📚 Pytania z Baz Danych - Przewodnik na Egzamin

1️⃣ Integralność bazy danych - jakie znasz warunki integralności? 🛡️

Integralność bazy danych = zestaw "strażników" 👮 pilnujących, żeby dane w bazie były poprawne, kompletne i spójne.

🎯 Główne rodzaje więzów integralności:

🔢 1. Integralność domenowa (co może być w komórce):

- **Typ danych** - wartość musi być odpowiedniego typu (liczba 📊 , tekst 📝)
- **Zakres wartości** - np. wiek 0-150, płeć M/K
- **NOT NULL** - brak pustych wartości tam gdzie nie wolno ❌

🆔 2. Integralność encji (unikalność):

- **Klucz główny** - unikalny identyfikator każdego rekordu 🗝️
- **Unikalność** - nie ma duplikatów ❌🔄

🔗 3. Integralność referencyjna (powiązania):

- **Klucz obcy** - wskazuje na istniejący rekord w innej tabeli 🔄
- **Brak "wiszących" odniesień** - nie można wskazywać na coś co nie istnieje 👻

⚖️ 4. Integralność warunkowa (reguły biznesowe):

- **Reguły biznesowe** - np. data_końca > data_początku 📅
- **Złożone warunki** między kolumnami 🔀

> 💡 **Zapamiętaj:** Integralność = "żeby w bazie nie było bałaganu!" 🧹

2️⃣ Co to są transakcje i jakie mają własności? 💸

Transakcja = paczka operacji SQL 📦 działających w systemie "wszystko albo nic"!

Wyobraź sobie przelew bankowy 🏦 : musisz odjąć pieniądze z jednego konta I dodać do drugiego. Gdyby tylko jedno się udało = katastrofa! 💥

🔤 Właściwości ACID (musisz znać na pamięć!):

🧬 A - Atomowość (Atomicity):

- **"Wszystko albo nic"** - transakcja jest jak atom (niepodzielna) ⚛️
- Jeśli jedna operacja nie wyjdzie → cofamy WSZYSTKO 🔄

⚖️ C - Spójność (Consistency):

- Baza przechodzi ze stanu "OK" 😊 do stanu "OK" 😊
- Wszystkie reguły integralności są zachowane ✅

🏝️ **I - Izolacja (Isolation):**

- Każda transakcja działa jakby była SAMA na wyspie 🏝️
- Nie widzą się nawzajem podczas pracy 👀❌

💾 **D - Trwałość (Durability):**

- Jak już zatwierdzisz → dane są na zawsze! 💎
- Nawet jak prąd wyłączą, dane zostają 🔌❌

> 🎯 **Mnemotechnika: A**dam **C**zeka **I**nteligentne **D**ziecko (ACID)

3️⃣ Jakie uprawnienia może mieć użytkownik i jak je zmieniać? 👤🔐

Uprawnienia = kontrola dostępu 🚪🔑 - kto, co i gdzie może robić w bazie!

🏢 **Poziomy uprawnień** (jak piętra w budynku):

1. 🌍 **Globalne** - władza nad całym serwerem (admin)
2. 🏠 **Baza danych** - władza nad jedną bazą
3. 🪑 **Tabela** - władza nad jedną tabelą
4. 📝 **Kolumna** - władza nad konkretnymi kolumnami
5. ⚙️ **Procedury/funkcje** - może wywołać procedury

🎭 **Rodzaje uprawnień** (co może robić):

- 👀 **SELECT** - może czytać dane
- ➕ **INSERT** - może dodawać nowe rekordy
- ✏️ **UPDATE** - może zmieniać istniejące dane
- 🗑️ **DELETE** - może usuwać rekordy
- 🏗️ **CREATE** - może tworzyć nowe obiekty
- 💥 **DROP** - może usuwać obiekty (NIEBEZPIECZNE!)

💼 **Zarządzanie uprawnień:**

```
-- 🎁 Nadawanie uprawnień (prezent!)
GRANT SELECT, INSERT ON tabela TO 'uzytkownik';

-- 🚫 Odbieranie uprawnień (zabieranie prezentu)
REVOKE DELETE ON tabela FROM 'uzytkownik';
```

> ⚠️ **UWAGA:** Nigdy nie dawaj DROP bez przemyślenia! 💀

4️⃣ Integralność referencyjna i ON DELETE CASCADE 🔗💥

Integralność referencyjna = klucz obcy zawsze wskazuje na coś co istnieje! 🎯

Wyobraź sobie bibliotekę 📚 : nie może być wypożyczonej książki od nieistniejącego czytelnika!

⚠ **Problem:** Co gdy chcesz usunąć "rodzica"?

Sytuacja: Chcesz usunąć KLIENTA, ale ma ZAMÓWIENIA 🛒

- Klucz obcy w ZAMÓWIENIA wskazuje na KLIENTA
- Co zrobić z "osieroconymi" zamówieniami? 👻

🛡️ **Rozwiązania referencyjne:**

🚫 **RESTRICT (domyślne)**

- **Co robi:** "NIE POZWOLĘ!" - blokuje usunięcie
- **Kiedy:** Bezpieczne, ale czasem uciążliwe

```
-- Nie usuną klienta jeśli ma zamówienia
FOREIGN KEY (klient_id) REFERENCES klienci(id) ON DELETE RESTRICT
```

💥 **CASCADE (kaskadowe)**

- **Co robi:** "Usuwam wszystko!" - usuwa też wszystkie powiązane
- **Kiedy:** Wygodne, ale NIEBEZPIECZNE! ⚠

```
-- Usuń klienta = usuń też jego zamówienia
FOREIGN KEY (klient_id) REFERENCES klienci(id) ON DELETE CASCADE
```

🔄 **SET NULL**

- **Co robi:** Ustaw klucz obcy na NULL
- **Kiedy:** Gdy dane mogą "zostać osierocone"

```
-- Usuń kierownika = ustaw manager_id na NULL
FOREIGN KEY (manager_id) REFERENCES managers(id) ON DELETE SET NULL
```

🔧 **SET DEFAULT**

- **Co robi:** Ustaw domyślną wartość
- **Kiedy:** Gdy masz "domyślnego rodzica"

> 💀 **UWAGA:** CASCADE może wywołać "efekt domina" - usuń jeden rekord, stracisz tysiące! Używaj ostrożnie!

5️⃣ Triggery (wyzwalacze) - co to i jak działają? ⚡🎯

Trigger = "automatyczny strażnik" 👮 który reaguje na zmiany w bazie danych!

Wyobraź sobie alarm w domu 🏠🔔 : jak ktoś otworzy drzwi → alarm się włącza automatycznie!

🎯 Do czego służą triggery:

🛡️ Kontrola integralności

- Sprawdzanie skomplikowanych reguł biznesowych
- "Pensja nie może być ujemna!" ❌💰

🤖 Automatyzacja

- Automatyczne timestampy "kto kiedy co zmienił"
- Automatyczne liczenie sum, średnich 📊

📝 Auditing/Logowanie

- Zapisywanie historii zmian
- "Kto usunął tego klienta?!" 🕵️

⏰ Kiedy się uruchamiają:

🔴 BEFORE (przed operacją)

- **Może zatrzymać** operację jeśli coś nie gra ✋
- **Może zmienić** dane przed zapisem 🔧

```
-- Zaokrąglij cenę przed zapisem
CREATE TRIGGER zaokraglij_cene
BEFORE INSERT ON produkty
FOR EACH ROW
BEGIN
    SET NEW.cena = ROUND(NEW.cena, 2);
END;
```

🟢 AFTER (po operacji)

- **Tylko logowanie** - nie może już nic zmienić 📋
- Świetne do historii zmian 📚

```
-- Zapisz do loga kto co usunął
CREATE TRIGGER log_usuniecia
AFTER DELETE ON klienci
FOR EACH ROW
BEGIN
```

```
    INSERT INTO log_zmian (akcja, uzytkownik, data)
    VALUES ('DELETE', USER(), NOW());
END;
```

🔥 Na jakie zdarzenia reagują:

- 📩 **INSERT** - dodawanie nowego rekordu
- ✏️ **UPDATE** - zmiana istniejącego rekordu
- 🗑️ **DELETE** - usuwanie rekordu

⚠️ UWAGI:

- **Niewidoczne** - programista może zapomnieć że istnieją! 👻
- **Trudne do debugowania** - ukryte problemy 🐛
- **Mogą się wywołać kaskadowo** - trigger wywołuje trigger! 🔄

> 💡 **Zapamiętaj:** BEFORE = może zmienić, AFTER = tylko loguje! 🤷‍♂️

6️⃣ Kiedy powstaje konflikt między transakcjami? ⚔️💥

Konflikt transakcji = gdy dwie transakcje "wchodzą sobie w drogę" przy tym samym rekordzie! 🚧

⚔️ Rodzaje konfliktów:

1️⃣ 📖➡️✏️ Read-Write Conflict

Scenariusz: Jedna czyta, druga pisze w to samo miejsce

```
👤 Transakcja A: czyta saldo = 1000zł
👨 Transakcja B: zmienia saldo na 500zł  ← KONFLIKT!
👤 Transakcja A: dalej myśli że ma 1000zł...
```

Problem: A działa na nieaktualnych danych! 😵

2️⃣ ✏️➡️✏️ Write-Write Conflict

Scenariusz: Obie chcą pisać w to samo miejsce

```
👤 Transakcja A: chce ustawić saldo = 800zł
👨 Transakcja B: chce ustawić saldo = 600zł  ← WOJNA!
```

Problem: Której zmiana zostanie? 🤷‍♀️

3️⃣ 📖➡️📖 Read-Read (NIE KONFLIKT!)

Scenariusz: Obie tylko czytają

```
👤 Transakcja A: czyta saldo = 1000zł
👨 Transakcja B: czyta saldo = 1000zł  ← OK! 👌
```

Wynik: Bez problemu - czytanie nie szkodzi! ✅

4️⃣ 🌀 Write Skew (zaawansowany)

Scenariusz: Czytają to samo, ale piszą w różne miejsca

```
👥 Mamy 2 lekarzy na dyżurze (min. 1 musi zostać)
👤 Lekarz A: widzi 2 lekarzy → prosi o urlop
👨 Lekarz B: widzi 2 lekarzy → prosi o urlop
💥 Wynik: 0 lekarzy na dyżurze! (złamana reguła)
```

🛡️ Jak unikać konfliktów:

- 🔒 **Blokady** - "zajęte, nie dotykaj!"
- ⏰ **Kolejność operacji** - kto pierwszy, ten lepszy
- 🏝️ **Izolacja** - każdy na swojej wyspie

> 💡 **Zapamiętaj:** Read-Read = OK ✅ , reszta = problemy! ⚠️

7️⃣ Relacja 1:1 (jeden do jednego) 👤↔👤

Relacja 1:1 = każdy rekord w jednej tabeli ma dokładnie JEDEN odpowiednik w drugiej tabeli (i odwrotnie)!

🎯 Przykłady z życia:

- **Osoba ↔ Paszport** 🆔 (jedna osoba = jeden paszport)
- **Pracownik ↔ Miejsce parkingowe** 🚗 (jeden pracownik = jedno miejsce)
- **Student ↔ Legitymacja** 🎓 (jeden student = jedna legitymacja)

🛠️ Jak zrobić relację 1:1:

💡 Opcja 1: Wspólny klucz główny

```sql
CREATE TABLE osoby (
    id INT PRIMARY KEY,
    imie VARCHAR(50),
    nazwisko VARCHAR(50)
);

CREATE TABLE paszporty (
    osoba_id INT PRIMARY KEY,  -- Ten sam ID co w osoby!
    numer_paszportu VARCHAR(20),
    data_waznosci DATE,
```

```
    FOREIGN KEY (osoba_id) REFERENCES osoby(id)
);
```

💡 **Opcja 2: Unikalny klucz obcy**

```
CREATE TABLE pracownicy (
    id INT PRIMARY KEY,
    imie VARCHAR(50)
);

CREATE TABLE miejsca_parkingowe (
    id INT PRIMARY KEY,
    numer_miejsca VARCHAR(10),
    pracownik_id INT UNIQUE,  -- UNIQUE = tylko jeden!
    FOREIGN KEY (pracownik_id) REFERENCES pracownicy(id)
);
```

🤔 Kiedy używać relacji 1:1:

- **Podział danych** ze względów bezpieczeństwa 🔒
- **Duże tabele** - podział na część podstawową i szczegółową 📊
- **Opcjonalne dane** - nie każdy ma te dodatkowe informacje ❓

⚠️ Uwagi:

- **Rzadko używane** - często można połączyć w jedną tabelę 📝
- **Klucz obcy może być NULL** (ale główny NIE!) ❌
- **Pomyśl dwukrotnie** - czy naprawdę potrzebujesz dwóch tabel? 🤔

> 💡 **Zapamiętaj:** 1:1 = "każdy ma swojego jedynego partnera!" 💕

8️⃣ Podzapytania (zagnieżdżone) - rodzaje i zastosowania 🪆 📦

Podzapytanie = zapytanie wewnątrz zapytania 🪆 - jak matrioszka!

🎯 Rodzaje podzapytań (wg tego co zwracają):

1️⃣ 🔢 Scalar Subquery - jedna wartość

Zwraca: Pojedynczą wartość (liczba, tekst, data)

```
-- Kto zarabia więcej niż średnia?
SELECT imie FROM pracownicy
WHERE pensja > (SELECT AVG(pensja) FROM pracownicy);
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                    Zwraca jedną liczbę!
```

2️⃣ 📋 Row Subquery - jeden wiersz

Zwraca: Jeden rekord (może mieć wiele kolumn)

```
-- Znajdź osobę o tym samym imieniu i nazwisku
SELECT * FROM osoby
WHERE (imie, nazwisko) = (SELECT imie, nazwisko FROM vip WHERE id = 1);
                         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                                  Zwraca jeden wiersz!
```

3️⃣ 📊 Table Subquery - wiele wierszy

Zwraca: Zestaw rekordów (jak mini-tabela)

```
-- Kto pracuje w działach IT?
SELECT imie FROM pracownicy
WHERE dzial_id IN (SELECT id FROM dzialy WHERE nazwa LIKE '%IT%');
                  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                           Zwraca wiele id działów!
```

🏗️ Gdzie używać podzapytania:

- **WHERE** - do filtrowania 🔍
- **FROM** - jako "tymczasowa tabela" 📋
- **SELECT** - do wyliczania wartości 🧮
- **HAVING** - do filtrowania grup 👥

⚠️ Pułapki podzapytań:

- **Wydajność** - mogą być wolne 🐌
- **NULL** - uwaga na podzapytania z NULL! 👻
- **Korelacja** - podzapytanie może odwoływać się do głównego zapytania 🔄

> 💡 **Rada:** Podzapytanie = "najpierw znajdź to, potem użyj tego!" 🎯

9️⃣ Współbieżność - problemy i rozwiązania 🏃‍♂️🏃‍♀️💥

Współbieżność = kilka transakcji działa jednocześnie 🎭🎪 = może być chaos!

😱 Główne problemy współbieżności:

1️⃣ 💸 Lost Update (Utracona aktualizacja)

Scenariusz: Dwoje ludzi jednocześnie wypłaca z konta!

```
👤 Kasia: czyta saldo 1000zł → wypłaci 200zł
👨 Tomek: czyta saldo 1000zł → wypłaci 300zł
👤 Kasia: zapisuje 800zł (1000-200)
👨 Tomek: zapisuje 700zł (1000-300) ← NADPISAŁ KASIĘ!
```

Wynik: Przepadła wypłata Kasi! 💀

2️⃣ 💩 Dirty Read (Brudny odczyt)

Scenariusz: Czytasz nieukończone zmiany!

```
👤 Transakcja A: zmienia saldo na 500zł (ale nie zatwierdza)
👨 Transakcja B: czyta 500zł ← CZYTA "BRUDNE" DANE!
👤 Transakcja A: ROLLBACK! (wraca do 1000zł)
👨 Transakcja B: działała na błędnych danych!
```

3️⃣ 🔄 Non-Repeatable Read (Niepowtarzalny odczyt)

Scenariusz: To samo zapytanie daje różne wyniki!

```
👤 Transakcja A: czyta saldo = 1000zł
👨 Transakcja B: zmienia saldo na 500zł i zatwierdza
👤 Transakcja A: znów czyta saldo = 500zł ← INNE!
```

🛡️ Rozwiązania:

- 🔒 **Blokady (Locks)** - "zajęte, nie dotykaj!"
- 🏝️ **Poziomy izolacji** - różne stopnie odizolowania
- ⏰ **Znaczniki czasu** - kto pierwszy ten lepszy
- 📚 **MVCC** - każdy widzi swoją "wersję" danych

> 🎯 **Zapamiętaj:** Współbieżność = szybkość vs bezpieczeństwo! ⚖️

1️⃣0️⃣ Operacje na relacjach + Funkcje agregujące 📊🧱

🛠️ Podstawowe operacje SQL:

- 👀 **SELECT** - czytanie danych
- ➕ **INSERT** - dodawanie rekordów
- ✏️ **UPDATE** - modyfikacja danych
- 🗑️ **DELETE** - usuwanie rekordów
- 🏗️ **CREATE/ALTER/DROP** - zarządzanie strukturą

🧱 Funkcje agregujące (ważne na egzaminie!):

📊 Podstawowe funkcje:

```
SELECT
    COUNT(*) as ilosc_wszystkich,     -- Liczba wierszy
    COUNT(telefon) as z_telefonem,    -- Pomija NULL!
    SUM(pensja) as suma_pensji,       -- Suma wartości
    AVG(pensja) as srednia_pensja,    -- Średnia
    MIN(pensja) as min_pensja,        -- Minimum
    MAX(pensja) as max_pensja         -- Maksimum
FROM pracownicy;
```

👥 GROUP BY - grupowanie danych

Co robi: Dzieli dane na grupy i liczy dla każdej grupy osobno

```
-- Ile osób w każdym dziale i jaka średnia pensja?
SELECT
    dzial,
    COUNT(*) as ilosc_osob,
    AVG(pensja) as srednia_pensja
FROM pracownicy
GROUP BY dzial;
```

🔍 HAVING - filtrowanie grup (nie wierszy!)

```
-- Tylko działy z więcej niż 5 osobami
SELECT dzial, COUNT(*) as ilosc
FROM pracownicy
GROUP BY dzial
HAVING COUNT(*) > 5;  ← HAVING dla grup, WHERE dla wierszy!
```

🔗 JOINy (łączenie tabel):

- 🎯 **INNER JOIN** - tylko pasujące rekordy z obu tabel
- ⬅️ **LEFT JOIN** - wszystko z lewej + pasujące z prawej
- ➡️ **RIGHT JOIN** - wszystko z prawej + pasujące z lewej
- 🔄 **FULL JOIN** - wszystko z obu tabel

> 💡 **Zapamiętaj:** WHERE = filtr wierszy, HAVING = filtr grup! 🎯

1️⃣1️⃣ Twierdzenie Heatha - kiedy można bezpiecznie podzielić tabelę? ✂️📊

Twierdzenie Heatha = przepis na bezpieczne "cięcie" tabeli na kawałki! ✂️🧩

Wyobraź sobie puzzle 🧩 : możesz je rozłożyć na części, ale tylko jeśli wiesz jak je później złożyć!

🎯 Podstawowa idea:

Rozkład odwracalny = podzielenie jednej tabeli na dwie (lub więcej) tak, żeby można było je później **bezbłędnie** połączyć bez utraty informacji! 🔄

🧩 Twierdzenie Heatha - przepis:

JEŚLI istnieje **zależność funkcyjna A → B**, **TO** można bezpiecznie podzielić tabelę R(A,B,C) na:

- **R1(A, B)**
- **R2(A, C)**

💡 Prosty przykład:

📋 Oryginalna tabela PRACOWNICY:

```
ImieNazwisko    | Zespol | Kierownik
----------------|--------|----------
Jan Kowalski    | IT     | Anna Nowak
Piotr Marek     | IT     | Anna Nowak
Maria Witacz    | HR     | Tomasz Bis
```

🔍 Zależność funkcyjna:

`Zespol → Kierownik` (każdy zespół ma **jednego** kierownika)

✂️ Bezpieczny podział:

```sql
-- Tabela 1: Pracownicy z zespołami
CREATE TABLE pracownicy_zespoly (
    imie_nazwisko VARCHAR(100),
    zespol VARCHAR(50)
);

-- Tabela 2: Zespoły z kierownikami
CREATE TABLE zespoly_kierownicy (
    zespol VARCHAR(50) PRIMARY KEY,
    kierownik VARCHAR(100)
);
```

🔄 Odzyskanie oryginalnej tabeli:

```sql
-- JOIN przywraca oryginalną tabelę!
SELECT p.imie_nazwisko, p.zespol, z.kierownik
FROM pracownicy_zespoly p
JOIN zespoly_kierownicy z ON p.zespol = z.zespol;
```

✅ Dlaczego to działa?

Bo znając **zespół**, zawsze wiemy kto jest **kierownikiem**! Żadna informacja się nie gubi! 🎯

❌ Kiedy NIE WOLNO dzielić?

💥 Brak zależności funkcyjnej = katastrofa!

Przykład błędnego podziału:

```
-- Tabela: ZAMOWIENIA(Klient, Produkt, Data)
-- BRAK zależności: Klient ↛ Produkt (klient może kupować różne produkty)

-- ❌ BŁĘDNY podział na:
-- TABELA1(Klient, Produkt)
-- TABELA2(Klient, Data)

-- 💥 Problem: przy JOIN dostaniemy WIĘCEJ rekordów niż oryginał!
```

🎭 Analogia - biblioteka:

- **Książka → Autor** ✅ (jedna książka = jeden autor)
  - Można podzielić: KSIĄŻKI(Tytuł, Autor) + AUTORZY(Autor, Biografia)
- **Czytelnik →/Książka** ❌ (czytelnik może wypożyczyć wiele książek)
  - NIE MOŻNA podzielić bezpiecznie!

📊 Kluczowe pojęcia:

🧬 Zależność funkcyjna A → B:

"Znając A, zawsze wiem B"

- Zespol → Kierownik ✅
- PESEL → Osoba ✅
- Klient → Adres ✅

🔄 Odwracalność (Lossless):

JOIN po podziale = oryginalna tabela (bez zmian!)

💥 Spurious tuples:

"Fałszywe rekordy" - powstają przy błędnym podziale

> 💡 **Zapamiętaj:** Heath = "Cięcie tylko tam gdzie nie boli!" Jeśli A → B, to można ciąć! ✂️🎯

1️⃣2️⃣ SQL Injection - cybernapad na bazę danych! 💉🔐

SQL Injection = wstrzyknięcie złośliwego kodu SQL przez hackera do aplikacji! 🗡️ 💀

Wyobraź sobie rozmowę 💬 : zamiast normalnej odpowiedzi, ktoś przemyci ci szkodliwe instrukcje!

😈 Jak działa atak:

🎯 Schemat ataku:

1. 🤖 **Aplikacja** pyta użytkownika o login/hasło
2. 😈 **Hacker** wpisuje złośliwy kod zamiast danych
3. 💀 **Aplikacja** ślepo przekazuje kod do bazy danych
4. 💥 **Baza** wykonuje szkodliwy kod!

💡 Przykład ataku:

🎯 Formularz logowania:

```
Login: [jan            ]
Hasło: [haslo123       ]
```

😈 Hacker wpisuje:

```
Login: [jan' OR 1=1 -- ]
Hasło: [cokolwiek      ]
```

💀 Powstałe zapytanie SQL:

```sql
-- ✅ Normalne zapytanie:
SELECT * FROM users WHERE login = 'jan' AND password = 'haslo123'

-- 💥 ZAATAKOWANE zapytanie:
SELECT * FROM users WHERE login = 'jan' OR 1=1 -- ' AND password =
'cokolwiek'
```

🎯 **Wynik:** 1=1 jest zawsze prawda → hacker zalogowany jako pierwszy użytkownik! 💀

🔥 Bardziej niszczycielskie ataki:

```sql
-- 💀 Usuń całą tabelę:
Login: jan'; DROP TABLE users; --

-- 🦹 Wyciągnij hasła:
Login: jan' UNION SELECT password FROM users WHERE '1'='1
```

```
-- 🔒 Dodaj admina:
Login: jan'; INSERT INTO users VALUES('hacker', 'admin'); --
```

🛡️ Jak się bronić:

1️⃣ 💎 Prepared Statements (najlepsze!)

```
-- ❌ NIEBEZPIECZNE (podatne na injection):
SELECT * FROM users WHERE login = '$login' AND password = '$password'

-- ✅ BEZPIECZNE (prepared statement):
PREPARE stmt FROM 'SELECT * FROM users WHERE login = ? AND password = ?'
EXECUTE stmt USING @login, @password
```

💡 **Dlaczego działa:** Dane i kod SQL są **całkowicie rozdzielone**! 🎯

2️⃣ 🔍 Walidacja danych

```
# ✅ Sprawdź format danych
if not re.match("^[a-zA-Z0-9_]+$", login):
    return "Błędny format loginu!"

# ✅ Ogranicz długość
if len(login) > 50:
    return "Login za długi!"
```

3️⃣ 🔒 Ograniczenia uprawnień

```
-- ✅ Stwórz użytkownika tylko do aplikacji
CREATE USER 'app_user'@'localhost'
IDENTIFIED BY 'secret_password';

-- ✅ Daj tylko potrzebne uprawnienia
GRANT SELECT, INSERT, UPDATE ON shop.users TO 'app_user'@'localhost';
-- NIE DAWAJ: DROP, CREATE, ALTER!
```

4️⃣ 🛡️ Escape'owanie (gorsze od prepared)

```
-- Zmień ' na '' (podwójny apostrof)
login = login.replace("'", "''")
```

5️⃣ 🔍 Monitorowanie

- **Logi podejrzanych zapytań** 📝
- **Alerty przy długich zapytaniach** ⚠️
- **Firewall aplikacji webowej (WAF)** 🔥

📊 Ranking metod ochrony:

Metoda	Skuteczność	Trudność	Polecam
💎 Prepared Statements	99.9%	Łatwe	✅✅✅
🔍 Walidacja	85%	Średnie	✅✅
🔒 Uprawnienia	70%	Łatwe	✅✅
🛡️ Escape'owanie	60%	Trudne	⚠️

🚨 Typowe błędy programistów:

❌ "Tylko szukam, więc bezpieczne"

```
-- BŁĄD! SELECT też może być niebezpieczny:
SELECT * FROM products WHERE name = '$search'
-- Atak: search = "' UNION SELECT password FROM users WHERE '1'='1"
```

❌ "Sprawdzam tylko apostrofy"

```
-- BŁĄD! Nie tylko apostrofy są niebezpieczne:
SELECT * FROM users WHERE id = $user_id
-- Atak: user_id = "1; DROP TABLE users;"
```

❌ "Mam escape'owanie, więc OK"

- Escape'owanie to **ostateczność**, nie główna obrona! ⚠️

🎯 Szybki test na bezpieczeństwo:

Czy twoja aplikacja jest bezpieczna?

1. ✅ Używam prepared statements wszędzie?
2. ✅ Waliduje wszystkie dane wejściowe?
3. ✅ Użytkownik bazy ma minimalne uprawnienia?
4. ✅ Loguję podejrzane zapytania?

Jeśli wszystko ✅ → prawdopodobnie bezpieczne! 🛡️

> 💀 **NIGDY nie ufaj danym od użytkownika!** Każdy input to potencjalny atak! 🚨

> 💎 **Zapamiętaj:** Prepared Statements = "niezatapialny okręt" dla SQL Injection! ⚓🛡️

1️⃣3️⃣ Perspektywy (Views) - do czego służą? 👁️📊

Perspektywa (View) = "wirtualna tabela" 🎭 - wygląda jak tabela, ale to tylko zapisane zapytanie!

Wyobraź sobie okno 🪟: widzisz przez nie krajobraz, ale okno to nie jest krajobraz - tylko sposób na patrzenie!

🎯 Do czego służą perspektywy:

🔒 Bezpieczeństwo i ochrona danych

```
-- View tylko z bezpiecznymi danymi (bez pensji!)
CREATE VIEW pracownicy_publiczne AS
SELECT imie, nazwisko, dzial, telefon
FROM pracownicy;  -- NIE MA pensji!
```

Efekt: Ludzie widzą kontakty, ale nie pensje! 💰🙈

🧹 Uproszczenie skomplikowanych zapytań

```
-- Zamiast pisać za każdym razem złożony JOIN:
CREATE VIEW raport_sprzedazy AS
SELECT
    k.nazwa as klient,
    p.nazwa as produkt,
    z.ilosc,
    z.data_zamowienia
FROM zamowienia z
JOIN klienci k ON z.klient_id = k.id
JOIN produkty p ON z.produkt_id = p.id;

-- Teraz wystarczy:
SELECT * FROM raport_sprzedazy WHERE data_zamowienia > '2024-01-01';
```

📊 Agregacje i raporty

```
-- View z gotowymi statystykami
CREATE VIEW statystyki_dzialu AS
SELECT
    dzial,
    COUNT(*) as ilosc_pracownikow,
    AVG(pensja) as srednia_pensja,
    MAX(pensja) as najwyzsza_pensja
FROM pracownicy
GROUP BY dzial;
```

🔗 Integracja i łączenie danych

- Łączenie danych z różnych tabel w jeden logiczny widok
- "Spłaszczanie" skomplikowanych relacji
- Ujednolicenie formatów danych

✅ Zalety Views:

- 📝 **Łatwość użycia** - skomplikowane = proste
- 🔒 **Bezpieczeństwo** - ukrywa wrażliwe dane
- 🔄 **Aktualne dane** - zawsze najnowsze (to tylko zapytanie!)
- 🎯 **Logiczne grupowanie** - dane pogrupowane tematycznie

⚠️ Wady Views:

- 🐌 **Wydajność** - za każdym razem wykonuje zapytanie
- ❌ **Ograniczenia UPDATE** - nie zawsze można modyfikować
- 🤔 **Złożoność** - view na view na view... może być chaos!

> 💡 **Zapamiętaj:** View = "zapisane zapytanie udające tabelę!" 🎭

1️⃣4️⃣ Wartość NULL - do czego się używa? 🤷‍♀️ ❓

NULL = "nie wiem" 🤷‍♀️ lub "nie ma wartości" 📪

UWAGA: NULL ≠ 0 ❌ i NULL ≠ "" (pusty tekst) ❌

🎯 Do czego służy NULL:

- **Brak informacji** - np. nie znamy telefonu klienta 📞 ❓
- **Opcjonalne pola** - np. drugie imię (nie każdy ma)
- **Przyszłe wartości** - np. data zakończenia projektu (jeszcze nie wiadomo) 📅 ❓

🔍 Jak sprawdzać NULL:

```
-- ✅ DOBRZE:
WHERE telefon IS NULL
WHERE telefon IS NOT NULL

-- ❌ ŹLE (zawsze FALSE!):
WHERE telefon = NULL
WHERE telefon != NULL
```

⚠️ PUŁAPKI NULL:

1️⃣ Funkcje agregujące:

```
COUNT(*) = 5         -- wszystkie wiersze
COUNT(telefon) = 3   -- tylko te z telefonem (pomija NULL!)
```

2️⃣ **Podzapytania z IN:**

```
-- NIEBEZPIECZNE! Jeśli w zbiorze jest NULL:
WHERE id IN (1, 2, NULL)  -- może nie działać jak myślisz!

-- BEZPIECZNE:
WHERE id IN (SELECT id FROM tabela WHERE id IS NOT NULL)
```

3️⃣ **Porównania z NULL:**

- `NULL = NULL` → **NULL** (nie TRUE!) 🤯
- `NULL + 5` → **NULL**
- `NULL OR TRUE` → **TRUE** ✅

> 💡 **Zapamiętaj:** NULL to "czarna dziura" - wszystko co dotknie staje się NULL! 🕳️

1️⃣5️⃣ Integralność referencyjna - co to i jakie rodzaje? 🔗🛡️

Integralność referencyjna = "żadnych wiszących linków!" 🚫🔗 - klucz obcy zawsze musi wskazywać na coś co istnieje!

Wyobraź sobie bibliotekę 📚: nie można wypożyczyć książki na nieistniejącego czytelnika!

🎯 Podstawowa zasada:

```
ZAMÓWIENIA.klient_id → musi wskazywać na istniejący KLIENT.id
```

Błąd: Zamówienie od klienta nr 999, ale klient nr 999 nie istnieje! 👻

⚡ Akcje referencyjne (co zrobić gdy "rodzic" znika):

🚫 RESTRICT (domyślne - najbezpieczniejsze)

```
FOREIGN KEY (klient_id) REFERENCES klienci(id) ON DELETE RESTRICT
```

Efekt: "NIE POZWOLĘ usunąć klienta jeśli ma zamówienia!" ✋ **Kiedy:** Bezpieczne, ale czasem uciążliwe

💥 CASCADE (kaskadowe - niebezpieczne!)

```
FOREIGN KEY (klient_id) REFERENCES klienci(id) ON DELETE CASCADE
```

Efekt: "Usuwam klienta = usuwam też wszystkie jego zamówienia!" 🌊 **UWAGA:** Efekt domina - możesz stracić więcej niż myślisz! ⚠️

🔄 SET NULL (ustaw na null)

```
FOREIGN KEY (manager_id) REFERENCES managers(id) ON DELETE SET NULL
```

Efekt: "Usuwam szefa = pracownicy zostają bez szefa (NULL)" 👤➡️❓ **Kiedy:** Gdy powiązanie może "zostać osierocone"

🔧 SET DEFAULT (ustaw domyślną wartość)

```
FOREIGN KEY (status_id) REFERENCES statusy(id) ON DELETE SET DEFAULT
```

Efekt: "Usuwam status = ustaw domyślny status 'aktywny'" 🔄 **Kiedy:** Gdy masz sensowną wartość domyślną

⏳ NO ACTION (sprawdź na końcu)

```
FOREIGN KEY (klient_id) REFERENCES klienci(id) ON DELETE NO ACTION
```

Efekt: Jak RESTRICT, ale sprawdza na końcu transakcji **Kiedy:** W złożonych operacjach z triggerami

📊 Tabela porównawcza:

Akcja	Bezpieczeństwo	Wygoda	Kiedy używać
RESTRICT	🔒🔒🔒	😐	Dane krytyczne
CASCADE	⚠️	😊😊	Dane zależne
SET NULL	🔒🔒	😊	Powiązania opcjonalne
SET DEFAULT	🔒	😊😊	Masz dobrą domyślną

> 💀 **UWAGA:** CASCADE to "broń masowego rażenia" - jeden klik może usunąć tysiące rekordów!

1️⃣6️⃣ Podstawy normalizacji - dlaczego organizujemy dane? 🏗️📊

Normalizacja = "sprzątanie w szafie" 🧹 - organizujemy dane tak, żeby nie było bałaganu i duplikatów!

Wyobraź sobie pokój studenta 🏠 : ubrania wszędzie, książki porozrzucane, nie wiadomo gdzie co znaleźć! Normalizacja to proces przywracania porządku.

🎯 Dlaczego normalizacja jest potrzebna:

💰 Problem 1: Redundancja (duplikaty)

```
ZAMÓWIENIA_ŹLEPRZEMYŚLANE:
+----+----------+----------------+----------+
| id | klient   | adres_klienta  | produkt  |
+----+----------+----------------+----------+
| 1  | Jan      | ul. Główna 1   | Laptop   |
| 2  | Jan      | ul. Główna 1   | Mysz     |  ← Duplikat adresu!
| 3  | Anna     | ul. Nowa 5     | Klawiatura|
+----+----------+----------------+----------+
```

Problem: Adres Jana powtarza się - marnowanie miejsca! 📦💸

🤔 Problem 2: Anomalie modyfikacji

- **Update anomaly:** Zmieniasz adres Jana w 1 miejscu, zapominasz w 2 miejscu → chaos! 🔄❌
- **Insert anomaly:** Nie możesz dodać nowego klienta bez zamówienia 🚫
- **Delete anomaly:** Usuwasz ostatnie zamówienie → tracisz dane klienta! 💀

🤹 Problem 3: Niespójność danych

```
Rekord 1: Jan, ul. Główna 1
Rekord 2: Jan, ul. Główna 1a  ← Która wersja jest prawidłowa?
```

🛠️ Rozwiązanie - podział na tabele:

✅ Po normalizacji:

```
-- Tabela KLIENCI (bez duplikatów!)
KLIENCI:
+----+----------+----------------+
| id | nazwa    | adres          |
+----+----------+----------------+
| 1  | Jan      | ul. Główna 1   |
| 2  | Anna     | ul. Nowa 5     |
+----+----------+----------------+

-- Tabela ZAMÓWIENIA (tylko potrzebne dane!)
ZAMÓWIENIA:
+----+------------+----------+
| id | klient_id  | produkt  |
```

```
+----+------------+----------+
| 1  | 1          | Laptop   |
| 2  | 1          | Mysz     |
| 3  | 2          | Klawiatura|
+----+------------+----------+
```

🎯 Korzyści normalizacji:

- 💾 **Oszczędność miejsca** - brak duplikatów
- 🔄 **Łatwiejsze zmiany** - zmiana w jednym miejscu
- ✅ **Spójność danych** - jedna wersja prawdy
- 🛡️ **Integralność** - brak sprzecznych informacji

📚 Analogia - biblioteka:

Przed normalizacją: Każda książka ma naklejkę z pełnym adresem biblioteki 📚 📮 **Po normalizacji:** Książki mają kod biblioteki, a adresy w oddzielnej tabeli 📚 🔗 📮

> 💡 **Zapamiętaj:** Normalizacja = "każda informacja w jednym miejscu!" 🎯

1️⃣7️⃣ Normalizacja - postacie normalne! 🏗️📊

Normalizacja = "porządkowanie stopniowe" 📶 - jak budowanie domu, piętro po piętrze!

Wyobraź sobie organizację szafy 👕 👖 👗 : najpierw grupujesz ubrania, potem kategoryzujesz, wreszcie układasz idealnie!

🎯 Główne postacie normalne:

1️⃣ 1NF (Pierwsza Postać Normalna) - "Podziel na kawałki" ✂️

Zasada: Każda komórka = jedna wartość atomowa!

❌ **Źle:**

```
KLIENCI:
+----+----------+------------------+
| id | nazwa    | telefony         |
+----+----------+------------------+
| 1  | Jan      | 123-456, 789-012 |  ← Dwie wartości w jednej komórce!
+----+----------+------------------+
```

✅ **Dobrze:**

```
KLIENCI:
+----+----------+----------+
| id | nazwa    | telefon  |
+----+----------+----------+
```

```
| 1  | Jan      | 123-456  |
| 1  | Jan      | 789-012  |
+----+----------+----------+
```

2️⃣ 2NF (Druga Postać Normalna) - "Cały klucz lub wcale" 🗝️

Zasada: Każda kolumna zależy od **CAŁEGO** klucza głównego!

❌ Problem - częściowa zależność:

```
ZAMÓWIENIA_PRODUKTY:
+------------+------------+----------+---------------+
| zamow_id   | produkt_id | ilosc    | cena_produktu |  ← Cena zależy
tylko od produkt_id!
+------------+------------+----------+---------------+
```

✅ Rozwiązanie - podział tabel:

```
ZAMÓWIENIA_PRODUKTY:  (ilosc zależy od obu kluczy)
+------------+------------+----------+
| zamow_id   | produkt_id | ilosc    |
+------------+------------+----------+

PRODUKTY:  (cena zależy tylko od produktu)
+------------+---------------+
| produkt_id | cena_produktu |
+------------+---------------+
```

3️⃣ 3NF (Trzecia Postać Normalna) - "Bez łańcuszków" 🔗❌

Zasada: Kolumny zależą od klucza, nie od siebie nawzajem!

❌ Problem - zależność przechodnia:

```
PRACOWNICY:
+----+----------+----------+---------------+
| id | nazwisko | dzial_id | nazwa_dzialu   |  ← nazwa_dzialu zależy od
dzial_id!
+----+----------+----------+---------------+
```

Łańcuszek: id → dzial_id → nazwa_dzialu

✅ Rozwiązanie:

```
PRACOWNICY:
+----+----------+----------+
| id | nazwisko | dzial_id |
+----+----------+----------+

DZIALY:
+----------+----------------+
| dzial_id | nazwa_dzialu   |
+----------+----------------+
```

🚀 BCNF (Boyce-Codd) - "Perfekcja" 💎

Zasada: Każdy determinant musi być kluczem kandydującym!

Bardzo zaawansowane - w praktyce rzadko potrzebne 🎓

🎯 Praktyczna rada:

- **1NF** - zawsze! (podstawa)
- **2NF** - prawie zawsze
- **3NF** - w 95% przypadków wystarcza! ✅
- **BCNF** - tylko w skomplikowanych systemach

📚 Analogia - biblioteka:

- **1NF:** Książka ma jeden tytuł, nie "Tytuł1, Tytuł2"
- **2NF:** ISBN określa książkę, nie autor+kategoria
- **3NF:** Adres wydawnictwa w osobnej tabeli, nie w książce

> 💡 **Zapamiętaj:** Normalizacja = "każda informacja w swoim miejscu, bez duplikatów!" 🎯

1️⃣8️⃣ Zakleszczenie - gdy transakcje grają w "przeciąganie liny"! 🔒💥

Zakleszczenie (deadlock) = impas! 🚧 Dwie transakcje czekają na siebie nawzajem i nie mogą się ruszyć!

🎭 Analogia - dwa auta na wąskim moście:

```
🚗 Auto A: stoi na moście, chce przejechać
🚚 Auto B: stoi na moście, chce przejechać
→ Żadne nie może przejechać! 🚧
```

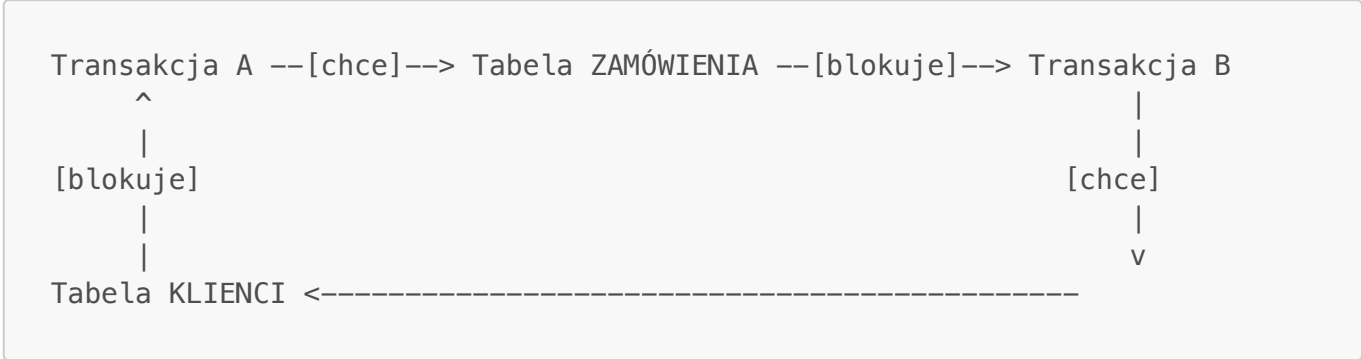
💥 Jak powstaje deadlock:

🔥 Klasyczny przykład:

```
⏰ Czas T1: Transakcja A zablokuje tabelę KLIENCI
⏰ Czas T2: Transakcja B zablokuje tabelę ZAMÓWIENIA
```

```
⏰ Czas T3: Transakcja A chce ZAMÓWIENIA (czeka na B!)
⏰ Czas T4: Transakcja B chce KLIENCI (czeka na A!)
🔄 → DEADLOCK! Obie czekają w nieskończoność!
```

🎯 Graficzne przedstawienie:



Cykl = Deadlock! 🔄 💀

🛡️ Metody rozwiązywania:

1️⃣ 🚫 Zapobieganie (najlepsze!):

🎯 Blokowanie w kolejności alfabetycznej:

```sql
-- ✅ ZAWSZE blokuj tabele w tej samej kolejności:
-- 1. KLIENCI (K), 2. ZAMÓWIENIA (Z)

-- Transakcja A:
LOCK TABLE KLIENCI;
LOCK TABLE ZAMÓWIENIA;

-- Transakcja B:
LOCK TABLE KLIENCI;  -- Czeka na A (OK!)
LOCK TABLE ZAMÓWIENIA;
```

🕐 Timeout:

```sql
-- Ustaw limit czasu - po 30 sekundach wycofaj
SET innodb_lock_wait_timeout = 30;
```

2️⃣ 🔍 Wykrywanie (automatyczne):

📊 Graf oczekiwań:

- System tworzy graf "kto na kogo czeka"
- Gdy znajdzie cykl → wykrywa deadlock
- Automatycznie wybiera "ofiarę" i ją wycofuje

🎯 Kto zostaje ofiarą:

- Transakcja z mniejszym kosztem wycofania
- Młodsza transakcja (mniej pracy do stracenia)

3️⃣ ⏰ Znaczniki czasu:

🏆 Starsze transakcje mają priorytet:

- **Wait-Die:** Starsza czeka, młodsza ginie
- **Wound-Wait:** Starsza zabija młodszą, młodsza czeka

🎯 Praktyczne porady:

✅ Jak zapobiegać:

1. **Kolejność blokowania** - zawsze w tej samej kolejności
2. **Krótkie transakcje** - mniej czasu na konflikty
3. **Odpowiednie indeksy** - szybsze blokowanie
4. **Timeout** - zabezpieczenie przed nieskończonością

🚨 Sygnały ostrzegawcze:

- Aplikacja "zawiesza się" losowo
- Wzrost czasu odpowiedzi
- Komunikaty o deadlock w logach

📚 Analogia - skrzyżowanie:

Bez sygnalizacji: Dwa auta jadą jednocześnie → wypadek! 💥 **Z sygnalizacją:** Jeden jedzie, drugi czeka → wszystko OK! ✅

> 💡 **Zapamiętaj:** Deadlock = "Pat w szachach", najlepiej go unikać! 🎯

> 🎯 **Złota reguła:** Blokuj tabele ZAWSZE w tej samej kolejności! 🔢

1️⃣9️⃣ Relacje M:N (wiele do wielu) i jak je realizować 🔗🔀

Relacje M:N = każdy rekord z jednej tabeli może być powiązany z wieloma z drugiej tabeli (i odwrotnie)! 🤹

🎯 Przykłady relacji M:N:

- 👨‍🎓 **Studenci** ↔ 📚 **Przedmioty** (student ma wiele przedmiotów, przedmiot ma wielu studentów)
- 👨‍⚕️ **Lekarze** ↔ 👥 **Pacjenci** (lekarz ma wielu pacjentów, pacjent może mieć wielu lekarzy)
- 🎬 **Filmy** ↔ 🤹 **Aktorzy** (film ma wielu aktorów, aktor gra w wielu filmach)

🛠️ Jak zrealizować relację M:N:

❌ BŁĘDNE podejście (nie działa!):

```sql
-- NIE MOŻNA! Kolumna nie może mieć wielu wartości
CREATE TABLE studenci (
    id INT PRIMARY KEY,
    imie VARCHAR(50),
    przedmioty VARCHAR(500)  -- "1,3,5,7" ← BŁĄD!
);
```

✅ **POPRAWNE podejście - tabela pośrednia:**

```sql
-- Tabele główne
CREATE TABLE studenci (
    id INT PRIMARY KEY,
    imie VARCHAR(50),
    nazwisko VARCHAR(50)
);

CREATE TABLE przedmioty (
    id INT PRIMARY KEY,
    nazwa VARCHAR(100),
    punkty_ects INT
);

-- TABELA POŚREDNIA (junction table)
CREATE TABLE student_przedmiot (
    student_id INT,
    przedmiot_id INT,
    ocena DECIMAL(2,1),  -- dodatkowe dane relacji!
    data_zaliczenia DATE,
    PRIMARY KEY (student_id, przedmiot_id),  -- klucz złożony
    FOREIGN KEY (student_id) REFERENCES studenci(id),
    FOREIGN KEY (przedmiot_id) REFERENCES przedmioty(id)
);
```

🔍 **Jak używać relacji M:N:**

```sql
-- Jakie przedmioty ma student Kowalski?
SELECT p.nazwa
FROM przedmioty p
JOIN student_przedmiot sp ON p.id = sp.przedmiot_id
JOIN studenci s ON sp.student_id = s.id
WHERE s.nazwisko = 'Kowalski';

-- Którzy studenci mają Matematykę?
SELECT s.imie, s.nazwisko
FROM studenci s
JOIN student_przedmiot sp ON s.id = sp.student_id
JOIN przedmioty p ON sp.przedmiot_id = p.id
WHERE p.nazwa = 'Matematyka';
```

💡 Zalety tabeli pośredniej:

- 📊 **Dodatkowe atrybuty** - ocena, data, uwagi
- 🔐 **Klucz złożony** - gwarantuje unikalność pary
- 🎯 **Elastyczność** - łatwo dodawać/usuwać powiązania

🔄 Porównanie z relacją 1:1:

Relacja 1:1 (już omówiona w pyt. 7):

- Jeden klucz obcy z UNIQUE
- Lub wspólny klucz główny
- Rzadko używana

> 💡 **Zapamiętaj:** M:N = zawsze tabela pośrednia! 🌉

2️⃣0️⃣ Indeksy w bazie danych - turbo dla zapytań! 🚀📊

Indeks = "skorowidz w książce" 📖 - zamiast przeszukiwać każdą stronę, skaczesz od razu do właściwej!

Wyobraź sobie bibliotekę 📚 : bez katalogu musiałbyś sprawdzić każdą książkę, z katalogiem → znajdziesz od razu!

🎯 Do czego służą indeksy:

⚡ Przyspieszenie wyszukiwania:

```
-- ❌ Bez indeksu: przeszukanie 1 000 000 rekordów
-- ✅ Z indeksem: przeszukanie ~20 rekordów (logarytmicznie!)

SELECT * FROM klienci WHERE email = 'jan@example.com';
```

🏆 Przyspieszenie sortowania:

```
-- ORDER BY wykorzystuje indeks automatycznie
SELECT * FROM produkty ORDER BY cena;  -- Szybko z indeksem na 'cena'
```

🔗 Przyspieszenie JOIN:

```
-- JOIN wykorzystuje indeksy na kluczach obcych
SELECT * FROM zamowienia z
JOIN klienci k ON z.klient_id = k.id;  -- Szybko z indeksem na klient_id
```

🏗️ Jak działają indeksy:

🌳 Struktura B-Tree (drzewo):

```
                  [50]
                /      \
          [25]                [75]
         /    \              /    \
    [10]      [40]        [60]     [90]
   /  \      /  \        /  \     /  \
[5][15][35][45]    [55][65][80][95]
```

🎯 Dlaczego szybkie:

- Zamiast sprawdzać 100 rekordów → tylko 7 kroków!
- Logarytmiczna złożoność: O(log n)

📚 Analogia - słownik:

Bez indeksu: Szukasz słowa "żyrafa" → czytasz od "A" do "Ż" **Z indeksem:** Otwierasz od razu przy "Ż" → znajdziesz w sekundę!

🎭 Rodzaje indeksów:

1️⃣ Indeks klastrowy (główny):

```sql
-- Automatycznie tworzony dla PRIMARY KEY
CREATE TABLE produkty (
    id INT PRIMARY KEY,  -- ← Indeks klastrowy
    nazwa VARCHAR(100)
);
```

Cechy:

- Tylko jeden na tabelę
- Dane fizycznie posortowane według tego indeksu
- Najszybszy dostęp

2️⃣ Indeks nieklastrowy (pomocniczy):

```sql
-- Ręcznie tworzony dla często wyszukiwanych kolumn
CREATE INDEX idx_email ON klienci(email);
CREATE INDEX idx_nazwisko ON klienci(nazwisko);
```

Cechy:

- Wiele na tabelę
- Wskaźnik do danych (nie same dane)
- Szybki, ale nie tak jak klastrowy

3️⃣ **Indeks złożony:**

```
-- Indeks na wielu kolumnach
CREATE INDEX idx_imie_nazwisko ON klienci(imie, nazwisko);
```

Kiedy działa:

- `WHERE imie = 'Jan'` ✅
- `WHERE imie = 'Jan' AND nazwisko = 'Kowalski'` ✅
- `WHERE nazwisko = 'Kowalski'` ❌ (tylko druga kolumna)

⚡ Kiedy tworzyć indeksy:

✅ **ZAWSZE:**

- **PRIMARY KEY** (automatycznie)
- **FOREIGN KEY** (łączenie tabel)
- **Często wyszukiwane kolumny** (email, login)

✅ **CZASAMI:**

- **Często sortowane kolumny** (data, cena)
- **Kolumny w GROUP BY**
- **Kolumny w ORDER BY**

❌ **NIGDY:**

- **Często zmieniane kolumny** (każda zmiana = aktualizacja indeksu)
- **Małe tabele** (< 1000 rekordów)
- **Kolumny z duplikatami** (płeć: M/K)

🎯 Praktyczne porady:

📊 **Jak sprawdzić czy indeks jest używany:**

```
-- MySQL
EXPLAIN SELECT * FROM klienci WHERE email = 'jan@example.com';

-- PostgreSQL
EXPLAIN ANALYZE SELECT * FROM klienci WHERE email = 'jan@example.com';
```

🚨 **Pułapki indeksów:**

1. **Więcej ≠ lepiej** - za dużo indeksów spowalnia INSERT/UPDATE
2. **Indeks nie zawsze używany** - np. przy funkcjach: `WHERE UPPER(nazwisko) = 'KOWALSKI'`
3. **Miejsce na dysku** - indeksy zajmują dodatkowe miejsce

💡 **Złote zasady:**

- Indeks na każdy klucz obcy (FOREIGN KEY)
- Indeks na kolumny w WHERE często używane
- Nie więcej niż 5-7 indeksów na tabelę
- Monitoruj wydajność zapytań

📚 Analogia - biblioteka:

Bez indeksu: Szukasz książki o kotach → sprawdzasz każdą książkę 📚 📚 📚 **Z indeksem:** Szukasz w katalogu "K" → Koty, półka 15 → gotowe! 📖 🎯

> 💡 **Zapamiętaj:** Indeks = "ekspres do danych" - szybko znajdziesz, ale kosztuje miejsce! 🚀

> 🎯 **Podstawowa zasada:** Indeks na każde częste WHERE! 🔍

2️⃣1️⃣ View vs Tabela tymczasowa - różnice 👁️🆚📄

Oba wyglądają jak tabela, ale działają CAŁKOWICIE inaczej! 🤷‍♂️

👁️ **PERSPEKTYWA (VIEW)** - "wirtualna tabela"

🎯 **Czym jest:**

- **Zapisane zapytanie** udające tabelę 📝
- **Nie przechowuje danych** - za każdym razem wykonuje zapytanie ♻️
- **Zawsze aktualna** - pokazuje najnowsze dane 🔄

💡 **Przykład:**

```
-- Tworzenie View
CREATE VIEW active_users AS
SELECT id, name, email
FROM users
WHERE status = 'active';

-- Używanie (wygląda jak tabela!)
SELECT * FROM active_users;  -- Ale to w rzeczywistości wykonuje pełne
zapytanie!
```

✅ **Zalety View:**

- 🔄 **Zawsze aktualne** - dane na żywo
- 💾 **Zero miejsca** - nie zajmuje dodatkowego miejsca

- 🔒 **Bezpieczeństwo** - kontrola dostępu
- ♾️ **Trwałe** - istnieją dopóki ich nie usunieś

❌ **Wady View:**

- 🐌 **Może być wolne** - za każdym razem wykonuje zapytanie
- ❌ **Ograniczone UPDATE** - nie zawsze można modyfikować

📄 TABELA TYMCZASOWA - "prawdziwa tabela na chwilę"

🎯 **Czym jest:**

- **Fizyczna tabela** z prawdziwymi danymi 💾
- **Przechowuje dane** na dysku/w pamięci 🗃️
- **Zdjęcie w momencie** - dane z momentu utworzenia 📸

💡 **Przykład:**

```
-- Tworzenie tabeli tymczasowej
CREATE TEMPORARY TABLE temp_sales AS
SELECT product_id, SUM(amount) as total
FROM orders
WHERE order_date = '2024-01-01'
GROUP BY product_id;

-- Używanie (prawdziwa tabela!)
SELECT * FROM temp_sales;  -- Szybko! Dane są już policzone
```

✅ **Zalety tabeli tymczasowej:**

- ⚡ **Szybkie** - dane już przeliczone i zapisane
- 🔧 **Pełne możliwości** - można wszystko (UPDATE, INDEX, etc.)
- 🎯 **Optymalizacja** - skomplikowane obliczenia raz
- 🔄 **Manipulacja** - można dodawać, zmieniać, usuwać

❌ **Wady tabeli tymczasowej:**

- 💾 **Zajmuje miejsce** - prawdziwe dane na dysku
- ⏰ **Nieaktualne** - dane z momentu utworzenia
- 🧹 **Trzeba sprzątać** - trzeba ją usunąć po użyciu
- ⏱️ **Czasochłonne tworzenie** - trzeba poczekać na skopiowanie danych

📊 Porównanie:

Aspekt	👁️ VIEW	📄 TEMP TABLE
Dane	Wirtualne	Fizyczne

Aspekt	👁️ VIEW	📄 TEMP TABLE
Szybkość	Wolniejsze	Szybsze
Aktualność	Zawsze aktualne	Zamrożone w czasie
Miejsce	0 MB	Zajmuje miejsce
Modyfikacja	Ograniczona	Pełna
Zarządzanie	Automatyczne	Ręczne sprzątanie

🎯 Kiedy co używać:

👁️ Użyj VIEW gdy:

- Chcesz zawsze najnowsze dane 🔄
- Zapytanie jest proste i szybkie ⚡
- Chcesz ukryć złożoność 🙈
- Potrzebujesz kontroli dostępu 🔒

📄 Użyj TEMP TABLE gdy:

- Robisz skomplikowane obliczenia 🧮
- Potrzebujesz manipulować wynikami ✏️
- Szybkość jest najważniejsza ⚡
- Pracujesz na "zdjęciu danych" 📸

> 💡 **Zapamiętaj:** View = "na żywo", Temp Table = "zamrożone"! 🔄 ❄️

2️⃣2️⃣ Od diagramu ER do SQL - jak przełożyć? 📊➡️💻

Diagram ER ➡️ **SQL** = tłumaczenie rysunków na kod! 🎨 💻

🗺️ Mapa drogowa ER → SQL:

1️⃣ 📦 Encje = Tabele

```
Diagram ER:     [STUDENT]
                ↓
SQL:           CREATE TABLE studenci (...)
```

2️⃣ 🏷️ Atrybuty = Kolumny

```
Diagram ER:     [STUDENT] – imię, nazwisko, nr_indeksu
                ↓
SQL:           CREATE TABLE studenci (
                   imie VARCHAR(50),
                   nazwisko VARCHAR(50),
```

```
    nr_indeksu VARCHAR(10)
);
```

3️⃣ 🗝️ Identyfikatory = Klucze główne

```
Diagram ER:     [STUDENT] – ID (podkreślone)
                ↓
SQL:           CREATE TABLE studenci (
                   id INT PRIMARY KEY,
                   ...
               );
```

4️⃣ 🔗 Związki = Klucze obce

👥 Związek 1:N (jeden do wielu):

```
Diagram ER:     [KLIENT] ---< ma >--- [ZAMÓWIENIE]
                ↓
SQL:           CREATE TABLE klienci (
                   id INT PRIMARY KEY,
                   nazwa VARCHAR(100)
               );

               CREATE TABLE zamowienia (
                   id INT PRIMARY KEY,
                   klient_id INT,  -- klucz obcy!
                   FOREIGN KEY (klient_id) REFERENCES klienci(id)
               );
```

👤 Związek 1:1 (jeden do jednego):

```
Diagram ER:     [OSOBA] ---|| ma ||--- [PASZPORT]
                ↓
SQL:           CREATE TABLE osoby (
                   id INT PRIMARY KEY,
                   imie VARCHAR(50)
               );

               CREATE TABLE paszporty (
                   osoba_id INT PRIMARY KEY,  -- wspólny klucz!
                   numer VARCHAR(20),
                   FOREIGN KEY (osoba_id) REFERENCES osoby(id)
               );
```

🔀 **Związek M:N (wiele do wielu):**

```
Diagram ER:     [STUDENT] >---< uczęszcza >---< [PRZEDMIOT]
                ↓
SQL:           CREATE TABLE studenci (
                   id INT PRIMARY KEY,
                   imie VARCHAR(50)
               );

               CREATE TABLE przedmioty (
                   id INT PRIMARY KEY,
                   nazwa VARCHAR(100)
               );

               -- TABELA POŚREDNIA!
               CREATE TABLE student_przedmiot (
                   student_id INT,
                   przedmiot_id INT,
                   PRIMARY KEY (student_id, przedmiot_id),
                   FOREIGN KEY (student_id) REFERENCES studenci(id),
                   FOREIGN KEY (przedmiot_id) REFERENCES przedmioty(id)
               );
```

5️⃣ 🎯 **Atrybuty związków = Kolumny w tabeli pośredniej**

```
Diagram ER:     [STUDENT] >---< uczęszcza(ocena, data) >---< [PRZEDMIOT]
                ↓
SQL:           CREATE TABLE student_przedmiot (
                   student_id INT,
                   przedmiot_id INT,
                   ocena DECIMAL(2,1),  -- atrybut związku!
                   data_zaliczenia DATE, -- atrybut związku!
                   PRIMARY KEY (student_id, przedmiot_id),
                   FOREIGN KEY (student_id) REFERENCES studenci(id),
                   FOREIGN KEY (przedmiot_id) REFERENCES przedmioty(id)
               );
```

🎭 Specjalne przypadki:

🔄 **Związek cykliczny (rekurencyjny):**

```
Diagram ER:     [PRACOWNIK] ---< zarządza >--- [PRACOWNIK]
                ↓
SQL:           CREATE TABLE pracownicy (
                   id INT PRIMARY KEY,
                   imie VARCHAR(50),
                   manager_id INT,  -- wskazuje na tego samego pracownika!
```

```
                    FOREIGN KEY (manager_id) REFERENCES pracownicy(id)
                );
```

🔺 **Związek ternary (3 encje):**

```
Diagram ER:     [LEKARZ] ---< konsultacja >--- [PACJENT]
                                  |
                              [CHOROBA]
                 ↓
SQL:            CREATE TABLE konsultacje (
                    lekarz_id INT,
                    pacjent_id INT,
                    choroba_id INT,
                    data_konsultacji DATE,
                    PRIMARY KEY (lekarz_id, pacjent_id, choroba_id),
                    FOREIGN KEY (lekarz_id) REFERENCES lekarze(id),
                    FOREIGN KEY (pacjent_id) REFERENCES pacjenci(id),
                    FOREIGN KEY (choroba_id) REFERENCES choroby(id)
                );
```

📋 Checklist przenoszenia:

- ✅ Każda encja = osobna tabela
- ✅ Każdy atrybut = kolumna
- ✅ Identyfikator = PRIMARY KEY
- ✅ Związek 1:N = klucz obcy
- ✅ Związek 1:1 = unikalny klucz obcy lub wspólny PK
- ✅ Związek M:N = tabela pośrednia
- ✅ Atrybuty związków = kolumny w tabeli pośredniej

> 💡 **Zapamiętaj:** ER to mapa, SQL to drogówki - musisz przetłumaczyć mapę na instrukcje jazdy! 🗺️🚗

2️⃣3️⃣ Do jakiej postaci normalnej najłatwiej sprowadzić bazę? 📊🎯

Odpowiedź: Do **1NF** zawsze można, do **3NF** prawie zawsze, ale czasem trzeba się zatrzymać! 🛑

✅ 1NF - ZAWSZE możliwe!

Pierwsza Postać Normalna = podstawowe porządki w tabeli! 🧹

🎯 Co robi 1NF:

- **Atomowość** - każda komórka = jedna wartość 🔧
- **Eliminuje grupy powtarzające** się 🔄❌
- **Unikalność wierszy** - każdy wiersz inny ✨

💡 Przykład przekształcenia do 1NF:

```sql
-- ❌ ŹŁE (nie jest 1NF):
CREATE TABLE studenci_zle (
    id INT,
    imie VARCHAR(50),
    telefony VARCHAR(200)  -- "123-456, 789-012, 555-666" ← BŁĄD!
);

-- ✅ DOBRE (1NF):
CREATE TABLE studenci (
    id INT,
    imie VARCHAR(50)
);

CREATE TABLE telefony_studentow (
    student_id INT,
    telefon VARCHAR(15),
    typ VARCHAR(20)  -- "domowy", "komórkowy"
);
```

🎯 3NF - prawie zawsze możliwe!

Trzecia Postać Normalna = eliminuje większość problemów praktycznych! 💪

❌ Kiedy NIE DA SIĘ sprowadzić do 1NF:

1️⃣ Kolumny wielowartościowe (najczęstszy problem)

```sql
-- Nie można przerobić jeśli dane są faktycznie wielowartościowe
CREATE TABLE produkty (
    id INT,
    nazwa VARCHAR(100),
    tags TEXT  -- "elektronika,komputer,laptop,gaming"
);
```

Problem: Jeśli tagi MUSZĄ być w jednej kolumnie (np. przez ograniczenia systemu)

2️⃣ Dane Legacy/Starsze systemy 👴

- Stare systemy mogą mieć ograniczenia
- Nie można zmienić struktury (np. system zewnętrzny)
- Koszty przepisania większe niż korzyści 💰

3️⃣ Denormalizacja celowa 🎯

```sql
-- Celowo zachowane dla wydajności
CREATE TABLE raport_sprzedazy (
    data_sprzedazy DATE,
```

```
    produkt VARCHAR(100),
    ilosc INT,
    cena DECIMAL(10,2),
    wartosc DECIMAL(10,2),  -- cena * ilosc (celowa redundancja!)
    klient VARCHAR(100),
    miasto_klienta VARCHAR(50),  -- denormalizacja dla szybkości
    region VARCHAR(50)
);
```

Powód: Rapory muszą być BARDZO szybkie ⚡

4️⃣ Typy danych BLOB/JSON 📄

```
CREATE TABLE dokumenty (
    id INT PRIMARY KEY,
    tresc JSON,  -- '{"imie": "Jan", "telefony": ["123", "456"]}'
    metadata BLOB
);
```

Problem: JSON/BLOB to "czarna skrzynka" - trudno normalizować 📦

5️⃣ Ograniczenia biznesowe 💼

- Firma wymaga określonej struktury
- Zgodność z innymi systemami
- Ograniczenia prawne/compliance 📜

🔧 Rozwiązania gdy nie da się 1NF:

💡 Opcja 1: Hybrydowe podejście

```
-- Główna tabela normalna + kolumna dodatkowa
CREATE TABLE produkty (
    id INT PRIMARY KEY,
    nazwa VARCHAR(100),
    kategoria_id INT  -- znormalizowane
);

CREATE TABLE produkty_tagi (
    produkt_id INT,
    tag VARCHAR(50)  -- znormalizowane tagi
);

CREATE TABLE produkty_meta (
    produkt_id INT,
    dodatkowe_info JSON  -- dla elastyczności
);
```

💡 Opcja 2: Kontrolowana denormalizacja

- Świadomie zachowaj redundancję
- Dodaj triggery dla spójności
- Dokumentuj decyzje 📝

📊 Statystyka praktyczna:

- **95%** tabel da się sprowadzić do 3NF ✅
- **80%** tabel da się sprowadzić do BCNF ✅
- **100%** tabel da się sprowadzić do 1NF (z małymi wyjątkami) ✅

> 💡 **Zapamiętaj:** 1NF = "podstawowe porządki", 3NF = "wystarczająco dobre", BCNF = "idealne ale czasem niemożliwe"! 🎯

2️⃣4️⃣ Rodzaje blokad i ich kolizje 🔒⚔️

Blokady = "zajęte, nie dotykaj!" 🚫✋ - sposób na unikanie konfliktów między transakcjami!

🎯 Rodzaje blokad według zakresu:

🔍 Blokada wiersza (Row Lock)

- **Zakres:** Pojedynczy wiersz 📏
- **Zaleta:** Najlepsza współbieżność ⚡
- **Wada:** Może być dużo blokad 📊

```
-- Blokuje tylko jeden wiersz
UPDATE klienci SET email = 'new@email.com' WHERE id = 123;
```

📄 Blokada strony (Page Lock)

- **Zakres:** Grupa wierszy (strona dysku) 📄
- **Zaleta:** Kompromis między wydajnością a współbieżnością ⚖️
- **Wada:** Czasem blokuje więcej niż trzeba 🤷‍♀️

🗃️ Blokada tabeli (Table Lock)

- **Zakres:** Cała tabela 📊
- **Zaleta:** Prosta w zarządzaniu 😌
- **Wada:** Słaba współbieżność 🐌

```
-- Blokuje całą tabelę
LOCK TABLE klienci IN EXCLUSIVE MODE;
```

📝 Blokada kolumny (Column Lock)

- **Zakres:** Pojedyncza kolumna 📝
- **Rzadko używane** w praktyce
- **Skomplikowane** w implementacji 🤯

🎭 Rodzaje blokad według dostępu:

🤝 Shared Lock (S) - dzielona

- **Kto może:** Wielu czytelników jednocześnie 👥
- **Blokuje:** Pisanie ✏️❌

```
-- Może być wiele jednocześnie
SELECT * FROM klienci WHERE id = 123;  -- S-lock
```

🚫 Exclusive Lock (X) - wyłączna

- **Kto może:** Tylko jedna transakcja 👤
- **Blokuje:** Wszystko inne 🚫

```
-- Tylko jedna na raz
UPDATE klienci SET saldo = saldo + 100 WHERE id = 123;  -- X-lock
```

⚔️ Macierz kolizji blokad:

Chce \ Ma	🤝 S	🚫 X
🤝 S	✅ OK	❌ Konflikt
🚫 X	❌ Konflikt	❌ Konflikt

Tłumaczenie:

- **S + S =** ✅ "Wielu może czytać jednocześnie" 📚👥
- **S + X =** ❌ "Nie można czytać gdy ktoś pisze" 📚❌✏️
- **X + X =** ❌ "Nie można pisać gdy ktoś pisze" ✏️❌✏️

🏗️ Kolizje według zakresu:

⚔️ Wiersz vs Tabela

```
Transakcja A: X-lock na wiersz 123
Transakcja B: chce X-lock na całą tabelę  ← KONFLIKT!
```

⚔️ Strona vs Wiersz

```
Transakcja A: X-lock na stronę 5
Transakcja B: chce S-lock na wiersz ze strony 5  ← KONFLIKT!
```

⚔️ Kolumna vs Wiersz

```
Transakcja A: X-lock na kolumnę 'email'
Transakcja B: chce X-lock na cały wiersz  ← KONFLIKT!
```

🎯 Hierarchia blokad:

```
TABELA
  ├── STRONA 1
  │    ├── Wiersz 1
  │    └── Wiersz 2
  └── STRONA 2
       ├── Wiersz 3
       └── Wiersz 4
```

Zasada: Blokada wyżej = blokuje wszystko niżej! ⬇️

💡 Praktyczne rady:

- **Używaj najmniejszy zakres** - wiersz > strona > tabela 🎯
- **Trzymaj krótko** - im krócej tym lepiej ⏱️
- **Unikaj eskalacji** - wiele małych blokad → jedna duża 📈
- **Kolejność zawsze ta sama** - unikaj deadlock! 🔄

> 💡 **Zapamiętaj:** S+S = OK, reszta = wojna! I im mniejsza blokada, tym lepiej! 🎯

2️⃣5️⃣ Redundancja danych - problem i rozwiązania 🔄💥

Redundancja = "te same dane w wielu miejscach" 📋📋📋 = zawsze źle (chyba że celowo)!

😱 Przykład redundancji:

```sql
-- ❌ REDUNDANTNA TABELA:
CREATE TABLE zamowienia_zle (
    id INT,
    klient_id INT,
    klient_imie VARCHAR(50),     -- REDUNDANCJA!
    klient_nazwisko VARCHAR(50), -- REDUNDANCJA!
    klient_email VARCHAR(100),   -- REDUNDANCJA!
    produkt_nazwa VARCHAR(100),  -- REDUNDANCJA!
    produkt_cena DECIMAL(10,2),  -- REDUNDANCJA!
```

```
    ilosc INT
);
```

Problem: Dane klienta powtarzają się w każdym zamówieniu! 🔁

💀 Skutki redundancji:

1️⃣ 🗄️ Marnotrawienie miejsca

```
1000 zamówień × 3 kolumny klienta × 50 znaków = 150,000 znaków!
Zamiast raz zapisać dane klienta = 150 znaków!
```

2️⃣ 🤦‍♀️ Anomalie aktualizacji

```
Klient zmienia email:
❌ Muszę zaktualizować 1000 wierszy zamówień!
✅ Powinienem zaktualizować 1 wiersz klienta!
```

3️⃣ 💥 Niespójność danych

```
Jan Kowalski w zamówieniu 1: email = jan@gmail.com
Jan Kowalski w zamówieniu 2: email = jan@yahoo.com
→ Który email jest prawdziwy?! 🤔
```

4️⃣ 🍼 Anomalie wstawiania

```
❌ Nie mogę dodać klienta bez zamówienia!
❌ Muszę powtórzyć dane klienta w każdym zamówieniu!
```

5️⃣ 🗑️ Anomalie usuwania

```
❌ Usunę ostatnie zamówienie klienta = stracę dane klienta!
```

✅ Jak usunąć redundancję:

🧹 Normalizacja - poziomy sprzątania:

📋 1NF - podstawowe porządki

- Każda komórka = jedna wartość
- "Nie pakuj wszystkiego do jednej szuflady!"

🔗 2NF - usuwa częściowe zależności

```sql
-- Problem: w kluczu (zamowienie_id, produkt_id)
-- nazwa_produktu zależy tylko od produkt_id!

-- ✅ Rozwiązanie:
CREATE TABLE zamowienia (
    id INT PRIMARY KEY,
    klient_id INT,
    data_zamowienia DATE
);

CREATE TABLE pozycje_zamowienia (
    zamowienie_id INT,
    produkt_id INT,
    ilosc INT,
    PRIMARY KEY (zamowienie_id, produkt_id)
);

CREATE TABLE produkty (
    id INT PRIMARY KEY,
    nazwa VARCHAR(100),  -- Tylko tutaj!
    cena DECIMAL(10,2)
);
```

🎯 3NF - usuwa przechodnie zależności

```sql
-- Problem: klient_miasto zależy od klient_id przez kod_pocztowy
-- klient_id → kod_pocztowy → miasto

-- ✅ Rozwiązanie:
CREATE TABLE klienci (
    id INT PRIMARY KEY,
    imie VARCHAR(50),
    kod_pocztowy VARCHAR(10)
);

CREATE TABLE kody_pocztowe (
    kod VARCHAR(10) PRIMARY KEY,
    miasto VARCHAR(50),  -- Tylko tutaj!
    wojewodztwo VARCHAR(50)
);
```

🏆 BCNF - perfekcyjna eliminacja

- Każda zależność funkcyjna musi wychodzić od klucza kandydującego
- "Najczystsza" forma, ale czasem niemożliwa do osiągnięcia

📊 Efektywność normalizacji:

Postać	Redundancja	Praktyczność
1NF	🔴🔴🔴	✅✅✅
2NF	🔴🔴	✅✅✅
3NF	🔴	✅✅
BCNF	✅	✅

⚠️ Kiedy redundancja jest OK:

📊 Tabele raportowe

```
-- Celowa redundancja dla szybkości
CREATE TABLE raport_sprzedazy (
    data DATE,
    klient_nazwa VARCHAR(100),    -- redundancja!
    produkt_nazwa VARCHAR(100),   -- redundancja!
    wartosc DECIMAL(10,2)
);
```

🎯 Cache/Denormalizacja

- Dla wydajności zapytań
- W tabelach tylko do odczytu
- Gdy szybkość > spójność

> 💡 **Zapamiętaj:** Redundancja = "kopia tego samego" = zło! 3NF eliminuje 95% problemów! 🎯

2️⃣6️⃣ Algebra relacji - podstawowe operacje 🧱🔢

Algebra relacji = matematyka dla tabel 🧱 📊 - jak manipulować danymi!

🎯 Operacje jednoargumentowe (na jednej tabeli):

🎂 Selekcja (σ - sigma) - wybieranie wierszy

Co robi: Wybiera wiersze spełniające warunek 🔍

```
σ(wiek > 18)(OSOBY)
↓
Tylko pełnoletnie osoby
```

📝 Projekcja (π - pi) - wybieranie kolumn

Co robi: Wybiera określone kolumny 📋

```
π(imie, nazwisko)(OSOBY)
↓
Tylko imiona i nazwiska (bez reszty)
```

🤝 Operacje dwuargumentowe (na dwóch tabelach):

➕ Unia (∪) - dodawanie

Co robi: Łączy tabele, usuwa duplikaty

```
STUDENCI ∪ PRACOWNICY = wszyscy ludzie (bez powtórek)
```

➖ Różnica (-) - odejmowanie

Co robi: Co jest w pierwszej, ale nie w drugiej

```
WSZYSCY - STUDENCI = tylko nie-studenci
```

∩ Przecięcie (∩) - część wspólna

Co robi: Co jest w obu tabelach

```
STUDENCI ∩ PRACOWNICY = studenci którzy też pracują
```

🔗 Złączenie (⋈) - łączenie przez warunek

Co robi: Łączy tabele na podstawie warunku

```
OSOBY ⋈ ADRESY = osoby ze swoimi adresami
```

> 🎯 **Mnemotechnika:** Selekcja = Selectuj wiersze, Projekcja = Pokaż kolumny! 🤘

2️⃣7️⃣ Zaawansowane SQL - Common Table Expressions (CTE) i Window Functions! 🪟 🚀

CTE i Window Functions = "zaawansowane sztuczki SQL" 🎩 ✨ - dla bardziej złożonych zapytań!

Wyobraź sobie, że podstawowy SQL to rower 🚲 , a CTE i Window Functions to samochód wyścigowy 🏎️ !

🔥 Common Table Expressions (CTE) - "nazwane podzapytania"

🎯 Co to jest CTE:

CTE = tymczasowa tabela, która istnieje tylko w ramach jednego zapytania! 📋

Analogia: Kartka robocza - piszesz na niej obliczenia, używasz, potem wyrzucasz! 📝

💡 Podstawowa składnia:

```
WITH nazwa_cte AS (
    -- Tu jest zapytanie
    SELECT kolumna1, kolumna2
    FROM tabela
    WHERE warunek
)
SELECT * FROM nazwa_cte;
```

🤹 Praktyczny przykład:

```
-- ❌ Bez CTE - trudne do czytania:
SELECT k.nazwa,
       (SELECT AVG(z.wartosc) FROM zamowienia z WHERE z.klient_id = k.id)
as avg_zamowienia
FROM klienci k
WHERE (SELECT COUNT(*) FROM zamowienia z WHERE z.klient_id = k.id) > 5;

-- ✅ Z CTE - czytelne!
WITH statystyki_klientow AS (
    SELECT
        klient_id,
        COUNT(*) as liczba_zamowien,
        AVG(wartosc) as avg_wartosc
    FROM zamowienia
    GROUP BY klient_id
)
SELECT k.nazwa, s.avg_wartosc
FROM klienci k
JOIN statystyki_klientow s ON k.id = s.klient_id
WHERE s.liczba_zamowien > 5;
```

🔄 Recursive CTE - "zapytania rekurencyjne":

```
-- Hierarchie - np. drzewo organizacyjne
WITH RECURSIVE hierarchia AS (
```

```
    -- Anchor: szefowie (brak przełożonego)
    SELECT id, nazwa, przelozony_id, 1 as poziom
    FROM pracownicy
    WHERE przelozony_id IS NULL

    UNION ALL

    -- Recursive: podwładni
    SELECT p.id, p.nazwa, p.przelozony_id, h.poziom + 1
    FROM pracownicy p
    JOIN hierarchia h ON p.przelozony_id = h.id
)
SELECT * FROM hierarchia ORDER BY poziom;
```

🪟 Window Functions - "funkcje okienkowe"

🎯 Co to są Window Functions:

Window Functions = obliczenia na "oknie" rekordów, ale **bez GROUP BY**! 🔍

Analogia: Patrzysz przez okno na sąsiadów - widzisz siebie I sąsiadów jednocześnie! 🏠👀

💡 Podstawowa składnia:

```
SELECT
    kolumna,
    WINDOW_FUNCTION() OVER (PARTITION BY kolumna ORDER BY kolumna)
FROM tabela;
```

🏆 Popularne Window Functions:

1️⃣ ROW_NUMBER() - numerowanie:

```
SELECT
    nazwa,
    pensja,
    ROW_NUMBER() OVER (ORDER BY pensja DESC) as ranking
FROM pracownicy;
```

2️⃣ RANK() - ranking z ex aequo:

```
SELECT
    nazwa,
    pensja,
    RANK() OVER (ORDER BY pensja DESC) as ranking
```

```
    -- Jeśli 2 osoby mają taką samą pensję → ten sam ranking
FROM pracownicy;
```

3️⃣ **LAG/LEAD - poprzedni/następny rekord:**

```
SELECT
    data_zamowienia,
    wartosc,
    LAG(wartosc) OVER (ORDER BY data_zamowienia) as poprzednie_zamowienie,
    LEAD(wartosc) OVER (ORDER BY data_zamowienia) as nastepne_zamowienie
FROM zamowienia;
```

4️⃣ **SUM/AVG z OVER - sumy bieżące:**

```
SELECT
    data_zamowienia,
    wartosc,
    SUM(wartosc) OVER (ORDER BY data_zamowienia) as suma_narastajaco
FROM zamowienia;
```

🤹 **Praktyczny przykład - ranking sprzedaży:**

```
WITH sprzedaz_miesiac AS (
    SELECT
        sprzedawca,
        MONTH(data_zamowienia) as miesiac,
        SUM(wartosc) as sprzedaz_miesiac
    FROM zamowienia
    GROUP BY sprzedawca, MONTH(data_zamowienia)
)
SELECT
    sprzedawca,
    miesiac,
    sprzedaz_miesiac,
    RANK() OVER (PARTITION BY miesiac ORDER BY sprzedaz_miesiac DESC) as
ranking_w_miesiacu,
    AVG(sprzedaz_miesiac) OVER (PARTITION BY sprzedawca) as
srednia_sprzedawcy
FROM sprzedaz_miesiac;
```

🎯 Kiedy używać:

✅ CTE używaj gdy:

- Zapytanie staje się skomplikowane
- Chcesz uniknąć duplikacji podzapytań
- Potrzebujesz hierarchii (recursive CTE)
- Chcesz poprawić czytelność kodu

✅ **Window Functions używaj gdy:**

- Potrzebujesz rankingu
- Chcesz porównać rekord z poprzednim/następnym
- Potrzebujesz sum narastających
- Chcesz analizować trendy

📚 Analogia - okno w budynku:

GROUP BY: Dzielisz ludzi na pokoje, liczysz osoby w każdym pokoju **Window Functions:** Patrzysz przez okno - widzisz siebie I sąsiadów jednocześnie

> 💡 **Zapamiętaj:** CTE = "kartka robocza", Window Functions = "patrzenie przez okno"! 🪟

> 🎯 **Dla zaawansowanych:** Te funkcje pojawiają się w trudniejszych zadaniach - warto znać! 🚀

2️⃣8️⃣ Opisz w jaki sposób w SQL realizowane są załączenia (JOIN)! 🔗

JOINy = "łączenie tabel" 🧩 - sposób na zebranie danych z wielu tabel w jedno miejsce!

Wyobraź sobie puzzle 🧩 : jedna część to klienci, druga to zamówienia - JOIN składa je w całość!

🎯 Rodzaje JOINów:

1️⃣ INNER JOIN - "przecięcie" ∩

Co robi: Pokazuje TYLKO te rekordy, które mają pary w obu tabelach!

```
SELECT k.nazwa, z.data_zamowienia, z.wartosc
FROM klienci k
INNER JOIN zamowienia z ON k.id = z.klient_id;
```

🎭 **Analogia:** Impreza tylko dla par - przychodzą tylko ci, którzy mają partnera! 👫

Diagram:

```
KLIENCI:        ZAMÓWIENIA:       WYNIK:
Jan (ID=1)      Zamów A (klient=1)  → Jan + Zamów A
Anna (ID=2)     Zamów B (klient=1)  → Jan + Zamów B
Piotr (ID=3)    Zamów C (klient=2)  → Anna + Zamów C
                                   Piotr = BEZ ZAMÓWIEŃ → NIE POKAZANY
```

2️⃣ LEFT JOIN - "wszystko z lewej" ←

Co robi: Pokazuje WSZYSTKIE rekordy z lewej tabeli + pasujące z prawej!

```
SELECT k.nazwa, z.data_zamowienia, z.wartosc
FROM klienci k
LEFT JOIN zamowienia z ON k.id = z.klient_id;
```

🤹 **Analogia:** Lista wszystkich studentów + ich oceny (studenci bez ocen też się liczą!) 📚

Diagram:

```
KLIENCI:        ZAMÓWIENIA:       WYNIK:
Jan (ID=1)      Zamów A (klient=1)  → Jan + Zamów A
Anna (ID=2)     Zamów B (klient=1)  → Jan + Zamów B
Piotr (ID=3)    Zamów C (klient=2)  → Anna + Zamów C
                                    → Piotr + NULL (bez zamówień)
```

3️⃣ RIGHT JOIN - "wszystko z prawej" →

Co robi: Pokazuje WSZYSTKIE rekordy z prawej tabeli + pasujące z lewej!

```
SELECT k.nazwa, z.data_zamowienia, z.wartosc
FROM klienci k
RIGHT JOIN zamowienia z ON k.id = z.klient_id;
```

🤹 **Analogia:** Lista wszystkich zamówień + dane klientów (zamówienia bez klientów też się liczą!) 📦

4️⃣ FULL OUTER JOIN - "wszystko" ∪

Co robi: Pokazuje WSZYSTKIE rekordy z obu tabel!

```
SELECT k.nazwa, z.data_zamowienia, z.wartosc
FROM klienci k
FULL OUTER JOIN zamowienia z ON k.id = z.klient_id;
```

🤹 **Analogia:** Kompletna lista - wszyscy klienci I wszystkie zamówienia! 📋

🎯 Praktyczne przykłady:

📊 Znajdź klientów bez zamówień:

```

```

```sql
-- LEFT JOIN + WHERE NULL
SELECT k.nazwa
FROM klienci k
LEFT JOIN zamowienia z ON k.id = z.klient_id
WHERE z.klient_id IS NULL;
```

💰 **Pokaż sumy zamówień dla każdego klienta:**

```sql
SELECT
    k.nazwa,
    COUNT(z.id) as liczba_zamowien,
    COALESCE(SUM(z.wartosc), 0) as suma_zamowien
FROM klienci k
LEFT JOIN zamowienia z ON k.id = z.klient_id
GROUP BY k.id, k.nazwa;
```

🔗 **Łączenie 3 tabel:**

```sql
SELECT
    k.nazwa as klient,
    z.data_zamowienia,
    p.nazwa as produkt,
    zp.ilosc
FROM klienci k
JOIN zamowienia z ON k.id = z.klient_id
JOIN zamowienia_produkty zp ON z.id = zp.zamowienie_id
JOIN produkty p ON zp.produkt_id = p.id;
```

🧠 Jak zapamiętać:

🎯 Mnemotechnika:

- **INNER** = "INside" - tylko to co jest w środku (przecięcie)
- **LEFT** = "wszystko z LEft" - lewa tabela kompletna
- **RIGHT** = "wszystko z RIGht" - prawa tabela kompletna
- **FULL** = "FULL list" - wszystko ze wszystkich tabel

🎭 Analogia - restauracja:

- **INNER JOIN:** Tylko stoliki z gośćmi (zajęte)
- **LEFT JOIN:** Wszystkie stoliki + goście (zajęte + wolne)
- **RIGHT JOIN:** Wszyscy goście + stoliki (z miejscem + bez miejsca)
- **FULL JOIN:** Wszystkie stoliki I wszyscy goście

🚨 Częste błędy:

❌ **Brak ON - produkt kartezjański:**

```
-- ŹLE! Każdy klient z każdym zamówieniem
SELECT * FROM klienci, zamowienia;  -- 1000 x 1000 = 1 000 000 rekordów!

-- DOBRZE! Tylko pasujące pary
SELECT * FROM klienci k JOIN zamowienia z ON k.id = z.klient_id;
```

❌ **Mylenie LEFT z RIGHT:**

```
-- Te zapytania RÓŻNIĄ SIĘ!
FROM klienci LEFT JOIN zamowienia    -- Wszyscy klienci
FROM zamowienia LEFT JOIN klienci    -- Wszystkie zamówienia
```

> 💡 **Zapamiętaj:** JOIN = "składanie puzzli" - każdy typ składa inaczej! 🧩

> 🎯 **Praktyczna rada:** LEFT JOIN używaj najczęściej - pokazuje "główną" tabelę kompletnie! ⭐

2️⃣9️⃣ Klucze w bazach danych - główny, kandydujący, obcy 🗝️🔐

Klucze = "identyfikatory" 🆔 - sposoby na rozpoznanie każdego rekordu w tabeli!

🏆 Klucz główny (PRIMARY KEY)

Klucz główny = "główny identyfikator" 👑 - unikalnie identyfikuje każdy rekord!

✅ **Właściwości klucza głównego:**

- 🆔 **Unikalny** - nie ma duplikatów w całej tabeli
- ❌ **NIE MOŻE być NULL** - zawsze musi mieć wartość
- 🔒 **Niezmienny** - raz ustawiony, raczej się nie zmienia
- 📍 **Jeden na tabelę** - może być tylko jeden PRIMARY KEY

💡 **Przykłady:**

```
-- Prosty klucz główny
CREATE TABLE klienci (
    id INT PRIMARY KEY,          -- ✅ Klucz główny
    imie VARCHAR(50),
    email VARCHAR(100)
);

-- Klucz główny złożony (composite key)
CREATE TABLE student_przedmiot (
    student_id INT,
    przedmiot_id INT,
```

```
    ocena DECIMAL(2,1),
    PRIMARY KEY (student_id, przedmiot_id)  -- ✅ Klucz złożony
);
```

🎯 Klucz kandydujący (CANDIDATE KEY)

Klucz kandydujący = "potencjalny klucz główny" 🤹 - może być ich kilka, ale tylko jeden zostanie wybrany!

🎯 Warunki klucza kandydującego:

1. 🆔 **Unikalność** - jednoznacznie identyfikuje każdy rekord
2. 🎯 **Minimalność** - nie zawiera zbędnych kolumn

💡 Przykład:

```
CREATE TABLE pracownicy (
    id INT,                     -- ✅ Kandydat 1
    pesel VARCHAR(11),          -- ✅ Kandydat 2 (unikalny)
    email VARCHAR(100),         -- ✅ Kandydat 3 (unikalny w firmie)
    nr_pracowniczy VARCHAR(10), -- ✅ Kandydat 4
    imie VARCHAR(50),           -- ❌ Nie kandydat (nie unikalny)

    PRIMARY KEY (id),           -- Wybrany główny
    UNIQUE (pesel),             -- Klucz alternatywny
    UNIQUE (email),             -- Klucz alternatywny
    UNIQUE (nr_pracowniczy)     -- Klucz alternatywny
);
```

🔗 Klucz obcy (FOREIGN KEY)

Klucz obcy = "wskaźnik na inne miejsce" 👉 - odnosi się do klucza głównego w innej tabeli!

✅ Właściwości klucza obcego:

- 🔗 **Odnosi się** do PRIMARY KEY w innej tabeli
- ✅ **MOŻE być NULL** - oznacza "nie ma powiązania"
- 🔄 **Może się powtarzać** - wiele rekordów może wskazywać na to samo
- 📊 **Wiele na tabelę** - może być kilka kluczy obcych

💡 Przykład:

```
CREATE TABLE zamowienia (
    id INT PRIMARY KEY,
    klient_id INT,                  -- ✅ Może być NULL
    produkt_id INT NOT NULL,        -- ❌ Nie może być NULL
    ilosc INT,
```

```
    FOREIGN KEY (klient_id) REFERENCES klienci(id),
    FOREIGN KEY (produkt_id) REFERENCES produkty(id)
);
```

📊 Porównanie kluczy:

Właściwość	🏆 PRIMARY	🎯 CANDIDATE	🔗 FOREIGN
Unikalny	✅ TAK	✅ TAK	❌ NIE
Może być NULL	❌ NIE	❌ NIE	✅ TAK
Ilość na tabelę	1️⃣ Jeden	🔢 Wiele	🔢 Wiele
Cel	Identyfikacja	Potencjalna ID	Powiązania

🎭 Klucze alternatywne

Klucz alternatywny = klucz kandydujący który NIE został wybrany jako główny

✅ Może być NULL?

- **Teoretycznie:** NIE (to klucz kandydujący!)
- **Praktycznie:** TAK (w niektórych bazach danych)
- **Najlepiej:** Unikaj NULL w kluczach alternatywnych! 🎯

💡 Praktyczne rady:

🎯 Wybór klucza głównego:

```
-- ✅ DOBRZE: Prosty INT z AUTO_INCREMENT
id INT AUTO_INCREMENT PRIMARY KEY

-- ✅ OK: UUID dla globalnej unikalności
id VARCHAR(36) PRIMARY KEY  -- UUID

-- ⚠️ OSTROŻNIE: Naturalne klucze
pesel VARCHAR(11) PRIMARY KEY  -- może się zmienić!

-- ❌ ŹLE: Kompozytowy bez potrzeby
PRIMARY KEY (imie, nazwisko, data_urodzenia)  -- za skomplikowane!
```

🔗 Klucze obce:

```
-- ✅ DOBRZE: Z akcją referencyjną
FOREIGN KEY (klient_id) REFERENCES klienci(id) ON DELETE CASCADE
```

```
-- ✅ OK: Może być NULL
klient_id INT NULL  -- "zamówienie bez przypisanego klienta"

-- ❌ ŹLE: Bez ograniczeń referencyjnych
klient_id INT  -- brak FOREIGN KEY = możliwe "wiszące" referencje
```

> 💡 **Zapamiętaj:** PRIMARY = nigdy NULL, FOREIGN = może NULL, CANDIDATE = powinien nie być NULL! 🗝️

3️⃣0️⃣ Jakie są poziomy izolacji transakcji? 🏝️🔐

Poziomy izolacji = jak bardzo transakcje są odizolowane od siebie 🏝️

Wyobraź sobie sali egzaminacyjnej 📝 - różne poziomy "ściągania":

📊 4 Poziomy izolacji (od najmniej do najbardziej bezpiecznego):

🚨 1. Read Uncommitted (Totalna anarchia!)

- **Problem:** Można czytać "brudne" dane (nie zatwierdzone) 💩
- **Analogia:** Patrzysz na kartkę kolegi zanim skończy pisać ✍️
- **Skutek:** Dirty read - możesz przeczytać coś co za chwilę zniknie!

✅ 2. Read Committed (Podstawowa przyzwoitość)

- **Zasada:** Czytasz tylko zatwierdzone dane
- **Analogia:** Patrzysz tylko na oddane kartki 📋
- **Nadal problem:** Non-repeatable read - między odczytami dane mogą się zmienić

🔒 3. Repeatable Read (Stabilność)

- **Zasada:** Raz przeczytane dane nie zmienią się podczas transakcji
- **Analogia:** Kartka kolegi jest "zamrożona" podczas twojego egzaminu ❄️
- **Nadal problem:** Phantom reads - nowe rekordy mogą się pojawić

🏛️ 4. Serializable (Totalnie bezpieczny)

- **Zasada:** Transakcje wykonują się jakby były jedną po drugiej
- **Analogia:** Każdy pisze egzamin w osobnej sali 🚪
- **Skutek:** Najwolniejszy, ale najsafiejszy! 🐌

> 💡 **Zapamiętaj:** Im wyższy poziom = bardziej bezpieczny ale wolniejszy! ⚖️

3️⃣1️⃣ Jakie warunki muszą spełniać tabele aby skutecznie użyć NATURAL JOIN? 🔗🤝

NATURAL JOIN = "automatyczny" join 🤖 - sam znajduje wspólne kolumny!

✅ Warunki skutecznego użycia:

1️⃣ Identyczne nazwy kolumn 👄

- Musi być conajmniej jedna kolumna o tej samej nazwie
- **Bez wspólnych nazw** = iloczyn kartezjański (katastrofa!) 💥
- System automatycznie szuka kolumn o identycznych nazwach

2️⃣ Te same typy danych 🔢📝

- `id INT` w jednej tabeli ≠ `id VARCHAR` w drugiej
- System musi umieć porównać wartości!

3️⃣ Logiczna zgodność danych 🧠✨

```
-- ❌ PUŁAPKA! Obie tabele mają "id" ale oznacza co innego:
Tabela KLIENCI: id (id klienta)
Tabela PRODUKTY: id (id produktu)
-- NATURAL JOIN połączy przez "id" = nonsens!

-- ✅ DOBRZE! Wspólna kolumna oznacza to samo:
Tabela KLIENCI: klient_id
Tabela ZAMOWIENIA: klient_id (odnosi się do tego samego!)
```

⚠️ Pułapki NATURAL JOIN:

- **Niekontrolowany** - nie wiesz przez co łączy! 🤷‍♀️
- **Niebezpieczny** - zmiana nazwy kolumny = zmiana wyniku 💀
- **Lepsze rozwiązanie:** Zawsze używaj `INNER JOIN ... ON ...` 🎯

> 🎯 **Rada na egzamin:** Powiedz, że NATURAL JOIN jest "leniwy" ale niebezpieczny!

3️⃣2️⃣ Blokady w bazach danych - "zajęte, wolne" system! 🔒🚦

Blokady = "znaki zajęte/wolne" 🚦 - kontrolują kto i kiedy może dostać się do danych!

Wyobraź sobie toaletę publiczną 🚻: gdy ktoś jest w środku, zamyka drzwi (blokada) - inni muszą czekać!

🎯 Po co są blokady:

🛡️ Ochrona przed chaosem:

- **Współbieżność** - gdy 1000 osób jednocześnie chce zmienić ten sam rekord 🏃‍♂️🏃‍♀️
- **Spójność danych** - żeby nie było "schizofrenicznych" wartości 🤪
- **Zapobieganie anomaliom**: dirty read, lost update, phantom read 👻

🎭 Analogia - biblioteka:

Bez blokad: 10 osób bierze tę samą książkę → chaos! 📚💥 **Z blokadami:** Kolejka, książka zajęta → porządek! 📚✅

🔒 Rodzaje blokad (Lock Modes):

📖 S - Shared (Dzielona) - "Czytanie razem"

```
-- Wielu może czytać jednocześnie
SELECT * FROM produkty WHERE kategoria = 'Elektronika';
```

🎭 **Analogia:** Biblioteka - wielu może czytać tę samą książkę (fotokopie) 📚👥

✏️ X - Exclusive (Wyłączna) - "Sam na sam"

```
-- Tylko jeden może pisać
UPDATE produkty SET cena = cena * 1.1 WHERE kategoria = 'Elektronika';
```

🎭 **Analogia:** Tablica w klasie - tylko jeden może pisać, reszta czeka 📝✋

👀 IS - Intent Shared - "Zamierzam czytać"

Co robi: "Hej, planuję czytać jakiś wiersz w tej tabeli!" **Kiedy:** Przed założeniem S na konkretny wiersz

✍️ IX - Intent Exclusive - "Zamierzam pisać"

Co robi: "Hej, planuję zmieniać jakiś wiersz w tej tabeli!" **Kiedy:** Przed założeniem X na konkretny wiersz

🔀 SIX - Shared + Intent Exclusive - "Czytam wszystko, zmieniam część"

Co robi: "Czytam całą tabelę, ale zamierzam zmienić kilka wierszy" **Kiedy:** SELECT + UPDATE w jednej transakcji

🏗️ Hierarchia blokowania:

```
🏢 BAZA DANYCH
   ├── 📋 TABELA (blokada IS/IX/S/X)
       ├── 📄 STRONA (blokada IS/IX/S/X)
           ├── 📝 WIERSZ (blokada S/X)
```

🎯 **Zasada:** Aby zablokować wiersz, musisz "zapowiedzieć" na wyższych poziomach!

