

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 wewnętrz 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 szeregow

d) Model grafowy

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

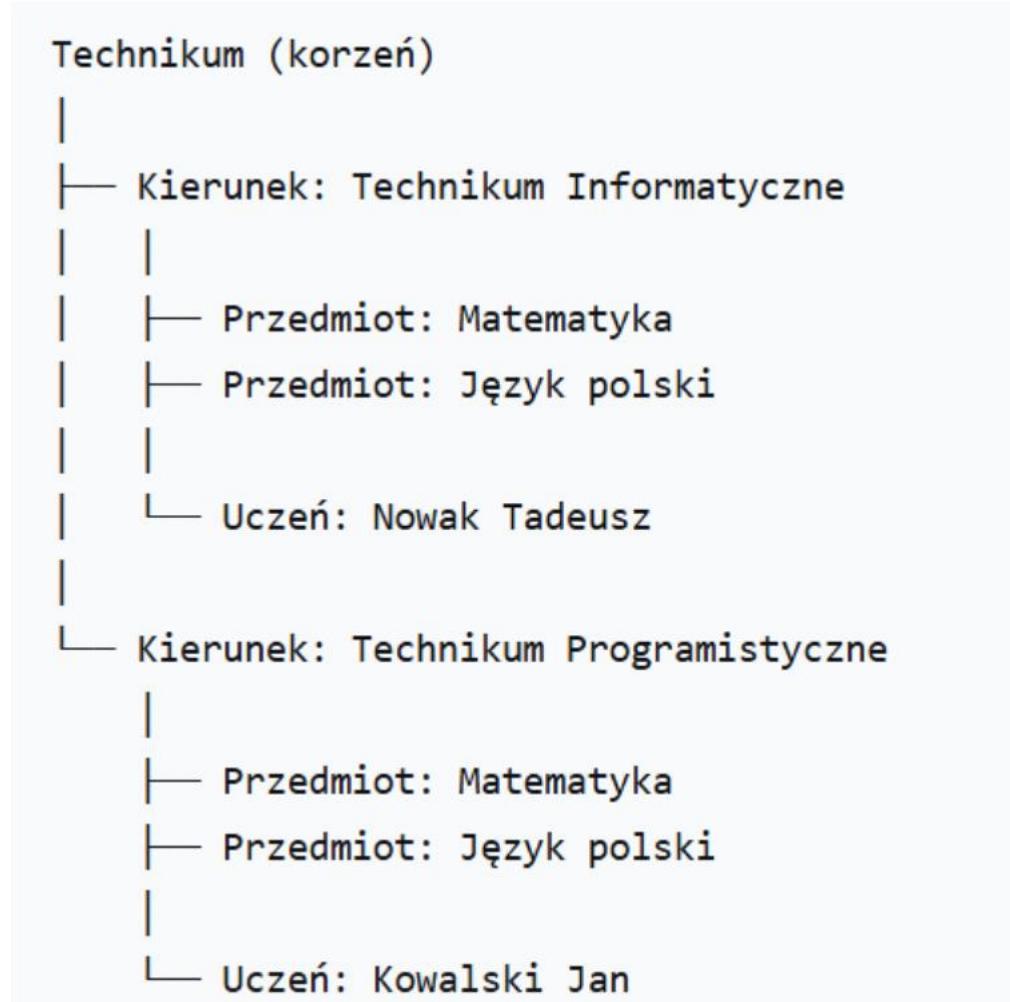
5. Model obiektowy i obiektowo-relacyjny

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



Model sieciowy

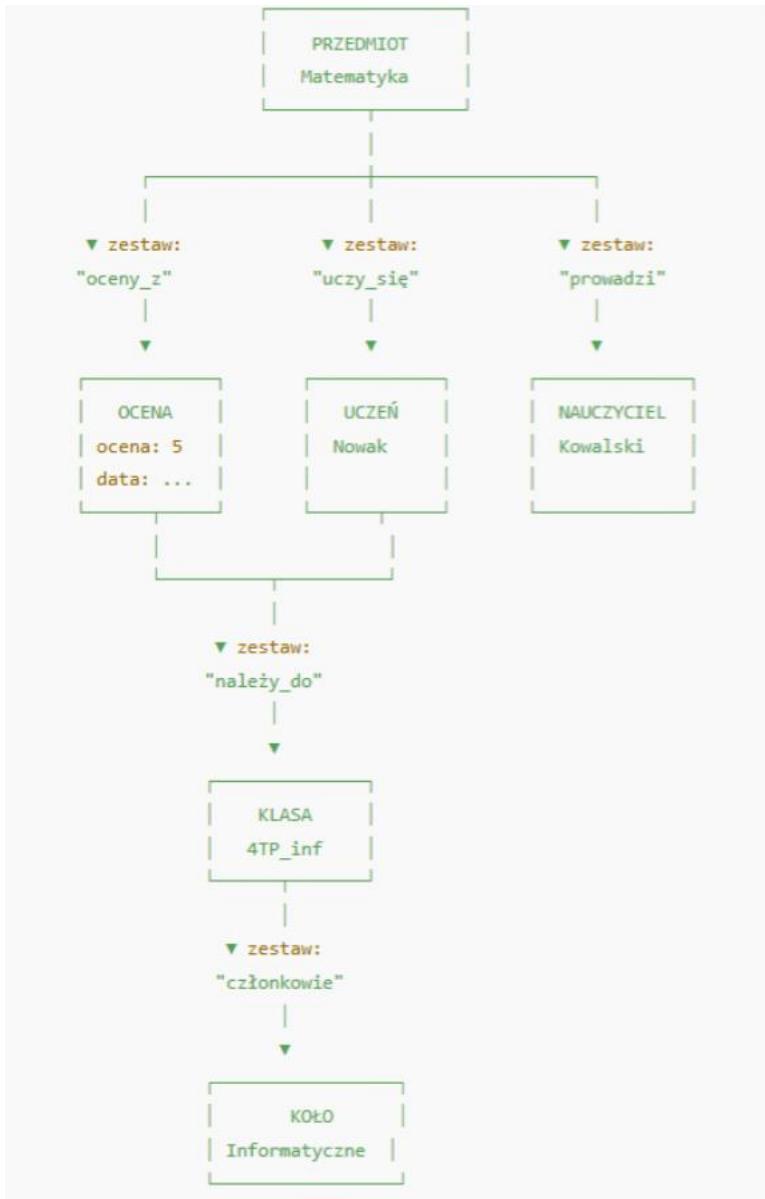
Model sieciowy powstał jako rozwinięcie modelu hierarchicznego, aby rozwiązać jego główne ograniczenia. Został sformalizowany przez grupę **CODASYL** (Conference on Data Systems Languages) w latach 60-70.

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



Model relacyjny

Model relacyjny to najpopularniejszy współczesny model baz danych, stworzony przez Edgara F. Codda w 1970 roku. Opiera się na matematycznej teorii mnogości i algebrze relacji.

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):

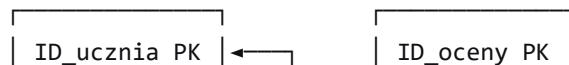
ID_ucznia	ID_przedmiotu
1	1
1	2
2	1
3	3

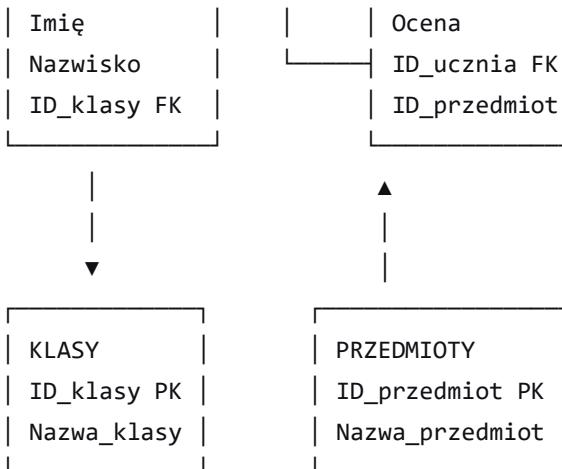
PK: (ID_ucznia, ID_przedmiotu) - klucz złożony

2. Relacje między tabelami (diagram ER):

UCZNIOWIE

OCENY





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:

- **Samoopisujące 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 # Usuń 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 wygasza 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 szerokokolumnowego: 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',
    22.5,
    45.0,
    'Warszawa'
);

-- Struktura przechowywania (kolumnowa):
-- Na dysku dane są grupowane KOLUMNOWO:
-- Wszystkie wartości 'temperature' obok siebie
```

```
-- Wszystkie wartości 'humidity' obok siebie

-- Zapytanie: średnia temperatura wg lokalizacji
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 Java
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

// Wizualizacja grafu:
// (Jan) --FRIENDS_WITH--> (Anna) --FOLLOWS--> (Tomasz)
//   |           /
//   ---KNOWS-->(Java)      ---KNOWS-->(Python)
//   |
//   ---KNOWS-->(Python)

```

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)

```
// Klasa 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 getPelnaNazwa() {
        return imie + " " + nazwisko;
    }
}

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

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

    public void wypożyczKsiążkę(Książka książka) {
        Wypozyczenie w = new Wypozyczenie(this, książka);
        wypozyczenia.add(w);
        książka.setWypożyczona(true);
    }

    public int getLiczbaWypożyczeń() {
        return wypozyczenia.size();
    }
}

public class Książka {
    private String isbn;
    private String tytuł;
    private Autor autor;
```

```

private boolean wypożyczona;

// Metoda w klasie przechowywanej w bazie
public boolean jestDostępna() {
    return !wypożyczona;
}

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

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

// 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.getLiczbaWypożyczeń() > 5;
    }
});

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

// Pobranie powiązanych obiektów
czytelnik.wypożyczKsiążkę(książka); // 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:
<ul style="list-style-type: none"> - id: "550e8400-e29b-41d4-a716-446655" - imię: "Jan" - nazwisko: "Kowalski" - dataRejestracji: 2024-01-20 - wypożyczenia: [OID:0x5678, OID:0x9ABC]
Metody (wskazania):
<ul style="list-style-type: none"> - getPelnaNazwa() - wypożyczKsiążkę() - getLiczbaWypożyczeń()

↓

```
Obiekt Wypozyczenie [OID: 0x5678]...
Obiekt Ksiazka [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. TABELE 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_t 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('Marszałkowska', '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 zagnieżdzonej 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 rozszerzania 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

Rozwiążanie: 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 konceptów 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:



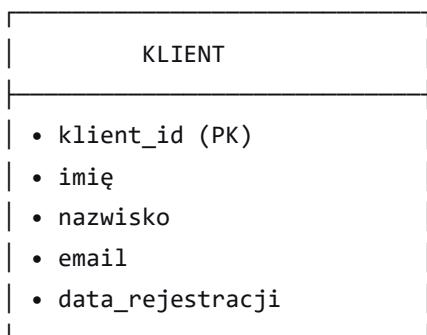
2. Atrybuty encji (Attributes)

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

Przykład encji Klient z atrybutami:



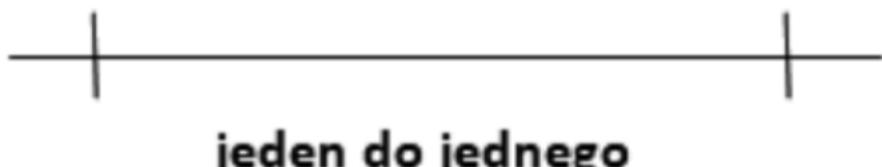
3. Związki między encjami (Relationships)

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

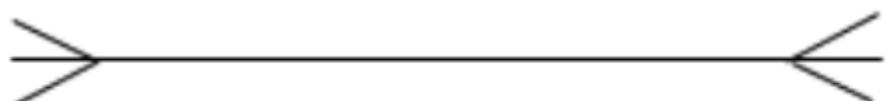
2. Jeden do wielu (1:N)



jeden do wielu

Przykład: Klient ↔ Zamówienia

3. Wiele do wielu (N:M)



wiele do wielu

Przykład: Student ↔ Kursy

b) Opcjonalność związku (Optionality)

Określa, czy związek jest opcjonalny, czy wymagany:

- **Wymagany (mandatory):**



związek wymagany

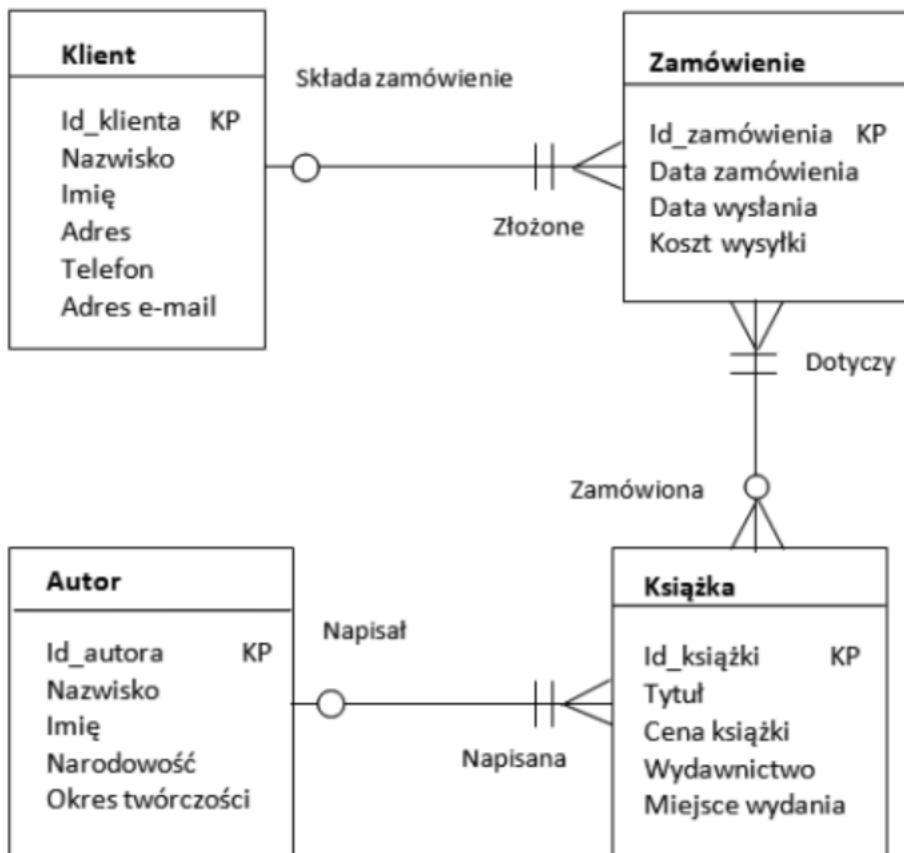
- **Opcjonalny (optional):**



związek opcjonalny

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

Prosty przykład diagramu ERD w notacji Martina:



Lekcja

Temat: Transformacja modelu konceptualnego na model relacyjny

1. Podstawowe Pojęcia



Model konceptualny – to abstrakcyjny, wysokopoziomowy opis danych, zwykle tworzony w **ERD (Entity-Relationship Diagram)**.

- Mówimy o **encjach** (np. Osoba, Paszport)
- Ich **atrybutach** (np. imię, nazwisko, numer_pasportu)
- Związkach między encjami (1:1, 1:N, N:M)



Model relacyjny – to konkretny sposób zapisania tych danych w tabelach w relacyjnej bazie danych.

- Tworzymy **tabele, kolumny, klucze główne i obce**
- Zachowujemy struktury z modelu konceptualnego

Transformacja = proces przekształcania **ERD** w **tabele relacyjne**.

2. Podstawowe Zasady Transformacji



Encja → Tabela

Każda encja staje się tabelą w bazie danych.



Model konceptualny:

ENCJA: Student

```
|  
|   Atrybuty:  
|   |   id_studenta (PK)  
|   |   imie  
|   |   nazwisko  
|   |   data_urodzenia  
|   |   email
```



Model relacyjny:

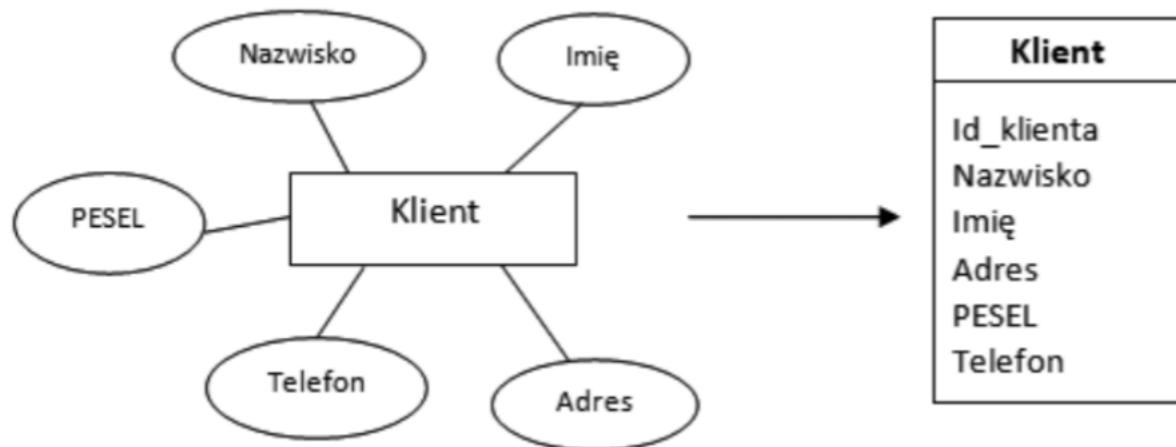
```
CREATE TABLE Student (  
    id_studenta INT PRIMARY KEY,  
    imie VARCHAR(50) NOT NULL,  
    nazwisko VARCHAR(50) NOT NULL,  
    data_urodzenia DATE,  
    email VARCHAR(100) UNIQUE  
);
```

Do opisu każdego zbioru podobnych encji stosuje się oddzielną tabelę. Jednej encji odpowiada jeden wiersz. Atrybutowi odpowiada kolumna. Dla każdego atrybutu określa się typ informacji.



Atrybuty → Kolumny

Atrybuty encji stają się kolumnami tabeli.



3. Transformacja Relacji



Relacja 1:1 (jeden do jednego)

Przykład: Student ↔ Legitymacja

Model konceptualny:

Student (1) — posiada — (1) Legitymacja

Model relacyjny - Opcja A (klucz obcy w jednej tabeli):

```

CREATE TABLE Student (
    id_studenta INT PRIMARY KEY,
    imie VARCHAR(50),
    nazwisko VARCHAR(50)
);

CREATE TABLE Legitymacja (
    id_legitymacji INT PRIMARY KEY,
    numer VARCHAR(20) UNIQUE,
    data_wydania DATE,
    id_studenta INT UNIQUE, -- Klucz obcy + UNIQUE
    FOREIGN KEY (id_studenta) REFERENCES Student(id_studenta)
);

```



Relacja 1:N (jeden do wielu)

Przykład: Wykładowca (1) ↔ Przedmioty (N)

Model konceptualny:

Wykładowca (1) — prowadzi — (N) Przedmiot

```

CREATE TABLE Wykladowca (
    id_wykladowcy INT PRIMARY KEY,
    imie VARCHAR(50),
    nazwisko VARCHAR(50),
    tytul_naukowy VARCHAR(50)
);

CREATE TABLE Przedmiot (
    id_przedmiotu INT PRIMARY KEY,
    nazwa VARCHAR(100),
    ects INT,
    id_wykladowcy INT,
    FOREIGN KEY (id_wykladowcy) REFERENCES Wykladowca(id_wykladowcy)
);

```



Relacja M:N (wiele do wielu)

Przykład: Student (N) ↔ Przedmiot (M)

Model konceptualny:

Student (N) — zapisuje się na — (M) Przedmiot

Model relacyjny (tabela asocjacyjna):

```
CREATE TABLE Student (
    id_studenta INT PRIMARY KEY,
    imie VARCHAR(50),
    nazwisko VARCHAR(50)
);

CREATE TABLE Przedmiot (
    id_przedmiotu INT PRIMARY KEY,
    nazwa VARCHAR(100),
    ects INT
);

