowalska, dziecko Kowalskie", pojawią się problemy:

1. **Redundancja danych** - ten sam gość powtarza się w wielu rezerwacjach
2. **Trudność wyszukiwania** - jak znaleźć wszystkie rezerwacje konkretnego gościa?
3. **Brak integralności** - nie ma kontroli nad poprawnością danych gości
4. **Problemy z modyfikacją** - zmiana danych gościa wymaga aktualizacji wielu rekordów



### Rozwiązanie: Wydzielenie encji Gości i tabeli łączącej

## Właściwe podejście:

```
-- Encja: Goście
CREATE TABLE guests (
  guest_id INT PRIMARY KEY AUTO_INCREMENT,
  first_name VARCHAR(50) NOT NULL,
  last_name VARCHAR(50) NOT NULL,
  email VARCHAR(100) UNIQUE,
  phone VARCHAR(20)
);

-- Encja: Rezerwacje
CREATE TABLE reservations (
  reservation_id INT PRIMARY KEY AUTO_INCREMENT,
  check_in DATE NOT NULL,
  check_out DATE NOT NULL,
  status ENUM('confirmed', 'pending', 'cancelled')
);

-- Tabela łącząca dla relacji wiele-do-wielu
CREATE TABLE reservation_guests (
  reservation_id INT,
  guest_id INT,
  is_main_guest BOOLEAN DEFAULT FALSE, -- dodatkowy atrybut relacji
  PRIMARY KEY (reservation_id, guest_id),
  FOREIGN KEY (reservation_id) REFERENCES reservations(reservation_id),
  FOREIGN KEY (guest_id) REFERENCES guests(guest_id)
);
```



### Zalety tego rozwiązania:

1. **Eliminacja redundancji** - dane każdego gościa przechowywane raz
2. **Łatwe wyszukiwanie** - proste zapytania o rezerwacje danego gościa
3. **Integralność danych** - kontrola przez klucze obce
4. **Elastyczność** - możliwość dodawania dowolnej liczby gości do rezerwacji
5. **Łatwość modyfikacji** - zmiana danych gościa w jednym miejscu



### Inne częste problemy i ich rozwiązania:



#### **Problem 1: Mieszanie jednostek miary w jednej kolumnie**

**Błąd:** price DECIMAL(10,2) bez określenia waluty

**Rozwiązanie:** Dodaj kolumnę currency VARCHAR(3) lub normalizuj do osobnej tabeli walut



#### **Problem 2: Przechowywanie historii zmian w głównej tabeli**

**Błąd:** Aktualne i historyczne dane mieszane w jednej tabeli

**Rozwiązanie:** Wydziel tabele historii z timestampami i flagą `is_current`



### **Problem 3: Nieatomowe atrybuty (adres w jednym polu)**

**Błąd:** `address VARCHAR(200)` zawierający ulicę, miasto, kod

**Rozwiązanie:** Podziel na osobne kolumny: `street`, `city`, `postal_code`, `country`

### **Zasada: "Jedna encja = jedna odpowiedzialność"**

Najczęstszy błąd w projektowaniu to próba upchnięcia zbyt wielu conceptów w jednej encji/tabeli. Każda tabela powinna reprezentować **jeden jasno zdefiniowany byt** z rzeczywistości. Jeśli zauważysz, że:

- dane się powtarzają
- masz puste pola dla niektórych rekordów
- trudno jest modyfikować dane
- zapytania stają się skomplikowane



## **Tworzenie modelu konceptualnego (diagramy ERD)**



### **Definicja modelu konceptualnego**

**Konceptualne projektowanie bazy danych to konstruowanie schematu danych niezależnego od wybranego modelu danych, docelowego systemu zarządzania bazą danych, programów użytkowych czy języka programowania.** Jest to abstrakcyjna reprezentacja struktury danych skupiona na ich znaczeniu i relacjach, a nie na implementacji.