🔥 Macierz kompatybilności - "kto z kim może":

✅ **= Można razem,** ❌ **= Konflikt**

	📖 S	✏️ X	👀 IS	✍️ IX	🔀 SIX
📖 **S**	✅ Wielu czyta	❌ Czyta vs pisze	✅	❌	❌

	📖 S	✏️ X	👀 IS	✍️ IX	🔀 SIX
✏️ X	❌	❌ Tylko jeden!	❌	❌	❌
👀 IS	✅	❌	✅ Zapowiedzi OK	✅	✅
✍️ IX	❌	❌	✅	✅	❌
🔀 SIX	❌	❌	✅	❌	❌

🎯 Praktyczne przykłady:

💫 Scenariusz 1: Dwie osoby czytają

```
-- Transakcja A:
SELECT * FROM klienci WHERE miasto = 'Warszawa';  -- Blokada S ✅

-- Transakcja B (równocześnie):
SELECT * FROM klienci WHERE miasto = 'Kraków';    -- Blokada S ✅
```

Wynik: Obie mogą czytać jednocześnie! 📖 📖

💥 Scenariusz 2: Jeden czyta, drugi pisze

```
-- Transakcja A:
SELECT * FROM klienci WHERE miasto = 'Warszawa';  -- Blokada S

-- Transakcja B (równocześnie):
UPDATE klienci SET telefon = '123' WHERE id = 1;  -- Chce X → CZEKA! ⏳
```

Wynik: B musi czekać aż A skończy! 😴

🚀 Scenariusz 3: Intent locks w akcji

```
-- Transakcja A chce zmienić jeden wiersz:
-- 1. IX na tabelę (zapowiadam pisanie)
-- 2. X na wiersz id=1 (pisuję konkretny wiersz)
UPDATE klienci SET telefon = '123' WHERE id = 1;

-- Transakcja B chce czytać całą tabelę:
-- 1. Chce S na tabelę
-- 2. Sprawdza czy IX koliduje z S → TAK! → CZEKA ⏳
SELECT * FROM klienci;
```

🧠 Jak zapamiętać:

🎯 Mnemotechnika:

- **S** = **S**hared = **S**połem czytamy 📚👥
- **X** = e**X**clusive = e**X**tremalnie sam! 🏝️
- **IS** = **I**ntent **S**hared = **I**nformuję że będę czytać 👀
- **IX** = **I**ntent e**X**clusive = **I**nformuję że będę pisać ✍️

🎭 Analogia - parking:

- **S lock:** Parking - wielu może parkować ✅
- **X lock:** Brama - tylko jeden może przejechać ❌
- **IS lock:** "Zamierzam parkować" - sygnalizacja 🚗💭
- **IX lock:** "Zamierzam wyjechać" - sygnalizacja 🚗➡️

🚨 Częste problemy:

⚠️ Lock escalation (eskalacja blokad):

```
Zbyt dużo blokad na wierszach → system zmienia na blokadę całej tabeli
Result: Gorsza wydajność! 📉
```

⚠️ Lock wait timeout:

```
-- Gdy transakcja czeka za długo:
ERROR: Lock wait timeout exceeded; try restarting transaction
```

> 💡 **Zapamiętaj:** Blokady = "system kolejki" - każdy czeka na swoją kolej! 🚦

> 🎯 **Praktyczna rada:** Im krótsze transakcje, tym mniej problemów z blokadami! ⚡

3️⃣3️⃣ Protokół blokowania dwufazowego - "zasady grzecznego dzielenia"! 🔒⚖️

Protokół dwufazowy (2PL) = "instrukcja obsługi blokad" 📋 - jak grzecznie pożyczać i oddawać zabawki w przedszkolu!

Wyobraź sobie, że masz skrzynkę z zabawkami 🧸 - są zasady kiedy możesz brać, a kiedy musisz oddawać!

🎯 Idea podstawowa:

📈 Faza 1: Rozszerzanie (Growing) - "Biję zabawki"

```
🔒 Zakładam blokady, zakładam blokady...
🔒 Nie zwalniam jeszcze ŻADNEJ!
```

```
🔒 Tylko biore, nie oddaję!
```

📉 Faza 2: Kurczenie (Shrinking) - "Oddaję zabawki"

```
🔒 Zwalniam blokady, zwalniam blokady...
🔒 NIGDY już nie biorę nowych!
🔒 Tylko oddaję, nie biore!
```

🚨 ZŁOTA ZASADA 2PL:

"Jak zaczynasz oddawać zabawki, nie możesz już brać nowych!" 🎯

🎭 Analogia - wypożyczalnia:

👤 Klient w wypożyczalni filmów:

1. 📈 **Faza Growing:** Biore film A, film B, film C... (tylko wypożyczam)
2. 📉 **Faza Shrinking:** Oddaję film A, film B, film C... (tylko zwracam)

❌ **BŁĄD:** Oddałem film A, ale potem chcę wypożyczyć film D → ZABRONIONE!

💡 Praktyczny przykład - przelew bankowy:

```
-- Transakcja: Przelej 100zł z konta A na konto B

-- 📈 FAZA GROWING:
LOCK(konto_A, X);           -- Biore blokadę na konto A
LOCK(konto_B, X);           -- Biore blokadę na konto B
                            -- Koniec fazy growing!

-- 💼 PRACA:
A.saldo = A.saldo - 100;    -- Odejmuje z A
B.saldo = B.saldo + 100;    -- Dodaje do B

-- 📉 FAZA SHRINKING:
UNLOCK(konto_A);            -- Zwalniam blokadę A
UNLOCK(konto_B);            -- Zwalniam blokadę B
                            -- Koniec fazy shrinking!
```

🛡️ Gwarantowana właściwość - SZEREGOWALNOŚĆ:

Szeregowalność = wynik jest taki sam, jakby transakcje działały jedna po drugiej! 🎯

🎭 Analogia - przepisy w kuchni:

Bez 2PL: Dwóch kucharzy miesza składniki na raz → zupa-masakra! 🍲💥 **Z 2PL:** Kolejno: pierwszy gotuje, drugi gotuje → każda zupa idealna! ✅

🔥 Odmiany protokołu 2PL:

1️⃣ Basic 2PL (Podstawowy)

```
📈 Growing: Biore blokady
📉 Shrinking: Zwalniam blokady po kolei
```

Problem: Można zwolnić za wcześnie → inne transakcje widzą nieskończone zmiany! 😱

2️⃣ Strict 2PL (Ścisły) - NAJLEPSZY! ⭐

```
📈 Growing: Biore blokady
💼 Work: Działam...
📉 Shrinking: Zwalniam WSZYSTKIE na samym końcu (przy COMMIT)
```

Zaleta: Żadna transakcja nie widzi "niedokończonej roboty"! ✅

3️⃣ Static 2PL (Statyczny)

```
📋 Pre-analysis: Z góry wiem jakie blokady potrzebuję
📈 Growing: Biore WSZYSTKIE blokady na raz
💼 Work: Działam...
📉 Shrinking: Zwalniam WSZYSTKIE na raz
```

Zaleta: Brak deadlocków! **Wada:** Trudne do implementacji

4️⃣ Conservative 2PL (Konserwatywny)

```
🛑 Wait: Czekam aż WSZYSTKIE potrzebne blokady będą dostępne
📈 Growing: Biore wszystkie na raz
💼 Work: Działam...
📉 Shrinking: Zwalniam wszystkie na raz
```

Zaleta: Brak deadlocków! **Wada:** Długie czekanie

🚨 Problemy z 2PL:

⚠️ Deadlock - nawzajemne czekanie:

```
Transakcja A: LOCK(X) → chce LOCK(Y)
Transakcja B: LOCK(Y) → chce LOCK(X)
→ Deadlock! 💀
```

⚠ Cascading abort - efekt domina:

```
T1 zmienia dane → T2 czyta te dane → T1 się wycofuje
→ T2 też musi się wycofać! 🏗️💥
```

🎯 Praktyczne porady:

✅ Jak unikać problemów:

1. **Używaj Strict 2PL** - najczęściej wystarczy
2. **Krótkie transakcje** - mniej czasu na konflikty
3. **Kolejność blokowania** - zawsze w tej samej kolejności
4. **Timeout** - nie czekaj w nieskończoność

🧠 Jak zapamiętać:

📈📉 **= "Góra-Dół"** - najpierw w górę (biore), potem w dół (oddaję)

📚 Analogia - biblioteka:

Growing: Wypożyczam książki jedna po drugiej 📚📚📚 **Shrinking:** Oddaję książki jedna po drugiej 📚 → 📚 →📚 **Zasada:** Jak zacznę oddawać, nie mogę już wypożyczać!

> 💡 **Zapamiętaj:** 2PL = "najpierw biore, potem oddaję" - bez mieszania! 🎯

> 🏆 **Na egzaminie:** Strict 2PL to standard - używany przez większość baz danych! ⭐

3️⃣4️⃣ 3NF vs BCNF - "dobra vs idealna" normalizacja! 🎯🏆

Pytanie: Do jakiej postaci normalnej zawsze można doprowadzić bez strat? 🤔

Odpowiedź: Do **3NF** zawsze można ✅ , **BCNF** czasami się nie da ❌

🎯 Podstawowe różnice:

😊 3NF - "Wystarczająco dobra"

```
✅ Zawsze osiągalna
✅ Zachowuje WSZYSTKIE zależności funkcyjne
✅ Usuwa większość redundancji
✅ Praktyczna w życiu
```

🏆 BCNF - "Idealna, ale trudna"

```
❓ Czasami nieosiągalna
❌ Może stracić niektóre zależności funkcyjne
✅ Usuwa CAŁĄ redundancję
😰 Czasami niepraktyczna
```

🎭 Analogia - sprzątanie pokoju:

🧹 3NF = "Pokój porządny"

- Większość rzeczy ma swoje miejsce
- Czasami coś leży nie tam gdzie powinno, ale ogólnie OK
- **Nadaje się do życia!** ✅

🏆 BCNF = "Pokój idealny"

- Wszystko w idealnych miejscach
- Czasami trzeba wyrzucić przydatne rzeczy, żeby było idealnie
- **Piękne, ale czy praktyczne?** 🤔

💡 Dlaczego 3NF zawsze można osiągnąć:

🔄 Algorytm syntezy dla 3NF:

1. **Krok 1:** Weź wszystkie zależności funkcyjne
2. **Krok 2:** Dla każdej zależności A → B stwórz tabelę (A, B)
3. **Krok 3:** Dodaj tabelę z kluczem kandydującym jeśli potrzeba
4. **Wynik:** Zawsze 3NF + wszystkie zależności zachowane! ✅

💥 Kiedy BCNF się nie da - przykład:

🎓 Problem z kursami uniwersyteckimi:

```
KURSY(Student, Wykładowca, Przedmiot)

Zależności funkcyjne:
1. Student + Przedmiot → Wykładowca  (student ma jednego wykładowcę na przedmiot)
2. Wykładowca → Przedmiot             (wykładowca uczy jednego przedmiotu)
```

🔥 Dlaczego BCNF nie działa:

🎯 **Problem:** Wykładowca nie jest super-kluczem, ale determinuje Przedmiot!

❌ Próba BCNF:

```
-- Tabela 1: WYKLADOWCY(Wykładowca, Przedmiot)
-- Tabela 2: KURSY_STUDENTOW(Student, Wykładowca)
```

💀 **Strata:** Stracimy zależność "Student + Przedmiot → Wykładowca"!

🤔 **Nie możemy już sprawdzić:** "Czy student X studiuje przedmiot Y u wykładowcy Z?"

✅ **3NF działa idealnie:**

```
-- Zostaw oryginalną tabelę - jest w 3NF!
KURSY(Student, Wykładowca, Przedmiot)
```

Zachowane: Wszystkie zależności funkcyjne! ✅

🎯 Praktyczne rozwiązanie:

🏆 **Złota zasada:**

"Zatrzymaj się na 3NF!" 🛑

📊 **Statystyki z praktyki:**

- **3NF używa:** 95% projektów baz danych ✅
- **BCNF używa:** 5% projektów (gdy naprawdę trzeba) 🔬
- **Problemy z BCNF:** Strata zależności = strata logiki biznesowej! 💼

🔥 Kiedy mimo wszystko iść w BCNF:

✅ **Można bezpiecznie, gdy:**

- Nie tracimy żadnych zależności funkcyjnych
- Dane są bardzo redundantne w 3NF
- Oszczędność miejsca jest krytyczna

❌ **NIE idź w BCNF, gdy:**

- Stracisz ważne reguły biznesowe
- 3NF już dobrze działa
- Zespół nie zna zaawansowanej teorii normalizacji

🧠 Jak to zapamiętać:

🎯 **Mnemotechnika:**

- **3NF** = **T**rzecia **N**ormalna **F**orma = **T**utaj **N**a pewno **F**inalnie! ✅
- **BCNF** = **B**oyce **C**odd **N**ormal **F**orm = **B**ardzo **C**iekawa, ale **N**ie zawsze **F**ajnie! 😅

📚 Analogia - gotowanie:

3NF: Smaczny obiad - ma wszystkie składniki, może nie idealny, ale jadalny! 🍽️✅ **BCNF:** Molekularna gastronomia - idealnie, ale trzeba wyrzucić składniki! 🧪❓

🎯 Na egzaminie powiedz:

> **"3NF zawsze można osiągnąć algorytmem syntezy zachowując wszystkie zależności funkcyjne. BCNF czasami wymaga poświęcenia zależności, więc w praktyce zatrzymujemy się na 3NF."** 🎓

> 💡 **Zapamiętaj:** 3NF = gwarancja sukcesu, BCNF = marzenie o ideale! 🎯

> 🏆 **Praktyczna rada:** W 95% przypadków 3NF wystarczy! ⭐

3️⃣5️⃣ Klucze kandydujące - "kandydaci na szefa tabeli"! 🗝️👑

Klucze kandydujące = "pretendenci do tronu" 👑 - każdy z nich może zostać kluczem głównym!

Wyobraź sobie wybory prezydenta 🗳️: jest wielu kandydatów, ale tylko jeden zostanie prezydentem!

🎯 Warunki klucza kandydującego:

1️⃣ 🆔 Unikalność - "Każdy inny"

```
-- W tabeli PRACOWNICY:
PESEL - unikalny ✅
Email - unikalny ✅
Nazwisko - NIE unikalny ❌ (może być dwóch Kowalskich)
```

Zasada: Żadne dwa rekordy nie mogą mieć takiej samej wartości!

2️⃣ ✂️ Minimalność - "Ani atrybutu więcej"

```
-- Przykład:
{PESEL} - kandydat ✅ (sam wystarcza)
{PESEL, Imię} - NIE kandydat ❌ (Imię jest zbędne)
{Imię, Nazwisko, Data_urodzenia} - kandydat ✅ (wszystkie potrzebne)
```

Zasada: Nie można usunąć żadnego atrybutu, bo przestanie być unikalny!

👑 Klucz główny (PRIMARY KEY):

🏆 "Wybrany kandydat"

```
-- Z kandydatów: {PESEL}, {Email}, {Nr_pracownika}
-- Wybieramy jednego jako PRIMARY KEY:
```

```
CREATE TABLE pracownicy (
    pesel VARCHAR(11) PRIMARY KEY,  -- ← Wybrany kandydat!
    email VARCHAR(100) UNIQUE,      -- ← Klucz alternatywny
    nr_pracownika INT UNIQUE        -- ← Klucz alternatywny
);
```

🔑 **Właściwości PRIMARY KEY:**

- 🚫 **NIGDY NULL** - prezydent musi istnieć!
- 🆔 **Zawsze unikalny** - jeden rekord = jeden klucz
- 🔒 **Najważniejszy** - przez niego identyfikujemy rekord

🤔 Czy klucze mogą być NULL?

❌ PRIMARY KEY → NIGDY NULL!

```
INSERT INTO pracownicy (pesel, imie) VALUES (NULL, 'Jan');
-- ERROR: Column 'pesel' cannot be null
```

Dlaczego: Jak znajdziesz rekord bez klucza? 🤷‍♀️

🟡 Klucz alternatywny → MOŻE, ale NIE POWINIEN!

```
-- Teoretycznie można:
INSERT INTO pracownicy (pesel, email) VALUES ('12345678901', NULL);
-- ✅ Przejdzie

-- Ale lepiej nie:
INSERT INTO pracownicy (pesel, email) VALUES ('12345678902',
'jan@firma.pl');
-- ✅ Lepiej!
```

🎭 Analogie dla łatwego zapamiętania:

🏫 Analogia - szkoła:

Kandydaci na klucz:

- **Numer w dzienniku** 📖 - unikalny w klasie
- **PESEL** 🆔 - unikalny w kraju
- **Email** 📧 - unikalny w systemie

PRIMARY KEY: Wybieramy numer w dzienniku (prosty i praktyczny) **Alternatywne:** PESEL i email (backup identyfikatory)

🏪 Analogia - sklep:

Kandydaci na klucz produktu:

- **Kod kreskowy** 📊 - unikalny globalnie
- **Numer seryjny** 🔢 - unikalny od producenta
- **SKU sklepu** 🏪 - unikalny w sklepie

🧠 Jak zapamiętać:

🎯 Mnemotechnika:

- **Kandydat** = **K**onkretny **A**tryb **N**ie **D**uplikuje **Y**dnych **D**anych **A**utentycznych **T**ustępnie!
- **PRIMARY** = **P**odstawowy **R**ekord **I**dentyfikuje **M**oją **A**plikację **R**eliably **Y**es!

📚 Złote zasady:

1. **Unikalny** - nie ma dwóch takich samych
2. **Minimalny** - ani atrybutu za dużo
3. **PRIMARY nie może być NULL** - zawsze musi istnieć
4. **Alternatywny może być NULL** - ale lepiej żeby nie był

🎯 Praktyczne przykłady:

💼 Tabela KLIENCI:

```
-- Kandydaci:
{email}           -- unikalny, minimalny ✅
{nr_telefonu}     -- unikalny, minimalny ✅
{imie, nazwisko, adres}  -- razem unikalny ✅

-- Wybór PRIMARY KEY:
email PRIMARY KEY  -- najlepszy (stabilny, nie zmienia się)
```

🏬 Tabela ZAMÓWIENIA:

```
-- Kandydaci:
{id_zamowienia}   -- sztuczny klucz ✅
{data, klient_id, nr_faktury}  -- naturalny klucz ✅

-- Wybór PRIMARY KEY:
id_zamowienia     -- prostsze i szybsze
```

> 💡 **Zapamiętaj:** Klucz kandydujący = "może zostać szefem", PRIMARY KEY = "został szefem"! 👑

> 🏆 **Praktyczna rada:** PRIMARY KEY nigdy NULL, alternatywny może ale nie powinien! 🎯

3️⃣6️⃣ Procedury składowane - "programy w bazie danych"! 🛠️💾

Procedura składowana = "mini-program" 🤖 zapisany w bazie danych - możesz go uruchomić kiedy chcesz!

Wyobraź sobie makro w Excelu 📊 : nagrywasz sekwencję operacji, potem uruchamiasz jednym kliknięciem!

🎯 Co to jest procedura:

🤖 "Robot w bazie danych"

```
-- Zamiast pisać za każdym razem:
SELECT COUNT(*) FROM zamowienia WHERE data > '2024-01-01';
UPDATE statystyki SET total_zamowien = X WHERE miesiac = 'styczen';
INSERT INTO logi VALUES ('Zaktualizowano statystyki', NOW());

-- Można napisać procedurę:
CALL aktualizuj_statystyki('2024-01-01');
```

🛠️ Co może robić procedura:

📊 Zapytania SQL

```
-- Wszystkie podstawowe operacje:
SELECT, INSERT, UPDATE, DELETE
```

🧠 Logika programistyczna

```
-- Warunki i pętle:
IF (warunek) THEN ... END IF;
WHILE (warunek) DO ... END WHILE;
CASE WHEN ... THEN ... END CASE;
```

🔄 Parametry i wyniki

```
-- Może przyjmować dane i zwracać rezultaty
IN parametr_wejściowy
OUT parametr_wyjściowy
INOUT parametr_obustronny
```

🎯 Po co używać procedur:

1️⃣ ⚡ Automatyzacja - "Zrób całość za mnie!"

```
-- Zamiast 10 zapytań SQL:
CALL przelew_pieniedzy(konto_z, konto_na, kwota);

-- Ta procedura robi:
-- 1. Sprawdź saldo
-- 2. Odejmij z konta A
-- 3. Dodaj do konta B
-- 4. Zapisz w historii
-- 5. Wyślij powiadomienie
```

2️⃣ 🔒 Bezpieczeństwo - "Nie dotykaj tabel!"

```
-- ❌ Użytkownik NIE MA dostępu do tabel
-- ✅ Użytkownik MA dostęp do procedury

-- Pracownik może:
CALL podwyzsz_pensje('Jan', 5000);

-- Ale NIE może:
UPDATE pracownicy SET pensja = 1000000 WHERE imie = 'Jan';
```

3️⃣ 🚀 Wydajność - "Już skompilowane!"

```
-- Procedura jest skompilowana raz
-- Każde wywołanie = szybkie wykonanie
-- Brak parsowania SQL za każdym razem
```

4️⃣ 🎯 Centralny kod - "Jedna zmiana = wszędzie działa!"

```
-- Logika biznesowa w jednym miejscu
-- Wszystkie aplikacje korzystają z tej samej procedury
-- Zmiana reguł = zmiana tylko procedury
```

💡 Praktyczny przykład - system sklepu:

```
-- Procedura realizacji zamówienia
CREATE PROCEDURE realizuj_zamowienie(
    IN p_klient_id INT,
    IN p_produkt_id INT,
    IN p_ilosc INT,
    OUT p_status VARCHAR(50),
    OUT p_koszt DECIMAL(10,2)
```

```
)
BEGIN
    DECLARE stan_magazynowy INT;
    DECLARE cena_jednostkowa DECIMAL(10,2);

    -- 1. Sprawdź stan magazynu
    SELECT ilosc INTO stan_magazynowy
    FROM magazyn
    WHERE produkt_id = p_produkt_id;

    -- 2. Sprawdź czy wystarczy towaru
    IF stan_magazynowy < p_ilosc THEN
        SET p_status = 'BRAK_TOWARU';
        SET p_koszt = 0;
    ELSE
        -- 3. Pobierz cenę
        SELECT cena INTO cena_jednostkowa
        FROM produkty
        WHERE id = p_produkt_id;

        -- 4. Utwórz zamówienie
        INSERT INTO zamowienia (klient_id, produkt_id, ilosc, koszt, data)
        VALUES (p_klient_id, p_produkt_id, p_ilosc,
                cena_jednostkowa * p_ilosc, NOW());

        -- 5. Zmniejsz stan magazynu
        UPDATE magazyn
        SET ilosc = ilosc - p_ilosc
        WHERE produkt_id = p_produkt_id;

        -- 6. Zwróć wynik
        SET p_status = 'SUKCES';
        SET p_koszt = cena_jednostkowa * p_ilosc;
    END IF;
END;

-- Użycie:
CALL realizuj_zamowienie(123, 456, 2, @status, @koszt);
SELECT @status as wynik, @koszt as koszt_total;
```

🎭 Analogie dla zapamiętania:

🤖 Analogia - automat vendingowy:

Bez procedury: Musisz znać kody produktów, mieć dokładną kwotę, znać procedurę... **Z procedurą:** Naciśnij "Coca-Cola" → automat załatwia resztę!

🏠 Analogia - smart home:

Bez procedury: Włącz światło, ustaw temperaturę, włącz muzykę, zamknij żaluzje... **Z procedurą:** Powiedz "tryb filmowy" → wszystko się dzieje automatycznie!

🔥 Rodzaje procedur:

1️⃣ Procedury (PROCEDURE) - "Rób coś"

```
CALL dodaj_klienta('Jan', 'jan@example.com');
-- Nie zwraca wartości bezpośrednio
```

2️⃣ Funkcje (FUNCTION) - "Policz coś"

```
SELECT oblicz_vat(1000) as vat;
-- Zwraca konkretną wartość
```

🚨 Wady procedur:

⚠ Trudniejsze w debugowaniu

```
-- Błąd w procedurze = trudno znaleźć
-- Brak zaawansowanych narzędzi debug
```

⚠ Przenośność między bazami

```
-- MySQL: CREATE PROCEDURE
-- PostgreSQL: CREATE FUNCTION
-- Oracle: CREATE PROCEDURE
-- → Różna składnia!
```

🧠 Jak zapamiętać:

🎯 Mnemotechnika:

PROCEDURE = **P**rogramik **R**obi **O**peracje **C**o **E**konomią **D**anych **U**łatwia **R**ealnie **E**xtraordinary!

📚 Złote zasady:

1. **Procedura = mini-program w bazie**
2. **Używaj do złożonych operacji**
3. **Świetne dla bezpieczeństwa**
4. **Szybsze niż wielokrotne zapytania**

> 💡 **Zapamiętaj:** Procedura = "makro w bazie danych" - raz napisz, wielokrotnie używaj! 🎯

> 🏆 **Praktyczna rada:** Używaj procedur do skomplikowanej logiki biznesowej! ⭐

3️⃣7️⃣ Znaczniki czasu - "numerki w kolejce" dla transakcji! ⏰🎫

Znaczniki czasu (Timestamp Ordering) = "system numerków" 🎫 - każda transakcja dostaje numerek i musi go respektować!

Wyobraź sobie urząd pocztowy 🏢: wszyscy dostają numerki, każda przesyłka ma zapisane "kto ją ostatni dotykał" i kiedy!

🎯 Podstawowa idea:

🎫 Każda transakcja = numerek w kolejce

```
Transakcja A → timestamp = 100
Transakcja B → timestamp = 101
Transakcja C → timestamp = 102
```

Zasada: Kto wcześniej przyszedł, ten ma pierwszeństwo! 🏃‍♂️

📋 Każdy rekord pamięta historię

```
Rekord X:
- Read_TS(X) = 101   (ostatni czytał transakcja 101)
- Write_TS(X) = 100  (ostatni pisał transakcja 100)
```

🔍 Jak działają zasady:

📖 Chcesz CZYTAĆ rekord?

Sprawdź: Czy nikt "z przyszłości" już nie pisał?

```
Transakcja 100 chce czytać rekord X:
Write_TS(X) = 99  ✅ OK! (99 < 100)
Write_TS(X) = 102 ❌ BŁĄD! (102 > 100)
```

🤷‍♂️ **Analogia:** Chcesz przeczytać dokument, ale ktoś z przyszłości już go zmienił → nie możesz!

✏️ Chcesz PISAĆ do rekordu?

Sprawdź: Czy nikt "z przyszłości" już nie czytał ANI nie pisał?

```
Transakcja 100 chce pisać do rekordu X:
Read_TS(X) = 99, Write_TS(X) = 98  ✅ OK!
Read_TS(X) = 102, Write_TS(X) = 98  ❌ BŁĄD! (ktoś już czytał)
Write_TS(X) = 101                   ❌ BŁĄD! (ktoś już pisał)
```

💥 Praktyczny przykład:

🏦 System bankowy:

```
Stan początkowy:
Konto A: saldo=1000, Read_TS=0, Write_TS=0

⏰ T=100: Transakcja A chce sprawdzić saldo
- Read_TS(konto) = 0 < 100 ✅ OK!
- Czyta: saldo = 1000
- Aktualizuje: Read_TS(konto) = 100

⏰ T=101: Transakcja B chce wpłacić 500
- Read_TS(konto) = 100 < 101 ✅ OK!
- Write_TS(konto) = 0 < 101 ✅ OK!
- Pisze: saldo = 1500
- Aktualizuje: Write_TS(konto) = 101

⏰ T=99: Transakcja C (opóźniona) chce pobrać 200
- Write_TS(konto) = 101 > 99 ❌ BŁĄD!
- Transakcja C zostaje ODRZUCONA!
```

⚖️ Zalety vs Wady:

✅ Zalety:

- 🚫 **Brak deadlocków** - nie ma wzajemnego czekania
- 🎯 **Deterministyczny** - zawsze taki sam rezultat
- ⚡ **Brak blokad** - nie ma czekania na zwolnienie

❌ Wady:

- 🔄 **Dużo restartów** - stare transakcje są odrzucane
- 📈 **Starvation** - nowe transakcje "zagłuszają" stare
- 💾 **Overhead** - trzeba pamiętać timestampy wszędzie

🎭 Analogie dla zapamiętania:

🏢 Analogia - urząd:

- **Numerek** = timestamp transakcji 🎫
- **Dokument** = rekord w bazie 📄
- **Pieczątka na dokumencie** = Read_TS/Write_TS
- **Zasada:** Nie możesz zmienić dokumentu, jeśli ktoś z większym numerkiem już go dotykał!

🚂 Analogia - pociąg:

- **Bilet z czasem** = timestamp transakcji 🎫
- **Wagon** = rekord w bazie 🚃
- **Lista pasażerów** = kto był w wagonie (Read_TS/Write_TS)
- **Zasada:** Nie możesz wejść do wagonu, jeśli już był tam ktoś z "przyszłości"!

🔥 Odmiany timestamp ordering:

1️⃣ Basic Timestamp Ordering

```
Ścisłe przestrzeganie zasad
→ Dużo abortów, ale gwarancja serializowalności
```

2️⃣ Conservative Timestamp Ordering

```
Czeka aż wszystkie wcześniejsze transakcje skończą
→ Mniej abortów, ale możliwe czekanie
```

3️⃣ Multiversion Timestamp Ordering

```
Trzyma wiele wersji rekordów z różnymi timestampami
→ Jeszcze mniej abortów!
```

🧠 Jak zapamiętać:

🎯 Mnemotechnika:

TIMESTAMP = **T**ransakcja **I**ma **M**ały **E**gzemplarz **S**pecjalnego **T**icketu **A**utoryzującego **M**odyfikacje **P**odczas!

📚 Złote zasady:

1. **Każda transakcja ma timestamp**
2. **Każdy rekord pamięta kto go ostatni używał**
3. **"Przyszłość" blokuje "przeszłość"**
4. **Brak deadlocków, ale dużo restartów**

🎯 Kiedy używać timestamp ordering:

✅ Dobrze dla:

- Systemów z krótkimi transakcjami
- Gdy deadlocki są problematyczne
- Systemów z przewidywalnym obciążeniem

❌ Źle dla:

- Długich transakcji (będą odrzucane)
- Systemów z dużą współbieżnością
- Gdy restarto transakcji są kosztowne

📊 Porównanie z innymi metodami:

Metoda	Deadlocki	Czekanie	Restarty	Złożoność
Blokady	❌ Tak	⚠ Tak	✅ Rzadko	😊 Niska
Timestamps	✅ Nie	✅ Nie	❌ Często	⚠ Średnia
MVCC	✅ Nie	✅ Nie	😊 Rzadko	❌ Wysoka

> 💡 **Zapamiętaj:** Timestamp = "numerek w kolejce" - kto wcześniej, ten ma prawo! 🎯

> 🏆 **Praktyczna rada:** Timestamp ordering dobry dla systemów z krótkimi, szybkimi transakcjami! ⚡

🎯 SZYBKIE RADY NA EGZAMIN

🔥 TOP 10 rzeczy do zapamiętania:

1. 🧬 **ACID** = **A**dam **C**zeka **I**nteligentne **D**ziecko
2. 🗝️ **Klucz główny** = unikalny + NOT NULL, **klucz obcy** = może NULL
3. 🔒 **NULL** = "nie wiem", sprawdzaj przez `IS NULL/IS NOT NULL`
4. 📊 **Normalizacja** = 1NF (atomowość) → 2NF (cały klucz) → 3NF (tylko klucz)
5. 🔄 **JOIN** = INNER (przecięcie), LEFT (wszystko z lewej), FULL (suma)
6. 🏝️ **Izolacja** = wyższy poziom = bezpieczniej ale wolniej
7. ⚛️ **Transakcja** = wszystko albo nic!
8. 🔐 **SQL Injection** = prepared statements ratują życie
9. 👥 **GROUP BY + HAVING** = grupuj i filtruj grupy
10. 🪆 **Podzapytania** = zapytanie w zapytaniu

💡 Jak mówić na egzaminie:

- **Używaj przykładów** - "jak przelew bankowy..." 🏦
- **Tłumacz problemy** - "co się stanie jak..." ⚠
- **Podawaj alternatywy** - "można też..." 🔄
- **Mów o praktyce** - "w rzeczywistości..." 🌍

🚨 Częste pułapki na egzaminie:

- **NULL** ≠ 0 ≠ "" (pusty string) ❌
- **NATURAL JOIN** = niebezpieczny! 💀
- **Współbieżność** = szybkość vs bezpieczeństwo ⚖️
- **3NF** wystarcza w większości przypadków ✅
- **CASCADE** = może usunąć więcej niż myślisz! 💥

> 🎓 **Powodzenia na egzaminie!** Pamiętaj - jeśli nie wiesz dokładnie, tłumacz własnymi słowami i używaj analogii! 💪