-- TABELA ASOCJACYJNA lub pośrednia
CREATE TABLE Zapis_na_przedmiot (
    id_zapisu INT PRIMARY KEY,
    id_studenta INT,
    id_przedmiotu INT,
    data_zapisu DATE,
    ocena DECIMAL(3,2),
    FOREIGN KEY (id_studenta) REFERENCES Student(id_studenta),
    FOREIGN KEY (id_przedmiotu) REFERENCES Przedmiot(id_przedmiotu),
    UNIQUE(id_studenta, id_przedmiotu)
);
```



Normalizacja tabel



Normalizację stosuje się, aby sprawdzić, czy zaprojektowane tabele mają prawidłową strukturę. Kiedy wstępny projekt tabel jest gotowy, rozpoczynamy normalizację. Pozwala określić, czy założenia projektu zostały przydzielone do właściwych tabel. Natomiast nie da odpowiedzi na pytanie, czy projekt bazy danych jest prawidłowy.



Stosowane są cztery reguły normalizacji, ale w większości projektów baz danych wystarczy sprawdzić trzy pierwsze.



Dla każdej z nich stosowane są określenia:

- ✓ pierwsza postać normalna (I PN),

- ✓ druga postać normalna (II PN),
- ✓ trzecia postać normalna (III PN).



Pierwsza postać normalna



Tabela jest w pierwszej postaci normalnej (I PN), gdy każdy wiersz w tabeli przechowuje informacje o pojedynczym obiekcie, a każde pole tabeli zawiera informację elementarną (atomową).



Przykłady:



NIEWŁAŚCIWE (nieatomowe wartości):

Tabla Klienci:

ID	Imię	Telefony
1	Jan	123-456-789, 987-654-321
2	Anna	555-111-222

Problem: Kolumna "Telefony" zawiera 2 numery w jednej komórce, rozdzielone przecinkiem.



WŁAŚCIWE (wartości atomowe):

Rozwiązanie 1: Dla telefonów - osobne rekordy

ID	ID_Klienta	Telefon

1	1	123-456-789
2	1	987-654-321
3	2	555-111-222



Jak sprawdzić, czy wartość jest atomowa?

Zadaj pytanie: "**Czy kiedykolwiek będę potrzebował tylko części tej wartości?**"

- Jeśli odpowiedź brzmi "**tak**" → prawdopodobnie nie jest atomowa
- Jeśli odpowiedź brzmi "**nie**, zawsze używam całej wartości" → prawdopodobnie jest atomowa



Druga postać normalna



Tabela jest w drugiej postaci normalnej (II PN), jeżeli jest w pierwszej postaci normalnej (I PN) oraz każde z pól niewchodzących w skład klucza podstawowego zależy od całego klucza, a nie od jego części.



PRZYKŁAD PRZED 2NF (spełnia 1NF, ale nie 2NF):

Wyobraźmy się sklep internetowy. Jedna tabela przechowuje **pozycje zamówień**:

Tabela Pozycje_Zamówienia:

ID_Zamówienia	ID_Produktu	Ilość	Nazwa_Produktu	Cena_Produktu	Kategoria
100	5	2	Laptop Gaming	4500	Elektronika
100	8	1	Mysz bezprzewodowa	200	Akcesoria
101	5	1	Laptop Gaming	4500	Elektronika
102	8	3	Mysz bezprzewodowa	200	Akcesoria

- **Klucz główny (złożony):** (ID_Zamówienia, ID_Produktu)
 - Ta kombinacja jednoznacznie identyfikuje każdą pozycję (w zamówieniu 100 może być tylko jeden produkt 5).
- **Koluby:**
 - Ilość zależy od **całego klucza**. Aby wiedzieć, ile sztuk produktu 5 zamówiono, muszę znać **zarówno numer zamówienia (100), jak i produkt (5)**.
 - Nazwa_Produktu, Cena_Produktu, Kategoria zależą **tylko od części klucza (ID_Produktu)**. Nazwa "Laptop Gaming" jest taka sama, niezależnie od tego, w którym zamówieniu (100 czy 101) się pojawia.



PROBLEMY (anomalie):

1. **Niespójność przy aktualizacji:** Jeśli zmienię nazwę produktu z "Laptop Gaming" na "Laptop Extreme", muszę zaktualizować **wszystkie wiersze** gdzie **ID_Produktu=5**. Jeśli pominę jeden, dane stają się niespójne.

2. **Redundancja (powtarzanie):** Dane o produkcie (nazwa, cena, kategoria) powtarzają się przy każdym jego zamówieniu.
3. **Wstawianie:** Nie mogę dodać nowego produktu do bazy, dopóki nie zostanie on zamówiony (bo `ID_Zamowienia` jest częścią klucza i nie może być NULL).
4. **Usuwanie:** Jeśli usunę ostatnie zamówienie zawierające produkt 5 (np. zamówienie 101), **tracę informację o tym produkcie** (jego nazwę, cenę) z bazy.



ROZWIĄZANIE PO NORMALIZACJI DO 2NF:

Dzielimy tabelę na dwie, usuwając zależności częściowe. Atrybuty zależne tylko od `ID_Produktu` przenosimy do nowej tabeli.

Tabela 1: Pozycje_Zamowienia (przechowuje fakty o konkretnym zamówieniu)

ID_Zamowienia	ID_Produku	Ilosc
100	5	2
100	8	1
101	5	1
102	8	3

- Klucz główny: (`ID_Zamowienia`, `ID_Produku`)
- Jedyna pozakluczowa kolumna `Ilość` zależy teraz od **całego klucza**.

Tabela 2: Produkty (przechowuje stałe dane katalogowe produktów)

ID_Produku	Nazwa_Produku	Cena_Produku	Kategoria
5	Laptop Gaming	4500 zł	Elektronika

- Klucz główny: ID_Produktu
- Wszystkie kolumny zależą od całego tego klucza.

Jak teraz wyglądają problemy?

1. **Aktualizacja:** Aby zmienić nazwę produktu, aktualizuję **jeden wiersz** w tabeli **Produkty**.
2. **Redundancja:** Dane o produkcie są przechowywane tylko raz.
3. **Wstawianie:** Mogę dodać nowy produkt do tabeli **Produkty**, nawet jeśli nikt go jeszcze nie zamówił.
4. **Usuwanie:** Usunięcie zamówienia z tabeli **Pozycje_Zamówienia** **nie usuwa informacji o produkcie** z katalogu.



Podsumowanie 2NF w praktyce:

Jeśli Twoja tabela ma **klucz pojedynczy** (np. ID_Zamówienia), to automatycznie spełnia 2NF (nie ma od czego być "częściowo zależna").

2NF ma sens głównie dla tabel z kluczami złożonymi, które przechowują dane z różnych "poziomów" lub encji (jak pozycja zamówienia + dane produktu).



Trzecia postać normalna



Tabela jest w trzeciej postaci normalnej (III PN), jeśli jest w pierwszej i w drugiej postaci normalnej oraz każde z pól niewchodzących w skład klucza podstawowego niesie informację bezpośrednio o kluczu i nie odnosi się do żadnego innego pola.

Przykład przed i po normalizacji:



Przed normalizacją:

Tabela Zamówienia:

ID_Zamowienia	Klient	Produkt	Cena_Produktu	Kategoria_Produktu
1	Kowal	Laptop	3000	Elektronika
2	Nowak	Laptop	3000	Elektronika

Problem: Powtarzające się dane o produkcie



Po normalizacji (3NF):

Tabela Zamówienia:

ID_Zamowienia	ID_Klienta	ID_Produktu
1	101	500
2	102	500

Tabela Klienci:

ID_Klienta	Nazwisko
101	Kowalski
102	Nowak

Tabela Produkty:

ID_Produktu	Nazwa	Cena	ID_Kategorii
-------------	-------	------	--------------

500

Laptop

3000

10

Tabela Kategorie:

ID_Kategorii	Nazwa
10	Elektronika



Zalety normalizacji:

- Mniejsza redundancja danych
- Lepsza integralność danych
- Łatwiejsze utrzymanie i modyfikacje
- Spójność danych



Wady:

- Złożone zapytania z wieloma JOIN
- Możliwy spadek wydajności przy bardzo rozdrobnionych tabelach
- Czasami stosuje się **denormalizację** celowo dla poprawy wydajności

Lekcja

Temat: Projektowanie bazy danych. Wybór struktur i określenie zasad dostępu do bazy danych



Od wersji MySQL 5.5 (wydanej w 2010) domyślnym i rekomendowanym silnikiem jest InnoDB. W 99% przypadków to właściwy wybór.



Zalety:

1. **Transakcje ACID** - gwarancja spójności danych
2. **Klucze obce** - integralność relacyjna
3. **Row-level locking** - lepsza współbieżność (**blokowanie tylko konkretne wiersze (rekordy)** w tabeli, a nie całą tabelę). **Współbieżność (concurrency)** to zdolność systemu do obsługi wielu użytkowników jednocześnie.

Im mniej danych jest blokowanych, tym: więcej zapytań może działać równolegle. Mniej operacji czeka na zwolnienie blokady system działa szybciej pod obciążeniem

4. **Crash recovery** - odtwarza się po awarii. Odtwarza zatwierdzone transakcje i cofa niezakończone (Jeśli jakaś transakcja: była rozpoczęta, ale NIE miała COMMIT,i nastąpił crash,to InnoDB ją cofa), przywracając bazę do spójnego stanu.
5. **MVCC (Multi-Version Concurrency Control)** - jednocześnie odczyty i zapisy. Zamiast blokować dane, baza przechowuje wiele wersji tego samego wiersza.
6. **Optymalizacje dla nowoczesnego hardware** (SSD, duża pamięć RAM)
7. Kompresja danych
8. Obsługa dużych obiektów (BLOB/TEXT)



```
CREATE TABLE users (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100),
    email VARCHAR(100) UNIQUE
) ENGINE=InnoDB;
```



Wady:

1. Większy rozmiar plików niż MyISAM
2. Brak FULLTEXT indeksów (ale od MySQL 5.6+ są!)
3. Wolniejsze COUNT(*) bez WHERE



2. MyISAM - STARSZY, CORAZ RZADZIEJ UŻYWANY



```
CREATE TABLE logs (
    id INT PRIMARY KEY,
    log_date DATE,
    message TEXT,
    FULLTEXT(message)
) ENGINE=MyISAM
```



Zalety:

- Szybsze odczyty dla tabel tylko do odczytu
- Mniejszy rozmiar dysku
- Pełnotekstowe indeksy (w starszych wersjach)
- Dobry do tabel tymczasowych



Wady:

- **Brak transakcji**
- **Table-level locking** (blokada całej tabeli przy zapisie)
- Brak kluczy obcych
- Podatny na uszkodzenie przy awarii
- Brak crash recovery

Użyj gdy: Tabele tylko do odczytu (archiwa), logi (gdy nie potrzebujesz transakcji), w systemach reportingowych.



3. MEMORY (HEAP) - TABELA W PAMIĘCI RAM



```
CREATE TABLE session_data (
    session_id VARCHAR(32) PRIMARY KEY,
    user_id INT,
    data TEXT,
    created TIMESTAMP
) ENGINE=MEMORY;
```



Zalety:

- **Ekstremalnie szybki** dostęp
- Niski overhead
- Hash indeksy domyślnie



Wady:

- **Dane tracone po restarcie MySQL**
- Obsługuje tylko typy o stałej długości
- Ograniczony rozmiar (zależy od max_heap_table_size)
- Table-level locking

Użyj gdy: Cache, dane sesji, tabele tymczasowe, szybkie lookup tables.



4. ARCHIVE - KOMPRESJA DANYCH

```
CREATE TABLE audit_logs (
    id INT AUTO_INCREMENT PRIMARY KEY,
    action VARCHAR(50),
    details TEXT,
    created DATETIME
) ENGINE=ARCHIVE;
```



Zalety:

- **Bardzo dobra kompresja** (do 90%)
- Mały rozmiar na dysku
- Szybkie INSERT (kompresja on-the-fly)



Wady:

- **Tylko INSERT i SELECT** (brak UPDATE/DELETE)
- Brak indeksów (poza PRIMARY KEY)
- Brak transakcji

Użyj gdy: Archiwizacja danych (logi, historyczne rekordy), backup historyczny.

5. CSV - DANE W PLIKU CSV

```
CREATE TABLE import_data (
    id INT,
    name VARCHAR(100),
    value DECIMAL(10,2)
) ENGINE=CSV;
```

Zalety:

- Prosty import/eksport do Excel
- Czytelny format
- Łatwa integracja z innymi narzędziami

Wady:

- Brak indeksów
- Brak typów danych BLOB/TEXT
- Wszystko jako tekst

Użyj gdy: Import/export danych, integracja z zewnętrznymi systemami.

6. BLACKHOLE - DANE ZNIKAJĄ

```
CREATE TABLE replication_filter (
    id INT,
    data VARCHAR(255)
) ENGINE=BLACKHOLE;
```

Zalety:

- **Dane "znikają" po INSERT**
- Niski overhead
- Przydatny do replikacji

Wady:

- Brak przechowywania danych
- Tylko struktura tabeli

Użyj gdy: Filtrowanie replikacji, testowanie wydajności, logowanie bez zapisu.

7. FEDERATED - DANE Z ZEWNĘTRZNEJ BAZY

```
CREATE TABLE remote_table (
    id INT PRIMARY KEY,
    name VARCHAR(100)
) ENGINE=FEDERATED
CONNECTION='mysql://user:pass@remote_host:3306/db/table';
```

Zalety:

- Dostęp do zdalnych danych jak do lokalnych
- Transparentność lokalizacji

Wady:

- Wolniejsze (dostęp przez sieć)
- Problemy z łącznością
- Ograniczone wsparcie

Użyj gdy: Integracja rozproszonych baz, agregacja danych.

TABELA PORÓWNAWCZA

Silnik	Transakcje	Klucze obce	Locking	MVCC	Crash Safe	Pamięć	Użycie
InnoDB	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Row-level	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Duża	Domyślny
MyISAM	<input type="checkbox"/>	<input type="checkbox"/>	Table-level	<input type="checkbox"/>	<input type="checkbox"/>	Mała	Tylko odczyt

MEMORY	✗	✗	Table-level	✗	✗	RAM	Cache
ARCHIVE	✗	✗	Row-level	✗	✗	Bardzo mała	Archiwa
CSV	✗	✗	Table-level	✗	✗	Średnia	Import/Export
BLACKHOLE	✗	✗	Table-level	✗	<input checked="" type="checkbox"/>	Brak	Replikacja

PRAKTYCZNE WSKAZÓWKI WYBORU

Wybierz InnoDB gdy:

- Tworzysz nową aplikację
- Potrzebujesz transakcji
- Masz relacje między tabelami
- Chcesz bezpieczeństwa danych
- Masz równoczesnych wielu użytkowników

Wybierz MyISAM gdy:

- Masz tabelę tylko do odczytu
- Masz stare aplikacje, które go wymagają
- Potrzebujesz FULLTEXT w MySQL < 5.6
- Masz ograniczony dysk



RODZAJE INDEKSÓW W MYSQL

Podstawowe typy:

```
-- 1. PRIMARY KEY (domyślnie clustered index w InnoDB)
CREATE TABLE users (
    id INT PRIMARY KEY AUTO_INCREMENT, -- Automatycznie indeks
    username VARCHAR(50)
);

-- 2. UNIQUE INDEX (gwarantuje unikalność)
CREATE UNIQUE INDEX idx_email ON users(email);

-- 3. INDEX (zwykły indeks)
CREATE INDEX idx_last_name ON customers(last_name);

-- 4. FULLTEXT (wyszukiwanie pełnotekstowe)
CREATE FULLTEXT INDEX idx_content ON articles(body);

-- 5. SPATIAL (dla danych geograficznych)
CREATE SPATIAL INDEX idx_location ON places(coordinates);
```



Struktury indeksów:

- **B-Tree** (domyślnie) - zakresy, sortowanie, przedrostki
- **Hash** (tylko MEMORY/NDB) - tylko równości (=)
- **R-Tree (SPATIAL)** - dane przestrzenne



STRATEGIE TWORZENIA INDEKSÓW

Kiedy tworzyć indeksy?

Twórz indeksy dla:

```
-- WHERE conditions
CREATE INDEX idx_status ON orders(status);
-- SELECT * FROM orders WHERE status = 'shipped'

-- JOIN columns
CREATE INDEX idx_customer_id ON orders(customer_id);
-- SELECT * FROM customers JOIN orders ON customers.id = orders.customer_id

-- ORDER BY / GROUP BY
CREATE INDEX idx_order_date ON orders(order_date);
-- SELECT * FROM orders ORDER BY order_date DESC

-- Combination
CREATE INDEX idx_status_date ON orders(status, order_date);
-- SELECT * FROM orders WHERE status = 'pending' ORDER BY order_date
```

Kiedy NIE tworzyć indeksów?

- Tabele małe (< 1000 wierszy)
- Kolumny często modyfikowane (INSERT/UPDATE/DELETE)
- Kolumny z niską selektywnością (np. "gender" z wartościami M/F)
- Kolumny typu BLOB/TEXT (chyba że prefiksy)



3. INDEKSY ZŁOŻONE (COMPOSITE) - NAJWAŻNIEJSZE!



Zasada kolejności kolumn:

```
-- DOBRZE: (last_name, first_name)
CREATE INDEX idx_name ON employees(last_name, first_name);

-- Zapytania wykorzystujące indeks:
SELECT * FROM employees WHERE last_name = 'Kowalski';
SELECT * FROM employees WHERE last_name = 'Kowalski' AND first_name = 'Jan';
SELECT * FROM employees WHERE last_name LIKE 'Kow%';

-- NIE wykorzysta indeksu dla:
SELECT * FROM employees WHERE first_name = 'Jan'; -- Brak Last_name!

-- RÓWNIEŻ DOBRE: (status, order_date, customer_id)
CREATE INDEX idx_order_filter ON orders(status, order_date, customer_id);
```



Kardynalność - kolejność kolumn w indeksie złożonym:

```
-- ZŁE: (gender, last_name) - gender ma niską kardynalność
CREATE INDEX idx_gender_name ON users(gender, last_name);

-- DOBRE: (last_name, gender) - last_name ma wysoką kardynalność
CREATE INDEX idx_name_gender ON users(last_name, gender);

-- REGUŁA: Kolumny z wyższą kardynalnością (więcej unikalnych wartości) NA PRZODZIE
```



4. INDEKSY POKRYWAJĄCE (COVERING INDEXES)

Indeks zawiera WSZYSTKIE kolumny potrzebne w zapytaniu:

```
-- Tabela: orders(id, customer_id, amount, status, order_date)

-- ZAPYTANIE 1: Potrzebuje skanowania tabeli
SELECT customer_id, amount, status
FROM orders
WHERE order_date > '2024-01-01';

-- ROZWIĄZANIE: Covering index
CREATE INDEX idx_covering ON orders(order_date, customer_id, amount, status);
-- Teraz MySQL pobierze dane TYLKO z indeksu (fast!)

-- ZAPYTANIE 2:
SELECT COUNT(*) FROM orders WHERE status = 'completed';
-- Covering index:
CREATE INDEX idx_status_covering ON orders(status);
```



5. ANALIZA WYKORZYSTANIA INDEKSÓW



EXPLAIN - najważniejsze narzędzie:

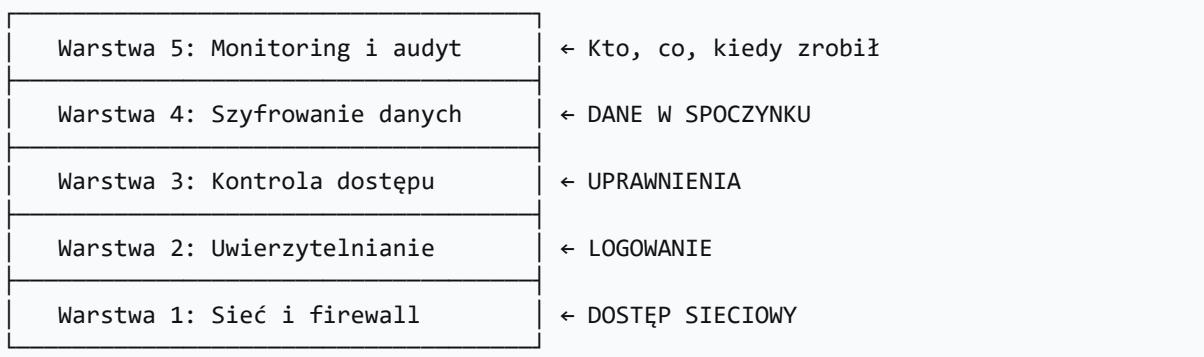
```
EXPLAIN FORMAT=JSON
SELECT o.*, c.name
FROM orders o
JOIN customers c ON o.customer_id = c.id
WHERE o.status = 'shipped'
AND o.order_date > '2024-01-01'
ORDER BY o.order_date DESC;

-- Kluczowe pola w wyniku EXPLAIN:
-- type: ALL (full scan) / index / range / ref / eq_ref / const
-- key: który indeks został użyty
```

```
-- rows: szacowana liczba przeszukiwanych wierszy
-- Extra: Using index (covering), Using filesort (problem!), Using temporary
```



Model warstwowy bezpieczeństwa:



Tworzenie ról (MySQL 8.0+):

```
-- 1. Definiowanie ról
CREATE ROLE 'app_READONLY';
CREATE ROLE 'app_EDITOR';
CREATE ROLE 'app_ADMIN';
CREATE ROLE 'reporting_USER';

-- 2. Przypisywanie uprawnień do ról
GRANT SELECT ON ecommerce.* TO 'app_READONLY';
GRANT SELECT, INSERT, UPDATE ON ecommerce.orders TO 'app_EDITOR';
GRANT SELECT, INSERT, UPDATE ON ecommerce.customers TO 'app_EDITOR';
GRANT ALL PRIVILEGES ON ecommerce.* TO 'app_ADMIN';
GRANT SELECT ON ecommerce.sales_view TO 'reporting_USER';

-- 3. Tworzenie użytkowników
CREATE USER 'api_user'@'10.0.%' IDENTIFIED BY 'StrongPass123!';
CREATE USER 'analyst_john'@'localhost' IDENTIFIED WITH caching_sha2_password BY
'AnalystPass!2024';
CREATE USER 'backup_user'@'localhost' IDENTIFIED BY 'BackupPass456$';
```

```
-- 4. Przypisywanie ról użytkownikom
GRANT 'app_editor' TO 'api_user'@'10.0.%';
GRANT 'reporting_user' TO 'analyst_john'@'localhost';
GRANT SELECT, RELOAD, LOCK TABLES, REPLICATION CLIENT ON *.* TO
'backup_user'@'localhost';

-- 5. Aktywacja ról (MySQL wymaga jawniej aktywacji)
SET DEFAULT ROLE ALL TO 'api_user'@'10.0.%';
```

To **kopia zapasowa** Twoich danych. Jak zdjęcie całej bazy w danym momencie.

Bo błędy się zdarzają:

- Ktoś usunie ważne dane
- Zepsuje się dysk serwera
- Atak hakerski (ransomware)
- Pożar/powódź w serwerowni

Bez backupu = TRACISZ DANE NA ZAWSZE

ZASADA 3-2-1-1-0 (NAJPROŚCIEJ)

Zapamiętaj te liczby:

3 - Trzy kopie tego samego

Przykład:

1. Na serwerze produkcyjnym
2. Na dysku lokalnym
3. W chmurze/innej lokalizacji

2 - Dwa różne nośniki

Przykład:

- Dysk twardy w serwerze
- Zewnętrzny dysk/taśma LTO

1 - Jedna kopia poza firmą

Backup w innym mieście/budynku.

Gdy spłonie serwerownia, backup jest bezpieczny.

1 - Jedna kopia OFFLINE

Dysk **niepodłączony do sieci**.

Ochrona przed ransomware - haker nie zaszyfruje odłączonego dysku!

0 - Zero błędów

Backup musi być sprawdzony i działający.

JAK CZĘSTO ROBIĆ BACKUP?

Dla małej firmy/serwera:

Codziennie 2:00 w nocy:
(cała baza) ← PEŁNY backup

Co 4 godziny:
(tylko zmiany) ← PRZYROSTOWY

Dla większego systemu:

Niedziela 2:00 → PEŁNY backup (całość)

Pon-Pt 1:00 → PRZYROSTOWY (tylko zmiany)

Co 15 minut → LOGI zmian (najmniejsze straty)

3 RODZAJE BACKUPÓW:

1. PEŁNY (FULL)

- **Co to?** Cała baza
- **Kiedy?** Raz w tygodniu
- **Plusy:** łatwe przywracanie
- **Minusy:** Duże miejsca, długie trwa

2. PRZYROSTOWY (INCREMENTAL)

- **Co to?** Tylko to, co się zmieniło od ostatniego backupu
- **Kiedy?** Codziennie
- **Plusy:** Mało miejsca, szybko
- **Minusy:** Przywracanie wolniejsze (trzeba połączyć kilka backupów)

3. LOGI (BINLOG)

- **Co to?** Zapis każdej zmiany w bazie (jak rejestrator)
- **Kiedy?** Co 15-60 minut
- **Plusy:** Przywróciś do dowolnej minuty!
- **Minusy:** Trzeba łączyć z backupem pełnym

PROSTA ANALOGIA:

Wyobraź sobie bibliotekę:

1. **Pełny backup** = Zrobienie ksero całej biblioteki (raz w tygodniu)
2. **Przyrostowy** = Ksero tylko nowych książek (codziennie)
3. **Logi** = Zapis kto i kiedy wypożyczył (co godzinę)

Co się stanie, gdy pożar zniszczy bibliotekę?

1. Odtwarzasz z ksero całej biblioteki (pełny backup)
2. Dodajesz nowe książki (przyrostowe)
3. Sprawdzasz wypożyczenia (logi)

KROK 1: Określ co backupować

- Tylko baza danych? ✓
- Konfiguracja serwera? ✓
- Pliki aplikacji? ✓
- Logi systemowe? (opcjonalnie)

KROK 2: Ustal częstotliwość

- **RPO** (Recovery Point Objective) = Ile danych możesz stracić?
 - o Bank: **0 minut** (każda transakcja ważna)
 - o Blog: **24 godziny** (można stracić komentarze z dnia)

KROK 3: Wybierz gdzie przechowywać

MUST HAVE:

- Dysk lokalny (szybki dostęp)
- Zewnętrzny dysk (offline)
- Chmura (offsite)

NIGDY NIE:

- Tylko jeden dysk
- Tylko w tym samym budynku
- Tylko online

KROK 4: Automatyzuj

```
# Przykładowy skrypt
#!/bin/bash
# Codziennie 2:00 wykonaj backup
mysqldump -u root -pHaslo123 baza_firmy > /backup/baza_$(date +%Y%m%d).sql
# Wyślij na zewnętrzny dysk
cp /backup/baza_*.sql /mnt/external_drive/
# Wyślij do chmury
rclone copy /backup/ backup_cloud:
```

KROK 5: TESTUJ!

Najważniejsza zasada:

Backup bez testu = brak backupu