### **Diagramy ERD (Entity Relationship Diagram)**

Do tworzenia modelu graficznego schematu bazy danych wykorzystywane są diagramy związków encji, z których najpopularniejsze są diagramy **ERD (ang. Entity Relationship Diagram)**. Pozwalają one na:

1. **Modelowanie struktur danych** oraz związków zachodzących między tymi strukturami
2. **Prawie bezpośrednie przekształcenie** diagramu w schemat relacyjny
3. **Analizę struktury** bazy danych na wysokim poziomie abstrakcji



## 4. Dokumentację tworzonego systemu baz danych

Na diagramy ERD składają się trzy rodzaje elementów:

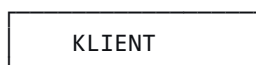
- ✓ zbiory encji,
- ✓ atrybuty encji,
- ✓ związki zachodzące między encjami.



### 1. Zbiory encji (Entities)

**Encja** to reprezentacja obiektu przechowywanego w bazie danych. Graficzną reprezentacją encji jest najczęściej prostokąt.

Przykład:



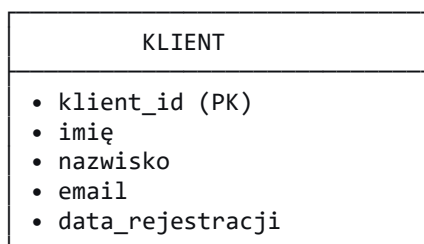
### 2. Atrybuty encji (Attributes)

**Atrybut** opisuje encję. Może on być liczbą, tekstem lub wartością logiczną. W relacyjnym modelu baz danych atrybut jest reprezentowany przez kolumnę tabeli.

**Reprezentacje atrybutów:**

- W notacji Chena: owal połączony z encją
- W notacji "pudełkowej": lista wewnątrz prostokąta encji

**Przykład encji Klient z atrybutami:**



### 3. Związki między encjami (Relationships)

**Związek** to powiązanie między dwoma zbiorami encji. Każdy związek ma dwa końce, do których są przypisane:



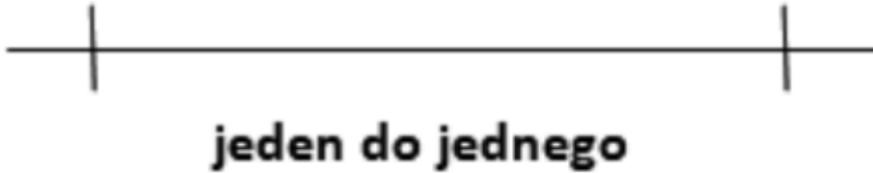
#### a) Stopień związku (Cardinality)

Określa, jakiego typu związek zachodzi między encjami:

1.



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

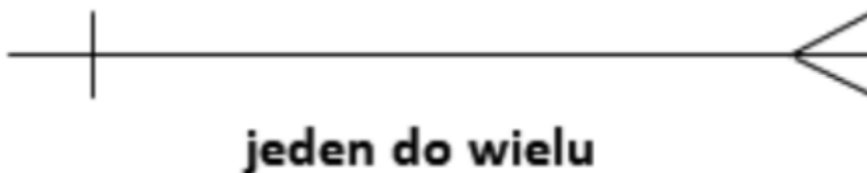


*Przykład: Pracownik ↔ Samochód służbowy*

2.



**Jeden do wielu (1:N)**

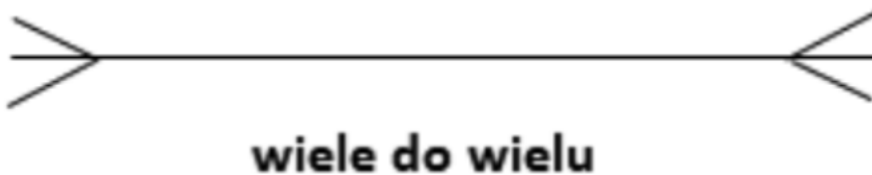


*Przykład: Klient ↔ Zamówienia*

3.



**Wiele do wielu (N:M)**



*Przykład: Student ↔ Kursy*

b)



**Opcjonalność związku (Optionality)**

Określa, czy związek jest opcjonalny, czy wymagany:


- 



**Wymagany (mandatory):**



związek wymagany

-  **Opcjonalny (optional):**



związek opcjonalny

Diagramy ERD spotyka się w wielu różnych notacjach, na przykład: Martina, Bachmana, Chena, IDEF1X

Prosty przykład diagramu ERD w notacji Martina:

